

Notion d'allocations statique et dynamique

Représentation (allocation) statique

L'allocation de l'espace se fait tout à fait au début d'un traitement. En termes techniques, on dit que l'espace est connu à la compilation. C'est donc la notion de tableau.

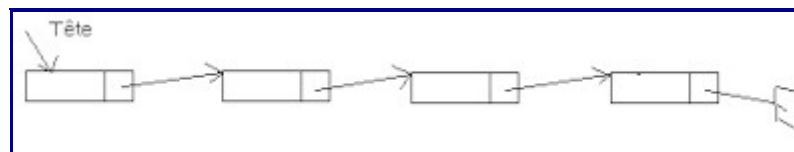
Représentation dynamique

L'allocation de l'espace se fait au fur et à mesure de l'exécution du programme. Pour pouvoir faire ce type d'allocation, l'utilisateur doit disposer des deux opérations : allocation et libération de l'espace. Si le langage offre ces possibilités, on les utilise directement, sinon on sera dans l'obligation de les simuler, c'est à dire gérer l'espace soi-même dans un grand tableau.

Définition d'une liste linéaire chaînée

Une liste linéaire chaînée **Llc** est un ensemble de maillons (alloués dynamiquement) chaînés entre eux.

Schématiquement, on peut la représenter comme suit :



Un élément d'une **Llc** est toujours une structure (objet composé) avec deux champs :

- Un champ Valeur : contenant l'information
- Un champ Adresse : donnant l'adresse du prochain maillon

A chaque maillon est associée une adresse. On introduit ainsi une nouvelle classe d'objet: le type **POINTEUR** en langage algorithmique. Une **Llc** est caractérisée par l'adresse de son premier élément. **NIL** constitue l'adresse qui ne pointe aucun maillon.

Dans le langage algorithmique, on définira le type d'un maillon comme suit :

```
TYPE Typedumaillon = STRUCTURE  
    Valeur : Typeqq           { désigne un type  
    quelconque }  
    Adresse : POINTEUR      (Typedumaillon)  
FIN
```

Modèle sur les listes linéaires chaînées

Afin de développer des algorithmes sur les Llcs, on construit une machine abstraite avec les opérations définies comme suit :

Allouer(T, P)	allocation d'un espace de taille spécifiée par le type T. L'adresse de cet espace est rendue dans la variable POINTEUR P
Libérer(P)	libération de l'espace pointé par P
Valeur(P)	consultation du champ Valeur du maillon d'adresse P
Suivant(P)	consultation du champ Adresse du maillon d'adresse P
Aff_Adr(P, Q)	dans le champ Adresse du maillon d'adresse P, on range l'adresse Q
Aff_Val(P, Val)	dans le champ Valeur du maillon d'adresse P, on range la valeur Val



On notera cet ensemble d'opérations MLlc. Cet ensemble est appelé modèle.

Exemple d'introduction

Supposons que l'on veuille résoudre le problème suivant :

**Trouver tous les nombres
premiers de 1 à n et les
stocker en mémoire**

Le problème réside dans le choix de la structure d'accueil. Si on utilise un tableau, il n'est pas possible de définir la taille de ce vecteur avec précision même si nous connaissons la valeur de n (par exemple 10000). Ne connaissant pas la valeur de n, on n'a aucune idée sur le choix de sa taille. On est donc, ici, en face d'un problème où la réservation de l'espace doit être dynamique.

solution

On utilise les deux faits suivants :

- un nombre n'est premier que s'il n'est pas divisible par les nombres premiers qui le précèdent
- tous les nombres premiers sont de la forme $6m+1$ ou $6m-1$.



L'Algorithme

{ Création de maillons pour les nombres premiers 2 et 3 }

```
Allouer(T,P)
Aff_Val(P,2)
Tête := p
Allouer(T, Q)
Aff_Val(Q,3) ; Aff_Adr(Q, NIL)
Aff_Adr(P,Q)
P := Q
M := 1 ; Continue := VRAI ; Aig := VRAI
TANTQUE Continue :
    Gen_nombre (M, Aig, Nombre)
    Aig := NON Aig
    SI Aig : M := M + 1 FSI
        SI Nombre <= N :
            SI Premier (Tête, Nombre) :
                Allouer (T, Q)
                Aff_Val(Q, Nombre);Aff_Adr( Q, NIL)
                Aff_Adr(P, Q)
                P := Q
            FSI
        SINON
            Continue := FAUX
    FSI
FINTANTQUE
```

Les modules précédents sont définis comme suit :

Module Gen_nombre(M, Aig, Nombre)

En entrée :

- M qui prend des valeurs dans { 1, 2, 3, .. }
- Aig pour l'alternance du "+" et "-" dans la formule.

En sortie :

- Nombre : le prochain nombre candidat pour le module Premier.

C'est tout simplement :

```

SI Aig : Nombre := 6M - 1
    SINON Nombre := 6M + 1 FSI

```

Module Premier (L, N)

en entrée

- L : une tête de LLC

- N : un nombre

en sortie

- la valeur VRAI si L ne contient pas de diviseurs de N, FAUX sinon.

```

P:= L ; Trouv := FAUX
TANTQUE P <> NIL ET NON Trouv :
    SI Divisible ( Nombre, Valeur(P) ) :
        Trouv := VRAI
    SINON
        P := Suivant(P)
    FSI
FINTANTQUE
Premier := NON Trouv

```

Enfin, le module Divisible (A, B) peut s'écrire comme suit :

C'est un prédicat qui est égal à VRAI si B est un diviseur de A , FAUX sinon.

Q étant une variable de type ENTIER,

```

Q := A / B {Division entière}
SI Q.B = A : Divisible := VRAI
    SINON Divisible := FAUX FSI

```

Soit T le type d'un maillon de la Llc, P et Q des variables de type POINTEUR.

En supposant que :

Gen_nombre est un module qui détermine le prochain nombre candidat pour le test (premier ou pas)

Premier est un prédicat égal à VRAI si un nombre donné est premier, FAUX sinon

Algorithmes sur les listes

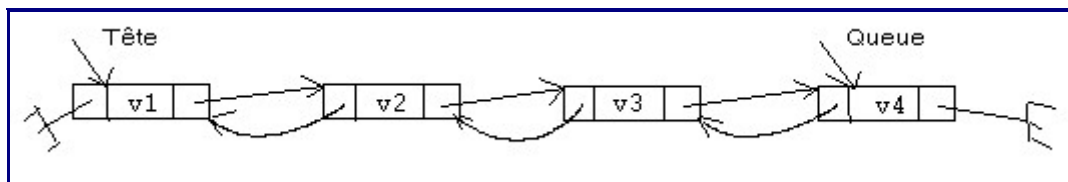
De même que sur les vecteurs, on peut classer les algorithmes sur les LLCs comme suit :

- **parcours** : accès par valeur, accès par position
- **mise à jour** : insertion, suppression
- **algorithmes sur plusieurs LLCs** : fusion, interclassement, éclatement,...
- **tri sur les LLCs**

Listes linéaires chaînées particulières

Liste bidirectionnelle

C'est une LLc que l'on peut *parcourir dans les deux sens*. Schématiquement, elle se présente comme suit :



$$MLlcb = MLlc - \{ \text{Aff_Adr} \} + \{ \text{Aff_Adrg}, \text{Aff_Adrd}, \text{Précédent} \}$$

Le type d'un maillon est défini comme suit :

```
TYPE      Typedumaillon = STRUCTURE
            Valeur : Typeqq
            Adresse1 : POINTEUR  (Typedumaillon)
            Adresse2 : POINTEUR  (Typedumaillon)
FIN
```

Le modèle des **Llc** est donc étendu par les opérations suivantes :

Précédent(P)	accès au champ adresse gauche du maillon d'adresse P.
Aff_Adrg(P, Q)	dans le champ Adresse1 (adresse gauche) du maillon d'adresse P, on range l'adresse Q.
Aff_Adrd(P, Q)	dans le champ Adresse2 (adresse droite) du maillon d'adresse P, on range l'adresse Q



Exemple

Suppression d'un élément pointé par P dans une liste linéaire chaînée bidirectionnelle L.

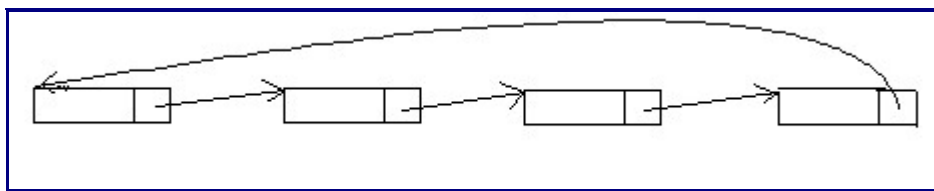
```

SI P # NIL :
    SI Précédent(P) # NIL :
        Aff_Adrd( Précédent(P), Suivant(P) )
    SINON
        L := Suivant (P)
    FSI
    SI Suivant(P) # NIL :
        Aff_Adrg( Suivant(P), Précédent(P) )
    FSI
    Libérer(P)
FSI

```

Liste circulaire ou anneau

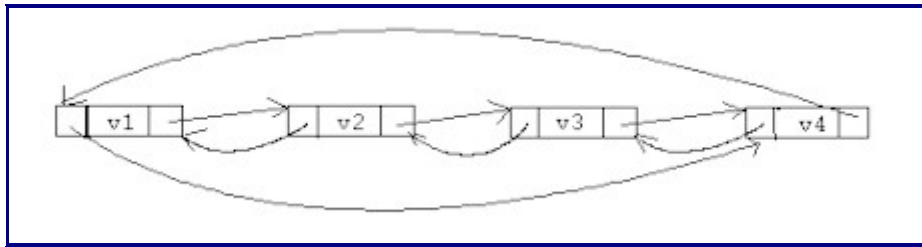
C'est une **Llc** telle que *le dernier élément pointe sur le premier*. Elle est définie par l'adresse d'un élément quelconque.



MLlcc = MLlc

Liste bidirectionnelle circulaire

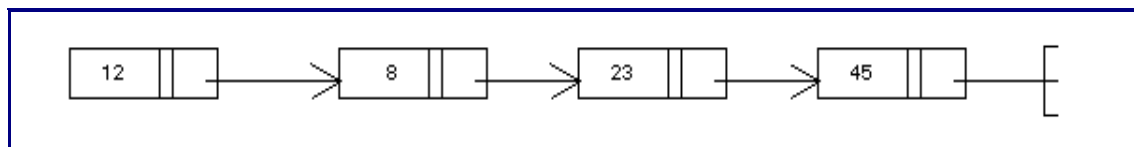
C'est une **Llc** à *double sens et dont le dernier(premier) pointe sur le premier(dernier)*.



MLlcbc = MLlcb

Implémentation des listes linéaires chaînées avec la représentation contigue

On peut représenter les **Lcs** dans un même tableau où chaque élément renferme au moins 2 champs : l'information et le lien. Par exemple la liste suivante



peut être représentée dans un tableau comme suit :

1	12	4	
2	23	7	
3			
4	8	2	
5			
6			
7	45	-1	

Définition de la structure

```
TYPE T = STRUCTURE
    Info : Typeqq
    Lien : ENTIER
    Vide : BOOLEEN { pour les positions vides du
tableau }
FIN
```



```
VAR T : TABLEAU ( 1 : Max ) DE T
```

Initialement les champs Vide de tous les éléments sont à vrai (pour indiquer des positions disponibles).

Une **Llc** est définie par l'indice de son premier élément dans le tableau T.

Définition, principe, domaine d'application

Une file d'attente peut être définie comme une collection d'éléments dans laquelle tout nouveau élément est inséré à la fin et tout élément ne peut être supprimé que du début.

C'est le principe **"FIFO"**, abréviation de **"First In, First Out"** qui veut dire " premier entré premier servi ".

La file d'attente est très utilisée dans les systèmes d'exploitation des ordinateurs et surtout dans les problèmes de simulation.

Nous verrons aussi que la file d'attente peut être utilisée pour le parcours des arbres et pour résoudre tant d'autres problèmes.

Modèle

C'est l'ensemble des opérations définies comme suit :

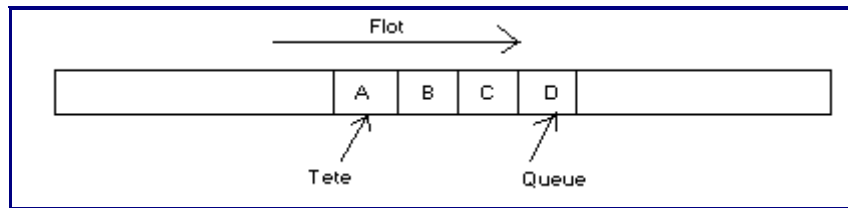
CréerFile(F)	créer une file vide
Enfiler(F,Val)	ajouter Val en queue de file.
Défiler(F,Val)	retirer dans Val l'élément en tête de file.
Filevide(F)	tester si la file est vide.
Filepleine(F)	tester si la file est pleine.

Implémentation

◆ ***Au moyen de tableaux*** (par flot par décalage Tableau circulaire)

◆ *Au moyen des listes linéaires chaînées*

par flot



la Description Algorithmique

```
TYPE Filedattente = STRUCTURE
    Elements : TABLEAU(1..Max) DE Typeqq
    Tête, Queue : ENTIER
FIN
VAR F : Filedattente

Créerfile(F) : F.Queue := 0 ET F.Tête := 1
Filevide : Filevide := (F.Queue < F.Tête)
Filepleine(F) : Filepleine := (F.Queue = Max)
Enfiler(F, X) : SI NON Filepleine(F) F.Queue := F.Queue + 1
                F.Elements(F.Queue) := X
                SINON "Overflow" FSI

Defiler(F,X) : SI NON Filevide(F) X := F.Elements(F.Tête)
                F.Tête := F.Tête + 1
                SINON "Underflow" FSI
```

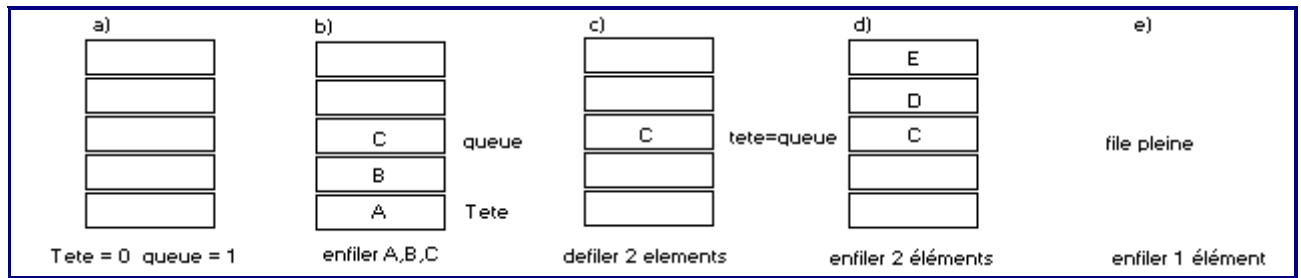
Remarque 1 :

A tout moment le nombre d'éléments est $F.queue - F.tête + 1$

Remarque 2 :

La file n'est pas vide si $F.queue \geq F.tête$, donc la file est vide si non ($F.Queue \geq F.Tête$), c'est à dire $F.Queue < F.Tête$.

Exemple :



C'est une solution inacceptable car on ne peut récupérer les emplacements X tels que $X < F.Tête$



par décalage

A chaque défilement, on fait un décalage vers le bas. La tête n'est plus une caractéristique de la file d'attente puisqu'elle est toujours égale à 1.



la Description Algorithmique

```

TYPE Filedattente = STRUCTURE
    Elements : TABLEAU(1..Max) DE Typeqq
    Queue : ENTIER
FIN
VAR F : Filedattente
Créerfile(F) : F.Queue := 0
Filevide : Filevide := (F.Queue = 0)
Filepeine : Filepleine := (F.Queue = Max)
Defiler(F, X) : SI NON Filevide(F) :
    X := F.Elements(1)
    POUR I := 1 à F.Queue - 1 :
        F.Elements(I) = F.Elements(I + 1)
    FINPOUR
    F.Queue := F.Queue - 1
    SINON " Filevide " FSI
Enfiler(F, X) : SI NON Filepleine(F) :
    F.Queue := F.Queue + 1
    F.Elements(F.Queue) := X
    SINON "Filepleine" FSI

```



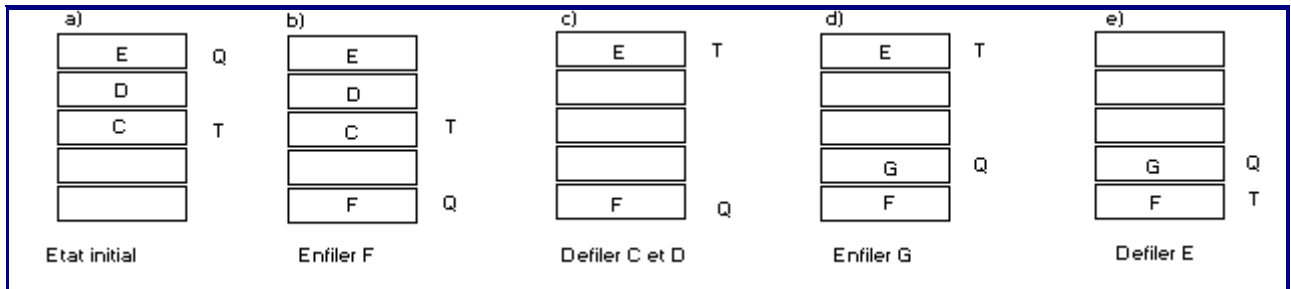
L'inconvénient de cette solution est que pour chaque défilement, on fait un décalage.



Tableau circulaire

Revenons à la solution par flot et essayons d'utiliser le tableau de façon circulaire.

Considérons l'exemple suivant :



➤ Comment initialiser la file ?

F.Queue < F.Tête impossible (contre exemple)

F.tête = F.Queue impossible (c'est le cas ou il reste un élément dans la file).

➤ Solution :

F.Tête : pointe l'élément qui précède le premier.

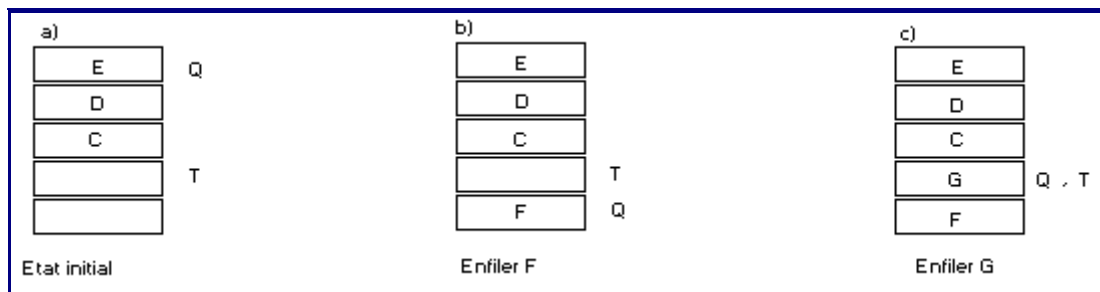
F.Queue : pointe le dernier élément.

(F.Tête = F.Queue) constitue alors le cas file vide.

Initialisation (F.Tête = F.Queue := Max)



Regardons ce qui se passe avec ces nouvelles considérations



F.tête = F.queue constitue aussi le cas file pleine.

La solution est donc de sacrifier un élément. le cas *par décalage* constitue le cas file pleine.



La Traduction

```

Créerfile(F) : F.Tête = F.Queue := Max
Filevide(F) : Filevide := (F.Tête = F.Queue)
Filepleine(F) : Filepleine := ( F.Tête = ( F.Queue Mod Max + 1 ) )
Enfiler(F, X) : SI NON Filepleine( F)
                (1) SI F.Queue = Max
                    F.Queue := 1

                SINON
                    F.Queue := F.Queue + 1
                FSI

                F.éléments(F.Queue) := X
                SINON " Overflow" FSI
Defiler(F, X) : SI NON Filevide(F)
                (2) SI F.Tête = Max
                    F.Tête := 1
                SINON
                    F.Tête := F.Tête + 1
                FSI

                X := F.éléments(F.Tête)
                SINON "Underflow" FSI

```

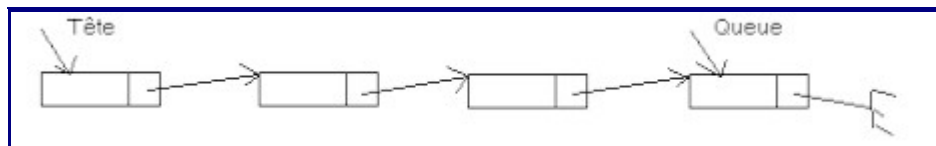
Remarque:

L'alternative (1) est équivalente à :

$F.Queue := F.Queue \text{ Mod } Max + 1$



◆ Au moyen des listes linéaires chaînées



La Description Algorithmique

```

TYPE S= STRUCTURE
    Info : Typeqq
    Suiv : POINTEUR(S)
FIN
TYPE Filedattente = STRUCTURE
    Tête, Queue : POINTEUR(S)

```

```

FIN
VAR F : Filedattente

Créerfile(F) : F.Tête := NIL
Filevide(F) : Filevide := ( F.Tête = NIL )
Enfiler(F, X) : Allouer Q(S)
                    Aff_Val(Q, X)
                    Aff_Adr(Q, NIL)
                    SI NON Filevide(F)
                        Aff_Adr(F.Queue, Q)
                    SINON
                        F.Tête := Q FSI
                    FSI

                    F.Queue := Q
Defiler(F, X) : SI NON Filevide(F)
                Sauv := F.Tête
                X := Valeur(F.Tête)
                F.Tête := Suivant(F.Tête)
                Libérer(Sauv)
                SINON "Underflow" FSI

```



TRAVAUX DIRIGÉS

1. Implémenter une file d'attente d'entiers en utilisant un tableau File(-1 : 100), où File(-1) est utilisée pour indiquer la tête, File(0) pour indiquer la queue et File(1:100) contient les éléments de la file d'attente. Comment initialiser la file vide ? Traduire les opérations du modèle.
2. Comment implémenter une file d'attente où chaque élément contient un nombre variable d'entiers.
3. Une *file d'attente avec priorité* est une file d'attente dans laquelle l'opération de défilement récupère l'élément le plus prioritaire. Définir le modèle et l'implémenter.

Définition, principe, domaine d'application

La pile constitue l'un des concepts les plus utilisés dans la science des ordinateurs.

Une pile peut être définie comme une collection d'éléments dans laquelle tout nouveau élément est inséré à la fin et tout élément ne peut être supprimé que de la fin.

C'est le principe "**LIFO**", abréviation de "**Last In First Out**" qui veut dire "*dernier entré premier servi*".

La pile est très utilisée dans le domaine de la compilation : résolution de la portée des objets, récursivité, évaluation d'expressions, etc..

Nous verrons aussi que la pile est utilisée pour le parcours des arbres et pour résoudre un grand nombre de problèmes.

Modèle

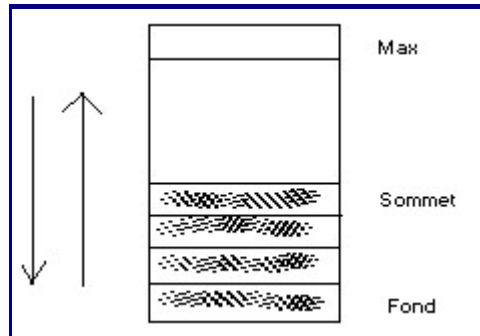
On définit une machine abstraite sur les piles avec l'ensemble des opérations suivantes :

Créerpile(P)	créer une pile vide
Empiler(P,Val)	ajouter Val en sommet de pile.
Dépiler(P,Val)	retirer dans Val l'élément en sommet de pile.
Pilevide((P)	tester si la pile est vide.
Pilepleine(P)	tester si la pile est pleine



Implémentation

◆ *Au moyen de tableaux*



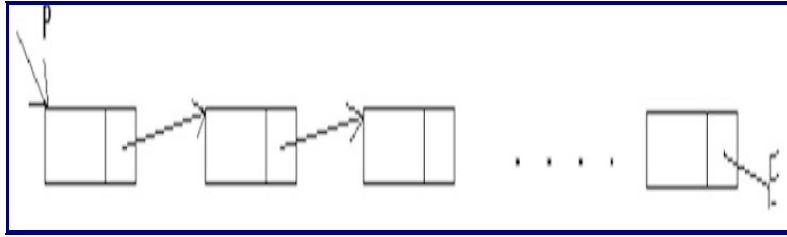
La Description Algorithmique

```
TYPE Typepile = STRUCTURE
    Som : ENTIER
    Tab : TABLEAU (1..Max) DE Typeqq
FIN
VAR P : Typepile

Créerpile(P) : P.Sommet := 0
Pilevide(P) : Pilevide := ( P.Sommet = 0 )
Pilepleine(P) : Pilepleine := ( P.Sommet = Max )
Empiler(P, X) : SI NON Pilepleine(P)
                P.Sommet := P.Sommet + 1
                P.Elements(P.Sommet) := X
                SINON "Overflow" FSI

Depiler(P, X) : SI NON Pilevide(P)
                X := P.Elements(P.Sommet)
                P.Sommet := P.Sommet - 1
                SINON " Underflow" FSI
```

◆ *Au moyen des listes linéaires chaînées*



Un empilement consiste à faire une insertion au début et un dépilement une suppression au début.



La Description Algorithmique

```

TYPE S = STRUCTURE
    Info : Typeqq
    Suiv : POINTEUR(S)
FIN
VAR P : POINTEUR(S)

Créerpile(P) : P := NIL
Pilevide(P) : Pilevide := (P = NIL)
Empiler(P, X) : Allouer Q(S)
                Aff_Adr(Q, P)
                Aff_Val(Q, X)
                P := Q

Dépiler(P, X) : SI NON Pilevide(P)
                X := Valeur(P)
                Sauv := P
                P := Suivant(P)
                Libérer(Sauv)
                SINON "Underflow" FSI

```

TRAVAUX DIRIGES

1. Représenter les expressions suivantes sous forme polonaise postfixée :

$a+b$, $(a+b)/d$, $((c+d) + (d-e)) + 5$, $-(a+b) + (5+b)c$, $-(((a+b) + (c-d))/5) + a5$

Essayer de donner l'algorithme d'évaluation sans utiliser la pile. Quels sont les problèmes rencontrés? Donner l'algorithme d'évaluation avec l'utilisation d'une pile.

2. Soit à analyser des expressions mathématiques qui utilisent les symboles (, { et pour l'ouverture des sous expressions et les symboles), } et pour leur fermeture respective. Chaque symbole de fermeture doit être associé à son symbole d'ouverture. Utiliser le modèle de pile pour écrire l'algorithme qui effectue une telle vérification.

3. Les éditeurs de texte utilisent, en général, deux caractères particuliers pour traiter une ligne d'un texte:

- le caractère d'effacement, noté #, permet d'effacer le caractère précédent,
- le caractère d'annulation, noté @, permet d'effacer toute la ligne courante.

Utiliser le modèle de pile pour écrire l'algorithme qui édite une ligne d'un texte.

4. Donner une implémentation possible(schéma+description) pour :

- une file de files,
- une file de piles,
- une pile de files,
- une pile de piles.

Exemples d'introduction

[La fonction factorielle](#)

[Multiplication des entiers naturels](#)

[Suite de Fibonacci](#)

[La recherche binaire](#)

♦ **La fonction factorielle**

Considérons la définition suivante de la factorielle :

$$\begin{aligned} n! &= 1 && \text{si } n=0 \\ n! &= n (n-1) (n-2) \dots 1 && \text{si } n > 0 \end{aligned}$$

```
Prod := N
POUR X = N-1 , 2 , -1 :
    Prod := Prod * X
FINPOUR
```

On peut aussi définir la factorielle par :

$$n! = 1 \quad \text{si } n = 0$$

$$n! = n (n-1)! \quad \text{si } n > 0$$

Dans cette définition, la factorielle est définie par elle-même. une telle définition est dite récursive.

Regardons maintenant comment évaluer 4! par cette définition ?

$$4! = 4 * 3!$$

$$3! = 3 * 2!$$

$$2! = 2 * 1!$$

$$1! = 1 * 0!$$

$$0! = 1$$

Chaque cas est défini par un cas plus réduit. La cinquième ligne définit directement la valeur. On remonte de 5 à 1 pour déterminer la valeur de 4!.

On peut écrire l'algorithme récursif, sous forme d'une fonction, comme suit :

Fact(N):

```
(1) SI N = 0
      (2) Fact := 1
(3) SINON
      (4) X := N - 1
      (5) Y := Fact(X)
      (6) Fact := N * Y
(7) FSI
```

Dans cet algorithme, (5) est la clé du problème. Cela veut dire qu'il faut exécuter de nouveau l'algorithme avec une nouvelle donnée, X. L'algorithme se termine puisque N atteindra la valeur 0. Puis, une séquence de multiplications est effectuée jusqu'au traitement de N!.



◆ Multiplication des entiers naturels

On peut définir la multiplication de deux entiers a et b de manière récursive comme suit :

$$a * b = a \quad \text{si } b = 1$$

$$a * b = a * (b-1) + a \quad \text{si } b > 1.$$

A partir de ces deux premiers exemples, on peut déjà constater que dans les définitions récursives :

- un cas simple est défini explicitement, ($0! = 1$; $a * 1 = a$). C'est ce qu'on appelle **le cas trivial**.

- les autres cas sont définis en appliquant une opération sur le résultat de l'évaluation d'un cas plus simple. $n!$ est défini au moyen de $(n-1)!$; $a * b$ est défini au moyen de $a * (b-1)$.

Il faut aussi qu'on soit sûr qu'à partir des simplifications successives on arrive au cas trivial. Pour la factorielle, n est diminuée successivement de 1, jusqu'à la valeur 0. Pour la multiplication, b diminue progressivement d'une unité jusqu'à la valeur 1.

Si on n'a pas ces conditions, la définition serait invalide.

Comme exemple, les définitions suivantes sont invalides:

$$n! = (n+1)! / (n+1)$$

$$a * b = a * (b+1) - a$$

Car il est impossible de déterminer $5!$ ou $6*3$ bien que les relations sont vraies.



◆ Suite de Fibonacci

Considérons la suite de Fibonacci :

$$\begin{array}{ll} \text{Fib}(n) = n & \text{si } n = 0 \text{ ou } 1 \\ \text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2) & \text{si } n \geq 2 \end{array}$$

La fonction Fib se référence à elle-même deux fois.

A travers cet exemple, nous voulons montrer qu'une fonction (algorithme) récursive peut faire appel à elle-même plusieurs fois.



◆ La recherche binaire

La récursivité est non seulement utilisée pour les fonctions mathématiques mais aussi pour de nombreuses applications informatiques. La recherche binaire en est un exemple.

L'algorithme récursif suivant fait une recherche binaire de l'élément X dans le tableau trié A [B_i , B_s]. l'algorithme rend dans Recherche l'indice Indice dans A tel que $A(\text{Indice}) = X$, 0 sinon.

Recherche (X , B_i , B_s) :

```

SI  Bi > Bs
    Recherche := 0
SINON
    Mil := ( Bi + Bs ) DIV 2
    SI X = A(Mil)
        Recherche := Mil
    SINON
        SI X < A(Mil)
            Rechercher(X, Bi, Mil - 1)
        SINON
            Rechercher (X , Mil + 1, Bs)
        FSI
    FSI
FSI

```

Propriétés

Un algorithme récursif ne doit pas générer une séquence infinie d'appels à lui-même. Il doit toujours y avoir une façon de sortir des appels récursifs **"Way out"**.

Dans les exemples précédents les **"Way out"** sont :

(1) Factorielle	$0! = 1$
(2) Multiplication	$a * 1 = a$
(3) Fibonnacci	$\text{fib}(0)=0$; $\text{fib}(1)=1$;
(4) Recherche binaire	Si $Bi > Bs$: $\text{Rech}:=0$ Si $A(\text{Mil})=X$: $R:= \text{Mil}$



Conception d'algorithmes récursifs

Pour certains problèmes, il est beaucoup plus facile de rechercher une solution récursive plutôt qu'une itérative. Le problème des tours de Hanoi constitue un très bon exemple comme nous allons le montrer ci-après.

Le problème des tours de Hanoi

On dispose de 3 tours A, B et C. Initialement, n disques de diamètres différents sont placés sur A. Un disque plus grand ne doit jamais se trouver sur un disque plus petit. Le problème est de transférer les n disques de A vers C, en utilisant B comme intermédiaire en respectant les deux règles suivantes :

➤ à un moment donné, seul le disque au sommet d'un piquet peut être transféré vers un autre.

➤ un disque plus grand ne doit pas se trouver sur un disque plus petit.

Il est très difficile de trouver une solution itérative (essayer...). Par contre, il existe une solution récursive simple et élégante.

Supposons que nous avons une solution pour $n-1$ disques. Nous pourrions alors définir une solution pour n disques au moyen de la solution de $n-1$ disques.

$n=1$ constitue le cas trivial : transférer le disque unique de A vers C.

Pour transférer n disques de A vers C, en utilisant B comme auxiliaire, on procède comme suit:

1. Si $n=1$, alors transférer l'unique disque de A vers C.
2. Transférer les $n-1$ disques au sommet de A vers B avec C comme auxiliaire.
3. Transférer le disque restant de A vers C.
4. Transférer les $n-1$ disques de B vers C avec A comme auxiliaire.

Nous remarquons que c'est simple car nous venons de développer une solution pour le cas trivial ($n=1$) et une solution pour le cas n en terme de solution pour le cas plus simple $n-1$.

L'algorithme est alors le suivant :

TourdeHanoi(N, De, Vers, Aux) :

```
Si N = 1    { cas trivial }
            "transfert du disque 1 de", De, "vers", Vers"
SINON
    TourdeHanoi(N-1, De,Aux,Vers)
{Transférer les N-1 disques au sommet de A vers B avec C comme
auxiliaire }
    "transférer le disque", n,"de",de,"vers", vers"
{ transférer le disque restant de A vers C }
    TourdeHanoi(N-1, Aux, Vers, De)
{ Transférer les N-1 disques de B vers C, utilisant A comme
auxiliaire }
FSI
```



Sémantique de la récursion

La récursivité est un outil puissant. Essayons de voir ce qui est caché derrière. En d'autres termes, nous voulons savoir qu'est-ce qui se passe quand on fait Fact(4) par exemple ?

A chaque appel, la fonction Fact est appelée avec un argument différent : 4, 3, 2 et 1. Tout se passe comme si on a plusieurs copies du code de la procédure. Ainsi pour chaque exécution, les variables locales ainsi que les paramètres en entrée sont réalloués. A un moment donné, seules les dernières allocations sont référencées. Quand un retour est fait, les plus récentes allocations sont libérées et une copie de la procédure est réactivée. C'est donc le mécanisme de pile. A chaque appel, une zone pour les variables locales et les paramètres est allouée. A chaque retour, il y a un dépilement.



Passage d'algorithmes récursifs en algorithmes itératifs

Plusieurs langages n'acceptent pas la récursivité (Fortran, Cobol, ...). Il est donc quelques fois nécessaire de faire le passage des algorithmes récursifs en algorithmes non récursifs.

Quelques notions de compilation doivent être connues pour comprendre la technique de récursion et donc le passage des algorithmes récursifs en algorithmes itératifs.

En général, la version non récursive s'exécute plus efficacement en temps et en espace. La raison principale est la pile. Quelquefois, la solution récursive est plus naturelle et plus logique pour résoudre un problème (Tour de Hanoi).

Notions de compilation

Technique de passage

Transformation sur un exemple : la factorielle

Autres règles pour la transformation



Notions de compilation

A chaque procédure est associée une zone de données qui contient les variables locales, les paramètres, les variables temporaires.

Les zones de données sont gérées en pile.

Au moment d'un appel, il y a sauvegarde de la zone de données de l'appelant, transmission des paramètres (par valeur ou par référence) et branchement vers le début de la procédure appelée. L'adresse de retour au niveau de l'appelant doit être également sauvegardée dans la zone de données de la procédure appelée.

Au moment d'un retour, il y a restauration de la zone de données et continuation de l'exécution de la procédure à partir de l'adresse de retour préalablement sauvegardée dans la zone de données de l'appelée.

Quand un paramètre est appelé par valeur, la procédure appelée crée une copie de ce paramètre dans sa zone de données. La procédure appelante ignore donc les changements effectués sur ce paramètre par la procédure appelée.

Quand un paramètre est appelé par référence, la procédure transmet l'adresse de ce paramètre. La procédure appelée travaille ainsi directement sur le paramètre de l'appelant par le phénomène de l'indirection.



Technique de passage

Dans un premier temps, on se place dans le contexte suivant : un algorithme fait appel à une fonction récursive. Donc, tous les paramètres sont appelés par valeur. Le procédé est le suivant :

Définir la zone de données.

- Elle contient toutes les variables locales et les paramètres.

Définir les points d'appel et de retour.

- Il y a donc toujours un appel dans l'algorithme appelant. C'est le premier appel.

Au moment de l'appel on fait les opérations suivantes:

- empiler (sauvegarder) la zone de données courante.
- préparer le nouvel appel en préparant la nouvelle zone de données avec les paramètres (passage de paramètres) et avec l'adresse de retour (point d'appel).
- se brancher vers le début de la fonction.

Au moment du retour, on fait les opérations suivantes:

- récupérer l'adresse de retour qui se trouve dans la zone de données courante.
- dépiler une zone de donnée (restaurer la dernière sauvegardée)
- se brancher vers cette adresse.

Afin d'éviter un "Underflow", on initialise la pile avec une zone de données "bidon".



Transformation sur un exemple : la factorielle

Nous allons traiter ci-après l'exemple de la factorielle. Une première solution automatique informelle (contenant des ALLERA) est donnée. Ensuite, cette solution est améliorée. Enfin les "ALLERA" sont éliminées afin d'obtenir un algorithme structuré.

La fonction *Fact(N)* est définie comme suit (X et Y étant des variables locales) :

```
SI N = 0
    Fact := 1
SINON
    X := N-1;
    Y := Fact(X);
    Fact := N * Y
FSI
```

Que contient la zone de données ?

N, X, Y et l'adresse de retour.

Il y a deux points de retour :

- après l'affectation de Fact(X) à Y
- dans l'algorithme appelant.

Pour effectuer un retour on fera

SI I=1 ALLERA 1

SI I=2 ALLERA 2

Si I vaut 1, il y a retour à l'algorithme appelant; si I = 2 il y a retour à l'affectation de exécution précédente.

La zone de données est par conséquent du type :

```
TYPE Zdd =    STRUCTURE
               Param : ENTIER
               X,Y : ENTIER
               Adressderetour : ENTIER
FIN
```

On aura donc une pile de zones de données.

Une première solution

Soit Zddc : la zone de données courante. Une première traduction nous donne :

```
Créerpile(P)
{ Empiler une zone de données bidon}
Empiler(P,Zddc)
{ Initialiser Zddc }
Zddc.Par := N
Zddc.Adressederetour := 1
{ Début de la fonction simulée }
10 :   SI Zddc.Par = 0
      {Simulation de Fact = 1 }
      Fact := 1
      { Simulation du retour }
      I := Zddc.Adressederetour;
      Dépiler(P,Zddc)
      SI I=1 ALLERA 1
      SI I=2 ALLERA 2
      FSI
      Zddc.X = Zddc.Par - 1
      { Simulation de l'appel récursif }
      Empiler(P,Zddc)
      Zddc.Par := Zddc.X
      Zddc.Adressederetour := 2
      ALLERA 10
2:   { Point de retour de l'appel récursif }
      { Simulation de Y := Fact(X) }
      Zddc.Y := Fact
      Fact := Zddc.Par * Zddc.Y
      { Simulation du Retour }
      I:= Zddc.Adressderetour
      Dépiler(P,Zddc)
      SI I=1 ALLERA 1
      SI I=2 ALLERA 2
1:   {Retour à l'appelant}
```

Trace

Nous donnons, dans ce qui suit, l'évolution de la pile pour le calcul de factorielle 4.

L'environnement de trace est composé :

- d'une pile
- une zone de données courante
- une variable F qui contiendra le résultat de F(4)
- une variable I pour les adresses

Nous résumons dans la figure suivante l'évolution de la pile :



Amélioration

Faut-il utiliser toutes les variables temporaires ?

Il est clair qu'il ne faut mettre dans la zone de données que les informations utiles après le point d'appel.

Dans l'exemple de la factorielle, X et Y ne sont pas nécessaires dans la zone de données: Y n'est jamais définie avant l'appel et X n'est pas utilisée après le point de l'appel.

Comme il existe une seule adresse de retour, il n'est pas nécessaire de la mettre dans la zone de données.

On peut aussi éviter l'empilement de la zone de données "bidon" au début en remplaçant l'opération Dépiler (P, Zddc) par Dépiler (P, Zddc, Possible), c'est à dire, si la pile est vide alors Possible prend la valeur FAUX sinon récupérer la zone de données se trouvant au sommet de pile.

Avec toutes ces considérations, la zone de données se trouve réduite au seul paramètre N.

Notre algorithme précédent devient :

X et Y étant deux variables temporaires.

```
Créerpile(P)
    Zddc := N
10 : SI Zddc = 0
```

```

        Fact := 1
        { Retour }
        Dépiler(P,Zddc, Possible)
        SI NON Possible
            ALLERA 1
        SINON ALLERA 2
        FSI
    FSI
    X := Zddc - 1
    { Appel }
    Empiler(P,Zddc)
    Zddc := X
    ALLERA 10
2: Y := Fact
    Fact := Zddc * Y
    Dépiler(P, Zddc, Possible)
    SI NON Possible
        ALLERA 1
    SINON ALLERA 2
1:    { Fin de l'algorithme }

```

Les variables X et Y peuvent facilement être éliminées.

Enlevons de notre esprit l'idée de zone de données en remplaçant Zddc par X. Remplaçons aussi Fact par Y.

Nous obtenons l'algorithme suivant :

```

Créerpile(P)
    X := N
10 :    SI X = 0
        Y := 1
        { Retour }
        Dépiler(P,X, Possible)
        SI NON Possible ALLERA 1
        SINON ALLERA 2 FSI
    FSI
    { Appel }
    Empiler(P, X)
    X := X - 1
    ALLERA 10
2:    Y := X * Y
    Dépiler(P, X, Possible)
    SI NON Possible
        ALLERA 1
    SINON ALLERA 2
1:    { Fin de l'algorithme }

```

Élimination des "ALLERA"

Il s'agit maintenant de transformer l'algorithme afin d'éliminer les "ALLERA" et d'obtenir un algorithme structuré.

Dans l'algorithme ci-dessus, la séquence

```
Dépiler (P, X, Possible)
SI NON Possible
    ALLERA 1
SINON
    ALLERA 2
FSI
```

apparaît deux fois pour les cas $X = 0$ et $X \neq 0$. Elle peut figurer une seule fois en écrivant l'algorithme comme suit:

```
Creerpile (P)
    X:=N
10:    SI X=0
        Y := 1
    SINON
        Empiler(P,X)
        X:= X-1
        ALLERA 10
    FSI
2:    Dépiler(P, X, Possible)
    SI NON Possible
        ALLERA 1
    SINON
        Y := X * Y
        ALLERA 2
    FSI
1:    { Retour }
```

A ce niveau, on constate qu'il y a deux boucles indépendantes. L'algorithme peut alors être transformé comme suit :

```
Créerpile(P)
X := N
TANTQUE X <> 0 :
    Empiler(P, X)
    X := X-1
FINTANTQUE
Y:= 1
Dépiler(P, X, Possible)
TANTQUE Possible
    Y := X * Y
    Dépiler(P, X, Possible)
FINTANTQUE
```

Un simple examen de l'algorithme nous permet d'éliminer complètement la pile. En effet, la première boucle range dans la pile les N premiers entiers naturels et la seconde boucle permet de récupérer ses éléments. Il suffit donc de générer ses nombres par une boucle **"POUR"**.

L'algorithme devient :

```
Y := 1
POUR X = 1 , N :
    Y := Y * X
FINPOUR
```



Autres règles pour la transformation

D'autres règles sont nécessaires pour la dérécursification des algorithmes :



Appel en fin de procédure :

On peut toujours éliminer tout appel récursif qui apparaît en fin de procédure. Ca revient à empiler une zone de données et tout de suite après la dépiler. Il est donc inutile de l'empiler. Il suffit de faire :

- changer les valeurs de la zone de données par les nouveaux paramètres.
- se brancher au début de la procédure.



Utilisation des variables globales.

Si l'algorithme manipule un ou plusieurs tableaux, il est conseillé de les mettre comme variables globales afin d'éviter leur empilement à chaque appel..



Eviter les tableaux comme paramètres.



Cas des procédures

On a montré la transformation dans le cas des fonctions. S'il s'agit d'une procédure, il ne faut mettre dans la zone de données que les paramètres d'entrée. Les paramètres de sortie sont considérés comme des variables globales.



Récurtivité en Pascal

Le langage PASCAL admet la récursivité. Nous donnons ci-après les programmes PASCAL correspondants aux exemples d'introduction (cliquer sur lien pour afficher le programme)

◆ La factorielle

Donnons deux solutions : l'une avec variables locales explicites et l'autre sans variables explicites.

```
FUNCTION Fact (N : INTEGER) : INTEGER;
  VAR X, Y : INTEGER;
  BEGIN
    IF N = 0
    THEN Fact := 1
    ELSE
      BEGIN
        X := N - 1;
        Y := Fact (X) ;
        Fact := N * Y
      END
    END
  *****
  FUNCTION Fact ( N : INTEGER) : INTEGER;
  BEGIN
    IF N = 0
    THEN Fact := 1
    ELSE Fact := N * Fact ( N-1)
  END
```

◆ La multiplication

```
FUNCTION Mult(A,B : INTEGER) : INTEGER;
  BEGIN
    IF B = 1
    THEN Mult := A
    ELSE Mult := Mult(A, B-1) + A
  END
```

◆ Suite de Fibonacci

```
FUNCTION Fib (N :INTEGER) : INTEGER;
  VAR X,Y : INTEGER;
  BEGIN
    IF N <= 1
    THEN Fib := N
    ELSE
      BEGIN
        X := Fib(N-1);
        Y := FIN(N-2);
        Fib := X + Y
      END
    END
```


END
END

◆ Recherche binaire

On considère que A est un tableau global pour le module Recherche suivant :

```
FUNCTION Recherche (X:INTEGER;Bi, Bs : INTEGER): INTEGER;  
  VAR  
    Mil : INTEGER;  
  BEGIN  
    IF Bi > Bs  
    THEN Recherche := 0  
    ELSE  
      BEGIN  
        Mil := (Bi + Bs ) DIV 2 ;  
        IF X = A[Mil]  
        THEN Recherche := Mil  
        ELSE  
          IF X < A[Mil]  
          THEN Recherche := Recherche(X, BI, Mil-1)  
          ELSE Recherche:= Recherche (X,Mil+1,Bs);  
        END  
      END  
    END  
  END
```



TRAVAUX DIRIGES

1. Ecrire l'algorithme récursif qui calcule la somme des n premiers entiers naturels.
2. Ecrire les algorithmes récursifs correspondant au:
 - (a) quotient de a par b,
 - (b) reste de a et b,
 - (c) pgcd (plus grand diviseur commun) de deux entiers non négatifs.
3. Donner une définition récursive de $a + b$, où a et b sont des entiers non négatifs. Ecrire l'algorithme récursif correspondant.
4. Soit A un tableau d'entiers. Présenter des algorithmes récursifs pour
 - (a) le maximum de A,

- (b) le minimum de A,
- (c) la somme des éléments de A,
- (d) le produit des éléments de A,
- (e) la moyenne des éléments de A.

5. Développer les algorithmes itératifs et récursifs pour:

- (a) la factorielle,
- (b) le produit $a*b$,
- (c) la séquence de Fibonacci.

Evaluer (trace) les algorithmes pour les cas suivants : $6!$; $9!$; $100*3$; $6*4$; $\text{fib}(10)$; $\text{fib}(11)$.
Comparer les algorithmes itératifs et récursifs

6. Algorithmes récursifs sur les listes linéaires chaînées (llc):

- (a) inverser une llc,
- (b) rechercher un élément donné dans une llc.

7. Soit la définition récursive suivante :

FONCTION Fib (N:ENTIER):ENTIER

VAR X,Y : ENTIER

SI N <= 1

Fib := N

SINON

X := Fib(N-1)

Y := Fib(N-2)

Fib := X + Y

FSI

- (a) Donner une expression complètement parenthésée de $\text{fib}(6)$, puis l'évaluer.
- (b) Définir les points de retour de la procédure.
- (c) Que contient la zone de données ? Définir la structure.
- (d) Transformer l'algorithme récursif en utilisant la pile. L'algorithme obtenu est donc informel.
- (e) Faire une trace.
- (f) Peut-on réduire la zone de données ? Redéfinir alors la zone de données.

(g) Essayer de rendre l'algorithme formel. Cela revient à supprimer les "ALLERA". Peut-on se passer complètement de la pile ?

8. Refaire la même chose pour la fonction suivante :

FONCTION Comb (N, M : ENTIER) : ENTIER

{ Calcul DE Cnm POUR $0 \leq M \leq N$ ET $N \geq 1$ }

SI (N=1) OU (M=0) OU (M=N)

Comb <-- 1

SINON

Comb <-- Comb (N-1, M) + Comb(N-1, M-

1)

FSI

9. Transformer l'algorithme récursif correspondant à la tour de Hanoi.

10. Reprendre l'algorithme récursif qui inverse une liste linéaire chaînée de l'exercice 6. , puis utiliser une pile pour éliminer la récursivité.

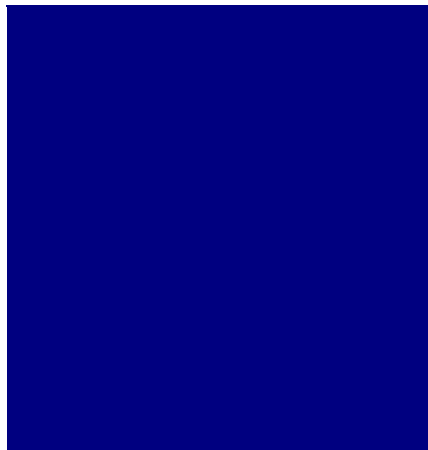
Définitions

C'est une structure de données hiérarchique, généralement dynamique.

Un arbre peut être considéré comme une liste chaînée non linéaire de maillons (ou noeuds). Ces derniers forment un graphe orienté où chaque noeud a au plus 1 prédécesseur et n successeurs ($n \geq 0$).

Exemple :

Le graphe orienté suivant représente un arbre.



En pratique, un arbre est représenté de haut en bas. Ce qui permet de ne pas orienter les sens des arcs. L'arbre précédent est désormais représenté comme suit :



Si le nombre de successeurs de tout noeud est au plus égal à 2, l'arbre est dit binaire. Si ce nombre est au plus égal à 3, l'arbre est dit ternaire. Et ainsi de suite. . .



Et d'une façon générale, *si le nombre de successeurs est au plus égal à n , l'arbre est dit arbre n -aire d'ordre n .*

Exemples :

- arbre généalogique
- table des matières d'un livre donné.

On peut aussi définir récursivement un arbre comme suit :

➤ un noeud unique est un arbre.

➤ si n est un noeud et T_1, T_2, \dots, T_n sont des arbres avec les racines R_1, R_2, \dots, R_n , alors on peut construire un nouvel arbre en faisant n le parent des noeuds R_1, R_2, \dots, R_n .



Terminologie

Tous les exemples qui suivent se rapportent à l'arbre suivant :



Sur les arbres on définit les termes suivants :

◆ Racine :	C'est le noeud qui n'a pas de prédécesseur. La racine constitue la caractéristique d'un arbre. Dans l'exemple c'est a
◆ Feuille :	c'est le noeud qui n'a pas de successeur. Une feuille est aussi appelée un noeud externe. Dans l'exemple, les feuilles sont : c, i, j, e, g et h.
◆ Noeud interne :	c'est tout noeud qui admet au moins un successeur. Dans l'exemple, les noeuds internes sont : a, b, f et d.
◆ Fils d'un noeud :	ce sont ses successeurs. Dans l'exemple, c, d et e sont les fils du noeud b.
◆ Frères :	ce sont les successeurs issus d'un même noeud. Dans l'exemple, c, d et e sont des frères.
◆ Père :	c'est un noeud qui admet au moins un successeur. Dans l'exemple, b est le père des noeuds c, d et e.
◆ Sous arbre :	C'est une portion de l'arbre. Dans l'exemple, le noeud f avec ces deux fils g et h constituent un sous arbre.
◆ Descendants d'un noeud :	ce sont tous les noeuds du sous arbre de racine noeud. Dans l'exemple, les descendants de h sont h, c, d, e, i et j.
◆ Ascendants d'un noeud :	ce sont tous les noeuds se trouvant sur la branche de la racine vers ce noeud. Dans l'exemple, les ascendants de d sont d, b et a. Les ascendants de g sont f et a.
◆ Branche :	Les ascendants d'une feuille constituent une branche de l'arbre. Dans l'exemple, les branches de l'arbre sont a-b-c, a-b-d-i, a-b-d-j, a-b-e, a-f-g et a-f-h.
◆ Degré d'un noeud :	C'est le nombre de ses fils. Dans l'exemple, le degré de b est 3.
◆ Niveau d'un noeud :	On dit que la racine a le niveau 0, ses fils ont le niveau 1, les fils des fils ont le niveau 2. etc... Dans l'exemple, a est de niveau 0, d est de niveau 2 et j est de niveau 3.
◆ Profondeur de l'arbre :	C'est le niveau maximal des feuilles de l'arbre. L'arbre de l'exemple est de profondeur 3.
◆ Forêt :	C'est un ensemble d'arbres.

Dans les chapitres qui vont suivre, on ne s'intéresse dans un premier temps qu'aux arbres binaires. Les arbres m-aires d'ordre n seront traités plus loin.



Les arbres binaires

Autres termes

En plus des termes définis dans la section précédente, on définit d'autres termes sur **les arbres binaires**.



En particulier, on parlera de *fils gauche* et *fils droit* , de **sous arbre gauche** et **sous arbre droit** d'un noeud donné de l'arbre.

Arbre strictement binaire

Un arbre est dit *strictement binaire* si chaque noeud qui n'est pas une feuille a exactement deux *fils* . Si un arbre strictement binaire a n feuilles, le nombre total de ses noeuds est $2n-1$. Le nombre de ses noeuds non feuilles (noeuds internes) est par conséquent $n-1$.

Arbre binaire complet

C'est un arbre *strictement binaire* où toutes les feuilles sont au même niveau. Dans un arbre binaire complet de profondeur d, le nombre de noeuds est

$$n = 2^0 + 2^1 + 2^2 + \dots + 2^d = 2^{d+1} - 1$$

Donc 2^d noeuds feuilles et $2^d - 1$ noeuds non feuilles.

Inversement, connaissant le nombre de noeuds d'un arbre binaire complet, on peut retrouver sa profondeur.

Il faut donc déterminer d tel que $n = 2^{d+1} - 1$

c'est à dire $n+1 = 2^{d+1}$

La valeur de d est donc $\log_2 (n+1) - 1$



Modèle sur les arbres binaires

Sur les arbres binaires, on définit le modèle (**machine abstraite**) suivant:

Un arbre est défini par son noeud racine.

Info(p)	permet d'accéder à l'information du noeud p
Fg(p)	permet d'accéder à l'information de fils gauche du noeud p
Fd(p)	permet d'accéder à l'information de fils droit du noeud p
Aff_info(p, x)	permet de modifier l'information du noeud p
Aff_fg(p, x)	permet de modifier l'information de fils gauche du noeud p
Aff_fd(p, x)	permet de modifier l'information de fils droit du noeud p
Creernoed(x)	crée un noeud avec x comme information et retourne la référence du noeud. Le noeud créé a Nil comme fils gauche et droit.
Liberernoed(p)	libère le noeud référencé par p.



Parcours des arbres binaires

Il n'y a pas un ordre naturel pour parcourir un arbre. Dans la littérature, une trentaine d'ordres de parcours ont été recensés. Cependant, Les trois ordres suivants sont les plus utilisés :

n désigne la racine d'un arbre binaire, **T1** et **T2** ses sous arbres gauche et droit.

Préordre : **n T1 T2**

Inordre : **T1 n T2**

Postordre : **T1 T2 n**

Exemple

Dans l'arbre ci-dessous,



Le parcours de l'arbre en préordre donne la liste : a, b, c, i, j, e, f, g, h.

Le parcours de l'arbre en inordre donne la liste : i, c, j, b, e, a, g, f, h

Le parcours de l'arbre en postordre donne la liste : i, j, c, e, b, g, h, f, a.



Arbre de recherche binaire

Définition

C'est une structure de données de base utilisée pour représenter des ensembles dont les éléments sont ordonnés par une relation d'ordre notée $<$ (au sens strict). Un **arbre de recherche binaire est un arbre dont les noeuds renferment les éléments d'un ensemble ordonné**. La propriété la plus importante est que tous les éléments stockés dans le sous arbre gauche d'un noeud contenant x sont strictement inférieurs à x, et tous les éléments rangés dans le sous arbre droit d'un noeud contenant x sont strictement supérieurs à x.



Une propriété importante d'un arbre de recherche binaire est la suivante : le parcours en inordre d'un arbre de recherche binaire donne la liste ordonnée de tous ses éléments.

Opérations

➤ Recherche

La recherche est **dichotomique**. A chaque étape, un sous arbre est éliminé. D'où l'appellation de l'arbre.

Le prédicat $R(X, A)$ qui suit recherche un élément X dans un arbre de recherche binaire de racine A .

```
SI  $A = NIL$  :  
     $R := FAUX$   
SINON  
     $SI$   $X = Info(A)$   
         $R := VRAI$   
    SINON  
         $SI$   $X < Info(A)$   
             $R := R(X, Fg(A))$   
        SINON  $R := R(X, Fd(A))$   
    FSI  
FSI  
FSI
```

➤ Insertion

L'insertion d'un élément se fait toujours au niveau d'une feuille ou à gauche (droite) d'un noeud ne possédant pas de fils gauche (droit). En d'autres termes, ***l'élément inséré est toujours une feuille.***

➤ Suppression

La suppression est plus délicate. Nous donnons dans ce qui suit les différents cas de suppression d'un élément et les actions à entreprendre pour chaque cas.

Considérons l'arbre suivant :



Soit n le pointeur de l'élément qu'on veut supprimer. Le tableau qui suit résume tous les cas.

\wedge désigne **Nil**

n		Action
fg	fd	
^	^	Remplacer n par ^
^	#^	Remplacer n par Fd(n)
#^	^	Remplacer n par Fg(n)
#^	#^	1. Rechercher le plus petit descendant du sous-arbre droit, soit p. 2. Remplacer Info(n) par Info(p) 3. Remplacer p par Fd(p)



Applications des arbres binaires

Un arbre binaire est utile quand une décision sur deux doit être prise à chaque étape d'un traitement.

Nous donnons, dans ce qui suit, quatre applications des arbres de recherche binaire.

Recherche de doubles dans une liste de nombres

On veut déterminer les doubles dans une suite de nombres se trouvant sur une machine-nombre où chaque ordre de lecture délivre le prochain nombre.

A un moment donné, pour savoir si un nombre venant d'être lu est double ou pas, il faut examiner tous les éléments déjà lus.

Essayons d'examiner où stocker les nombres déjà lus.

Cas d'un tableau

Si les éléments sont rajoutés à la fin, on recherchera séquentiellement. Si à un moment donné, le tableau contient n éléments, on en consultera en moyenne $n/2$ avant de retrouver l'élément.

Si on maintient le tableau ordonné, on recherchera dichotomiquement mais on constate une perte de temps considérable due aux décalages effectués causés par les insertions de données. Si à un moment donné, le tableau contient n éléments, on en décalera en moyenne $n/2$ afin de le maintenir ordonné.

Cas d'une liste linéaire chaînée

Que l'on maintienne la liste linéaire chaînée ordonnée ou pas, on procède séquentiellement pour rechercher un élément. Si à un moment donné, la liste linéaire chaînée contient n éléments, on en consultera en moyenne $n/2$ avant de retrouver l'élément.

Cas d'un arbre de recherche binaire

On insère les éléments dans leur ordre d'arrivée dans un arbre de recherche binaire. Si à un moment donné, ce dernier contient n éléments, on en consultera en moyenne $\lceil \log_2(n) \rceil / 2$ avant de retrouver l'élément. Noter aussi que l'insertion est très rapide puisqu'elle permet la modification d'un seul pointeur.

Nous venons de voir ainsi que l'utilisation d'un arbre de recherche binaire pour cette application simple permet de gagner considérablement du temps.

L'algorithme qui suit recherche l'élément *Nombre* dans l'arbre de recherche binaire de racine *Arbre*. Si *Nombre* n'est pas trouvée, il y est inséré.

Nombre est du type entier, *P* et *Q* sont des variables permettant de référencer les noeuds.

```
LIRE(Nombre)
Arbre := Creernoead(Nombre)
POUR I = 2 , N :
    LIRE (Nombre)
    P := Arbre ; Q := Arbre
    TANTQUE Nombre <> Info(P) ET Q <> NIL :
        P := Q
        SI Nombre < Info(P)
            Q := Fg(P)
        SINON Q := Fd(P) FSI
    FINTANTQUE
    SI Nombre = Info(P)
        " C'est un double"
    SINON
        SI Nombre < Info(P)
            Aff_Fg(P, Creernoead(Nombre))
        SINON Aff_Fd(P, Creernoead(Nombre))
        FSI
    FSI
FINPOUR
```

Représentation des expressions arithmétiques

Une expression arithmétique peut être représentée par un arbre strictement binaire comme suit :



-

Les opérandes sont au niveau des feuilles. les noeuds non feuilles sont des opérateurs.

Le parcours en préordre donne la forme polonaise préfixée, le postordre la forme postfixée et l'inordre la forme infixée (forme normale sans parenthèses)

Algorithme récursif d'évaluation

Nous utilisons le prédicat Opérande(T), égal à vrai si Info(T) est un opérande, égal à faux sinon.



Remarque :

➤ Opérande(T) peut être implémenté comme un champ additionnel.

➤ le champ info est donc sous forme canonique.

Soit A un arbre représentant une expression arithmétique. L'algorithme utilise le parcours postordre et est le suivant :

Eval(A) :

```
SI Opérande(A) :  
    Eval := Info(A)  
    SINON  
        Eval := Oper(Info(A),  
Eval(Fg(A), Eval(Fd(A))  
FSI
```

avec Oper(Op, A, B) une fonction qui effectue l'opération Op entre deux valeurs A et B.

Représentation des listes par des arbres binaires

Plusieurs opérations sont possibles sur les listes :

- ajouter un élément à la tête ou à la queue d'une liste,

- supprimer le premier ou le dernier élément d'une liste,
- retrouver le k-ième élément ou le dernier d'une liste,
- insérer un élément avant ou après un élément donné,
- supprimer le prédécesseur ou le successeur d'un élément.

Les algorithmes dépendent de la représentation (structure de données) choisie.

On peut représenter une liste linéaire chaînée par un arbre binaire de la façon suivante : les éléments de la liste sont au niveau des feuilles. Chaque noeud qui n'est pas une feuille contient le nombre de feuilles de son sous arbre gauche.

Exemple

La liste a, b, c, d, e est représentée comme suit :



La représentation en arbre binaire d'une liste répond avec efficacité au problème de la recherche du K-ième élément et à la suppression d'un élément donné.

Algorithme de recherche du K-ième élément

L'algorithme qui suit recherche le K-ième élément d'une liste représentée dans un arbre de racine Arbre :

```

R := K
P := Arbre
TANTQUE NON Feuille(P) :
    SI R <= Compte(P) :
        P := Fg(P)
    SINON
        R := R - Compte(P)
        P := Fd(P)

```

Feuille(P) est un prédicat égal à VRAI si P est une feuille, FAUX sinon.

Code de Huffman

Soient les caractères a, b, c, d, e avec les probabilités .12, .4, .15, .08, .25 respectivement. Nous voulons coder chaque caractère en une séquence de 0 et de 1 de telle sorte qu'aucun code n'est le préfixe d'un autre code. Grâce à cette propriété dite *propriété du préfixe*, on peut décoder une suite de bits en répétant des suppressions de chaînes.

Exemples : 2 codes possibles :

Caractères	Probabilités	Code1	Code2
------------	--------------	-------	-------

a	.12	000 000	
---	-----	---------	--

b	.40	001 11	
---	-----	--------	--

c	.15	010 01	
---	-----	--------	--

d	.08	011 001	
---	-----	---------	--

e	.25	100 10	
---	-----	--------	--

Le code 1 a *la propriété du préfixe* : aucune séquence de trois bits ne peut être le préfixe d'un autre code.

Le code 2 a aussi la propriété du préfixe. la séquence 1101001 est décodée en bcd.

Soit un ensemble de caractères et leurs probabilités d'apparition. Il s'agit de déterminer un code avec la propriété du préfixe telle que la longueur moyenne d'un code pour un caractère est minimale.

Le code 1 a une moyenne de 3. Le code 2 a une moyenne de 2.2. Peut on faire mieux ? c'est *l'algorithme de Huffman*.

Technique

1. Sélectionner deux caractères a et b ayant les plus faibles probabilités.

2. Remplacer les par un caractère fictif x tel que

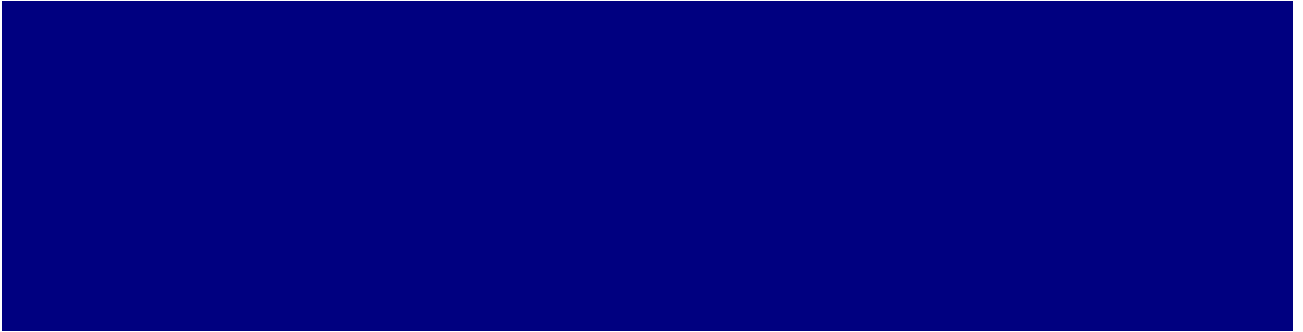
$$p(x) = p(a) + p(b)$$

3. Répéter 1. et 2. récursivement.

Un code ayant la propriété du préfixe peut être représenté par un arbre binaire. Les branches étiquetées par 0 ou 1 représentent les codes des caractères. Les caractères sont au niveau des feuilles.

La propriété du préfixe garantit qu'aucun caractère ne peut avoir un code qui est un nœud interne. Inversement, si on arrive à étiqueter les feuilles d'un arbre binaire, on obtient un code avec la propriété du préfixe.

Comme exemple, les arbres associés aux codes précédents sont les suivants :



Scénario de l'algorithme de Huffman sur un exemple

Nous donnons, dans ce qui suit, le scénario de l'algorithme de Huffman à partir d'un exemple.



Nous avons ainsi construit un arbre à partir d'une forêt. C'est ce qu'on appelle *l'arbre de Huffman*. Le code obtenu a une longueur moyenne de 2.17 et est le suivant :

a 1111

b 0

c 110

d 1110

e 10



Implémentation des arbres binaires

Nous avons dit au début de ce chapitre que les arbres sont généralement des structures de données dynamiques. Nous allons examiner, dans ce qui suit, comment on représente les arbres en dynamique et même en statique.

En dynamique

L'arbre binaire est un ensemble de noeuds(maillons) alloués dynamiquement. La structure d'un noeud de l'arbre est la suivante :

```
TYPE Typenoeud =          STRUCTURE
    Info :                 Typeqq
    Fg, Fd :               POINTEUR ( Typenoeud)
FIN
```

VAR Arbre : POINTEUR(Typenoeud)

En statique

Représentation standard

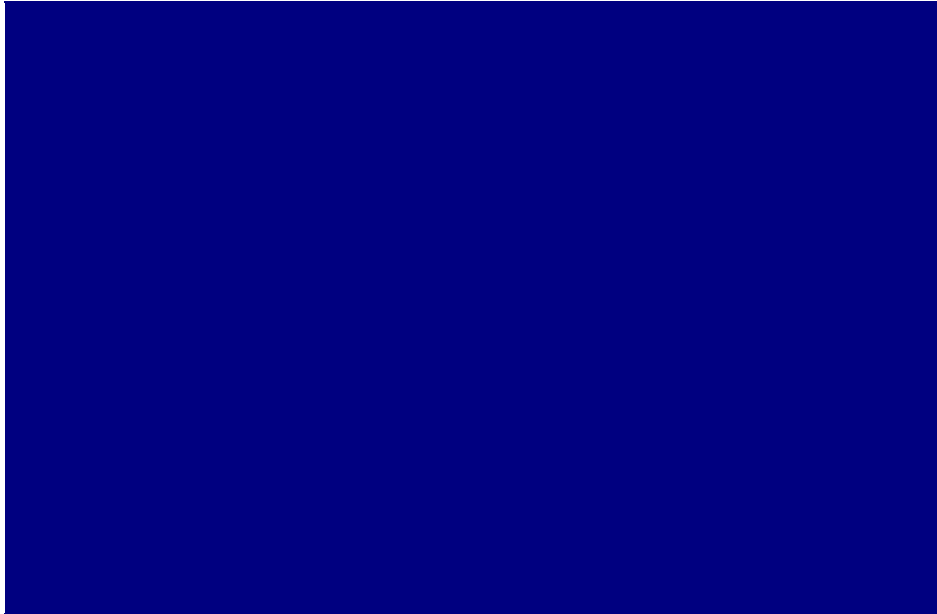
On range l'arbre dans un tableau. Chaque élément du tableau possède trois champs: un pour l'information, un pour le fils gauche et un pour le fils droit.

```
TYPE Typenoeud=          STRUCTURE
    Info :                 Typeqq
    Gauche, Droite :       ENTIER
FIN
```

VAR Arbre = TABLEAU(. 1..N.) DE Typenoeud

Dans cette représentation la racine est toujours à la position 1 du tableau.

Exemple :



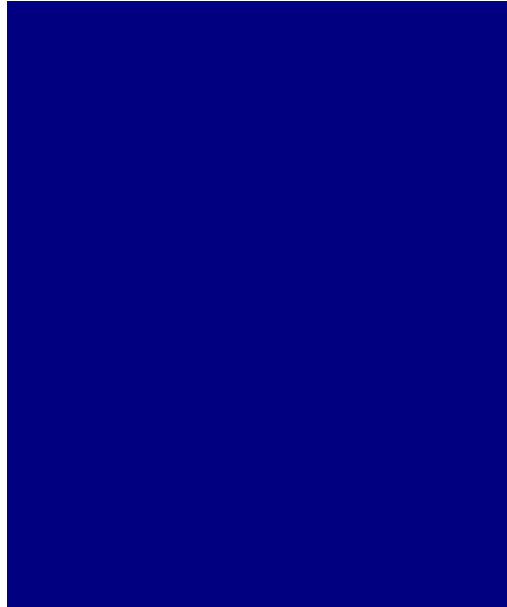
Si on veut que la racine soit n'importe où dans le tableau, alors l'arbre sera défini par un couple : tableau et indice de la racine.

On aura donc la description suivante :

```
TYPE Typearbre =          STRUCTURE
  T :                      TABLEAU[1..M] DE Typenoeud
  Racine :                  ENTIER
FIN
```

VAR Arbre : Typearbre.

Exemple :



Dans les deux représentations, on a un seul arbre par tableau.

Si on veut maintenant ranger plusieurs arbres dans le même tableau, il suffit de considérer un tableau de type `Typenoeud` et définir un arbre comme suit : *VAR Arbre : ENTIER*

Représentation séquentielle

L'idée dans la représentation séquentielle c'est *de ne pas représenter les indices des fils*. On convient donc de ranger les éléments de l'arbre selon un ordre prédéfini.

Une représentation séquentielle pourrait être la suivante:

La racine est à la position 1. Le noeud à la position p est le père implicite des noeuds $2p$ (fils gauche) et $2p+1$ (fils droit). Gauche(p) est le noeud $2p$, Droit(p) est le noeud $2p+1$. Si un fils gauche est à la position p , son frère droit est à la position $p+1$; Si un fils droit est à la position p son frère gauche est la position $p-1$. Père(p) c'est le noeud $p \text{ DIV } 2$. Etc.



Les arbres m-aires

Parcours des arbres m-aires

De même que sur les arbres binaires, on peut définir les types de parcours suivants sur les arbres m-aires.

n étant la racine de l'arbre et T_1, T_2, \dots, T_p ses sous arbres de gauche à droite.

Préordre :	n T1 T2 ... Tp
Inordre :	T1 n T2 T3 ...Tp
Postordre :	T1 T2 T3 ... Tp n

Exemple :

Soit l'arbre m-aire suivant d'ordre 4 (arbre quaternaire):



Le parcours de l'arbre en préordre donne la liste : a, b, e, f, h, i, j, k, c, d, e.

Le parcours de l'arbre en inordre donne la liste : e, b, h, f, i, j, k, a, c, g, d.

Le parcours de l'arbre en postordre donne la liste : e, h, i, j, k, f, b, c, g, d, a.

Implémentation

En contigu

L'arbre est représenté dans un tableau. Nous donnons, dans ce qui suit, quelques représentations.



Une première représentation est la suivante :

chaque élément du tableau représente un noeud de l'arbre qui est composé d'un champ information et d'un champ tableau qui contient les indices des fils du noeud.

Ce qui donne la description suivante :

TYPE Typenoeud = **STRUCTURE**

Info :	Typeqq
Fils :	TABLEAU(1..N) DE ENTIER
FIN	

VAR Arbre : TABLEAU(1..M.) DE Typenoeud

La racine se trouve en première position du tableau.



Une deuxième représentation est la suivante :

au lieu de représenter tous les indices des fils au niveau d'un noeud, on convient de ne représenter que deux indices : l'indice du fils le plus à gauche et l'indice de son frère immédiatement à droite. C'est donc une représentation en arbre binaire.

On a ainsi la description :

TYPE Typenoeud	=	STRUCTURE
Info :		Typeqq
Fils , Prochain :		ENTIER
FIN		

VAR Arbre : TABLEAU(1..M.) DE Typenoeud

La racine se trouve en première position du tableau.



Une troisième représentation pourrait être la suivante :

on ne représente pas les indices des fils. Les relations de parenté entre les noeuds sont définies comme suit :

$A(i) = j$ si j est le parent de i

$A(i) = 0$ si i est la racine.

En dynamique

L'arbre est une liste chaînée de maillons alloués dynamiquement.



Une première représentation est la suivante :

chaque noeud de l'arbre a la structure suivante :

```

TYPE Typenoeud =          STRUCTURE
    Info :                Typeqq
    Fils :                TABLEAU(1..N) DE POINTEUR(Typenoeud)
FIN

```

VAR Arbre : POINTEUR(Typenoeud)



Une deuxième représentation peut être la suivante :

au niveau de chaque noeud, on ne représente que l'adresse du fils le plus à gauche et celle du frère immédiatement à droite.

Chaque noeud a donc la structure suivante :

```

TYPE Typenoeud  =          STRUCTURE
    Info :          Typeqq
    Fils, Frere :    POINTEUR(Typenoeud)
FIN

```

VAR Arbre = POINTEUR(Typenoeud)

Grâce à cette nouvelle représentation, nous verrons par la suite que l'on peut transformer toute *arbre m-aire* en *arbre binaire*. Et d'une façon générale, toute *forêt en un arbre binaire*.



Donnons une troisième représentation :

une table principale contient tous les noeuds de l'arbre. chaque élément de cette table pointe vers une liste linéaire chaînée contenant les fils du noeud.

Chaque élément de cette liste a la structure :

```

TYPE Typenoeud =          STRUCTURE
    Info :                Typeqq
    Suiv :                POINTEUR(Typenoeud)
FIN

```

Chaque élément de la table est du type :

```

TYPE Typeelement =        STRUCTURE
    Info :                Typeqq
    Liste:                POINTEUR(Typenoeud)

```

FIN

VAR Arbre : TABLEAU(1..M) DE Typeelement.

Transformation d'un arbre m-aire en un arbre binaire.

Nous illustrons, dans ce qui suit, à partir d'un exemple, les différentes étapes permettant de transformer un arbre m-aire en un arbre binaire.

Soit l'arbre m-aire suivant :



◆ Première étape

Détachons tous les fils qui ne sont pas les plus à gauche et lions les de telle sorte que les frères soient dans une liste linéaire chaînée comme le montre la figure suivante:



◆ Deuxième étape

Faisons une rotation des listes linéaires chaînées de 45° dans le sens des aiguilles d'une montre. Nous obtenons l'arbre binaire suivant :



Exemple :



L'arbre est défini comme suit :

```
TYPE      Typenoeud = STRUCTURE
          Info : Typeqq
          Vide : BOOLEEN
FIN
VAR Arbre: Tableau[1..M] DE
Typenoeud
```

Les opérations Fg, Fd, Info, Affinfo sont définies comme suit :

```
Fg(P) :      Fg := 2p
Fd(P) :      Fd := 2p+1
Info(P) :    Info := Arbre(P).Info
Aff_info(P, Val) : Arbre(P).Info := Val
```




Les opérations Aff-Fg et aff-Fd n'ont pas de sens pour cette représentation.

La fonction Allouer est remplacée par les deux fonctions **Creerfg** et **Creerfd** suivantes :

Creerfg(P):

```

SI 2p > M :
    Creerfg := - 1
SINON
    SI NON Arbre(2p).Vide
        Creerfg := - 1
    SINON
        Creerfg := 2p
        Arbre(2p).Vide := FAUX
    FSI
FSI

```



La fonction **Créerfd** est définie de manière analogue. Il suffit de remplacer dans la fonction précédente **Créerfg** par **Créerfd** et **2p** par **2p + 1**.



La fonction Libérer est définie comme suit : ***Libérer(P) : Arbre(P).Vide := VRAI***

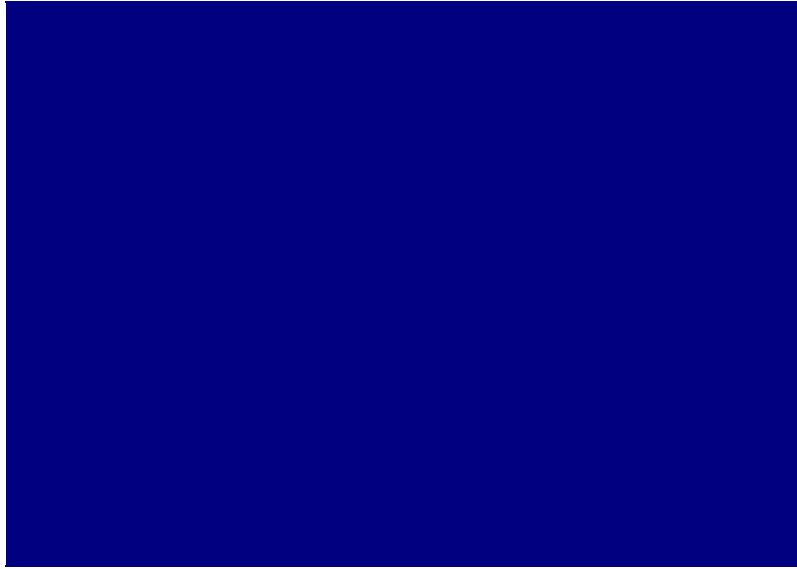
Ce qui suppose donc que le tableau Arbre doit être initialisé à VRAI sur son champ Vide.

Transformation d'une forêt en un arbre binaire.

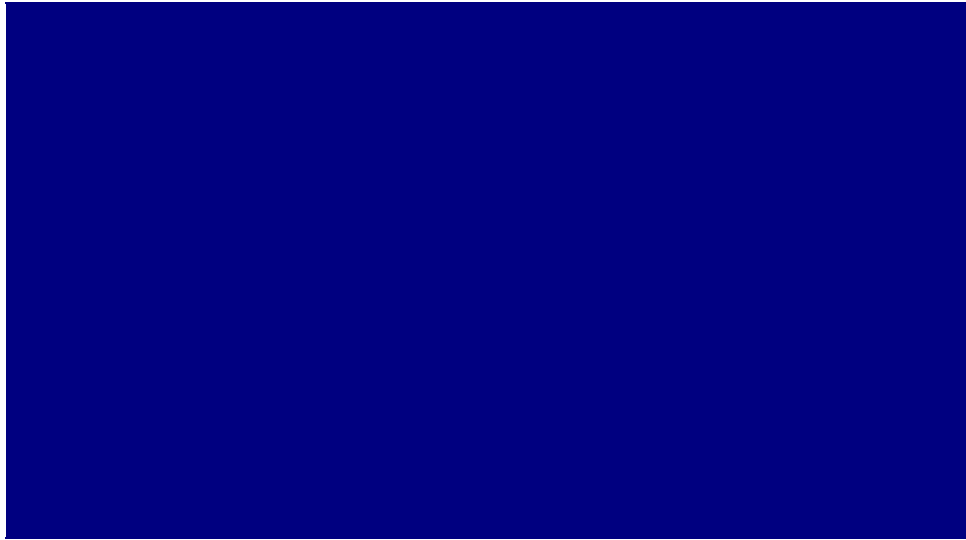
Dans la transformation précédente, la racine de l'arbre résultant n'a pas de fils droit. On exploite ce champ en le faisant pointer sur la racine du prochain arbre transformé de la forêt. De même, le fils droit de la racine de l'arbre résultant de cette opération n'a pas de sous arbre droit dans son noeud le plus à droite. Il pointera alors le troisième arbre transformé. Et ainsi de suite...

Transformation d'un arbre binaire en une forêt.

Inversement, on peut transformer tout arbre binaire en une forêt. L'arbre binaire suivant :



se transforme en la forêt suivante :



TRAVAUX DIRIGES

1. Ecrire des algorithmes récursifs et non récursifs pour déterminer dans un arbre binaire:

- (a) le nombre de noeuds,
- (b) le nombre de feuilles,
- (c) la somme des contenus de tous les noeuds,
- (d) la profondeur.

2. Ecrire un algorithme qui détermine si un arbre binaire est :

(a) strictement binaire,

(b) complet.

3. Développer un algorithme pour trouver les doubles dans une suite de n nombres.

(a) avec une liste linéaire chaînée,

(b) avec un arbre de recherche binaire.

Dénombrer les comparaisons dans chaque cas.

4. Trouver les algorithmes récursifs de parcours d'un arbre binaire.

6. Trouver les algorithmes de recherche, d'insertion et de suppression dans un arbre de recherche binaire.

7. Etude du parcours Inordre

7.1 Utiliser le modèle défini en cours pour écrire l'algorithme récursif de parcours *inordre* dans un arbre binaire.

7.2 Trouver l'algorithme équivalent non récursif.

7.3 Un arbre binaire enfilé est défini de telle sorte qu'un noeud, au lieu de contenir un pointeur NIL dans son champ droit, il contient un pointeur vers son successeur inordre (arbre binaire enfilé à droite). Pour implémenter ces sortes d'arbres, il suffit de rajouter un booléen au niveau du noeud pour dire si celui-ci est enfilé ou non.

Montrer qu'avec ces sortes d'arbres, on peut éliminer la pile dans l'algorithme non récursif de parcours inordre. Il s'agit donc d'écrire l'algorithme.

7.4 En considérant en plus l'opération Père(p) dans le modèle (qui délivre le père du noeud référencé par p), on peut développer l'algorithme de parcours inordre sans utiliser la pile. Ecrire l'algorithme correspondant.

8. Représentation d'une liste linéaire chaînée par un arbre

On peut représenter une liste linéaire chaînée par un arbre binaire de la façon suivante : Les éléments de la liste sont au niveau des feuilles. Chaque noeud qui n'est pas une feuille contient le nombre de feuilles de son sous arbre gauche.

8.1 Trouver le K-ième élément de la liste représentée ainsi.

8.2 Etudier et développer l'algorithme de construction.

8.3 Etudier et développer l'algorithme de suppression.

9 Implémentation des arbres

9.1 Implémenter le modèle défini sur les arbres au moyen des représentations définies en cours :

(a) représentation contigue

(b) représentation contigue séquentielle

9.2 Un arbre binaire peut être représenté comme suit:

Si A est un tableau à n éléments : $A(i)=j$ si j est le parent du noeud i. $A(i)=0$ si i est la racine. C'est ce qu'on appelle la représentation des parents. Traduire le modèle.

9.3 Représentation des arbres au moyen des listes de fils.

On a un tableau T de listes, indexé par des noeuds que nous supposons numérotés de 1 à n. Chaque liste pointée par Tab (i) contient les fils du noeud i.

La description peut être la suivante :

```
TYPE S = STRUCTURE           { Maillon de la
liste }
```

```
    Val : ENTIER
```

```
    ADR : POINTEUR(S)
```

```
FIN
```

```
TYPE List = POINTEUR(S)      { Ensemble des
listes }
```

```
TYPE Typearbre = STRUCTURE   { Ensemble des
arbres }
```

```
    Tete : TABLEAU(1..Max) DE List
```

```
    Racine: ENTIER
```

```
FIN
```

```
VAR Arbre : Typearbre { Un arbre }
```

Implémenter le modèle. (Structures de données et algorithmes)

10. Algorithme de Huffman

Supposons que nous avons des messages composés de séquences de caractères. Dans chaque message, les caractères sont indépendants et apparaissent avec des probabilités connues. Nous voulons coder chaque caractère en une séquence de 0 et de 1 de telle sorte qu'aucun code n'est le préfixe d'un autre code. Grâce à cette propriété dite du préfixe on peut décoder une chaîne de 0 et de 1. Nous voulons écrire l'algorithme de Huffman permettant de réaliser ce code. Pour le faire on utilise les structures de données suivantes :

1) un tableau ARBRE où chaque élément est du type T1 défini comme suit :

```
TYPE T1 = Structure
```

Fg, Fd, Pere : ENTIER
FIN

pour représenter les arbres binaires.

Le champ Père permet de retrouver les chemins des feuilles vers la racine, aussi nous pouvons trouver le code pour chaque caractère.

2) un tableau ALPHABET d'éléments du type T2 défini comme suit :

```
TYPE T2 = STRUCTURE
    Symbole : CAR
    Probabilité : REEL
    Feuille : ENTIER      { Indice vers
l'arbre }
FIN
```

pour associer à chaque symbole de l'alphabet à coder avec sa feuille correspondante. Ce tableau enregistre aussi la probabilité pour chaque caractère.

3) un tableau FORET d'éléments qui représente les arbres. Chaque élément est alors du type T3 défini comme suit:

```
TYPE T3 = STRUCTURE
    Poids : REEL
    Racine : ENTIER
FIN
```

10.1 Comment initialiser les tableaux FORET, ALPHABET et ARBRE.

10.2 Donner le pseudo algorithme en faisant ressortir les modules nécessaires.

10.3 Développer ces modules.

10.4 Donner l'algorithme de Huffman.

10.5 Application : Ecrire l'algorithme qui permet d'archiver un programme écrit en PASCAL, puis l'algorithme qui permet de le désarchiver en utilisant le code de Huffman.

11. les arbres m-aires

11.1 Rappeler le modèle défini sur les arbres m-aires. Ecrire les algorithmes de parcours.

11.2 Implémenter le modèle des arbres m-aires avec les représentations définies en cours. Peut on utiliser les représentations définies dans les arbres binaires ? Traduire le modèle dans chaque représentation possible.

11.3 Transformer des arbres m-aires en arbres binaires.

11.4 Transformer des forêts en arbres binaires.

Introduction, définitions

Supposons que l'on veuille ranger des données qui arrivent dans un ordre quelconque dans un tableau.

Il existe, en général, deux façons de les ranger :

➤ ou bien on veut garder le tableau ordonné:

l'insertion d'une donnée provoque alors des décalages.

Par contre, la recherche est rapide puisqu'elle est dichotomique.

➤ ou bien on ne veut pas garder le tableau ordonné:

l'insertion d'une donnée est donc rapide puisqu'elle se fait à la fin.

Par contre, si on veut rechercher un élément, on est obligé de procéder séquentiellement.

Nous constatons que pour les deux types de rangement, l'avantage de l'un fait l'inconvénient de l'autre. C'est à dire, pour rechercher rapidement ($O(\log(n))$), il faut ranger lentement. Et si on veut ranger rapidement, il faudrait rechercher lentement ($O(n)$).

➡ Une troisième possibilité de ranger une donnée dans un tableau :

Affecter un emplacement calculé par une fonction f .

$f(K)$ est alors l'emplacement dans le tableau où sera rangée la donnée K .

Dans ce type de rangement, que ce soit pour insérer ou rechercher une donnée, on procédera toujours aussi rapidement ($O(1)$).

Le problème avec ce nouveau type de rangement est la détermination d'une fonction bijective,

c'est à dire une fonction qui attribue pour chaque donnée à insérer un nouvel emplacement dans le tableau.

Il n'est pas facile de découvrir une telle fonction f . Nous allons montrer ceci à partir d'exemples.

Il y a $41! = 41 \times 40 \times \dots \times 1 = 41 \times 40 \times \dots \times 1$ fonctions possibles d'un ensemble de 41 éléments dans un ensemble de 41 éléments.

Et seulement $41 \times 40 \times \dots \times 11 = 41! / 10! = 41 \times 40 \times \dots \times 11$ de ces fonctions ont des valeurs distinctes pour chaque argument.

Environ une fonction sur 10 millions est convenable.

On peut aussi montrer que:

si on a 4000 données et 5000 adresses possibles il existe 10120000 fonctions possibles et seulement une est parfaite!

A ce propos, le fameux "paradoxe de l'anniversaire" cité dans le livre de *D.E KNUTH*, affirme que:

Si 23 personnes ou plus sont présentes dans une salle, les chances sont bonnes pour que deux d'entre elles aient le même jour et le même mois de naissance

En d'autres termes, si nous sélectionnons une fonction aléatoire parmi les fonctions de 23 clés dans l'ensemble

des 365 jours, la probabilité pour qu'il n'existe pas 2 données avec la même image est seulement de 0.4927 (inférieure à 1/2).

Pour une taille de table moyenne, une fonction convenable peut être trouvée après une journée entière de travail.

Un véritable puzzle ! Et même si on arrive à trouver une telle fonction, le problème réapparaîtra lors des insertions ultérieures.



Cette classe d'algorithmes est appelée *hachage* (en anglais "*Hashing*") ou technique de rangement dispersé.



Le paradoxe cité ci-dessus nous affirme qu'il existe toujours des données distinctes K1 et K2 pour lesquelles $f(K1) = f(K2)$.

Une telle situation est appelée *collision*.

En conclusion, pour utiliser une table à rangement dispersé ou une technique de hachage l'utilisateur doit définir :

- une fonction de hachage
- une méthode dite méthode de résolution des collisions.



Terminologie

Les données qui ont la même image par la fonction de hachage sont appelées *synonymes*.

L'adresse primaire d'une donnée c'est l'adresse *f(donnée)*.

Toute donnée qui n'est pas dans son adresse primaire est appelée *débordement*.

On dit aussi qu'elle est rangée dans *une adresse secondaire*.



Fonctions de hachage

Il s'agit de trouver une fonction f telle que :

$$0 \leq f(K) < M$$

qui réduit au maximum le nombre de collisions.

L'idéal, bien sûr, c'est d'avoir f bijective. Le pire des cas, c'est lorsque toute donnée est hachée en une même adresse.

Une solution acceptable est une solution où certaines données partagent la même adresse(f est surjective).

⇒ *Un algorithme simple de hachage*

- Représenter la donnée sous forme numérique.
- Concaténer et ajouter (Folk and Add).
- Diviser par un nombre premier et utiliser le reste comme adresse.

Exemple : transformation de "**LOWELL**" par cet algorithme.

a)	76 79 87 69 76 76 32 32 32 32 32
b)	
	$7679 + 8769 = 16448 \text{ Mod } 20000 = 16448$
	$16448 + 7676 = 24124 \text{ Mod } 20000 = 4124$
	$4124 + 3232 = 7356 \text{ Mod } 20000 = 7356$
	$7356 + 3232 = 10588 \text{ Mod } 20000 = 10588$
	$10588 + 3232 = 13820 \text{ Mod } 20000 = 13820$
c)	$a = 13820 \text{ mod } 101 = 84$ C'est l'adresse ou sera logée la donnée.



Mod 20000 est utilisé afin qu'*il n'y ait pas de débordement*.

Méthode de division

$$h(K) = K \text{ mod } M$$

Il faut bien choisir M . On démontre que c'est:

➤ *très mauvais de choisir M une puissance de 2.*

➤ *M premier constitue généralement un bon choix.*

Méthode dite du milieu du carré ("middle square")

On élève la donnée au carré et on prend les chiffres du milieu.

Exemple:

$(453)^2 = 205209$
$h(453) = 52$

Cette méthode donne de bons résultats si le nombre au carré n'a pas de zéros.

Méthode dite "transformation radix".

On convertit la donnée dans une certaine base de numération et on prend le reste de la division de la donnée

transformée par la taille du tableau.

Exemple :

$453 = (382)_{11}$ (en base 11)
$382 \bmod 99 = 85$
$h(453) = 85$



Méthodes de résolution des collisions

Il existe plusieurs façons de résoudre les collisions. Dans les paragraphes qui suivent, nous présenterons

quelques unes de ces méthodes. Les trois premiers algorithmes correspondants sont présentés tels qu'ils l'ont été

dans le livre de *D.E KNUTH, "The art of computer programming, Vol 3"*. Nous avons aussi jugé utile de rajouter

une version de l'algorithme du chaînage séparé avec le même formalisme.

1. [Essai linéaire](#)
2. [Double hachage](#)
3. [Chaînage interne](#)
4. [Chaînage séparé](#)



Essai linéaire

S'il se produit une collision sur la case k du tableau $T[0..M-1]$, on insère la donnée, si elle n'existe pas, dans

la première case libre fournie par la séquence cyclique : $h(k)-1, \dots, 0, M-1, M-2, \dots, h(k)+1$

En d'autres termes, on essaie de chercher linéairement une case libre dans la séquence ci-dessus.

D'où l'appellation *de la méthode*.

Exemple:

L'insertion des données suivantes avec leurs transformées (entre parenthèses) : **a(3), b(2), c(3), d(2), e(1)**

dans une table T de 6 éléments donne :

0	d
1	c
2	b
3	a
4	
5	e

Table T

- L'insertion de a se fait en position 3.
- L'insertion de b se fait en position 2.
- L'insertion de c provoque une collision sur la case 3. On recherche la première case libre dans la séquence 2, 1, ... C'est la case 1. Donc c sera rangée en 1.
- L'insertion de d provoque aussi une collision. Elle est donc rangée en position 0.
- De même, e sera rangée en 5.



Algorithme:

Cet algorithme recherche une donnée K dans une table de M éléments. Si K n'est pas trouvée et la table n'est

pas pleine, alors K est insérée.

Les éléments de la table sont dénotés
 $T(i)$, $0 \leq i < M$.

Un élément de la table peut être vide ou occupé.

Un élément occupé contient un champ $\text{Donnée}(i)$

Une variable auxiliaire N est utilisée. Elle contient le nombre d'éléments occupés. A chaque insertion d'une donnée, N est incrémentée.

L1. [Hacher]

$i := h(K) \quad \{ 0 \leq i < M \}$

L2. [Comparer]

SI $\text{Donnée}(K) = K$, l'algorithme se termine avec succès.

Autrement SI $T(i)$ est vide aller à L4.

L3. [Avancer au prochain]

$i := i - 1$

SI $i < 0$

$i := i + M$

ALLER A L2.

L4. [Insérer] {la recherche est sans succès}

SI $N = M - 1$

l'algorithme se termine avec débordement

SINON

$N := N + 1$

Marquer $T(i)$ occupé

$\text{Donnée}(i) := K$

Remarquer que la table est remplie quand $N = M - 1$ et non quand $N = M$. On dit que l'on sacrifie un élément du tableau pour arrêter le processus de recherche puisque le critère d'arrêt est toujours le vide.



Double hachage

Cette méthode est presque analogue à la précédente mais au lieu que la séquence soit linéaire, elle est

construite par *une autre fonction de hachage*. D'où l'appellation *de la méthode*.

Autrement dit, s'il y a collision sur une case k , on calcule un pas p par une autre fonction de hachage

et la séquence cyclique à consulter serait $h(k)-p$, $h(k)-2p$,

Les valeurs sont calculées *modulo M*, la taille de la table.

Deux fonctions de hachage sont alors utilisées $h(K)$ et $h'(K)$. Le choix de M est très important.

➡ Un mauvais choix peut entraîner une non couverture de l'ensemble des adresses possibles.

On démontre que lorsque M est *un nombre premier* et la fonction de hachage est *aléatoire*, il y a *couverture*

de l'ensemble des adresses.

Exemple:

L'insertion des données suivantes avec leurs transformées (entre parenthèses) : **a(3), b(2), c(3), d(2), e(1)**

et en plus avec $h'(c) = 3$; $h'(d) = 1$; $h'(e) = 3$ (h' étant la deuxième fonction de hachage) dans une table T de 6 éléments donne :

0	c
1	d
2	b
3	a
4	e
5	

Table T

- a est insérée à la position 3 de T.
- b est insérée en position 2.
- c provoque une collision sur la case 3. Comme $h'(c) = 3$, la donnée c est par conséquent rangée en position 0.
- d provoque une collision sur la case 2. Comme $h'(d) = 1$, la donnée d est par conséquent rangée en position 1.
- e provoque une collision sur la case 1. Comme $h'(e) = 3$, la donnée c est par conséquent rangée en position 4.



Algorithme

Cet algorithme recherche une donnée K dans une table de M éléments. Si K n'est pas dans la table et cette

dernière n'est pas pleine alors K est insérée.

D1. [premier hachage]

$i := h(K)$

D2. [premier test]

SI $T(i)$ vide ALLERA D6.

SI Donnée(i) = K l'algorithme se termine avec succès.

D3. [second hachage]
 $c := h'(K)$

D4. [Avancer au prochain]
 $i := i - c$
SI $i < 0$ ALORS $i := i + M$

D5. [Comparer]
SI $T(i)$ est vide ALLERA D6.
SI Donnée(i) = K l'algorithme se termine avec succès.
SINON ALLERA D4

D6. [Insérer]
SI $N = M - 1$ ALORS "débordement"
SINON $N := N + 1$
Rende $T(i)$ occupé
Donnée(i) := K



Chaînage interne

Afin de réduire le temps de recherche on ajoute au niveau de chaque élément du tableau un champ LIEN.

Ce champ contient la prochaine position de la table à examiner pour la recherche d'un élément donné.

Les *synonymes* sont ainsi rangés dans une *liste linéaire chaînée représentée dans un tableau*.

D'où l'appellation de la méthode. Une liste contient des groupes de synonymes.

Quand une collision se produit sur une case k , on parcourt la liste des débordements commençant en k .

Si la donnée n'est pas trouvée, on cherche un emplacement vide dans le tableau.

➡ Une bonne façon de rechercher une position vide c'est de parcourir le tableau à partir de la position

R (initialisée à $N+1$). On décrémente R une ou plusieurs fois jusqu'à ce que l'on trouve une position vide.

Ainsi, à un moment donné, toutes les positions p telles que $p > R$ sont occupées.

Exemple:

L'insertion des données suivantes avec leurs transformées (entre parenthèses) : **a(3), b(2), c(3), d(2), e(1)** dans une table T de 6 éléments donne :

0		
1	e	.
2	b	5
3	a	6
4		
5	d	.
6	c	.

Table T

- a est insérée à la position 3 du tableau.
- b est insérée à la position 2 du tableau.
- L'insertion de c provoque une collision sur la case 3. La donnée c est rangée à la position 6 du tableau. Le champ Lien de l'élément 3 de T pointe cette position.
- L'insertion de d provoque une collision sur la case 2. La donnée d est rangée à la position 5 du tableau. Le champ Lien de l'élément 2 de T pointe cette position.
- e est insérée à la première position du tableau.



Algorithme

Cet algorithme recherche une donnée K dans une table de M éléments.

Si K n'est pas dans la table et celle-ci n'est pas pleine alors K est insérée.

- un élément occupé contient un champ Donnée(i) et un champ LIEN(i)
 - une variable auxiliaire R est utilisée pour nous aider à déterminer les espaces vides. Quand la table est vide $R = M + 1$.
- Après plusieurs insertions on a : T(j) occupé pour tout j tel que $R \leq j \leq M$.

Par convention T(0) sera toujours vide.

C1. [Hash]

$i := h(K) + 1$ { donc $1 \leq i \leq M$ }

C2. [Existe-t-il une liste ?]

SI T(i) est vide ALLERA C6

{ dans les autres cas T(i) est occupé; on consulte alors la liste des noeuds occupés }

C3. [Comparer]

SI $K = \text{Donnée}(i)$,

l'algorithme se termine avec succès.

C4. [Avancer au prochain]
 SI LIEN(i) \neq 0 alors
 i := LIEN(i)
 ALLERA C3

C5. [Trouver le noeud vide]
 { la recherche est sans succès et nous voulons chercher une position vide dans la table }
 Décrémenter R une ou plusieurs fois jusqu'à ce que T(R) soit vide.
 SI R = 0 l'algorithme se termine avec débordement.
 Autrement faire :
 LIEN(i) := R
 i := R

C6. [Insérer la nouvelle clé]
 Rendre T(i) occupé avec
 Donnée(i) := K
 LIEN(i) := 0

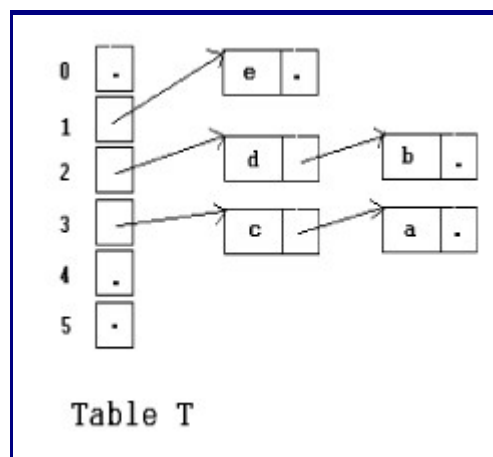


Chaînage séparé

Les *synonymes* sont rangées dans *une liste linéaire chaînée séparée*. D'où l'appellation de la méthode.

Exemple

L'insertion des données suivantes avec leurs transformées (entre parenthèses) : **a(3)**, **b(2)**, **c(3)**, **d(2)**, **e(1)** dans une table de 6 éléments donne :



- Une liste est créée avec un seul maillon contenant la valeur a. La tête de cette liste est rangée à la position 3 du tableau.
- Une liste est créée avec un seul maillon contenant la valeur b. La tête de cette liste est rangée à la

position 2 du tableau.

- La donnée c est un synonyme de a. Un maillon est donc alloué pour la contenir et est rajouté dans la liste correspondante.
- La donnée d est un synonyme de b. Un maillon est donc alloué pour la contenir et est rajouté dans la liste correspondante.
- Une liste est créée avec un seul maillon contenant la valeur e. La tête de cette liste est rangée à la position 1 du tableau.



Algorithme

Cet algorithme recherche une donnée K dans une table de M éléments.

Si K n'est ni dans la table ni dans la liste correspondante, elle est insérée.

- un élément occupé du tableau contient un champ T(i) qui contient la liste des synonymes.
- un élément de la liste contient deux champs : Donnée(p) et LIEN(p).
- Initialement T(i) := Nil pour tout i dans l'intervalle[0..M-1]

S1. [Hacher]

i := h(K)

S2. [Existe-t-il une liste ?]

SI T(i) est vide ALLERA S5

{ dans les autres cas T(i) est occupé; on consulte alors la liste des noeuds occupés }

p := T(i)

S3. [Comparer]

Si K = Donnée(p)

l'algorithme se termine avec succès.

S4. [Avancer au prochain]

Si Lien(p) <> Nil alors

p := LIEN(p)

ALLERA S3

S5. [Insérer la nouvelle clé]

Allouer un maillon, soit q

Donnée(q) <---- K

LIEN(q) <---- T(i)

T(i) := q



Les algorithmes de suppression



Essai linéaire

Soit i l'adresse de l'élément à supprimer. Les étapes suivantes permettent de réaliser une suppression :

1. Rendre $T(i)$ vide.

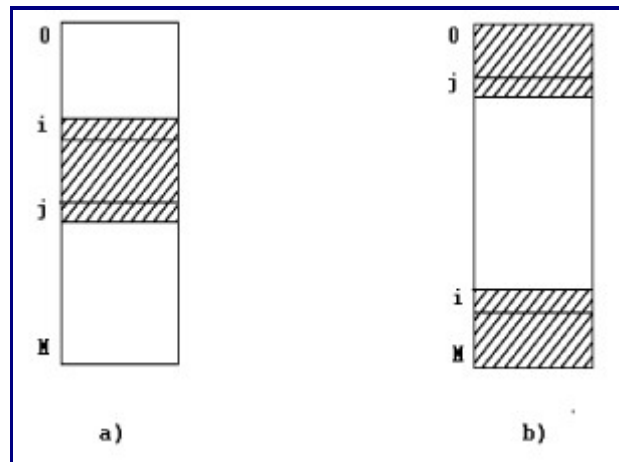
Poser $j := i$.

2. $i := i - 1$; Si $i < 0$: $i := i + M$ Fsi

3. Si $T(i)$ vide l'algorithme se termine.

Sinon Soit $r := h(T(i))$

Les deux cas de figure suivants peuvent se présenter :



a) $i < j$

Si $r < i$ ou $r \geq j$:

Déplacer l'élément, c'est à dire $T(j) := T(i)$

Fsi

b) $i > j$

Si $j \leq r < i$:

Déplacer l'élément, c'est à dire $T(j) := T(i)$

Fsi

4. Recommencer à partir de 1.

◆ Double Hachage

On ne peut trouver un algorithme analogue à celui de l'**essai linéaire**. Un moyen simple consiste à faire

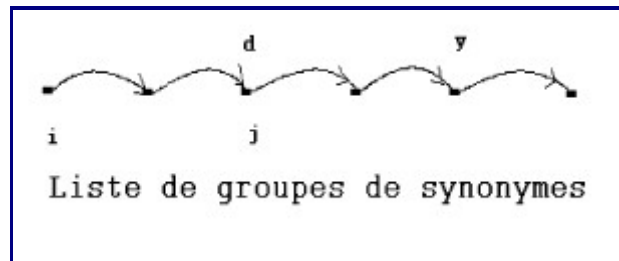
une **suppression logique**, c'est à dire le positionnement d'un bit qu'on rajoute au niveau des entrées de la table.

◆ Chaînage interne

Soit à supprimer l'élément d. Rechercher l'élément.

Soit i son adresse primaire, et j l'indice de l'élément d. (i peut être égal à j)

i est donc la liste qui contient d.



L'algorithme est le suivant :

1. Voir s'il existe plus loin dans la liste un synonyme y de d, c'est à dire un y tel que $h(y) = d$.
 2. Si y n'existe pas, on supprime d de la liste en ajustant le chaînage et l'algorithme se termine.
- Si y existe, on le déplace à la position j. Poser $j := \text{indice de } y$ et $d := y$ et recommencer à partir de 1.

Dans les deux cas, on met à jour la variable R comme suit:

⇒ (k étant l'indice de l'élément supprimé) **SI $k > R$: $R := k + 1$ FSI**

◆ Chaînage séparé

L'algorithme de suppression d'un élément est très simple. Ca consiste tout simplement à supprimer un élément d'une liste linéaire chaînée.



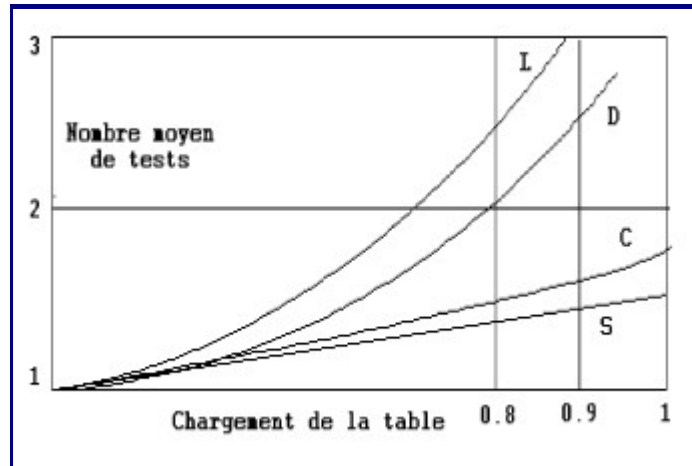
Comparaison entre les différentes méthodes

La figure suivante résume les courbes des nombres moyens de tests pour une recherche par rapport

au chargement de la table (N/M , N étant le nombre des éléments présents dans la table) pour les quatres

méthodes présentées.

L désigne l'essai linéaire, **D** le double hachage, **C** le chaînage interne et **S** le chaînage séparé.



Comme la figure le montre, les méthodes de chaînage semblent les meilleures. Cependant, leur inconvénient

réside dans le champ additionnel représentant les liens. Les méthodes de hachage ont de très bonnes

performances. Pour l'essai linéaire par exemple, même quand la table est remplie à 90 % , le nombre de tests

est au voisinage de 5.

Pour avoir un temps rapide (de l'ordre de 1) , on convient de ne pas dépasser un chargement de 70%.



Etude de la distribution des données

Nous montrons dans cette étude que

si la fonction de hachage est aléatoire, alors on peut prévoir le nombre de collisions, et donc comment les réduire.

[Distribution de Poisson collisions](#)

[Prévention des collisions](#)

[Réduction des](#)



Distribution de Poisson

Supposons N adresses possibles et considérons les 2 événements :

A : une adresse donnée n'est pas choisie

B : une adresse donnée est choisie.

⇒ Cas où une donnée est hachée

Quand une donnée est hachée, l'un des événements (A ou B) se produit pour une adresse donnée.

Soit **$P(A) = a$** et **$P(B) = b$**

$P(A)$ désigne la probabilité qu'une adresse donnée ne soit pas choisie.

$P(B)$ désigne la probabilité qu'une adresse donnée soit choisie.

On a donc :

$$\mathbf{P(B) = 1/N = b}$$

$$\mathbf{P(A) = 1 - 1/N = (N-1)/N = a}$$

Par exemple, si $N=10$ alors $a=0.1$ et $b=0.09$.

⇒ Cas où deux données hachent la même adresse

Qu'elle est la probabilité pour que deux données hachent la même adresse ?

$$\mathbf{P(BB) = b.b = 1/N . 1/N} \text{ (événements indépendants)}$$

Qu'elle est la probabilité pour que la seconde donnée soit hachée dans une adresse différente de la première ?

$$\mathbf{P(BA) = b.a = 1/N . (N-1/N)}$$

⇒ Cas où deux données parmi quatre hachent la même adresse

Noter d'abord que **$P(BABBA) = b.a.b.b.a = b^3a^2$** .

Nous voulons connaître la probabilité pour qu'il existe un certain nombre de fois de B et de A

(sans tenir compte de l'ordre). Par exemple, supposons que nous voulons hacher 4 données et que nous voulons

savoir de quelles façons deux données hachent la même adresse.

On peut avoir les 6 événements suivants : **AABB, BAAB, BABA, BBAA, ABBA, ABAB**

$$P = P(AABB) + P(BAAB) + \dots = 6 a^2 b^2$$

$$P = C_{42} a^2 b^2.$$

C42 représente le *nombre de façons qu'on peut placer 2 A (et 2 B) dans 4 places.*

◆ Généralisation

Si x données parmi r hachent la même adresse on a x fois B et (r-x) fois A. La probabilité pour que x données

parmi r hachent la même adresse est **$C_{rx} \cdot a^{r-x} b^x$** avec **$C_{rx} = x! / (x! (r-x)!)$**

Ce qui veut aussi dire : *Probabilité qu'une adresse donnée soit choisie x fois et ne soit pas choisie r-x fois.*

Si N adresses possibles

$$P(x) = C_{rx} a^{r-x} b^x$$

$$P(x) = C_{rx} (1-1/N)^{r-x} (1/N)^x$$

P(x=0) veut dire probabilité pour qu'une adresse donnée ne soit jamais choisie.

$$P(0) = C_{r0} (1-1/N)^r (1/N)^0$$

P(x=1) veut dire probabilité pour qu'une adresse donnée soit choisie une seule fois

$$P(1) = C_{r1} (1-1/N)^{r-1} (1/N)^1$$

⇒ L'inconvénient de la formule est qu'elle est difficile à calculer pour N et r grands. La fonction de **POISSON**

est une bonne approximation.

$$P(x) \approx f(x) = \left(\left(\frac{r}{N} \right)^x \cdot e^{-\left(\frac{r}{N} \right)} \right) / x!$$

Si N est le nombre d'adresses possibles,

r est le nombre de données insérées

x est le nombre de données ayant la même adresse

alors P(x) donne la probabilité que x données hachent la même adresse parmi r données insérées.

Par exemple

pour N=1000 adresses et r= 1000 données insérées on obtient les valeurs suivantes:

$$P(0) = .368 ; P(1) = .368 ; P(2) = .184 ;$$

$$P(3) = .061 ; \text{ etc.}$$

En général, s'il existe N adresses, N.P(x) est le nombre de données qui hachent x fois la même adresse.

$P(x)$ est aussi la proportion d'adresses ayant x données qui lui sont attribuées par hachage.
Ceci nous permet donc de prévoir le nombre de collisions.



Prévention des collisions

Prenons $N = 10\,000$ adresses possibles
et $r = 10\,000$ données insérées.

Quel est le nombre d'adresses qui n'ont aucune donnée attribuée ?

$$10\,000 * P(0) = 3679$$

Quel est le nombre d'adresses qui n'ont qu'une seule donnée attribuée ?

$$10\,000 * P(1) = 1839$$

Quel est le nombre d'adresses qui n'ont que deux données attribuées ?

$$10\,000 * P(2) = 613$$

Quel est le nombre d'adresses qui n'ont que trois données attribuées ?

$$10\,000 * P(3) = 183$$

Pour 3679 données il n'y a pas de collisions.

Pour 1839 données il y a collision (1839 seront en débordement)

Pour 613 il y a collision (613 * 2 seront en débordement)

C'est une mauvaise répartition : nous avons des milliers d'adresses (3679) avec aucune donnée attribuée.



On peut ainsi prévoir le nombre de données en débordement à l'avance. Il est donné par la formule

$$N (P(2) + 2P(3) + ...iP(i+1) +)$$



Réduction des collisions

Montrons à l'aide d'exemples d'une part, comment l'augmentation du nombre d'adresses possibles et d'autre

part, comment l'utilisation des cases peuvent réduire les collisions.

Augmentation de l'espace des adresses

Nous définissons la densité d par :

$$d = r / N$$

où r est le nombre de données rangées et N est le nombre d'adresses possibles.

Regardons le comportement des fonctions de hachage (collisions) pour différentes valeurs de d .

$$P(x) = \frac{(r/N)^x e^{-(r/N)}}{x!}$$

$$(dx \cdot e^{-d}) / x!$$

$P(x)$ dépend donc du rapport r/N c'est à dire de d .

Aussi, on constate un même comportement pour 500 données distribuées parmi 1000 adresses que 500.000 données distribuées parmi 1 million d'adresses. ($d = 50\%$ pour les deux cas)

Prenons $d = 0.5$ ($N = 1000$ et $r = 500$ données)

Combien d'adresses auront 0 donnée attribuée ?

$$1\,000 * P(0) = 607$$

Combien d'adresses auront 1 donnée attribuée ?

$$1\,000 * P(1) = 303$$

Combien d'adresses auront au moins deux données attribuées ?

$$1\,000 * (P(2) + P(3) + P(4) + \dots) = 90$$

avec $P(2) = 0.0758$; $P(3) = 0.0126$; $P(4) = 0.0016$; etc.

Quel est le nombre de données en débordement ?

$$1\,000 * (2 * P(2) + 3 * P(3) + 4 * P(4) + \dots) = 107$$

Quel est le pourcentage des données en débordement ?

$$107 / 500 = 21,4 \%$$

On peut donc conclure :

Si la densité est de 50 %, on peut s'attendre à 79% de données rangées dans leur adresse primaire et 21 % rangées ailleurs.

Utilisation des cases

On accepte b données par adresse possible.

Dans ce cas : $d = r / (b.N)$

b : nombre de données par case

N : nombre de cases

r : nombre de données insérées.

Raisons de l'utilisation de l'espace secondaire

On peut énumérer les raisons suivantes qui nous poussent à mettre les informations sur les supports externes : - l'espace mémoire est limité. Si on a une grande masse d'informations à ranger, on est obligé de la mettre sur un support externe. - la RAM (Random Access Memory, c'est à dire la mémoire centrale) coûte chère. - la RAM est volatile (légère). L'information n'est pas en sécurité puisqu'elle peut être perdue à la moindre panne.

Problèmes avec le stockage externe

Le problème avec le stockage sur les supports externes réside dans la lenteur des accès. Pour rechercher une information sur le disque par exemple, on fait généralement plusieurs accès avant de la trouver. Et pendant ce temps là, la machine peut dérouler des millions d'opérations. Pour concevoir une méthode de rangement de l'information sur un support externe, il faut toutefois mettre à l'esprit qu'au delà de 5-6 accès, ça devient insupportable. Par conséquent, on est contraint à trouver une façon efficace pour le stockage des informations.



Différence entre la RAM et la mémoire secondaire

La différence essentielle entre la RAM et la mémoire secondaire est que pour deux informations différentes le temps d'accès est le même pour la RAM et peut être très différent pour la mémoire secondaire.



Fichier

Vue logique

Le fichier est un ensemble d'articles. A chaque article est associée une clé qui l'identifie de façon unique. C'est ce qu'on appelle la clé primaire. Les autres champs peuvent constituer des clés secondaires. (identification non unique)

Le fichier peut aussi être vu comme un ensemble d'octets représentant des informations : les octets sont organisés hiérarchiquement pour former des champs , des articles ou des blocs.

Vue physique

Physiquement le fichier est un ensemble de blocs. Un bloc contient n articles. Un article contient n champs. Le bloc constitue généralement l'unité de transfert entre la RAM et la mémoire secondaire.



Bloc d'en-tête

Un ensemble d'informations sont nécessaires pour l'exploitation du fichier, c'est ce qu'on appelle les caractéristiques du fichiers. Ces informations sont rangées généralement dans un bloc spécial du fichier appelé bloc d'en-tête. Il existe deux types d'information :

➤ informations statiques, telles que le nom du fichier, la date de création, etc.

➤ informations dynamique telles que le nombre courant d'articles, l'adresse du dernier bloc, etc.



Fichiers statique et dynamique

Fichier statique

C'est un fichier qui subit très peu d'insertions et de suppressions. On l'appelle aussi *fichier off-line*.

Fichier dynamique

C'est un fichier où les insertions et les suppressions sont très fréquentes. On l'appelle aussi *fichier on-line*.



Méthode d'accès

C'est l'approche utilisée pour localiser un article dans un fichier. En général, on peut classer les méthodes en deux catégories : *accès séquentiel* et *accès direct*.



Organisation du fichier

C'est la combinaison de structures physique et conceptuelle utilisées pour distinguer un article d'un autre, un champ d'un autre, etc. La longueur de l'article peut être fixe ou variable. Si l'article est de longueur variable, il peut être à cheval sur deux blocs.



Adressage

Bloc

Un bloc est repéré par son adresse. Généralement on distingue deux types d'adresses : physique et logique. Il existe donc toujours une fonction d'association entre l'adresse physique et l'adresse logique. Nous nous considérons par la suite que les adresses logiques.

Article

Un article est repéré par son adresse octet par rapport au début du fichier ou par le couple (numéro du bloc, déplacement dans le bloc). De la même manière, on distingue l'adresse logique et l'adresse physique.



Critère de mesure d'une méthode d'accès

Pour comparer ou mesurer des méthodes d'accès, on considère généralement les facteurs suivants :

➤ *taux d'occupation = nombre d'articles / (nombre de blocs alloués au fichiers * nombre maximum d'articles possible dans le bloc)*. Un taux de **70 %** constitue généralement un bon compromis.

➤ nombre d'accès disque.

➤ encombrement et complexité des algorithmes de maintenance : recherche, insertion suppression, etc.

➤ réaction de la méthode aux pannes systèmes.



Dans les langages de programmation

Dans les langages de programmation, on parle de fichiers physique et logique. Le fichier physique est le fichier réel qui se trouve sur le support. Le fichier logique n'est connu qu'à l'intérieur d'un programme. Ainsi, à un même fichier physique, on peut correspondre plusieurs fichiers logiques. Il existe toujours un moyen de faire le lien entre le fichier physique et le fichier logique. Il peut se faire à l'intérieur du programme ou à l'extérieur.

Dans les langages de programmation, on définit aussi un ensemble d'opérations : création, ouverture, fermeture, lecture, écriture, position de la tête de lecture/écriture.



Algorithmes

Pour développer les algorithmes sur les structures de fichiers, on commence toujours par définir le type des blocs utilisés. Si le format est fixe, le bloc est généralement un tableau d'articles. Dans le cas où le format est variable, le bloc est considéré comme un tableau de caractères. Le découpage de l'article ainsi que la séparation des articles sont à la charge du programmeur.

Au niveau algorithmique, un fichier F peut être défini comme suit :

F : fichier de Typebloc

où Typebloc est *le type du bloc du fichier*.

Les opérations suivantes sont nécessaires :

Lirebloc(F, n, Zone) :

lecture, dans la buffer Zone, du bloc n du fichier F

Ecrirebloc(F, n, Zone) :

écriture du buffer Zone dans le bloc n du fichier F



Objectifs du cours

Généralement, dans un langage de programmation, la méthode d'accès utilisée est cachée de l'utilisateur. Dans les chapitres qui suivent, nous tenterons d'analyser les principales méthodes d'accès.

L'objectif de notre cours consiste donc à répondre aux deux questions suivantes :

Comment définir :

- L'organisation du fichier : séparation des champs, des articles,
- La méthode d'accès : comment localiser un article sur le fichier ? Ca revient donc à étudier les

opérations de base : - recherche - insertion - suppression

➤ Requête à intervalle sous différentes formes d'accès qui sont : - fichier vu comme un tableau - fichier vu comme une liste linéaire chaînée

➤ Les méthodes d'index

➤ L'accès multi-clés

➤ Les méthodes d'arbres : arbres de recherche m-aires et arbres B

➤ Les méthodes de hachage.

Fichier vu comme un tableau

Le fichier est un ensemble de *M blocs contigus* sur le disque. C'est par conséquent *un tableau*.

Le fichier peut être ordonné ou pas.

S'il n'est pas ordonné, les ajouts d'articles se font à la fin du fichier. L'accès est alors *séquentiel*.

S'il est ordonné, les ajouts d'articles causent des décalages. L'accès est alors soit *dichotomique* soit *séquentiel*.

Fichier vu comme une liste linéaire chaînée

Le fichier peut être vu comme *une liste linéaire chaînée de blocs*, c'est à dire constitué de *M blocs non contigus sur le disque*.

Le fichier peut être ordonné ou pas.

S'il n'est pas ordonné, les ajouts d'articles se font à la fin du fichier.

S'il est ordonné, les ajouts d'articles causent des insertions de nouveaux blocs (allocation dynamique).

Dans les deux cas, l'accès est exclusivement *séquentiel*.



Remarques

Remarque 1

L'organisation du fichier est quelconque : séparation des champs, des articles, format fixe ou variable.

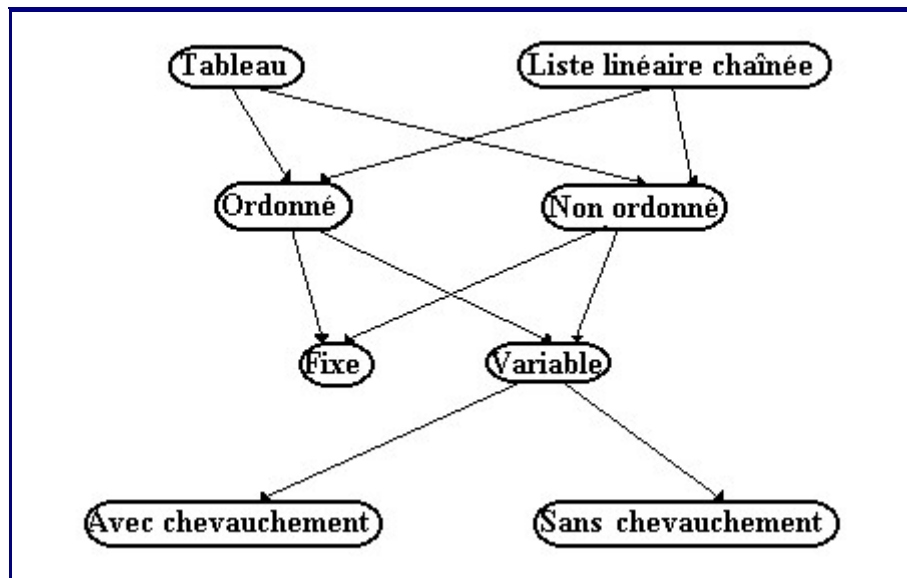
Remarque 2

Si le fichier est ordonné, on fait d'abord un chargement initial en remplissant les blocs à 60-70%. On économise ainsi les décalages intra-blocs.



Récapitulatif

En combinant les méthodes d'accès et les différentes organisations internes possibles d'un fichier on peut construire une multitude de structures simples de fichiers que l'on peut résumer dans le graphe suivant :



En suivant les flèches de haut en bas, on peut construire une douzaine de structures de fichiers. Comme exemple, on peut citer :

1. **Tableau, Ordonné, Fixe**
 2. **Liste linéaire chaînée, Non ordonné, Variable, Avec chevauchement**
 3. **Liste linéaire chaînée, Ordonné, Fixe**
- etc.

Dans le premier cas, le fichier est un tableau. Il est ordonné, ce qui veut dire que pour deux blocs p1 et p2, si p1 précède p2 alors tous les articles dans le bloc p1 sont inférieurs (selon la clé) aux articles du bloc p2. De plus, à l'intérieur d'un bloc, les articles sont de longueur fixe.

Dans le deuxième cas, le fichier est une liste linéaire chaînée de blocs. Les articles ne sont pas ordonnés et sont de longueur variable. Un article peut commencer dans un bloc et finir dans le bloc suivant.



Utilisation

Ces techniques sont très efficaces pour le traitement séquentiel, c'est à dire quand le seul traitement consiste à appliquer une opération pour chaque élément du fichier.

Si l'opération de recherche d'éléments est très fréquente, on préfère de loin utiliser un tableau ordonné avec format fixe parmi toutes les structures présentées dans la figure précédente. Ceci bien entendu pour pouvoir pratiquer la recherche dichotomique. Toutefois, on se limite à des petits fichiers. Rappelez vous, au delà de 5-6 accès, une recherche externe est considérée comme insupportable. Par conséquent, on accepte pas plus de $26 = 64$ blocs. Si chaque bloc renferme 100 articles et si le facteur de chargement est de 70%, alors on peut construire des fichiers de l'ordre de $64 \times 100 \times 70\% = 4480$ articles dans lesquels toute recherche d'article est accomplie en 3 accès disque en moyenne.



Exemple : recherche dichotomique

On se place dans le cas, où le fichier est un tableau de M blocs sur le disque. Le fichier est ordonné et le format des articles est fixe. Donnons l'algorithme de la recherche dichotomique.

Le bloc est défini comme suit :

TYPE Tbloc =	STRUCTURE
Nb :	ENTIER {Nombre d'éléments dans le bloc }
Articles :	TABLEAU[1..B] de Typearticle
FIN	

TYPE Typearticle =	STRUCTURE
Cle :	Typecle
Info :	Typeqq
FIN	



Typecle peut être tout ensemble muni d'une relation d'ordre.



Typeqq est un type quelconque qui dépend de l'application.

On définit, en mémoire, un buffer ou une zone pour les transferts entre la mémoire principale et la mémoire secondaire.

Var Zone : Tbloc

L'algorithme de la recherche dichotomique est le suivant:

```

Bi := 1
Bs := M
Trouv := FAUX
TANTQUE Bi <= Bs ET NON Trouv
    Mil := (Bi + Bs) / 2 { divison entière }
    Lire(F, Mil, Zone)
    SI Cle >= Zone.Articles[1].Cle ET
Cle <= Zone.Articles[Zone.Nb].Cle
        Rechinterne(Clé, Trouv)
    SINON
        SI Cle < Zone.Articles[1].Cle
            Bs := Mil - 1
        SINON
            Bi := Mil + 1
    FSI
FSI
FINTANTQUE

```

Rechinterne est le module qui fait la recherche dichotomique dans le buffer Zone. Si l'article est trouvé, il rend dans la variable Trouv la valeur VRAI sinon la valeur FAUX.

Commentaires

Noter l'importance de *l'opération de lecture*. Sans cette opération, on ne peut consulter un bloc du fichier.

Remarquer aussi qu'il y a *une double dichotomie* : une recherche dichotomique externe et une interne.

Si le format des articles est variable, *la recherche interne est séquentielle*.

Le temps occupé par l'algorithme est $NL.u + \mu$ où NL est le nombre de lecture et u est le temps moyen d'un accès disque (caractéristique de la machine). μ est négligeable devant le produit NL.u. La complexité de l'algorithme est par conséquent $O(\log_2(M))$.



TRAVAUX DIRIGES

1. Considérer les douze cas de la figure présentée en cours donnant les différentes méthodes d'accès. Pour chaque cas, développer les algorithmes de

- recherche
- insertion
- suppression
- requête à intervalle

Dans le cas où le fichier est un tableau ordonné, développer en plus l'opération de chargement initial qui consiste à remplir le fichier à raison de $\mu\%$ par bloc par des articles qui sont lus en ordre croissant selon leur clé.

La requête à intervalle consiste à récupérer tous les articles dont les clés sont comprises entre deux clés a et b données.

Des structures simples aux méthodes d'index

Si l'opération de recherche est très fréquente pour une application donnée, la meilleure alternative parmi les structures simples de fichiers est le choix d'un tableau ordonné avec un format fixe des articles. Par conséquent, la méthode se trouve limitée à des fichiers dont la taille ne peut excéder quelques milliers d'articles.

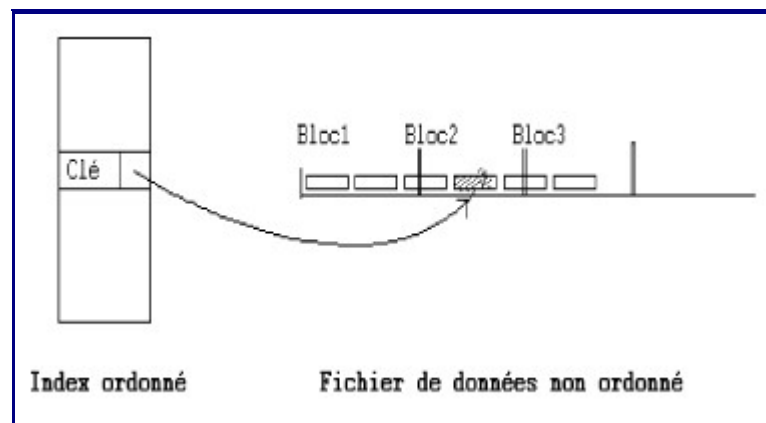
Au lieu de trier le fichier et de pratiquer une recherche dichotomique, ce qui est trop chère sur un fichier, on utilise un index. La recherche dichotomique se fait alors entièrement en mémoire.

Pour ces méthodes, deux fichiers sont utilisés : *le fichier d'index* et *le fichier au données*. En règle générale, le fichier de données n'est pas trié. Il peut être un tableau ou une liste linéaire chaînée de blocs sur le disque. Le fichier d'index est supposé en mémoire et est trié selon les clés des articles. L'index peut être à un ou plusieurs niveaux comme nous allons le décrire par la suite. L'organisation interne du fichier est quelconque.



Index à un niveau

L'index est un ensemble de couples(clé, adresse) rangés entièrement en mémoire centrale. Le fichier est un ensemble de blocs (dans l'exemple les blocs sont contigus) sur le disque. Le bloc contient un ensemble d'articles. Les articles peuvent être à cheval sur deux blocs logiquement consécutifs. C'est le cas de la figure suivante :



Pour rechercher un article de clé donnée dans le fichier, on commence par faire une recherche

dichotomique sur l'index. Si la clé est trouvée, le bloc contenant l'article est ramené en mémoire centrale afin de récupérer l'article. Si ce dernier est à cheval sur deux bloc, le deuxième bloc est ramené pour récupérer le reste de l'article.

Une insertion d'article est réalisée comme suit :

- si la clé n'est pas trouvée dans l'index, elle est y insérée et peut donc entraîner des décalages.
- l'article est inséré en fin de fichier.

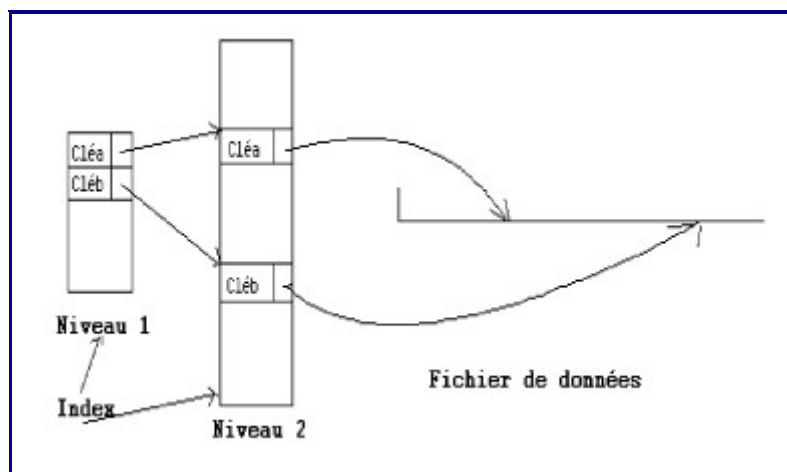
Une suppression est généralement logique. Elle est donc extrêmement rapide.

Une réorganisation est donc à prévoir afin *d'éliminer physiquement* les articles supprimés. Généralement, on construit un autre fichier.



Index à 2 niveaux

Afin d'accélérer la recherche en mémoire centrale, on crée deux niveaux d'index. Dans l'index du niveau 1, on range certaines clés, par exemple le 100-ième, le 200-ième, le 300-ième, ... et dans l'index de niveau 2 toutes les clés.



Pour rechercher un article de clé donnée, on commence par rechercher sa clé dans l'index du niveau 1. un intervalle est alors sélectionné. La recherche continue dans l'index de niveau 2 uniquement dans cet intervalle. Si la clé est trouvée, on procède de la même façon que dans le cas avec un seul index. Il est clair que les deux recherches dichotomiques sur les deux petits vecteurs (index de niveau 1 et une partie de l'index de niveau 2) sont beaucoup plus rapides qu'une seule recherche dichotomique sur un seul grand vecteur (index de niveau 2).



Opérations de base

Pour mettre au point une méthode d'index, les opérations suivantes sont nécessaires :

- ➡ créer un fichier d'index et de données
- ➡ charger le fichier d'index en mémoire avant utilisation
- ➡ réécrire l'index après utilisation,
- ➡ rechercher un article de clé donnée,
- ➡ insérer un article,
- ➡ supprimer un article,
- ➡ modifier un article.



Remarques

Généralement pour ces méthodes, l'index est supposé en RAM pendant l'exploitation du fichier.

Si l'index est large pour être contenu en mémoire ((i) - recherche binaire chère et (ii) réarrangement coûteux à cause des décalages) on a recours à d'autres techniques :

➡ **Hachage** : ces méthodes assurent une extrême rapidité (entre 1 et 2 accès) mais elles ne sont efficaces que pour les fichiers statiques et non ordonnés.

➡ **Arbres** : ces méthodes sont rapides mais de degré moindre par rapport aux méthodes de hachage (de l'ordre de 3 accès). Certaines sont très adaptées pour les fichiers ordonnés et même dynamiques.



Avantages des méthodes d'index

Même si l'index n'est pas en mémoire, les méthodes d'index ont les avantages suivants :

- ➡ permettent la recherche dichotomique même pour les articles de longueur variable.
- ➡ faire la recherche dichotomique sur le fichier d'index est beaucoup plus rapide que sur le fichier de données lui-même.
- ➡ L'effacement des articles par des "flag" se fait sans accès disque.



TRAVAUX DIRIGES

1. Fichier avec un seul index (accès par clé primaire).

1.a) Le fichier est un ensemble de blocs contigus 1, 2,N. A chaque article est associée une clé qui l'identifie de façon unique. Les articles sont de longueur fixe. Le fichier peut être le suivant :
Le fichier d'index est ordonné. Le fichier de données ne l'est pas. l'index est supposé en mémoire.
Concevoir les algorithmes de :

- définition et de création des fichiers d'index et de données.
- chargement du fichier d'index en mémoire avant son utilisation. Nous supposons que le fichier d'index est assez petit pour qu'il puisse entrer et rester en mémoire. Ça revient donc à ranger l'index dans un tableau en mémoire.

- réécriture du fichier d'index de la mémoire après son utilisation. Si l'index a été modifié, le réécrire comme un nouveau fichier.
- recherche d'un article de clé donnée. Trouver la position de la clé au niveau de l'index (recherche binaire), puis faire un accès direct pour retrouver la donnée.
- insertion des articles au fichier de données et d'index. On l'ajoute en fin du fichier de données. Sur l'index, les décalages sont obligatoires, ce qui n'est pas très grave car le traitement se fait entièrement en mémoire (aucun accès disque).
- suppression des articles du fichier. Considérer le cas où elle est logique (simple) et le cas où elle est physique (plus complexe)
- modification des articles. 2 sortes de modifications selon que la clé est modifiée ou non.

1.b) Même chose dans le cas où les articles sont de longueur variable.

2. Fichier avec deux niveaux d'index (accès par clé primaire).
Considérer le cas où les articles sont de longueur fixe et refaire la même chose qu'en 1).

Introduction

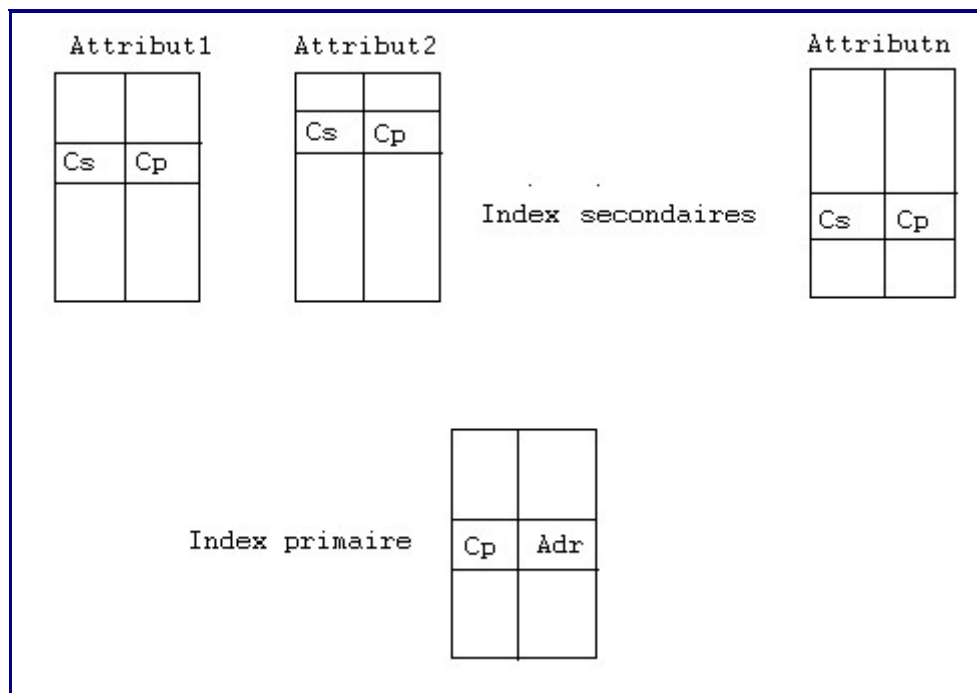
On désire récupérer l'article d'un fichier qui a tel et/ou tel attribut. Par exemple, si l'article d'un fichier bibliothèque possède les rubriques suivantes :

Référence, Titre, Année, Auteur

on peut vouloir faire les requêtes du genre :

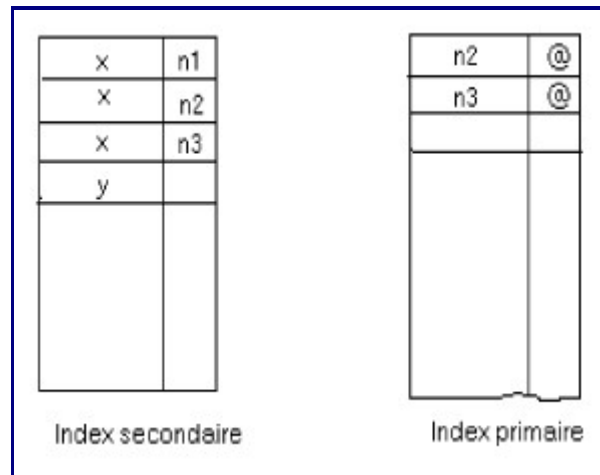
- retrouver tous les livres parus en 1973.
- retrouver tous les livres parus en 1973 de l'auteur X donné.

Une solution simple consiste à créer *des index secondaires pour chaque attribut* comme le montre la figure ci-dessous. Nous verrons plus loin pourquoi on ne met pas directement l'adresse de la donnée mais on passe toujours par l'index primaire.



Organisation d'un index secondaire

Essayons de voir maintenant comment on organise un index secondaire. Une première solution peut être la suivante :



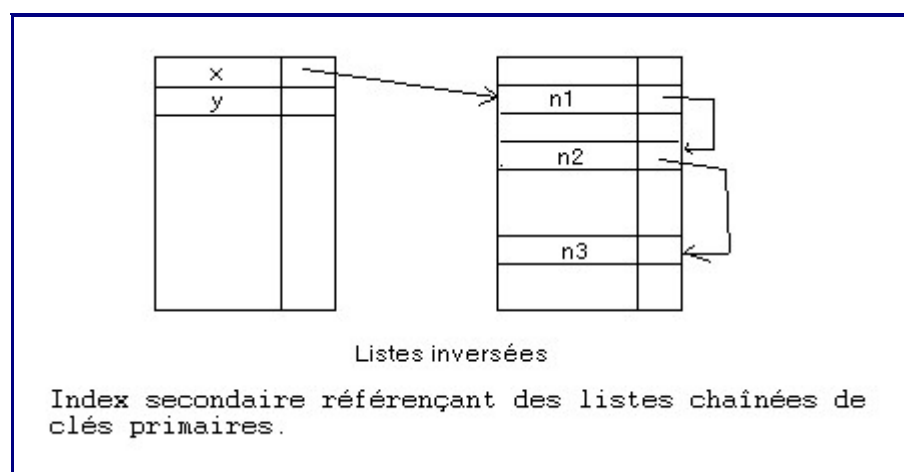
Une entrée d'un index secondaire est *le couple (Cs, Cp)* où Cs est une clé secondaire et Cp la clé primaire qui lui est associée. Les clés secondaires sont dupliquées, ce qui peut rendre l'index volumineux.

Afin de ne pas répéter les clés secondaires, une solution consiste à organiser l'index secondaire comme suit :

x	n1	n2	n3
y	n1		
z	p1	p2	

C'est à dire par un tableau et donc un nouveau problème apparaît : c'est celui de la taille (nombre de clés primaires par clé secondaire)

➡ Une autre solution (et c'est la meilleure) consiste à utiliser les *"Listes inversées"* comme le montre la figure suivante :





On les appelle ainsi car pour retrouver un article on commence par la fin c'est à dire : clé secondaire, puis la clé primaire avant d'arriver à l'article lui-même.



Exemple de combinaison de clés secondaires

On veut récupérer les articles possédant l'attribut X et l'attribut Y. On suppose donc l'existence des index sur les champs X et Y. L'algorithme suivant permet de retrouver les articles en question :

L'index sur X donne l'ensemble des clés primaires avec l'attribut X.

L'index sur Y donne l'ensemble des clés primaires avec l'attribut Y.

L'intersection donne l'ensemble des clés primaires ayant l'attribut X et Y. Pour chaque élément de cet ensemble, on fait alors un accès disque pour retrouver et lister l'article, après avoir bien entendu retrouvé son adresse dans l'index primaire.



Remarque

Pourquoi passer par l'index primaire?

Le problème réside au niveau de *l'opération de suppression*. Essayons de voir comment supprimer un article de clé secondaire donnée ?



Il faut l'enlever de tous les index secondaires, donc les réarranger tous, ce qui est excessivement cher surtout s'ils ne sont pas présents en mémoire centrale.

Si on ne les supprime pas et si la clé référence directement la donnée, il est impossible de dire quelles sont les références non valides.



La solution est donc de ne pas toucher aux index secondaires (les clés secondaires référencent les clés primaires) et ajouter un bit d'effacement au niveau de l'index primaire. Une recherche commence par l'index secondaire, puis l'index primaire. Dans ce dernier, on s'aperçoit que la clé a été effacée. D'où l'intérêt de ne pas pointer directement la donnée à partir de l'index secondaire.



TRAVAUX DIRIGES

1. Le fichier est un ensemble de blocs contigus 1, 2,N. A chaque article est associée une clé primaire qui l'identifie de façon unique. Les articles sont de longueur fixe. On convient de mettre B articles par bloc.

On considère deux index secondaires pour l'attribut X et Y de l'article. Ces index sont organisés sous forme de "listes inversées". Les fichiers d'index sont ordonnés. Le fichier de données ne l'est pas.

Concevoir les algorithmes de :

- chargement des fichiers d'index secondaire et primaire.
- sauvegarde des fichiers d'index secondaire et primaire.
- listage des articles de clé secondaire, selon l'attribut X , donnée.
- listage des articles ayant une clé secondaire, selon l'attribut X, donnée et une clé secondaire, selon l'attribut Y, donnée.
- insertion d'un article ayant A comme clé primaire, Ax et Ay comme clés secondaires selon les attributs X et Y respectivement.
- suppression d'un article de clé primaire donnée.

Des méthodes d'index aux méthodes d'arbres

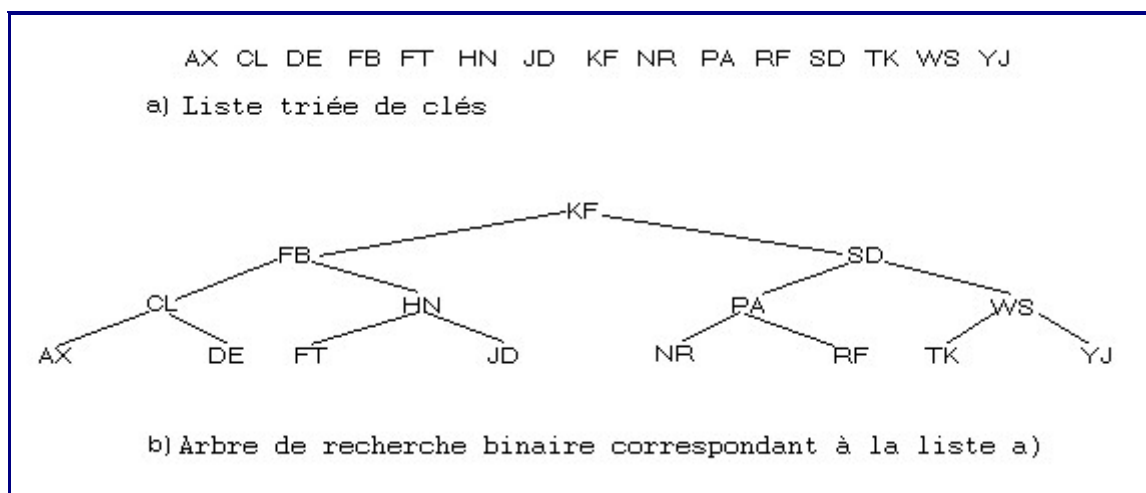
Oublions l'index secondaire et revenons à l'index primaire. Si le fichier est très volumineux, on ne peut garder l'index en RAM. De plus, garder l'index en mémoire secondaire coûte cher car :

- 1) la recherche dichotomique exige beaucoup d'accès disque.
- 2) il faut maintenir l'index ordonné (pour pouvoir faire la recherche dichotomique).

Nous devons donc trouver une meilleure façon de faire les insertions et les suppressions d'articles avec seulement une réorganisation locale.

Arbre de recherche binaire comme solution

Etant donnée une liste triée de nombres, on peut construire un arbre de recherche binaire comme le montre la figure suivante :



L'arbre peut être représenté dans un tableau comme suit :

Racine====> 9														
8			13			2		1	3	12			5	
FB	JD	RF	SD	AX	YJ	PA	FT	HN	KF	CL	NR	DE	WS	TK
10			6			11		7	0	4			14	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Avec cette structure on a pas besoin de trier l'index pour pratiquer la recherche binaire. Donc, c'est une solution pour le point 2).

Cette représentation admet les deux inconvénients suivants :

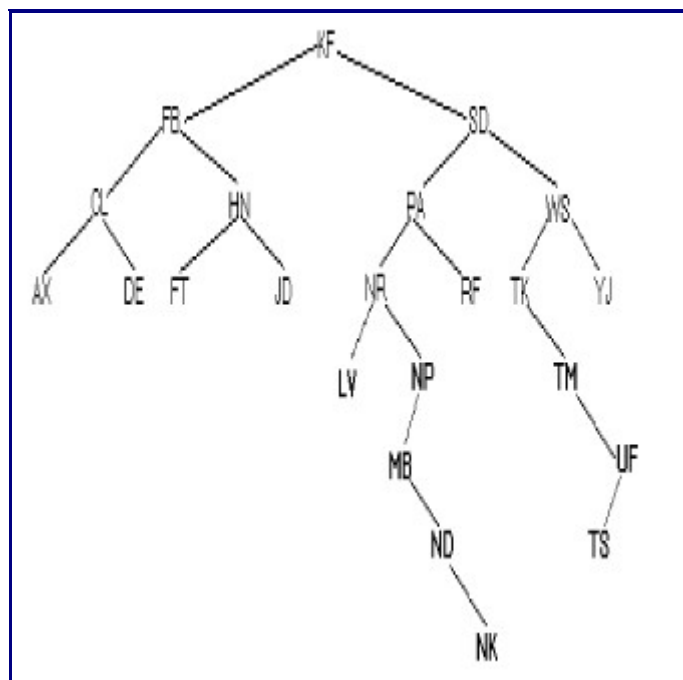
a) **Plus de la moitié des liens ne sont pas utilisés car se sont des noeuds feuilles. Ceci entraîne la perte d'un espace considérable.**

b) **Les performances de recherche sont très mauvaises si l'arbre est déséquilibré. Pour maintenir l'arbre équilibré, on utilise les techniques d'arbre AVL que nous développerons par la suite.**

Il faut aussi noter que la structure de l'arbre dépend de l'ordre d'arrivée des clés. Si l'arbre est équilibré on a de bonnes performances de recherche. Quand les clés arrivent de façon aléatoire, l'arbre peut se déséquilibrer et donc les performances se détériorent.

Si on ajoute à l'arbre précédent les clés : **NP , MB, TM, LA, UF, ND, TS, NK**

L'arbre devient :



Pour certaines clés, on peut avoir 5, 6 ou plusieurs accès avant de les retrouver. Une solution consiste à maintenir l'arbre équilibré. Un arbre maintenu équilibré est appelé *arbres AVL* (G.M Adel'son-Velskii et E.M Landis). Les arbres AVL garantissent les performances de la recherche et approchent celles de l'arbre binaire complet. Le maintien d'un arbre sous sa forme AVL implique l'application de 4 sortes de rotations. La plus complexe exige la modification de 5 liens (Voir **chapitre suivant**).

Revenons aux deux problèmes cités au début :

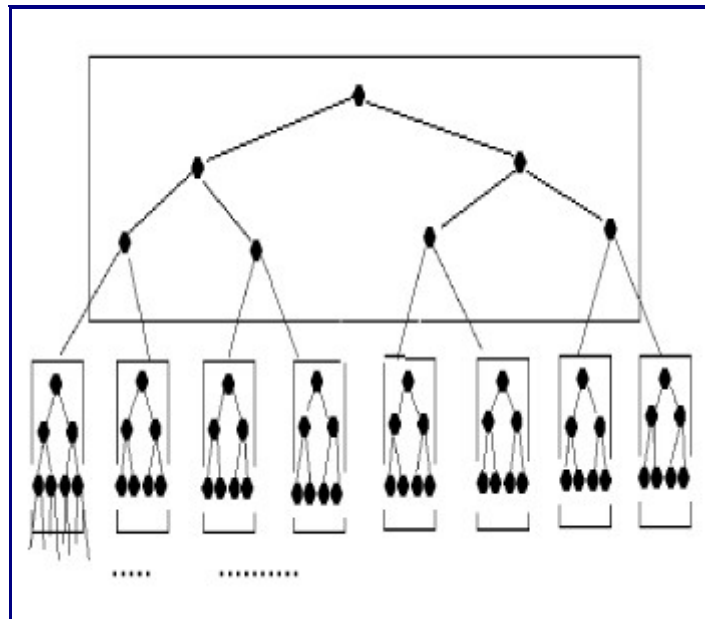
- 1) **recherche binaire exige plusieurs accès disque.**
- 2) **garder l'index ordonné coûte cher.**

Les arbres AVL apportent une solution au point 2.

Regardons le point 1) : une solution consiste à *paginer l'arbre*.

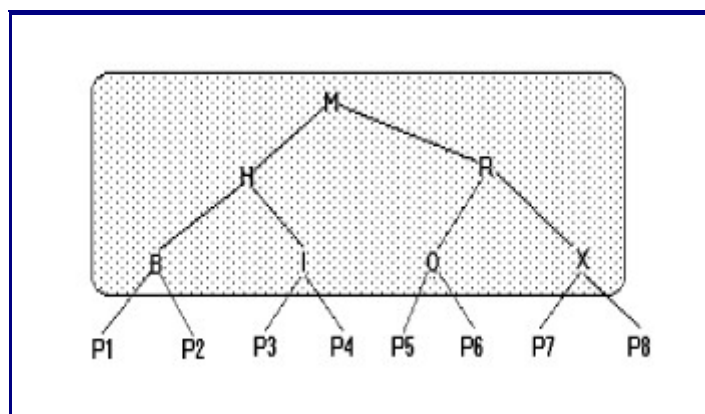
Pagination de l'arbre

Ça consiste à organiser l'arbre en pages. Chaque page contient une portion de l'arbre de recherche binaire. Les pages sont rangées dans des blocs sur le disque. L'organisation est alors la suivante :



Si on prend une page(bloc) de 8K-octets, ce qui permet d'avoir à peu près 511 (clés, références), et si on suppose que chaque page contient un arbre binaire complet alors avec 3 accès on peut retrouver un article parmi $511 \times 511 \times 511 = 134\,217\,727$.

Si on veut éviter la perte d'espace au détriment du temps on peut garder l'arbre de recherche binaire sous sa forme séquentielle "inordre". Ca revient donc à la structure de tableau. Par exemple, l'arbre suivant:



peut se mettre sous la forme du tableau suivant : (p1, B, p2, H, p3, I, p4, M, p5, O, p6, P, p7, X, p8)

➡ Si une clé est *insérée* on procède par *décalage*, ce qui n'est pas grave car la page est entièrement en mémoire.

➡ Si l'arbre de recherche binaire *n'est pas découpé* en pages on a **LOG2 (N + 1)** accès au maximum.

➡ Si l'arbre *est découpé* en pages à raison de k clés par page, on a **LOG k+1 (N+1)**.

Le premier cas est un cas particulier du deuxième avec k=1 (une clé par page).

Pour l'exemple précédent on :

$$\text{LOG}_2(134\ 217\ 727) = 27 \text{ accès}$$

$$\text{LOG}_{511+1}(134\ 217\ 727) = 3 \text{ accès.}$$

Donc c'est une solution pour le point 1).

On aboutit à ce qu'on appelle *un arbre de recherche m-aire*. C'est donc un arbre de recherche binaire découpé en pages.



Arbres AVL

Définitions

Techniques d'équilibrage

Algorithme d'insertion

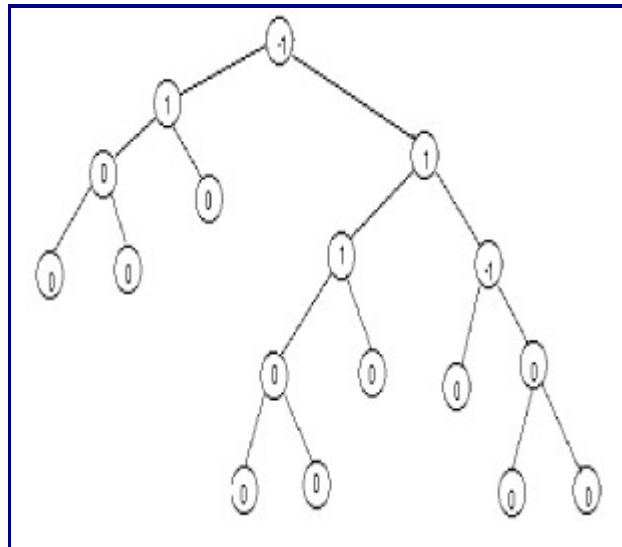
Analyse théorique

Exemple d'insertions de données dans un arbre AVL

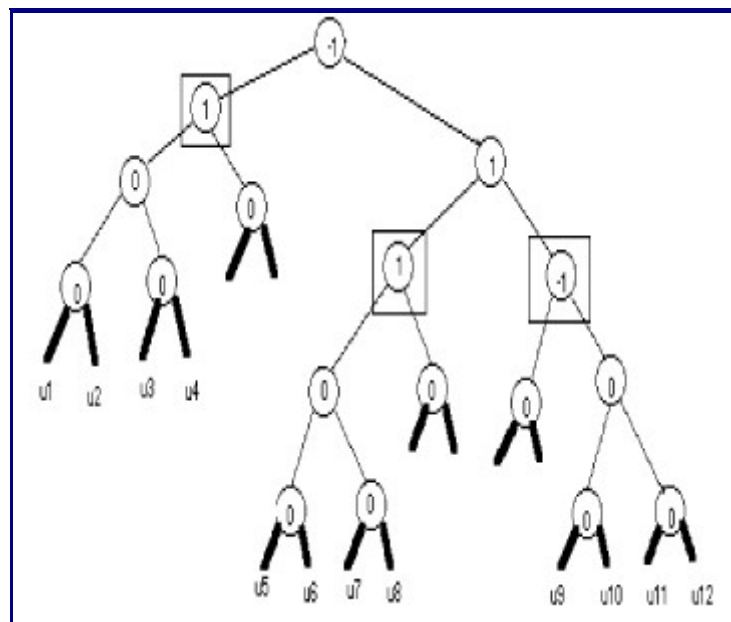


Définitions

Un *arbre binaire équilibré* (ou *arbre AVL*) est un arbre dans lequel les profondeurs des deux sous arbres de chaque noeud ne diffèrent pas plus d'un. A chaque noeud est associé un facteur d'équilibrage qui est égal à la différence entre la profondeur du sous arbre gauche et celle du sous arbre droit. Dans la figure suivante, la profondeur de chaque noeud est placée dans le noeud.



Quand on insère un élément, l'arbre peut devenir non équilibré. La figure qui suit illustre tous les cas possibles d'insertions. Les **ui** désignent les cas où **l'arbre se déséquilibre**.



Il est facile de voir que l'arbre devient non équilibré quand le nouveau noeud inséré :

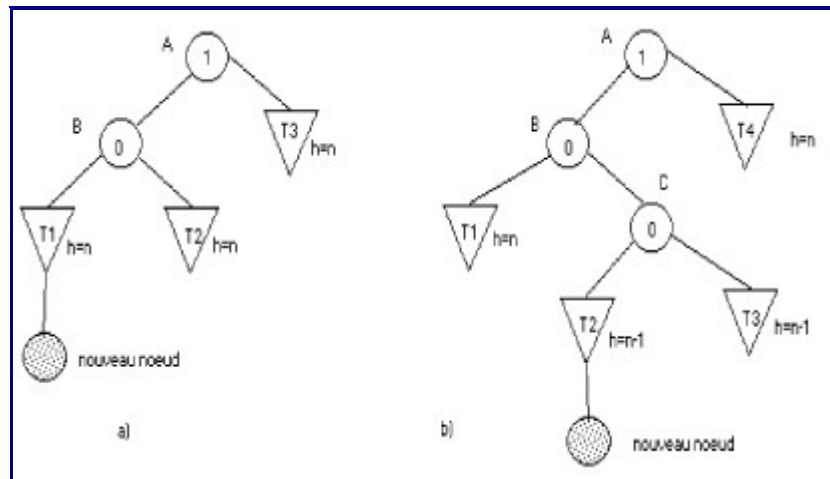
- est un descendant gauche d'un noeud qui avait un facteur d'équilibrage égal à 1 (u1 à u8)
- est un descendant droit d'un noeud qui avait un facteur d'équilibrage égal à -1 (u9 à u12).

Le plus jeune antécédent qui devient non équilibré est représenté dans un carré.



Techniques d'équilibrage

Examinons un sous arbre de racine le plus jeune antécédent qui devient non équilibré suite à une insertion. Prenons le cas où le facteur d'équilibrage est 1 pour ce jeune antécédent. La figure suivante illustre ce cas.



A désigne le plus jeune antécédent devenu non équilibré. Puisque $f(A)=1$, son sous arbre gauche est non NIL. Soit donc B le fils gauche. $f(B)$ doit donc avoir la valeur 0. Ce noeud B devait avoir avant l'insertion des sous arbres (droit et gauche) la même profondeur, soit $h=n$. Puisque $f(A)=1$, le sous arbre de A doit aussi avoir $h=n$.

➡ Deux cas sont à considérer figure a) et b) :

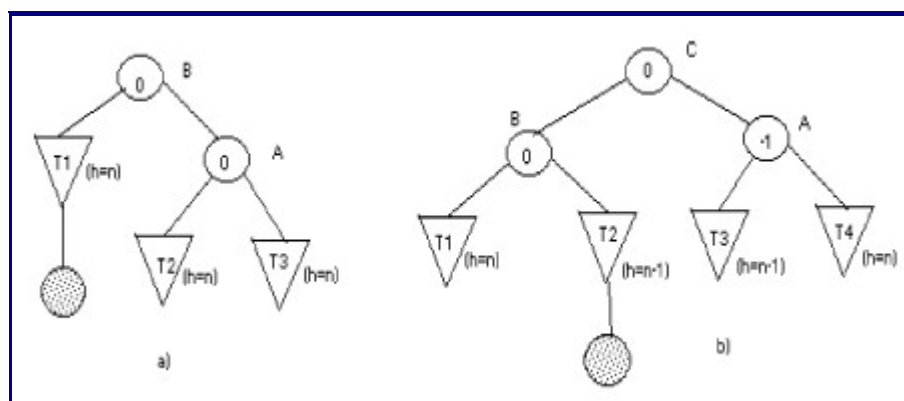
Dans a), le nouveau noeud est inséré dans le sous arbre gauche de B. Donc $f(B)$ devient 1 et $f(A)$ devient 2.

Dans b), le nouveau noeud est inséré dans le sous arbre droit de B. $f(B)$ devient -1 et $f(A)$ devient 2.

Afin de maintenir l'arbre binaire équilibré, il est nécessaire de transformer l'arbre de telle sorte que :

- (i) -l'inordre soit préservé.
- (ii) -l'arbre transformé soit équilibré.

1. Si une rotation droite est faite sur le sous arbre de racine A dans la figure a), on obtient l'arbre a) suivant qui respecte (i) et (ii).



2. Considérons la figure b) dans laquelle le nouveau noeud est inséré dans le sous arbre droit de B. Soit C le fils droit de B. Trois cas peuvent se présenter:

- C est le nouveau noeud créé.
- le noeud créé est dans le sous arbre gauche de C.
- le noeud créé est dans le sous arbre droit de C.

La figure b) illustre le cas où le nouveau noeud est dans le sous arbre gauche de C. Si on fait une rotation gauche du sous arbre de racine B suivie par une rotation droite du sous arbre de racine A on obtient l'arbre b) qui respecte (i) et(ii).

Le raisonnement est le même pour le cas où le facteur d'équilibrage est -1.



Algorithme d'insertion

La première partie de l'algorithme consiste à insérer la clé dans l'arbre sans tenir compte du facteur d'équilibrage. Elle garde aussi la trace du plus jeune antécédent, soit Y qui devient non équilibré. La deuxième partie fait la transformation à partir de Y.



Analyse théorique

On démontre que *la profondeur maximale d'un arbre binaire équilibré est $1.44 * \log_2 n$* . La recherche dans un tel arbre n'exige jamais plus de 44% de plus de comparaisons que pour un arbre binaire complet. Pour n grand, l'arbre de recherche binaire équilibré se comporte bien avec *un temps de recherche égal $\log_2(n) + 0.25$* . En moyenne une rotation est faite pour 46.5% des insertions.

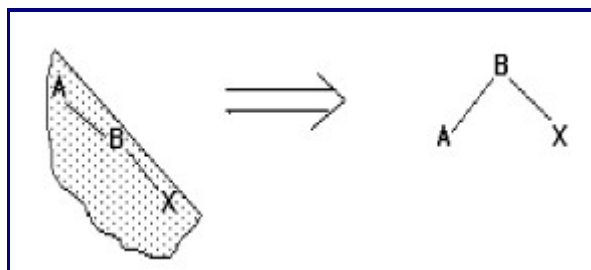
En guise de conclusion de cette section, nous illustrons, ci-après, le mécanisme de construction.



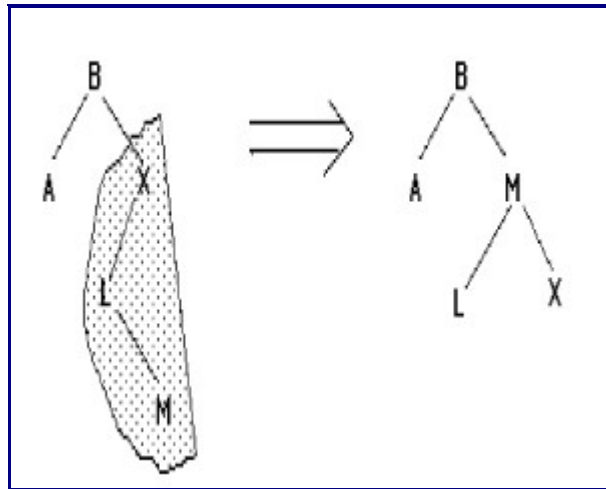
Exemple d'insertions de données dans un arbre AVL

Insérons la séquence : A, B, X, L, M, C, D, E, H, R, S, F dans un arbre AVL.

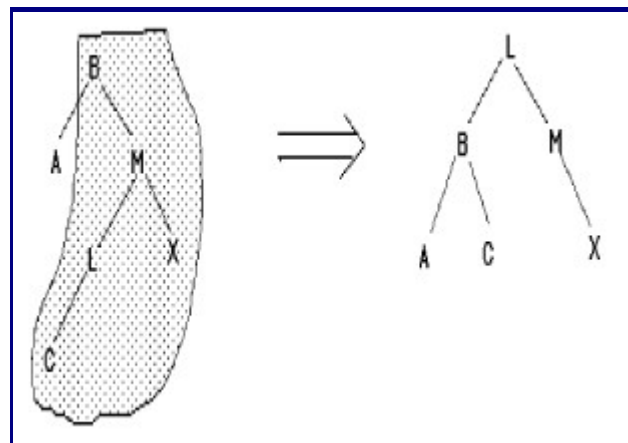
Insertion A, B, X



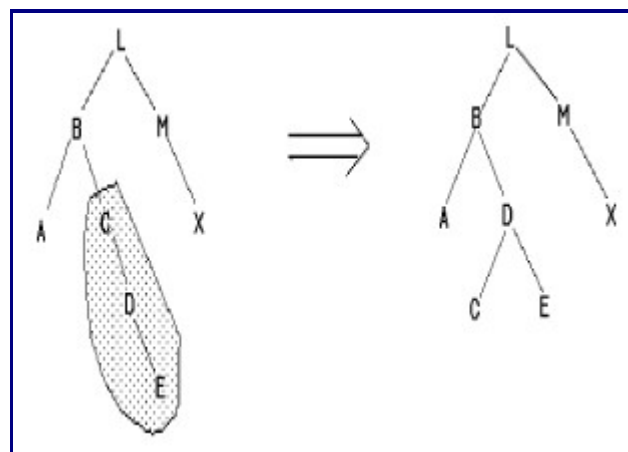
Insertion de L, M



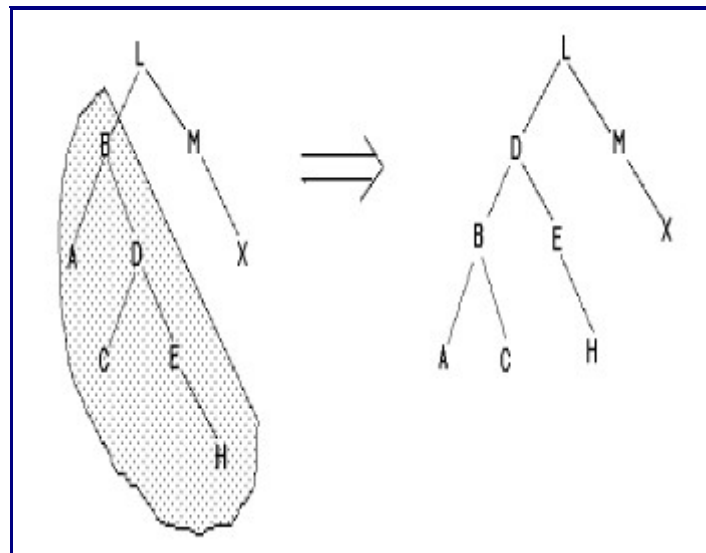
Insertion de C



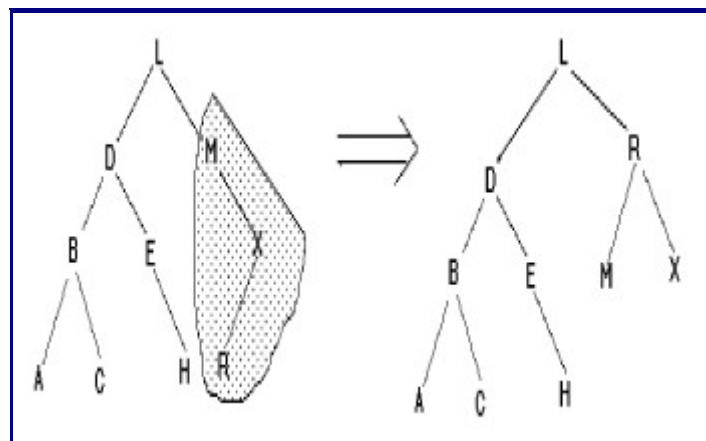
Insertion de D, E



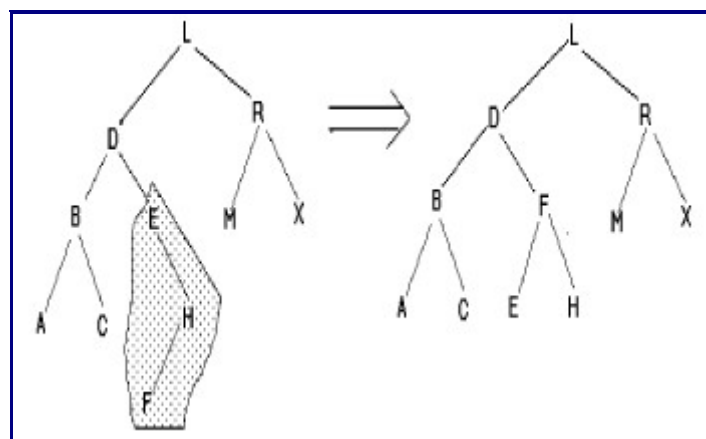
Insertion de H



Insertion de R



Insertion de F



Arbres de recherche m-aires

Définitions

Modèle

Opérations

Utilisation d'un arbre de recherche m-aires comme méthode d'accès

Implémentation

ISAM: Indexed Sequential Method (de IBM)



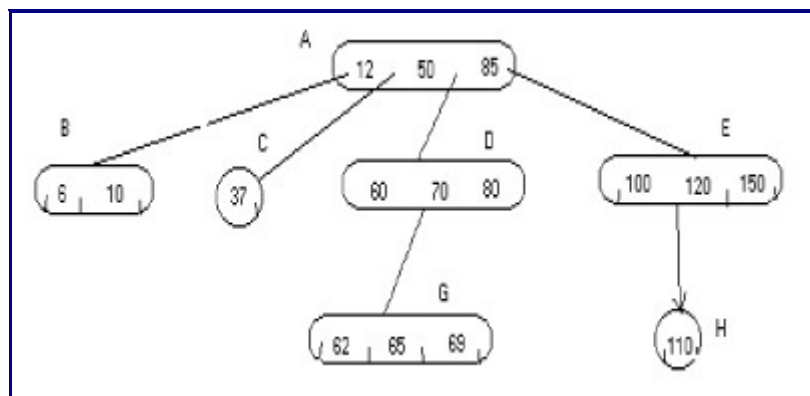
Définitions

Un *arbre de recherche m-aire* peut être défini comme une généralisation de l'arbre de recherche binaire. Au lieu d'avoir une clé et deux pointeurs (cas d'un arbre de recherche binaire), on a $(n-1)$ clés et n pointeurs. Un arbre de recherche m-aire d'ordre n est un arbre dans lequel chaque noeud peut avoir n fils.

⇒ Si s_1, s_2, \dots, s_n sont les n sous arbres issus d'un noeud donné avec les clés k_1, k_2, \dots, k_{n-1} dans l'ordre ascendant, alors :

- toutes les clés dans s_1 sont $< k_1$
- toutes les clés dans s_j ($j=2,3, \dots, n-1$) sont $> k_{j-1}$ et $< k_j$
- toutes les clés dans s_n sont $> k_{n-1}$.

La figure qui suit illustre un arbre de recherche m-aire d'ordre 4. les noeuds **A, D, E et G** contiennent le maximum de sous arbres : 4. On dit qu'ils sont *remplis ou complets*.



Un arbre de recherche m-aire "**Top-Down**" est tel que tout noeud non rempli est une feuille. L'arbre ci-dessus est "**Top-Down**" car les noeuds non remplis sont des feuilles.

⇒ Un arbre de recherche m-aire est dit *équilibré* si tous *ses feuilles sont au même niveau*.



Modèle

On peut définir les opérations suivantes sur les arbres de recherche m-aires :

Nbr(P) :	accès au nombre de sous arbres du noeud P
Fils (P,1), Fils(P, 2),,Fils(P, Nbr(P)) :	accès aux différents fils.
Cle(P, 1), Clé(P,2),Clé(P, Nbr(P)-1) :	accès aux différentes clés
Aff_nbr(P, n) :	modifier le champ Nbr par la valeur n
Aff_fils(P, i, j) :	faire de j le i-ième fils du noeud P
Aff_Clé(P, i, clef) :	faire de clef la i-ième clé du noeud P
Allouer(P) :	allocation d'un noeud
Liberer(P) :	libération d'un noeud

Le sous arbre pointé Par fils(P, i) contient toutes les clés entre Clé(P, i-1) et Clé(P,i).

Fils(P,1) Pointe le sous arbre des clés inférieures à Clé(P, 1).

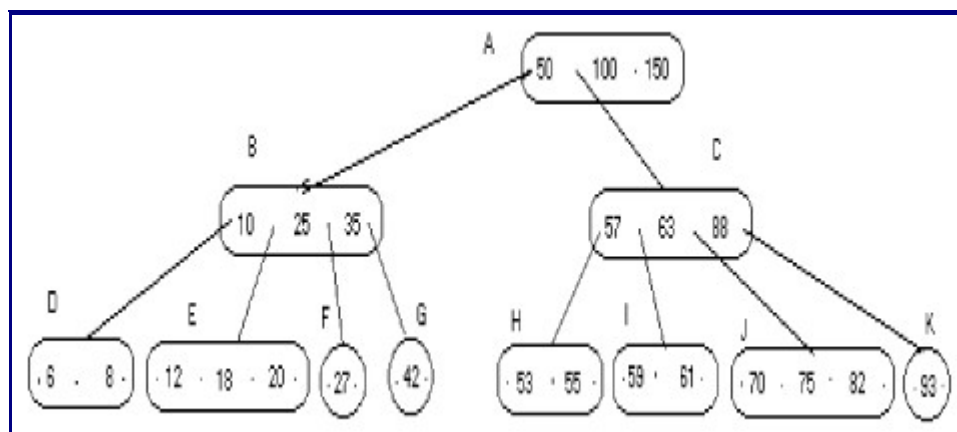
Fils(P, Nbr(P)) pointe le sous arbre contenant toutes les clés supérieures à Clé(P,nbr(P)-1).



Opérations

◆ Recherche

L'algorithme de recherche est défini comme une généralisation de l'arbre de recherche binaire. Dans la figure suivante :



si nous recherchons les clés 63, 71, 22 et 41, nous obtenons les résultats suivants :

a)
Clé=63
Noeud=C
Position=2

b)
Clé=71
Noeud=J
Position=2

c)
Clé=22
Noeud=E
Position=4

d)
Clé=41
Noeud=G
Position=1

Trouv=Vrai

Trouv=Faux

Trouv=Faux

Trouv=Faux

avec :

Clé : clé à rechercher.

Noeud : le noeud qui contient ou qui devrait contenir Clé.

Position : la position de la clé dans le noeud.

Trouv : existence de Clé.

◆ Insertion

La construction est de haut en bas comme les arbres de recherche binaire et elle se fait en deux étapes . Rappelons que les clés dupliquées ne sont pas permises.

➡ *Première étape* : consiste à rechercher la clé.

➡ *Deuxième étape* : Si la clé n'est pas trouvée, on est sur un noeud feuille, soit P.

Si celui-ci n'est pas complet (c'est à dire $Nbr(P) < n$) on l'insère.

Si celui-ci est complet, on fait les opérations suivantes :

- allocation d'un nouveau noeud.
- insérer la clé(l'article)dans ce noeud.
- placer le noeud comme fils du noeud pointé par Noeud.

L'arbre ainsi construit dépend de l'ordre d'arrivée des clés. Il peut être très déséquilibré et donc beaucoup d'espace peut être perdu. Cette technique ne garantit donc pas l'équilibrage bien que l'arbre est "Top-Down", c'est à dire qu'un noeud n'est créé que si son noeud père est rempli.

◆ Suppression

La suppression peut être logique ou physique.

Dans le premier cas, il suffit tout simplement de laisser la clé au niveau du noeud et la marquer.

(implémentation : pointeur à l'article égal à NIL ou carrément utilisation d'un "flag"). La clé reste dans le noeud et est utilisée pour l'algorithme de recherche mais seulement elle ne représente pas une clé du fichier. Le grand inconvénient c'est quand il ya beaucoup de suppressions (un espace considérable est perdu). Si une clé supprimée est réinsérée de nouveau, le même espace est réutilisé.

Dans le second cas, la technique est similaire à celle des arbres de recherche binaire, c'est à dire :

- si la clé à supprimer a un sous arbre gauche ou droit vide, alors simplement supprimer la clé et tasser le noeud. Si c'est la seule clé dans le noeud, libérer le noeud.
- si la clé à supprimer a des sous arbres gauche et droit tous les deux non vides, alors trouver la clé successeur (qui doit avoir un sous arbre gauche vide). Remplacer la clé à supprimer par ce successeur et tasser le noeud qui contenait ce successeur. Si le successeur est la seule clé dans le noeud, libérer le noeud.



Ces deux techniques ne garantissent pas le maintien de l'arbre "Top-Down" même si avant la suppression ces qualités existaient. Il faut donc trouver d'autres techniques pour préserver de telles qualités.

◆ Parcours

L'algorithme récursif suivant parcourt un arbre de recherche m-aire et liste les éléments dans l'ordre ascendant :

Traverse (Arbre) :

```

SI Arbre <> NIL :
  NT := Nbr(Arbre)
  POUR I :=1, Nt-1
    Traverse(Fils(Arbre, I))
    ECRIRE(Clé(Arbre,I))
  FINPOUR
  Traverse(Fils(Arbre,Nt))
FSI

```



Utilisation d'un arbre de recherche m-aire comme méthode d'accès

L'arbre de recherche m-aire est surtout utilisé pour le stockage en mémoire externe(disque). Un noeud de l'arbre est alors une page sur le disque. A un moment donné, seule une page est ramenée en mémoire centrale. L'utilisation de ces sortes d'arbre pour le stockage nécessite des buffets de taille très grande(ordre de 200). Cette taille est cependant limitée par le système.



Implémentation

L'arbre de recherche m-aire est implémenté par l'utilisation des fils explicites de chaque noeud au lieu de la représentation comme arbre binaire(liste de frères). La raison est que pour un tel arbre le nombre de fils est limité (ordre) et un noeud est toujours bien rempli.

Les articles peuvent être rangés de deux façons :

- (i) - **dans les noeuds avec les clés,**
- (ii) - **séparément. On utilise alors un pointeur vers l'information. L'arbre de recherche m-aire joue alors le rôle d'un index.**

Si le fichier est grand on préfère de loin utiliser (ii) même si on fait une lecture supplémentaire pour retrouver l'article.



I S A M : Indexed Sequential Access Method (de IBM)

On peut définir le séquentiel indexé comme suit :

- a) pour localiser un article on utilise un index.
- b) pour avoir le suivant, on ne repasse pas par l'index.

Nous décrivons ci-après **la méthode ISAM**. Nous verrons plus loin une autre forme du séquentiel indexé.

ISAM utilise *un arbre de recherche m-aire*. De plus, ISAM est basée sur les caractéristiques physiques du disque. Il existe 3 zones pour le fichier :

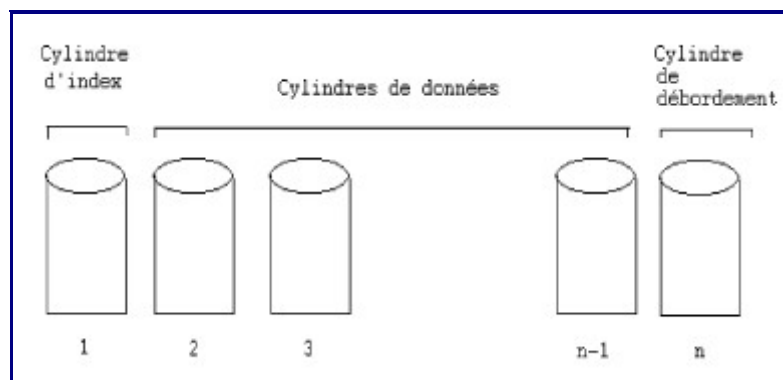
- *une zone d'index,*
- *une zone de données,*
- *zone de débordements.*

⇒ Il y a 2 niveaux d'index :

- *un index de cylindres*
- *des indexes de surfaces*

◆ Structure du fichier

Le fichier est un ensemble de cylindres numérotés logiquement 1, 2, ... ,n. Le premier cylindre est réservé pour l'index. Les cylindres 2 à n-1 sont destinés aux données. Le cylindre n contiendra les débordements. Schématiquement, on a la figure suivante :



◆ Structure d'un cylindre d'index

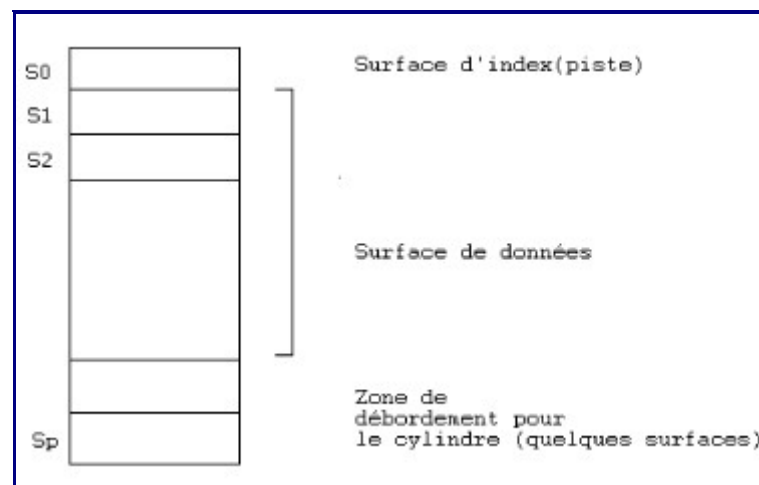
Le cylindre d'index est une table de couples (Clei, Ci) avec Cléi, la plus grande clé dans le cylindre Ci.

Clé1	C1
Clé2	C2
⋮	
Clén	Cn

◆ Structure d'un cylindre de données

Un cylindre de données est un ensemble de surfaces divisé comme suit :

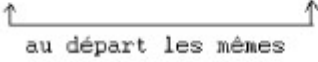
- **une surface d'index,**
- **des surfaces de données,**
- **quelques surfaces pour les débordements du cylindre.**



◆ Structure de la surface d'index

Elle a la structure suivante :

Entrée primaire		Entrée débordement	
Clé 1	Surface 1	Clé 1	Adresse 1
Clé 2	Surface 2		



 au départ les mêmes

Pour les entrées primaires, Cléi désigne la plus grande clé dans la surface surfaci. Pour les entrées débordements, Cléi désigne la plus grande clé dans la liste des débordements. Une adresse est le couple (n° surface, n° article).

◆ Structure de la surface de données

C'est un tableau ordonné d'articles. Chaque article contient la clé et l'information.

Art1	Art2
------	------	-------

◆ Structure de la surface de débordements

Art	adr	Art	adr
-----	-----	-----	-----	-------

Les débordements sont chaînés. Une adresse est composée de 3 champs :

- C : numéro de cylindre
- S : numéro de surface
- r : position de l'article

◆ Structure du cylindre de débordements

C'est un tableau de couples (Art, Adr). Art désigne l'article et Adr l'adresse du prochain article. Comme précédemment l'adresse est composée des trois champs C, S et r.

Art	Adr
Art	Adr

◆ Opérations

Chargement initial :

cette opération consiste à créer le fichier à partir de données triées selon leur clé. Les surfaces de données ne sont pas remplies à 100 %. Les zones de débordements sont vides.

Recherche :

une recherche d'article commence par la consultation de l'index des cylindres afin de sélectionner le cylindre de données. Ensuite la recherche continue au niveau de l'index des surfaces. Si la clé n'est pas dans son adresse primaire, la liste de débordements est consultée.

Insertion :

si l'article n'existe pas, il peut être inséré :

- a) dans une surface de données si la place est disponible,
- b) dans la zone de débordements du même cylindre,
- c) dans le cylindre de débordements.

Suppression :

la suppression est logique au niveau de l'article lui-même.

Réorganisation :

périodiquement, une réorganisation est nécessaire afin d'éliminer les débordements et pour récupérer les articles effacés logiquement.



Les arbres B

Une introduction aux arbres B

Définition

Illustration du mécanisme d'insertion

Techniques de suppression

Amélioration des Arbres B

S I S : Scope Indexed Sequential (de CDC)



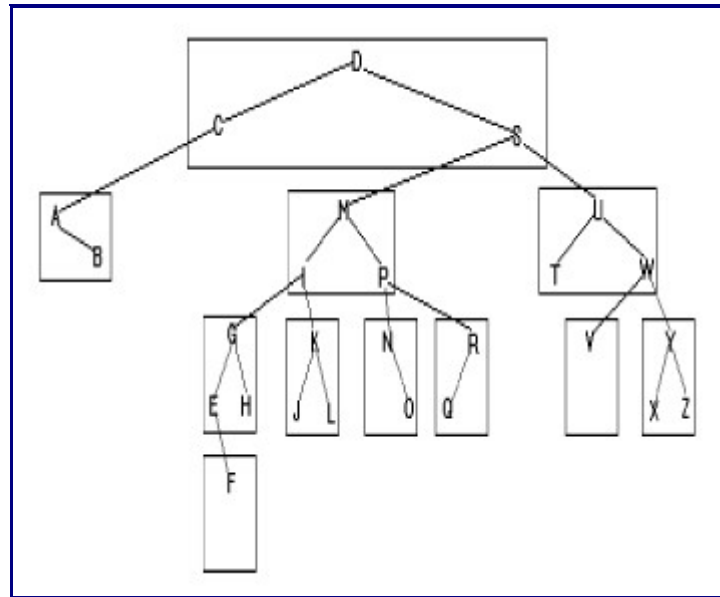
Une introduction aux arbres B

Le problème avec les arbres de recherche m-aires est celui du maintien de l'équilibre de l'arbre paginé. Si l'ensemble des clés est connu à l'avance c'est simple. Le problème est compliqué si les clés arrivent de façon aléatoire. On peut garder les clés ordonnées à l'intérieur de la page (arbre AVL) mais il s'avère très dur de garder l'arbre de pages équilibré : l'équilibrage avec le maintien de l'ordre n'est pas évident. Essayer

Insérons la séquence de clés suivantes dans un arbre AVL paginé :

C S D T A M P I B W N G U R K E H O L J Y Q Z F X V

Nous obtenons l'arbre paginé suivant :



Problèmes :

1. Comment être sûr que les clés contenues dans la page racine sont dans leurs bonnes positions ?
2. comment garder l'arbre de pages équilibré ?

Bayer et McCreight ont fourni une solution à ces problèmes par l'invention *des arbres B*.



Définition

C'est une technique plus complexe mais garantit l'équilibrage de l'arbre(donc recherche plus rapide). En plus, chaque noeud de l'arbre(sauf la racine) est au moins rempli à moitié.(donc peu d'espace perdu). C'est la raison pour laquelle elle est la plus utilisée dans les systèmes de fichier actuels.

C'est donc *une construction Bottum-Up*, c'est à dire que l'arbre est construit de bas en haut. Par conséquent , l'équilibrage est garanti.

Les éclatements peuvent être en cascade.



Un arbre B d'ordre n est tel que :

- la racine a au moins 2 fils
- chaque noeud, qui n'est pas racine, a entre $n/2$ et n fils
- tous les noeuds feuilles sont au même niveau

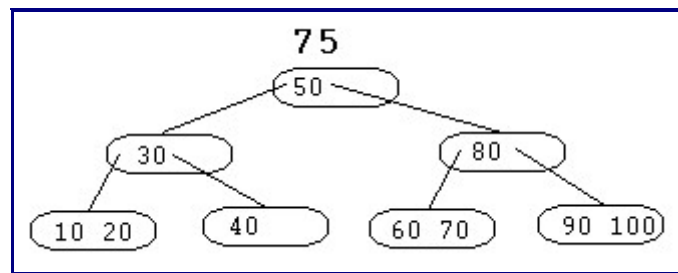
Remarque : B pour Bayer / Boeng / Balanced

Dans ce qui suit, nous illustrons d'abord le mécanisme de construction d'un arbre B, ensuite nous donnons les techniques de la suppression à l'aide d'exemples.

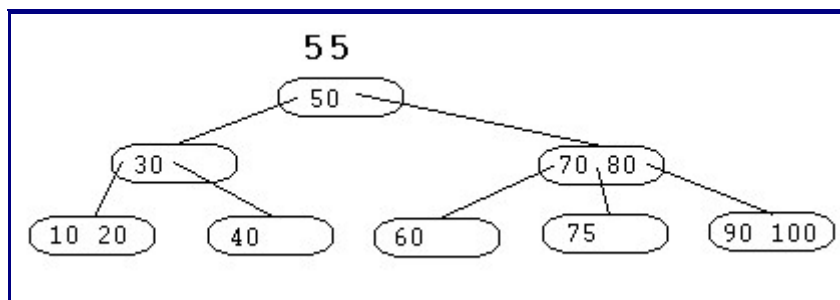


Illustration du mécanisme d'insertion

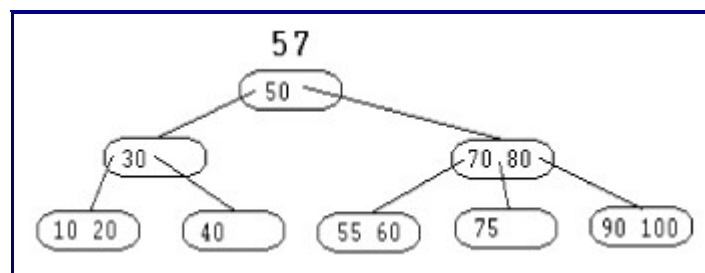
Considérons l'arbre B suivant d'ordre 3 (au maximum 2 clés et 3 pointeurs) :



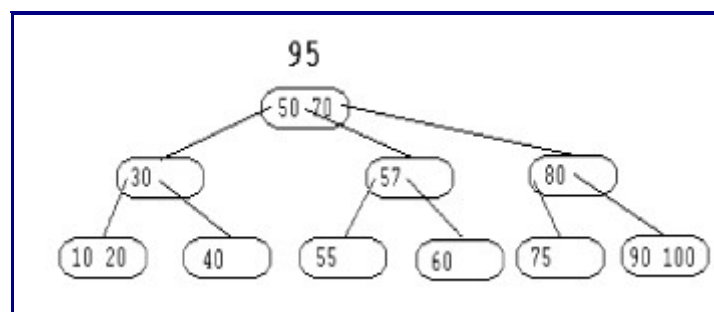
La clé 75 provoque l'éclatement du noeud (60 70). La clé du milieu, c'est à dire 70 est transférée dans le noeud père comme le montre la figure suivante :



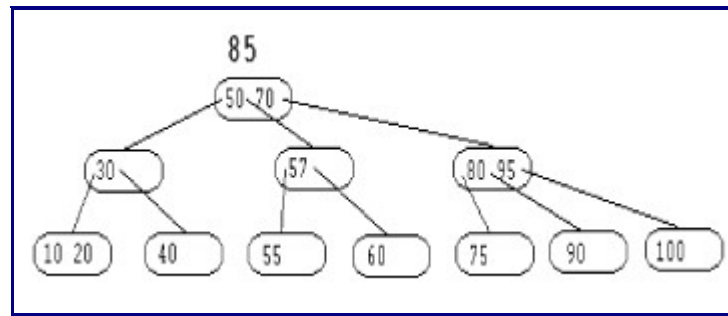
La clé 55 est insérée dans le noeud contenant la clé 60. L'arbre devient :



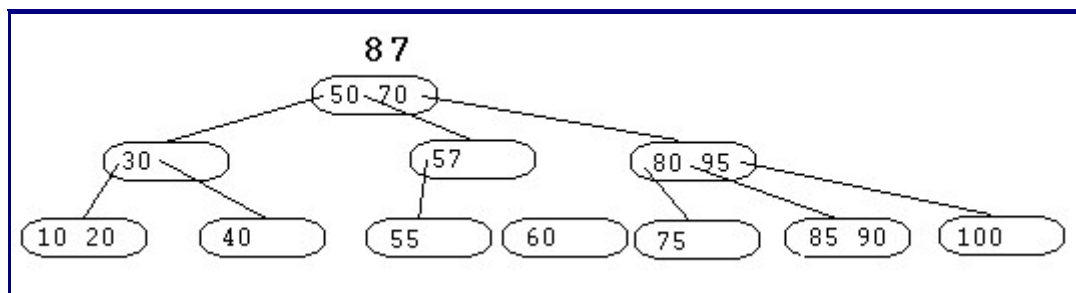
La clé 57 provoque une collision sur le noeud (55 60). Le noeud père (70 80) est à son tour éclaté. La clé 70 migre vers le noeud racine.



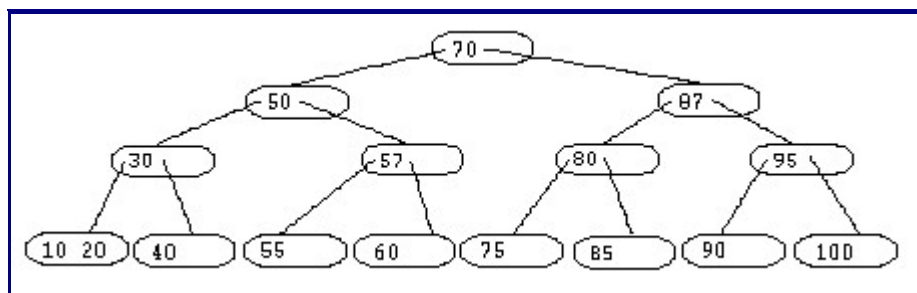
La clé 95 provoque l'éclatement du noeud (90 100). La clé 90 est donc déplacée dans le noeud contenant la clé 80 comme le montre la figure suivante :



La clé 85 est insérée dans le noeud contenant la clé 90. L'arbre devient :



La clé 87 provoque une collision sur le noeud (85 90). Le noeud (80 95) est éclaté à son tour. Le noeud racine est aussi éclaté et le niveau de l'arbre augmente. On obtient ainsi l'arbre suivant :



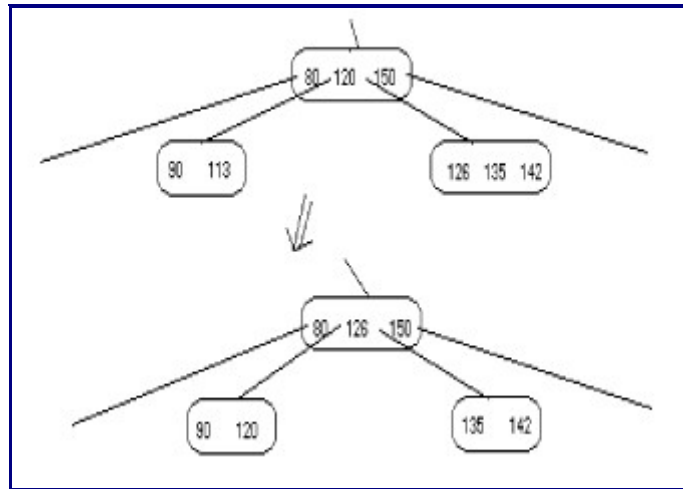
▲

Techniques de suppression

Il faut supprimer l'article tout en préservant la qualité de l'arbre B, c'est à dire en gardant *au moins $n \div 2$ clés dans le noeud.*

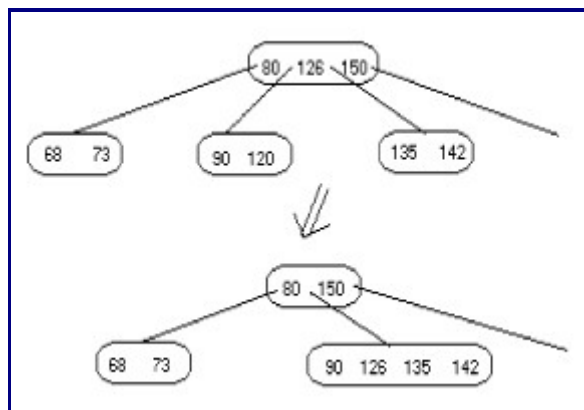
C'est le cas de la suppression physique dans un arbre de recherche m-aire. En plus, si le noeud feuille qui contenait le successeur a moins de $n \div 2$ clés l'action suivante est entreprise :

➡ Si l'un des frères(gauche ou droit) contient plus de $n \div 2$ clés, alors la clé, soit K_s , dans le noeud père qui sépare entre les deux frères est ajoutée au noeud "*underflow*" et la dernière ou la première clé(première si frère droit, dernière si frère gauche) est ajoutée au père à la place de K_s . La figure suivante illustre le principe



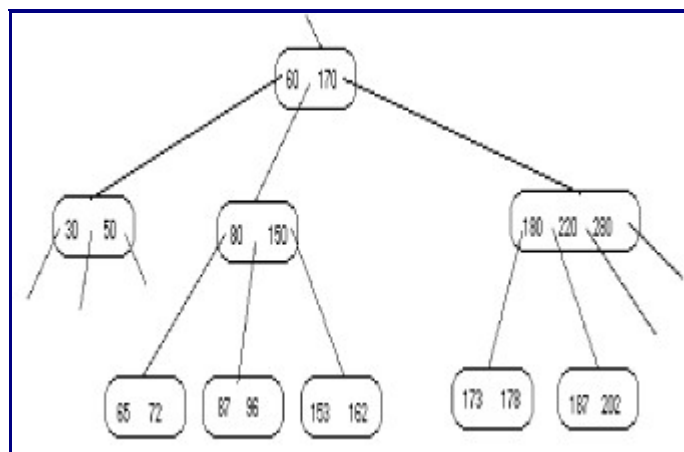
Suppression de la clé 113

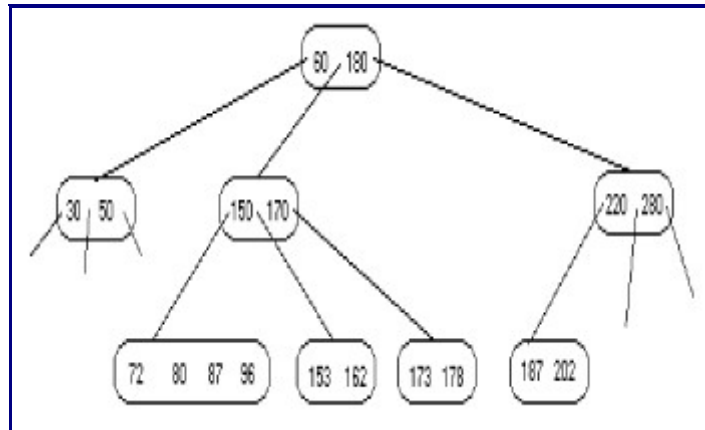
➡ Si les deux frères contenaient exactement $n \div 2$ clés, le noeud "*underflow*" et l'un de ses frères seront concaténés (fusionnés ou consolidés) en un seul noeud qui contient aussi la clé séparatrice de leur père. La figure suivante illustre le principe :



Suppression de la clé 120 et concaténation

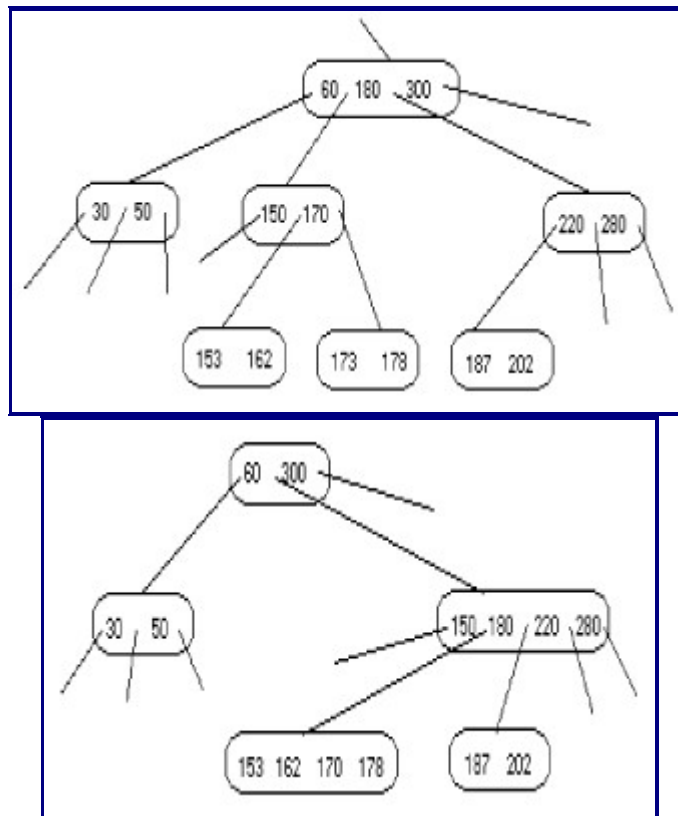
Il est aussi possible que le père contienne seulement $n \div 2$ clés et par conséquent il n'a pas de clé à donner. Dans ce cas, il peut emprunter de son père et frère comme dans la figure suivante :





Suppression de 65, concaténation et emprunt

Dans le pire des cas, quand les frères du père n'ont pas de clés à donner, le père et son frère peuvent aussi être concaténés et une clé est prise du grand père. C'est le cas de la figure suivante :



Suppression de 173

➡ Si tous les antécédents d'un noeud et leurs frères contiennent exactement $n \div 2$ clés, une clé sera prise de la racine (due aux concaténations en cascades). Si la racine avait plus d'une clé, ceci termine le processus.

➡ Si la racine contenait une seule clé, elle sera utilisée dans la concaténation. Le noeud racine est libéré et le niveau de l'arbre est réduit d'une unité.



Amélioration des Arbres B

[Arbres B*](#)

[Arbres B préfixés](#)

[Arbres B+](#)



[Arbres B*](#)

Une façon d'améliorer le chargement des arbres B consiste *à retarder la division quand un noeud est plein*. On fait alors une redistribution équitable des clés contenues dans le noeud en question et l'un de ses frères. Si le noeud et son frère sont tous les deux pleins, les deux noeuds sont divisés en 3. Ceci garantit un minimum de chargement à 67%. Un tel arbre B est appelé **arbre B***.



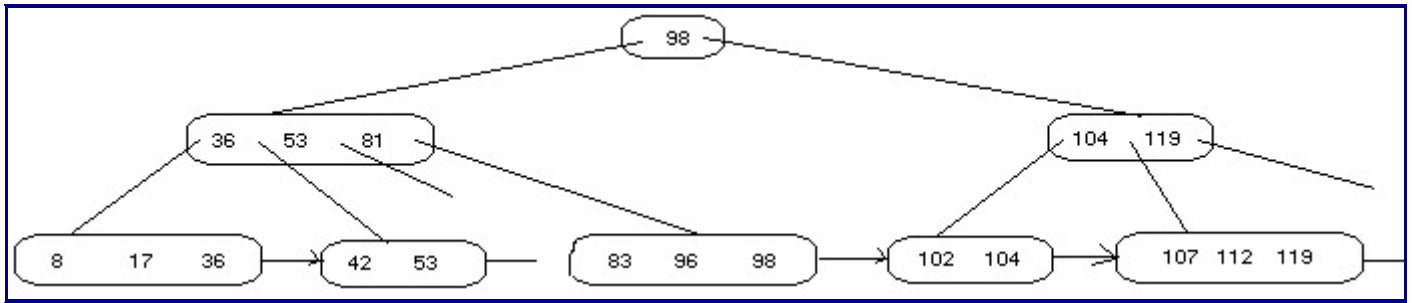
[Arbres B préfixés](#)

Au lieu d'utiliser les clés intégralement on en utilise que des parties. C'est ce qu'on appelle des séparateurs. On ne peut utiliser la recherche binaire à cause de la longueur variable des séparateurs au niveau des noeuds. L'avantage des arbres B préfixés est que *la profondeur diminue et donc l'accès est meilleur*.



[Arbres B+](#)

Dans un arbre B+ *toutes les clés sont maintenues au niveau des feuilles*. De plus, les clés sont dupliquées dans les noeuds non feuilles. Les articles(ou les pointeurs vers les articles) sont au niveau des feuilles. Les noeuds feuilles sont chaînés.



La liste liée des noeuds feuille est appelée ensemble des séquences.

La recherche ne s'arrête pas quand la clé est trouvée comme dans le cas des arbres B. La recherche se termine donc toujours au niveau d'un noeud feuille (le signe < est remplacé par le signe <=).

C'est une généralisation du séquentiel indexé vu précédemment. Chaque niveau de l'arbre est un index au niveau suivant et le dernier niveau, l'ensemble des séquences, est un index au fichier lui-même.

L'insertion dans un arbre B+ est similaire à celle d'un arbre B sauf que le noeud de division est retenu dans le sous arbre gauche (et bien sûr transféré dans le noeud père).

Quand une clé est supprimée, elle peut être retenue dans les noeuds non feuilles puisqu'elle reste un séparateur entre les clés dans les noeuds plus bas.

L'arbre B+ garde l'efficacité des opérations de recherche et d'insertion des arbres B mais améliore beaucoup l'efficacité de la recherche du suivant ($O(\log(n))$ pour les arbres B et $O(1)$ pour les arbres B+)



SIS : Scope Indexed Sequential (de CDC)

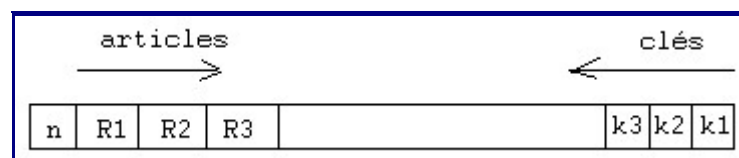
Nous décrivons, dans ce qui suit, la méthode d'accès SIS développée par Control Data Corporation.

On a deux ensembles de bloc :

- un ensemble de blocs de données
- un ensemble de blocs d'index

Le bloc est l'unité de transfert entre la RAM et la mémoire secondaire.

Format d'un bloc de données



L'espace libre est toujours au milieu.

L'article est de longueur fixe ou variable. Si c'est variable, la clé est de longueur fixe.

Format d'un bloc d'index

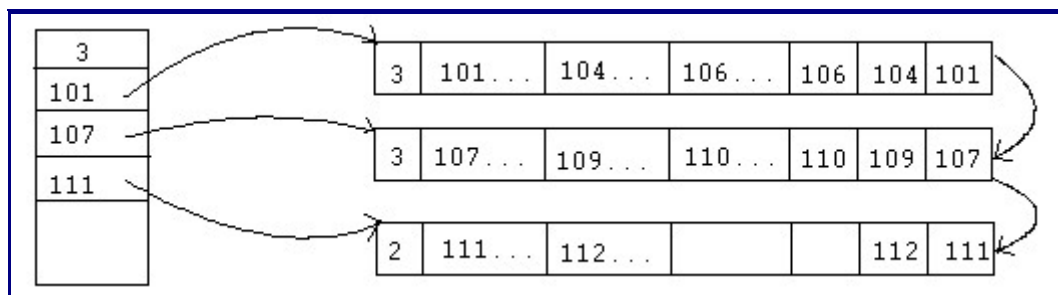
Nombre	
Clé1	Adr1
Clé2	Adr2
Clé3	Adr3
Cléi	Adri
Espace libre	

Les blocs de données et d'index sont organisés dans un arbre B+. Cléi désigne la plus petite valeur dans le bloc d'adresse Adri. Les blocs de données sont chaînés (pour fournir l'accès séquentiel).L'expansion se fait par l'éclatement des noeuds.

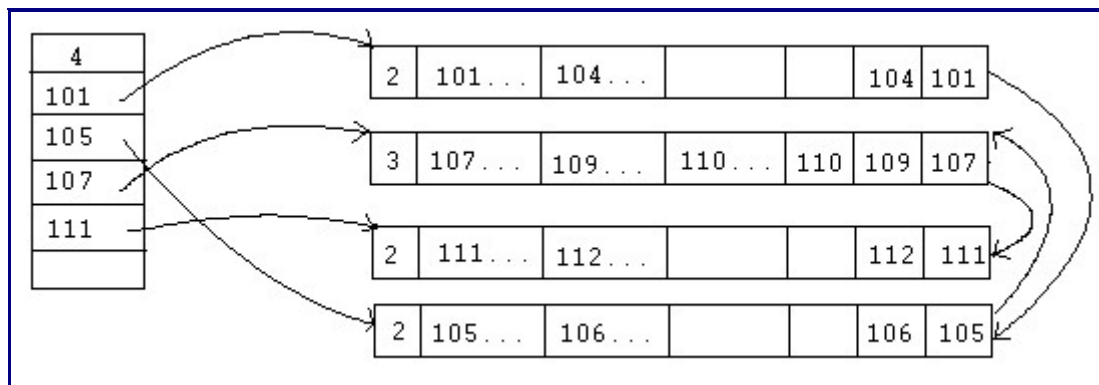
Exemple

Montrons l'évolution de l'index et des blocs de données à partir d'un exemple.

Etat initial :

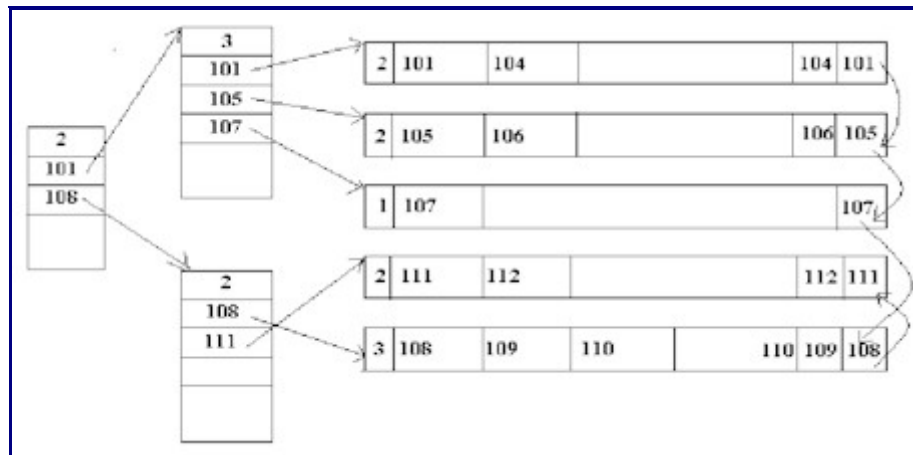


Insertion de 105 :



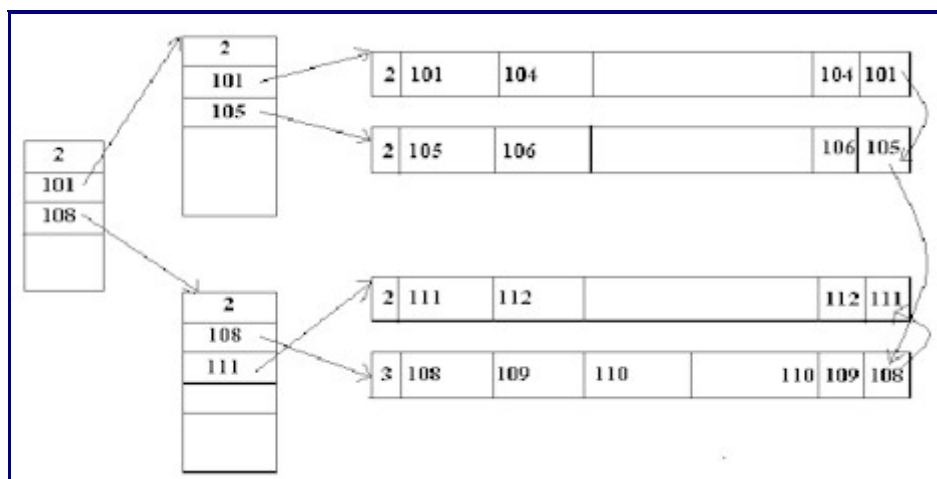
Il y a éclatement du bloc de données en deux.

Insertion de 108



Il y a éclatement du bloc en 2. De même, il y a éclatement du bloc d'index en 2.
 Le fichier est dynamique, donc sans réorganisation future. On note ainsi l'absence des débordements.
 Les blocs de données et d'index ne sont pas nécessairement sur des positions contiguës.

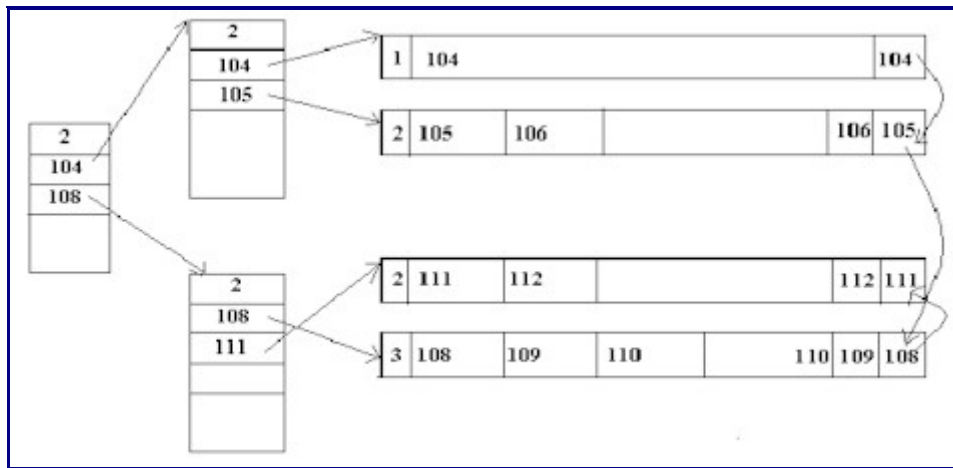
Suppression de 107



Le bloc de données contenant 107 est libéré. Il est rangé dans une liste linéaire chaînée de blocs libérés pour utilisation ultérieure.

Suppression de 101

Il faut mettre 104 à la place de 101 au niveau de l'index.



TRAVAUX DIRIGES

1. Retrouver les algorithmes de recherche, insertion, suppression, requête à intervalle et de parcours en ordre croissant dans un fichier structuré en arbre de recherche m-aire. On considère le cas où les articles sont rangés :

- avec les clés
- séparément

On examine le cas où les articles sont de longueur :

- fixe
- variable

NB. Pour l'algorithme de recherche utiliser la fonction `Rechnoeud (P, Clef)` qui retourne soit le plus petit indice J tel que $Clef \leq Clé(P, J)$ ou $Nbr(P)$ si clé est supérieur à toutes les clés dans $Nœud(P)$.

Pour l'algorithme d'insertion on peut utiliser :

**** La procédure Trouver** qui a comme paramètre

En entrée : Arbre et Clef

En sortie

Si la clé est trouvée

Rech : pointe le nœud contenant la clé trouvée

Position : contient la position de la clé à l'intérieur du nœud.

Si la clé n'est pas trouvée

Rech : pointe la feuille qui pourrait contenir la clé.

Position : contient alors l'index de la plus petite clé dans le nœud pointé par Rech qui est supérieure à la clé recherchée Clef. Si toutes les clés dans le nœud pointé par Rech sont inférieures à Clef, Position est mis $Nbr(Rech)$.

Une variable Trouv est mise à VRAI ou à FAUX selon l'appartenance ou non de la clé.

**** La procédure Insérercle (Rech, Clef, Position)** qui insère la clé Clef à la position Position dans le nœu Rech.

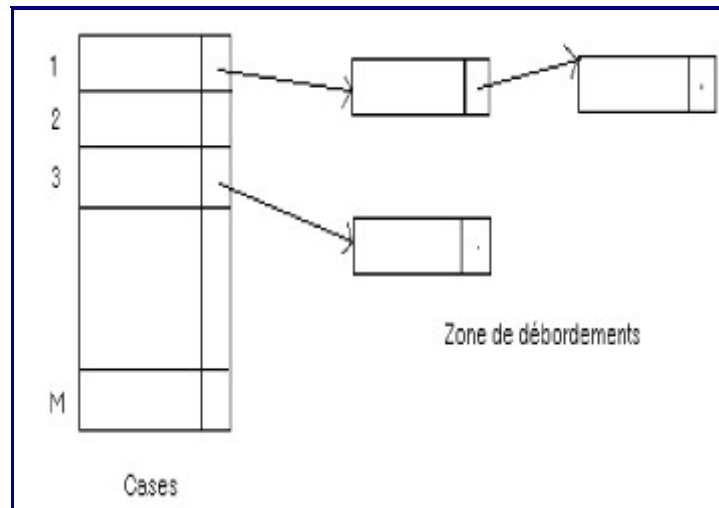
2. ISAM : Retrouver les algorithmes de recherche, insertion, suppression et de requête à intervalle pour la méthode d'accès ISAM présentée en cours.
3. Etudier et développer les algorithmes de recherche, insertion et de suppression dans un arbre AVL.
4. Retrouver les algorithmes de recherche, insertion et de suppression dans un fichier structuré en B-arbre.
5. SIS : Retrouver les algorithmes de recherche, insertion, suppression et de requête à intervalle pour la méthode d'accès SIS présentée en cours.

Introduction

Les techniques du hachage sont bien utilisées pour l'accès au fichier se trouvant sur disque. L'objectif principal est *de minimiser le nombre d'accès au fichier*. Le fichier est généralement divisé en M blocs contenant chacun b articles. Une collision apparaît quand plus de b articles ont la même adresse de bloc.

Les deux approches suivantes de résolution de collisions semblent les meilleures :

Chaînage avec des listes séparées



Si plus de b articles arrivent sur le même bloc, un lien à un article de débordement est inséré à la fin du premier bloc. Ces articles en débordement sont rangés dans une zone spéciale appelée *zone de débordements*. Il n'est pas avantageux de garder les débordements. Comme les débordements sont liés entre eux l'accès (au $b+k$) article exige $(1+K)$ accès disque.



C'est généralement une bonne idée de garder l'espace des débordements dans chaque cylindre du disque, pour que la plupart des accès se fassent dans le même cylindre.



Bien que cette façon de garder les débordements semble inefficace, le nombre de débordements est statistiquement assez petit que le temps de recherche moyen soit très bon.



Adressage ouvert

Comme pour les mémoires internes on peut procéder sans chaînage en utilisant une méthode ouverte. L'essai *linéaire* est probablement meilleur que l'essai *aléatoire (double hachage)* pour les fichiers externes, parce que l'incrément c peut souvent être choisi afin de minimiser le délai entre les accès disque.



L'essai linéaire donne des résultats satisfaisant à moins que le fichier devient très rempli. Pour un facteur de chargement de 90% et une capacité de case de 50, le nombre d'accès moyen pour une recherche avec succès est seulement de 1.04. Ceci est meilleur que 1.08 exigé par la méthode de chaînage avec la même taille de case.



Avantages et inconvénients

Les méthodes de hachage n'exigent pratiquement pas de mémoire pour l'index. Elles facilitent *l'insertion* et la *suppression* des données et permettent de *trouver une donnée avec un temps d'accès très faible*, généralement inférieur à 2.

Elles sont efficaces pour les fichiers statiques. Le plus grand *inconvénient* de ces méthodes est que *les fichiers construits sont désordonnés*.



TRAVAUX DIRIGES

1. Développer les algorithmes de recherche, insertion et suppression pour les méthodes de hachage suivantes appliquées aux fichiers :

- essai linéaire
- double hachage
- chaînage interne
- chaînage séparé

On écrira également le module d'initialisation.

La troisième et quatrième parties sont consacrées à la présentation d'un grand nombre d'exercices programmés dans *les langages procéduraux PASCAL et C*. Ces programmes constituent une sorte de manuel de référence pour les langages en question. En plus, la plupart des programmes existent en PASCAL et en C, ce qui permet de *voir comment passer d'un langage à un autre*. Nous avons bien commenté les programmes afin de les rendre plus clairs.



Nous avons également présenté pour chaque *programme*

- *les données*
- *les résultats*

Ces derniers sont parfois très détaillés pour permettre de suivre la trace, le plus souvent pour montrer la construction de la structure de données en question.

EXERCICE 1:

[solution PASCAL](#)

[solution C](#)

Les listes linéaires chaînées

Programmer les deux algorithmes suivants :

- 1. Création de deux listes linéaires chaînées à partir d'un fichier de données,**
- 2. Listage des éléments des listes créées.**

Rajouter un ensemble de fonctions récursives permettant d'accomplir les tâches suivantes :

- 1. Recherche d'un élément donné.**
- 2. Existence d'éléments communs entre deux listes linéaires chaînées.**
- 3. Inclusion d'une liste dans une autre.**

Donner deux versions pour 2. et 3.

Solution en C

```
*****
* 1. LE PROGRAMME      *
* 2. LES DONNEES       *
* 3. LES RESULTATS     *
*****
```

PROGRAMME 1

```
#include <Alloc.H>
#include <Stdlib.H>
#include <Stdio.H>
```

```
struct Maillon
{
    int Val ;
    struct Maillon *Suiv ;
} ;
```

```
struct Maillon *Tete1, *Tete2;
int V ;
FILE *Fs, *Fe;
```

```
/*-----*/
/* Procédures d'implémentation du modèle sur les listes linéaires chaînées */
/*-----*/
```

```
struct Maillon *Allouer ( )
{
    return ( (struct Maillon *) malloc( sizeof(struct Maillon)) );
}
```

```
void Affval(struct Maillon *P, int V)
{ P->Val =V; }
```

```
void Affadr( struct Maillon *P, struct Maillon *Q)
{ P->Suiv = Q; }
```

```
struct Maillon *Suivant( struct Maillon *P)
{ return( P->Suiv ); }
```

```
int Valeur( struct Maillon *P)
{ return( P->Val ) ; }
/*-----*/
/*      Création      */
/*-----*/
```

```

struct Maillon *Creer()
{
    struct Maillon *P, *Q, *Prem ;
    int I, Nombre, V;
    fscanf(Fe, "%d" , &Nombre);
    fprintf(Fs, "Nombre d'éléments de la liste à créer : %d \n", Nombre);
    Prem = NULL ;
    P = NULL ;
    for (I=1; I<=Nombre ;I++)
    {
        fscanf(Fe, "%d" , &V);
        fprintf(Fs, " Elément inséré : %d \n", V);
        Q = Allouer();
        Affval(Q, V);
        Affadr(Q, NULL);
        if (Prem != NULL )
            Affadr(P, Q) ;
        else Prem = Q ;
        P = Q ;
    }
    return(Prem);
}

```

```

/*-----*/
/*      Listage des éléments      */
/*-----*/

```

```

void Lister ( struct Maillon *L)
{
    struct Maillon *P;
    fprintf(Fs, "Impression des éléments de la liste \n");
    P = L;
    while ( P != NULL)
    {
        fprintf(Fs, "%d \n" ,P->Val);
        P = Suivant(P) ;
    }
}

```

```

/*-----*/
/*      Procédures récursives      */
/*-----*/

```

```

/* Liste sans le premier élément */
struct Maillon *Reste( struct Maillon *Liste)
{
    return( Suivant( Liste ) );
}

```

```

/* Existence d'un élément */

```

```

int Trouv ( struct Maillon *Liste, int Val)
{
    if (Liste == NULL)
        return( 0 ) ;
    else
        if (Valeur (Liste) == Val)
            return(1) ;
        else
            return( Trouv (Reste(Liste), Val )) ;
}

/* Existence d'éléments communs */
int Commun ( struct Maillon *Liste1, struct Maillon *Liste2)
{
    if (Liste1 == NULL)
        return( 0 ) ;
    else
        if (Trouv (Liste2 , Valeur(Liste1)) )
            return(1) ;
        else
            return( Commun (Reste(Liste1), Liste2 )) ;
}

/* Inclusion */
int Tous ( struct Maillon *Liste1, struct Maillon *Liste2)
{
    if ( Liste1 == NULL )
        return( 1 ) ;
    else
        if ( ! Trouv(Liste2, Valeur(Liste1) ) )
            return( 0);
        else
            return( Tous (Reste(Liste1), Liste2 )) ;
}

main()
{
    Fs = fopen ("R_llc.C", "w");
    Fe = fopen ("D_llc.C", "r");
    Tete1 = Creer();
    fprintf(Fs, "\n");
    Lister( Tete1);
    fscanf(Fe, "%d", &V);
    fprintf(Fs, "\nElément à rechercher : %d \n", V);
    fprintf(Fs, "Résultat : %d \n", Trouv(Tete1, V) );
    fprintf(Fs, "\n");
    Tete2 = Creer();
    fprintf(Fs, "\n");
    Lister(Tete2);
    fprintf(Fs, "\nLes deux listes ont-ils des éléments en commun ? \n");
}

```

```
fprintf(Fs, " ---> la réponse est : %d", Commun(Tete1, Tete2 ) );  
fprintf(Fs, "\nLa première liste est-elle incluse dans la seconde ? \n");  
fprintf(Fs, " ---> la réponse est : %d", Tous(Tete1, Tete2 ) );  
}
```

Données : (Contenu du fichier D_llc)

8
12
45
67
75
45
23
87
23
34
5
34
54
12
321
654

Résultats : (Contenu du fichier R_llc.c)

Nombre d'éléments de la liste à créer : 8

Élément inséré : 12

Élément inséré : 45

Élément inséré : 67

Élément inséré : 75

Élément inséré : 45

Élément inséré : 23

Élément inséré : 87

Élément inséré : 23

Impression des éléments de la liste

12
45
67
75
45
23
87
23

Élément à rechercher : 34

Résultat : 0

Nombre d'éléments de la liste à créer : 5

Elément inséré : 34
Elément inséré : 54
Elément inséré : 12
Elément inséré : 321
Elément inséré : 654

Impression des éléments de la liste
34
54
12
321
654

Les deux listes ont-ils des éléments en commun ?
---> la réponse est : 1
La première liste est-elle incluse dans la seconde ?
---> la réponse est : 0

EXERCICE 2.

solution PASCAL

Récurtivité

Ecrire des procédures récursives dans un tableau pour calculer la somme, le produit et la moyenne de ses éléments.

Ecrire une procédure récursive qui détermine le plus grand élément d'un tableau.

Ecrire des procédures récursives qui effectuent :

- la somme des N premiers naturels
- le quotient de A par B
- le reste de A par B
- le pgcd de A et B

```
*****
* 1. LE PROGRAMME *
* 2. LES DONNEES *
* 3. LES RESULTATS *
*****
```

PROGRAMME 2

{
 Récursivité.

Algorithmes récursifs dans un tableau :

- . Somme
- . Produit
- . Moyenne
- . Maximum

Autres :

- . Somme des N premiers naturels
- . Quotient de A par B
- . Reste de A par B
- . Pgcd de A et B

}

PROGRAM Récursivite ;

VAR

 A,B : INTEGER;

 T : Array [1..10] OF INTEGER ;

 Fs : TEXT;

FUNCTION Max (A,B : INTEGER) : INTEGER ;

 BEGIN

 IF A > B

 THEN Max := A

 ELSE Max := B

 END;

{ A + B en fonction de Succ }

FUNCTION Succ (B, A: INTEGER) : INTEGER;

 BEGIN

 IF B <= 1

 THEN Succ := (A + B)

 ELSE Succ := (Succ (B-1, Succ(1,A)))

 END;

{ Maximum d'un tableau }

FUNCTION Maximum (N : INTEGER) : INTEGER;

 BEGIN

 IF N = 1

 THEN Maximum := T[1]

 ELSE Maximum := Max(Maximum(N-1), T[N])

 END;

{ Somme des éléments d'un tableau }

FUNCTION Somme (N : INTEGER): INTEGER;

 BEGIN

 IF N = 1

```

    THEN Somme := T[1]
    ELSE Somme := Somme(N-1) + T[N]
END;
```

```

{ Produit des éléments d'un tableau }
FUNCTION Produit ( N : INTEGER ): INTEGER;
BEGIN
    IF N = 1
    THEN Produit := T[1]
    ELSE Produit := Produit(N-1) * T[N]
END;
```

```

{ Moyenne des éléments d'un tableau }
FUNCTION Moy ( N : INTEGER ): REAL;
BEGIN
    IF N = 1
    THEN Moy := T[1]
    ELSE Moy := ( Moy(N-1) + T[N] ) / 2
END;
```

```

{ Somme des N Premiers naturels }
FUNCTION Som ( N : INTEGER ): INTEGER;
BEGIN
    IF N = 0
    THEN Som := 0
    ELSE Som := Som(N-1) + N
END;
```

```

{ Quotient de A par B }
FUNCTION Quo( A, B : INTEGER ) : INTEGER;
BEGIN
    IF A < B
    THEN Quo := 0
    ELSE Quo := 1 + Quo( A-B, B)
END;
```

```

{ Reste de A par B }
FUNCTION Rest(A, B : INTEGER) : INTEGER;
BEGIN
    IF A < B
    THEN Rest := A
    ELSE Rest := Rest(A-B,B)
END;
```

```

{ Pgcd de A par B }
FUNCTION Pgcd( A, B : INTEGER ): INTEGER;
BEGIN
    IF B = 0
    THEN Pgcd := A
    ELSE Pgcd := Pgcd(B, Rest(A,B) )
END;
```



```

END;

BEGIN
  ASSIGN(Fs, 'R_rekurs.Pas');
  REWRITE(Fs);
  Writeln(Fs, 'Somme de 12 et 9 : ', Succ(12, 9));

  T[1] := 5;   T[2] := 2 ; T[3] := 8;
  T[4] := 17;  T[5] := 0;
  T[6] := 7;   T[7] := 6 ; T[8] := 21;
  T[9] := 9 ;   T[10] := 10;
  Writeln(Fs, 'Maximum dans le tableau T : ', Maximum(10));
  Writeln(Fs, 'Somme des éléments de T : ', Somme(10) );
  Writeln(Fs, 'Produit des 4 premiers éléments de T : ', Produit(4) );
  Writeln(Fs, 'Moyenne des éléments de T : ', Moy(10):6:2 );
  Writeln(Fs, 'Somme des 45 premiers naturels : ', Som(45) );
  Writeln(Fs, 'Quotient de 37 par 5 : ', Quo(37,5));
  Writeln(Fs, 'Reste de 37 par 5 : ', Rest(37,5));
  Writeln(Fs, 'Pgcd de 12 et 4 : ', Pgcd(12,4));
  CLOSE(Fs)
END.

```

Résultats (Contenu du fichier R_rekurs.pas) :

```

Somme de 12 et 9 : 21
Maximum dans le tableau T : 21
Somme des éléments de T : 85
Produit des 4 premiers éléments de T : 1360
Moyenne des éléments de T : 10.65
Somme des 45 premiers naturels : 1035
Quotient de 37 par 5 : 7
Reste de 37 par 5 : 2
Pgcd de 12 et 4 : 4

```

EXERCICE 3.



Tour de Hanoi

Ecrire une procédure récursive qui simule le jeu des tours de "Hanoi". La procédure imprimera la liste des déplacements à effectuer.

```

*****
* 1. LE PROGRAMME *
* 2. LES DONNEES *
* 3. LES RESULTATS *

```

PROGRAMME 3

```
{  
  Récursivité : Tour de Hanoi.  
}
```

PROGRAM Hanoi ;

VAR

N : INTEGER;

Fs : TEXT ;

PROCEDURE H(N : INTEGER; De, Vers, Aux : CHAR);

BEGIN

IF N = 1

THEN WRITELN(Fs, De , ' Vers ', Vers)

ELSE

BEGIN

H (N-1, De, Aux, Vers);

WRITELN(Fs, De, ' Vers ', Vers);

H(N-1, Aux, Vers, De)

END;

END;

BEGIN

ASSIGN(Fs, 'R_hanoi.Pas');

REWRITE(Fs);

N := 5 ;

WRITELN(Fs, 'Voici la série des déplacements pour 5 disques : ');

WRITELN(Fs);

H(N, 'A', 'B', 'C');

CLOSE(Fs);

END.

Résultats (Contenu du fichier R_hanoi.pas) :

Voici la série des déplacements pour 5 disques

A Vers B

A Vers C

B Vers C

A Vers B

C Vers A

C Vers B

A Vers B

A Vers C

B Vers C

B Vers A

C Vers A
 B Vers C
 A Vers B
 A Vers C
 B Vers C
 A Vers B
 C Vers A
 C Vers B
 A Vers B
 C Vers A
 B Vers C
 B Vers A
 C Vers A
 C Vers B
 A Vers B
 A Vers C
 B Vers C
 A Vers B
 C Vers A
 C Vers B
 A Vers B

EXERCICE 4.



Etude du parcours "Postordre" sur les arbres binaires

Ecrire un programme qui crée un arbre binaire représentant l'expression arithmétique : $(A+B) * (C-(D+E))$. Lister cet arbre en inordre récursif pour montrer qu'il est bien créé.

Ecrire une procédure non récursive qui utilise par conséquent une pile pour parcourir l'arbre en postordre.

Ecrire une procédure non récursive qui utilise le parcours Postordre avec l'emploi d'une pile pour évaluer une expression arithmétique.

```

*****
* 1. LE PROGRAMME *
* 2. LES DONNEES *
* 3. LES RESULTATS *
*****
  
```

```

PROGRAMME 4
{
  
```

Etude du Postordre.

Création d'un arbre binaire représentant une expression arithmétique.

Parcours Postordre avec utilisation d'une pile.

Listage en ordre intérieur (inordre).

Evaluation d'une expression arithmétique utilisant le parcours Postordre avec utilisation d'une pile.

}

PROGRAM Postordre;

CONST Limitpile = 50;

TYPE

T1 = RECORD

Val : INTEGER;

Car : CHAR;

END;

T = ^Noeud;

Noeud = RECORD

Element : CHAR;

Fg, Fd : T ;

Feuille : BOOLEAN;

END;

VAR

Tab : ARRAY(1..10.) OF T1;

Fs : TEXT;

Racine, P1, P2, P : T;

{ Modèle sur les piles }

TYPE Pile = RECORD

Som : INTEGER;

Tab : ARRAY(1..Limitpile.) OF T ;

END;

PROCEDURE Creerpile (VAR P : Pile);

BEGIN P.Som := 0 END;

FUNCTION Pilevide(P : Pile) : BOOLEAN;

BEGIN Pilevide := (P.Som = 0) END;

FUNCTION Pilepleine (P : Pile) : BOOLEAN;

BEGIN Pilepleine := (P.Som = Limitpile) END;

FUNCTION Top (P : Pile) : T;

BEGIN Top := P.Tab(.P.Som.) END;

PROCEDURE Empiler (VAR P : Pile ; Val : T);

BEGIN

IF NOT Pilepleine(P)

THEN

```

BEGIN
  P.Som := P.Som + 1 ;
  P.Tab(.P.Som.) := Val ;
END
ELSE
BEGIN
  WRITELN(Fs,' Pile saturée');
  Halt;
END
END;

```

```

PROCEDURE Depiler (VAR P: Pile; VAR Val : T);
BEGIN
  IF NOT Pilevide(P)
  THEN
    BEGIN
      Val := P.Tab(.P.Som.);
      P.Som := P.Som - 1
    END
  ELSE
    BEGIN
      WRITELN(Fs, 'Pile Vide');
      Halt;
    END;
  END;
END;

```

{ Pile de valeurs entières }

```

TYPE Pile1 = RECORD
  Som : INTEGER;
  Tab : ARRAY(1..Limitpile.) OF INTEGER ;
END;

```

```

PROCEDURE Creerpile1 ( VAR P : Pile1);
  BEGIN P.Som := 0 END;

```

```

FUNCTION Pilevide1( P : Pile1) : BOOLEAN;
  BEGIN Pilevide1:= (P.Som = 0) END;

```

```

FUNCTION Pilepleine1 (P : Pile1) : BOOLEAN;
  BEGIN Pilepleine1 := (P.Som = Limitpile) END;

```

```

PROCEDURE Empiler1 (VAR P : Pile1 ; Val : INTEGER);
BEGIN
  IF NOT Pilepleine1(P)
  THEN
    BEGIN
      P.Som := P.Som + 1 ;
      P.Tab(.P.Som.) := Val ;
    END
  END

```

```

ELSE
  BEGIN
    WRITELN(Fs,' Pile Saturee');
    Halt;
  END
END;

```

```

PROCEDURE Depiler1 (VAR P: Pile1; VAR Val : INTEGER);
BEGIN
  IF NOT Pilevide1(P)
  THEN
    BEGIN
      Val := P.Tab(.P.Som.);
      P.Som := P.Som - 1
    END
  ELSE
    BEGIN
      WRITELN(Fs, 'Pile Vide');
      Halt;
    END;
  END;
END;

```

{ Modèle sur les arbres binaires }

```

FUNCTION Info(P : T) : CHAR ;
  BEGIN Info := P^.Element END;

```

```

FUNCTION Fg( P : T) : T;
  BEGIN Fg := P^.Fg END;

```

```

FUNCTION Fd( P : T) : T;
  BEGIN Fd := P^.Fd END;

```

```

PROCEDURE Affinfo ( VAR P : T; Val : CHAR);
  BEGIN P^.Element := Val END;

```

```

PROCEDURE Aff_fg( VAR P : T; Q : T);
  BEGIN P^.Fg := Q END;

```

```

PROCEDURE Aff_fd( VAR P : T; Q : T);
  BEGIN P^.Fd := Q END;

```

```

PROCEDURE Aff_feuille( VAR P: T; B : BOOLEAN);
  BEGIN P^.Feuille := B END;

```

```

FUNCTION Creernoed( Val : CHAR) : T;
  VAR
    P : T;
  BEGIN
    NEW ( P );

```

```

    Creernoead := P ;
    P^.Element := Val;
    P^.Fg := NIL;
    P^.Fd := NIL
END;

```

```

{ Association caractère <--> valeur }
FUNCTION Valeur ( C : CHAR) : INTEGER;
VAR
    I : INTEGER;
    Trouv : BOOLEAN;
BEGIN
    I:= 0; Trouv := FALSE;
    WHILE (I <= 10) AND ( NOT Trouv ) DO
        IF C = Tab(.I).Car
        THEN Trouv := TRUE
        ELSE I := I + 1 ;
    IF Trouv
    THEN Valeur := Tab(.I).Val
    ELSE WRITELN(Fs, '***** Erreur dans l"identificateur');
END;

```

```

{ Opération }
PROCEDURE Oper( C : CHAR; V1, V2 : INTEGER; VAR Res : INTEGER);
BEGIN
    IF C = '+'
    THEN Res := V1 + V2
    ELSE
        IF C = '-'
        THEN Res := V2 - V1
        ELSE
            IF C = '*'
            THEN Res := V1 * V2
            ELSE WRITELN(Fs, '***** Erreur dans l"opérateur');
END;

```

```

{ Listage en inordre }
PROCEDURE Lister ( A : T);
BEGIN
    IF A <> NIL
    THEN
        BEGIN
            Lister ( Fg(A));
            WRITELN(Fs,Info(A) );
            Lister ( Fd(A) )
        END;
    END;
END;

```

```

{ Parcours Postordre avec utilisation d'une pile }

```

```

PROCEDURE Post(Racine : T);
VAR
  P : Pile;
  Possible, Cond : BOOLEAN;
  N, Nprime, Q : T;
BEGIN
  N := Racine;
  Creerpile(P);
  Possible := TRUE;
  WHILE Possible DO
    BEGIN
      WHILE N <> NIL DO
        BEGIN  Empiler(P, N); Nprime := N; N := Fg(N) END;
        IF NOT Pilevide(P)
        THEN
          BEGIN
            IF Fd(Nprime) = NIL
            THEN
              BEGIN
                Cond := TRUE;
                WHILE NOT Pilevide(P) AND Cond DO
                  BEGIN
                    WRITELN(Fs,'Visite de ' , Info(Nprime) );
                    Q := Nprime ;
                    Depiler(P, Nprime);
                    IF NOT Pilevide(P)
                    THEN
                      BEGIN
                        Nprime := Top(P);
                        Cond := ( Fd(Nprime) = Q )
                      END
                    ELSE Possible := FALSE
                    END
                  END;
                N := Fd(Nprime)
              END
            ELSE
              Possible := FALSE
            END;
          END;
        END;
      END;
    END;
  END;

```

```

PROCEDURE Evalpost(Racine : T);
VAR
  P : Pile;
  P1 : Pile1;
  Possible, Cond : BOOLEAN;
  N, Nprime, Q : T;
  Val1, Val2, Res : INTEGER;
BEGIN
  N := Racine;

```



```

Creerpile(P);
Creerpile1(P1);
Possible := TRUE;
WHILE Possible DO
  BEGIN
    WHILE N <> NIL DO
      BEGIN Empiler(P, N); Nprime := N; N := Fg(N) END;
      IF NOT Pilevide(P)
      THEN
        BEGIN
          IF Fd(Nprime) = NIL
          THEN
            BEGIN
              Empiler1(P1, Valeur(Info(Nprime)) );
              Cond := TRUE;
              WHILE NOT Pilevide(P) AND Cond DO
                BEGIN
                  WRITELN(Fs, 'Visite de :', Info(Nprime) );
                  Q := Nprime ;
                  Depiler(P, Nprime);
                  IF NOT Pilevide(P)
                  THEN
                    BEGIN
                      Nprime := Top(P);
                      IF Fd(Nprime) = Q
                      THEN
                        BEGIN
                          Depiler1(P1, Val1);
                          Depiler1(P1, Val2) ;
                          Oper(Info(Nprime), Val1, Val2, Res);
                          WRITELN(Fs, 'Résultat intermédiaire = ', Res);
                          Empiler1(P1, Res);
                        END
                      ELSE Cond := FALSE;
                    END
                  ELSE Possible := FALSE
                END
              END;
              N := Fd(Nprime)
            END
          ELSE
            Possible := FALSE
          END;
        END;
      END;
    END;
  BEGIN
    Tab(1.).Car := 'A';
    Tab(1.).Val := 5 ;
    Tab(2.).Car := 'B';
    Tab(2.).Val := 13 ;
  END

```

```

Tab(.3.).Car := 'C';
Tab(.3.).Val := 16 ;
Tab(.4.).Car := 'D';
Tab(.4.).Val := 21 ;
Tab(.5.).Car := 'E';
Tab(.5.).Val := 5 ;

```

```

ASSIGN(Fs, 'R_post.Pas');
REWRITE(Fs);

```

```

{ Construction de l'arbre binaire représentant
l'expression : (A+B) * ( C-(D+E))}
WRITELN(Fs, 'Création de l'arbre représentant l'expression (A+B) * ( C-(D+E))');
WRITELN(Fs, '---> Résultat en mémoire . . .');
WRITELN(Fs);
Racine := Creernoeud('*');
Aff_feuille (Racine,FALSE);
P:=Creernoeud('+');
Aff_feuille (P,FALSE);
Aff_fg(Racine, P);
P1 := P;
P:=Creernoeud('-');
Aff_feuille (P,FALSE);
Aff_fd(Racine, P);
P2 := P ;
P:=Creernoeud('A');
Aff_feuille (P,TRUE);
Aff_fg(P1, P);
P:=Creernoeud('B');
Aff_feuille (P,TRUE);
Aff_fd(P1, P);
P:=Creernoeud('C');
Aff_feuille (P,TRUE);
Aff_fg(P2, P);

P:=Creernoeud('+');
Aff_feuille (P,FALSE);
Aff_fd(P2, P);
P1 := P;
P:=Creernoeud('D');
Aff_feuille (P,TRUE);
Aff_fg(P1, P);
P:=Creernoeud('E');
Aff_feuille (P,TRUE);
Aff_fd(P1, P);
{ Lister les noeuds de l'arbre en Inordre }
WRITELN(Fs, 'Listage des noeuds de l'arbre en Inordre');
WRITELN(Fs);
Lister(Racine);
WRITELN(Fs);

```

```

{ Lister les noeuds de l'arbre en Postordre }
WRITELN(Fs, 'Listage des noeuds de l'arbre en Postordre');
WRITELN(Fs);
Post(Racine);
WRITELN(Fs);
WRITELN(Fs, 'Evaluation de l\'expression');
WRITELN(Fs);
Evalpost(Racine);
CLOSE(Fs)
END.

```

Résultats (Contenu du fichier R_post.pas) :

Création de l'arbre représentant l'expression $(A+B) * (C-(D+E))$
 ---> Résultat en mémoire . . .

Listage des noeuds de l'arbre en Inordre

```

A
+
B
*
C
-
D
+
E

```

Listage des noeuds de l'arbre en Postordre

```

Visite de :A
Visite de :B
Visite de :+
Visite de :C
Visite de :D
Visite de :E
Visite de :+
Visite de :-
Visite de :*

```

Evaluation de l'expression

```

Visite de :A
Visite de :B
Résultat intermédiaire = 18
Visite de :+
Visite de :C
Visite de :D
Visite de :E
Résultat intermédiaire = 26

```

Visite de :+
Résultat intermédiaire = -10
Visite de :-
Résultat intermédiaire = -180
Visite de :*

EXERCICE 5.

solution PASCAL

Traduction automatique de la procédure récursive de Fibonacci.

Considérer la procédure récursive de Fibonacci présentée en cours. Faire une traduction automatique de la procédure, c'est à dire le passage vers une procédure itérative, et la programmer directement avec donc l'emploi des "GOTOS". Rendre la procédure formelle, c'est à dire la réécrire sans l'utilisation des "GOTOS". Tester les deux procédures pour les 15 premiers entiers naturels.

```
*****  
* 1. LE PROGRAMME *  
* 2. LES DONNEES *  
* 3. LES RESULTATS *  
*****
```

PROGRAMME 5

{ Récursivité.

Programmation

- de la traduction automatique (algorithme informel avec des Allera)
- de la version formelle (algorithme formel après élimination des Allera)

}

PROGRAM Transformationdelarecursivite;

CONST

Limitpile = 50;

TYPE

Typezdd = RECORD { Définition de la zone de données }

N, X, Y : INTEGER;

Adr : INTEGER

END;

```

Typepile = RECORD { Implémentation de la pile }
  Tab : ARRAY[1..Limitpile] OF Typezdd;
  Som : INTEGER;
END;
VAR
  N,I : INTEGER; { On veut calculer Fib(N) }
  Fs : TEXT;

{ Procédures d'implémentation de la pile }

PROCEDURE Creerpile ( VAR P : Typepile );
  BEGIN P.Som := 0 END;

FUNCTION Pilevide( P : Typepile ) : BOOLEAN;
  BEGIN Pilevide := (P.Som = 0) END;

FUNCTION Pilepleine( P : Typepile ) : BOOLEAN;
  BEGIN Pilepleine := (P.Som = Limitpile) END;

PROCEDURE Empiler(VAR P : Typepile; Val : Typezdd) ;
  BEGIN
    IF NOT Pilepleine(P)
    THEN
      BEGIN
        P.Som := P.Som + 1;
        P.Tab[P.Som] := Val
      END
    ELSE WRITELN(Fs,' *** Pile saturée ')
    END;

PROCEDURE Depiler ( VAR P : Typepile; VAR Val : Typezdd);
  BEGIN
    IF NOT Pilevide(P)
    THEN
      BEGIN
        Val := P.Tab[P.Som];
        P.Som := P.Som - 1
      END
    ELSE WRITELN(Fs,' *** Pile vide ')
    END;

{ Version automatique informelle }

FUNCTION Auto ( Nombre : INTEGER ) : INTEGER;
  VAR
    P : Typepile;
    Zdd : Typezdd ;
    I, Fib : INTEGER;
    Label 10, 1, 2, 3;
  BEGIN

```

```

Creerpile(P);
Empiler(P, Zdd);
{ Préparer le premier Appel }
Zdd.N := Nombre;
Zdd.Adr := 1;

```

```

10 : IF Zdd.N <= 1
    THEN
        BEGIN
            Fib := Zdd.N;
            { Retour }
            I := Zdd.Adr;
            Depiler(P, Zdd);
            Case I OF
                1 : Goto 1;
                2 : Goto 2;
                3 : Goto 3;
            END;{case}
        END;

```

```

    { Premier Appel }
    Empiler(P, Zdd);
    Zdd.N := Zdd.N - 1;
    Zdd.Adr := 2;
    Goto 10;

```

```

2 : Zdd.X := Fib;
    { Deuxième appel }
    Empiler( P, Zdd);
    Zdd.N := Zdd.N - 2;
    Zdd.Adr := 3;
    Goto 10;
3 : Zdd.Y := Fib;
    Fib := Zdd.X + Zdd.Y ;
    { Retour }
    I := Zdd.Adr;
    Depiler(P, Zdd);
    Case I OF
        1 : Goto 1;
        2 : Goto 2;
        3 : Goto 3;
    END;

```

```

1 : Auto := Fib END;

```

{ Version formelle (avec élimination des Gotos) }

```

FUNCTION Formel( Nombre : INTEGER ) : INTEGER;
VAR
    P : Typepile;

```

```

Zdd : Typezdd;
Fib, I : INTEGER;
Fin, Sort : BOOLEAN;
BEGIN
  Creerpile (P);
  Empiler(P,Zdd);
  Zdd.N := Nombre;
  Zdd.Adr := 1;
  Fin := FALSE;
  WHILE NOT Fin DO
    BEGIN
      Sort := FALSE;
      WHILE NOT Sort DO
        BEGIN
          WHILE Zdd.N > 1 DO
            BEGIN
              Empiler(P, Zdd);
              Zdd.N := Zdd.N - 1;
              Zdd.Adr := 2
            END;
            Fib := Zdd.N;
            I := Zdd.Adr ;
            Depiler(P, Zdd);
            IF I=2
            THEN
              BEGIN
                Zdd.X := Fib;
                Empiler(P, Zdd);
                Zdd.N := Zdd.N - 2;
                Zdd.Adr := 3;
              END
            ELSE Sort := TRUE
            END ;
          { à ce niveau I vaut 1 ou 2 }
          WHILE I=3 DO
            BEGIN
              Zdd.Y := Fib;
              Fib := Zdd.X + Zdd.Y ;
              I := Zdd.Adr;
              Depiler(P, Zdd)
            END;
            IF I = 2
            THEN
              BEGIN
                Zdd.X := Fib;
                Empiler(P, Zdd);
                Zdd.N := Zdd.N - 2;
                Zdd.Adr := 3
              END
            ELSE Fin := TRUE

```

```

    END;
    Formel := Fib;
END;{ Formel }

BEGIN
    ASSIGN(Fs, 'R_trad.Pas');
    REWRITE(Fs);
    N := 15;
    FOR I:=1 TO N DO
        BEGIN
            Writeln(Fs, '*** Calcul de Fib( ', I:2, ') ');
            Writeln(Fs);
            Writeln(Fs, ' . Résultat de la traduction automatique');
            Writeln(Fs, ' ---> Auto ( ', I:2, ') = ', Auto(I):3 );
            Writeln(Fs, ' . Résultat de la version formelle');
            Writeln(Fs, ' ---> Formel( ', I:2, ') = ', Formel(I):3 );
            Writeln(Fs);
        END;
    CLOSE(Fs)
END.

```

Résultats (Contenu du fichier R_trad.pas) :

*** Calcul de Fib(1)

```

. Résultat de la traduction automatique
---> Auto ( 1 ) =  1
. Résultat de la version formelle
---> Formel( 1 ) =  1

```

*** Calcul de Fib(2)

```

. Résultat de la traduction automatique
---> Auto ( 2 ) =  1
. Résultat de la version formelle
---> Formel( 2 ) =  1

```

*** Calcul de Fib(3)

```

. Résultat de la traduction automatique
---> Auto ( 3 ) =  2
. Résultat de la version formelle
---> Formel( 3 ) =  2

```

*** Calcul de Fib(4)

```

. Résultat de la traduction automatique
---> Auto ( 4 ) =  3
. Résultat de la version formelle
---> Formel( 4 ) =  3

```


*** Calcul de Fib(5)

. Résultat de la traduction automatique

---> Auto (5) = 5

. Résultat de la version formelle

---> Formel(5) = 5

*** Calcul de Fib(6)

. Résultat de la traduction automatique

---> Auto (6) = 8

. Résultat de la version formelle

---> Formel(6) = 8

*** Calcul de Fib(7)

. Résultat de la traduction automatique

---> Auto (7) = 13

. Résultat de la version formelle

---> Formel(7) = 13

*** Calcul de Fib(8)

. Résultat de la traduction automatique

---> Auto (8) = 21

. Résultat de la version formelle

---> Formel(8) = 21

*** Calcul de Fib(9)

. Résultat de la traduction automatique

---> Auto (9) = 34

. Résultat de la version formelle

---> Formel(9) = 34

*** Calcul de Fib(10)

. Résultat de la traduction automatique

---> Auto (10) = 55

. Résultat de la version formelle

---> Formel(10) = 55

*** Calcul de Fib(11)

. Résultat de la traduction automatique

---> Auto (11) = 89

. Résultat de la version formelle

---> Formel(11) = 89

*** Calcul de Fib(12)

. Résultat de la traduction automatique

---> Auto (12) = 144

. Résultat de la version formelle

---> Formel(12) = 144

*** Calcul de Fib(13)

. Résultat de la traduction automatique

---> Auto (13) = 233

. Résultat de la version formelle

---> Formel(13) = 233

*** Calcul de Fib(14)

. Résultat de la traduction automatique

---> Auto (14) = 377

. Résultat de la version formelle

---> Formel(14) = 377

*** Calcul de Fib(15)

. Résultat de la traduction automatique

---> Auto (15) = 610

. Résultat de la version formelle

---> Formel(15) = 610

EXERCICE 6.

solution PASCAL

solution C

Arbres de recherche binaire

Programmer les algorithmes suivants dans un arbre de recherche binaire :

1. Algorithmes récursifs :

- insertion dans un arbre de recherche binaire
- listage des noeuds d'un arbre en ordre croissant
- nombre de noeuds
- nombre de feuilles
- somme des éléments
- profondeur d'un arbre

2. Algorithme récursif préordre qui détermine en un seul parcours le nombre de feuilles, le nombre de noeuds et la somme de tous les éléments de l'arbre.

3. Algorithme récursif préordre qui détermine la profondeur de l'arbre.

4. Algorithmes de parcours non récursifs avec utilisation de pile

- préordre
- inordre
- postordre

solution pascalle

```
*****  
* 1. LE PROGRAMME *  
* 2. LES DONNEES *  
* 3. LES RESULTATS *  
*****
```

PROGRAMME 6

{
Les arbres

Insertion dans un arbre de recherche binaire
Listage de l'arbre en inordre

Nombre de noeuds
Nombre de feuilles
Somme des éléments
Profondeur

Algorithme récursif préordre qui détermine en un seul parcours :

- le nombre de feuilles,
- le nombre de noeuds,
- la somme des éléments,

Algorithme récursif préordre qui détermine la profondeur.

Parcours préordre avec utilisation d'une pile
Parcours inordre avec utilisation d'une pile
Parcours Postordre avec utilisation d'une pile

}

PROGRAM Arbre_de_recherche_binaire;

VAR

Fs : TEXT;

Nbrnoeud : INTEGER;

```
Somme : INTEGER;
Feuille : INTEGER;
D, Max : INTEGER;
```

```
{ Modèle sur les arbres binaires }
```

```
TYPE
```

```
  T = ^Noeud;
  Noeud = RECORD
    Element : INTEGER;
    Fg, Fd : T ;
  END;
```

```
FUNCTION Info(P : T) : INTEGER;
  BEGIN Info := P^.Element  END;
```

```
FUNCTION Fg( P : T) : T;
  BEGIN Fg := P^.Fg  END;
```

```
FUNCTION Fd( P : T) : T;
  BEGIN Fd := P^.Fd  END;
```

```
PROCEDURE Affinfo ( VAR P : T; Val : INTEGER);
  BEGIN P^.Element := Val  END;
```

```
PROCEDURE Aff_fg( VAR P : T; Q : T);
  BEGIN P^.Fg := Q  END;
```

```
PROCEDURE Aff_fd( VAR P : T; Q : T);
  BEGIN P^.Fd := Q  END;
```

```
FUNCTION Creernoeud( Val : INTEGER) : T;
  VAR
    P : T;
  BEGIN
    NEW ( P );
    Creernoeud := P ;
    P^.Element := Val;
    P^.Fg := NIL;
    P^.Fd := NIL;
  END;
```

```
{ Modèle sur les piles }
```

```
TYPE
```

```
  Pile = RECORD
    Som : INTEGER;
    Tab : ARRAY(1..50.) OF T ;
  END;
```

```
PROCEDURE Creerpile ( VAR P : Pile);
```

```
BEGIN P.Som := 0 END;
```

```
FUNCTION Pilevide( P : Pile) : BOOLEAN;  
  BEGIN Pilevide := (P.Som = 0) END;
```

```
FUNCTION Pilepleine (P : Pile) : BOOLEAN;  
  BEGIN Pilepleine := (P.Som = 50) END;
```

```
FUNCTION Top ( P : Pile) : T;  
  BEGIN Top := P.Tab(.P.Som.) END;
```

```
PROCEDURE Empiler (VAR P : Pile ; Val : T);  
  BEGIN  
    IF NOT Pilepleine(P)  
    THEN  
      BEGIN  
        P.Som := P.Som + 1 ;  
        P.Tab(.P.Som.) := Val ;  
      END  
    ELSE  
      BEGIN  
        WRITELN(Fs,' Pile Saturee');  
        HALT;  
      END  
    END;  
END;
```

```
PROCEDURE Depiler (VAR P: Pile; VAR Val : T;  VAR Possible : BOOLEAN);  
  BEGIN  
    IF NOT Pilevide(P)  
    THEN  
      BEGIN  
        Possible := TRUE;  
        Val := P.Tab(.P.Som.);  
        P.Som := P.Som - 1  
      END  
    ELSE Possible := FALSE;  
  END;
```

```
PROCEDURE Depiler1 (VAR P: Pile; VAR Val : T);  
  BEGIN  
    IF NOT Pilevide(P)  
    THEN  
      BEGIN  
        Val := P.Tab(.P.Som.);  
        P.Som := P.Som - 1  
      END  
    ELSE  
      BEGIN  
        WRITELN(Fs, 'Pile Vide');  
        HALT;  
      END  
    END;  
END;
```

```
END;  
END;
```

```
{ Maximum de A et B }  
FUNCTION Maxi ( A,B : INTEGER) : INTEGER;  
BEGIN  
  IF A > B THEN Maxi := A ELSE Maxi := B  
END;
```

```
{ Insertion dans un arbre de recherche binaire : algorithme récursif }
```

```
PROCEDURE Insert ( X : INTEGER; VAR A : T);  
VAR  
  Y : T;  
BEGIN  
  IF A = NIL  
  THEN A := Creernoed(X)  
  ELSE IF X < Info(A)  
    THEN Insert( X, A^.Fg )  
    ELSE Insert( X, A^.Fd)  
END;
```

```
{ Listage de l'arbre en inordre }  
PROCEDURE Lister ( A : T);  
BEGIN  
  IF A <> NIL  
  THEN  
    BEGIN  
      Lister ( Fg(A));  
      WRITELN(Fs,Info(A) );  
      Lister ( Fd(A) )  
    END;  
END;
```

```
{ Nombre de noeuds }
```

```
FUNCTION Nbnoeud ( P : T ) : INTEGER;  
BEGIN  
  IF P = NIL  
  THEN Nbnoeud := 0  
  ELSE Nbnoeud := 1 + Nbnoeud(Fg(P)) + Nbnoeud(Fd(P))  
END;
```

```
{ Nombre de feuilles }
```

```
FUNCTION Nbfeuille ( P : T ) : INTEGER;  
BEGIN  
  IF P = NIL  
  THEN Nbfeuille := 0
```

```

ELSE
  IF (Fg(P) = NIL) AND ( Fd(P) = NIL)
  THEN Nbfeuille := 1
  ELSE Nbfeuille := Nbfeuille(Fg(P)) + Nbfeuille(Fd(P))
END;

```

{ Somme des éléments }

```

FUNCTION Som (P : T) : INTEGER;
BEGIN
  IF P = NIL
  THEN Som := 0
  ELSE Som := Info(P) + Som(Fg(P)) + Som(Fd(P))
END;

```

{ Profondeur }

```

FUNCTION Profond ( P : T) : INTEGER;
VAR
  Profondeurgauche, Profondeurdroite : INTEGER;
BEGIN
  IF P = NIL
  THEN Profond := 0
  ELSE
    BEGIN
      Profondeurgauche := 1 + Profond(Fg(P));
      Profondeurdroite := 1 + Profond(Fd(P)) ;
      Profond := Maxi (Profondeurgauche, Profondeurdroite)
    END;
  END;
END;

```

{ Algorithme récursif préordre qui détermine la profondeur }

```

PROCEDURE Profondeur ( A: T);
BEGIN
  D:= D + 1 ;
  WRITE(Fs,' ', A^.Element);
  WRITELN(Fs,' [', D:2,']');
  IF { Feuille } (A^.Fg = NIL) AND (A^.Fd= NIL)
  THEN IF Max < D THEN Max := D;

  IF A^.Fg <> NIL
  THEN
    BEGIN
      Profondeur (A^.Fg); { Visite du sous arbre gauche }
      D:= D - 1;          { au retour on fait -1 :
                           c'est une remontée par la gauche}
    END;

  IF A^.Fd <> NIL

```

```

THEN
  BEGIN
    Profondeur (A^.Fd);  { Visite du sous arbre droit }
    D:= D - 1;          { au retour on fait -1 :
                        c'est une remontée par la droite }
  END
END;

```

```

{ Algorithme récursif préordre qui détermine en un seul parcours
- le nombre de feuilles,
- le nombre de noeuds,
- la somme des éléments,
}

```

```

PROCEDURE Preordre ( A: T);
BEGIN
  Nbrnoeud := Nbrnoeud + 1;
  Somme := Somme + A^.Element;
  IF (A^.Fg = NIL) AND (A^.Fd= NIL)
  THEN
    BEGIN
      Feuille := Feuille + 1;
      WRITELN(Fs, ' ', A^.Element);
    END
  ELSE
    BEGIN
      WRITELN(Fs,' ', A^.Element);
      IF A^.Fg <> NIL THEN Preordre (A^.Fg);
      IF A^.Fd <> NIL THEN Preordre (A^.Fd)
    END;
  END;
END;

```

```

{ Parcours préordre avec utilisation d'une pile }

```

```

PROCEDURE Preordre1 ( Racine : T );
VAR
  N : T;
  P : Pile;
  Possible : BOOLEAN;
BEGIN
  Creerpile(P);
  N := Racine;
  Possible := TRUE;
  WHILE Possible DO
    BEGIN
      WHILE N <> NIL DO
        BEGIN
          WRITELN(Fs,'Elément visité ',Info(N) );
          IF Fd(N) <> NIL THEN Empiler ( P, Fd(N) );
          N := Fg(N)
        END
      END
    END
  END

```



```

        END;
    Depiler(P, N, Possible);
END;
END;

```

{ Parcours inordre avec utilisation d'une pile }

```

PROCEDURE Inordre1( Racine : T);
VAR
    M,N : T;
    P : Pile;
    Possible : BOOLEAN;
BEGIN
    Creerpile(P);
    N := Racine;
    Possible := TRUE;
    WHILE Possible DO
        BEGIN
            WHILE ( N <> NIL) DO { Parcourir la branche La plus à gauche avec
                                sauvegarde des noeuds parcourus }

                BEGIN
                    Empiler( P, N);
                    N := Fg(N);
                END;
            IF NOT Pilevide(P)    { Test si c'est terminé }
            THEN
                BEGIN
                    Depiler1(P, N);
                    WRITELN(Fs,'Elément visité ', Info(N) );
                    { Visiter la racine }
                    N := Fd(N); { Pour le parcours du sous arbre droit }
                END
            ELSE Possible := FALSE;
            END;
        END;
    END;
END;

```

{ Parcours Postordre avec utilisation d'une pile }

```

PROCEDURE Postordre(Racine : T);
VAR
    P : Pile;
    Possible, Cond : BOOLEAN;
    N, Nprime, Q : T;
BEGIN
    N := Racine;
    Creerpile(P);
    Possible := TRUE;
    WHILE Possible DO
        BEGIN
            WHILE N <> NIL DO

```

```

BEGIN
  Empiler(P, N);
  Nprime := N;
  N := Fg(N)
END;
IF NOT Pilevide(P)
THEN
  BEGIN
    IF Fd(Nprime) = NIL
    THEN
      BEGIN
        Cond := TRUE;
        WHILE NOT Pilevide(P) AND Cond DO
          BEGIN
            WRITELN(Fs,'Elément visité ', Info(Nprime) );
            Q := Nprime ;
            Depiler1(P, Nprime);
            IF NOT Pilevide(P)
            THEN
              BEGIN
                Nprime := Top(P);
                Cond := ( Fd(Nprime) = Q )
              END
            ELSE Possible := FALSE
            END
          END;
          N := Fd(Nprime)
        END
      ELSE
        Possible := FALSE
      END;
    END;
  END;

```

{ Variables du programme principal }

```

VAR
  I: INTEGER;
  A : T;
  Taille : INTEGER;
  Pourcent : REAL;

```

{ Programme principal }

```

BEGIN
  ASSIGN(Fs, 'R_arbre.Pas');
  REWRITE(Fs);
  Feuille :=0;
  Nbrnoeud := 0;
  Somme := 0;

```

```

{ Création d'un arbre de recherche binaire }
WRITELN(Fs, '*** Création d'un arbre de recherche binaire ');
WRITELN(Fs, ' ---> Résultat en mémoire . . ');
A := NIL;
Insert(3, A);  Insert(2, A);  Insert(5, A);  Insert(8, A);
Insert(7, A);  Insert(12, A);
Insert(32, A); Insert(45, A); Insert(20, A); Insert(17, A);
Insert(25, A); Insert(23, A); Insert(21, A);
Insert(24, A);
Insert(65, A); Insert(40, A); Insert(35, A); Insert(42, A);
Insert(18, A);

WRITELN(Fs);
WRITELN(Fs,
'*** Listage de l'arbre en ordre croissant : inordre ');
Lister(A);
WRITELN(Fs);

WRITELN(Fs, 'Fonction Nbnoeud :', Nbnoeud(A) );
WRITELN(Fs, 'Fonction Nbfeuille :', Nbfeuille(A) );
WRITELN(Fs, 'Fonction Somme :', Som(A) );
WRITELN(Fs, 'Fonction Profondeur :', Profond(A) );

WRITELN(Fs, '*** Parcours de l'arbre en préordre récursif avec calcul du :');
WRITELN(Fs, ' - nombre de noeuds');
WRITELN(Fs, ' - nombre de feuilles');
WRITELN(Fs, ' - somme des éléments');
WRITELN(Fs);
WRITELN(Fs, 'Ordre des éléments visités');
Preordre (A);
WRITELN(Fs);
WRITELN(Fs, 'Nombre de feuilles =', Feuille);
WRITELN(Fs, 'Nombre de noeuds =', Nbrnoeud);
WRITELN(Fs, 'Somme des éléments =', Somme);
WRITELN(Fs);

D:= -1;Max :=-1;
WRITELN(Fs,
'*** Parcours de l'arbre en préordre récursif avec calcul de ');
WRITELN(Fs, ' la profondeur :');
WRITELN(Fs);
WRITELN(Fs,
'Ordre des éléments visités avec le niveau entre crochets');
Profondeur(A);
WRITELN(Fs);
WRITELN(Fs, 'Profondeur =', Max);
WRITELN(Fs);
WRITELN(Fs, 'Parcours de l'arbre en Préordre non récursif :');
WRITELN(Fs);
Preordre1(A);

```

```

WRITELN(Fs);
WRITELN(Fs,'Parcours de l"arbre en Inordre non récursif :');
WRITELN(Fs);
Inordre1(A);
WRITELN(Fs);
WRITELN(Fs,'Parcours de l"arbre en Postordre non récursif :');
WRITELN(Fs);
Postordre(A);
WRITELN(Fs);
CLOSE(Fs)
END.

```

Résultats (Contenu du fichier R_arbre.pas) :

*** Création d'un arbre de recherche binaire
 ---> Résultat en mémoire . . .

*** Listage de l'arbre en ordre croissant : inordre

2
 3
 5
 7
 8
 12
 17
 18
 20
 21
 23
 24
 25
 32
 35
 40
 42
 45
 65

Fonction Nbnoeud :19

Fonction Nbfeuille :8

Fonction Somme :444

Fonction Profondeur :9

*** Parcours de l'arbre en préordre récursif avec calcul du :

- nombre de noeuds
- nombre de feuilles
- somme des éléments

Ordre des éléments visités

. 3
 . 2

. 5
. 8
. 7
. 12
. 32
. 20
. 17
. 18
. 25
. 23
. 21
. 24
. 45
. 40
. 35
. 42
. 65

Nombre de feuilles =8

Nombre de noeuds =19

Somme des éléments =444

*** Parcours de l'arbre en préordre récursif avec calcul de
la profondeur :

Ordre des éléments visités avec le niveau entre crochets

. 3 [0]
. 2 [1]
. 5 [1]
. 8 [2]
. 7 [3]
. 12 [3]
. 32 [4]
. 20 [5]
. 17 [6]
. 18 [7]
. 25 [6]
. 23 [7]
. 21 [8]
. 24 [8]
. 45 [5]
. 40 [6]
. 35 [7]
. 42 [7]
. 65 [6]

Profondeur =8

Parcours de l'arbre en Préordre non récursif :

Elément visité 3
Elément visité 2
Elément visité 5
Elément visité 8
Elément visité 7
Elément visité 12
Elément visité 32
Elément visité 20
Elément visité 17
Elément visité 18
Elément visité 25
Elément visité 23
Elément visité 21
Elément visité 24
Elément visité 45
Elément visité 40
Elément visité 35
Elément visité 42
Elément visité 65

Parcours de l'arbre en Inordre non récursif :

Elément visité 2
Elément visité 3
Elément visité 5
Elément visité 7
Elément visité 8
Elément visité 12
Elément visité 17
Elément visité 18
Elément visité 20
Elément visité 21
Elément visité 23
Elément visité 24
Elément visité 25
Elément visité 32
Elément visité 35
Elément visité 40
Elément visité 42
Elément visité 45
Elément visité 65

Parcours de l'arbre en Postordre non récursif :

Elément visité 2
Elément visité 7
Elément visité 18
Elément visité 17
Elément visité 21
Elément visité 24

Elément visité 23
Elément visité 25
Elément visité 20
Elément visité 35
Elément visité 42
Elément visité 40
Elément visité 65
Elément visité 45
Elément visité 32
Elément visité 12
Elément visité 8
Elément visité 5
Elément visité 3

Pourcentage = 0.31
Total =1446

Solution C

```
*****  
* 1. LE PROGRAMME      *  
* 2. LES DONNEES       *  
* 3. LES RESULTATS     *  
*****
```

PROGRAMME 2

```
#include <Alloc.H>  
#include <Stdlib.H>  
#include <Stdio.H>
```

```
#define True 1  
#define False 0
```

```
typedef int Bool;  
struct Noeud  
{  
    int Element ;  
    struct Noeud *Fg ;  
    struct Noeud *Fd ;  
} ;
```

```
struct Pile  
{
```

```

    int Som ;
    struct Noeud *Tab[50];
} ;

struct Noeud *Racine;
int Nbrnoeud, Feuille, Somme;
int Max, D ; /* Pour la profondeur */
FILE *Fs;

/*-----*/
/* Procédures d'implémentation du modèle sur les arbres binaires */
/*-----*/
struct Noeud *Creernoeud ( int Val)
{
    struct Noeud *P;
    P = (struct Noeud *) malloc( sizeof(struct Noeud)) ;
    P->Element = Val ;
    P->Fg = NULL;
    P->Fd = NULL;
    return (P) ;
}

struct Noeud *Fg ( struct Noeud *P)
{
    return ( P->Fg ); }

struct Noeud *Fd ( struct Noeud *P)
{
    return ( P->Fd ) ; }

int Info ( struct Noeud *P)
{
    return ( P->Element ); }

void Affinfo(struct Noeud *P, int V)
{
    P->Element =V; }

void Aff_fg(struct Noeud *P, struct Noeud *Q)
{
    P->Fg = Q; }

void Aff_fd(struct Noeud *P, struct Noeud *Q)
{
    P->Fd = Q; }

/*-----*/
/* Procédures d'implémentation du modèle sur les piles */
/*-----*/
void Creerpile( struct Pile *P)
{
    struct Pile Q ;
    Q=*P;
    Q.Som = 0 ;
    *P = Q;
}

```



```
Bool Pilepleine(struct Pile P)
{   return ( P.Som == 49 ); }
```

```
Bool Pilevide(struct Pile P)
{   return ( P.Som == 0 ); }
```

```
void Empiler ( struct Pile *P, struct Noeud *Val)
{
    struct Pile Q;
    Q = *P;
    if ( !Pilepleine(*P) )
        { Q.Som++;
          Q.Tab[Q.Som] = Val ;
          *P = Q;
        }
    else { fprintf(Fs, "Pile saturée\n"); exit(0) ;}
}
```

```
void Depiler ( struct Pile *P, struct Noeud **Val)
{
    struct Pile Q;
    Q = *P;
    if ( !Pilevide(*P) )
        { *Val = Q.Tab[Q.Som] ;
          Q.Som--;
          *P = Q ;
        }
    else { fprintf(Fs, "Pile vide\n"); exit(0) ;}
}
```

```
void Depiler1 ( struct Pile *P, struct Noeud **Val, Bool *Ok)
{
    struct Pile Q;
    Q = *P;
    if ( !Pilevide(*P) )
        {
            *Val = Q.Tab[Q.Som] ;
            Q.Som--;
            *P = Q ;
            *Ok = True;
        }
    else *Ok = False;
}
```

```
struct Noeud *Top(struct Pile P)
{   return ( P.Tab[P.Som] ) ; }
```

```
/*-----*/
/* Algorithmes récursifs sur les arbres de recherche binaire */
```

```

/*-----*/

/*-----*/
/*      Listage en ordre croissant      */
/*-----*/
void Lister ( struct Noeud *A)
{
    if ( A != NULL )
    {
        Lister ( Fg(A) );
        fprintf (Fs, " %d \n", Info(A) );
        Lister ( Fd(A) );
    }
}

/*-----*/
/*      Insertion dans un arbre de recherche binaire      */
/*-----*/
void Insert ( int X, struct Noeud **A )
{
    struct Noeud *P;
    if ( *A == NULL )
        *A = Creernoeud(X);
    else
        if (X < Info(*A) )
        {
            P = *A;
            Insert(X, &(P->Fg) ) ;
        }
        else { P = *A; Insert(X, &( P->Fd)) ;}
}

/*-----*/
/*      Parcours en Preordre en vue de déterminer      */
/*      - Nombre de feuilles      */
/*      - Nombre de noeuds      */
/*      - Somme des noeuds      */
/*-----*/

void Preordre( struct Noeud *A)
{
    Nbrnoeud++;
    Somme += Info(A);
    if ( Fg(A) == NULL && Fd(A) == NULL )
    {
        Feuille++;
    }
    else
    {
        if (Fg(A) != NULL ) Preordre( Fg(A) );

```

```

        if (Fd(A) != NULL ) Preordre( Fd(A) );
    }
}

/*-----*/
/*  Modules indépendants pour déterminer :    */
/*  - Nombre de noeuds                        */
/*  - Nombre de feuilles                      */
/*  - Somme des éléments                      */
/*-----*/

int Nbnoeud ( struct Noeud *A)
{
    if ( A == NULL )
        return(0);
    else return( 1 + Nbnoeud(Fg(A)) + Nbnoeud (Fd(A)) );
}

int Nbrfeuille ( struct Noeud *A)
{
    if ( A == NULL ) return(0);
    else if ( Fg(A)==NULL && Fd(A) == NULL ) return(1);
    else return( Nbrfeuille(Fg(A)) + Nbrfeuille (Fd(A)) );
}

int Som ( struct Noeud *A)
{
    if ( A == NULL )
        return(0);
    else return( Info(A) + Som(Fg(A)) + Som (Fd(A)) );
}

/*-----*/
/*  Profondeur utilisant le parcours "Preordre"    */
/*-----*/

void Profondeur1 ( struct Noeud *A)
{
    D++;
    fprintf(Fs, "Visite de %d\n", Info(A));
    fprintf(Fs, "D = %d\n", D );
    if ( Fg(A) == NULL && Fd(A) == NULL )
        if (Max < D) Max = D ;
    if ( Fg(A) != NULL )
    {
        Profondeur1(Fg(A) );
        D--;
        /* Au retour, on fait -1. C'est une remontée par la gauche*/
    }
    if ( Fd(A) != NULL )
    {

```

```

        Profondeur1(Fd(A) );
        D--;
        /* Au retour, on fait -1. C'est une remontée par la droite*/
    }
}

int Maxi ( int X, int Y)
{ if ( X < Y ) return( Y ); return( X ) ; }

int Profondeur2 ( struct Noeud *A)
{
    if ( A == NULL )
        return(-1);
    else return( Maxi( 1 + Profondeur2(Fg(A)), 1 + Profondeur2(Fd(A) ))) ;
}

/*-----*/
/*    Parcours Préordre, Inordre et Postordre    */
/*    avec utilisation d'une pile                */
/*-----*/

/* Préordre */
void Preordre1 ( struct Noeud *Racine)
{
    struct Noeud *N;
    struct Pile P;
    Bool Possible;

    Creerpile(&P);
    N = Racine;
    Possible = True;

    while ( Possible)
    {
        while ( N != NULL )
        {
            /* Parcours D'une épine gauche avec*/
            /* sauvegrade des noeuds parcourus */
            fprintf(Fs, "Visite de %d\n", Info(N) );
            if (Fd(N) != NULL) Empiler(&P, Fd(N)) ;
            N = Fg(N);
        }
        Depiler1(&P, &N, &Possible);
    }
}

/* Inordre */
void Inordre1 ( struct Noeud *Racine)
{
    struct Noeud *N;

```

```

struct Pile P;
Bool Possible;

Creerpile(&P);
N = Racine;
Possible = True;
while ( Possible)
{
    while ( N != NULL )
    {
        Empiler(&P, N) ; /* Parcours d'une épine gauche avec*/
        N = Fg(N);      /* sauvegrade des noeuds parcourus */
    }
    if ( ! Pilevide(P) )
    {
        Depiler(&P, &N);
        fprintf(Fs, "Visite de %d\n", Info(N) );
        N = Fd(N);
    }
    else Possible = False;
}
}

/* Postordre */
void Postordre1 ( struct Noeud *Racine)
{
    struct Noeud *N, *Nprime, *Q;
    struct Pile P;
    Bool Possible, Cond;

    Creerpile(&P);
    N = Racine;
    Possible = True;
    while ( Possible)
    {
        while ( N != NULL )
        {
            Empiler(&P, N) ;
            Nprime = N;
            N = Fg(N);
        }
        if ( ! Pilevide(P) )
        {
            if ( Fd(Nprime) == NULL)
            {
                Cond = True;
                while ( !Pilevide(P) && Cond )
                {
                    fprintf(Fs, "Visite de %d\n", Info(Nprime) );
                    Q = Nprime;

```

```

        Depiler(&P, &Nprime);
        if (!Pilevide(P) )
        {
            Nprime = Top(P);
            Cond = ( Fd(Nprime) == Q);
        }
        else Possible = False;
    }
    }
    N = Fd(Nprime) ;
}
else Possible = False;
}
}

main()
{
    Fs = fopen("R_arbre1.C","w");
    Nbrnoeud, Feuille, Somme = 0;
    Max, D = -1;
    Racine = NULL;
    fprintf(Fs, "Création d'une liste de nombres en RAM . . . \n");
    Insert(3,&Racine); Insert(5,&Racine); Insert(15,&Racine);
    Insert(78,&Racine); Insert(43,&Racine); Insert(12,&Racine);
    Insert(233,&Racine); Insert(133,&Racine); Insert(32,&Racine);
    fprintf(Fs, "\n");
    fprintf(Fs, "Impression des éléments de l'arbre créé \n");
    Lister (Racine);
    fprintf(Fs, "\n");
    Preordre(Racine);
    fprintf(Fs, "Parcours en préordre et calcul . . . \n");
    fprintf(Fs, "Nombre de noeuds : %d\n", Nbrnoeud);
    fprintf(Fs, "Nombre de feuilles : %d\n", Feuille);
    fprintf(Fs, "Somme des éléments : %d\n", Somme);
    fprintf(Fs, "\n");
    fprintf(Fs, "Fonctions récursives pour déterminer . . . \n");
    fprintf(Fs, "Nombre de noeuds : %d\n", Nbrnoeud(Racine) );
    fprintf(Fs, "Nombre de feuilles : %d\n", Nbrfeuille(Racine));
    fprintf(Fs, "Somme des éléments : %d\n", Som(Racine));
    fprintf(Fs, "\n");
    fprintf(Fs, "Trace pour le calcul de la profondeur ( D ) \n");
    Profondeur1(Racine);
    fprintf(Fs, "Profondeur1 : %d\n", Max );
    fprintf(Fs, "\n");
    fprintf(Fs, "Une autre façon de calculer la profondeur \n");
    fprintf(Fs, "Profondeur2 : %d\n", Profondeur2(Racine) );
    fprintf(Fs, "\n");
    fprintf(Fs, "Parcours en Inordre avec une pile \n");
    Inordre1(Racine) ;
    fprintf(Fs, "\n");
}

```

```

    fprintf(Fs, "Parcours en Préordre avec une pile \n");
    Preordre1(Racine) ;
    fprintf(Fs, "\n");
    fprintf(Fs, "Parcours en Postordre avec une pile \n");
    Postordre1(Racine) ;
    fclose(Fs);
}

```

Résultats : (Contenu du fichier R_arbre1.c)

Création d'une liste de nombres en RAM . . .

Impression des éléments de l'arbre créé

```

3
5
12
15
32
43
78
133
233

```

Parcours en préordre et calcul . . .

Nombre de noeuds : 9

Nombre de feuilles : 3

Somme des éléments : 554

Fonctions récursives pour déterminer . . .

Nombre de noeuds : 9

Nombre de feuilles : 3

Somme des éléments : 554

Trace pour le calcul de la profondeur (D)

Visite de 3

D = 0

Visite de 5

D = 1

Visite de 15

D = 2

Visite de 12

D = 3

Visite de 78

D = 3

Visite de 43

D = 4

Visite de 32

D = 5

Visite de 233

D = 4

Visite de 133
D = 5
Profondeur1 : 5

Une autre façon de calculer la profondeur
Profondeur2 : 5

Parcours en Inordre avec une pile

Visite de 3
Visite de 5
Visite de 12
Visite de 15
Visite de 32
Visite de 43
Visite de 78
Visite de 133
Visite de 233

Parcours en Préordre avec une pile

Visite de 3
Visite de 5
Visite de 15
Visite de 12
Visite de 78
Visite de 43
Visite de 32
Visite de 233
Visite de 133

Parcours en Postordre avec une pile

Visite de 12
Visite de 32
Visite de 43
Visite de 133
Visite de 233
Visite de 78
Visite de 15
Visite de 5
Visite de 3

EXERCICE 7.



Représentation d'une liste de nombres par un arbre de recherche binaire

On peut représenter une liste de nombres par un arbre binaire (représentation dynamique) de la façon suivante :

Les éléments de la liste sont au niveau des feuilles. Chaque noeud qui n'est pas une feuille contient le nombre de feuilles de son sous arbre gauche.

Programmer l'algorithme de construction d'un tel arbre en montrant les différentes étapes. Appliquer un algorithme de parcours pour montrer la validité de l'algorithme de construction

Considérations :

. On convient d'ajouter, au niveau de chaque noeud interne, le nombre de feuilles de son sous arbre droit. Ceci dans le but de faciliter l'algorithme de construction.

. Une file d'attente est utile pour l'algorithme de construction.

Pascale

```
*****
*  1. LE PROGRAMME      *
*  2. LES DONNEES       *
*  3. LES RESULTATS     *
*****
```

PROGRAMME 7

```
{
    Les arbres

    Création d'un arbre à partir d'une suite de nombres.
    Les nombres sont au niveau des feuilles.
    Chaque noeud interne contient deux champs :
    - le nombre de feuilles à sa gauche et
    - le nombre de feuilles à sa droite.

}
```

```
PROGRAM Creer_un_arbre_binaire_a_partir_d_une_liste;
VAR
    Fs : TEXT;
```

```
{ Modèle sur les files d'attente }
```

```
TYPE
    T = ^Noeud;
    Noeud = RECORD
        Element : INTEGER;
        Lcount : INTEGER;
```

```
Rcount : INTEGER;  
Fg, Fd : T ;  
END;
```

```
T1 = ^Elm ;  
Elm = RECORD  
  Val : T;  
  Suiv : T1  
END;
```

```
Filedattente = RECORD  
  Tete, Queue : T1  
END;
```

```
PROCEDURE Creerfile(VAR Fil : Filedattente);  
  BEGIN  Fil.Tete := NIL  END;
```

```
FUNCTION Filevide (Fil : Filedattente) : BOOLEAN;  
  BEGIN  Filevide := Fil.Tete = NIL  END;
```

```
PROCEDURE Enfiler (VAR Fil : Filedattente; Val : T );  
  VAR  
    P : T1;  
  BEGIN  
    NEW(P);  
    P^.Val := Val;  
    P^.Suiv := NIL;  
    IF NOT Filevide(Fil)  
    THEN Fil.Queue^.Suiv := P  
    ELSE Fil.Tete := P;  
    Fil.Queue := P;  
  END;
```

```
PROCEDURE Defiler (VAR Fil : Filedattente ; VAR Val : T );  
  BEGIN  
    IF NOT Filevide(Fil)  
    THEN  
      BEGIN  
        Val := Fil.Tete^.Val;  
        Fil.Tete := Fil.Tete^.Suiv;  
      END  
    ELSE WRITELN(Fs,' File Vide ');  
  END;
```

```
PROCEDURE Mettre_entete_def ( VAR Fil : Filedattente; Q : T);  
  VAR  
    P : T1;  
  BEGIN { Suppose la file F non vide }  
    NEW(P) ;  
    P^.Val := Q;
```

```

    P^.Suiv := Fil.Tete;
    Fil.Tete := P;
END;

```

{ Modèle sur les arbres binaires }

```

FUNCTION Info(P : T) : INTEGER;
    BEGIN Info := P^.Element END;

```

```

FUNCTION Lcount (P : T) : INTEGER;
    BEGIN Lcount := P^.Lcount END;

```

```

FUNCTION Rcount (P : T) : INTEGER;
    BEGIN Rcount := P^.Rcount END;

```

```

FUNCTION Fg( P : T) : T;
    BEGIN Fg := P^.Fg END;

```

```

FUNCTION Fd( P : T) : T;
    BEGIN Fd := P^.Fd END;

```

```

PROCEDURE Affinfo ( VAR P : T; Val : INTEGER);
    BEGIN P^.Element := Val END;

```

```

PROCEDURE Afflcount ( VAR P : T; Val : INTEGER);
    BEGIN P^.Lcount := Val END;

```

```

PROCEDURE Affrcount ( VAR P : T; Val : INTEGER);
    BEGIN P^.Rcount := Val END;

```

```

PROCEDURE Aff_fg( VAR P : T; Q : T);
    BEGIN P^.Fg := Q END;

```

```

PROCEDURE Aff_fd( VAR P : T; Q : T);
    BEGIN P^.Fd := Q END;

```

```

FUNCTION Creernoead( Val : INTEGER) : T;
    VAR
        P : T;
    BEGIN
        NEW ( P );
        Creernoead := P ;
        P^.Element := Val;
        P^.Fg := NIL;
        P^.Fd := NIL;
        P^.Lcount:= 0;
        P^.Rcount := 0;
    END;

```

{ Listage de l'arbre en inordre }

```
PROCEDURE Lister ( A : T);
```

```
  BEGIN
```

```
    IF A <> NIL
```

```
    THEN
```

```
      BEGIN
```

```
        Lister ( Fg(A));
```

```
        WRITELN(Fs,'Info = ', Info(A) );
```

```
        WRITELN(Fs,' Lcount = ', Lcount(A) );
```

```
        WRITELN(Fs,' Rcount = ', Rcount(A) );
```

```
        Lister ( Fd(A) )
```

```
      END;
```

```
  END;
```

```
{ Listage des éléments de la file d'attente pour le niveau D }
```

```
PROCEDURE Listerfile ( Fil : Filedattente; D :INTEGER );
```

```
  VAR
```

```
    P : T1;
```

```
  BEGIN
```

```
    WRITELN(Fs);
```

```
    WRITELN(Fs,'Contenu de la file d"attente pour le niveau', D :2);
```

```
    P := Fil.Tete;
```

```
    WHILE P <> NIL DO
```

```
      BEGIN
```

```
        WRITELN(Fs,'Info = ', Info(P^.Val),
```

```
          ' Lcompte = ', Lcount(P^.Val ),' Rcompte = ', Rcount(P^.Val));
```

```
        P := P^.Suiv;
```

```
      END;
```

```
    WRITELN(Fs,'Fin de file');
```

```
  END;
```

```
{ Création d'un arbre à partir d'une suite de nombres en vue d'effectuer avec efficacité  
l'opération de recherche du K-ième grand élément }
```

```
PROCEDURE List_arbre;
```

```
FUNCTION Deuxpuissancej ( J : INTEGER ) : INTEGER;
```

```
  VAR
```

```
    I,K : INTEGER;
```

```
  BEGIN
```

```
    K := 1;
```

```
    FOR I := 1 TO J DO K := K * 2;
```

```
    Deuxpuissancej := K
```

```
  END;
```

```
VAR
```

```
  S, Puis, D : INTEGER;
```

```
  I, J, K : INTEGER;
```

```
  Fil : Filedattente;
```

```
  P, P1, P2, Q : T;
```

```

N : INTEGER;
Ok : BOOLEAN;
BEGIN
N := 9; { Nombre d'éléments }
S := 1 ;

{ Déterminer le plus petit D tel que  $2D \geq N$  }
Puis := 1;
D := 0;
WHILE S < N-1 DO
BEGIN
D := D + 1;
Puis := Puis * 2;
S := S + Puis
END;

WRITELN(Fs, ' --> Profondeur de l'arbre à créer D=', D);

Creerfile(Fil);
FOR I := 1 TO N DO
BEGIN
P := Creernoead(I);
Enfiler(Fil, P)
END;
Listerfile(Fil, D); { Pour la trace }
FOR I := Deuxpuissancej( D) TO N-1 DO
{ N- 1 : Nombre de noeuds internes }
BEGIN
Q := Creernoead(0);
Defiler(Fil, P1);
Defiler(Fil, P2);
Aff_fg(Q, P1);
Aff_fd(Q, P2);
Afflcount(Q, 1);
Affrcount(Q, 1);
Mettre_entete_def (Fil, Q);
END;
Ok := TRUE;
FOR J := D-1 DOWNTO 0 DO
BEGIN
FOR K := Deuxpuissancej(J) TO Deuxpuissancej(J+1) - 1 DO
BEGIN
Q := Creernoead(0);
Defiler(Fil, P1);
Defiler(Fil, P2);
Aff_fg(Q, P1);
Aff_fd(Q, P2);
IF Ok
THEN
BEGIN

```

```

    IF Odd(N) AND ( K = N DIV 2 )
    THEN
        BEGIN
            Afflcount( Q,2 );
            Affrcount( Q, 1);
        END
    ELSE
        BEGIN
            Afflcount(Q, 1);
            Affrcount(Q, 1)
        END;
    END
ELSE
    BEGIN
        Afflcount(Q, Lcount(P1) + Rcount(P1) );
        Affrcount(Q, Lcount(P2) + Rcount(P2) )
    END;
    Enfiler(Fil, Q)
END;
Ok := FALSE;
Listerfile(Fil, J);
END;
WRITELN(Fs);
WRITELN(Fs,'Arbre créé : ');
WRITELN(Fs);
Lister(Q);
END;

{ Programme principal }
BEGIN
    ASSIGN(Fs, 'R_arbre1.Pas');
    REWRITE(Fs);
    WRITELN(Fs,
    'Création d'un arbre à partir de la liste 1,2,3,...9');
    WRITELN(Fs,
    'Chaque noeud interne contient le nombre de feuilles à sa gauche');
    WRITELN(Fs,'Chaque noeud externe contient un élément de la liste à créer');
    WRITELN(Fs);
    List_arbre;
    CLOSE(Fs)
END.

```

Résultats (Contenu du fichier R_arbre1.pas) :

Création d'un arbre à partir de la liste 1,2,3,...9
 Chaque noeud interne contient le nombre de feuilles à sa gauche
 Chaque noeud externe contient un élément de la liste à créer

--> Profondeur de l'arbre à créer D=3

Contenu de la file d'attente pour le niveau 3

Info = 1 Lcompte = 0 Rcompte = 0

Info = 2 Lcompte = 0 Rcompte = 0

Info = 3 Lcompte = 0 Rcompte = 0

Info = 4 Lcompte = 0 Rcompte = 0

Info = 5 Lcompte = 0 Rcompte = 0

Info = 6 Lcompte = 0 Rcompte = 0

Info = 7 Lcompte = 0 Rcompte = 0

Info = 8 Lcompte = 0 Rcompte = 0

Info = 9 Lcompte = 0 Rcompte = 0

Fin de file

Contenu de la file d'attente pour le niveau 2

Info = 0 Lcompte = 2 Rcompte = 1

Info = 0 Lcompte = 1 Rcompte = 1

Info = 0 Lcompte = 1 Rcompte = 1

Info = 0 Lcompte = 1 Rcompte = 1

Fin de file

Contenu de la file d'attente pour le niveau 1

Info = 0 Lcompte = 3 Rcompte = 2

Info = 0 Lcompte = 2 Rcompte = 2

Fin de file

Contenu de la file d'attente pour le niveau 0

Info = 0 Lcompte = 5 Rcompte = 4

Fin de file

Arbre créé :

Info = 1

Lcount = 0

Rcount = 0

Info = 0

Lcount = 1

Rcount = 1

Info = 2

Lcount = 0

Rcount = 0

Info = 0

Lcount = 2

Rcount = 1

Info = 3

Lcount = 0

Rcount = 0

Info = 0

Lcount = 3

Rcount = 2

Info = 4

Lcount = 0

```

    Rcount = 0
Info = 0
    Lcount = 1
    Rcount = 1
Info = 5
    Lcount = 0
    Rcount = 0
Info = 0
    Lcount = 5
    Rcount = 4
Info = 6
    Lcount = 0
    Rcount = 0
Info = 0
    Lcount = 1
    Rcount = 1
Info = 7
    Lcount = 0
    Rcount = 0
Info = 0
    Lcount = 2
    Rcount = 2
Info = 8
    Lcount = 0
    Rcount = 0
Info = 0
    Lcount = 1
    Rcount = 1
Info = 9
    Lcount = 0
    Rcount = 0

```

Solution C

```

*****
*  1. LE PROGRAMME      *
*  2. LES DONNEES       *
*  3. LES RESULTATS     *
*****

```

PROGRAMME 3

```

#include <Alloc.H>
#include <Stdlib.H>

```



```

#include <Stdio.H>

#define True 1
#define False 0

typedef int Bool;

struct Noeud
{
    int Element ;
    int Lcount, Rcount;
    struct Noeud *Fg ;
    struct Noeud *Fd ;
} ;

struct Filedattente
{
    struct Maillon *Tete, *Queue;
} ;

struct Maillon
{
    struct Noeud *Val ;
    struct Maillon *Suiv ;
} ;

struct Noeud *P, *P1, *P2, *Q;
int I, J, K, S, Puis, D, N;
Bool Ok;
struct Filedattente F;
FILE *Fs;

/*-----*/
/* Procedures d'implémentation du modèle sur les listes */
/*-----*/

struct Maillon *Allouer ( )
{
    return ( (struct Maillon *) malloc( sizeof(struct Maillon)) );
}

void Affval(struct Maillon *P, struct Noeud *V)
{ P->Val =V; }

void Affadr( struct Maillon *P, struct Maillon *Q)
{ P->Suiv = Q; }

struct Maillon *Suivant( struct Maillon *P)
{ return( P->Suiv ); }

```

```

struct Noeud *Valeur( struct Maillon *P)
{ return( P->Val) ; }

/*-----*/
/* Procédures d'implémentation du modèle sur les arbres binaires */
/*-----*/

struct Noeud *Creernoeud ( int Val)
{
    struct Noeud *P;
    P = (struct Noeud *) malloc( sizeof(struct Noeud)) ;
    P->Element = Val ;
    P->Fg = NULL;
    P->Fd = NULL;
    return (P) ;
}

struct Noeud *Fg ( struct Noeud *P)
{ return ( P->Fg ); }

struct Noeud *Fd ( struct Noeud *P)
{ return ( P->Fd ) ; }

int Info ( struct Noeud *P)
{ return ( P->Element ); }

void Affinfo(struct Noeud *P, int V)
{ P->Element =V; }

void Aff_fg(struct Noeud *P, struct Noeud *Q)
{ P->Fg = Q; }

void Aff_fd(struct Noeud *P, struct Noeud *Q)
{ P->Fd = Q; }

int Lcount(struct Noeud *P)
{ return ( P->Lcount ) ; }

int Rcount(struct Noeud *P)
{ return ( P->Rcount ) ; }

void Aff_lcount( struct Noeud *P, int Val)
{ P->Lcount = Val ; }

void Aff_rcount( struct Noeud *P, int Val)
{ P->Rcount = Val ; }

/*-----*/
/*          Listage en ordre croissant          */
/*-----*/

```

```

void Lister ( struct Noeud *A)
{
    if ( A != NULL )
    {
        Lister ( Fg(A) ) ;
        if ( Fg(A) == NULL && Fd(A) == NULL)
            fprintf (Fs, " Feuille <%d> \n", Info(A) ) ;
        else fprintf (Fs, " Noeud interne (%d,%d) \n",  Lcount(A),Rcount(A) ) ;
        Lister ( Fd(A) );
    }
}

```

```

/*-----*/
/* Procédures d'implémentation du modèle sur les files d'attente */
/*-----*/

```

```

void Creerfile( struct Filedattente *F)
{
    struct Filedattente Q ;
    Q=*F;
    Q.Tete = NULL ;
    *F = Q;
}

```

```

Bool Filevide(struct Filedattente F)
{ return ( F.Tete == NULL ); }

```

```

void Enfiler ( struct Filedattente *F, struct Noeud *Val)
{
    struct Filedattente Q;
    struct Maillon *P;
    P = Allouer();
    Affval(P, Val);
    Affadr(P, NULL);

    Q = *F;
    if ( !Filevide(*F) )
        Affadr( Q.Queue, P);
    else Q.Tete = P;
    Q.Queue = P;
    *F = Q;
}

```

```

void Defiler ( struct Filedattente *F, struct Noeud **Val)
{
    struct Filedattente Q;
    Q = *F;
    if ( !Filevide(*F) )
    {

```

```

        *Val = Valeur(Q.Tete) ;
        Q.Tete = Suivant(Q.Tete);
        *F = Q ;
    }
    else { fprintf(Fs, "File Vide\n"); exit(0) ;}
}

```

```

void Mettre_en_tetedef (struct Filedattente *F, struct Noeud *Q)

```

```

{
    struct Maillon *P;
    struct Filedattente Qq;
    /* Suppose La File Non Vide */
    Qq = *F;
    P = Allouer();
    Affval(P, Q);
    Affadr(P, Qq.Tete);
    Qq.Tete = P;
    *F = Qq ;
}

```

```

/* Imprime les éléments de la file d'attente */

```

```

void Listerfile( struct Filedattente F)

```

```

{
    struct Maillon *P;
    P = F.Tete;
    while( P != NULL)
    {
        fprintf(Fs, "[ %d , ( %d , %d ) ] \n", Info(Valeur(P)),
            Lcount(Valeur(P)), Rcount(Valeur(P)) );
        P = Suivant(P);
    }
}

```

```

int Deuxpuissance (int J)

```

```

{
    int I, K;
    K = 1;
    for ( I=1; I<=J; ++I ) K *= 2;
    return(K);
}

```

```

main()

```

```

{
    Fs = fopen("R_arbre2.C", "w");
    N = 9;
    S = 1; Puis = 1; D = 0;
    while ( S < N-1 )
    {
        D++; Puis *= 2; S+=Puis;
    }
}

```

```

fprintf(Fs,"Détermination de la profondeur (D) \n");
fprintf(Fs,"D = %d\n", D);
fprintf(Fs,"\n");

Creerfile(&F);
for ( I=1; I <=N ; I++ )
{
    P = Creernoead(I);
    Aff_lcount(P, 0);
    Aff_rcount(P, 0);
    Enfiler(&F, P);
}
fprintf(Fs,"Eléments de la file d'attente \n");
Listerfile(F);
fprintf(Fs,"\n");
for (I = Deuxpuissance(D); I<= N-1; I++)
{
    Q = Creernoead(0);
    Defiler(&F, &P1);
    Defiler(&F, &P2);
    Aff_fg(Q, P1);
    Aff_fd(Q, P2);
    Aff_lcount(Q, 1);
    Aff_rcount(Q, 1);
    Mettre_en_tetedef (&F, Q);
}
Ok = True;
for ( J=D-1; J>=0; J--)
{
    for (K=Deuxpuissance(J); K <= Deuxpuissance(J+1)-1;K++)
    {
        Q = Creernoead(0);
        Defiler(&F, &P1);
        Defiler(&F, &P2);
        Aff_fg(Q, P1);
        Aff_fd(Q, P2);
        if (Ok)
            if ( ((N % 2)!=0) && (K == N / 2) )
            {
                Aff_lcount(Q, 2);/* 2*/
                Aff_rcount(Q, 1);
            }
            else
            {
                Aff_lcount(Q, 1);
                Aff_rcount(Q, 1);
            }
            else
            {
                Aff_lcount(Q, Lcount(P1) + Rcount(P1) );
            }
        }
    }
}

```

```

        Aff_rcount(Q, Lcount(P2) + Rcount(P2) ) ;
    }
    Enfiler(&F, Q);
}
Ok = False;
fprintf(Fs,"Eléments de la file d'attente \n");
Listerfile( F) ;
fprintf(Fs,"\n");
};
fprintf(Fs," Listage des éléments de l'arbre créé \n");
Lister(Q);
}

```

Résultats : (Contenu du fichier R_arbre2.c)

Détermination de la profondeur (D)

D = 3

Eléments de la file d'attente

```

[ 1 , ( 0 , 0 ) ]
[ 2 , ( 0 , 0 ) ]
[ 3 , ( 0 , 0 ) ]
[ 4 , ( 0 , 0 ) ]
[ 5 , ( 0 , 0 ) ]
[ 6 , ( 0 , 0 ) ]
[ 7 , ( 0 , 0 ) ]
[ 8 , ( 0 , 0 ) ]
[ 9 , ( 0 , 0 ) ]

```

Eléments de la file d'attente

```

[ 0 , ( 2 , 1 ) ]
[ 0 , ( 1 , 1 ) ]
[ 0 , ( 1 , 1 ) ]
[ 0 , ( 1 , 1 ) ]

```

Eléments de la file d'attente

```

[ 0 , ( 3 , 2 ) ]
[ 0 , ( 2 , 2 ) ]

```

Eléments de la file d'attente

```

[ 0 , ( 5 , 4 ) ]

```

Listage des éléments de l'arbre créé

Feuille <1>

Noeud interne (1,1)

Feuille <2>

Noeud interne (2,1)

Feuille <3>

Noeud interne (3,2)

Feuille <4>

Noeud interne (1,1)
 Feuille <5>
 Noeud interne (5,4)
 Feuille <6>
 Noeud interne (1,1)
 Feuille <7>
 Noeud interne (2,2)
 Feuille <8>
 Noeud interne (1,1)
 Feuille <9>

EXERCICE 8.

solution PASCAL

Hachage par essai linéaire

Programmer les algorithmes de recherche, insertion et suppression dans la méthode de hachage par essai linéaire. On montrera les différentes étapes pour insérer et supprimer des éléments.

Nous rappelons, pour cette méthode, que s'il se produit une collision sur la case d'indice k d'un tableau $T[0..M-1]$, on insère la donnée, si elle n'existe pas, dans la première case libre fournie par la séquence cyclique

$h(k)-1, \dots, 0, M-1, M-2, \dots, h(k)+1$

h désigne la fonction de hachage.

```
*****
* 1. LE PROGRAMME      *
* 2. LES DONNEES       *
* 3. LES RESULTATS     *
*****
```

PROGRAMME 8

```
{
    Essai linéaire

    Hachage par l'essai linéaire
    Recherche, insertion et suppression
}
```

```

PROGRAM Essai_lineaire;
CONST
  M = 10;

TYPE
  Typecle = INTEGER;
  Typelement = RECORD
    Cle : Typecle;
    Vide : BOOLEAN
  END;

VAR
  T : ARRAY[0..9] OF Typelement;
  N : INTEGER; { Nombre d'éléments présents dans la table }
  I : INTEGER;
  Fs : TEXT ;

{ Imprime la table }
PROCEDURE Imprimer ;
VAR
  I : INTEGER;
BEGIN
  WRITELN(Fs) ; WRITELN(Fs);
  WRITELN(Fs,'          Contenu de la table ');
  WRITELN(Fs,'          -----');
  FOR I := 0 TO M-1 DO
    BEGIN
      WRITE(Fs,I : 5, ' : ');
      IF T(I).Vide
      THEN WRITELN(Fs,'  ')
      ELSE WRITELN(Fs,T(I).Cle : 5);
    END;
  END;

{ Fonction de hachage }
FUNCTION Hacher ( K : Typecle ) : INTEGER;
BEGIN
  Hacher := K MOD M
END;

{ Recherche la clé K dans la table. Si K est trouvée, Trouv est positionnée à VRAI.
  I pointe soit l'élément trouvé, soit une position ouverte }

PROCEDURE Recherche ( K : Typecle; VAR Trouv : BOOLEAN;
  VAR I : INTEGER );
BEGIN
  I := Hacher(K);
  WRITELN(Fs, '* H(',K:3,')=',I);
  WHILE ( ( T(I).Cle <> K ) AND ( NOT T(I).Vide ) ) DO

```



```

BEGIN
  I := I - 1;
  IF I < 0 THEN I := I + M
END;
Trouv := T(I).Cle = K
END;

```

{ Insère la clé K dans la table }

```

PROCEDURE Insérer ( K : Typecle);
VAR
  I : INTEGER;
  Trouv : BOOLEAN;
BEGIN
  WRITELN(Fs); WRITELN(Fs);
  WRITELN(Fs,' ***** Insertion de la clé ', K : 3, ' *****');
  WRITELN(Fs);
  Recherche( K, Trouv, I);
  WRITELN(Fs,'* Recherche de la clé ',K, '====> Trouv = ', Trouv, ' I = ', I);
  IF Trouv
  THEN WRITELN(Fs,' La clé ', K : 3, 'existe')
  ELSE
    IF N = M-1
    THEN
      WRITELN(Fs,
        'Il reste une case libre ==> Débordement( une position est sacrifiée)' )
    ELSE
      BEGIN
        IF I<>Hacher(K)
        THEN
          WRITELN(Fs,
            '* Il se produit une collision sur la case ',
            Hacher(K));
          WRITELN(Fs, '* La clé ', K : 3,
            ' est insérée à la position ',I:3);
          N := N + 1;
          T(I).Cle := K;
          T(I).Vide := FALSE;
        END;
      END;
    END;
  END;

```

{ Supprime la clé K dans la table }

```

PROCEDURE Supprimer ( K : INTEGER );
VAR
  I, J, R, Etape : INTEGER;
  Trouv, Sort, Continue : BOOLEAN;
BEGIN
  WRITELN(Fs); WRITELN(Fs);
  WRITELN(Fs,' ***** Suppression de la clé ', K : 3, ' *****');
  WRITELN(Fs);

```

```

Recherche(K, Trouv, I);
WRITELN(Fs, '* Recherche de la clé ', K, ' ====> Trouv = ', Trouv, ' I = ', I);

IF NOT Trouv
THEN WRITELN(Fs, 'Clé inexistante')
ELSE
BEGIN
  Etape := 0;
  Continue := TRUE;
  WHILE Continue DO
  BEGIN
    WRITELN(Fs); WRITELN(Fs);
    Etape := Etape + 1 ;
    WRITELN(Fs, '          Etape ', Etape : 3);
    WRITELN(Fs, '          ----');
    WRITELN(Fs, '- Marquer T('I:2,') vide et sauver ', 'I = ', I:2, ' dans J ');
    { Marquer T(I) vide et sauver I dans J }
    T(I.I).Vide := TRUE;
    J := I;

    WRITELN(Fs,
'- Rechercher (soit I) parmi les cases I-1,I-2, ... une clé qui est un');
    WRITELN(Fs,
' débordement d"une case R =H(T(I).Cle) telle que R >=',
J:2, ' ou R < I. ');
    WRITELN(Fs,
' Ou bien cette case I est trouvée ou bien une case vide est rencontrée');
    WRITELN(Fs); WRITELN(Fs);
    WRITELN(Fs, '** Parcours des cases ... ');

    I := I-1; IF I<0 THEN I := I + M;
    { Rechercher une clé aux positions I-1, I-2, ... qui est un débordement des cases
      R >= J ou R < I c'est à dire des cases qui ne lient pas cycliquement I et J }

    Sort := FALSE;
    WHILE ( (NOT T(I.I).Vide) AND ( NOT Sort) ) DO
    BEGIN
      WRITE(Fs, ' I : ', I:3);
      R := Hacher( T(I.I).Cle);
      WRITE (Fs, ' R=', R);
      IF ( R < J) AND (R >=I)
      THEN
      BEGIN
        WRITELN(Fs, ' . Condition non vérifiée');
        I := I-1; IF I<0 THEN I := I + M;
      END
      ELSE
      BEGIN
        WRITELN(Fs, ' . Condition vérifiée' );
        Sort := TRUE
      END
    END
  END

```

```

        END
    END;
    IF T(I).Vide
    THEN WRITELN(Fs,'
        Une position ouverte est rencontrée : fin de la suppression')
    ELSE
    BEGIN
        WRITELN(Fs, ' La clé recherchée (en débordement)
trouvée : ', T(I).Cle, ' à la position ', I:2);
        WRITELN(Fs,'- Transférer T(',I:2,')= ',T(I).Cle:3,
            ' à la position', J :3,' de la table');
        END;

        IF T(I).Vide
        THEN Continue := FALSE
        ELSE BEGIN T(J.) := T(I.) END;
    END;
    END;
END;

BEGIN
    ASSIGN(Fs, 'R_essail.Pas');
    REWRITE(Fs) ;
    N := 0;
    FOR I := 0 TO M-1 DO T(I).Vide := TRUE;
    FOR I := 0 TO M-1 DO T(I).Cle := - 1 ;

    Insérer(13); Insérer(12); Insérer(23); Insérer(35);
    Insérer(4); Insérer(18); Insérer(14); Insérer(5);
    Insérer(1); Insérer(2); Insérer(15); Imprimer;

    Supprimer(35); Supprimer(4); Supprimer(23); Supprimer(18);
    Supprimer(1); Supprimer(12); Supprimer(5); Imprimer;
    CLOSE(Fs)
END.

```

Résultats (Contenu du fichier R_essail.pas) :

```

*****      Insertion de la clé  13      *****

* H( 13)=3
* Recherche de la clé 13 ==> Trouv = FALSE I = 3
* La clé 13 est insérée à la position 3

*****      Insertion de la clé  12      *****

* H( 12)=2
* Recherche de la clé 12 ==> Trouv = FALSE I = 2
* La clé 12 est insérée à la position 2

```

***** Insertion de la clé 23 *****

- * $H(23)=3$
- * Recherche de la clé 23 =====> Trouv = FALSE I = 1
- * Il se produit une collision sur la case 3
- * La clé 23 est insérée à la position 1

***** Insertion de la clé 35 *****

- * $H(35)=5$
- * Recherche de la clé 35 =====> Trouv = FALSE I = 5
- * La clé 35 est insérée à la position 5

***** Insertion de la clé 4 *****

- * $H(4)=4$
- * Recherche de la clé 4 =====> Trouv = FALSE I = 4
- * La clé 4 est insérée à la position 4

***** Insertion de la clé 18 *****

- * $H(18)=8$
- * Recherche de la clé 18 =====> Trouv = FALSE I = 8
- * La clé 18 est insérée à la position 8

***** Insertion de la clé 14 *****

- * $H(14)=4$
- * Recherche de la clé 14 =====> Trouv = FALSE I = 0
- * Il se produit une collision sur la case 4
- * La clé 14 est insérée à la position 0

***** Insertion de la clé 5 *****

- * $H(5)=5$
- * Recherche de la clé 5 =====> Trouv = FALSE I = 9
- * Il se produit une collision sur la case 5
- * La clé 5 est insérée à la position 9

***** Insertion de la clé 1 *****

- * $H(1)=1$

* Recherche de la clé 1 ==> Trouv = FALSE I = 7
 * Il se produit une collision sur la case 1
 * La clé 1 est insérée à la position 7

***** Insertion de la clé 2 *****

* $H(2)=2$
 * Recherche de la clé 2 ==> Trouv = FALSE I = 6
 Il reste une case libre ==> Débordement(une position est sacrifiée)

***** Insertion de la clé 15 *****

* $H(15)=5$
 * Recherche de la clé 15 ==> Trouv = FALSE I = 6
 Il reste une case libre ==> Débordement(une position est sacrifiée)

Contenu de la table

 0 : 14
 1 : 23
 2 : 12
 3 : 13
 4 : 4
 5 : 35
 6 :
 7 : 1
 8 : 18
 9 : 5

***** Suppression de la clé 35 *****

* $H(35)=5$
 * Recherche de la clé 35 ==> Trouv = TRUE I = 5

Etape 1

 - Marquer T(5) vide et sauver I = 5 dans J
 - Rechercher (soit I) parmi les cases I-1, I-2, ... une clé qui est un débordement d'une case $R = H(T(I).Cle)$ telle que $R \geq 5$ ou $R < I$.
 Ou bien cette case I est trouvée ou bien une case vide est rencontrée

** Parcours des cases ...

I : 4 R=4 . Condition non vérifiée
 I : 3 R=3 . Condition non vérifiée

I : 2 R=2 . Condition non vérifiée
 I : 1 R=3 . Condition non vérifiée
 I : 0 R=4 . Condition non vérifiée
 I : 9 R=5 . Condition vérifiée
 La clé recherchée (en débordement) est trouvée : 5 à la position 9
 - Transférer T(9) = 5 à la position 5 de la table

Etape 2

- Marquer T(9) vide et sauver I = 9 dans J
 - Rechercher (soit I) parmi les cases I-1, I-2, ... une clé qui est un débordement d'une case $R = H(T(I).Cle)$ telle que $R \geq 9$ ou $R < I$.
 Ou bien cette case I est trouvée ou bien une case vide est rencontrée

** Parcours des cases ...

I : 8 R=8 . Condition non vérifiée
 I : 7 R=1 . Condition vérifiée
 La clé recherchée (en débordement) est trouvée : 1 à la position 7
 - Transférer T(7) = 1 à la position 9 de la table

Etape 3

- Marquer T(7) vide et sauver I = 7 dans J
 - Rechercher (soit I) parmi les cases I-1, I-2, ... une clé qui est un débordement d'une case $R = H(T(I).Cle)$ telle que $R \geq 7$ ou $R < I$.
 Ou bien cette case I est trouvée ou bien une case vide est rencontrée

** Parcours des cases ...

Une position ouverte est rencontrée : fin de la suppression

***** Suppression de la clé 4 *****

* $H(4) = 4$
 * Recherche de la clé 4 ==> Trouv = TRUE I = 4

Etape 1

- Marquer T(4) vide et sauver I = 4 dans J
 - Rechercher (soit I) parmi les cases I-1, I-2, ... une clé qui est un débordement d'une case $R = H(T(I).Cle)$ telle que $R \geq 4$ ou $R < I$.
 Ou bien cette case I est trouvée ou bien une case vide est rencontrée

** Parcours des cases ...

I : 3 R=3 . Condition non vérifiée
 I : 2 R=2 . Condition non vérifiée
 I : 1 R=3 . Condition non vérifiée
 I : 0 R=4 . Condition vérifiée
 La clé recherchée (en débordement) est trouvée : 14 à la position 0
 - Transférer T(0)= 14 à la position 4 de la table

Etape 2

- Marquer T(0) vide et sauver I = 0 dans J
 - Rechercher (soit I) parmi les cases I-1, I-2, ... une clé qui est un débordement d'une case $R = H(T(I).Cle)$ telle que $R \geq 0$ ou $R < I$.
 Ou bien cette case I est trouvée ou bien une case vide est rencontrée

** Parcours des cases ...

I : 9 R=1 . Condition vérifiée
 La clé recherchée (en débordement) est trouvée : 1 à la position 9
 - Transférer T(9)= 1 à la position 0 de la table

Etape 3

- Marquer T(9) vide et sauver I = 9 dans J
 - Rechercher (soit I) parmi les cases I-1, I-2, ... une clé qui est un débordement d'une case $R = H(T(I).Cle)$ telle que $R \geq 9$ ou $R < I$.
 Ou bien cette case I est trouvée ou bien une case vide est rencontrée

** Parcours des cases ...

I : 8 R=8 . Condition non vérifiée
 Une position ouverte est rencontrée : fin de la suppression

***** Suppression de la clé 23 *****

* $H(23)=3$
 * Recherche de la clé 23 =====> Trouv = TRUE I = 1

Etape 1

- Marquer T(1) vide et sauver I = 1 dans J
 - Rechercher (soit I) parmi les cases I-1, I-2, ... une clé qui est un débordement d'une case $R = H(T(I).Cle)$ telle que $R \geq 1$ ou $R < I$.
 Ou bien cette case I est trouvée ou bien une case vide est rencontrée

** Parcours des cases ...

I : 0 R=1 . Condition vérifiée

La clé recherchée (en débordement) est trouvée : 1 à la position 0

- Transférer T(0)= 1 à la position 1 de la table

Etape 2

- Marquer T(0) vide et sauver I = 0 dans J

- Rechercher (soit I) parmi les cases I-1,I-2, ... une clé qui est un débordement d'une case $R = H(T(I).Cle)$ telle que $R \geq 0$ ou $R < I$.
Ou bien cette case I est trouvée ou bien une case vide est rencontrée

** Parcours des cases ...

Une position ouverte est rencontrée : fin de la suppression

***** Suppression de la clé 18 *****

* H(18)=8

* Recherche de la clé 18 =====> Trouv = TRUE I = 8

Etape 1

- Marquer T(8) vide et sauver I = 8 dans J

- Rechercher (soit I) parmi les cases I-1,I-2, ... une clé qui est un débordement d'une case $R = H(T(I).Cle)$ telle que $R \geq 8$ ou $R < I$.
Ou bien cette case I est trouvée ou bien une case vide est rencontrée

** Parcours des cases ...

Une position ouverte est rencontrée : fin de la suppression

***** Suppression de la clé 1 *****

* H(1)=1

* Recherche de la clé 1 =====> Trouv = TRUE I = 1

Etape 1

- Marquer T(1) vide et sauver I = 1 dans J

- Rechercher (soit I) parmi les cases I-1,I-2, ... une clé qui est un débordement d'une case $R = H(T(I).Cle)$ telle que $R \geq 1$ ou $R < I$.
Ou bien cette case I est trouvée ou bien une case vide est rencontrée

** Parcours des cases ...

Une position ouverte est rencontrée : fin de la suppression

***** Suppression de la clé 12 *****

* $H(12)=2$

* Recherche de la clé 12 =====> Trouv = TRUE I = 2

Etape 1

- Marquer T(2) vide et sauver I = 2 dans J
- Rechercher (soit I) parmi les cases I-1, I-2, ... une clé qui est un débordement d'une case $R = H(T(I).Cle)$ telle que $R \geq 2$ ou $R < I$.
Ou bien cette case I est trouvée ou bien une case vide est rencontrée

** Parcours des cases ...

Une position ouverte est rencontrée : fin de la suppression

***** Suppression de la clé 5 *****

* $H(5)=5$

* Recherche de la clé 5 =====> Trouv = TRUE I = 5

Etape 1

- Marquer T(5) vide et sauver I = 5 dans J
- Rechercher (soit I) parmi les cases I-1, I-2, ... une clé qui est un débordement d'une case $R = H(T(I).Cle)$ telle que $R \geq 5$ ou $R < I$.
Ou bien cette case I est trouvée ou bien une case vide est rencontrée

** Parcours des cases ...

I : 4 R=4 . Condition non vérifiée

I : 3 R=3 . Condition non vérifiée

Une position ouverte est rencontrée : fin de la suppression

Contenu de la table

0 :

1 :

2 :

3 : 13

4 : 14

5 :

6 :

7 :
8 :
9 :

EXERCICE 9.

solution PASCAL

Mécanisme de construction d'un arbre B

Programmer l'algorithme d'insertion dans un arbre B d'une suite de nombres générés aléatoirement. Programmer aussi l'algorithme de listage des éléments en ordre croissant dans le but de montrer que la création est bien faite.

Le programme imprimera une trace complète pour montrer le principe de construction de l'arbre B.

On prendra l'ordre 5.

```
*****  
* 1. LE PROGRAMME      *  
* 2. LES DONNEES       *  
* 3. LES RESULTATS     *  
*****
```

PROGRAMME 9

```
{  
Les arbres B
```

```
Création d'un arbre B à partir d'une suite de nombres générés aléatoirement.  
Listage des éléments d'un arbre B
```

```
}
```

```
PROGRAM Recherche_insertion_listage_dans_un_arbreb;
```

```
CONST
```

```
Ordremax = 10;
```

```
Ordremaxplusun = 11;
```

```
Capacite_bloc = 10;
```

```
Nulle = -1;
```

```
Max_niveau = 10 ;
```

```
TYPE
```

```
Typecle = STRING(.5.);
Typearticle = STRING(.5.);
Typepointeur = INTEGER;
Typeadr = INTEGER ;
```

{ Structure d'un noeud de l'arbre B sur le disque }

TYPE

```
Typenoeud = RECORD
  Nbr : INTEGER;
  Tabfils : ARRAY (.1..Ordremaxplusun.) OF Typepointeur;
  Tabcle : ARRAY (.1..Ordremax.) OF Typecle;
  Tabadr : ARRAY (. 1..Ordremax .) OF Typeadr
END;
```

{ Structure d'un bloc du fichier de données }

TYPE

```
Typebloc = RECORD
  Nbr : INTEGER;
  Tabinfo : ARRAY(.1..Capacite_bloc.) OF Typearticle;
END;
```

VAR

```
F_arbre : File OF Typenoeud;
Fichier : File OF Typebloc;
Fs : TEXT;
Noeud : Typenoeud; {Buffer du fichier d'index }
Bloc : Typebloc; {Buffer du fichier de données}
Ordre : INTEGER;
I : INTEGER;
Nbrarticle : INTEGER;
Clef : Typecle;
Arbre : Typepointeur; {Racine de l'arbre B}
Lindex : INTEGER; {Pointe le dernier noeud créé}
L : INTEGER; {Pointe le dernier bloc de données}
```

{ Opérations du modèle }

```
FUNCTION Fils(I :INTEGER ; Noeud : Typenoeud) : Typepointeur;
  BEGIN Fils := Noeud.Tabfils(I.) END;
```

```
FUNCTION Cle(I : INTEGER; Noeud : Typenoeud) : Typecle;
  BEGIN Cle := Noeud.Tabcle(I.); END;
```

```
FUNCTION Adr (I : INTEGER;Noeud : Typenoeud) : Typeadr;
  BEGIN Adr := Noeud.Tabadr(I.); END;
```

```
FUNCTION Nbr (Noeud : Typenoeud) : INTEGER; { Désigne le nombre de clés }
  BEGIN Nbr := Noeud.Nbr; END;
```

```
PROCEDURE Aff_fils(VAR Noeud : Typenoeud; I : INTEGER; J : Typepointeur );
  BEGIN Noeud.Tabfils(I.) := J ; END;
```

```
PROCEDURE Aff_cle( VAR Noeud : Typenoeud ;I : INTEGER; Cle : Typecle);
  BEGIN Noeud.Tabcle(I.) := Cle; END;
```

```
PROCEDURE Aff_adr( VAR Noeud : Typenoeud;I : INTEGER; Adr : Typeadr);
  BEGIN Noeud.Tabadr(I.) := Adr; END;
```

```
PROCEDURE Aff_nbr( VAR Noeud : Typenoeud ; Nb : INTEGER );
  BEGIN Noeud.Nbr := Nb; END;
```

```
FUNCTION Indexdanspere ( F : Typepointeur; Clef : Typecle ) : INTEGER;
VAR
  I : INTEGER;
  Trouv : BOOLEAN;
  Node : Typenoeud;
BEGIN
  IF F <> Nulle
  THEN
    BEGIN
      Seek(F_arbre, F-1);
      READ(F_arbre, Node);
      Trouv := FALSE;
      I := 1;
      WHILE ( (NOT Trouv) AND (I < Nbr(Node) ) )DO
        IF Cle(I,Node)>= Clef THEN Trouv := TRUE ELSE I := I + 1 ;
      Indexdanspere := I ;
    END
  ELSE Indexdanspere := -1
END;
```

{ Implémentation de la pile }

```
TYPE
  Typepile= RECORD
    Som : INTEGER;
    Tab : ARRAY(1..Max_niveau.) OF Typepointeur ;
  END;
VAR
  Pile : Typepile ;      { Une pile }
```

```
PROCEDURE Creerpile ( VAR P : Typepile);
  BEGIN P.Som := 0 END;
```

```
FUNCTION Pilevide( P : Typepile) : BOOLEAN;
  BEGIN Pilevide := (P.Som = 0) END;
```

```
FUNCTION Pilepleine (P : Typepile) : BOOLEAN;
```

```
BEGIN Pilepleine := (P.Som = Max_niveau) END;
```

```
FUNCTION Sommet ( P : Typepile ) : Typepointeur;  
BEGIN  
  IF NOT Pilevide(P)  
  THEN Sommet := P.Tab(.P.Som.)  
  ELSE Sommet := Nulle  
END;
```

```
PROCEDURE Empiler (VAR P : Typepile ; Val : Typepointeur);  
BEGIN  
  IF NOT Pilepleine(P)  
  THEN  
    BEGIN  
      P.Som := P.Som + 1 ;  
      P.Tab(.P.Som.) := Val ;  
    END  
  ELSE  
    BEGIN  
      WRITELN(Fs,' Pile saturée');  
      HALT;  
    END  
END;
```

```
PROCEDURE Depiler (VAR P: Typepile; VAR Val : Typepointeur);  
BEGIN  
  IF NOT Pilevide(P)  
  THEN  
    BEGIN  
      Val := P.Tab(.P.Som.);  
      P.Som := P.Som - 1  
    END  
  ELSE Val := Nulle  
END;
```

{ Affichage d'un noeud }

```
PROCEDURE Afficher ( Noeud : Typenoeud ; Num : INTEGER );  
VAR  
  I : INTEGER;  
BEGIN  
  WRITE(Fs,' . Contenu du noeud', Num:3,':');  
  IF Num = Arbre  
  THEN WRITELN(Fs,' [ Racine ]')  
  ELSE WRITELN(Fs);  
  FOR I:=1 TO Nbr(Noeud)-1 DO  
    WRITE(Fs, Fils(I,Noeud):2, '(' ,Cle(I,Noeud):4,  
      ' ,',Adr(I,Noeud):3,') ' );  
  WRITELN(Fs, Fils(Nbr(Noeud),Noeud): 3 );  
  WRITELN(Fs)
```

END;

{ Insertion d'un article (réduit à sa clé) dans le fichier de données }

PROCEDURE Insérerfichier (Clef : Typecle ; VAR Adr : INTEGER);

BEGIN

Seek(Fichier,L -1);

READ(Fichier, Bloc);

IF Bloc.Nbr < Capacite_bloc

THEN

BEGIN

Bloc.Nbr := Bloc.Nbr + 1;

Bloc.Tabinfo(.Bloc.Nbr.) := Clef;

Seek(Fichier, L-1);

WRITE(Fichier, Bloc);

Adr := (L-1)*Capacite_bloc + Bloc.Nbr;

END

ELSE

BEGIN

Bloc.Nbr := 1;

Bloc.Tabinfo(.1.) := Clef;

Seek (Fichier, L);

WRITE(Fichier, Bloc);

Adr :=L*Capacite_bloc + 1;

L := L + 1;

END;

END;

{ Impression du fichier de données }

PROCEDURE Imprimerfichier;

VAR

I, J : INTEGER;

BEGIN

WRITELN(Fs,'Contenu du fichier');

FOR I:= 0 TO L-1 DO

BEGIN

Seek(Fichier, I);

READ(Fichier, Bloc);

WRITE(Fs,'Bloc ',I+1 : 3,' : ');

FOR J := 1 TO Bloc.Nbr DO

WRITE(Fs, Bloc.Tabinfo(.J.),' ') ;

WRITELN(Fs);

END;

END;

{ Impression de l'arbre }

PROCEDURE Imprimerarbre;

```

VAR
  I, J : INTEGER;
  Noeud : Typenoeud;
BEGIN
  WRITELN(Fs,'Contenu de l''arbre');
  FOR I:= 0 TO Lindex - 1 DO
    BEGIN
      Seek(F_arbre, I);
      READ(F_arbre, Noeud);
      Afficher(Noeud, I+1)
    END;
  END;

```

{ Listage des articles en ordre croissant }

```

PROCEDURE Lister ( Arbre : Typepointeur;Noeud : Typenoeud );
VAR
  Nt,I : INTEGER;
BEGIN
  IF Arbre <> Nulle
  THEN
    BEGIN
      Seek(F_arbre,Arbre-1);
      READ ( F_arbre, Noeud);
      Nt := Nbr(Noeud) ;
      FOR I:=1 TO Nt - 1 DO
        BEGIN
          Lister ( Fils(I,Noeud), Noeud );
          WRITELN( Fs, Cle(I,Noeud) );
        END;
      Lister (Fils(Nt,Noeud), Noeud )
    END;
  END;

```

{ Rechercher l'indice de la plus petite clé <= Clef }

```

PROCEDURE Rech ( Clef : Typecle; VAR I : INTEGER );
VAR
  Trouv : BOOLEAN;
BEGIN
  Trouv := FALSE;
  I := 1;
  WHILE (NOT Trouv) AND (I < Nbr(Noeud) ) DO
    IF Cle(I,Noeud)>= Clef
    THEN Trouv := TRUE
    ELSE I := I + 1 ;
  END;

```

{ Module de recherche }

```

PROCEDURE Recherche (Arbre: Typepointeur ; Clef : Typecle;
    VAR Trouv : BOOLEAN ; VAR Q : Typepointeur;
    VAR I : INTEGER);

```

```

VAR

```

```

    P : Typepointeur;

```

```

BEGIN

```

```

    Q := Nulle;

```

```

    P := Arbre;

```

```

    Trouv := FALSE;

```

```

    WHILE ( (P<>Nulle) AND( NOT Trouv )) DO

```

```

        BEGIN

```

```

            Seek(F_arbre, P-1);

```

```

            READ( F_arbre, Noeud );

```

```

            Rech ( Clef, I);

```

```

            Q := P;

```

```

            IF Clef= Cle(I,Noeud)

```

```

            THEN Trouv := TRUE

```

```

            ELSE P := Fils(I,Noeud);

```

```

            IF ( NOT Trouv) AND (P <> Nulle)

```

```

            THEN Empiler(Pile, Q)

```

```

        END;

```

```

    END;

```

```

{ Insertion dans un arbre B }

```

```

{ Insérer la paire ( (Clef, Adresse), Nouveaunoeud) à la position Pos }

```

```

PROCEDURE Inserernoeud( Pos: INTEGER; Clef :Typecle; Adresse: Typeadr;
    Nouveaunoeud : Typepointeur );

```

```

VAR

```

```

    I,Nt : INTEGER;

```

```

BEGIN

```

```

    Nt := Nbr(Noeud) ;

```

```

    Aff_nbr(Noeud, Nt+1);

```

```

    FOR I:=Nt DOWNT0 Pos+1 DO

```

```

        BEGIN

```

```

            Aff_fils(Noeud,I+1, Fils(I,Noeud) );

```

```

            Aff_cle( Noeud,I, Cle(I-1,Noeud) );

```

```

            Aff_adr( Noeud,I, Adr( I-1,Noeud ) );

```

```

        END;

```

```

    Aff_fils( Noeud, Pos+1, Nouveaunoeud );

```

```

    Aff_cle ( Noeud, Pos, Clef);

```

```

    Aff_adr ( Noeud, Pos, Adresse);

```

```

END;

```

```

{ Division d'un noeud }

```

```

PROCEDURE Diviser ( Nd : Typepointeur; Nouvellecle : Typecle;

```

```

    Nouvelleadr : Typeadr; Pos : Typepointeur;

```

```

    Nouveaunoeud : Typepointeur;VAR Nd2 :Typepointeur;

```


VAR Clemilieu: Typecle ;VAR Adrmilieu : Typeadr);

VAR

I, J, K: INTEGER;

Sauv_pere, Sauv_index, Sauv_nbr : Typepointeur;

BEGIN

Inserernoed(Pos, Nouvellecle, Nouvelleadr, Nouveaunoeud);

Sauv_nbr := Nbr(Noeud);

I := Nbr(Noeud) DIV 2;

Aff_nbr(Noeud, I);

WRITELN(Fs,' . Noeud à Gauche Nd=',Nd:3);

Afficher(Noeud, Nd);

Seek(F_arbre, Nd-1);

WRITE(F_arbre, Noeud);

K := 1;

FOR J:= I+1 TO Sauv_nbr - 1 DO

BEGIN

Aff_fils(Noeud,K, Fils(J,Noeud));

Aff_cle(Noeud,K, Cle(J,Noeud));

Aff_adr(Noeud,K, Adr(J,Noeud));

K := K + 1;

END;

Aff_fils(Noeud,K, Fils(Sauv_nbr,Noeud));

Aff_nbr(Noeud,K);

Seek(F_arbre, Lindex);

WRITE(F_arbre, Noeud) ;

Lindex := Lindex + 1;

Nd2 := Lindex;

WRITELN(Fs,' . Noeud à Droite Nd2=',Nd2:3);

Afficher(Noeud, Nd2);

Clemilieu := Cle(I,Noeud);

Adrmilieu := Adr(I,Noeud);

WRITELN(Fs,' . Clemilieu=',Clemilieu);

END;

{ Module d'insertion }

PROCEDURE Inserer (VAR Arbre : Typepointeur; Clef : Typecle);

VAR

Nd, Nd2 : Typepointeur;

Pos : INTEGER;

Trouv : BOOLEAN;

Adresse : INTEGER;

Nouveaunoeud : Typepointeur;

Nouvellecle : Typecle;

Nouvelleadr : Typeadr;

Clemilieu : Typecle;

```

Adrmilieu : Typeadr;
F : Typepointeur;
Sauvpos : INTEGER;
BEGIN
  WRITELN(Fs);
  WRITELN(Fs,'<> Insertion de la clé ', Clef);
  IF Arbre = Nulle
  THEN
    BEGIN
      Insererfichier(Clef, Adresse);
      Aff_nbr(Noeud, 2);
      Aff_cle(Noeud,1, Clef);
      Aff_adr(Noeud,1, Adresse);
      Aff_fils(Noeud,1, Nulle); Aff_fils(Noeud,2, Nulle);
      Seek(F_arbre,Lindex);
      WRITE(F_arbre, Noeud) ;
      Lindex := Lindex + 1;
      Arbre := Lindex;
      Imprimerarbre;
      Imprimerfichier;
    END
  ELSE
    BEGIN
      Recherche ( Arbre, Clef, Trouv, Nd , Pos );
      WRITE(Fs,'Résultats du module de recherche : ');
      WRITELN(Fs,'Trouv=',Trouv:5, ' Nd=',Nd:5, ' Pos=',Pos:3);
      IF NOT Trouv
      THEN
        BEGIN
          Insererfichier(Clef, Adresse);
          Nouveaunoeud := Nulle;
          Nouvellecle := Clef;
          Nouvelleadr := Adresse;
          F := Sommet(Pile);
          WHILE ( (F <> Nulle) AND (Nbr(Noeud) = Ordre ) ) DO
            BEGIN
              Sauvpos := Indexdanspere(F,Clef);
              WRITELN(Fs,'Division de la page ',Nd:3 );
              Diviser ( Nd, Nouvellecle, Nouvelleadr, Pos, Nouveaunoeud, Nd2, Clemilieu,
                Adrmilieu);
              Nouveaunoeud := Nd2;
              Pos := Sauvpos;
              Nd := F;
              Seek(F_arbre, F-1);
              READ(F_arbre, Noeud);
              Depiler(Pile, F);
              F := Sommet(Pile);
              Nouvellecle := Clemilieu;
              Nouvelleadr := Adrmilieu;
            END;
          END;
        END
      END;
    END
  END;

```

```

IF Nbr(Noeud) < Ordre
THEN
BEGIN
    Inserernoed ( Pos, Nouvellecle,Nouvelleadr,
                  Nouveaunoed);
    Seek(F_arbre, Nd-1);
    WRITE( F_arbre, Noeud);
END
ELSE
BEGIN
    WRITELN(Fs,'Division de la page racine');
    Diviser (Nd, Nouvellecle, Nouvelleadr, Pos,
             Nouveaunoed, Nd2,
             Clemilieu, Adrmilieu);
    Aff_nbr( Noeud, 2);
    Aff_fils(Noeud,1, Nd);
    Aff_fils(Noeud,2, Nd2);
    Aff_cle(Noeud,1, Clemilieu);
    Aff_adr(Noeud,1, Adrmilieu );
    Seek(F_arbre, Lindex);
    WRITE(F_arbre, Noeud) ;
    Lindex := Lindex + 1;
    Arbre := Lindex;
END;
Imprimerarbre;
Imprimerfichier;
END
ELSE WRITELN('Clé existe déjà');
END;
END;

```

{ Génération aléatoire de clés }

```

PROCEDURE Generer( VAR Cle : Typecle);
VAR
    I : INTEGER;
BEGIN
    Cle := "";
    FOR I:= 1 TO 4 DO
        Cle := Cle + CHR( 64 + (Random(26)+ 1));
    END;

```

{ Programme principal : Insertion, listage de l'arbre B et impression du fichier de données }

```

BEGIN
    Arbre := Nulle;
    ASSIGN(F_arbre, 'R1_arbreb.Pas'); { Fichier d'index : arbre B }
    ASSIGN(Fichier, 'R2_arbreb.Pas'); { Fichier de données }
    ASSIGN(Fs,'R_arbreb.Pas');        { Pour imprimer les résultats }

```

```

REWRITE(F_arbre);
REWRITE(Fichier);

Lindex := 0;
{ Création du fichier Fichier avec le bloc 1 }
Bloc.Nbr := 0 ;
Seek(Fichier, 0);
WRITE(Fichier, Bloc);
L := 1;

REWRITE(Fs);
Nbrarticle :=30;
Ordre := 5 ;
FOR I := 1 TO Nbrarticle DO
  BEGIN
    Generer ( Clef);
    Creerpile(Pile);
    Insérer ( Arbre, Clef );
  END;
WRITELN(Fs);
WRITELN(Fs, 'Listage du fichier en ordre croissant');
WRITELN(Fs);
Lister(Arbre, Noeud);
CLOSE(Fs);
END.

```

Résultats (Contenu du fichier R_arb-B.pas) :

<> Insertion de la clé AAWF

Contenu de l'arbre

. Contenu du noeud 1: [Racine]
-1(AAWF , 1) -1

Contenu du fichier

Bloc 1 : AAWF

<> Insertion de la clé HRIE

Contenu de l'arbre

. Contenu du noeud 1: [Racine]
-1(AAWF , 1) -1(HRIE , 2) -1

Contenu du fichier

Bloc 1 : AAWF HRIE

<> Insertion de la clé JLCM

Contenu de l'arbre

. Contenu du noeud 1: [Racine]
-1(AAWF , 1) -1(HRIE , 2) -1(JLCM , 3) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM

<> Insertion de la clé BVBH

Contenu de l'arbre

. Contenu du noeud 1: [Racine]

-1(AAWF , 1) -1(BVBH , 4) -1(HRIE , 2) -1(JLCM , 3) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH

<> Insertion de la clé XJUI

Division de la page racine

. Noeud à Gauche Nd= 1

. Contenu du noeud 1: [Racine]

-1(AAWF , 1) -1(BVBH , 4) -1

. Noeud à Droite Nd2= 2

. Contenu du noeud 2:

-1(JLCM , 3) -1(XJUI , 5) -1

. Clemlieu=HRIE

Contenu de l'arbre

. Contenu du noeud 1:

-1(AAWF , 1) -1(BVBH , 4) -1

. Contenu du noeud 2:

-1(JLCM , 3) -1(XJUI , 5) -1

. Contenu du noeud 3: [Racine]

1(HRIE , 2) 2

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI

<> Insertion de la clé SVSH

Contenu de l'arbre

. Contenu du noeud 1:

-1(AAWF , 1) -1(BVBH , 4) -1

. Contenu du noeud 2:

-1(JLCM , 3) -1(SVSH , 6) -1(XJUI , 5) -1

. Contenu du noeud 3: [Racine]

1(HRIE , 2) 2

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH

<> Insertion de la clé EIMG

Contenu de l'arbre

. Contenu du noeud 1:

-1(AAWF , 1) -1(BVBH , 4) -1(EIMG , 7) -1

. Contenu du noeud 2:

-1(JLCM , 3) -1(SVSH , 6) -1(XJUI , 5) -1

. Contenu du noeud 3: [Racine]

1(HRIE , 2) 2

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG

<> Insertion de la clé VHMD

Contenu de l'arbre

. Contenu du noeud 1:

-1(AAWF , 1) -1(BVBH , 4) -1(EIMG , 7) -1

. Contenu du noeud 2:

-1(JLCM , 3) -1(SVSH , 6) -1(VHMD , 8) -1(XJUI , 5) -1

. Contenu du noeud 3: [Racine]

1(HRIE , 2) 2

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD

<> Insertion de la clé WHUZ

Division de la page 2

. Noeud à Gauche Nd= 2

. Contenu du noeud 2:

-1(JLCM , 3) -1(SVSH , 6) -1

. Noeud à Droite Nd2= 4

. Contenu du noeud 4:

-1(WHUZ , 9) -1(XJUI , 5) -1

. Clemlieu=VHMD

Contenu de l'arbre

. Contenu du noeud 1:

-1(AAWF , 1) -1(BVBH , 4) -1(EIMG , 7) -1

. Contenu du noeud 2:

-1(JLCM , 3) -1(SVSH , 6) -1

. Contenu du noeud 3: [Racine]

1(HRIE , 2) 2(VHMD , 8) 4

. Contenu du noeud 4:

-1(WHUZ , 9) -1(XJUI , 5) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD WHUZ

<> Insertion de la clé MXVA

Contenu de l'arbre

. Contenu du noeud 1:

-1(AAWF , 1) -1(BVBH , 4) -1(EIMG , 7) -1

. Contenu du noeud 2:

-1(JLCM , 3) -1(MXVA , 10) -1(SVSH , 6) -1

. Contenu du noeud 3: [Racine]

1(HRIE , 2) 2(VHMD , 8) 4

. Contenu du noeud 4:

-1(WHUZ , 9) -1(XJUI , 5) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD WHUZ MXVA

<> Insertion de la clé DDNA

Contenu de l'arbre

. Contenu du noeud 1:

-1(AAWF , 1) -1(BVBH , 4) -1(DDNA , 11) -1(EIMG , 7) -1

. Contenu du noeud 2:

-1(JLCM , 3) -1(MXVA , 10) -1(SVSH , 6) -1

. Contenu du noeud 3: [Racine]

1(HRIE , 2) 2(VHMD , 8) 4

. Contenu du noeud 4:

-1(WHUZ , 9) -1(XJUI , 5) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD WHUZ MXVA

Bloc 2 : DDNA

<> Insertion de la clé PAUQ

Contenu de l'arbre

. Contenu du noeud 1:

-1(AAWF , 1) -1(BVBH , 4) -1(DDNA , 11) -1(EIMG , 7) -1

. Contenu du noeud 2:

-1(JLCM , 3) -1(MXVA , 10) -1(PAUQ , 12) -1(SVSH , 6) -1

. Contenu du noeud 3: [Racine]

1(HRIE , 2) 2(VHMD , 8) 4

. Contenu du noeud 4:

-1(WHUZ , 9) -1(XJUI , 5) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD WHUZ MXVA

Bloc 2 : DDNA PAUQ

<> Insertion de la clé USOF

Division de la page 2

. Noeud à Gauche Nd= 2

. Contenu du noeud 2:

-1(JLCM , 3) -1(MXVA , 10) -1

. Noeud à Droite Nd2= 5

. Contenu du noeud 5:

-1(SVSH , 6) -1(USOF , 13) -1

. Clemlieu=PAUQ

Contenu de l'arbre

. Contenu du noeud 1:

-1(AAWF , 1) -1(BVBH , 4) -1(DDNA , 11) -1(EIMG , 7) -1

. Contenu du noeud 2:

-1(JLCM , 3) -1(MXVA , 10) -1

. Contenu du noeud 3: [Racine]

1(HRIE , 2) 2(PAUQ , 12) 5(VHMD , 8) 4

. Contenu du noeud 4:

-1(WHUZ , 9) -1(XJUI , 5) -1

. Contenu du noeud 5:

-1(SVSH , 6) -1(USOF , 13) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD WHUZ MXVA

Bloc 2 : DDNA PAUQ USOF

<> Insertion de la clé RPYQ

Contenu de l'arbre

. Contenu du noeud 1:

-1(AAWF , 1) -1(BVBH , 4) -1(DDNA , 11) -1(EIMG , 7) -1

. Contenu du noeud 2:

-1(JLCM , 3) -1(MXVA , 10) -1

. Contenu du noeud 3: [Racine]

1(HRIE , 2) 2(PAUQ , 12) 5(VHMD , 8) 4

. Contenu du noeud 4:

-1(WHUZ , 9) -1(XJUI , 5) -1

. Contenu du noeud 5:
-1(RPYQ , 14) -1(SVSH , 6) -1(USOF , 13) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD WHUZ MXVA

Bloc 2 : DDNA PAUQ USOF RPYQ

<> Insertion de la clé ZGRH

Contenu de l'arbre

. Contenu du noeud 1:
-1(AAWF , 1) -1(BVBH , 4) -1(DDNA , 11) -1(EIMG , 7) -1

. Contenu du noeud 2:
-1(JLCM , 3) -1(MXVA , 10) -1

. Contenu du noeud 3: [Racine]
1(HRIE , 2) 2(PAUQ , 12) 5(VHMD , 8) 4

. Contenu du noeud 4:
-1(WHUZ , 9) -1(XJUI , 5) -1(ZGRH , 15) -1

. Contenu du noeud 5:
-1(RPYQ , 14) -1(SVSH , 6) -1(USOF , 13) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD WHUZ MXVA

Bloc 2 : DDNA PAUQ USOF RPYQ ZGRH

<> Insertion de la clé CUMW

Division de la page 1

. Noeud à Gauche Nd= 1
. Contenu du noeud 1:
-1(AAWF , 1) -1(BVBH , 4) -1

. Noeud à Droite Nd2= 6
. Contenu du noeud 6:
-1(DDNA , 11) -1(EIMG , 7) -1

. Clemlieu=CUMW

Contenu de l'arbre

. Contenu du noeud 1:
-1(AAWF , 1) -1(BVBH , 4) -1

. Contenu du noeud 2:
-1(JLCM , 3) -1(MXVA , 10) -1

. Contenu du noeud 3: [Racine]
1(CUMW , 16) 6(HRIE , 2) 2(PAUQ , 12) 5(VHMD , 8) 4

. Contenu du noeud 4:
-1(WHUZ , 9) -1(XJUI , 5) -1(ZGRH , 15) -1

. Contenu du noeud 5:
-1(RPYQ , 14) -1(SVSH , 6) -1(USOF , 13) -1

. Contenu du noeud 6:
-1(DDNA , 11) -1(EIMG , 7) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD WHUZ MXVA

Bloc 2 : DDNA PAUQ USOF RPYQ ZGRH CUMW

<> Insertion de la clé NOYR

Contenu de l'arbre

. Contenu du noeud 1:
-1(AAWF , 1) -1(BVBH , 4) -1

. Contenu du noeud 2:
-1(JLCM , 3) -1(MXVA , 10) -1(NOYR , 17) -1

. Contenu du noeud 3: [Racine]
1(CUMW , 16) 6(HRIE , 2) 2(PAUQ , 12) 5(VHMD , 8) 4

. Contenu du noeud 4:
-1(WHUZ , 9) -1(XJUI , 5) -1(ZGRH , 15) -1

. Contenu du noeud 5:
-1(RPYQ , 14) -1(SVSH , 6) -1(USOF , 13) -1

. Contenu du noeud 6:
-1(DDNA , 11) -1(EIMG , 7) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD WHUZ MXVA

Bloc 2 : DDNA PAUQ USOF RPYQ ZGRH CUMW NOYR

<> Insertion de la clé IASZ

Contenu de l'arbre

. Contenu du noeud 1:
-1(AAWF , 1) -1(BVBH , 4) -1

. Contenu du noeud 2:
-1(IASZ , 18) -1(JLCM , 3) -1(MXVA , 10) -1(NOYR , 17) -1

. Contenu du noeud 3: [Racine]
1(CUMW , 16) 6(HRIE , 2) 2(PAUQ , 12) 5(VHMD , 8) 4

. Contenu du noeud 4:
-1(WHUZ , 9) -1(XJUI , 5) -1(ZGRH , 15) -1

. Contenu du noeud 5:
-1(RPYQ , 14) -1(SVSH , 6) -1(USOF , 13) -1

. Contenu du noeud 6:
-1(DDNA , 11) -1(EIMG , 7) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD WHUZ MXVA

Bloc 2 : DDNA PAUQ USOF RPYQ ZGRH CUMW NOYR IASZ

<> Insertion de la clé UTFD

Contenu de l'arbre

. Contenu du noeud 1:
-1(AAWF , 1) -1(BVBH , 4) -1

. Contenu du noeud 2:
-1(IASZ , 18) -1(JLCM , 3) -1(MXVA , 10) -1(NOYR , 17) -1

. Contenu du noeud 3: [Racine]
1(CUMW , 16) 6(HRIE , 2) 2(PAUQ , 12) 5(VHMD , 8) 4

. Contenu du noeud 4:
-1(WHUZ , 9) -1(XJUI , 5) -1(ZGRH , 15) -1

. Contenu du noeud 5:
-1(RPYQ , 14) -1(SVSH , 6) -1(USOF , 13) -1(UTFD , 19) -1

. Contenu du noeud 6:
-1(DDNA , 11) -1(EIMG , 7) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD WHUZ MXVA

Bloc 2 : DDNA PAUQ USOF RPYQ ZGRH CUMW NOYR IASZ UTFD

<> Insertion de la clé XUPW

Contenu de l'arbre

. Contenu du noeud 1:
-1(AAWF , 1) -1(BVBH , 4) -1

. Contenu du noeud 2:
-1(IASZ , 18) -1(JLCM , 3) -1(MXVA , 10) -1(NOYR , 17) -1

. Contenu du noeud 3: [Racine]
1(CUMW , 16) 6(HRIE , 2) 2(PAUQ , 12) 5(VHMD , 8) 4

. Contenu du noeud 4:
-1(WHUZ , 9) -1(XJUI , 5) -1(XUPW , 20) -1(ZGRH , 15) -1

. Contenu du noeud 5:

-1(RPYQ , 14) -1(SVSH , 6) -1(USOF , 13) -1(UTFD , 19) -1

. Contenu du noeud 6:

-1(DDNA , 11) -1(EIMG , 7) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD WHUZ MXVA

Bloc 2 : DDNA PAUQ USOF RPYQ ZGRH CUMW NOYR IASZ UTFD XUPW

<> Insertion de la clé RHCQ

Division de la page 5

. Noeud à Gauche Nd= 5

. Contenu du noeud 5:

-1(RHCQ , 21) -1(RPYQ , 14) -1

. Noeud à Droite Nd2= 7

. Contenu du noeud 7:

-1(USOF , 13) -1(UTFD , 19) -1

. Clemilieu=SVSH

Division de la page racine

. Noeud à Gauche Nd= 3

. Contenu du noeud 3: [Racine]

1(CUMW , 16) 6(HRIE , 2) 2

. Noeud à Droite Nd2= 8

. Contenu du noeud 8:

5(SVSH , 6) 7(VHMD , 8) 4

. Clemilieu=PAUQ

Contenu de l'arbre

. Contenu du noeud 1:

-1(AAWF , 1) -1(BVBH , 4) -1

. Contenu du noeud 2:

-1(IASZ , 18) -1(JLCM , 3) -1(MXVA , 10) -1(NOYR , 17) -1

. Contenu du noeud 3:

1(CUMW , 16) 6(HRIE , 2) 2

. Contenu du noeud 4:

-1(WHUZ , 9) -1(XJUI , 5) -1(XUPW , 20) -1(ZGRH , 15) -1

. Contenu du noeud 5:

-1(RHCQ , 21) -1(RPYQ , 14) -1

. Contenu du noeud 6:

-1(DDNA , 11) -1(EIMG , 7) -1

. Contenu du noeud 7:

-1(USOF , 13) -1(UTFD , 19) -1

. Contenu du noeud 8:

5(SVSH , 6) 7(VHMD , 8) 4

. Contenu du noeud 9: [Racine]

3(PAUQ , 12) 8

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD WHUZ MXVA

Bloc 2 : DDNA PAUQ USOF RPYQ ZGRH CUMW NOYR IASZ UTFD XUPW

Bloc 3 : RHCQ

<> Insertion de la clé HWEH

Division de la page 2

. Noeud à Gauche Nd= 2

. Contenu du noeud 2:

-1(HWEH , 22) -1(IASZ , 18) -1

. Noeud à Droite Nd2= 10

. Contenu du noeud 10:

-1(MXVA , 10) -1(NOYR , 17) -1

. Clemlieu=JLCM

Contenu de l'arbre

. Contenu du noeud 1:

-1(AAWF , 1) -1(BVBH , 4) -1

. Contenu du noeud 2:

-1(HWEH , 22) -1(IASZ , 18) -1

. Contenu du noeud 3:

1(CUMW , 16) 6(HRIE , 2) 2(JLCM , 3) 10

. Contenu du noeud 4:

-1(WHUZ , 9) -1(XJUI , 5) -1(XUPW , 20) -1(ZGRH , 15) -1

. Contenu du noeud 5:

-1(RHCQ , 21) -1(RPYQ , 14) -1

. Contenu du noeud 6:

-1(DDNA , 11) -1(EIMG , 7) -1

. Contenu du noeud 7:

-1(USOF , 13) -1(UTFD , 19) -1

. Contenu du noeud 8:

5(SVSH , 6) 7(VHMD , 8) 4

. Contenu du noeud 9: [Racine]

3(PAUQ , 12) 8

. Contenu du noeud 10:

-1(MXVA , 10) -1(NOYR , 17) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD WHUZ MXVA

Bloc 2 : DDNA PAUQ USOF RPYQ ZGRH CUMW NOYR IASZ UTFD XUPW

Bloc 3 : RHCQ HWEH

<> Insertion de la clé OYEE

Contenu de l'arbre

. Contenu du noeud 1:

-1(AAWF , 1) -1(BVBH , 4) -1

. Contenu du noeud 2:

-1(HWEH , 22) -1(IASZ , 18) -1

. Contenu du noeud 3:

1(CUMW , 16) 6(HRIE , 2) 2(JLCM , 3) 10

. Contenu du noeud 4:

-1(WHUZ , 9) -1(XJUI , 5) -1(XUPW , 20) -1(ZGRH , 15) -1

. Contenu du noeud 5:

-1(RHCQ , 21) -1(RPYQ , 14) -1

. Contenu du noeud 6:

-1(DDNA , 11) -1(EIMG , 7) -1

. Contenu du noeud 7:

-1(USOF , 13) -1(UTFD , 19) -1

. Contenu du noeud 8:

5(SVSH , 6) 7(VHMD , 8) 4

. Contenu du noeud 9: [Racine]

3(PAUQ , 12) 8

. Contenu du noeud 10:

-1(MXVA , 10) -1(NOYR , 17) -1(OYEE , 23) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD WHUZ MXVA

Bloc 2 : DDNA PAUQ USOF RPYQ ZGRH CUMW NOYR IASZ UTFD XUPW

Bloc 3 : RHCQ HWEH OYEE

<> Insertion de la clé GEOM

Contenu de l'arbre

. Contenu du noeud 1:

-1(AAWF , 1) -1(BVBH , 4) -1

. Contenu du noeud 2:

-1(HWEH , 22) -1(IASZ , 18) -1

. Contenu du noeud 3:

1(CUMW , 16) 6(HRIE , 2) 2(JLCM , 3) 10

. Contenu du noeud 4:

-1(WHUZ , 9) -1(XJUI , 5) -1(XUPW , 20) -1(ZGRH , 15) -1

. Contenu du noeud 5:

-1(RHCQ , 21) -1(RPYQ , 14) -1

. Contenu du noeud 6:

-1(DDNA , 11) -1(EIMG , 7) -1(GEOM , 24) -1

. Contenu du noeud 7:

-1(USOF , 13) -1(UTFD , 19) -1

. Contenu du noeud 8:

5(SVSH , 6) 7(VHMD , 8) 4

. Contenu du noeud 9: [Racine]

3(PAUQ , 12) 8

. Contenu du noeud 10:

-1(MXVA , 10) -1(NOYR , 17) -1(OYEE , 23) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD WHUZ MXVA

Bloc 2 : DDNA PAUQ USOF RPYQ ZGRH CUMW NOYR IASZ UTFD XUPW

Bloc 3 : RHCQ HWEH OYEE GEOM

<> Insertion de la clé HEPT

Contenu de l'arbre

. Contenu du noeud 1:

-1(AAWF , 1) -1(BVBH , 4) -1

. Contenu du noeud 2:

-1(HWEH , 22) -1(IASZ , 18) -1

. Contenu du noeud 3:

1(CUMW , 16) 6(HRIE , 2) 2(JLCM , 3) 10

. Contenu du noeud 4:

-1(WHUZ , 9) -1(XJUI , 5) -1(XUPW , 20) -1(ZGRH , 15) -1

. Contenu du noeud 5:

-1(RHCQ , 21) -1(RPYQ , 14) -1

. Contenu du noeud 6:
-1(DDNA , 11) -1(EIMG , 7) -1(GEOM , 24) -1(HEPT , 25) -1

. Contenu du noeud 7:
-1(USOF , 13) -1(UTFD , 19) -1

. Contenu du noeud 8:
5(SVSH , 6) 7(VHMD , 8) 4

. Contenu du noeud 9: [Racine]
3(PAUQ , 12) 8

. Contenu du noeud 10:
-1(MXVA , 10) -1(NOYR , 17) -1(OYEE , 23) -1

Contenu du fichier

Bloc 1 : AAWF HRIE JLCM BVBH XJUI SVSH EIMG VHMD WHUZ MXVA
Bloc 2 : DDNA PAUQ USOF RPYQ ZGRH CUMW NOYR IASZ UTFD XUPW
Bloc 3 : RHCQ HWEH OYEE GEOM HEPT

Listage du fichier en ordre croissant

AAWF
BVBH
CUMW
DDNA
EIMG
GEOM
HEPT
HRIE
HWEH
IASZ
JLCM
MXVA
NOYR
OYEE
PAUQ
RHCQ
RPYQ
SVSH
USOF
UTFD
VHMD
WHUZ
XJUI
XUPW
ZGRH

EXERCICE 10:

solution C

Essai linéaire interne

Programmer les algorithmes de recherche, insertion et suppression dans la méthode de hachage par essai linéaire. On montrera les différentes étapes pour insérer et supprimer des éléments.

Nous rappelons, pour cette méthode, que s'il se produit une collision sur la case k d'un tableau $T[0..M-1]$, on insère la donnée, si elle n'existe pas, dans la première case libre fournie par la séquence cyclique

$h(k)-1, \dots, 0, M-1, M-2, \dots, h(k)+1$

h désigne la fonction de hachage.

```
*****
* 1. LE PROGRAMME      *
* 2. LES DONNEES       *
* 3. LES RESULTATS     *
*****
```

PROGRAMME 4

```
#include <Stdio.H>
```

```
#define M 10
```

```
#define True 1
```

```
#define False 0
```

```
typedef int Typecle;
```

```
typedef int Bool;
```

```
struct Typelement
```

```
{
    Typecle Cle ;
    Bool Vide ;
} ;
```

```
int I, N ;
```

```
struct Typelement T[M] ;
```

```
FILE *Fs;
```

```

/*-----*/
/* Impression de la table */
/*-----*/

```

```

void Imprimer()
{
    int I ;
    fprintf(Fs, "\nContenu de la table \n");
    fprintf(Fs, "----- \n");
    for (I=0; I<= M-1 ; I++ )
    {
        fprintf(Fs, "%d : ", I);
        if (T[I].Vide == 1 )
            fprintf(Fs, " \n");
        else fprintf(Fs, "%d \n", T[I].Cle);
    }
    fprintf(Fs, "\n");
}

```

```

/*-----*/
/* Fonction de hachage */
/*-----*/

```

```

int Hacher( Typecle K )
{
    return( K % M);
}

```

```

/*-----*/
/* recherche d'un élément */
/*-----*/

```

```

void Recherche ( Typecle K, int *I, Bool *Trouv )
{
    /* Recherche la clé K dans la table. Si K est trouvée, I > 0 .. */
    *I = Hacher(K);
    fprintf(Fs, " La clé %d est hachée en %d \n", K, *I);
    while ( (T[*I].Cle != K) && ( !T[*I].Vide) )
    {
        *I = *I-1 ;
        if ( *I < 0 ) *I = *I + M ;
    } ;
    if ( K == T[*I].Cle ) *Trouv = True; else *Trouv = False;
}

```

```

/*-----*/
/* Insère un élément */
/*-----*/

```

```

void Insérer ( Typecle K )
{
    int I; Bool Trouv ;
    fprintf(Fs,"    **** Insertion de la clé  %d  ****\n", K);
    fprintf(Fs,"\n");
    Recherche(K, &I ,&Trouv);
    fprintf(Fs," Recherche de la clé %d ==> Trouv = %d I = %d \n", K,Trouv, I);
    if (Trouv)
        fprintf(Fs," La clé %d existe \n", K);
    else
        if ( N == M -1 )
            fprintf(Fs,"Débordement\n");
        else
            {
                fprintf(Fs, " %d étant la position libre où sera rangée la clé\n", I);
                N = N + 1;
                T[I].Cle = K;
                T[I].Vide = False;
            }
}

/*-----*/
/* Supprime un élément          */
/*-----*/

```

```

void Supprimer ( Typecle K )
{
    int I, J, R , Etape ;
    Bool Sort, Trouv, Continu;
    fprintf(Fs,"    **** Suppression de la clé  %d  ****\n", K);
    fprintf(Fs,"\n");
    Recherche(K, &I, &Trouv);
    fprintf(Fs, " Recherche de la clé %d ==> Trouv = %d I = %d \n", K, Trouv, I);
    if (!Trouv)
        fprintf(Fs," La clé %d n' existe pas \n", K);
    else
        {
            N--;
            Etape = 0;
            Continu = True;
            while ( Continu)
            {
                Etape = Etape + 1;
                fprintf(Fs, " Etape : %d \n", Etape);
                fprintf(Fs, " ----- \n");
                T[I].Vide = True;
                J = I;
                fprintf(Fs, " Marquer T[ %d ] Vide et sauver %d dans J\n", I,I);
                /* Marquer T(I) Vide et sauver I dans J */
            }
        }
}

```

```

I = I -1; if (I<0) I = I + M ;
Sort = False;
while ( (!T[I].Vide) && (! Sort) )
{
    R = Hacher ( T[I].Cle ) ;
    if (I<J)
        if ( (R>=J) || (R>I) ) Sort = True;
        else {I = I -1; if (I<0) I = I + M ;}
    else
        if ( (R>=J) && (R<I) ) Sort = True ;
        else {I = I -1; if (I<0) I = I + M ;}
    }
    if (T[I].Vide) Continu = False;
    else T[J] = T[I];
}
}
}

```

```

/* Programme principal */
main()
{
    Fs = fopen("S_essail.C","w");
    N = 0;
    for ( I=0 ; I<= M-1 ; I++ )
    {
        T[I].Vide = True;
        T[I].Cle = -1;
    }
    Inserer(13); Imprimer();
    Inserer(12); Imprimer();
    Inserer(23); Imprimer();
    Inserer(35); Imprimer();
    Inserer(4); Imprimer();
    Inserer(18); Imprimer();
    Inserer(14); Imprimer();
    Inserer(5); Imprimer();
    Inserer(1); Imprimer();
    Inserer(2); Imprimer();

    Supprimer(18); Imprimer();
    Supprimer(23); Imprimer();
    Supprimer(4); Imprimer();
    Supprimer(13); Imprimer();

    fclose(Fs);

}

```

Résultats :

**** Insertion de la clé 13 ****

La clé 13 est hachée en 3

Recherche de la clé 13 =====> Trouv = 0 I = 3

3 étant la position libre où sera rangée la clé

Contenu de la table

0 :

1 :

2 :

3 : 13

4 :

5 :

6 :

7 :

8 :

9 :

**** Insertion de la clé 12 ****

La clé 12 est hachée en 2

Recherche de la clé 12 =====> Trouv = 0 I = 2

2 étant la position libre où sera rangée la clé

Contenu de la table

0 :

1 :

2 : 12

3 : 13

4 :

5 :

6 :

7 :

8 :

9 :

**** Insertion de la clé 23 ****

La clé 23 est hachée en 3

Recherche de la clé 23 =====> Trouv = 0 I = 1

1 étant la position libre où sera rangée la clé

Contenu de la table

0 :

1 : 23

2 : 12

3 : 13

4 :
5 :
6 :
7 :
8 :
9 :

**** Insertion de la clé 35 ****

La clé 35 est hachée en 5

Recherche de la clé 35 =====> Trouv = 0 I = 5

5 étant la position libre où sera rangée la clé

Contenu de la table

0 :
1 : 23
2 : 12
3 : 13
4 :
5 : 35
6 :
7 :
8 :
9 :

**** Insertion de la clé 4 ****

La clé 4 est hachée en 4

Recherche de la clé 4 =====> Trouv = 0 I = 4

4 étant la position libre où sera rangée la clé

Contenu de la table

0 :
1 : 23
2 : 12
3 : 13
4 : 4
5 : 35
6 :
7 :
8 :
9 :

**** Insertion de la clé 18 ****

La clé 18 est hachée en 8

Recherche de la clé 18 =====> Trouv = 0 I = 8

8 étant la position libre où sera rangée la clé

Contenu de la table

0 :
1 : 23
2 : 12
3 : 13
4 : 4
5 : 35
6 :
7 :
8 : 18
9 :

**** Insertion de la clé 14 ****

La clé 14 est hachée en 4

Recherche de la clé 14 =====> Trouv = 0 I = 0
0 étant la position libre où sera rangée la clé

Contenu de la table

0 : 14
1 : 23
2 : 12
3 : 13
4 : 4
5 : 35
6 :
7 :
8 : 18
9 :

**** Insertion de la clé 5 ****

La clé 5 est hachée en 5

Recherche de la clé 5 =====> Trouv = 0 I = 9
9 étant la position libre où sera rangée la clé

Contenu de la table

0 : 14
1 : 23
2 : 12
3 : 13
4 : 4
5 : 35
6 :
7 :
8 : 18

9 : 5

**** Insertion de la clé 1 ****

La clé 1 est hachée en 1

Recherche de la clé 1 =====> Trouv = 0 I = 7

7 étant la position libre où sera rangée la clé

Contenu de la table

0 : 14

1 : 23

2 : 12

3 : 13

4 : 4

5 : 35

6 :

7 : 1

8 : 18

9 : 5

**** Insertion de la clé 2 ****

La clé 2 est hachée en 2

Recherche de la clé 2 =====> Trouv = 0 I = 6

Débordement

Contenu de la table

0 : 14

1 : 23

2 : 12

3 : 13

4 : 4

5 : 35

6 :

7 : 1

8 : 18

9 : 5

**** Suppression de la clé 18 ****

La clé 18 est hachée en 8

Recherche de la clé 18 =====> Trouv = 1 I = 8

Etape : 1

Marquer T[8] Vide et sauver 8 dans J

Contenu de la table

0 : 14
1 : 23
2 : 12
3 : 13
4 : 4
5 : 35
6 :
7 : 1
8 :
9 : 5

**** Suppression de la clé 23 ****

La clé 23 est hachée en 3

Recherche de la clé 23 ==> Trouv = 1 I = 1

Etape : 1

Marquer T[1] Vide et sauver 1 dans J

Etape : 2

Marquer T[0] Vide et sauver 0 dans J

Etape : 3

Marquer T[9] Vide et sauver 9 dans J

Contenu de la table

0 : 5
1 : 14
2 : 12
3 : 13
4 : 4
5 : 35
6 :
7 : 1
8 :
9 :

**** Suppression de la clé 4 ****

La clé 4 est hachée en 4

Recherche de la clé 4 ==> Trouv = 1 I = 4

Etape : 1

Marquer T[4] Vide et sauver 4 dans J

Etape : 2

Marquer T[1] Vide et sauver 1 dans J

Etape : 3

Marquer T[0] Vide et sauver 0 dans J

Contenu de la table

0 :
1 : 5
2 : 12
3 : 13
4 : 14
5 : 35
6 :
7 : 1
8 :
9 :

**** Suppression de la clé 13 ****

La clé 13 est hachée en 3

Recherche de la clé 13 =====> Trouv = 1 I = 3

Etape : 1

Marquer T[3] Vide et sauver 3 dans J

Etape : 2

Marquer T[1] Vide et sauver 1 dans J

Contenu de la table

0 :
1 :
2 : 12
3 : 5
4 : 14
5 : 35
6 :
7 : 1
8 :
9 :

EXERCICE 11:

 solution C

Essai linéaire externe

Programmer les algorithmes de recherche, insertion et suppression dans la méthode de hachage par essai linéaire. On suppose que la table est sur le disque. Chaque élément est donc un bloc pouvant renfermer b données.

Nous rappelons, pour cette méthode, qu'une collision se produit quand on tente d'insérer plus de b données par bloc.

Nous rappelons aussi que s'il se produit une collision sur le bloc k du disque, on insère la donnée, si elle n'existe pas, dans le premier bloc non rempli fourni par la séquence cyclique :

$h(k)-1, \dots, 0, M-1, M-2, \dots, h(k)+1$

h désigne la fonction de hachage.

On montrera les différentes étapes pour insérer et supprimer des données.

```
*****
* 1. LE PROGRAMME      *
* 2. LES DONNEES       *
* 3. LES RESULTATS     *
*****
```

PROGRAMME 5

```
#include <Stdio.H>
```

```
#define B 2
```

```
#define True 1
```

```
#define False 0
```

```
typedef int Typecle;
```

```
typedef int Bool;
```

```
struct Typebloc
```

```
{
    int Nb ;
    Typecle Cle[B];
} ;
```

```
struct Typebloc Sauvbuffer, Buffer;
```

```
int M;
```

```
int I, N ;
```

```
FILE *Fs;
```

```
FILE *Fp;
```

```
/*-----*/
```

```
/* Imprime la table */
```

```
/*-----*/
```

```

void Imprimer()
{
    int I, J ;

    fprintf(Fs, "\nContenu du fichier :\n");
    fseek(Fp, 0l, 0);
    for (I=0; I< M ; I++ )
    {
        fprintf(Fs," Bloc ");
        fprintf(Fs," %d : ", I);

        fread(&Buffer, sizeof(Buffer), 1, Fp);
        for (J=0; J<Buffer.Nb; J++)
            fprintf(Fs,"%d ", Buffer.Cle[J] );
        fprintf(Fs,"\n");
    }
    fprintf(Fs, "\n");
}

/*-----*/
/* Initialisation      */
/*-----*/

void Init()
{
    int I;
    fprintf(Fs,"Initialisation du fichier . . .\n\n");
    rewind(Fp);
    for ( I=0 ; I< M ; I++ )
    {
        Buffer.Nb = 0;
        fwrite(&Buffer, sizeof(Buffer), 1, Fp);
    }
}

/*-----*/
/* fonction de hachage      */
/*-----*/

int Hacher( Typecle K )
{
    return( K % M);
}

/*-----*/
/* Recherche une clé      */
/*-----*/

void Recherche ( Typecle K, int *H, int *I, Bool *Trouv )
{

```

```
Bool Continu;  
int Hh, Ii;
```

```
Hh = Hacher(K);  
fprintf(Fs, " La clé %d est hachée en %d \n", K, Hh);
```

```
Continu = True;  
while (Continu)  
{  
    /* Ramener le bloc N° *H */  
    fseek( Fp, (long) ( Hh ) * sizeof( struct Typebloc ), 0);  
    fread(&Buffer, sizeof(Buffer), 1, Fp);  
  
    /* Rechercher dans le buffer */  
    *Trouv = False; Ii = 0;  
    while ( ( Ii < Buffer.Nb ) && ( ! *Trouv ) )  
        if (Buffer.Cle[Ii] == K) *Trouv = True ;else Ii++;  
    if (*Trouv) Continu = False ;  
    else  
        if ( Ii != B ) Continu = False;  
        else { Hh--; if ( Hh < 0 ) Hh += M ; }  
}  
*H = Hh;  
*I = Ii;  
}
```

```
/*-----*/  
/* Insère une clé      */  
/*-----*/
```

```
void Inserer ( Typecle K )  
{  
    int I,H; Bool Trouv ;  
    fprintf(Fs, "      **** Insertion de la clé %d ****\n\n", K);  
    Recherche(K, &H, &I ,&Trouv);  
    fprintf(Fs,  
    " Recherche de la clé %d ==> Trouv = %d H = %d I=%d\n",  
        K,Trouv, H, I);  
    if (Trouv)  
        fprintf(Fs, " La clé %d existe \n", K);  
    else  
        if ( N == M*(B+1) -1 )  
            fprintf(Fs, "Débordement\n");  
        else  
            {  
                N = N + 1;  
                Buffer.Nb++;  
                Buffer.Cle[ Buffer.Nb - 1 ] = K;  
                /* Ecriture sur le disque */  
                fseek( Fp, (long) ( H ) * sizeof( struct Typebloc ), 0);
```

```

        fwrite(&Buffer, sizeof( Buffer), 1, Fp);

    }
}

/*-----*/
/* Supprime une clé */
/*-----*/

void Supprimer ( Typecle K )
{
    int H, J, R , Etape ;
    Bool Place, Cond, Trouv, Continu;
    fprintf(Fs,"      **** Suppression de la clé  %d  ****\n\n", K);
    Recherche(K, &H, &I, &Trouv);
    fprintf(Fs,
        " Recherche de la clé %d ==> Trouv = %d H = %d I=%d \n",
            K, Trouv, H, I);
    if (!Trouv)
        fprintf(Fs," La Cle %d n' existe pas \n", K);
    else
    {
        N --;
        Continu = True;
        while ( Continu)
        {
            /* Supprimer le I-ème élément du buffer */
            if ( I != Buffer.Nb-1 ) /* Déplacer l'article */
                Buffer.Cle[I] = Buffer.Cle [ Buffer.Nb-1 ];
            Buffer.Nb --;
            /* Ecriture sur le disque */
            fseek( Fp, (long) ( H * sizeof( struct Typebloc)), 0);
            fwrite(&Buffer, sizeof( Buffer), 1, Fp);

            Place = (Buffer.Nb != B - 1 ) ;
            Cond = False;
            J = H;
            Sauvbuffer = Buffer;
            while ( !Place && !Cond )
            {
                /* L'entrée H était pleine */
                H = H -1; if (H<0) H += M ;

                /* Ramener le bloc N° H */
                fseek( Fp, (long)(H * sizeof(struct Typebloc)), 0);
                fread(&Buffer, sizeof(Buffer), 1, Fp);

                K = 0;
                while ( ( K < Buffer.Nb) && (! Cond) )
                {

```

```

        R = Hacher ( Buffer.Cle[K] ) ;
        if (H<J)
            if ( (R>=J) || (R<H) ) Cond = True;
            else ;
        else
            if ( (R>=J) && (R<H) ) Cond = True ;
            if ( !Cond ) K++;
        }
        if (! Cond && Buffer.Nb < B) Place = True ;
    }
    if (Place) Continu = False;
    else
    {
        Sauvbuffer.Cle[B-1] = Buffer.Cle[K];
        Sauvbuffer.Nb++;

        fseek( Fp,(long)( J * sizeof(struct Typebloc)), 0);
        fwrite(&Sauvbuffer, sizeof( Sauvbuffer), 1, Fp);
        I = K;
    }
}
}
}

```

/* Programme principal */

```

main()
{
    Fs = fopen("R_essex.c","w");
    Fp = fopen("Ff.C","w+b");
    M = 10;
    N = 0;

    Init();      Insérer(13); Imprimer(); Insérer(12);
    Imprimer(); Insérer(23); Imprimer(); Insérer(35); Imprimer();
    Insérer(4); Imprimer(); Insérer(18); Imprimer(); Insérer(14);
    Imprimer(); Insérer(5); Imprimer(); Insérer(1); Imprimer();
    Insérer(2); Imprimer(); Insérer(3); Imprimer(); Insérer(15);
    Imprimer(); Supprimer(18);Imprimer(); Supprimer(23);Imprimer();
    Supprimer(5);Imprimer(); Supprimer(13);Imprimer(); supprimer(3);
    Imprimer();
    fclose(Fs);

}

```

Résultats : (Contenu du fichier R_essex.c)

Initialisation du fichier . . .

**** Insertion de la clé 13 ****

La clé 13 est hachée en 3

Recherche de la clé 13 ==> Trouv = 0 H = 3 I = 0

Contenu du fichier :

Bloc 0 :
Bloc 1 :
Bloc 2 :
Bloc 3 : 13
Bloc 4 :
Bloc 5 :
Bloc 6 :
Bloc 7 :
Bloc 8 :
Bloc 9 :

**** Insertion de la clé 12 ****

La clé 12 est hachée en 2

Recherche de la clé 12 ==> Trouv = 0 H = 2 I = 0

Contenu du fichier :

Bloc 0 :
Bloc 1 :
Bloc 2 : 12
Bloc 3 : 13
Bloc 4 :
Bloc 5 :
Bloc 6 :
Bloc 7 :
Bloc 8 :
Bloc 9 :

**** Insertion de la clé 23 ****

La clé 23 est hachée en 3

Recherche de la clé 23 ==> Trouv = 0 H = 3 I = 1

Contenu du fichier :

Bloc 0 :
Bloc 1 :
Bloc 2 : 12
Bloc 3 : 13 23
Bloc 4 :
Bloc 5 :
Bloc 6 :
Bloc 7 :
Bloc 8 :
Bloc 9 :

**** Insertion de la clé 35 ****

La clé 35 est hachée en 5

Recherche de la clé 35 =====> Trouv = 0 H = 5 I = 0

Contenu du fichier :

Bloc 0 :

Bloc 1 :

Bloc 2 : 12

Bloc 3 : 13 23

Bloc 4 :

Bloc 5 : 35

Bloc 6 :

Bloc 7 :

Bloc 8 :

Bloc 9 :

**** Insertion de la clé 4 ****

La clé 4 est hachée en 4

Recherche de la clé 4 =====> Trouv = 0 H = 4 I = 0

Contenu du fichier :

Bloc 0 :

Bloc 1 :

Bloc 2 : 12

Bloc 3 : 13 23

Bloc 4 : 4

Bloc 5 : 35

Bloc 6 :

Bloc 7 :

Bloc 8 :

Bloc 9 :

**** Insertion de la clé 18 ****

La clé 18 est hachée en 8

Recherche de la clé 18 =====> Trouv = 0 H = 8 I = 0

Contenu du fichier :

Bloc 0 :

Bloc 1 :

Bloc 2 : 12

Bloc 3 : 13 23

Bloc 4 : 4

Bloc 5 : 35

Bloc 6 :

Bloc 7 :

Bloc 8 : 18

Bloc 9 :

**** Insertion de la clé 14 ****

La clé 14 est hachée en 4

Recherche de la clé 14 =====> Trouv = 0 H = 4 I =1

Contenu du fichier :

Bloc 0 :

Bloc 1 :

Bloc 2 : 12

Bloc 3 : 13 23

Bloc 4 : 4 14

Bloc 5 : 35

Bloc 6 :

Bloc 7 :

Bloc 8 : 18

Bloc 9 :

**** Insertion de la clé 5 ****

La clé 5 est hachée en 5

Recherche de la clé 5 =====> Trouv = 0 H = 5 I =1

Contenu du fichier :

Bloc 0 :

Bloc 1 :

Bloc 2 : 12

Bloc 3 : 13 23

Bloc 4 : 4 14

Bloc 5 : 35 5

Bloc 6 :

Bloc 7 :

Bloc 8 : 18

Bloc 9 :

**** Insertion de la clé 1 ****

La clé 1 est hachée en 1

Recherche de la clé 1 =====> Trouv = 0 H = 1 I =0

Contenu du fichier :

Bloc 0 :

Bloc 1 : 1

Bloc 2 : 12

Bloc 3 : 13 23

Bloc 4 : 4 14

Bloc 5 : 35 5

Bloc 6 :

Bloc 7 :

Bloc 8 : 18

Bloc 9 :

**** Insertion de la clé 2 ****

La clé 2 est hachée en 2

Recherche de la clé 2 ==> Trouv = 0 H = 2 I = 1

Contenu du fichier :

Bloc 0 :

Bloc 1 : 1

Bloc 2 : 12 2

Bloc 3 : 13 23

Bloc 4 : 4 14

Bloc 5 : 35 5

Bloc 6 :

Bloc 7 :

Bloc 8 : 18

Bloc 9 :

**** Insertion de la clé 3 ****

La clé 3 est hachée en 3

Recherche de la clé 3 ==> Trouv = 0 H = 1 I = 1

Contenu du fichier :

Bloc 0 :

Bloc 1 : 1 3

Bloc 2 : 12 2

Bloc 3 : 13 23

Bloc 4 : 4 14

Bloc 5 : 35 5

Bloc 6 :

Bloc 7 :

Bloc 8 : 18

Bloc 9 :

**** Insertion de la clé 15 ****

La clé 15 est hachée en 5

Recherche de la clé 15 ==> Trouv = 0 H = 0 I = 0

Contenu du fichier :

Bloc 0 : 15

Bloc 1 : 1 3

Bloc 2 : 12 2

Bloc 3 : 13 23

Bloc 4 : 4 14

Bloc 5 : 35 5

Bloc 6 :

Bloc 7 :

Bloc 8 : 18

Bloc 9 :

**** Suppression de la clé 18 ****

La clé 18 est hachée en 8

Recherche de la clé 18 =====> Trouv = 1 H = 8 I =0

Contenu du fichier :

Bloc 0 : 15

Bloc 1 : 1 3

Bloc 2 : 12 2

Bloc 3 : 13 23

Bloc 4 : 4 14

Bloc 5 : 35 5

Bloc 6 :

Bloc 7 :

Bloc 8 :

Bloc 9 :

**** Suppression de la clé 23 ****

La clé 23 est hachée en 3

Recherche de la clé 23 =====> Trouv = 1 H = 3 I =1

Contenu du fichier :

Bloc 0 :

Bloc 1 : 1 15

Bloc 2 : 12 2

Bloc 3 : 13 3

Bloc 4 : 4 14

Bloc 5 : 35 5

Bloc 6 :

Bloc 7 :

Bloc 8 :

Bloc 9 :

**** Suppression de la clé 5 ****

La clé 5 est hachée en 5

Recherche de la clé 5 =====> Trouv = 1 H = 5 I =1

Contenu du fichier :

Bloc 0 :

Bloc 1 : 1

Bloc 2 : 12 2

Bloc 3 : 13 3

Bloc 4 : 4 14

Bloc 5 : 35 15

Bloc 6 :

Bloc 7 :

Bloc 8 :
Bloc 9 :

**** Suppression de la clé 13 ****

La clé 13 est hachée en 3
Recherche de la clé 13 =====> Trouv = 1 H = 3 I = 0

Contenu du fichier :

Bloc 0 :
Bloc 1 : 1
Bloc 2 : 12 2
Bloc 3 : 3
Bloc 4 : 4 14
Bloc 5 : 35 15
Bloc 6 :
Bloc 7 :
Bloc 8 :
Bloc 9 :

**** Suppression de la clé 3 ****

La clé 3 est hachée en 3
Recherche de la clé 3 =====> Trouv = 1 H = 3 I = 0

Contenu du fichier :

Bloc 0 :
Bloc 1 : 1
Bloc 2 : 12 2
Bloc 3 :
Bloc 4 : 4 14
Bloc 5 : 35 15
Bloc 6 :
Bloc 7 :
Bloc 8 :
Bloc 9 :

EXERCICE12



Hachage virtuel linéaire

Soit une table $T(0..M-1)$ de M cases, Chaque case peut renfermer B données. Initialement, seulement la sous-table $T(0..N-1)$ de N cases primaires ($N \ll M$) est réservée aux données, c'est à dire toute donnée D est rangée par la fonction de hachage $HL(D) = D \bmod (2L.N)$ avec $L = 0$. La méthode augmente l'espace des adresses progressivement par l'éclatement des cases

dans un ordre prédéfini : d'abord la case 0, puis la case 1,...N-1. Un pointeur P garde la trace de la prochaine case à éclater. l'éclatement d'une case implique le partage (par la fonction hL+1) de ses données avec une nouvelle case ajoutée à la fin de la sous-table T(0..N-1). Quand les N cases ont été éclatées et donc le nombre de cases devient égal à 2N, P est remis à 0 et le processus d'éclatement est recommencé de nouveau avec la même fonction H mais avec L incrémenté d'une unité c'est à dire égal à 1. A ce moment, P variera de 0 à 2N-1, doublant ainsi la taille à 4N. Etc.

L'éclatement d'une case est provoqué quand le facteur de chargement & devient supérieur à un seuil S donné. Par conséquent, si on tente d'insérer une donnée D dans une case C alors que celle-ci est pleine, D sera rangée dans une case allouée dynamiquement qui sera chaînée à la case C. Pour chaque case primaire, on peut avoir une liste linéaire chaînée de cases secondaires.

Quand une donnée est supprimée, la contraction sera faite quand & devient inférieur ou égal à un seuil S' donné. C'est l'opération inverse de l'éclatement.

Programmez les algorithmes suivants :

- initialisation.
- transformation donnée-adresse.
- recherche d'une donnée D.
- insertion d'une donnée D.
- suppression d'une donnée D

Pour les modules d'insertion et de suppression, le programme imprimera la trace permettant de montrer pas à pas les insertions et les suppressions des données.

Considérations :

. &=nombre de données insérées / (nombre de cases utilisées X B)

. Lors de l'éclatement, il est préférable de construire une liste linéaire chaînée des données avant de les partager par la nouvelle fonction HL+1.

. Une case n'a de suivant que si elle est pleine. Par conséquent, la suppression ne doit pas engendrer des "trous".

```
*****
* 1. LE PROGRAMME      *
* 2. LES DONNEES       *
* 3. LES RESULTATS     *
*****
```

PROGRAMME 6

```
#include <Alloc.H>
#include <Stdlib.H>
```

```

#include <Stdio.H>

#define Mm 50;

#define True 1
#define False 0

typedef int Bool;
typedef int Typedonnee;
typedef struct Typecase *Pointeur;
typedef struct M *Pt;

struct Typecase
{
    int Nb ;
    Typedonnee Tab[11]; /* Position 0 non utilisé */
    Pointeur Lien ;
} ;

struct M
{
    Typedonnee Val;
    struct M *Adr;
} ;

int B, N, L, P ;
float Smin, Smax, A ;
int I;
Typedonnee D;
struct Typecase Tabhvl[50];
int Nb_donnees, Nb_cases ;
int Pos;
FILE *Fs;

/*-----*/
/* Imprime la table */
/*-----*/

void Imprimer()
{
    int I,J, K;
    Pointeur Lien;
    fprintf(Fs,"nContenu de la table : \n");
    fprintf(Fs,"-----\n\n");
    fprintf(Fs,"Dernière case utilisée Pos - 1 = %d\n", Pos - 1);
    fprintf(Fs,"Prochaine case à éclater P = %d \n",P);
    fprintf(Fs,"Niveau de la fonction L = %d \n\n",L);
    for ( I = 0; I <= Pos-1 ; I++)
    {
        fprintf(Fs,"Case primaire : %d ==> ", I);
    }
}

```

```

    for ( J= 1; J<= Tabhvl[I].Nb;J++)
        fprintf(Fs,"  %d", Tabhvl[I].Tab[J] );
    fprintf(Fs,"\n");
    Lien = Tabhvl[I].Lien;
    if ( Lien != NULL) fprintf(Fs,"Case en débordement \n");
    K = 0;
    while (Lien != NULL)
    {
        K++;
        fprintf(Fs,"          Case secondaire %d ==> ", K);
        for (J=1; J<= Lien->Nb; J++)
            fprintf(Fs, "  %d",Lien->Tab[J]);fprintf(Fs,"\n");
        Lien = Lien->Lien;
    }
}

```

```

/*-----*/
/* Initialisation      */
/*-----*/

```

```

void Init()
{
    int I;
    L = 0;
    P = 0;
    Pos = N;
    for ( I=0 ; I<= 49; I++)
    {
        Tabhvl[I].Lien = NULL;
        Tabhvl[I].Nb = 0;
    }
}

```

```

int Puis2 (int L )
{
    int I, Res;
    Res = 1;
    for (I = 1; I<= L;I++) Res *= 2;
    return Res;
}

```

```

/*-----*/
/* Fonction de hachage    */
/*-----*/

```

```

int H (int L, Typedonnee K)
{
    return K % (Puis2(L)*N) ;
}

```



```

/*-----*/
/* Transformation Donnée --> Adresse */
/*-----*/

```

```

int T ( Typedonnee D )
{
    int Adr;
    Adr = H(L, D);
    if (Adr < P) Adr = H(L+1, D);
    return Adr;
}
/*-----*/
/* Module utilisé par la recherche */
/*-----*/

```

```

void Recherche_case (Typedonnee D, struct Typecase Zone, Bool *Trouv,
                    int *I)
{
    int Ii;
    Ii = 1; *Trouv = False;
    /* Writeln(Fs,'Zone.Nb=', Zone.Nb); */
    while ( ( Ii<= Zone.Nb ) && ( ! *Trouv ) )
        if (Zone.Tab[Ii] == D ) *Trouv = True;
        else Ii++;
    *I = Ii;
}

/*-----*/
/* Recherche une donnée */
/*-----*/

```

```

void Recherche (Typedonnee D, int *K, Bool *Trouv, int *Indice,
                Bool *Zoneprimaire, Pointeur *Prec_lien ,
                Pointeur *Lien)
{
    struct Typecase Zone, Zone1 ;
    int Indice1;
    Bool Trouv1;

    *K = T(D);
    *Prec_lien = NULL;
    *Zoneprimaire = True;
    Recherche_case( D, Tabhvl[*K], &Trouv1, &Indice1 );
    if ( ! Trouv1 )
        { *Lien = Tabhvl[*K].Lien;
          while ( ! Trouv1 && ( *Lien != NULL ) )
              { *Zoneprimaire = False;
                Zone1 = **Lien;

```

```

    Zone.Nb = Zone1.Nb ;
    { int I;
      for (I =1; I<=10; I++) Zone.Tab[I] = Zone1.Tab[I] ;
    } ;
    Zone.Lien = Zone1.Lien;

    Recherche_case(D, Zone, &Trouv1, &Indice1);
    if ( ! Trouv1)
        { *Prec_lien = *Lien;
          *Lien = Zone1.Lien;
        }
    }
}
*Trouv = Trouv1 ;
*Indice = Indice1 ;
}

/*-----*/
/* Modules utilisés par l'insertion      */
/*-----*/

void Ajouter (Typedonnee D ,int K, Bool Zprim, Pointeur Lien)
{
    Pointeur P ;
    if (Zprim)
        if ( Tabhvl[K].Nb < B )
            { Tabhvl[K].Nb++;
              Tabhvl[K].Tab[ Tabhvl[K].Nb ] = D;
            }
        else
            { P =(struct Typecase *)malloc(sizeof(struct Typecase ));
              Nb_cases++;
              Tabhvl[K].Lien = P;
              P->Nb = 1;
              P->Lien = NULL;
              P->Tab[1] = D;
            }
        else
            if ( Lien->Nb < B )
                { Lien->Nb++;
                  Lien->Tab[ Lien->Nb] = D;
                }
            else
                { P =(struct Typecase *) malloc( sizeof(struct Typecase));
                  Nb_cases++;
                  Lien->Lien = P ;
                  P->Nb = 1;
                  P->Lien = NULL;
                  P->Tab[1] = D;
                }
}

```

```
}
```

```
void Ranger ( Typedonnee Val, int Pos, int *N, Pointeur *Lien)
```

```
{
    int Ii;
    Pointeur P;
    Ii = *N;
    if( Ii < B)
    {
        Ii++;
        Tabhvl[Pos].Nb = Ii;
        Tabhvl[Pos].Tab[Ii] = Val ;
    }
    else
    {
        struct Typecase Zone;
        Zone = **Lien;
        if ( (Ii % B) != 0 ) /* N multiple de B */
        {
            Zone.Nb++;
            Ii++ ;
            Zone.Tab[ Zone.Nb ] = Val;
        }
        else
        {
            P=(struct Typecase *)malloc( sizeof(struct Typecase));
            Nb_cases++;
            if (Ii != B) Zone.Lien = P ; else Tabhvl[Pos].Lien = P;
            Ii++;
            *Lien = P;

            P->Nb = 1;
            P->Lien = NULL;
            P->Tab[1] = Val;
        }
    }
    *N = Ii;
}
```

```
void Generer_llc( int K, Pt *Q)
```

```
{
    Pt S, P, Qq;
    Pointeur Sauv, Lien;
    int L;
    Qq = NULL;
    S = NULL;
    for (L=1 ; L<= Tabhvl[K].Nb; L++)
    {
        Qq = (struct M *) malloc( sizeof(struct M )) ;
        Qq->Val = Tabhvl[K].Tab[L];
        Qq->Adr = S;
```

```

        S = Qq;
    }

    Lien = Tabhvl[K].Lien;
    while ( Lien != NULL )
    {
        for (L=1; L <= Lien->Nb; L++)
        { Qq =(struct M *) malloc( sizeof(struct M )) ;
          Qq->Val = Lien->Tab[L];
          Qq->Adr =S;
          S =Qq;
        }
        Sauv = Lien;
        Lien = Lien->Lien;
        free(Sauv);

        Nb_cases--;
    }
    Tabhvl[K].Nb = 0;
    Tabhvl[K].Lien = NULL;
    *Q = Qq;
}

void Imprimerliste( Pt L)
{
    Pt Q;
    fprintf(Fs, "Liste des éléments de la classe P = %d \n", P);
    Q = L;
    while ( Q != NULL)
    { fprintf(Fs," %d ", Q->Val) ;
      Q = Q->Adr;
    };
    fprintf(Fs,"\n");
}

void Partager( )
{
    int I, J, Adr, N, M ;
    Pointeur Lien ;
    Pt R, Q ;
    Pos++;
    Nb_cases++;
    N = 0;
    M = 0;
    Generer_llc(P, &Q);
    Imprimerliste(Q);

    while ( Q != NULL)
    {
        Adr = H(L+1, Q->Val );

```

```

    fprintf(Fs,
    " . %d est rangée dans la case %d \n", Q->Val,Adr);

    if (Adr != P ) Ranger ( Q->Val, Adr, &N, &Lien );
    else Ranger ( Q->Val, P, &M, &Lien);

    R = Q;
    Q = Q->Adr ;

    free(R) ;
}
}

void Eclater ()
{
    fprintf(Fs, "Eclatement de la case P = %d \n", P);
    Partager();
    P++;
    if( P == N * Puis2(L) )
    {
        L++;
        P = 0 ;
    }
}

/*-----*/
/* Insère une donnée */
/*-----*/

void Inserer (Typedonnee D )
{
    int K, Indice;
    Bool Trouv, Zprim;
    Pointeur Prelien, Lien ;

    fprintf(Fs,
    "\n      * * * Insertion de %d * * * \n\n", D);
    Recherche ( D, &K, &Trouv, &Indice, &Zprim, &Prelien, &Lien);
    if (Trouv) fprintf(Fs,"%d existe\n", D);
    else
    {
        Ajouter ( D, K, Zprim, Prelien );
        Nb_donnees++;
        A = (float) Nb_donnees / ( Nb_cases * B);
        fprintf(Fs,"Facteur de chargement Alpha = %f\n",A);
        if (A >= Smax) Eclater();
    }
}

/*-----*/
/* Modules utilisés par la suppression */

```

```
/*-----*/
```

```
void Rechercher_derniere_case ( int K, Bool Zprim, Pointeur Lien,  
                             Pointeur *Q, Pointeur *P )
```

```
{  
    Pointeur Qq, Pp;  
    Qq = NULL;  
    if (Zprim )  
    {  
        Pp = Tabhvl[K].Lien;  
        Qq = NULL ;  
    }  
    else  
    {  
        Pp = Lien->Lien;  
        Qq = Lien;  
    }  
    if (Pp != NULL)  
        while ( Pp->Lien != NULL )  
        {  
            Qq = Pp;  
            Pp = Pp->Lien;  
        }  
    *Q = Qq;  
    *P = Pp;  
}
```

```
void Enlever ( /*Typedonnee D ,*/ int K, int Indice, Bool Zprim,  
              Pointeur Prelien, Pointeur Lien)
```

```
{  
    Pointeur Q, P;  
    Rechercher_derniere_case ( K, Zprim, Lien, &Q, &P);  
    if ( Zprim)  
        if (P == NULL)  
        { if ( Indice != Tabhvl[K].Nb )  
            Tabhvl[K].Tab[Indice] = Tabhvl[K].Tab[ Tabhvl[K].Nb ];  
            Tabhvl[K].Nb--;  
        }  
        else  
        { Tabhvl[K].Tab[Indice] = P->Tab[ P->Nb ];  
            P->Nb--;  
            if (P->Nb == 0)  
            { Nb_cases-- ;  
                free (P);  
                if (Q == NULL) Tabhvl[K].Lien = NULL;  
                else Q->Lien = NULL;  
            }  
        }  
    }  
    else  
        if ( P == NULL )
```

```

{
    if (Indice != Lien->Nb )
        Lien->Tab[Indice] = Lien->Tab [Lien->Nb ];
    Lien->Nb--;
    if ( Lien->Nb == 0)
        { Nb_cases--;
          free(Lien);
          if (Preclien == NULL) Tabhvl[K].Lien = NULL;
          else Preclien->Lien = NULL ;
        }
    }
else
    {
        Lien->Tab[Indice] = P->Tab[ P->Nb ];
        P->Nb--;
        if ( P->Nb == 0)
            { Nb_cases--;
              free(P);
              if (Q == Lien) Lien->Lien = NULL;
              else Q->Lien = NULL;
            }
    }
}

```

void Transfert (int Pos)

```

{
    Pointeur Q, R, L ;
    int I;
    /* Rechercher la première case disponible dans la classe P*/
    Q = Tabhvl[P].Lien;
    if (Q != NULL)
        while (Q->Lien != NULL) Q = Q->Lien ;

    /* Déplacement Des Donnees De La Case Primaire */
    for (I=1 ; I<= Tabhvl[Pos].Nb; I++)
        if (Q == NULL)
            if ( Tabhvl[P].Nb < B )
                { Tabhvl[P].Nb++;
                  Tabhvl[P].Tab[ Tabhvl[P].Nb ] = Tabhvl[Pos].Tab[I] ;
                }
            else
                {
                    R=(struct Typecase *)malloc(sizeof(struct Typecase));
                    Nb_cases++;
                    R->Lien = NULL;
                    R->Nb = 1;
                    R->Tab[1] = Tabhvl[Pos].Tab[I] ;
                    Tabhvl[P].Lien = R;
                    Q = R;
                }
}

```

```

else
    if (Q->Nb < B)
    {
        Q->Nb++;
        Q->Tab[ Q->Nb] = Tabhvl[Pos].Tab[I] ;
    }
else
    {
        R=(struct Typecase *)malloc(sizeof(struct Typecase));
        Nb_cases++;
        R->Lien = NULL;
        R->Nb = 1;
        R->Tab[1] = Tabhvl[Pos].Tab[I] ;
        Q->Lien = R;
        Q = R;
    }
Nb_cases--;
/* {Déplacement des Données des cases secondaires */
L = Tabhvl[Pos].Lien;
while (L != NULL )
{
    if (Q == NULL)
        if (Tabhvl[P].Nb < B )
        {
            Tabhvl[P].Nb++;
            Tabhvl[P].Tab[ Tabhvl[P].Nb ] = L->Tab[I] ;
        }
    else
    {
        R=(struct Typecase *)malloc(sizeof(struct Typecase));
        Nb_cases++;
        R->Lien = NULL;
        R->Nb = 1;
        R->Tab[1] = L->Tab[I] ;
        Tabhvl[P].Lien = R;
        Q = R;
    }
    else
        if (Q->Nb < B)
        {
            Q->Nb++;
            Q->Tab[Q->Nb] = L->Tab[I] ;
        }
    else
    {
        R=(struct Typecase *)malloc(sizeof(struct Typecase));
        Nb_cases++;
        R->Lien = NULL;
        R->Nb = 1;
        R->Tab[1] = L->Tab[I] ;
        Q->Lien = R;
        Q = R;
    }
    R = L;
    L = L->Lien;
}

```



```

        free(R);
        Nb_cases--;
    }
}

```

```

void Fusionner ()
{
    fprintf(Fs,"Fusion des cases %d et %d \n", P , Pos-1);
    P--;
    if (P < 0)
    {
        L--;
        P = N * Puis2 (L) - 1;
    }
    Transfert(Pos-1);
    Pos--;
    Tabhvl[Pos].Lien = NULL;
    Tabhvl[Pos].Nb = 0;
}

```

```

/*-----*/
/* Supprime une donnée */
/*-----*/

```

```

void Supprimer (Typedonnee D)
{
    int K, Indice ;
    Bool Trouv, Zprim ;
    Pointeur Prelien, Lien;

    fprintf(Fs,
"\n      * * * Suppression de %d * * * \n\n", D);
    Recherche ( D, &K, &Trouv, &Indice, &Zprim, &Prelien, &Lien);

    /*
Rechercher D dans les cases de la classe K.
Si D existe Trouv = Vrai et Indice est le rang de D dans la case
Zprim = Vrai veut dire que D est trouvée dans une case primaire
Zprim = Faux veut dire que D est trouvée dans une case secondaire
Dans ce dernier cas, Lien est l'adresse de la case secondaire
*/

    if (! Trouv)
        fprintf(Fs," %d n'existe pas\n", D);
    else
    {
        Enlever( /*D,*/ K, Indice, Zprim, Prelien, Lien);
        Nb_donnees--;
        A = (float) Nb_donnees / ( Nb_cases * B );
        fprintf(Fs, "Facteur de chargement Alpha= %f\n",A);
    }
}

```

```

        if ( A <= Smin) Fusionner();
    }
}

/* programme principal */
main()
{
    Fs = fopen("R_hvl.C", "w");
    N=5; B=2; Smax =0.6; Smin=0.5;
    Init();
    Nb_donnees = 0;
    Nb_cases = N;
    Insérer(11) ; Insérer(0); Insérer(5);
    Insérer(6) ; Insérer(16); Insérer(31);
    Insérer(34) ; Insérer(51);Insérer(12);
    Imprimer();
    Insérer(7) ;
    Imprimer();

    Supprimer(6);Imprimer();
    Supprimer(31);Imprimer();
    Supprimer(5); Imprimer();
    Supprimer(16);Imprimer();

    Insérer(15); Imprimer();
    Insérer(5); Imprimer();
    Insérer(37) ; Imprimer();
    fclose(Fs);
}

```

Résultats : (Contenu de la table R_hvl.c)

*** Insertion de 11 ***

Facteur de chargement Alpha = 0.100000

*** Insertion de 0 ***

Facteur de chargement Alpha = 0.200000

*** Insertion de 5 ***

Facteur de chargement Alpha = 0.300000

*** Insertion de 6 ***

Facteur de chargement Alpha = 0.400000

*** Insertion de 16 ***

Facteur de chargement Alpha = 0.416667

* * * Insertion de 31 * * *

Facteur de chargement Alpha = 0.500000

* * * Insertion de 34 * * *

Facteur de chargement Alpha = 0.583333

* * * Insertion de 51 * * *

Facteur de chargement Alpha = 0.571429

* * * Insertion de 12 * * *

Facteur de chargement Alpha = 0.642857

Eclatement de la case P = 0

Liste des éléments de la classe P = 0

5 0

. 5 est rangée dans la case 5

. 0 est rangée dans la case 0

Contenu de la table :

Dernière case utilisée Pos - 1 = 5

Prochaine case à éclater P = 1

Niveau de la fonction L = 0

Case primaire : 0 ==> 0

Case primaire : 1 ==> 11 6

Case en débordement

Case secondaire 1 ==> 16 31

Case secondaire 2 ==> 51

Case primaire : 2 ==> 12

Case primaire : 3 ==>

Case primaire : 4 ==> 34

Case primaire : 5 ==> 5

* * * Insertion de 7 * * *

Facteur de chargement Alpha = 0.625000

Eclatement de la case P = 1

Liste des éléments de la classe P = 1

51 31 16 6 11

. 51 est rangée dans la case 1

. 31 est rangée dans la case 1

. 16 est rangée dans la case 6

. 6 est rangée dans la case 6

. 11 est rangée dans la case 1

Contenu de la table :

Dernière case utilisée Pos - 1 = 6

Prochaine case à éclater P = 2

Niveau de la fonction L = 0

Case primaire : 0 ==> 0

Case primaire : 1 ==> 51 31

Case en débordement

Case secondaire 1 ==> 11

Case primaire : 2 ==> 12 7

Case primaire : 3 ==>

Case primaire : 4 ==> 34

Case primaire : 5 ==> 5

Case primaire : 6 ==> 16 6

* * * Suppression de 6 * * *

Facteur de chargement Alpha= 0.562500

Contenu de la table :

Dernière case utilisée Pos - 1 = 6

Prochaine case à éclater P = 2

Niveau de la fonction L = 0

Case primaire : 0 ==> 0

Case primaire : 1 ==> 51 31

Case en débordement

Case secondaire 1 ==> 11

Case primaire : 2 ==> 12 7

Case primaire : 3 ==>

Case primaire : 4 ==> 34

Case primaire : 5 ==> 5

Case primaire : 6 ==> 16

* * * Suppression de 31 * * *

Facteur de chargement Alpha= 0.571429

Contenu de la table :

Dernière case utilisée Pos - 1 = 6

Prochaine case à éclater P = 2

Niveau de la fonction L = 0

Case primaire : 0 ==> 0
Case primaire : 1 ==> 51 11
Case primaire : 2 ==> 12 7
Case primaire : 3 ==>
Case primaire : 4 ==> 34
Case primaire : 5 ==> 5
Case primaire : 6 ==> 16

* * * Suppression de 5 * * *

Facteur de chargement Alpha= 0.500000
Fusion des cases 2 et 6

Contenu de la table :

Dernière case utilisée Pos - 1 = 5
Prochaine case à éclater P = 1
Niveau de la fonction L = 0

Case primaire : 0 ==> 0
Case primaire : 1 ==> 51 11
Case en débordement
 Case secondaire 1 ==> 16
Case primaire : 2 ==> 12 7
Case primaire : 3 ==>
Case primaire : 4 ==> 34
Case primaire : 5 ==>

* * * Suppression de 16 * * *

Facteur de chargement Alpha= 0.500000
Fusion des cases 1 et 5

Contenu de la table :

Dernière case utilisée Pos - 1 = 4
Prochaine case à éclater P = 0
Niveau de la fonction L = 0

Case primaire : 0 ==> 0
Case primaire : 1 ==> 51 11
Case primaire : 2 ==> 12 7
Case primaire : 3 ==>
Case primaire : 4 ==> 34

* * * Insertion de 15 * * *

Facteur de chargement $\text{Alpha} = 0.700000$

Eclatement de la case $P = 0$

Liste des éléments de la classe $P = 0$

15 0

. 15 est rangée dans la case 5

. 0 est rangée dans la case 0

Contenu de la table :

Dernière case utilisée $\text{Pos} - 1 = 5$

Prochaine case à éclater $P = 1$

Niveau de la fonction $L = 0$

Case primaire : 0 ==> 0

Case primaire : 1 ==> 51 11

Case primaire : 2 ==> 12 7

Case primaire : 3 ==>

Case primaire : 4 ==> 34

Case primaire : 5 ==> 15

* * * Insertion de 5 * * *

Facteur de chargement $\text{Alpha} = 0.666667$

Eclatement de la case $P = 1$

Liste des éléments de la classe $P = 1$

11 51

. 11 est rangée dans la case 1

. 51 est rangée dans la case 1

Contenu de la table :

Dernière case utilisée $\text{Pos} - 1 = 6$

Prochaine case à éclater $P = 2$

Niveau de la fonction $L = 0$

Case primaire : 0 ==> 0

Case primaire : 1 ==> 11 51

Case primaire : 2 ==> 12 7

Case primaire : 3 ==>

Case primaire : 4 ==> 34

Case primaire : 5 ==> 15 5

Case primaire : 6 ==>

* * * Insertion de 37 * * *

Facteur de chargement $\text{Alpha} = 0.562500$

Contenu de la table :

Dernière case utilisée Pos - 1 = 6

Prochaine case à éclater P = 2

Niveau de la fonction L = 0

Case primaire : 0 ==> 0

Case primaire : 1 ==> 11 51

Case primaire : 2 ==> 12 7

Case en débordement

Case secondaire 1 ==> 37

Case primaire : 3 ==>

Case primaire : 4 ==> 34

Case primaire : 5 ==> 15 5

Case primaire : 6 ==>

I- Introduction aux structures de fichiers

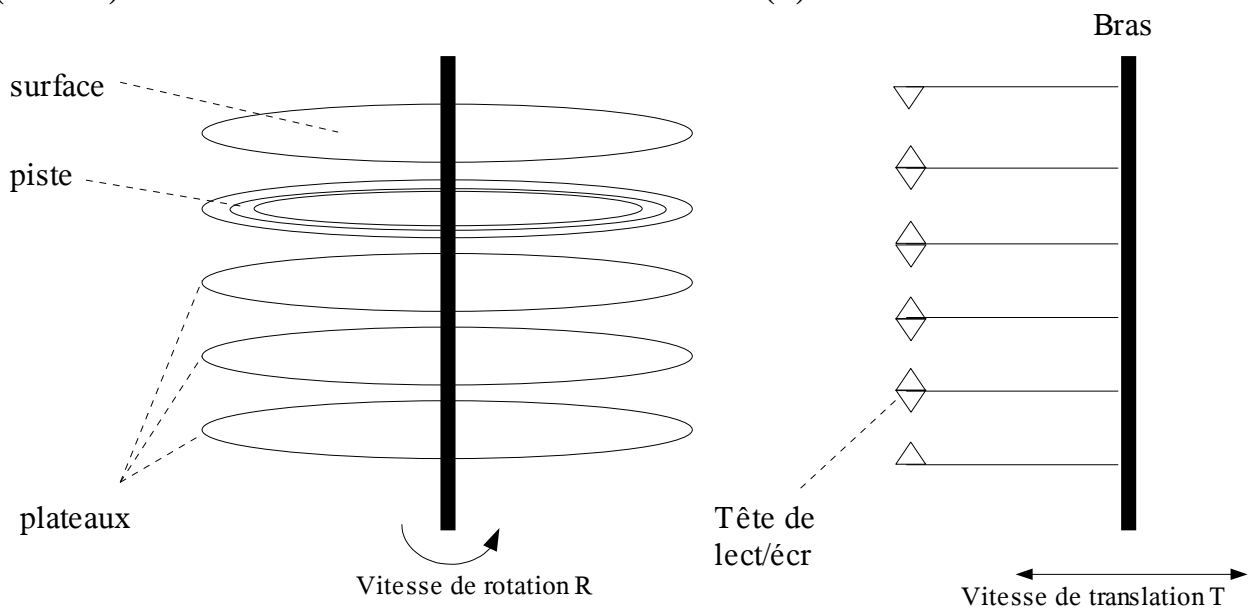
1) Entrées/Sorties (E/S) sur disque

La mémoire secondaire (MS) est généralement de grande taille mais le temps d'accès est très grand relativement à la mémoire centrale (MC).

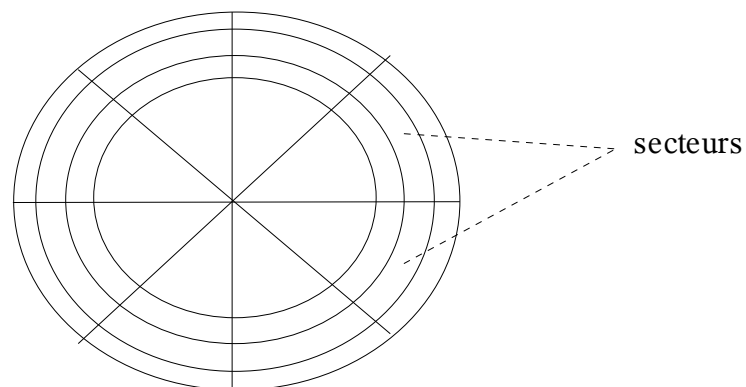
De plus, la MS est généralement non volatile et le coût à l'unité de stockage est moindre que celui de la MC.

Le type de MS le plus utilisé de nos jours reste le disque dur (DD).

Un DD est un ensemble de plateaux superposés, chacun formé de deux faces magnétiques (surfaces) et tournant avec une vitesse de rotation constante (R).



Chaque surface est composée d'un nombre fixe de pistes concentriques, pouvant stocker la même capacité en octets. Chaque piste est découpée en un nombre (généralement fixe) de secteurs, constituant l'unité de transfert entre la MS et la MC (le secteur est donc la plus petite quantité de données pouvant être lue ou écrite sur le disque à l'aide d'une opération E/S).



Le DD dispose d'une tête de lecture/écriture par piste. Toutes les têtes de lecture/écriture sont attachées à un même bras qui se déplace latéralement pour se positionner sur une piste donnée au niveau de chaque surface. L'ensemble des pistes sur lesquels les têtes de lecture/écriture sont positionnés à un moment donné, représentent un cylindre.

Temps d'accès

Pour transférer le contenu d'un secteur donné, le contrôleur du disque déplace le bras pour le positionner sur le bon cylindre, attend que le secteur passe sous la tête de lecture/écriture puis active la tête associée à la surface sur laquelle le secteur se trouve pour récupérer l'information stockée.

Le temps nécessaire à cette opération (avec la technologie actuelle) est au voisinage de quelques ms en moyenne. Si on le compare au temps d'accès à la mémoire centrale (quelques ns), la différence est énorme (le temps d'accès au disque est un million de fois plus lent que celui de la MC).

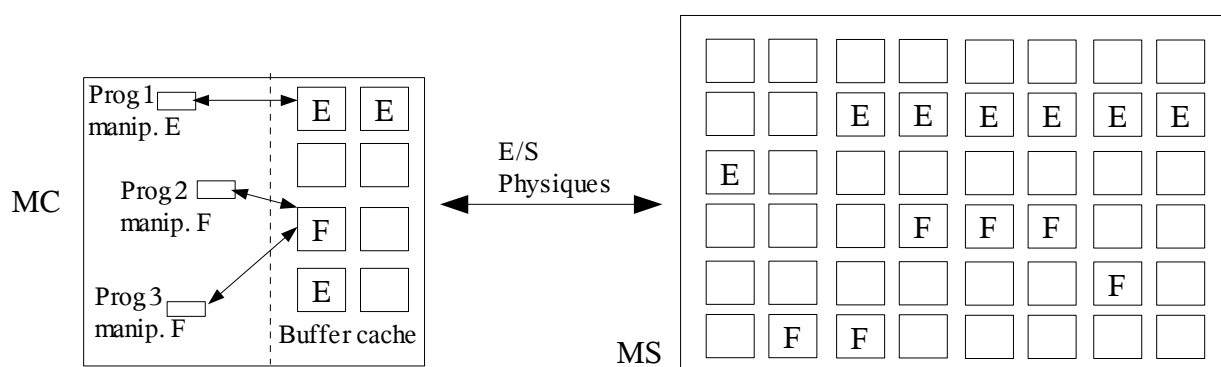
Parmi les méthodes utilisées pour diminuer l'effet de cette grande différence dans le temps d'accès, on retrouve le « **buffering** ». Il s'agit de réserver une zone en mémoire centrale pouvant contenir plusieurs blocs en même temps et utiliser des stratégies de remplacement (comme LRU, FIFO, ...) afin de minimiser le nombre d'accès disque. D'autres techniques, comme la multi-programmation (durant l'opération d'E/S d'un programme, le processeur est alloué à un autre programme) ou la parallélisation des E/S (les données sont réparties sur plusieurs disques – RAID) sont aussi utilisées pour diminuer le goulot d'étranglement sur les E/S.

De plus, comme le temps de déplacement du bras est plus grand que celui nécessaire pour une rotation du disque, l'accès à des secteurs physiquement proches (situés sur la même piste ou le même cylindre) est beaucoup plus rapide que l'accès à des secteurs nécessitant un déplacement du bras. Le « **clustering** » est une technique de stockage rassemblant les données susceptibles d'être traitées ensemble dans des secteurs physiquement proches.

2) Notion de fichier

Au niveau applicatif (logique), un fichier est un ensemble de données (enregistrements) stockées en mémoire secondaire (ex en Pascal : File of Tenreg). On peut aussi définir un fichier comme étant une suite d'octets, on parle alors de flux (ex en C : FILE *). C'est des fichiers « non typés ».

Au niveau physique, un fichier est un ensemble de blocs, organisés d'une certaine manière en MS. Les données (enregistrements) du fichier sont stockés à l'intérieur des blocs.



Les programmes d'applications manipulent les fichiers à travers des appels systèmes (read, write, ...) pour accéder aux données en mémoire cache. Le système synchronise périodiquement (et quand c'est nécessaire) les buffers en cache avec le disque à l'aide d'E/S physiques (lireBloc/écrireBloc)

Caractéristiques

Pour pouvoir manipuler correctement un fichier, on doit donc connaître sa structure et son organisation. Certaines de ces informations (caractéristiques) sont sauvegardées en MS et chargées en MC lors de la manipulation du fichier.

Parmi les caractéristiques courantes, on retrouve :

- le nombre de blocs utilisés
- le nombre d'enregistrements
- la date de création du fichier
- la date du dernier accès au fichier
- l'identité du propriétaire du fichier (cas des systèmes multi-utilisateurs)
- les droits d'accès au fichier (cas des systèmes multi-utilisateurs)
- ... etc

Méthode d'accès

Une structure de fichier (méthode d'accès) consiste à définir :

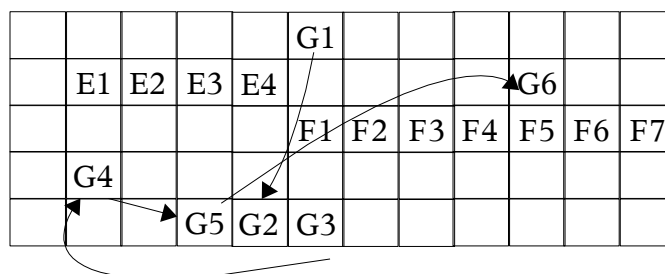
- une manière d'organiser les blocs d'un fichier sur MS
- le placement des enregistrements à l'intérieur des blocs
- l'implémentation des opérations d'accès (recherche, insertion, suppression, ...).

3) Modèle de fichiers

Afin d'étudier les méthodes d'accès aux fichiers dans un cadre général, on modélise la MS par une zone contigüe de blocs numérotés séquentiellement (adresses de blocs).

Les blocs sont des zones contigües d'octets de même taille, renfermant les données (enregistrements) des fichiers.

Dans le schéma suivant, la MS contient 3 fichiers E, F et G



Le fichier E est formé de 4 blocs contigus, le fichier F est formé de 7 blocs contigus et le fichier G est une liste de blocs.

Pour écrire des algorithmes sur les structures de fichiers on utilisera la machine abstraite définie par le modèle suivant:

{ouvrir, fermer, lireDir, écrireDir, lireSeq, écrireSeq, aff_entete, entete, allocbloc }

Dans ce modèle, on manipule des numéros de blocs relatifs au début de chaque fichier (adresses logiques). L'utilisation des adresses physiques n'est pas d'une utilité particulière à ce niveau.

un fichier est donc un ensemble de blocs **numérotés logiquement** (1, 2, 3, ... n)

Déclaration d'un fichier 'f' et de ses zones tampons 'buf1', 'buf2', ... :

var

f : FICHIER de TypeBloc BUFFER buf1, buf2, ... ENTETE (type1, type2, ...typem);

dans cette ligne de déclaration, il y a la définition de :

- la variable f de type FICHIER
- les variables buf1, buf2, ...de type 'TypeBloc' qui servent comme zones tampons pour lire et écrire les données du fichier.
- la structure de l'entête du fichier, formée par m caractéristiques dont les types sont spécifiés

Ouvrir(f, nomfichier, mode)

pour ouvrir ou créer un fichier de nom 'nomfichier' selon la valeur de 'mode'

« a » : ouvrir un ancien fichier en lecture/écriture et charge ses caractéristiques en MC

« n » : créer un nouveau fichier en lecture/écriture et alloue une zone en MC pour ses caractéristiques

Fermer(f)

ferme le fichier f, en sauvegardant ses caractéristiques sur disque

LireDir(F, i, buf)

lire dans 'buf' le contenu du bloc numéro 'i' du fichier 'f' (num logique de blocs)

EcrireDir(F, i, buf)

écrit le contenu de la variable 'buf' dans le bloc numéro 'i' du fichier (num logique)

LireSeq(F, buf)

lire dans 'buf' le contenu du bloc courant du fichier 'f'. (lecture séquentielle)

EcrireSeq(F, buf)

écrit le contenu de la variable 'buf' dans le bloc courant du fichier (écriture séquentielle)

Entete(f, i)

retourne la valeur de la 'i'ème caractéristique du fichier 'f' (en MC)

Aff_entete(f, i, val)

affecte 'val' dans la 'i'ème caractéristique du fichier 'f' (en MC)

AllocBloc(f)

alloue un nouveau bloc au fichier et retourne son numéro

4) Objectifs du cours

Etudier les méthodes d'accès aux fichiers (structures et algorithmes) ainsi que leurs performances (temps d'exécution et occupation mémoire)

Pour la manipulation d'un fichier d'enregistrements donné, on étudie les différentes possibilités de stockage sur MS. Un enregistrement est un ensemble de champs dont un ou plus jouant le rôle de clés de recherche (champs utilisés dans la recherche et l'identification des enregistrements).

Ex:

```
Tenreg = structure { matricule : entier, nom : chaine[20], adr : chaine[80] };  
// matricule est (par exemple) la clé de l'enregistrement
```

Les principaux paramètres à prendre en considération, lors de l'élaboration d'une méthode d'accès sont:

- Le temps d'accès des opérations de base (recherche, insertion, ...)
- Le facteur de chargement (pourcentage de remplissage de l'espace alloué au fichier)
- La taille du fichier en MS
- La taille des structures auxiliaires de la méthode en MC et en MS
- Le type d'accès:
 - fichier statique – peu ou pas d'insertions/suppressions
 - fichier dynamique – en constante évolution

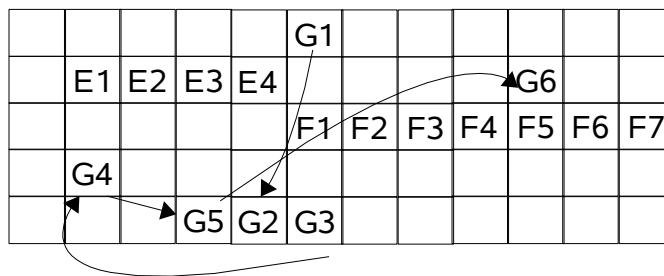
II- Fichiers – Structures simples

1) organisation globale des blocs

Dans un premier temps, on étudiera deux possibilités distinctes d'organiser les blocs au sein d'un fichier:

- soit le fichier est « vu comme un tableau » : tous les blocs qui le forment sont contigus
- soit le fichier est « vu comme une liste » : les blocs ne sont pas forcément contigus, mais sont chaînés entre eux.

Dans la fig ci-dessous, on a deux fichiers vus comme tableau (E et F) et un fichier vu comme une liste (G)



parmi les caractéristiques nécessaires pour manipuler un fichier vu comme tableau, on pourra avoir :

- Le numéro du dernier bloc (ou alors le nombre de blocs utilisés)

pour un fichier vu comme liste, il suffirait par contre de connaître le numéro du premier bloc (la tête de la liste), car dans chaque bloc, il y a le numéro du prochain bloc (comme le champ suivant dans une liste). Dans le dernier bloc, le numéro du prochain bloc pourra être mis à une valeur spécial (ex : -1) pour indiquer la fin de la liste

2) Organisation interne des blocs

les blocs sont censés contenir les enregistrements d'un fichier. Ces derniers peuvent être de longueur fixe ou variable.

Si on est intéressé par des enregistrements de longueur fixe, chaque bloc pourra alors contenir un tableau d'enregistrements de même type.

Ex:

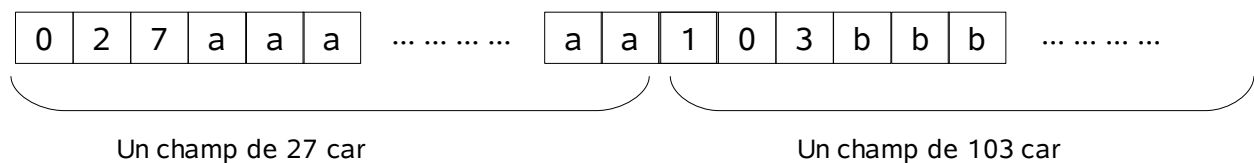
```
Type Tenreg = structure
    matricule : chaine(10);
    nom : chaine(20);
    age : entier;
    ...
fin;
```

```
Type Tbloc = structure
    tab : tableau[1..b] de Tenreg;    // tab pouvant contenir (au max) b enreg.
    NB : entier;                      // nb d'enregistrements insérés dans tab.
fin;
```

Si on opte pour des enregistrements de tailles variables, chaque enreg sera vu comme étant une chaîne de car (de longueur variable).

Les enreg seront de longueur variable, car par exemple, il y a un ou plusieurs champs ayant des tailles variables, ou alors le nombre de champs varie d'un enreg à un autre.

Pour séparer les champs entre eux (à l'intérieur de l'enregistrement), on pourra soit utiliser un caractère spécial ('#') ne pouvant pas être utilisé comme valeur légale, ou alors préfixer le début des champs par leur taille (sur un nombre fixe de positions). Dans la fig ci-dessous, on utilise 3 positions pour indiquer la taille des champs.



Le bloc ne peut pas être défini comme étant un tableau d'enregistrements, car les éléments d'un tableau doivent toujours être de même taille. La solution c'est de considérer le bloc comme étant (ou contenant) une grande chaîne de caractères renfermant les différents enregistrements (stockés caractère par caractère).

Pour séparer les enreg entre eux, on utilise les mêmes techniques que celles utilisées dans la séparation entre les champs d'un même enregistrement (soit un car spécial '\$', soit on préfixe chaque enregistrement par sa taille).

Voici un exemple de déclaration d'un type de bloc pouvant être utilisé dans la définition d'un fichier vu comme liste avec format (taille) variable des enregistrements.

```
Type Tbloc = structure
    tab : tableau[1..b'] de caractères; // tableau de car pour les enreg.
    suiv : entier;                      // num du bloc suivant dans la liste
fin;
```

remarque: même si les enregistrements sont de longueurs variables, la taille des blocs reste toujours fixe.

Pour minimiser l'espace perdu dans les blocs (dans le cas : format variable uniquement), on peut opter pour une organisation avec chevauchement entre deux ou plusieurs blocs: quand on veut insérer un nouvel enreg dans un bloc non encore plein et où l'espace vide restant n'est pas suffisant pour contenir entièrement cet enreg, celui-ci sera découpé en 2 parties de telle sorte à occuper tout l'espace vide du bloc en question par la 1ere partie, alors que le reste (la 2e partie) sera insérée dans un nouveau bloc alloué au fichier. On dit que l'enregistrement se trouve à cheval entre 2 blocs.

3) Taxonomie des structures simples de fichiers

En combinant entre l'organisation globale des fichiers (tableau ou liste) et celle interne aux blocs (format fixe ou variable des enregistrements), on peut définir une classe de

méthodes d'accès (dites « simples ») pour organiser des données sur disque.

Si de plus on prend en compte la possibilité de garder le fichier ordonné ou non, suivant les valeurs d'un champ clé particulier, on doublera le nombre de méthodes dans cette classe de structures simples de fichiers.

utilisant la notation suivante:

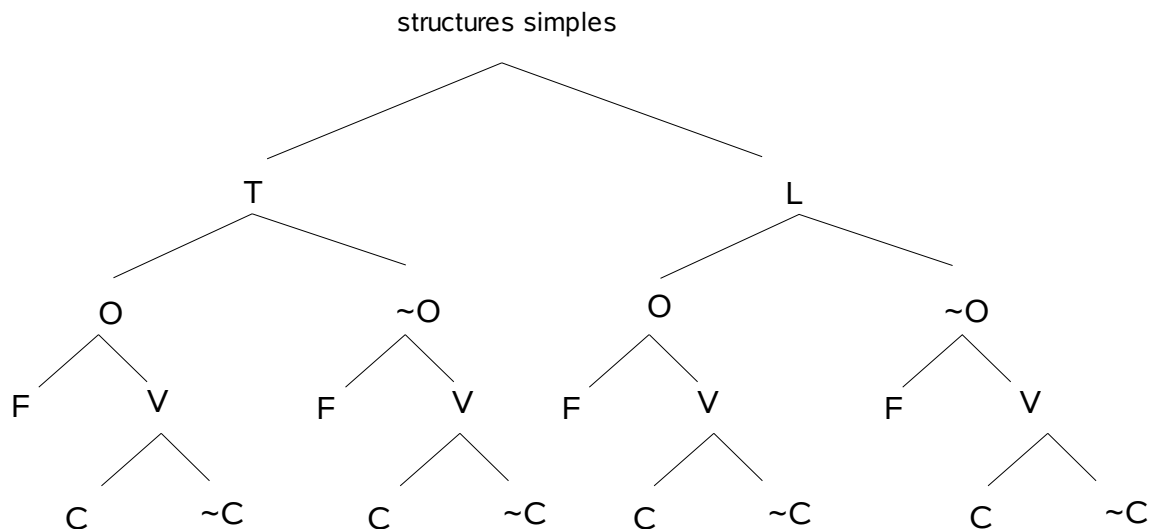
T (pour fichier vu comme tableau), L (pour liste)

O (pour fichier ordonné), ~O (non ordonné)

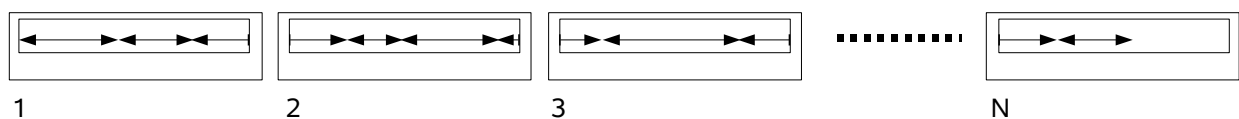
F (pour format fixe des enreg), V (pour format variable)

C (pour chevauchement des enreg entre blocs), ~C (pour pas de chevauchement)

les feuilles de l'arbre suivant, représentent les 12 méthodes d'accès simples:

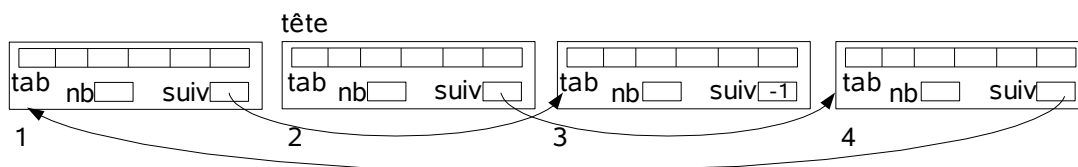


Par exemple la méthode T~OVC représente l'organisation d'un fichier vu comme tableau, non ordonné, avec des enregistrements de taille variables et acceptant les chevauchements entre blocs :



La recherche est séquentielle, l'insertion en fin de fichier et la suppression est logique.

Dans le cas d'un fichier LOF, chaque bloc contient un tableau d'enregistrements, un entier indiquant le nombre d'enregistrements dans le tableau et un entier pour garder la trace du bloc suivant dans la liste :



La recherche est séquentielle, l'insertion provoque des décalages intra-blocs et la

suppression est physique.

4) Exemple complet: fichier de type « TOF »

(fichier vu comme tableau, ordonné, et enregistrements à taille fixe)

- La recherche d'un enregistrement est dichotomique (rapide).
- L'insertion peut provoquer des décalages intra et inter-blocs (coûteuse).
- La suppression peut être réalisée par des décalages inverses (suppression physique coûteuse) ou alors juste par un indicateur booléen (suppression logique beaucoup plus rapide). Optant pour cette dernière alternative.
- On commence généralement par faire un chargement initial du fichier en laissant un peu de vide dans chaque bloc, afin de minimiser les décalages pouvant être provoqués par les futures insertions => C'est l'opération de chargement initial des fichiers ordonnés.
- Avec le temps, le facteur de chargement du fichier (nombre d'insertions / nombre de places disponibles dans le fichier) augmente à cause des insertions futures, de plus les suppressions logiques ne libèrent pas de places. Donc les performances se dégradent avec le temps. Il est alors conseillé de réorganiser le fichier en procédant à un nouveau chargement initial => C'est l'opération de réorganisation.

Déclaration du fichier:

```
const
    b = 30;          // capacité maximale des blocs (en nombre d'enregistrements)

type
    Tenreg = structure
        effacé : boolean;    // booléen pour la suppression logique
        cle : typeqlq;       // le champs utilisé comme clé de recherche
        champ2 : typeqlq;    // les autres champs de l'enregistrement,
        champ3 : typeqlq;    // sans importance ici.
        ...
    Fin;

    Tbloc = structure        // le bloc renferme :
        tab : tableau[1..b] de Tenreg;    // un tableau d'enreg d'une capacité b
        NB : entier;           // nombre d'enreg dans tab ( <= b)
    Fin;

var // globales
    F : Fichier de Tbloc Buffer buf Entete (entier, entier);
/*
    L'entête contient deux caractéristiques:
        - la première sert à garder la trace du nombre de bloc utilisés (ou alors le
          numéro logique du dernier bloc du fichier)
        - la deuxième servira comme un compteur d'insertions pour pouvoir calculer
          rapidement le facteur de chargement, et donc voir s'il y a nécessité de
          réorganiser le fichier.
*/
```


Module de recherche: (dichotomique)

en entrée la clé à chercher et le nom externe du fichier.

en sortie le booléen Trouv, le num de bloc (i) contenant (c) et le déplacement (j)

Rech(c:typeqlq; nomfich:chaine; var Trouv:bool; var i,j:entier)

var

bi, bs, inf, sup : entier;

trouv, stop : booléen;

DEBUT

Ouvrir(F, nomfich, 'A');

bs := **entete(F, 1)**; *// la borne sup (le num du dernier bloc de F)*

bi := 1; *// la borne inf (le num du premier bloc de F)*

Trouv := faux; stop := faux; j := 1;

TQ (bi <= bs et Non Trouv et Non stop)

 i := (bi + bs) div 2; *// le bloc du milieu entre bi et bs*

LireDir(F, i, buf);

 SI (c >= buf.tab[1].cle et c <= buf.tab[buf.NB].cle)

// recherche dichotomique à l'intérieur du bloc (dans la variable buf)...

 inf := 1; sup := buf.NB;

 TQ inf <= sup et Non Trouv

 j := (inf + sup) div 2;

 SI c = buf.tab[j].cle: Trouv := vrai

 SINON

 SI c < buf.tab[j].cle: sup := j-1

 SINON inf := j+1

 FSI

 FSI

 FTQ

 SI (Non Trouv)

 j := inf

 FSI

// fin de la recherche interne. j indique l'endroit où devrait se trouver c

 stop := vrai

SINON *// non (c >= buf.tab[1].cle et c <= buf.tab[buf.NB].cle)*

 SI (c < buf.tab[1].cle)

 bs := i-1

 SINON *// c > buf.tab[buf.NB].cle*

 bi := i+1

 FSI

FSI

FTQ

SI (Non Trouv)

 i := bi

FSI

fermer(F)

FIN

Module d'insertion: (avec éventuellement des décalages intra et inter blocs)

Inserer(e:Tenreg; nomfich:chaîne)

var

trouv : booleen;

i,j,k : entier;

e,x : Tenreg;

DEBUT

// on commence par rechercher la clé e.cle avec le module précédent pour localiser l'emplacement (i,j)

// où doit être insérer e dans le fichier.

Rech(e.cle, nomfich, trouv, i, j);

SI (Non trouv)

// e doit être inséré dans le bloc i à la position j

Ouvrir(F,nomfich, 'A');

// en décalant les enreg j, j+1, j+2, ... vers le bas

continu := vrai;

// si i est plein, le dernier enreg de i doit être inséré dans i+1

TQ (continu et i <= entete(F,1))

// si le bloc i+1 est aussi plein son dernier enreg sera

LireDir(F, i, buf);

// inséré dans le bloc i+2, etc ... donc une boucle TQ.

// avant de faire les décalages, sauvegarder le dernier enreg dans une var x ...

x := buf.tab[buf.NB];

// décalage à l'intérieur de buf ...

k := buf.NB;

TQ k > j

buf.tab[k] := buf.tab[k-1];

k := k-1

FTQ

// insérer e à la pos j dans buf ...

buf.tab[j] := e;

// si buf n'est pas plein, on remet x à la pos NB+1 et on s'arrête ...

SI (buf.NB < b)

// b est la capacité max des blocs (une constante)

buf.NB := buf.NB+1;

buf.tab[buf.NB] := x;

EcrireDir(F, i, buf);

continu := faux;

SINON *// si buf est plein, x doit être inséré dans le bloc i+1 à la pos 1 ...*

EcrireDir(F, i, buf);

i := i+1;

j := 1;

e := x; *// cela se fera (l'insertion) à la prochaine itération du TQ*

FSI *// non (buf.NB < b)*

FTQ

// si on dépasse la fin de fichier, on rajoute un nouveau bloc contenant un seul enregistrement e

SI i > entete(F, 1)

buf.tab[1] := e;

buf.NB := 1;

EcrireDir(F, i, buf); *// il suffit d'écrire un nouveau bloc à cet emplacement*

Aff-entete(F, 1, i); *// on sauvegarde le num du dernier bloc dans l'entete 1*

FSI

Aff-entete(F, 2, entete(F,2)+1);

// on incrémente le compteur d'insertions

Fermer(F);

FSI

FIN

La suppression logique consiste à rechercher l'enregistrement et positionner le champs 'effacé' à vrai :

Suppression(c:typeqlq; nomfich:chaîne)

```
var
    trouv : booleen;
    i,j : entier;
DEBUT
    // on commence par rechercher la clé c pour localiser l'emplacement (i,j) de l'enreg à supprimer
    Rech( c, nomfich, trouv, i, j );
    // ensuite on supprime logiquement l'enregistrement
    SI ( trouv )
        Ouvrir( F,nomfich, 'A');
        LireDir( F, i, buf );      // lecture pas vraiment nécessaire à cause de l'effet de bord de Rech sur buf
        buf.tab[j].effacé := VRAI;
        EcrireDir( F, i, buf );
        Fermer( F )
    FSI
FIN // suppression
```

Le chargement initial d'un fichier ordonné consiste à construire un nouveau fichier contenant dès le départ n enregistrements. Ceci afin de laisser un peu de vide dans chaque bloc, qui pourrait être utilisé plus tard par les nouvelles insertions tout en évitant les décalages inter-blocs (très coûteux en accès disque) :

Chargement_Initial(nomfich : chaîne; n : entier; u : reel)

// u est un réel compris entre 0 et 1 et désigne le taux de chargement voulu au départ

```
var
    e : Tenreg;
    i,j,k : entier;
DEBUT
    Ouvrir( F, nomfich, 'N' );      // un nouveau fichier
    i := 1;                          // num de bloc à remplir
    j := 1;                          // num d'enreg dans le bloc
    ecrire( 'Donner les enregistrements en ordre croissant suivant la clé : ' );
    POUR k:=1 , n
        lire( e );
        SI j <= u*b                  // ex: si u=0.5, on remplira les bloc jusqu'à b/2 enreg
            buf.tab[j] := e
            j := j+1;
        SINON                        // j > u*b : buf doit être écrit sur disque
            buf.NB = j-1;
            EcrireDir( F, i, buf );
            buf.tab[1] := e; // le kème enreg sera placé dans le prochain bloc, à la position 1
            i := i+1;
            j := 2;
        FSI
    FP
    // à la fin de la boucle, il reste des enreg dans buf qui n'ont pas été sauvegardés sur disque
    buf.NB := j-1;
    EcrireDir( F, i, buf );
    // mettre à jour l'entête (le num du dernier bloc et le compteur d'insertions)
    Aff-entete( F, 1, i );
    Aff-entete( F, 2, n );
    Fermer( F )
FIN // chargement-initial
```

La réorganisation du fichier consiste à recopier les enreg vers un nouveau fichier de telle sorte à ce que les nouveaux blocs contiennent un peu de vide (1-u). Cette opération ressemble au chargement initial sauf que les enregistrements sont lus à partir de l'ancien fichier.

Fusion de 2 fichiers ordonnés (TOF)

On parcourt les 2 fichiers en parallèle dans 2 buffers (buf1 et buf2) et on remplit le buffer (buf3) du 3e fichier en ordre croissant

Les déclarations sont celles utilisées dans les fichier TOF standard

Fusion (nom1,nom2, nom3: chaine)

var

F1 : Fichier de Tbloc Buffer buf1 Entete(entier, entier);

F2 : Fichier de Tbloc Buffer buf2 Entete(entier, entier);

F3 : Fichier de Tbloc Buffer buf3 Entete(entier, entier);

i1, i2, i3 : entier;

j1, j2, j3 : entier;

continu : booleen;

e, e1, e2 : Tenreg;

buf : Tbloc;

i, j, indic : entier;

Debut

ouvrir(F1, nom1, 'A');

ouvrir(F2, nom2, 'A');

ouvrir(F3, nom3, 'N');

i1:=1; i2:=1; i3 :=1; *// les num de blocs de F1, F2 et F3*

j1:=1; j2:=1; j3 :=1; *// les num d'enreg dans buf1, buf2 et buf3*

LireDir(F1, 1, buf1)

LireDir(F2, 1, buf2)

continu := vrai

TQ continu

// tant que non fin de fichier dans F1 et F2 faire

```
SI ( j1 <= buf1.NB et j2 <= buf2.NB )
    // choisir le plus petit enreg, dans buf1 et buf2
    e1:=buf1.tab[j1];
    e2 := buf2.tab[j2]
    SI ( e1.cle <= e2.cle )
        e := e1; j1:= j1 + 1;
    SINON
        e := e2; j2:= j2 + 1;
    FSI

    // et le mettre dans buf3
    SI ( j3 <= b )
        buf3.tab[j3] := e; j3 := j3 + 1
    SINON
        buf3.NB := j3 - 1;
        EcrireDir(F3, i3, buf3 );
        i3 := i3 + 1;
        buf3.tab[1] := e;
        J3 := 2;
    FSI

SINON // c-a-d : non ( j1 <= buf1.NB et j2 <= buf2.NB )
    // si tous les enreg d'un des blocs (buf1 ou buf2) ont été traités, passer au prochain
    SI ( j1 > buf1.NB )
        SI ( i1 < entete(F1, 1) )
            i1 := i1 + 1;
            LireDir( F1, i1, buf1 )
            j1 := 1
        SINON // ( i1 < entete(F1, 1) )
            continu := faux
            i := i1; // pour la suite du TQ
            j:= j1;
            N := entete(F1,1)
            buf := buf1
            Indic := 1
        FSI // ( i1 < entete(F1, 1) )
    SINON // c-a-d ( j2 > buf2.NB )
        SI ( i2 < entete(F2, 1) )
            i2 := i2 + 1;
            LireDir( F2, i2, buf2 )
            j2 := 1
        SINON // ( i2 < entete(F2, 1) )
            continu := faux
            i := i2; // pour la suite du TQ
            j:= j2;
            N := entete(F2,1)
            buf := buf2;
            Indic = 2;
        FSI // ( i2 < entete(F2, 1) )

    FSI // ( j1 > buf1.NB )

FSI // ( j1 <= buf1.NB et j2 <= buf2.NB )
```

FTQ

```

// continuer à recopier les enregistrement d'un seul fichier (i,j,buf) dans F3
continu := vrai;
TQ continu // tant que non fin de fichier dans F1 ou F2 faire
    SI ( j <= buf.NB )
        SI ( j3 <= b )
            buf3.tab[j3] := buf.tab[j]; j3 := j3 + 1
        SINON
            buf3.NB := j3 - 1;
            EcrireDir(F3, i3, buf3 );
            i3 := i3 + 1;
            buf3.tab[1] := buf.tab[j];
            J3 := 2;
        FSI // ( j3 <= b )
        j := j + 1

    SINON // c-a-d non ( j <= buf.NB )
        SI ( i <= N )
            i := i + 1;
            SI Indic = 1
                LireDir( F1, i, buf )
            SINON
                LireDir( F2, i, buf )
            FSI
            j := 1
        SINON
            continu := faux
        FSI

    FSI // ( j <= buf.NB )

FTQ

// Le dernier buffer (buf3) n'a pas encore été écrit sur disque ...
buf3.NB := j3 - 1
EcrireDir( F3 , i3, buf3 )
Aff-entete( F3, 1, i3) // le nombre de blocs dans F3
Aff-entete( F3, 2, entete(F1,1) + entete(F2,1) ) // le nombre d'enregistrements dans F3

Fermer( F1 )
Fermer( F2 )
Fermer( F3 )

```

Fin

III- Méthodes d'index

1) Index primaire

Avec les structures de fichiers simples, quand le fichier de données devient volumineux, les opérations d'accès (recherche, insertion, ...) deviennent très inefficaces.

Les méthodes d'index permettent d'améliorer, dans une certaine mesure, les performances en gérant une structure auxiliaire (table d'index) accélérant la recherche. Un index est (généralement) une table ordonnée contenant les couples <clé, adr > utilisée pour accélérer la recherche des enregistrements d'un fichier. Si le champ clé ne contient de valeurs en double, l'index est alors « primaire »

Un index est dit « dense » s'il contient toutes les clés du fichier de données. Dans ce cas le fichier n'a pas à être gardé ordonné.

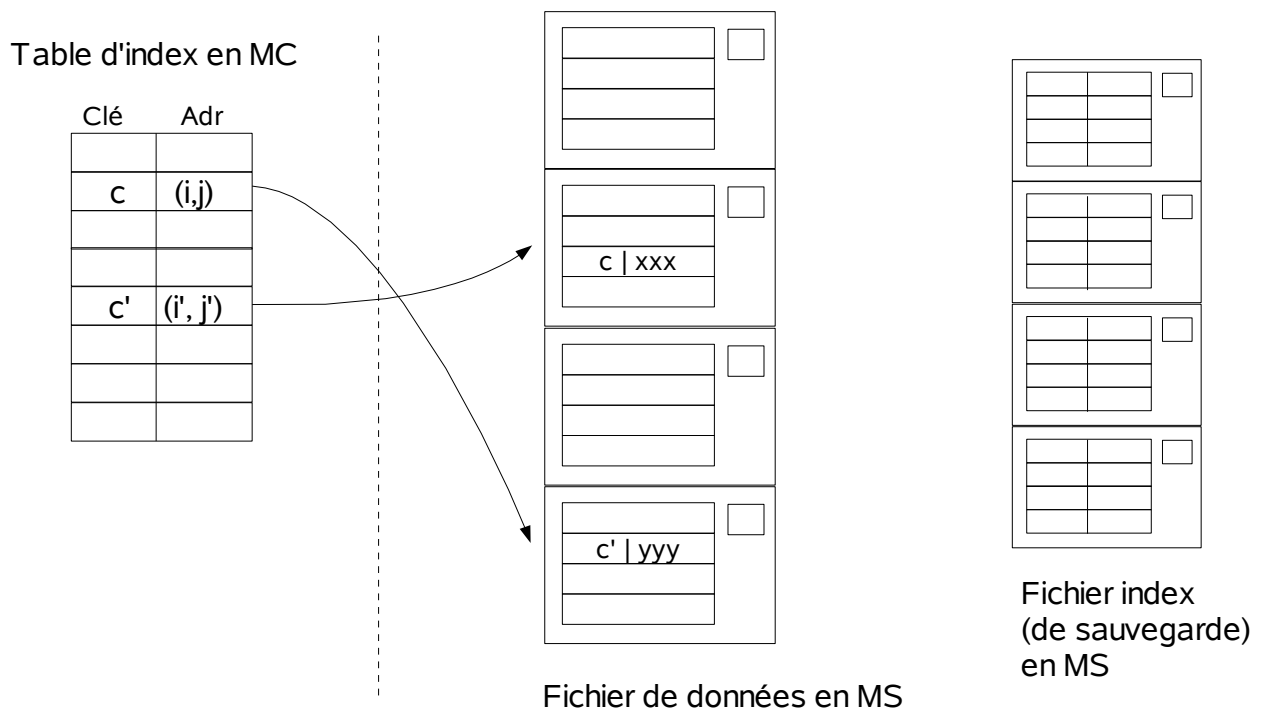
Un index est dit « non dense » s'il ne contient pas toutes les clés du fichier de données (par exemple on garde une clé par bloc). Dans ce cas le fichier doit être ordonné.

Index à un niveau

Généralement, le fichier de données n'est pas ordonné
=> simplifie l'insertion et la suppression

On maintient en MC une table ordonnée, contenant toutes les clés du fichier de données (index dense). A chaque clé est alors associée l'adresse de l'enregistrement dans le fichier de données. L'adresse est un couple de nombres : <num_bloc, déplacement>

Pour ne pas avoir à reconstruire la table d'index à chaque démarrage, on sauvegarde la table dans un fichier en fin de traitement.



Les fichiers de données et d'index peuvent être de n'importe quelle structure (blocs contigus, blocs chaînés, ...). De même que les enregistrements peuvent être à format fixe ou variable (avec ou sans chevauchement).

Les opérations de base

- La recherche d'un enregistrement consiste à faire une recherche dichotomique de sa clé dans la table d'index, si elle existe, on récupère l'enregistrement à partir du fichier de donnée avec un seul accès disque.

=> coût de l'opération : 1 accès disque (au max).

- La requête à intervalle consiste à rechercher tous les enregistrements dont la clé appartient à un intervalle de valeurs donné $[a,b]$.

- 1- On commence par rechercher la plus petite clé $\geq 'a'$ dans l'index (recherche dichotomique en MC).

- 2- Puis on continue séquentiellement dans la table jusqu'à trouver une clé $> 'b'$.

- 3- Pour chaque clé, on accède au fichier de données pour récupérer l'enregistrement.

Ce dernier point peut être amélioré si on tri les numéros de blocs trouvés avant d'accéder au fichier de données (afin de ne pas lire 2 fois le même bloc)

=> coût de l'opération en nombre d'accès : le nombre de clés vérifiant la requête (au max).

- L'insertion d'un nouvel enregistrement se fait en fin de fichier de données. Sa clé est insérée dans la table d'index (en MC) avec décalages pour garder l'ordre des clés.

=> coût de l'opération : 2 accès disques

- La suppression dans le cas d'un format variable avec chevauchement sera logique s'il n'y a pas de gestion de trous.

=> coût : 0 accès si le bit d'effacement se trouve dans la table d'index, 1 à 2 accès sinon.

Dans les autres cas, la suppression peut être physique en déplaçant par exemple le dernier enregistrement du fichiers à la place de celui effacé.

=> coût : 4 accès disques (au max)

Cas particuliers

Si le fichier de données est ordonné, on ne garde qu'une clé par bloc (par exemple la plus grande du bloc) dans la table d'index. C'est alors un index non dense (car il ne contient pas toutes les clés). Cela permet de diminuer la taille de la table d'index tout en gardant pratiquement les mêmes performances de recherche qu'une méthode d'index avec fichier non ordonné.

La requête à intervalle est beaucoup plus performantes, car dans un même bloc on peut récupérer plusieurs enregistrements vérifiant la condition de la requête.

L'insertion est par contre coûteuse, à cause des décalages dans le fichier de données.

Une solution serait alors de maintenir une zone de débordement dédiée aux enregistrements issus des décalages inter-blocs.

Les suppressions sont logiques.

On peut aussi représenter l'index en mémoire centrale sous forme d'un arbre de recherche binaire (au lieu d'une table ordonnée). L'avantage est d'éviter les décalages lors de l'insertion de nouvelles clés dans l'index. L'inconvénient est le risque du déséquilibre de l'arbre ou alors le surcoût associé au maintien de l'équilibrage par des algorithmes

appropriés.

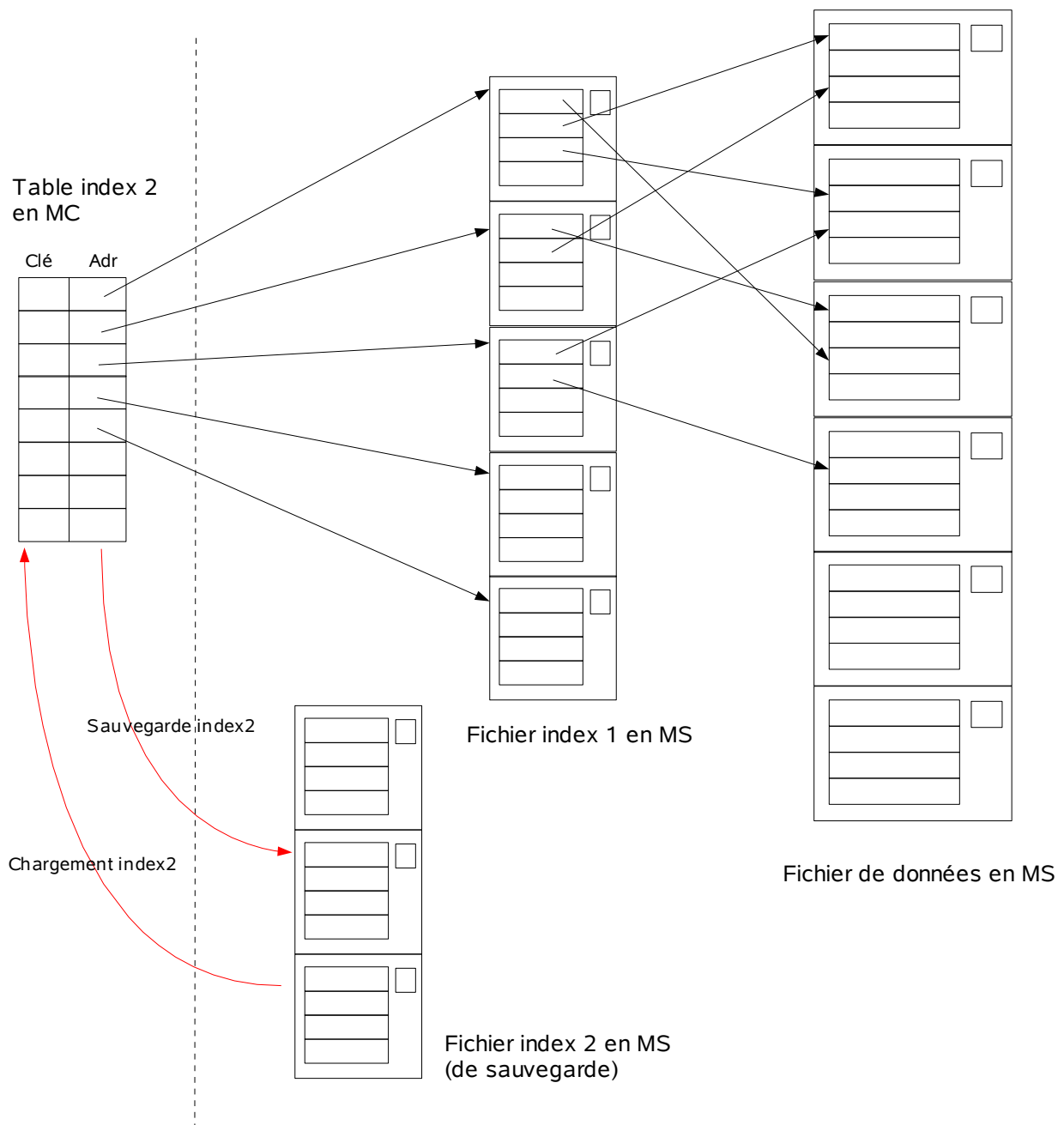
Index multiniveaux

Si l'index est trop grand pour résider en MC, on construit un deuxième index sur le fichier index (ordonné). Dans ce cas, on choisira une seule clé pour chaque bloc du fichier index (index non dense) pour construire le 2e index.

Si le 2e index est encore trop grand pour résider en MC, on le stocke sur disque (fichier 2e index) et on construit un 3e index en choisissant une clé par bloc du 2e fichier index.

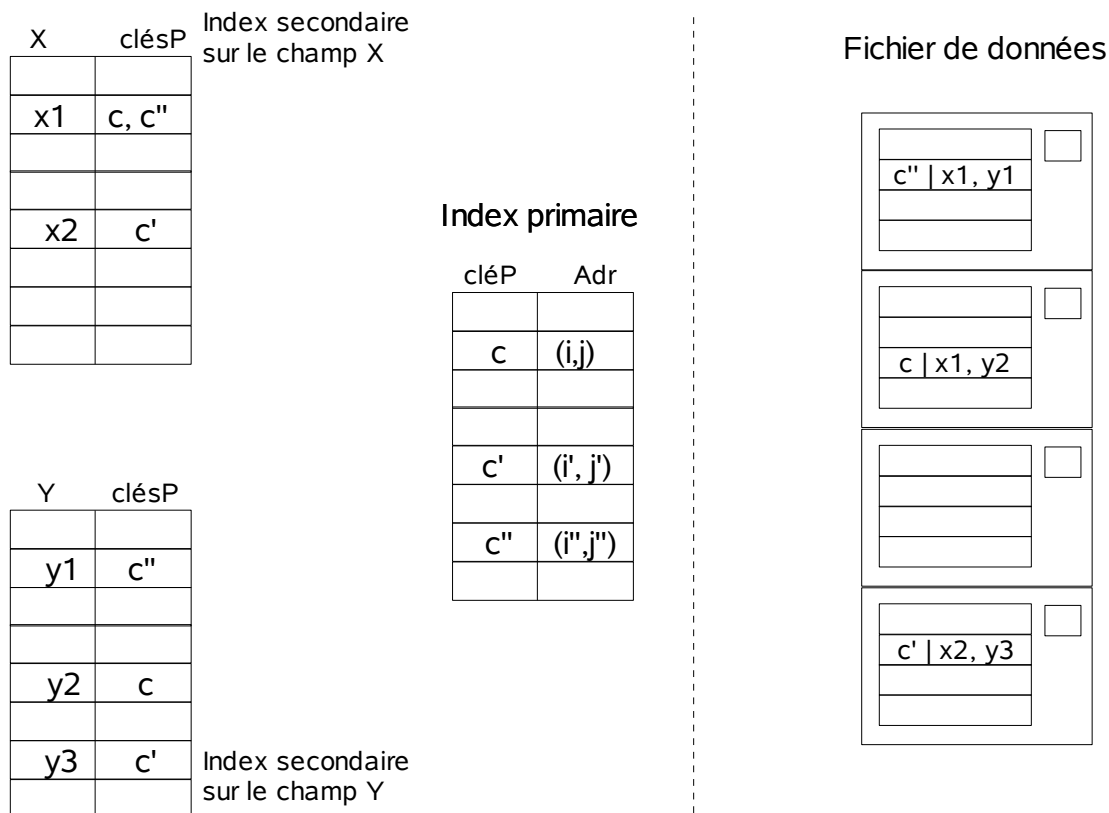
On peut répéter ce procédé au tant que nécessaire.

Dans la figure ci-dessous, un index à 2 niveaux:

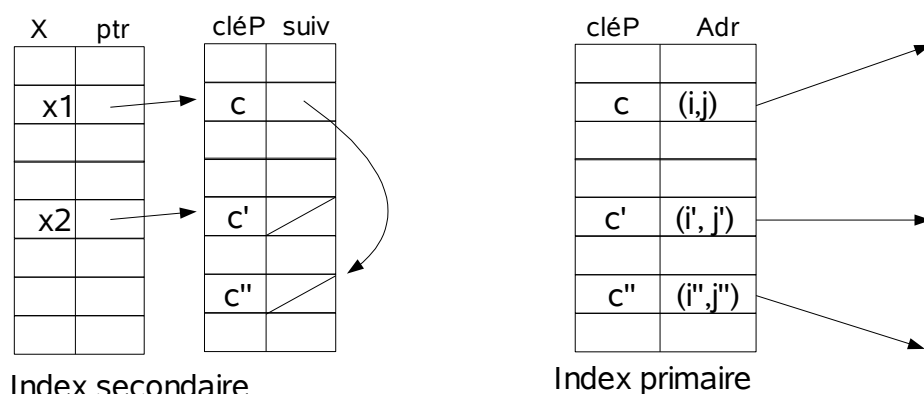


2) Index secondaire

Pour améliorer les recherches basées sur des champs non clés (appelés aussi clés secondaires), on peut construire des index secondaires sur ces champs.



Le problème avec les clés secondaires, est qu'il peut exister plusieurs enregistrements pour une valeur du champ indexé. On implémente généralement cette multiplicité à travers des listes de clés primaires



Quand on recherche les enregistrements suivant une clé secondaire (par exemple $X=a$), on utilise l'index secondaire sur ce champ pour récupérer la ou les clés primaires associées à la valeur cherchée (a). Pour chaque clé primaire trouvée, on utilise l'index primaire pour localiser l'enregistrement sur le fichier de données (numéro de bloc et déplacement). C'est la méthode des listes inversées.

Pour les recherches multi-critères de la forme « trouver tous les enregistrements dont la valeur de $X = a$ ET la valeur de $Y = b$ ET ... » avec X, Y, \dots des clés secondaires, on procède comme suit :

- a- En utilisant l'index secondaire X , trouver la liste L_x de clés primaires associées à la valeur de X (a).
- b- Refaire la même action pour chaque clé secondaire mentionnée dans la requête...
- c- Faire l'intersection des listes de clés primaires L_x, L_y, \dots pour trouver les clés primaires associées avec chaque valeur de clé secondaire mentionnée dans la requête.
- d- Utiliser alors l'index primaire pour retrouver les enregistrements du fichier de données

Pour insérer un enregistrement $\langle c, x, y, \dots \rangle$ avec c sa clé primaire et x, y, \dots ses clés secondaires, on procède comme suit :

- a- recherche c dans l'index primaire pour vérifier qu'elle n'existe pas déjà et pour trouver l'indice ip où doit être insérée cette clé (recherche dichotomique)
- b- Insérer l'enregistrement à la fin du fichier de données. Soit (i, j) son adresse
- c- Insérer le couple $\langle c, (i, j) \rangle$ dans la table d'index primaire, à l'indice ip (en procédant par décalages)
- d- rechercher la valeur x dans l'index secondaire X ,
 - si x existe, rajouter c à la liste pointée par x
 - si x n'existe pas, insérer x (par décalages) dans la table X . La nouvelle entrée x , pointe une liste formée par une seule clé primaire (c).
- e- refaire l'étape d) pour chaque clé secondaire restante.

Pour supprimer un enregistrement de clé primaire c , il suffit de positionner un bit d'effacement au niveau de la table d'index primaire, au niveau de l'entrée c .

Cela permet de ne pas avoir à mettre à jour tous les index secondaires.

Au niveau du fichier de données, l'enregistrement peut être supprimé physiquement si la structure du fichier le permet (comme par exemple T~OF).

Les méthodes d'index comme celles présentées dans ce chapitre, sont dédiées aux fichiers statiques (c-a-d dans les cas où le nombre d'insertions et de suppressions est relativement faible).