4. Récursivité et structures de données avancées

Nous allons nous initier à la **programmation récursive** et

Construire des algorithmes sur **des structures de données plus avancées** telles que les listes chaînées, piles, files d’attente, tables et arbres.

4.1. Listes linéaires chaînées

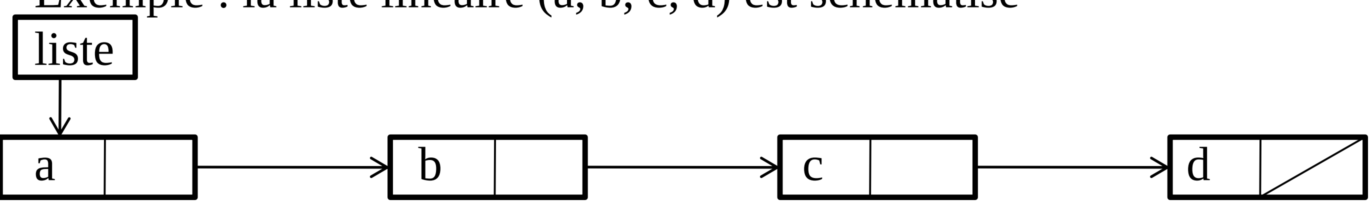
4.1.1. Définition

Une cellule est un couple composé d’une information et d’une adresse.

Une **liste linéaire chaînée** (liste chaînée) est un ensemble de **cellules chaînées entre elles**. C’est l’adresse de la première de ces cellules qui détermine la liste. Cette adresse se trouve dans une variable que nous appellerons très souvent **liste**.

Par convention, on notera **liste+** la suite des éléments contenus dans la liste dont l’adresse de la première cellule se trouve dans la variable **liste**.

Exemple : la liste linéaire (a, b, c, d) est schématisé



liste+ = (a, b, c, d). On notera que :

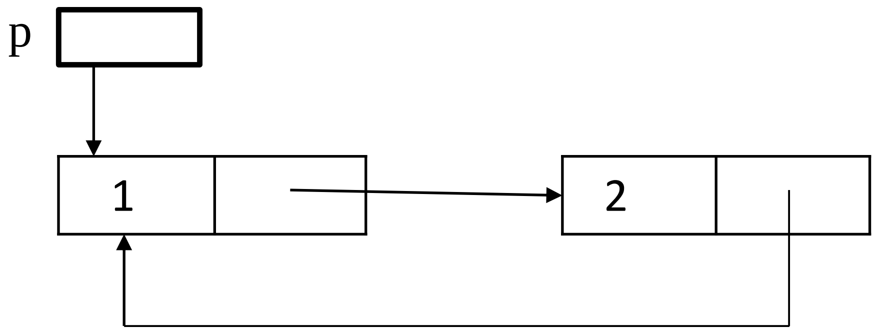
- si **liste = nil**, **liste+** est vide,
- il y a autant de cellules que d’éléments,
- le champ **suivant** d’une cellule contient l’ adresse de la cellule suivante,
- le champ **suivant** de la dernière cellule contient la valeur **nil**.

- il suffit de connaître l’adresse de la première cellule rangée dans la variable **liste** pour avoir accès à tous les éléments en utilisant : **liste->info**, l**iste->suivant**, **liste->suivant->info, …**
liste->info et **liste->suivant->info** ne sont définie que si liste est différente de **nil**.
<u>Notation</u> : **a<liste+**, indique que **a** est inférieur à tous les éléments de **liste+**.
<u>Sous-liste</u> : une sous-liste **p+** de **liste+** est une liste chainée telle que les éléments de **p+** soient une suite d’éléments **consécutifs** de **liste+**.
Exemple : liste+ = (a, b, c, d, e).
p+ = (b, c) est une sous-liste de **liste+**.
p+ = (b, d) n’est une sous-liste de **liste+**.
On notera que : **liste+ = p- || p+**,
On remarque que si **liste+** = **(a, b, c, d)**, **liste-** est **vide** et que si **liste=nil** on a **liste+** est **vide**.

4.1.2. Algorithme de création d’une liste chaînée

Créer une liste chaînée qui est la représentation de (a, b).
<u>Déclaration</u>
structure cellule
    t info**;**
    structure cellule***** suivant**;**
fin;
Où **t** est un type simple, vecteur, structure,…Puis on peut déclarer une variable liste :
structure cellule***** liste**; structure** cellule*** p**;
ou bien
type structure cellule***** pointeur;
pointeur liste**,** p**;**

Exercice 1, Ecrire une procédure qui crée la chose.



Afficher 1, puis 2, puis de nouveau 1.

Exercice 2. soit la déclaration : entier n, x ;

Calculer x^n | $x^0 = 1$;
$x^n = x * x^{n-1}$;

En écrivant la fonction récursive appelée **puissance**.

Exercice 3.

Calculer x^n | $x^0 = 1$
$x^{2n} = (x^n)^2$
$x^{2n+1} = x * x^{2n}$

En écrivant la fonction récursive appelée **puissance**.

Créer la liste (a, b) à partir de son dernier élément.

Etape 1 : création d’une liste vide d’adresse **liste**.

liste := nil;

Etape 2 : création de la liste d’adresse **liste** et contenant l’information **b**.

- création d’une cellule d’adresse **p** contenant l’information **b**

nouveau(p);

p->info:=‘b’;

- chaînage avec la liste précédente

p->suivant:=liste;

- création de la liste d’adresse **liste** représentation de (b)

liste:=p;

Etape 3 : création de la liste d’adresse **liste** contenant l’information (a, b).

- création d’une cellule d’adresse **p** contenant l’information **a**

nouveau(p**);**

p->info:=‘a’;

- chaînage avec la liste précédente

p->suivant:=liste;

- création de la liste d’adresse **liste** représentation de (a, b)

liste:=p;

Algorithme de création d'une liste chaînée à partir d'un fichier

```
pointeur fonction creerliste(d fichier de t f)
debfonc pointeur l, p; t val;
  l:=nil; relire(f);
  tantque non fdf(f) faire
    nouveau(p);
    lire(f, val);
    p->info:=val; p->suivant:=l; l:=p;
  finfaire;
  retourner l;
finfonc;
```

Passage d'un vecteur à une liste chaînée

```
pointeur fonction vecliste( d t v[], d entier n)
debfonc entier i; pointeur l, p;
  l:=nil; i:=n;
  tantque i ≥ 1 faire
    nouveau(p);
    p->info:=v[i]; p->suivant:=l;
    l:=p; i:=i-1;
  finfaire;
  retourner l;
finfonc;
```

Pour obtenir les éléments du fichier dans le même ordre, on doit créer la liste à partir de son premier élément. Pour ce faire, on doit d'abord prévoir le cas du premier élément qui permet la création de la première cellule de la liste et dont l'adresse doit être préservée car elle permettra l'accès à toutes les cellules créées.

Créer une liste à l'endroit à partir d'un fichier.

```
pointeur fonction creerliste(d fichier de t f)
debfonc pointeur l, p, der; t val;
   relire(f);
   si fdf(f) alors l:=nil;
   sinon
      nouveau(l);
      lire(f, val);  l->info:=val; der:=l;
      tantque non fdf(f) faire
          nouveau(p);
          lire(f, val); p->info:=val;
          der->suivant:=p; der:=p;
      finfaire;
      der->suivant:=nil;
      retourner l;
    finsi;
finfonc;
```

4.1.3. Parcours d’une liste

a) schéma récursif

Pour écrire des algorithmes récursifs, on utilise la définition suivante d’une liste chaînée :

• soit une liste est vide (liste = nil),

• soit une liste est composée d’une cellule (la 1ière) chaînée à une sous-liste (obtenue après suppression de la 1ière cellule).

<u>Exemple</u> : (a,b,c,d) est composée d’une cellule contenant **a** et d’une sous-liste contenant (**b, c, d**). Cette sous-liste a pour adresse **liste->suivant**+.

<u>Premier parcours</u>

- on traite la 1ière cellule,
- on effectue le parcours de la sous-liste **liste->suivant+**.

Algorithme de parcours d’une liste **à l’endroit.**

procédure parcours1**(**d **pointeur l)**
debproc
   si l ≠ **nil alors**
      traiter**(**l->info**);**
      parcours1**(**l->suivant**);**
   finsi;
finproc;

<u>Deuxième parcours</u>

- on effectue le parcours de la sous-liste **liste->suivant +**.
- on traite la 1ière cellule.

Algorithme de parcours de la liste **à l’envers.**

```
procédure parcours2(d pointeur l)
debproc si l ≠ nil alors
                parcours2(l->suivant);
                traiter(l->info);
             finsi;
finproc;
```
Où **traiter** est une procédure quelconque.

b) <u>Schéma itératif</u>. Le 2ième parcours n'est pas simple à obtenir en itératif. Il est nécessaire de disposer d'une **pile**. Algorithme du schéma itératif du 1ier parcours.

```
procédure parcours1(d pointeur l)
debproc tantque l≠nil faire
                traiter(l->info);
                l:=l->suivant;
              finfaire;
finproc;
```
Schéma itératif de parcours1, protection de l'adresse de la 1ière cellule, cas ou **l** est un résultat,

```
procédure parcours1(dr pointeur l)
debproc  pointeur p;
    p:=l; //protection de l'adresse
    tantque p≠nil faire
      traiter(p->info);
      p:=p->suivant;
   finfaire;
finproc;
```

Exercice 4. Fonction qui crée une liste chainée de nombres saisis au clavier, à partir de son dernier élément, puis à partir de son premier élément.

4.1.4. Algorithmes d'écriture des éléments d'une liste

Ecrire sous forme récursive 2 algorithmes d'écriture des éléments d'une liste.

a) première version

On utilise le 1^{ier} parcours **parcours1**. On écrit les éléments de la liste à partir du 1^{ier} élément

```
procédure ecritliste1(d pointeur l)
deproc  si l≠nil alors
              écrire(l->info);
              ecritliste1(l->suivant);
            finsi;
finproc;
```

b) deuxième version

On utilise le $2^{ième}$ parcours **parcours2** qui permet d'obtenir l'écriture des éléments à partir du dernier élément.

```
procédure ecritliste2(d pointeur l)
debproc  si l≠nil alors
              ecritliste2(l->suivant);
              écrire(l->info);
            finsi;
finproc;
```

Exercice 5. Que fait la fonction suivante ?

```
procédure ecritliste(d pointeur liste)
debproc
  si liste ≠ nil alors écrire(liste->info);
     ecritliste(liste->suivant); écrire(liste->info);
  finsi;
finproc;
```

Exercice 6. Que fait la fonction suivante ?

```
procédure ecritliste(d pointeur liste)
debproc
  si liste≠nil alors ecritliste(d liste->suivant);
     écrire(liste->info); ecritliste(liste->suivant);
  finsi;
finproc;
```

Dans les deux exercices, **liste** prend <a, b, c>.

Exercice 7. Ecrire la fonction récursive espacer dont l'en-tête est :

vide espacer(d entier n)

//spécification(n≥0)➔(écrit n en séparant les chiffres par un espace)

Exemple : espacer(123) donne 1 2 3.

4.1.5. Exemples d'algorithmes sur les listes chaînées

4.1.5.1. Algorithme de calcul de la longueur d'une liste

a) <u>schéma itératif</u>

entier fonction long1**(**d **pointeur** l**)**
debfonc
  entier nomb:**=**0**;**
  tantque l**≠nil faire**
    nomb **:=** nomb+1;
    l:=l->suivant**;**
  finfaire;
  retourner nomb;
finfonc;

b) <u>schéma récursif</u>

Raisonnement :

- liste+ est vide
  - sa longueur est alors égale à zéro.
  - **retourner 0** ; terminé
- liste+ n'est pas vide
  - elle est composée d'une cellule (1ier élément) chaînée à la sous-liste **liste->suivant+**.
    La longueur de la liste est donc égale à **1** plus la longueur de la liste **liste-> suivant+**.
  - **retourner(1 + long2(liste->suivant**));

```
entier fonction long2(d pointeur l)
debfonc   si l=nil alors retourner 0;
              sinon retourner 1+long2(l->suivant);
              finsi;
finfonc;
```

4.1.5.2. Algorithme de calcul du nombre d'occurrences d'un élément dans une liste

a) schéma itératif

```
entier fonction nbocc1(d pointeur l, d entier val)
debfonc
   entier nomb:=0;
   tantque l≠nil faire
      si l->info = val alors nomb:=nomb+1;
         l:=l->suivant;
      finsi;
   finfaire;
   retourner nomb;
finfonc;
```

b) Schéma récursif

Raisonnement :

- liste+ est vide.

  Le nombre d’occurrences est égale à zéro : **retourner 0** ; *****

- liste+ n’est pas vide.

  Elle est donc composée d’un élément et d’une sous-liste **liste->suivant+**.

  °° liste->info = val. Le nombre d’occurrences est égal à 1 plus le nombre d’occurrences de **val** dans la sous-liste **liste->suivant+** :

  retourner (1+nbocc2(liste->suivant, val));

  °° liste->info ≠ val. Le 1^{ier} élément ne contient pas la valeur **val**, le nombre d’occurrences de **val** dans la liste est donc égal au nombre d’occurrences de **val** dans la sous-liste **liste->suivant+** :

  retourner (nbocc2(liste->suivant, val));

```
entier fonction nbocc2(d pointeur l, d entier val)
defonc  si l=nil alors retourner 0;
          sinon
            si l->info=val alors retourner(1+ nbocc2(l->suivant, val));
            sinon  retourner(nbocc2(l->suivant, val));
            finsi;
          finsi;
finfonc;
```

4.1.5.3. Algorithmes d'accès dans une liste

On distingue 2 sortes d'accès : accès par position ou accès associatif.

a) **Algorithme d'accès par position**

- <u>schéma itératif</u>

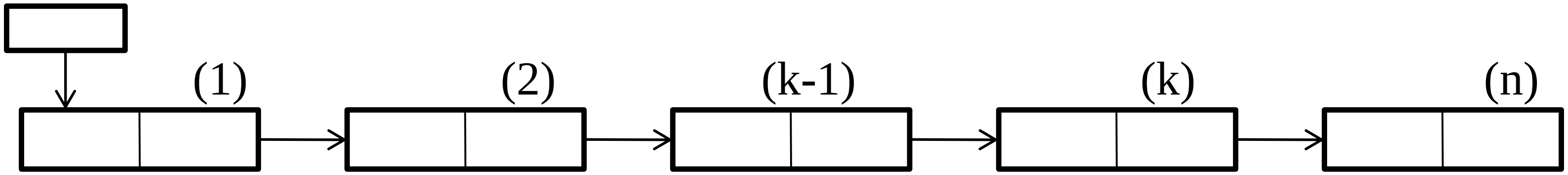
On utilise le même algorithme que celui donné sur les fichiers séquentiels.

```
procédure accesk(d pointeur l, d entier k, r booléen trouve, r pointeur
pointk)
 debproc
   entier i:=1; pointeur pointk;
   tantque (i<k) et (l≠nil) faire
      i:=i+1;
      l:=l->suivant;
   finfaire;
   trouve:=((i=k) et (l≠nil));
   pointk:=l;
 finproc;
```

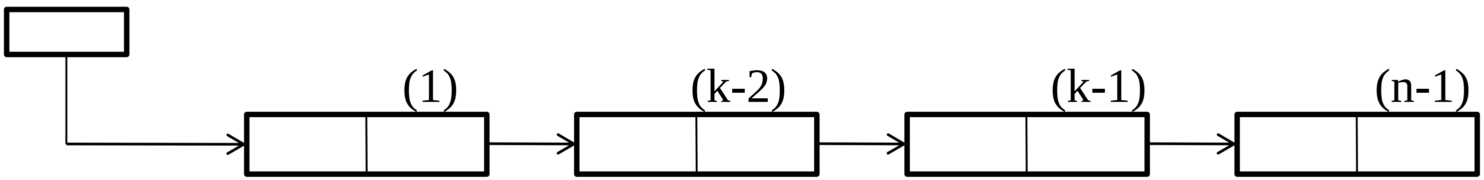
- <u>schéma récursif</u>

Le raisonnement est fondé sur la constatation suivante : chercher le **k^{ième}** (k>1) élément dans **liste** revient à chercher le **k-1^{ième}** élément dans **liste->suivant+.**

liste



liste->suivant



Raisonnement

• liste+ est vide. Algorithme est terminé, le k^{ième} élément n'existe pas et **pointk** prend la valeur **nil : retourner liste**; *

• liste+ n'est pas vide.

°° k = 1. Algorithme est terminé et **pointk** prend la valeur de l'adresse contenue dans la variable **liste** : **retourner liste**;

°° k ≠ 1. On rappelle la fonction pour chercher le k-1^{ième} élément dans la sous-liste **liste->suivant+** : **retourner (pointk(liste->suivant, k-1))**;

On peut mettre en facteur **retourner liste;**

Accès par position, schéma récursif.

pointeur fonction pointk**(**d **pointeur** l**, d entier** k**)**
defonc
  si (l**=nil) ou (**k**=1) alors retourner** l**;**
  sinon retourner (pointk**(**l->suivant, k-1**));**
  finsi;
finfonc;

b) Accès associatif dans une liste

- <u>schéma itératif</u>

procédure accesv**(d pointeur** l**, d entier** val**, r pointeur** point**, r booléen** acces**)**
debproc booléen trouve**:=faux;**
  tantque l **!= nil) et (non** trouve**) faire**
   si l->info=val **alors** trouve**:=vrai;**
   sinon l:=l->suivant**;**
   finsi; finfaire;
  acces**:=**trouve**;**
  point:**=**l**;**
finfonc;

-schéma récursif

En utilisant une convention ci-après, on peut écrire cet algorithme sous forme d'une fonction récursive dont l'en-tête est **pointeur** point**(pointeur** liste**, entier** val**).**

Convention

- **point** devra contenir l'adresse de la cellule contenant la valeur de la 1ière occurrence de la valeur **val** si elle existe dans **liste+**.
- **point** devra contenir la valeur **nil** si la valeur **val** n'existe pas dans **liste+**.

Raisonnement

• liste+ est vide. L'algorithme est terminé, **point** délivre la valeur **nil** (val $\notin$ liste+).

retourner nil;

• liste+ n'est pas vide.

°° liste**-**>info=val. L'algorithme est terminé et **point** délivre la valeur contenue dans la variable **liste**. **retourner liste**;

°° liste**-**>info ≠ val. On recherche la présence de **val** dans la sous-liste **liste->suivant+.**

retourner (point(liste->suivant, val));

Accès par valeur, schéma récursif

```
pointeur fonction point(d pointeur l, d entier val)
debfonc   si l=nil alors retourner l;
          sinon
               si l->info=val alors retourner l;
               sinon retourner(point(l->suivant, val));
               finsi;
          finsi;
finfonc;
```

On notera qu'il est impossible de mettre en facteur l'action **retourner l** ; à cause des risques de l'incohérence dus aux conditions **l=nil et l->info** qui ne sont pas indépendantes.

Ecrivez sous forme récursive les exercices 8, 9, et 10.

Exercice 8 : booléen pluslongue (pointeur l1, pointeur l2)
//spécification()➔(pluslongue = l1 a un nombre d'éléments ≥ au nombre d'éléments de l2).

Exercice 9 : booléen memeliste (pointeur l1, pointeur l2)
//spécification()➔(l1 et l2 ont les mêmes éléments dans le même ordre)

Exercice 10 : booléen appartient (entier n, pointeur l)
//spécification()➔(appartient = x est une information d'une cellule de l)

c) Accès associatif dans une liste triée

Définition d'une liste triée

• une liste vide est triée,

• une liste d'un élément est triée,

• une liste de plus d'un élément est triée si tous les éléments consécutifs vérifient la relation d'ordre : **liste ≠ nil, liste->suivant ≠ nil, liste->info ≤ liste->suivant->info.**

-schéma récursif. On écrit une fonction d'en-tête **pointeur point(d pointeur l, d t val).**

• liste+ est vide. La valeur **val** n’est pas présente dans liste+, **return nil;** *

• liste+ n’est pas vide.

  °° liste->info<val, on doit chercher la $1^{\text{ière}}$ occurrence de **val** dans **liste->suivant+**.
  retourner point(liste->suivant, val);

  °° liste->info>val, la valeur **val** n’est pas dans la liste : **return nil**; *

  °° liste-> info=val, la cellule d’adresse liste contient la $1^{\text{ière}}$ occurrence de **val**.
  retourner liste ; *

pointeur fonction point**(**d **pointeur** l**,** d **entier** val**)**
debfonc **si** l**=nil alors retourner nil;**
sinon
si l->info<val **alors retourner** point**(**l->suivant, val**);**
sinon
si l->info>val **alors retourner nil;**
sinon retourner l**;**
finsi;
finsi; finsi;
finfonc;

Exercice 11. Ecrire sous forme itérative, un algorithme de recherche de la première occurrence d’une valeur dans une liste triée.

Exercice 12. Ecrire sous forme itérative et sous forme récursive, un algorithme de recherche de la dernière occurrence d’une valeur dans une liste triée.

4.1.6. Algorithmes de mise à jour dans une liste

Ces algorithmes ont une grande importance en raison de leur facilité de mise en œuvre. En effet, la mise en œuvre d'une liste n'entraîne que la modification d'un ou deux pointeurs sans recopie ni décalage des éléments non concernés par la mise à jour.

4.1.6.1. Algorithmes d'insertion dans une liste

a) Insertion d'un élément en tête de liste

C'est un cas très particulier car l'insertion en tête modifie l'adresse de la liste.

[Schéma]

Créer la cellule d'adresse p.

Le champ information reçoit la valeur de l'élément.

Terminer par la réalisation des 2 liaisons (a) et (b) dans cet ordre. Les éléments suivants sont décalés automatiquement d'une position.

procédure insertete**(dr pointeur** l**, entier** elem**)**
debproc
  pointeur p**;**
  nouveau(p); //création d'une cellule d'adresse p.
  p->info:=elem**;** //remplissage du champ info
  p->suivant:=l**;** //liaison a
  l:=p**;** //liaison b
finproc;

b) Algorithme d'insertion d'un élément en fin de liste

- <u>schéma itératif</u>

Si la liste est vide, on est ramené à une insertion en tête. Dans le cas général, la liste n'est pas vide.

[schéma]

Après avoir créé une cellule d'adresse **p** contenant l'information **elem**, on doit effectuer les liaisons (a) et (b). Pour pouvoir effectuer la liaison (b), il faut connaître l'adresse **der** de la dernière cellule.

Disposant de la fonction **dernier** dont l'en-tête est **pointeur dernier(pointeur liste)**, nous pouvons écrire l'algorithme.

```
procédure inserfin(dr pointeur l, d entier elem)
debproc  pointeur der, p;
    si l=nil alors insertete(l, elem);
    sinon
      der:=dernier(l);
       nouveau(p);        //1
       p->info:=elem;   //2
       p->suivant:=nil;  //3 fin de liste
       der->suivant:=p; //4 chaînage en fin de liste
     finsi;
finproc;
```

Les actions 1, 2, 3 et 4 correspondent à **insertete (der->suivant, elem).** En effet, insérer en fin de liste est équivalent à insérer en tête de la liste qui suit (c'est-à-dire la liste vide).

insertion en fin de liste, schéma itératif version 2.

procédure inserfin (dr **pointeur** l**,** d **t** elem)
//Spécification (l+ = la+) ➔ (l+ = la+ ||elem)
debproc
    pointeur der, p**;**
    si l=nil **alors** insertete(l, elem)**;**
    sinon
      der:=dernier(l);
      insertete(der->suivant, elem);
    finsi;
finproc;

c) Ecriture de la fonction dernier

- schéma itératif

On effectue un parcours de tous les éléments de la liste en préservant à chaque fois l'adresse de la cellule précédente.

pointeur fonction dernier**(**d **pointeur** l)
debproc pointeur precedent;
    tantque l≠nil **faire**
       precedent:=l;
       l:=l->suivant;
    finfaire;
    retourner precedent; **finproc;**

- <u>schéma récursif</u>

Le raisonnement est :

• liste ne contient qu'une seule cellule (l-> suivant = nil). La liste contient l'adresse de la dernière cellule.

retourner l ; *****

• liste contient plus d'une cellule (l->suivant ≠ nil).

retourner dernier(l->suivant);

Algorithme

pointeur fonction dernier(d **pointeur** l)
defonc
  si l->suivant=nil **alors** retourner l;
  sinon retourner dernier(l->suivant)**;**
finfonc;

- <u>schéma récursif de inserfin</u>

On peut également donner une version récursive de la fonction **inserfin** en utilisant la fonction **insertete**.

Le raisonnement :

• liste+ = vide, On insère l'élément. **retourner insertete(liste, elem)**; *

• liste+ ≠ vide, **retourner inserfin(liste)->suivant, elem);**

Algorithme

procédure inserfin(dr **pointeur** l, d **entier** elem)
debproc
  si l=nil **alors** insertete(l, elem)**;**
  sinon inserfin(l->suivant, elem)**;**
  finsi;
finproc;

d) **Algorithme d'insertion d'un élément à la k^ième^ place**

[Schéma]

- <u>schéma itératif</u>

L'insertion d'un élément à la **k**^**ième**^ place consiste à créer les liaisons (a) et (b) dans cet ordre. Les éléments suivants sont automatiquement décalés d'une position. Pour réaliser la liaison (b), il faut connaître l'adresse de la cellule précédente (k-1^ème^). L'insertion n'est possible que si **k∈[1..n+1]** où **n** est le nombre d'éléments de la liste. Il faudra prévoir l'insertion en tête (k==1) car elle modifie l'adresse de la liste. On utilisera la fonction **pointk** pour déterminer l'adresse de la **k-1**^**ème**^ cellule.

Insertion d’un élément à la **k**$^{\textbf{ième}}$ place, schéma itératif.

```
procédure insertk(dr pointeur l, d entier k, d entier elem, r booléen possible)
debproc  pointeur p, precedent;
   possible:=faux;
   si k=1 alors
     insertete(l, elem); possible:=vrai;
   sinon
      precedent:=pointk(l, k-1);
      si precedent≠nil alors
         nouveau(p);                                   //1
         p->info:=elem;                                //2
         p->suivant:=precedent ->suivant;              //3
         precedent->suivant:=p;                        //4
         possible:=vrai;
      finsi;
   finsi;
finproc;
```

On peut également constater que l’exécution des actions 1, 2, 3, 4 de l’algorithme correspond à l’exécution de l’action **insertete** pour la liste **precedent->suivant+** en écrivant à la place de ces actions, l’action **insertete(precedent->suivant, elem)**;

On obtient la deuxième version.

```
procédure insertk(dr pointeur l, d entier k, d entier elem, r booléen possible)
debproc
   pointeur p, precedent;
   possible:=faux;
   si k=1 alors
     insertete(l, elem);
     possible:=vrai;
   sinon
      precedent:=pointk(l, k-1);
      si precedent≠nil alors
         insertete(precedent->suivant, elem);
         possible:=vrai;
      finsi;
   finsi;
finproc;
```

- <u>schéma récursif</u>

• **k = 1**, On effectue une insertion en tête de **liste+**
    insertete(liste, elem); *
• **k ≠ 1**
  °° **liste+ est vide,** l'insertion est impossible *
  °° **liste+ n'est vide**
    insertk(liste->suivant, k-1, elem, possible);

```
procédure insertk(dr pointeur l, d entier k, d entier elem, r booléen possible)
debproc
  si k=1 alors
     possible:=vrai; insertete(l, elem);
  else
     si l=nil alors possible:=faux;
     sinon insertk(l->suivant, k-1, elem, possible);
     finsi;
  finsi;
finproc;
```

e) **Algorithme d'insertion de la valeur** elem **après la première occurrence de la valeur** val. Il suffit de connaître l'adresse de la cellule qui contient la 1^{ière} occurrence de la valeur **val** pour pouvoir réaliser les liaisons (a), (b) dans cet ordre (schéma).

Insertion de **elem** après la première occurrence de **val**, schéma non récursif.

procédure insert(dr **pointeur** l, d **entier val,** d **entier** elem, r **booléen** possible)
debproc
    pointeur ptval;
    possible:=faux;
    ptval:=point(l, val);
    si ptval≠**nil** alors
        possible:**=**vrai;
        insertete(ptval->suivant, elem);
    finsi;
finproc;

- schéma récursif : on souhaite écrire la fonction **insert** sous-forme récursive.
Le raisonnement :
• **liste+ est vide,** l'insertion est impossible car val $\notin$ liste+.   *
• **liste+ n'est pas vide**
    °° **liste->info=val,** on effectue l'insertion en tête **liste->suivant**.
        insertete(liste->suivant, elem)   *
    °° **liste->info≠val**
        insert(liste->suivant, val, elem)

Insertion de elem après la 1ière occurrence de **val**, schéma récursif.

procédure insert(dr **pointeur l,** d **entier val,** d **entier elem,** r possible)
debproc
  si l=**nil alors** possible:=faux;
  sinon
    si l->info=val **alors**
      insertete(l->suivant, elem);
      possible:=vrai;
    sinon
      insert(l->suivant, val, elem);
    finsi;
finproc;

Exercice 13. Ecrire, sous forme itérative et récursive, les algorithmes d’insertion d’un élément avant la première occurrence d’une valeur.
Exercice 14. Ecrire, sous forme itérative et récursive, les algorithmes d’insertion d’un élément avant toutes les occurrences d’une valeur.

f) <u>Algorithme d'insertion dans une liste triée</u>

On considère que la liste est la concaténation de 2 sous-listes **la** et **lb** telle : liste+ =la+||lb+ et la+<elem≤lb+. Nous avons mis l'égalité à droite afin de minimiser le temps de parcours.

Cas particuliers :

liste+ est vide, il suffit d'insérer en tête de **liste+**.

la+ est vide, **elem** est donc inférieur ou égal à la 1ière valeur de la liste. On effectue une insertion en tête de **liste+**.

lb+ est vide, **elem** est supérieur à tous les éléments de la liste. On insère en fin de liste, donc en tête de la liste dont le point d'entrée se trouve dans le champ suivant de la dernière cellule.

L'algorithme consiste à parcourir toute la liste **la+** et à insérer **elem** en tête de la liste **lb+**, dont le point d'entrée se trouve dans le champ **suivant** de la cellule d'adresse **dernier** (la+).

<u>-schéma récursif</u> : l'en-tête de la fonction est **void insertri(pointeur *l, t elem).**

Raisonnement

- **liste+ est vide,** on effectue l'insertion en tête de **liste+** : **insertete (liste, elem) ;** *
- **liste+ n'est pas vide**
  - °° **liste->info < elem**, **liste->info** appartient à la+. **insertri (liste->suivant, elem);**
  - °° **liste->info ≥ elem,** on est positionné sur le 1er élément de **lb+ : insertete(liste, elem);**

Algorithme

procédure insertri(dr **pointeur** l, **entier** elem)
debproc
  si l=**nil alors** insertete(l, elem);
  sinon
    si elem≤l->info **alors** insertete(l, elem);
    sinon insertri(l->suivant, elem);
    finsi;
  finsi;
finproc;

- schéma itératif : il faut effectuer un parcours séquentiel afin de déterminer les listes **la+** et **lb+.** On peut utiliser une variable auxiliaire qui contient l'adresse de la cellule précédente **preced**.

[schéma]

Initialisation

On a 2 cas particuliers :

liste+ est vide, il faut effectuer une insertion en tête de **liste+** : **insertete(liste, elem);**

liste+ n'est pas vide, elem ≤ liste->info, on effectue une insertion en tête de **liste+.**

  insertete (liste, elem);

Ces 2 cas particuliers étant traités, on réalise l'initialisation **preced = liste;**
p = liste->suivant;

<u>Algorithme d’insertion d’un élément dans une liste triée, schéma itératif</u>

```
procédure insertri(dr pointeur l, d entier elem)
debproc  pointeur p, precedent; booléen super;
   si l=nil alors insertete(l, elem);
   sinon
      si elem ≤ l->info alors insertete(l, elem);
      sinon
         precedent:=l; p:=l->suivant; super:=vrai;
         tantque (p != NULL) et super faire
            si elem>  p->info alors
              precedent=p; p=p->suivant;
            sinon super:=faux;
            finsi;
          finfaire;
          insertete(preced->suivant, elem);
      finsi;
   finsi;
finproc;
```

4.1.6.2. Algorithme de suppression d’un élément dans une liste

On a le cas particulier de suppression du 1ier élément qui a pour conséquence de modifier l’adresse de la liste.

Dans le cas général, il suffit de modifier le contenu d’un pointeur.

[schéma]

a) Algorithme de suppression du premier élément

On suppose que la liste n'est pas vide.

[schéma]

Il faut préserver l'adresse de la tête de liste avant d'effectuer la modification de cette adresse.

procédure supptete(dr **pointeur** l)
debproc
  pointeur p;
  p:=l;
  l:= l->suivant;
  laisser(p**);**
finproc;

b) Algorithme de suppression par position : **s**upprimer le $k^{ième}$ élément.

- <u>schéma itératif</u>

Il faut déterminer l'adresse **precedent** de la cellule qui précède celle que l'on veut supprimer. C'est-à-dire l'adresse de la $\mathbf{k\text{-}1^{ième}}$ cellule qui sera obtenue par la fonction **pointk**. Ensuite si le $\mathbf{k^{ième}}$ cellule existe, on modifie la valeur d'adresse **precedent**.

[schéma]

Algorithme : au préalable, on aurait pris soin de traiter le cas particulier du premier élément.

procédure supprimek(dr **pointeur** l, **entier** k, **booléen** possible)
debproc **pointeur** ptk, precedent;
    si l≠**nil et** (k=1) **alors**
      possible:=vrai; supptete(l);
    sinon
      possible:=faux; precedent:=pointk(l, k-1);
        si precedent≠**nil alors**
            ptk:=precedent->suivant;
            si ptk≠**nil alors**
             //ptk = adresse de la k^{ième} cellule et precedent = adresse de la cellule précédente.
              possible:=vrai;
              precedent->suivant:=ptk->suivant;
              laisser(ptk);
            finsi;
         finsi;
     finsi;
finproc;

-<u>schéma récursif.</u> Le raisonnement est :

•**liste+ est vide,** la suppression est impossible. car val $\notin$ liste+. On exécute **possible:=faux;** *

•**liste+ n'est pas vide**

°° **k=1,** on effectue la suppression en tête liste. **Possible:=vrai; supptete(liste);** *

°° **k≠1, supprimek(liste->suivant, k-1, possible);**

Suppression du k^{ième} élément, schéma récursif.

```
procédure supprimek(dr pointeur l, entier k, r booléen possible)
debproc
    si l=nil alors possible:=faux
    sinon
          si k=1 alors
            possible:=vrai; supptete (l);
          sinon supprimek(l->suivant, k-1, possible);
          finsi;
     finsi;
finproc;
```

<u>Algorithme de suppression associative</u>

Supprimer la 1^{ière} occurrence de **val** de la liste. Il faut définir une variable booléenne **possible** qui nous permettra de savoir si cette suppression a été réalisée ou non.

- <u>schéma récursif</u>. On souhaite écrire une fonction dont l'en-tête est :

procédure suppval(dr pointeur liste, int val, r booléen possible)

<u>Raisonnement</u>

• **liste+ est vide,** la suppression est impossible, car val $\notin$ liste+ : **possible:=faux;** *

• **liste+ n'est pas vide**

  °° **liste->info=val,** on effectue la suppression en tête : **possible:=vrai;**
supptete(liste); *

  °° **liste->info≠val,** on fait l'appel récursif : **suppval(liste->suivant, val, possible);**

Suppression associative, schéma récursif.

```
procédure suppval(dr pointeur l, entier val, r booléen possible)
debproc
    si l=nil alors possible:=faux;
    sinon
       si l->info=val alors
            supptete(l);
            possible:=vrai;
        sinon suppval (l->suivant, val, possible);
        finsi;
    finsi;
finproc;
```

Exercice 15. Ecrire sous forme itérative et récursive, les algorithmes de suppression de toutes les occurrences d'une valeur.

Exercice 16. Ecrire sous forme itérative et récursive, les algorithmes de suppression de la dernière occurrence d'une valeur.

Exercice 17. Ecrire sous forme itérative et récursive, les algorithmes de suppression de la première occurrence d'une valeur dans une liste triée.