

# Chapitre 14

## Listes chaînées

### 14.1 La notion de liste

Une *liste* est une structure de données qui permet de stocker une séquence d'objets d'un même type. En cela, les listes ressemblent aux *tableaux*. La séquence d'entiers 3, 7, 2, 4 peut être représentée à la fois sous forme de tableau ou de liste. La notation [3, 7, 2, 4] représentera la liste qui contient cette séquence.

Il y a cependant des différences fondamentales entre listes et tableaux :

- Dans un tableau on a accès immédiat à n'importe quel élément par son indice (accès dit *aléatoire*), tandis que dans une liste chaînée on a accès aux éléments un après l'autre, à partir du premier élément (accès dit *séquentiel*).
- Un tableau a une taille fixe, tandis qu'une liste peut augmenter en taille indéfiniment (on peut toujours rajouter un élément à une liste).

#### Définition (récursive) :

En considérant que la liste la plus simple est *la liste vide* (notée [ ]), qui ne contient aucun élément, on peut donner une définition récursive aux listes chaînées d'éléments de type T :

- la liste vide [ ] est une liste ;
- si  $e$  est un élément de type T et  $l$  est une liste d'éléments de type T, alors le couple  $(e, l)$  est aussi une liste, qui a comme premier élément  $e$  et dont le reste des éléments (à partir du second) forment la liste  $l$ .

Cette définition est *récursive*, car une liste est définie en fonction d'une autre liste. Une telle définition récursive est correcte, car une liste est définie en fonction d'une liste plus courte, qui contient un élément de moins. Cette définition permet de construire n'importe quelle liste en partant de la liste vide.

**Conclusion :** la liste chaînée est une structure de données récursive.

La validité de cette définition récursive peut être discutée d'un point de vue différent. Elle peut être exprimée sous la forme suivante : quelque soit la liste  $l$  considérée,

- soit  $l$  est la liste vide [ ],
- soit  $l$  peut être décomposée en un premier élément  $e$  et un reste  $r$  de la liste,  $l=(e, r)$ .

Cette définition donne *une décomposition récursive* des listes. La condition générale d'arrêt de récursivité est pour la liste vide. Cette décomposition est valide, car la liste est réduite à chaque pas à une liste plus courte d'une unité. Cela garantit que la décomposition mène toujours à une liste de longueur 0, la liste vide (condition d'arrêt).

## 14.2 Représentation des listes chaînées en Java

Il y a plusieurs façons de représenter les listes chaînées en Java. L'idée de base est d'utiliser *un enchaînement de cellules* :

- chaque cellule contient un élément de la liste ;
- chaque cellule contient une référence vers la cellule suivante, sauf la dernière cellule de la liste (qui contient une référence nulle) ;
- la liste donne accès à la première cellule, le reste de la liste est accessible en passant de cellule en cellule, suivant leur enchaînement.

La figure 14.1 illustre cette représentation pour la liste exemple ci-dessus [3, 7, 2, 4].

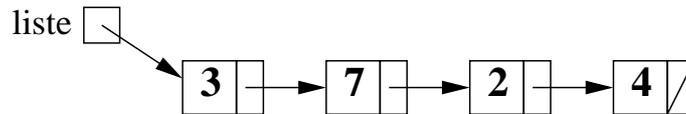


FIG. 14.1 – Représentation par enchaînement de cellules de la liste [3, 7, 2, 4]

En Java, les références ne sont pas définies explicitement, mais implicitement à travers les objets : un objet est représenté par une référence vers la zone de mémoire qui contient ses variables d'instance.

Comme une liste est définie par une référence vers une cellule (la première de la liste), *la méthode la plus simple est de représenter une liste par sa première cellule*. La classe suivante définit une liste d'entiers.

---

```

class Liste {
    int element;
    Liste suivant;

    Liste(int premier, Liste reste) {
        element = premier;
        suivant = reste;
    }
}
  
```

---

Le constructeur de la classe `Liste` crée une liste comme une première cellule qui contient le premier élément de la liste et une référence vers le reste de la liste.

**Remarque :** Cette solution a l'inconvénient de ne représenter que *des listes avec au moins une cellule*. La liste vide, qui selon la définition est également une liste, ne peut pas être un objet de la classe `Liste`. Il est néanmoins possible de représenter la liste vide *par une référence nulle*. Cette représentation imparfaite des listes a l'avantage de la simplicité et nous l'utiliserons dans le reste de ce cours.

## 14.3 Opérations sur les listes chaînées

Nous étudierons les opérations les plus importantes sur les listes chaînées, qui seront représentées comme des méthodes de la classe `Liste`.

**Remarque :** Puisque la classe `Liste` ne peut pas représenter la liste vide, des opérations de base comme le test de liste vide ou l'action de vider une liste ne peuvent pas être des méthodes de la

classe `Liste`. Ces opérations doivent être effectuées à l'extérieur de la classe `Liste`, par le programme qui utilise les listes.

Conformément à leur définition, les listes sont des structures *récurives*. Par conséquent, les opérations sur les listes s'expriment naturellement par des algorithmes récursifs. En même temps, les listes sont des structures linéaires, parfaitement adaptées à un parcours *itératif*. Nous donnerons aussi la variante itérative des opérations sur listes, à titre comparatif.

La représentation de la classe `Liste` ci-dessous montre sous forme de méthodes les opérations sur listes que nous discuterons.

---

```

class Liste{
    int element;
    Liste suivant;

    public Liste(int premier , Liste reste){
        element = premier;
        suivant = reste;
    }

    public int premier(){...}
    public Liste reste(){...}
    public void modifiePremier(int elem){...}
    public void modifieReste(Liste reste){...}
    public int longueur(){...}
    public boolean contient(int elem){...}
    public Liste insertionDebut(int elem){...}
    public void concatenation(Liste liste){...}
    public Liste suppressionPremier(int elem){...}
}

```

---

### 14.3.1 Opérations sans parcours de liste

Pour ces opérations il n'y a pas de variantes itératives et récursives, l'action est réalisée directement sur la tête de la liste.

#### Obtenir le premier élément et le reste de la liste

Ces opérations sont réalisées par les méthodes `premier` (qui retourne le premier élément de la liste), respectivement `reste` (qui retourne le reste de la liste).

En fait, ces méthodes ne sont pas nécessaires, au sens strict du terme, puisque dans la classe `Liste` on a accès direct aux variables d'instance *element* et *suivant*.

Nous verrons plus tard que l'accès direct aux variables d'instance est déconseillé dans la programmation orientée-objet. A la place, on préfère "cacher" les variables d'instance et donner accès à leur valeurs à travers des méthodes. Ceci fera l'objet d'un prochain cours.

Aussi, ces méthodes correspondent aux éléments de la décomposition récursive d'une liste et sont bien adaptées à l'écriture d'algorithmes récursifs sur les listes.

---

```

public int premier(){
    return element;
}

```

```
public Liste reste(){
    return suivant;
}
```

---

**Remarque :** Il ne faut pas oublier que `premier` et `reste` sont des méthodes de la classe `Liste`, donc une instruction comme `return element` signifie `return this.element`. Une liste est représentée par sa première cellule, donc `return this.element` retourne l'élément de cette première cellule.

### Modifier le premier élément et le reste de la liste

Ces opérations sont réalisées par les méthodes `modifiePremier` (qui modifie le premier élément de la liste), respectivement `modifieReste` (qui modifie le reste de la liste).

Ces méthodes permettent donc de *modifier* les composants `element` et `suivant` de la première cellule de la liste. Elles sont complémentaires aux méthodes `premier` et `reste`, qui permettent de *consulter* ces mêmes composants.

Leur présence dans la classe `Liste` correspond au même principe que celui évoqué pour `premier` et `reste` : remplacer l'accès direct aux variables d'instance par des appels de méthodes.

```
public void modifiePremier(int elem){
    element = elem;
}

public void modifieReste(Liste reste){
    suivant = reste;
}
```

---

### Insérer un élément en tête de liste

L'insertion en tête de liste est réalisée par la méthode `insertionDebut`. C'est une opération simple, car elle nécessite juste un accès à la tête de la liste initiale. On crée une nouvelle cellule, à laquelle on enchaîne l'ancienne liste.

La méthode `insertionDebut` retourne une nouvelle liste, car la liste initiale est modifiée par l'insertion (la première cellule de la nouvelle liste est différente).

```
public Liste insertionDebut(int elem){
    return new Liste(elem, this);
}
```

---

### 14.3.2 Opérations avec parcours de liste - variante itérative

Ces opérations nécessitent un parcours complet ou partiel de la liste. Dans la variante itérative, le parcours est réalisé à l'aide d'une référence qui part de la première cellule de la liste et suit l'enchaînement des cellules.

La figure 14.2 illustre le principe du parcours itératif d'une liste à l'aide d'une référence *ref*.

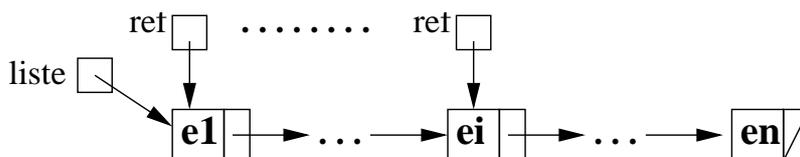


FIG. 14.2 – Parcours itératif d'une liste à l'aide d'une référence

### Calculer la longueur d'une liste

Cette opération, réalisée par la méthode `longueur`, nécessite un parcours complet de la liste pour compter le nombre de cellules trouvées.

---

```

public int longueur(){
    int compteur = 0;
    Liste ref=this;
    while(ref != null){
        compteur++;
        ref=ref.reste(); //ou ref=ref.suivant
    }
    return compteur;
}

```

---

**Remarque :** La référence `ref` est positionnée au début sur la première cellule de la liste et avance jusqu'à ce qu'elle devienne nulle à la fin de la liste. Chaque fois qu'elle pointe vers une nouvelle cellule, le compteur est incrémenté. Remarquez que l'avancement dans la liste se fait en utilisant la méthode `reste` de la cellule courante, ou sinon l'accès direct à la variable d'instance `suivant`.

### Vérifier l'appartenance d'un élément à une liste

Cette opération, réalisée par la méthode `contient`, nécessite un parcours partiel de la liste pour chercher l'élément en question. Si l'élément est retrouvé, le parcours s'arrête à ce moment-là, sinon il continue jusqu'à la fin de la liste.

---

```

public boolean contient(int elem){
    Liste ref=this;
    while(ref != null){
        if(ref.premier() == elem) //ou ref.element==elem
            return true; //lé'élément aé ét éretrouv
        else ref=ref.reste(); //ou ref=ref.suivant
    }
    return false; //lé'élément n'a pasé ét éretrouv
}

```

---

**Remarque :** L'arrêt du parcours en cas de succès se fait en retournant directement `true` pendant l'exécution de la boucle, ce qui termine à la fois la boucle et la fonction. Remarquez que le test de l'élément courant pointé par la référence utilise la méthode `premier` de la cellule courante, ou sinon l'accès direct à la variable d'instance `element`.

### Concaténation de deux listes

Cette opération, réalisée par la méthode `concatenation`, rajoute à la fin de la liste courante toutes les cellules de la liste passée en paramètre. Elle nécessite un parcours complet de la liste courante, afin de trouver sa dernière cellule.

La liste obtenue par concaténation a toujours la même première cellule que la liste initiale. On peut dire donc que la liste initiale a été modifiée, en lui rajoutant par concaténation une autre liste.

---

```

public void concatenation(Liste liste){
    Liste ref=this;
    while(ref.reste() != null){           //ou ref.suivant!=null
        ref=ref.reste();                   //ou ref=ref.suivant
    }
    //ref pointe maintenant sur la dernière cellule de la liste
    ref.modifieReste(liste);              //ou ref.suivant=liste
}

```

---

**Remarque :** La condition de la boucle `while` change ici, car on veut s’arrêter sur la dernière cellule et non pas la dépasser comme dans les méthodes précédentes. Remarquez que l’enchaînement des deux listes se fait en modifiant la variable d’instance `suivant` de la dernière cellule à l’aide de la méthode `modifieReste`, ou sinon par modification directe de celle-ci.

### Suppression de la première occurrence d’un élément

Cette opération, réalisée par la méthode `suppressionPremier`, élimine de la liste la première apparition de l’élément donné en paramètre (s’il existe). La méthode retourne une liste, car la liste obtenue par suppression peut avoir une autre première cellule que la liste initiale. Ceci arrive quand la cellule éliminée est exactement la première de la liste.

La suppression d’une cellule de la liste se fait de la manière suivante (voir la figure 14.3) :

- si la cellule est la première de la liste, la nouvelle liste sera celle qui commence avec la seconde cellule.
- sinon la cellule a un prédécesseur ; pour éliminer la cellule il faut la “court-circuiter”, en modifiant la variable `suivant` du prédécesseur pour qu’elle pointe vers la même chose que la variable `suivant` de la cellule.

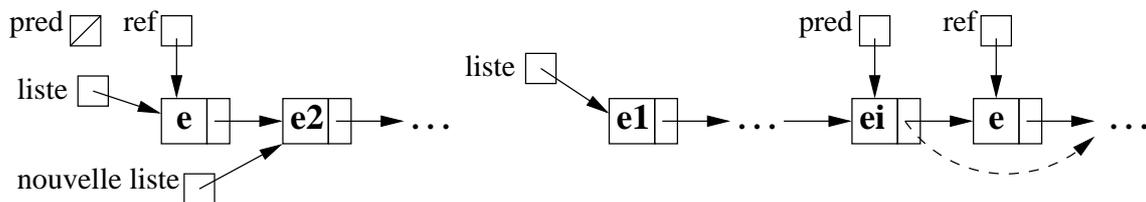


FIG. 14.3 – Suppression d’une cellule de la liste

Le parcours de la liste à la recherche de l’élément donné a une particularité : si la référence s’arrête sur la cellule qui contient l’élément, la suppression n’est plus possible, car on n’a plus accès au prédécesseur (dans les listes chaînées on ne peut plus revenir en arrière).

La méthode utilisée est alors de parcourir la liste avec *deux références* :

- une (*ref*) qui cherche la cellule à éliminer ;
- une autre (*pred*) qui se trouve toujours sur le prédécesseur de *ref*.

---

```

public Liste suppressionPremier(int elem){
  Liste ref=this; //référence qui cherche elem
  Liste pred=null; //éprcdesseur, éinitialisé à null
  while(ref != null && ref.premier() != elem){ //recherche elem
    pred = ref; //avance pred
    ref = ref.reste(); //avance ref
  }
  if(ref != null){ //elem trouvé dans la cellule ref
    if(pred == null) //èpremière cellule de la liste
      return ref.reste(); //retourne le reste de la liste
    else { //milieu de la liste
      pred.modifieReste(ref.reste()); //modifie le suivant du éprcdesseur
      return this; //èpremière cellule non émodifiée
    }
  }
  else return this; //éélément non étrouvé
}

```

---

### 14.3.3 Opérations avec parcours de liste - variante récursive

Ces opérations sont basées sur une décomposition récursive de la liste en *un premier élément et le reste de la liste* (voir la figure 14.4). Le cas le plus simple (condition d'arrêt) est la liste avec une seule cellule.

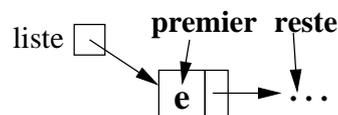


FIG. 14.4 – Décomposition récursive d'une liste

#### Calcul récursif de la longueur d'une liste

La formule récursive est :

$$\begin{aligned}
 \text{liste.longueur}() &= 1 && \text{si } \text{reste} == \text{null} \\
 &1 + \text{reste.longueur}() && \text{si } \text{reste} \neq \text{null}
 \end{aligned}$$

La fonction récursive longueur s'écrit alors très facilement, comme suit :

---

```

public int longueur(){
  Liste resteListe = reste(); //ou resteListe=suivant;
  if(resteListe == null) //condition d'arrêt
    return 1;
  else return 1 + resteListe.longueur();
}

```

---

### Vérification récursive de l'appartenance à une liste

La formule récursive est :

```
liste.contient(e) = true           si premier==e
                   false          si premier!=e et reste==null
                   reste.contient(e) si premier!=e et reste!=null
```

Remarquez que dans ce cas il y a deux conditions d'arrêt : quand on trouve l'élément recherché ou quand la liste n'a plus de suite (reste est vide). La fonction récursive `contient` s'écrit alors comme suit :

---

```
public boolean contient(int elem){
    if(elem == premier()) return true; //ou elem==element
    Liste resteListe = reste();       //ou resteListe=suivant;
    if(resteListe == null)             //condition d'arrêt
        return false;
    else return resteListe.contient(elem);
}
```

---

### Concaténation récursive de deux listes

L'action récursive est :

```
liste.concatenation(l): reste=l           si reste==null
                       reste.concatenation(l) si reste!=null
```

Si la liste a une seule cellule (pas de cellule suivante, reste est vide), alors il faut modifier la référence vers la cellule suivante pour enchaîner la liste paramètre. Sinon, il suffit de concaténer le reste avec la liste paramètre.

---

```
public void concatenation(Liste liste){
    Liste resteListe = reste(); //ou resteListe=suivant;
    if(resteListe == null)      //condition d'arrêt
        modifieReste(liste);    //ou suivant=liste
    else resteListe.concatenation(liste);
}
```

---

### Suppression récursive de la première occurrence d'un élément

La formule récursive est :

```
liste.suppressionPremier(e) =
    reste           si premier==e
    Liste(premier, null) si premier!=e et reste==null
    Liste(premier, reste.suppressionPremier(e)) si premier!=e et
                                                reste != null
```

Si le premier élément de la liste est celui recherché, le résultat sera le reste de la liste. Sinon, il faut supprimer récursivement l'élément recherché du reste. Mais le résultat de cette suppression est une liste qui provient de *reste*, donc elle n'a plus le premier élément de la liste initiale. Il faut donc le rajouter en tête de liste.

---

```

public Liste suppressionPremier(int elem){
    Liste resteListe = reste (); //ou resteListe=suivant;
    if(elem == premier ()) //ou elem==element
        return resteListe;
    else if(resteListe == null) return new Liste(premier (), null);
    else return new Liste(premier (), resteListe.suppressionPremier(elem));
}

```

---

**Remarque :** L'inconvénient de cette méthode est qu'on crée des copies de toutes les cellules qui ne contiennent pas l'élément recherché. Pour éviter cela, la définition doit mélanger calcul et effets de bord, comme suit :

```

liste.suppressionPremier(e) =
    reste    si premier==e
    liste    si premier!=e et reste==null
    liste après l'action reste=reste.suppressionPremier(e)
             si premier!=e et reste!=null

```

---

```

public Liste suppressionPremier(int elem){
    Liste resteListe = reste (); //ou resteListe=suivant;
    if(elem == premier ()) //ou elem==element
        return resteListe;
    else if(resteListe == null) return this;
    else{
        modifieReste(resteListe.suppressionPremier(elem));
        return this;
    }
}

```

---

## 14.4 Listes triées

Nous nous intéressons maintenant aux listes chaînées dans lesquelles les éléments respectent un ordre croissant. Dans ce cas, les méthodes de recherche, d'insertion et de suppression sont différentes. L'ordre des éléments permet d'arrêter plus tôt la recherche d'un élément qui n'existe pas dans la liste. Aussi, l'insertion ne se fait plus en début de liste, mais à la place qui préserve l'ordre des éléments.

Nous utiliserons une classe `ListeTrie` qui est très similaire à la classe `Liste`, mais dans laquelle on s'intéresse uniquement aux méthodes de recherche (`contientTrie`), d'insertion (`insertionTrie`) et de suppression (`suppressionTrie`). Les autres méthodes sont identiques à celles de la classe `Liste`.

---

```

class ListeTrie {
    int element;
    ListeTrie suivant;

    public ListeTrie(int premier, ListeTrie reste){
        element = premier;
        suivant = reste;
    }
}

```

```

public int premier(){...}
public ListeTriee reste(){...}
public void modifiePremier(int elem){...}
public void modifieReste(ListeTriee reste){...}

public boolean contientTrie(int elem){...}
public ListeTriee insertionTrie(int elem){...}
public ListeTriee suppressionTrie(int elem){...}
}

```

---

#### 14.4.1 Recherche dans une liste triée

La différence avec la méthode `contient` de `Liste` est que dans le parcours de la liste on peut s'arrêter dès que la cellule courante contient un élément plus grand que celui recherché. Comme tous les éléments suivants vont en ordre croissant, ils seront également plus grands que l'élément recherché.

**Variante itérative :**

```

public boolean contientTrie(int elem){
    ListeTriee ref=this;
    while(ref != null && ref.premier() <= elem){
        if(ref.premier() == elem)
            return true;
        else ref=ref.reste();
    }
    return false;
}

```

---

**Variante récursive :**

```

public boolean contientTrie(int elem){
    if(elem == premier()) return true;
    if(elem < premier()) return false;
    ListeTriee resteListe = reste();
    if(resteListe == null)
        return false;
    else return resteListe.contientTrie(elem);
}

```

---

#### 14.4.2 Insertion dans une liste triée

L'insertion d'un élément doit se faire à *la bonne place* dans la liste. La bonne place est *juste avant la première cellule qui contient un élément plus grand que celui à insérer*.

**Variante itérative :**

La figure 14.5 montre les deux cas d'insertion :

- en début de liste.

Elle est similaire alors à l'opération `insertionDebut` de la classe `Liste`

- en milieu de liste.

La recherche de la position d'insertion se fait alors avec une méthode similaire à celle utilisée pour la méthode `suppressionPremier` de la classe `Liste`, avec deux références. Le prédécesseur doit être modifié pour pointer vers la nouvelle cellule, tandis que celle-ci doit pointer vers la cellule courante (*ref*).

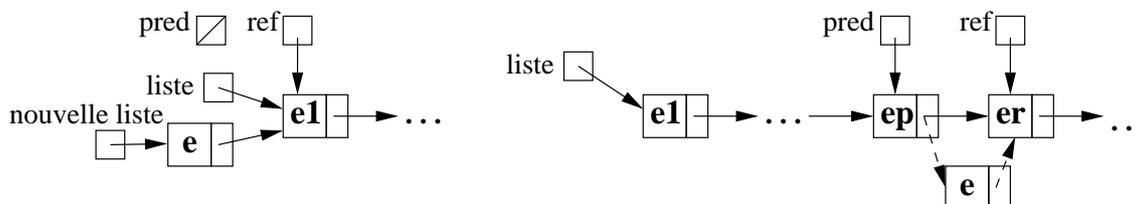


FIG. 14.5 – Insertion dans une liste triée

Dans la figure 14.5, à droite, la référence *ref* peut ne pas pointer vers une cellule. Ceci arrive quand l'élément à insérer est plus grand que tous les éléments de la liste. Dans ce cas, l'insertion se fait à la fin de la liste, donc au moment de l'arrêt de la recherche, *ref* est nulle et *pred* est sur la dernière cellule de la liste. Ce cas est identique à celui présenté dans la figure 14.5, à la variable suivant de la nouvelle cellule on donne tout simplement la valeur de *ref*.

---

```

public ListeTrie insertionTrie(int elem){
    ListeTrie ref=this; //réfrence qui cherche elem
    ListeTrie pred=null; //ééprdcesseur, éinitialisé à null
    while(ref != null && ref.premier() < elem){ //recherche position
        pred = ref; //avance pred
        ref = ref.reste(); //avance ref
    }
    if(pred == null) //insertion au édbut
        return new ListeTrie(elem, this);
    else { //insertion au milieu de la liste
        ListeTrie nouveau = new ListeTrie(elem, ref);
        pred.modifieReste(nouveau); //modifie le suivant du ééprdcesseur
        return this; //èpremière cellule non émodifie
    }
}

```

---

#### Variante récursive :

Nous utilisons le même schéma de décomposition récursive de la liste, en un premier élément (*premier*) et un reste de la liste (*reste*). L'insertion récursive suit le schéma suivant :

```

liste.insertionTrie(e) =
    ListeTrie(e, liste)                si e<=premier
    liste après reste=ListeTrie(e, null) si e>premier et reste==null
    liste après reste=reste.insertionTrie(e) si e>premier et reste!=null

```

Si l'élément à insérer est plus petit ou égal à *premier*, l'insertion se fera en tête de liste. Sinon, l'insertion se fera récursivement dans le reste. Si *reste* est vide, alors *reste* sera remplacé par la nouvelle cellule à insérer. Si *reste* n'est pas vide, l'insertion se fait récursivement dans *reste* et le résultat remplace l'ancien *reste*.

---

```

public ListeTrie insertionTrie(int elem){
    if(elem <= premier()){
        return new ListeTrie(elem, this);
    }
    else{
        ListeTrie resteListe = reste();
        if(resteListe == null)
            modifieReste(new ListeTrie(elem, null));
        else
            modifieReste(resteListe.insertionTrie(elem));
        return this;
    }
}

```

---

### 14.4.3 Suppression dans une liste triée

La suppression de la première occurrence d'un élément dans une liste triée est assez similaire à la suppression dans une liste non triée. La seule chose qui change est la recherche de la cellule à supprimer, qui bénéficie du tri des valeurs. Aussi, si plusieurs occurrences de l'élément existent, elles sont forcément l'une à la suite de l'autre à cause du tri. Il serait donc plus simple d'éliminer *toutes* les occurrences de l'élément que dans le cas des listes non triées. Nous nous limiterons cependant ici à la suppression de la première occurrence seulement.

#### Variante itérative :

Par rapport à la méthode `suppressionPremier` de la classe `Liste`, ici le parcours de la liste à la recherche de l'élément ne se fait que tant que `ref` pointe une valeur plus petite que celui-ci. Une fois le parcours arrêté, la seule différence est qu'il faut vérifier que `ref` pointe vraiment la valeur recherchée.

---

```

public Liste suppressionTrie(int elem){
    Liste ref=this; //référence qui cherche elem
    Liste pred=null; //éprcdcesseur, éinitialisé à null
    while(ref != null && ref.premier() < elem){ //recherche elem
        pred = ref; //avance pred
        ref = ref.reste(); //avance ref
    }
    if(ref != null && ref.premier() == elem){ //elem trouvé dans la cellule ref
        if(pred == null) //èpremiere cellule de la liste
            return ref.reste(); //retourne le reste de la liste
        else{ //milieu de la liste
            pred.modifieReste(ref.reste()); //modifie le suivant du éprcdcesseur
            return this; //èpremiere cellule non émodifie
        }
    }
    else return this; //éélément non étrouv
}

```

---

#### Variante récursive :

Par rapport à la variante récursive de la méthode `suppressionPremier` de la classe `Liste`, une nouvelle condition d'arrêt est rajoutée :

quand l'élément à éliminer est plus petit que *premier*, il n'existe sûrement pas dans la liste et celle-ci est retournée non modifiée.

---

```
public Liste suppressionTrie(int elem){
    Liste resteListe = reste();
    if(elem == premier())
        return resteListe;
    else if(elem < premier()) return this; //nouvelle condition d'arrêt
    else if(resteListe == null) return this;
    else{
        modifieReste(resteListe.suppressionTrie(elem));
        return this;
    }
}
```

---

## 14.5 Un exemple de programme qui utilise les listes

Considérons un exemple simple de programme qui manipule des listes chaînées en utilisant la classe `Liste`.

Le programme lit une suite de nombres entiers terminée par la valeur 0 et construit une liste avec ces entiers (sauf le 0 final). La liste doit garder les éléments dans l'ordre dans lequel ils sont introduits. Egalement, une valeur ne doit être stockée qu'une seule fois dans la liste.

Le programme lit ensuite une autre suite de valeurs, également terminée par un 0, avec lesquelles il construit une autre liste, sans se soucier de l'ordre des éléments. Les éléments de cette seconde liste devront être éliminés de la liste initiale.

A la fin, le programme affiche ce qui reste de la première liste.

**Remarque :** Pour garder les éléments dans l'ordre d'introduction, l'insertion d'un nouvel élément doit se faire *en fin de liste*. La méthode `insertionDebut` ne convient donc pas. La solution est d'utiliser la méthode `concatenation`.

**Remarque :** Pour qu'un élément soit stocké une seule fois dans la liste, il faut d'abord vérifier s'il n'y est pas déjà, à l'aide de la méthode `contient`.

Voici le programme qui réalise ces actions :

---

```
public class ExempleListes{
    public static void main(String[] args){
        //1. éCration èpremiere liste
        Terminal.ecrireStringln("Entrez_les_valeurs_étermines_par_un_0_");
        Liste liste = null; //liste à construire
        do{
            int val = Terminal.lireInt();
            if(val==0) break;
            if(liste==null)
                liste = new Liste(val, null);
            else if(!liste.contient(val))
                liste.concatenation(new Liste(val, null));
        } while(true);

        //2. éCration seconde liste
        Terminal.ecrireStringln("Valeurs_à_éliminer_étermines_par_un_0_");
```

```
Liste liste2 = null;
do{
    int val = Terminal.lireInt();
    if(val==0) break;
    if(liste2==null)
        liste2 = new Liste(val, null);
    else liste2 = liste2.insertionDebut(val);
} while(true);

//3. Elimination de la première liste
for(Liste ref = liste2; ref != null; ref = ref.reste()){
    if(liste==null) break; //plus rien à éliminer
    liste = liste.suppressionPremier(ref.premier());
}

//4. Affichage liste restante
Terminal.ecrireStringln("Valeurs restantes :");
for(Liste ref = liste; ref != null; ref = ref.reste())
    Terminal.ecrireString(" " + ref.premier());
Terminal.sautDeLigne();
}
}
```

---