Algorithmique & programmation

Chapitre 3 : Fichiers séquentiels

Algorithme traitant un seul fichier

Fichiers triés

Accès associatif fichier trié

Fichier trié

- Définition
  - un fichier vide est trié,
  - un fichier comportant un seul élément est trié,
  - soit un fichier $f = <x_1, x_2, ..., x_n>$, $n > 1$
    - f est trié si $\forall i \in [1..n-1]$, $x_i \leq x_{i+1}$

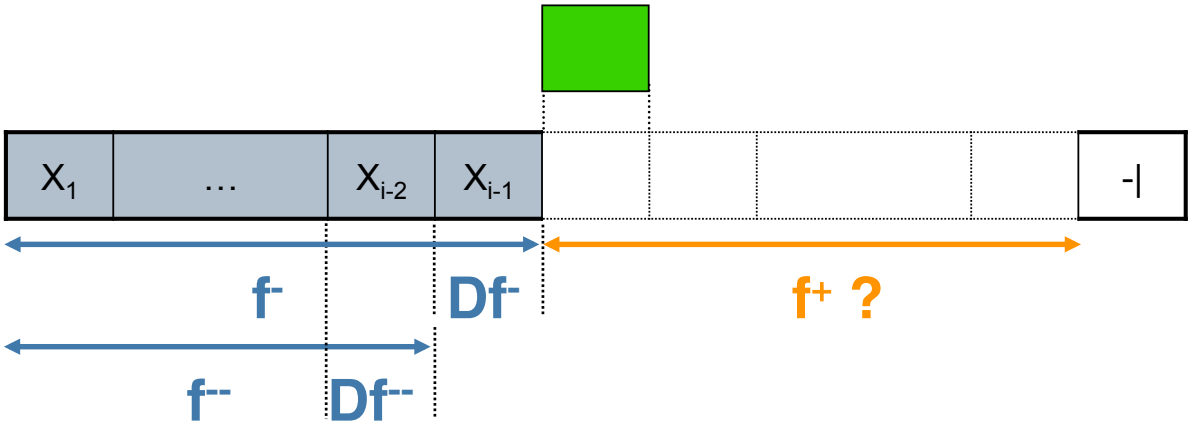
- Définition récursive :
  - un fichier vide est trié,
  - un fichier à un seul élément est trié,
  - **si** f_1^i trié, $x_i \leq x_{i+1}$
    alors f_1^{i+1} trié pour $i \in [1..n-1]$

Vérifier qu'un fichier est trié

- Vers l'hypothèse dans le cas général
  - Il faut disposer de deux éléments à comparer pour vérifier qu'ils sont dans l'ordre
    - le dernier élément dont on dispose est **Df^{-}**
    - l'avant dernier élément est forcément **Df^{--}**
      - on l'appellera **précéd**
  - Le sous-fichier précédemment examiné est trié

Vérifier qu'un fichier est trié

- En schématisant



f^{--} trié, précéd = Df^{--}, c = Df^{-}

Vérifier qu'un fichier est trié

- Raisonnement par récurrence

Hypothèse f^{--} trié, précéd = Df^{--}, c = Df^{-}

- précédent > c ⇨ *f n'est pas trié* ✳
- précédent ≤ c
  - $f^{+} = <>$ ⇨ *f* est trié ✳
  - $f^{+} \neq <>$ ⇨ précéd := c ; lire (f , c) ; ➡ ***H***

Itération tantque (précéd ≤ c) et non fdf(f) faire...

Vérifier qu'un fichier est trié

- Raisonnement par récurrence (suite)

Hypothèse f^{--} trié, précéd = Df^{--}, c = Df^{-}

Initialisation

relire (f) ;

- $f^{+} = <>$ ⇨ *f est vide donc trié* ✳
- $f^{+} \neq <>$ ⇨

  lire (f , précéd) ;
  - $f^{+} = <>$ ⇨ *f a un élément donc trié* ✳
  - $f^{+} \neq <>$ ⇨ lire (f , c) ; ➡ ***H***

Vérifier qu'un fichier est trié

- Tableau de sortie

fdf(f)	précéd > c	résultat
vrai	vrai	faux
vrai	faux	vrai
faux	vrai	faux
faux	faux	impossible (tantque)

- trouvé est vrai lorsque précéd ≤ c
  - **trouvé := précéd ≤ c ;**

fonction **trié**

fonction trié (d f : fichier de t) : booléen ;
spécification *{ }* ➔ *{résultat = f est trié}*
    c , précéd : t ;
debfonc
    relire (f) ; *{f = < >, f trié}*
    si fdf (f) **alors**
        retour vrai ; *{un fichier vide est trié}*
    sinon
        lire (f , précéd) ; *{précéd = Df}*
        si fdf (f) **alors**
            retour vrai ; *{un fichier ayant un seul élément est trié}*
        sinon
            lire (f , c) ; *{f⁻ trié , précéd = Df⁻ , c = Df}*
            tantque non fdf (f) **et** (précéd ≤ c) **faire**
                précéd := c ; *{f trié , précéd = Df , c = Df}*
                lire (f , c) ; *{f⁻ trié , précéd = Df⁻ , c = Df}*
            finfaire ; *{fdf (f)) ou (précéd > c)}*
            retour précéd ≤ c ;
        finsi ;
    finsi ;
finfonc ;

procédures **trie** & **acces** (fichier d'entiers)

```
-- Importer le module générique Sequential_io, use inutile
with Sequential_Io;

package P_Fentier is
   -- Instancier le paquetage Sequential_Io
   -- Pour manipuler des fichiers de Integer
   package P_Entier_Io is new Sequential_Io(Integer);
   use P_Entier_Io;

   -- Première version de la fonction trie
   function trié1(F : in P_Entier_Io.File_Type) return boolean;

   -- Seconde version de la fonction trie
   function trié2(F : in P_Entier_Io.File_Type) return boolean;

   -- Accès associatif dans un fichier trié
   function accest(F : in P_Entier_Io.File_Type ;
                                   val : in integer) return boolean;

end P_Fentier;
```


fonction **trié** (version1)

```
function trié1 (F : in P_Entier_Io.File_Type) retrun boolean is
--spec { } ➔ {résultat = f est trié}
   c , précéd : integer ;
begin
   reset(f) ;
   if fdf (f) then
      retrun true ;    --un fichier vide est trié
   else
      read (f , précéd) ;
      if End_Of_File(F) then
         return true ; --un fichier d'un seul élément est trié
      else
         read (f , c) ;
         while not End_Of_File(F) and (précéd <= c) loop
             précéd := c ;
             read (f , c) ;
         end loop ;
         return précéd ≤ c ;
      end if ;
   end if ;
end trié1;
```

fonction **trié** (algo simplifié par l'initialisation)

fonction trié (d f : fichier de t) : booléen ;
spécification { } ➔ *{résultat = f est trié}*
    c , précéd : t ;
debfonc
    relire (f) ; *{f = < >, f trié}*
    si fdf (f) **alors**
        retour vrai ; *{un fichier vide est trié}*
    sinon
        lire (f , précéd) ; *{précéd = Df}*
        c := précéd ; *{c = précéd}*
        tantque non fdf (f) **et** (précéd ≤ c) **faire**
            précéd := c ; *{f trié , précéd = Df , c = Df}*
            lire (f , c) ; *{f⁻ trié , précéd = Df⁻ , c = Df}*
        finfaire ; *{fdf (f)) ou (précéd > c)}*
        retour précéd ≤ c ;
    finsi ;
finfonc ;

fonction **trié2** (version1 simplifiée)

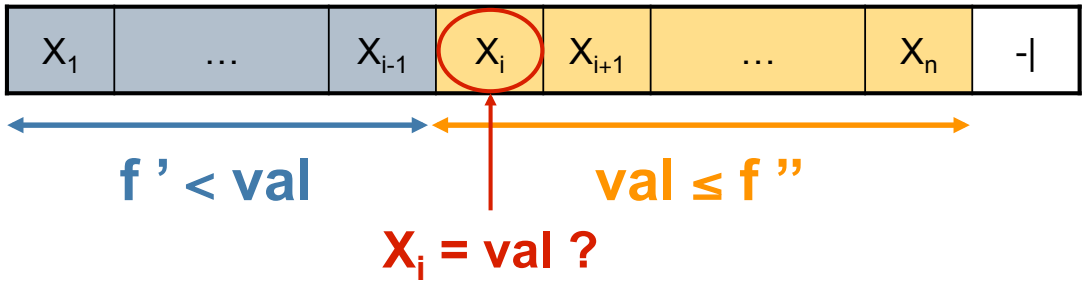
```
function trié2 (F : in P_Entier_Io.File_Type) retrun boolean is
--spec { } ➔ {résultat = f est trié}
  c , précéd : integer ;
begin
  reset(F) ;
  if End_Of_File(F) then
    retrun true ;    --un fichier vide est trié
  else
    read (f , précéd) ;
    c := précéd ;
    while not End_Of_File(F) and (précéd <= c) loop
      précéd := c ;
      read (f , c) ;
    end loop ;
    return précéd <= c ;
  end if ;
end trié2;
```

Accès associatif (fichier trié)

- Deux versions comme pour l'accès associatif dans un fichier non trié
- On peut toujours décomposer un fichier f trié et non vide en deux sous-fichiers f ' et f ''
  - tels que : f = f ' || f '' , f ' < val ≤ f ''
- Remarques
  - si f ' = <> , alors val ≤ f ''
  - si f '' = <> , alors f ' < val
- Algorithme en 2 parties :
  - obtenir f^{--} = f ',
  - vérifier si Df^{-} existe, et alors si val=Df^{-}

Accès associatif (fichier trié)

- En schématisant



Première partie (construction de f ' = f^{--})

Raisonnement par récurrence

Hypothèse $f^{--} < val$, $c = Df^{-}$

- $c \geq val$ ⇨ *c premier élément de* f^{-} ✳
- $c < val$ ⇨ $f^{-} < val$
  - $f^{+} = <>$ ⇨ $f < val, val \notin f$ ✳
  - $f^{+} \neq <>$ ⇨ lire (f , c) ; ➡ ***H***

Itération tantque (c < val) et non fdf(f) faire...

Initialisation relire (f) ;

- $f^{+} = <>$ ⇨ $val \notin f$ ✳
- $f^{+} \neq <>$ ⇨ lire (f , c) ; ➡ ***H***

Seconde partie (produire le résultat)

Tableau de sortie

fdf(f)	c ≥ val	résultat
vrai	vrai	➤ c = val ➔ résultat = vrai ➤ c > val ➔ résultat = faux
vrai	faux	faux
faux	vrai	➤ c = val ➔ résultat = vrai ➤ c > val ➔ résultat = faux
faux	faux	impossible (tantque)

On a trouvé lorsque c = val

- **résultat := c = val ;**

fonction **accèst**

fonction accest (d f : fichier de t; d val : t) : booléen ;
spécification *{f trié}* ➔ *{résultat = val ∈f}*
    c : t ;
debfonc
    relire (f) ;
    si fdf (f) **alors**
        retour faux ;
    sinon
        lire (f , c) ; *{f⁻ < val}*
        tantque non fdf (f) **et** (c < val) **faire** *{f < val}*
            lire (f , c) ; *{f⁻ < val}*
        finfaire ;
        {f⁻ < val , fdf (f) ou (c ≥ val)}
        retour val = c ;
    finsi ;
finfonc ;

fonction **accèst**

```
function accest (F : in P_Entier_Io.File_Type ;
                 val : in integer) retrun boolean is
--spec {f trié} ➔ {résultat = val ∈f}
  c : integer ;
debfonc
  reset(F) ;
  if End_Of_File(F) then
    return false ;
  else
    read (F , c) ;
    while not End_Of_File(F) and (c < val) loop
      read (F , c) ;
    end loop ;
    return val = c ;
   end if ;
end accest ;
```