3. FICHIERS

La structure d'information à étudier est la **structure de fichier séquentiel**. L'accès séquentiel à des fichiers garde encore, une importance pratique dans des **problèmes de gestion**. Nous allons définir, les objets formant un fichier comme une **collection d'objets** et les propriétés fonctionnelles de ces objets comme les caractéristiques formelles de cette collection.

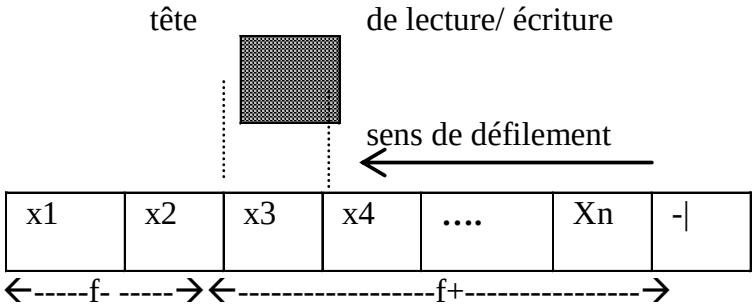
La plupart des **fonctions** ou **primitives** permettant la manipulation des fichiers séquentiels.

3.1. Définition et propriétés

Un **fichier séquentiel** à valeurs de V est une application f de P dans V.
où P est un ensemble de places. Et V est un ensemble de valeurs.
Un **fichier séquentiel** est représenté séquentiellement par la suite ordonnée de ses éléments, notée : **<x1, x2, x3,…,xn>** où xi € V.
Propriétés de fichiers séquentiels :
- les fichiers se trouvent en état d'**écriture** ou bien en état de **lecture**.
- On **accède** uniquement à un seul enregistrement se trouvant en face de la tête de L/ E.
- Après chaque **accès**, la tête de L/ E est déplacée derrière la donnée lue en dernier lieu.
- Un fichier peut être vide.

Un **sous-fichier** est la **restriction** d'un fichier **f** à un intervalle **P** (suite d'éléments **consécutifs**). **P** est un ensemble fini totalement ordonné.
Ex : **f** = <5, 7, 59, 67, -1, 0, 18> alors <5, 7, 59> et <67, -1, 0, 18> sont des sous-fichiers de **f**. Par contre, <5, 59, 18> n'est pas un sous-fichier.

Tout fichier peut être schématisé par un ruban défilant devant une **tête** de L/ E.
Elément **accessible** ou élément **courant** est l' élément en **face** de la tête de L/ E.
Un fichier étant une **suite finie**, on introduit une marque sur le ruban notée **-|** (**fin** de fichier).



L'élément courant (x3) permet, par définition, de décomposer un fichier en deux sous-fichiers : f = f- || f+
Où f- est le sous-fichier déjà lu et f+ est le sous-fichier qui reste à lire. L'élément courant est le premier élément de f+.
Dans cet exemple : f- = <x1, x2>, f+ = <x3,…,xn>, l'élément courant est x3.

3.2. Actions primitives d'accès

Elles donnent un **modèle** du **comportement dynamique** d'un fichier.

3.2.1. Prédicat fin de fichier

fdf(nom de fichier)

Ce prédicat prend la valeur **vrai** si on a atteint la fin de fichier **et faux** dans le cas contraire.

C'est l'exécution des primitives : (relire, lire, récrire et écrire) qui permet de donner une valeur au prédicat **fdf**.

3.2.2. Primitives d'accès au premier élément

relire(nom de fichier)

Elle rend possible l'accès au premier élément s'il existe. Et on peut lire ce premier élément.

Afin de permettre la détection de la fin de fichier, le prédicat **fdf** prend la valeur **vrai** si la tête de lecture/écriture est positionnée en face de la marque de fin de fichier, **faux** dans le cas contraire.

3.2.3. Primitives d'accès à l'élément courant

lire(nom de fichier, val)

Elle permet d'accéder à la valeur de l'élément courant et de sélectionner l'élément suivant.
Elle provoque dans l'ordre :

- la **copie** de **l'élément courant** (lecture) pour la mettre dans **val**.
- l'avancement du ruban d'une position. Si la marque de fin de fichier est alors sélectionnée alors **fdf** prend la valeur **vrai** sinon **fdf** conserve la valeur f**aux**.

Cette primitive ne peut pas être utilisée si on a atteint la fin de fichier.

Exemple

Soit le fichier f =<24, -10, 53>

Action exécutée	f^-	f^+	val	fdf(f)
			indéfini	indéfini
relire(f)	<>	<24, -10, 53>	indéfini	faux
lire(f, val)	<24>	<-10, 53>	24	faux
lire(f, val)	<24, -10>	<53>	-10	faux
lire(f, val)	<24, -10, 53>	<>	53	vrai
lire(f, val)	Impossible fdf			

3.2.4. Primitives de création d'un nouveau fichier vide

récrire(nom de fichier)

Cette primitive a pour objet de créer un nouveau fichier vide ayant pour nom "nom de fichier ". Le prédicat **fdf (nom de fichier)** prend la valeur **vrai**.

Si le fichier contenait au préalable des informations, celui-ci deviendrait vide et toutes les informations seraient perdues.

3.2.5. Primitives d'ajout d'un élément

écrire(nom de fichier, val)

On ne peut pas réaliser cette primitive que si la **tête d'écriture est positionnée en fin de fichier** (**fdf** a la valeur **vrai**).

L'interprétation de cette primitive est la suivante ; on effectue dans l'ordre :

- **recopie**, **en fin de fichier**, de la valeur contenue dans la variable **val**,
- **avancement** du ruban d'une position, afin de positionner la tête d'écriture sur la marque de fin de fichier,
- après exécution, le prédicat **fdf** a toujours la valeur **vrai**. Exemple. Construction d'u n fichier f.

Action exécutée	val	f
récrire(f)		<>
val:=5; écrire(f, val);	5	<5>
val:=8; écrire(f, val);	8	<5, 8>
écrire(f, val);	8	<5, 8, 8>
écrire(f, 18);	18	<5, 8, 8, 18>

3.3. Algorithmes traitant un seul fichier

Nous proposons des algorithmes mettant en jeu un seul fichier.

3.3.1. Somme des éléments d'un fichier

On se propose d'écrire une procédure **somme** qui calcule la somme de tous les éléments du fichier composés de nombres entiers.

```
procédure somme(d fichier de entier f, r entier som)
debproc
  entier s, c;
  relire(f); lire(f, s);
  tantque non fdf(f) faire
    lire(f, c); s:=s+c;
  finfaire;
  som:=s;
finproc;
```

Remarque

Cas d'un fichier vide

Dans le cas d'un fichier vide, il est interdit d'utiliser cet algorithme. On doit alors écrire un autre plus général à condition de préciser une convention.

Convention

La somme des éléments d'un fichier vide est égale à zéro. Avec cette convention, il suffit d'

Exécuter, à l'initialisation, les actions suivantes :

relire(f); s:=0;

On peut écrire cet algorithme sous-forme de fonction :

```
entier fonction sommef(d fichier de entier f)
debfonc
  entier som, c;
  relire(f); som:=0;
  tantque non fdf(f) faire
    lire(f, c); som:=som+c;
  finfaire;
  retourner som;
finfonc;
```

Cette écriture est préférée à la procédure dans le cas où il n' a qu'un seul résultat de type simple : entier, booléen, caractère, chaine.

Exercice 3.1. Ecrire un algorithme qui délivre le nombre d'éléments d'un fichier. On écrira cet algorithme sous forme de procédure puis de fonction.

Exercice 3.2. Ecrire une fonction qui délivre la somme des éléments de rang pair d'un fichier de nombres entiers ($x_2+x_4+\ldots$).

Exercice 3.3. Ecrire une fonction qui délivre la différence entre la somme des éléments de rang impair et celle de rang pair ($x_1-x_2+x_3-x_4+\ldots$).

Exercice 3.4. Ecrire une fonction qui délivre la valeur du dernier élément du fichier.

Exercice 3.5. Ecrire une fonction qui délivre la somme du premier élément et du dernier élément du fichier (par convention si le fichier ne contient qu'un seul élément x_1, le résulta sera égal à $2x_1$).

3.3.2. Recherche de la valeur maximale contenue dans un fichier

Soit un fichier **f** non vide, tel que l'ensemble des valeurs soit muni d'une relation d'ordre total.

On cherche une valeur $\mathbf{m \in f}$ et telle que $\forall \mathbf{y \in f}$, on ait $\mathbf{m \geq y}$. Une telle valeur est dite maximale.

L'algorithme est le suivant :

```
t maximum(d fichier de t f)
defonc
   t max, c;
   relire(f); lire(f, max);
   tantque non fdf(f) faire
    lire(f, c);
    si(max<c)max:=c;
   }
   retourner max;
finfonc;
```

Pour les algorithmes précédents, l'itération est toujours de la forme :

relire(f);

…

tantque non fdf(f) faire

  lire(f, c);

  traiter(c);

finfaire;

…

Où **traiter** est une procédure quelconque agissant ou non sur le contenu de **c**. A chaque exécution de l'action **lire(f, c),** le nombre d'éléments de **f+** diminue de 1.

3.3.3. Accès à un élément d'un fichier

Ce problème est habituellement formulé de 2 manières :

- Soit on connaît le rang **k** de l'élément et on recherche alors le **k^{ème}** élément du fichier,
- Soit on connaît la valeur de l'élément que l'on cherche et on recherche le premier élément qui correspond à cette valeur (la première occurrence)

On parle alors de recherche ou d'accès :

- par **position** dans le premier cas,
- par **valeur** ou **associatif** dans le second.

3.3.3.1. Accès par position (accès au k^{ème} élément)

L'en-tête de la procédure est le suivant :

procédure accesk(d fichier de t f, d entier k, r booleen trouve, r t valk)

On peut décomposer le fichier **f** en deux sous-fichiers $\mathbf{f_1^{k-1}}$ et $\mathbf{f_k^n}$,

d'où : $\mathbf{f = f_1^{k-1} \,||\, f_k^n}$

Pour que le **k^{ème}** existe, le sous-fichier $\mathbf{f_k^n}$ ne doit pas être vide ($\mathbf{f_k^n \neq <>}$). Dans ce cas, et si on a effectué la lecture des **k-1** premiers éléments, **f+** commence par le **k^{ème}** élément du fichier **f**.

L'algorithme est alors composé de 2 parties :

- Parcours des **k-1** premiers éléments de **f** (**$f^- = f_1^{k-1}$**)
- Ensuite, suivant que le fichier **f^+** est vide ou non, on aura les résultats suivants :

** f^+ est vide, **trouve** prendra la valeur **faux** et **valk** sera indéfinie.

** f^+ n'est pas vide, **trouve** prendra la valeur **vrai** et **valk** contiendra la valeur du **$k^{ème}$** élément en effectuant l'action **lire(f, valk) ;**

a) construction de $f^- = f_1^{k-1}$

```
relire(f);
i:=1;
tantque (i<k) et non fdf(f) faire
  lire(f, c);
  i:=i+1;
finfaire;
```

On a obtenu la $1^{ière}$ partie de l'algorithme.

b) Suite de l'algorithme

A la sortie du ″tantque″on a ¬((i < k), (¬fdf(f)), soit d'après la loi de De Morgan :
(i ≥k) v fdf(f).

i > k signifie que k ≤ 0.
En effet, **i** est initialisé avec la valeur 1.
D'autre part i augmente de 1 à l'intérieur de **tantque**.
Si **k>0**, la négation de **i<k** ne peut être que **i=k**. On ne peut avoir **i>k** sans être passé au préalable par la valeur de **i** égale à celle de **k**.
Donc **i>k** ne peut jamais se produire.

Nous allons étudier tous les cas possibles de l'assertion **(i=k)||fdf(f).**

• i = k, fdf(f)

On est à la fin de fichier, et on a trouvé que le rang de l'élément était égal à **k**. Le fichier a donc **k-1** éléments, le sous-fichier $\mathbf{f^+}$ est vide et le $\mathbf{k^{ième}}$ élément n'existe pas : $\mathbf{f = f_1^{k-1}}$, **trouve** prend la valeur **faux** et **valk** est indéfinie.

• i = k, ¬ fdf(f)

signifie qu'on a trouvé le $\mathbf{k^{ième}}$ élément sans atteindre la fin du fichier.

On a donc : ($f^- = f_1^{i-1}$, i = k) ➔ $f^- = f_1^{k-1}$

Il s'ensuit que $\mathbf{f^+ = f_1^n}$ avec $\mathbf{k \leq n}$; **lire(f, valk)** permet à **valk** de prendre la valeur du $\mathbf{k^{ième}}$ élément et **trouve** doit prendre la valeur **vrai**.

• i < k, fdf(f)

On a parcouru tous les éléments du fichier sans trouver le $\mathbf{k^{ième}}$ élément.

$\mathbf{f^- = f_1^n}$**, f+** est vide, **trouve** prend la valeur **faux** et **valk** est indéfinie.

• i < k, ¬ fdf(f)

Ce cas ne peut jamais se produire car l'expression **(i=k)||(fdf(f))** est fausse et n'est pas donc une assertion.

On peut résumer ces 4 cas dans un tableau appelé **tableau de sortie**.

Tableau 2 : tableau de sortie de tantque

i =k	**fdf(f)**	**résultat**
vrai	vrai	trouve := false
vrai	faux	trouve:=vrai; lire(f, valk)
faux (i<k)	vrai	trouve := false
faux (i<k)	faux	impossible (à cause de tantque)

En examinant les 3 premières lignes du tableau, on constate que la valeur de **trouve** est identique è celle de ¬**fdf(f)**. Il faut donc continuer l'algorithme en utilisant **un test sur la valeur du prédicat**.

Par contre, utiliser un test sur la valeur de **i=k**, entraîne une erreur dans le cas de la 1ière ligne. On aurait donc écrit un algorithme qui aurait fonctionné dans la plupart des cas, sauf si par hasard on avait rencontré le cas d'un fichier contenant **k-1** éléments.

D'où l'intérêt de la construction et de l'analyse du tableau de sortie qui permet de compléter l'algorithme.

On poursuit l'algorithme en écrivant :

si non fdf(f) alors
trouve:=vrai;
lire(f, valk);
sinon trouve = faux;
finsi;

ou encore

trouve = non fdf(f);
si trouve alors
lire(f, valk);
finsi;

Une autre solution consiste à initialiser **trouve** à **faux** au début de l'algorithme, il suffit alors de terminer l'algorithme avec :

if non fdf(f) alors
trouve=vrai;
lire(f, valk);
}

C'est cette dernière solution que nous utilisons fréquemment, car elle permet d'alléger l'algorithme en évitant l'écriture de **sinon**.

Algorithme d'accès au **k^{ième}** élément.

```
procédure accesk(d fichier de t f, d entier k, r booléen trouve, r t valk)
debproc
  entier i; t c;
  relire(f);
  trouve := faux; i:=1;
  tantque (i<k) et non fdf(f) faire
    lire(f, c);
     i:=i+1;
  finfaire;
  si non fdf(f) alors
    trouve := vrai;
    lire(f, valk);
  finsi;
finproc;
```

3.3.3.2. Accès associatif

Soit un fichier **f** et une valeur **val** de même type que les éléments de **f**. On veut savoir si la valeur contenue dans **val** est présente dans le fichier **f (= f_1^n)**.

val∈f signifie que **∃i ∈ [1..n], x_i = val**.

```
booléen fonction acces(d fichier de t f, d t val)
debfonc
  t c; booléen egal;
  relire(f); egal:=faux;
  tantque non fdf(f) et non egal faire
    lire(f, c);
    si c=val alors egal:=vrai;
    finsi;
  retourner egal;
finfonc;
```

Tableau 3 : Tableau de sortie

fdf(f)	**egal**	**résultat : val ∈ f**
vrai	vrai	trouve = vrai (val = Df)
vrai	faux	trouve = faux
faux	vrai	trouve = vrai
faux	faux	impossible (tantque)

On constate que **trouve** a la même valeur que **egal**.

Evaluation du coût d'un accès dans un fichier séquentiel.

Un accès séquentiel nécessite au mieux une lecture au plus n lectures si n est le nombre d'éléments du fichier.

Tout accès séquentiel est proportionnel au nombre d'éléments du fichier (le facteur de proportionnalité dépend du dispositif sur lequel se trouve le fichier et des temps de lecture associés)

Exercice 3.6. Ecrire une fonction qui calcule le nombre d'occurrences d'une valeur **val** dans un fichier.
Exercice 3.7. Ecrire une fonction qui calcule le rang de la dernière occurrence de la valeur **val** dans un fichier.
Exercice 3.8. Ecrire une fonction qui vérifie qu'un fichier possède au moins n éléments.
Exercice 3.9. Ecrire une fonction qui calcule le nombre d'occurrences de la valeur **val** entre le $i^{ème}$ et $j^{ème}$ élément.
Exercice 3.10. Ecrire une fonction qui délivre le rang de la première occurrence de la valeur **val** (cette fonction délivre zéro si la valeur val est absente du fichier).
Exercice 3.11. Ecrire une fonction qui délivre le nombre d'occurrences de la valeur **val**1 entre les premières occurrences des valeurs **val2** et **val3** dans cet ordre.

3.3.4. Fichier ordonné (fichier trié)

Si tous les éléments consécutifs d'un fichier vérifient la relation d'ordre ($\mathbf{x_i \leq x_{i+1}}$), on dit que le fichier est trié par ordre croissant.
On appellera fichier **trié** (ou **ordonné**) un fichier ordonné **par ordre croissant**.
On admettra que le **fichier vide** et que le fichier ne contenant qu'**un seul élément** sont **triés**.
En d'autres termes, la définition d'un fichier trié :
- un fichier vide est trié,
- un fichier comportant un seul élément est trié,
- soit $f = <x_1, x_2, \ldots, x_n>$, avec n>1, le fichier f est trié si $\forall i \in [1..n-1]$, la relation $x_i \leq x_{i+1}$ est vraie.

On peut donner une **définition récursive** d'un fichier trié :
- un fichier vide est trié,
- un fichier contenant un seul élément est trié,
-(f_1^i trié, $x_i \leq x_{i+1}$)➔ f_1^{i+1} trié pour i $\in$ [1..n-1].

3.3.4.1. Algorithme vérifiant qu'un fichier est trié.
Ecrire un algorithme qui vérifie si un fichier est trié ou non.

```
booléen fonction trie(d fichier de t f)
debfonc
  t precedent, c;
  relire(f);
  si fdf(f) alors retourner vrai;
  sinon
    lire(f, precedent);
    c:=precedent;
    tantque (non fdf(f)) et (precedent≤c) faire
      precedent:=c; lire(f, c);
    finfaire;
    retourner precedent≤c;
  finsi;
finfonc;
```

3.3.4.2. Algorithme d'accès à un élément dans un fichier trié

En général, l'accès associatif dans un fichier ordonné est plus rapide que dans le cas d'un fichier non ordonné.
On cherche à écrire une fonction dont l'en-tête est :
booléen fonction accest**(**d **fichier de t** f**,** d **t** val**)**

On peut toujours décomposer un fichier **f** trié et non vide en 2 sous-fichiers f' et f'' tel que : f = f' || f'', f'< val ≤ f''.
Remarque
Si f' est vide, la relation s'écrit : val ≤ f''
Si f'' est vide, la relation s'écrit f' < val.

On contrôle l'itération par une variable booléenne **infer** définie par :
(infer, f- < val) || (!infer, Df- ≥ val).

Algorithme

booléen **fonction accest(**d **fichier de t** f**,** entier **val)**
debfonc
  t **c**;
  relire(f);
  si fdf(f) alors retourner faux;
  sinon
     lire(f,c);
     tantque non fdf(f) et c<val faire
       lire(f, c);
     finfaire;
     retourner val=c;
   finsi;
finfonc;

Exercice 3.11. Ecrire une fonction qui délivre le nombre d'occurrence de la valeur **val** dans un fichier).

Exercice 3.12. Ecrire une fonction qui délivre le rang de la dernière occurrence de la valeur **val** dans un fichier trié.

Exercice 3.13. Ecrire une fonction qui recherche la valeur **val** après le rang i dans un fichier trié.

3.4. algorithmes traitant plusieurs fichiers

Ces algorithmes prennent des éléments sur un ou plusieurs fichiers et construisent un ou plusieurs fichiers.

3.4.1. Algorithme de création, par copie, d'un fichier

Problème : créer à partir d'un fichier **f=<x_1, x_2, …, x_n>**, un fichier identique

g=<x_1, x_2, …, x_n>.

Algorithme : parcourir le fichier f et ranger dans g chaque élément de f.

```
procédure copie(d ficher de t f, r fichier de t g)
debproc
  t c;
  récrire(g); relire(f);
  tantque non fdf(f) faire
    lire(f, c); écrire(g, c);
  finfaire;
finproc;
```

3.4.2. Somme ou concaténation de deux fichiers

A partir de deux fichiers **f** et **g**, on veut construire un troisième **h** tel que : **h = f || g**, L'algorithme consiste à copier **f** dans **h**, puis **h** complété par les éléments de **g**.

```
procédure concat(d fichier de t f, d fichier de t g, r fichier de t h)
debproc
  t c;
  relire(f), récrire(h);
  tantque non fdf(f) faire
    lire(f, c); écrire(h, c);
  finfaire;
  relire(g);
  tantque non fdf(g) faire
    lire(g, c); écrire(h, c);
  finfaire;
finproc;
```

3.4.3. Eclatement d'un fichier en plusieurs fichiers

On veut construire m fichiers (g1, g2, …, gm) à partir d'un seul fichier séquentiel f en utilisant un critère d'éclatement. Ce problème est très courant dans toutes les applications de gestion : ventilation de fichier.

3.4.3.1. Eclatement d'un fichier en deux fichiers suivant le critère de parité

On veut construire deux fichiers, l'un contenant les nombres entiers pairs du fichier de données, l'autre les nombres entiers impairs. On appellera **fpair** et **fimpair** ces deux fichiers. Le fichier **f** de données est le fichier pilote.

On utilise un choix à deux alternatives (si sinon) permettant de choisir le fichier **fpair** ou **fimpair** suivant la parité de l'élément de **f**.

```
procédure pair1pair(d fichier de entier f, r fichier de entier fpair, r fichier de entier fimpair)
debproc
  entier c;
  relire(f), récrire(fpair); récrire(fimpair);
  tantque non fdf(f) faire
    lire(f, c);
    si pair(c) alors écrire(fpair, c);
    sinon écrire(fimpair, c);
    finsi;
  finfaire;
finproc;
```

3.4.3.2. Eclatement d'un fichier en deux fichiers suivant le rang de l'élément

Eclater un fichier f en deux fichiers de telle sorte que tous les éléments de rang pair sont copiés dans le fichier f2 et tous les éléments de rang impars dans le fichier f1.

Exemple : f=<2, 3, 1, 6, 8, 5, 4> alors f1=<2, 1, 8,4> et f2=<3, 6, 5>,

Solution1 : à chaque itération, on effectue deux lectures et de ranger le premier élément dans f1 et le second dans f2. Pour la deuxième lecture, on doit s'assurer que l'élément existe bien en se protégeant par le prédicat **fdf**.

```
procédure rpairimpair(d fichier de t f, r fichier de t f1, r fichier de t f2)
debproc
  entier c ;
  relire(f), récrire(f1); récrire(f2);
  tantque non fdf(f) faire
    lire(f, c); écrire(f1, c);
    si non fdf(f) alors
       lire(f, c); écrire(f2, c);
    finsi;
  finfaire;
finproc;
```

Solution 2 : utiliser une variable booléenne **rangimpair** qui prend :

- la valeur **vrai** lorsque **c** contient une valeur de rang impair.
- la valeur **faux** lorsque **c** contient une valeur de rang pair.

L'algorithme est :

```
procédure rpairimpair(d fichier de t f, r fichier de t f1, r fichier de t f2)
debproc
  entier c ; booléen rangimpair;
  relire(f), récrire(f1); récrire(f2);
  rangimpair := faux;
  tantque non fdf(f) faire
    lire(f, c);
    rangimpair := non rangimpair;
    si rangimpair alors écrire(f1, c);
    sinon écrire(f2, c);
    finsi;
  finfaire;
finproc;
```

Exercice 3.12. Reprise de l'exercice 3.2. en utilisant une bascule : écrire une fonction qui délivre la somme des éléments de rang pair d'un fichier de nombres entiers (x_2+x_4+---).

Exercice 3.13. Reprise de l'exercice 3.3. en utilisant une bascule : écrire une fonction qui délivre la différence entre la somme des éléments de rang impair et la somme des éléments de rang ($x_1-x_2+x_3-x_4+x_5-x_6+$---).

3.4.5. Fusion de plusieurs fichiers en un seul

Ce problème consiste à construire, à partir de plusieurs fichiers séquentiels, un fichier suivant un critère de fusion. Prenons l'exemple de l'interclassement de deux fichiers triés. Soient deux fichiers triés f et g, on veut construire un fichier h trié contenant tous les éléments de f et de g.

```
procédure interclassement(d fichier de t fs, d fichier de t gs, r fichier de t hs)
debproc
  t cf, cg;
  relire(fs), relire(gs); récrire(hs);
  lire(fs, cf); lire(gs, cg);
  tantque non fdf(fh) ou non fdf(gh) faire
    si cf ≤ cg alors
      écrire(hs, cf); lire(fs, cf);
    sinon
       écrire(hs, cg); lire(gs, cg);
    finsi;
  finfaire; //fdf(fs) ou fdf(gs)
  écrire(hh, cf); {ou écrire(hh, cg);}
  tantque non fdf(fs) faire
    écrire(hs, cf); lire(fs, cf);
  finfaire;
  tantque non fdf(gs) faire
    écrire(hs, cg); lire(gs, cg);
  finfaire;
finproc;
```

Exercice 3.13. Ecrire une procédure d'union de deux fichiers triés sans répétition (fichier obtenu doit être sans répétitions).
Exercice 3.14. Ecrire une procédure d'intersection de deux fichiers triés sans répétitions (le fichier obtenu doit être sans répétitions).

3.4.6. Algorithmes de mise à jour

Il s'agir d'insérer ou supprimer des éléments dans un fichier, La mise à jour ne porte que sur un seul élément. L'élément peut être désigné soit par position soit par valeur. Le fichier peut être trié ou non,

3.4.6.1. Insertion d'un élément dans un fichier

a) Aucun critère n'est donné

Soit un fichier f et une valeur **valinsert** à ajouter dans le fichier. On a deux possibilités : ajouter l'élément en tête de fichier f ou bien l'ajouter en fin de fichier f.
Ce qui revient à la concaténation de f avec une valeur. Le résultat est :
Soit : <valinsert> ||f
Soit : f || <valinsert>
Cas1. Il faut créer un autre fichier ce qui donne :

```
procédure insertdeb (d t valinsert, d fichier de t f, r fichier de t g)
debproc
  t c;
  récrire(g); écrire(g, valinsert); relire(f);
  tantque non fdf(f) faire
    lire(f, c); écrire(g, c);
  finfaire;
finproc;
```

Cas 2. On crée d'abord un autre fichier par copie du fichier f; puis on ajoute la valeur **valinsert** à la fin. On obtient :

```
procédure insertfin (d t valinsert, d fichier de t f, r fichier de t g)
debproc
  t c;
  récrire(g); relire(f);
  tantque non fdf(f) faire
    lire(f, c); écrire(g, c);
  finfaire;
  écrire(g, valinsert);
finproc;
```

b) Insertion par position

On donne la place **k** du nouvel élément **valinsert**. Il s'agit d'écrire un algorithme d'insertion à la $\mathbf{k}^{\text{ième}}$ place de l'élément **valinsert** dans le fichier **f**. On construit un fichier **g** résultat de la concaténation de trois fichiers :

$$g = f_1^{k-1} \,\|\, \langle valinsert \rangle \,\|\, f_k^{n}$$

L'algorithme se décompose de trois parties :

- copies des **k-1** premiers éléments de **f**,
- ajout de l'élément **valinsert**.
- copies de **n-k+1** derniers éléments de **f**.

```
procédure inserk(d fichier de t f, d entier k, d t valinsert, r fichier de t g, r boooléen possible)
debproc
  t c;

//construction de g = f_1^(k-1)
  récrire(g); relire(f);
  tantque non fdf(f) et i<k faire
    lire(f, c); écrire(g, c); i:=i+1;
  finfaire;
  si i=k alors
  //ajout de l'élément valinsert
    écrire(g, valinsert);
  //copie de n-k+1 derniers éléments de f,
    tantque non fdf(f) faire
      lire(f, c); écrire(g, c);
    finfaire;
    possible := vrai;
  sinon possible:=faux;
  finsi;
finproc;
```

c) Insertion associative

On veut insérer la valeur **valinsert** après ou avant (on choisit après) la valeur **val** dans le fichier **f**.

Solution : en utilisant l'algorithme de copie, il suffit de copier la valeur **valinsert** après la première occurrence de **val.**

```
procédure inserv(d fichier de t f, d t val, d t valinsert, r fichier de t g, r boooléen possible)
debproc
  t c; booléen trouve;
  récrire(g); relire(f); trouve := faux;
  tantque non fdf(f) faire
    lire(f, c); écrire(g, c);
    si (c=val) et non trouve alors
      trouve:=vrai; écrire(g, valinsert);
    finsi;
  finfaire;
  possible:=trouve;
finproc;
```

Exercice 3.14. Ecrire une procédure d'insertion de la valeur **valinsert** après chaque occurrence de la valeur **val** dans le fichier **f**.

Exercice 3.15. Ecrire une procédure d'insertion de la valeur **valinsert** après la dernière occurrence de la valeur **val**.dans le fichier trié **f**.

3.4.6.2. Suppression d'un élément dans un fichier

On peut supprimer le k^{ième} élément (suppression par position) ou supprimer une valeur (suppression associative).

a) Suppression par position

On veut supprimer le **k^{ième}** élément du fichier **f**.

On considère que l'on construit un fichier **g** qui est la concaténation de sous-fichiers : $\mathbf{f_1^{k-1}}$ et $\mathbf{f_{k+1}^{n}}$.

L'algorithme se décompose alors de trois parties :

- copie des **k-1** premiers éléments de **f**.
- lecture du **k^{ième}** élément (sans copie)
- copie des **n-k** derniers éléments de **f**.

```
procédure supprimek(d fichier de t f, d entier k, r fichier de t g, r booléen possible)
debproc
  entier i; t c;
  // copie des k-1 premiers éléments de f.
  relire(f); i:=1; récrire(g);
  tantque non fdf(f) et i<k faire
    lire(f, c); écrire(g, c); i:=i+1;
  finfaire;
  si (i=k) et non fdf(f) alors
  //lecture du kième élément (sans copie)
    lire(f,c);
```

```
    tantque non fdf(f) faire
      lire(f, c); écrire(g, c);
    finfaire;
    possible := vrai;
  sinon possible:=faux;
  finsi;
finproc;
```

a) Suppression par position

On veut supprimer la première occurrence de la valeur **val** dans un fichier non vide. La suppression n'est possible que si **val** appartient au fichier.

```
procédure supprime(d fichier de t f, d t val, r fichier de t g, r booléen possible)
debproc
  t c;
  relire f; récrire g;
  posible :=faux; relire(f, c);
  tantque non fdf(f) et (c≠val) faire
    écrire(g, c); lire(f, c);
  finfaire;
  si c=val alors
    possible :=vrai;
    tantque non fdf(f) faire
      lire(f, c); écrire(g, c);
    finfaire;
  finsi;
finproc;
```