# Khawarizm

Environnement d'écriture d'algorithmes abstraits

et de passage vers les langages PASCAL et C

Pr D.E ZEGOUR

Ecole Supérieure d'Informatique

http:://Zegour.esi.dz d_zegour@esi.dz

# S O M M A I R E

# 1. Présentation générale

## Présentation

KHAWARIZM est un environnement pour apprendre et approfondir les principales structures de données et de fichiers.

KHAWARIZM offre la possibilité d'écrire des algorithmes dans un langage algorithmique (langage Z), de les arranger, de les dérouler ou les simuler et de fournir toute la documentation nécessaire pour les traduire vers les langages de programmation PASCAL et C.

KHAWARIZM vise la conception assistée des algorithmes.

KHAWARIZM assiste aussi l'utilisateur pour traduire son algorithme en PASCAL ou C.

## Menus

KHAWARIZM offre plusieurs fenêtres montrant :

- les données (lectures)
- les résultats de l'exécution (écritures)
- les résultats de la simulation (trace complète)

A tout moment dans KHAWARIZM, vous pouvez invoquer l'aide (F1) ou actionner les opérations à l'aide de boutons

## Traitements

KHAWARIZM offre les services suivants

- Un éditeur pour écrire vos algorithmes fournissant toute la documentation sur le langage Z.

- Un indenteur pour arranger vos algorithmes.
  Ses principales fonctions sont :

  . Chaque instruction est écrite sur une ligne différente,
  . Les mots-clé sont réécrits en majuscule (ou minuscule),
  . Le premier caractère de tout identificateur est réécrit en majuscule,
  . Les structures de contrôle sont mises en relief,
  . Les instructions de même niveau commencent sur la même colonne.
  . Le "pas d'aération" est variable.

- Un interpréteur pour exécuter vos algorithmes en donnant comme résultat l'ensemble des écritures émises (Fenêtre Résultats).

- Un simulateur pour donner le déroulement complet (fenêtre "Simulation") de vos algorithmes en montrant l'évolution de tous les objets manipulés. Ce qui vous aide à corriger, voir construire vos algorithmes.

- Une documentation importante pour montrer le passage d'un Z-algorithme vers un programme PASCAL ou C grâce un hypertexte intégré.

## Langage Z

Dans KHAWARIZM, Les algorithmes sont exprimés dans un langage algorithmique (le langage Z).

La particularité du langage Z réside dans le fait de pouvoir écrire des algorithmes sur des machines abstraites simulant les principales structures de données.
Le langage Z est conçu principalement pour les objectifs suivants :

- l'expérimentation sur les principales structures de données, peu importe leurs implémentations, en développant des algorithmes sur
  . Les vecteurs,
  . Les structures,
  . Les listes linéaires chaînées,
  . Les listes bilatérales,
  . Les files d'attente,
  . Les piles,
  . Les arbres de recherche binaire,
  . Les arbres de recherche m-aire.

- la création et la manipulation de structures de données complexes telles que
  . Liste de files d'attente,
  . Liste de piles,
  . Arbre de listes,
  . Liste de piles de vecteurs,
  . Etc.

- l'écriture d'algorithmes récursifs.

Grâce à sa machine abstraite définie sur les fichiers, le langage Z permet aussi l'utilisation des fichiers et la construction aussi bien de structures simples que complexes de fichiers.

## Documentation

KHAWARIZM offre toute la documentation sur le langage Z.

KHAWARIZM fournit les équivalents Z --> PASCAL et Z --> C.

KHAWARIZM donne quelques implémentations possibles en PASCAL et en C des différentes machines abstraites considérées dans le langage Z.

## 2. Etapes de réalisation d'un programme sous KHAWARIZM

### Familiarisation avec le langage algorithmique Z

Apprendre le langage algorithmique utilisé. Utiliser l'aide en ligne.

### Edition de l'algorithme

Ecrire un algorithme ou corriger un algorithme existant.

### Vérification syntaxique

Lancer le module Arranger.
Répéter tant qu'il y a des erreurs
. Corriger les erreurs
. Relancer le module Arranger

A ce stade, votre algorithme est bien écrit et il a été indenté pour vous.( Vous pouvez changer les modes de présentation de votre algorithme ( voir "Options" du menu )

### Exécution

Lancer l'exécution de votre algorithme
Les fenêtres montrent alors
- les données lues par votre algorithme (Bouton Données)
- les écritures émises par votre algorithme (Bouton Résultats)

Ou bien votre algorithme donne les résultats attendus ou pas. Dans ce dernier cas, lancer la simulation pour essayer de déterminer les erreurs de logique.

### Simulation

Lancer la simulation de votre algorithme. Il s'agit d'une exécution avec une trace.
Les fenêtres montrent alors
- les données lues par votre algorithme (Bouton Données)
- les écritures émises par votre algorithme (Bouton Résultats)
- tous les changements effectués sur les objets utilisés (Bouton Simulation)

Vous avez ainsi la trace complète de votre algorithme que vous pouvez imprimer et l'analyser pour détecter les erreurs.

Si vous désirez voir de plus près les différents pas de votre algorithme, demander une trace.

### Trace

Redemander la simulation avec trace. Vous pouvez alors suivre pas à pas l'évolution de votre algorithme, sortir de la boucle courante ou même du module courant.

Afin d'éviter d'avoir une trace complète qui peut être longue il est possible de limiter la longueur des boucles utilisées dans votre algorithme. Vous pouvez changer les modes de simulation (voir "Options" du menu).

## Passage vers un langage de programmation

Une fois que votre algorithme "tourne", il est possible de le traduire en PASCAL ou en C.Pour cela, vous devez aller à l'éditeur, ouvrir deux fenêtres que vous organisez en "Tuile" par exemple. L'une contient votre algorithme et l'autre le résultat de votre traduction. Utiliser alors l'aide concernant le passage vers PASCAL ou C.

Dans cette aide, vous trouverez
- les équivalents Z vers PASCAL et Z vers C.
- toutes les implémentations des machines Z.

La tâche de Khawarizm s'arrête à ce niveau là.

## Programmation PASCAL ou C

Utiliser le compilateur PASCAL ou C, pour finaliser définitivement votre programme. En particuliers, vous devez rajouter tous les modules de saisie des données et de restitution des résultats.

## Exemple d'un Z-algorithme

```
{Inclusion d'une liste linéaire chaînée dans une autre ?}
  SOIENT
     L1 , L2 DES LISTES ;
     Rech , Tous : FONCTION ( BOOLEEN ) ;

  DEBUT
     CREER_LISTE ( L1 , [ 2 , 5 , 9 , 8 , 3 , 6 ] ) ;
     CREER_LISTE ( L2 , [ 12 , 5 , 19 , 8 , 3 , 6 , 2 , 9 ] ) ;
     ECRIRE ( Tous ( L1 , L2 ) )
  FIN

  {Recherche d'une valeur dans une liste linéaire chaînée}
  FONCTION Rech ( L , Val ) : BOOLEEN
  SOIENT
     L UNE LISTE ;
     Val UN ENTIER ;

  DEBUT
     SI L = NIL
        Rech := FAUX
     SINON
        SI VALEUR ( L ) = Val
           Rech := VRAI
        SINON
           Rech := Rech ( SUIVANT ( L ) , Val )
        FSI
     FSI
  FIN
```

```
{Inclusion d'une liste dans une autre}
FONCTION Tous ( L1 , L2 ) : BOOLEEN
SOIENT
   L1 , L2 DES LISTES ;

DEBUT
   SI L1 = NIL
      Tous := VRAI
   SINON
      SI NON Rech ( L2 , VALEUR ( L1 ) )
         Tous := FAUX
      SINON
         Tous := Tous ( SUIVANT ( L1 ) , L2 )
      FSI
   FSI
FIN
```

**Equivalent PASCAL**

```
PROGRAM S11;

{ Implémentation de la liste }
TYPE
  Typeelem = INTEGER;
  Pointeur = ^Maillon;
  Maillon = RECORD
    VAL : Typeelem;
    Suiv : Pointeur
  END;

{ Implémentation des listes linéaires chaînées }
PROCEDURE Allouer ( VAR P : Pointeur ) ;
  BEGIN  NEW(P)  END;

PROCEDURE Liberer ( P : Pointeur ) ;
  BEGIN  DISPOSE(P)  END;

PROCEDURE Aff_val(P : Pointeur; VAL : Typeelem );
  BEGIN P^.VAL := VAL  END;

FUNCTION Valeur (P : Pointeur) : Typeelem;
  BEGIN   Valeur := P^.VAL  END;


FUNCTION Suivant( P : Pointeur) : Pointeur;
  BEGIN  Suivant := P^.Suiv END;

PROCEDURE Aff_adr( P, Q : Pointeur ) ;
  BEGIN P^.Suiv := Q    END;

{-------   CREER_LISTE   ----------------------}
TYPE T = ARRAY[1..10] OF INTEGER;
```

```
PROCEDURE Creer_liste( V:T; N : BYTE; VAR L : Pointeur);
    VAR
     I : BYTE;
     Q, P : Pointeur;
    BEGIN
     FOR I:= 1 TO N DO
      BEGIN
       Allouer(Q);
       Aff_val(Q, V[I]);
       IF  I<>1 THEN Aff_adr(P, Q) ELSE L := Q;
       P := Q
      END;
    END;

  {-----------------------------------------------------}
  FUNCTION Rech( L: Pointeur; VAL : INTEGER) : BOOLEAN;
    BEGIN
     IF L = NIL
     THEN Rech := FALSE
     ELSE IF Valeur(L) = VAL
          THEN Rech := TRUE
          ELSE Rech := Rech( Suivant(L), VAL)
    END;

  FUNCTION Tous ( L1, L2 : Pointeur ) : BOOLEAN;
    BEGIN
     IF L1 = NIL
     THEN Tous := TRUE
     ELSE
      IF NOT Rech(L2, Valeur(L1) )
      THEN Tous := FALSE
      ELSE
       Tous :=  Tous(Suivant(L1), L2)
    END;

  {Variables globales}
  VAR
    L1, L2 : Pointeur;
  CONST
    V1 : T = (2,5, 9, 8, 7, 0, 0, 0, 0, 0);
    V2 : T = (2,15, 19, 18, 7, 2, 5, 8,9, 0);

  {Programme principal}
  BEGIN
    Creer_liste( V1, 6, L1);
    Creer_liste (V2, 10, L2);
    WRITELN( Tous(L1, L2) );
  END.
```

**Equivalent C**

```
#include <alloc.h>
#include "stdio.h"

/* Maillon*/
struct Maillon
  {
   int  Val ;
   struct Maillon *Suiv ;
  }  ;

/* Opérations du modèle */
struct Maillon *Allouer ( )
  {
   return  (  (struct Maillon *) malloc( sizeof(struct Maillon))   );
  }

void Aff_val(struct Maillon *P, int V)
  { P->Val =V; }

void Aff_adr( 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) ; }

//-------   CREER_LISTE   ----------------------
typedef int  T [9]  ;
void Creer_liste( T V, int N , struct Maillon **L )
  {
   int I;
   struct Maillon *Q, *P;
   for (I=0; I<N; I++)
         {
          Q = Allouer();
          Aff_val(Q, V[I]);
          if (I != 0)  Aff_adr(P, Q) ; else *L = Q;
          P = Q ;
         }
    Aff_adr(P,NULL);
  }
//-----------------------------------------------------
typedef int Boolean;
#define FALSE 0
#define TRUE 1
```

```
Boolean Rech( struct Maillon *L, int Val )
  {
    if (L == NULL)
          return ( FALSE );
    else if (Valeur(L) == Val)
              return (TRUE )  ;
             else return( Rech( Suivant(L), Val) )  ;
  }

Boolean Tous ( struct Maillon *L1, struct Maillon *L2  )
  {
    if ( L1 == NULL)
         return( TRUE ) ;
    else
          if ( ! Rech(L2, Valeur(L1) ) )
           return ( FALSE );
          else
            return ( Tous(Suivant(L1), L2) );
  }
// Variables globales
  struct Maillon *L1, *L2;
  T V1;
  T V2;

// Programme principal
main()
 {
  V1[0]= 2;      V1[1]= 5;     V1[2]= 9;
  V1[3]= 8;      V1[4]= 7;     V1[5]= 0;

  V2[0]= 2;      V2[1]= 15;     V2[2]= 19;
  V2[3]= 18;     V2[4]= 7;      V2[5]= 0;
  V2[6]= 5;      V2[7]= 8;      V2[8]= 9;

  Creer_liste( V1, 6, &L1);
  Creer_liste (V2, 9, &L2);
  printf( "Résultat : %d",Tous(L1, L2) );
 }
```

## 3. Langage Z (Base)

### Généralités sur le langage Z

> Un Z-algorithme est un ensemble de modules parallèles dont le premier est principal et les autres sont soient des actions composées soient des fonctions.

> La communication entre les modules se fait via les paramètres et/ou les variables globales.

> Les objets globaux sont définis dans le module principal.

> Le langage permet tout type de paramètres : scalaires, structures, file, vecteur, ..., et mêmes les types complexes.

> Le langage Z admet les modules récursifs.

> Le langage permet l'allocation dynamique de tableaux et de structures.

> Le langage permet l'affectation globale de tout type.

> Quatre types standards (scalaires) sont autorisés : **ENTIER**, **BOOLEEN**, **CARACTERE**, **CHAINE**.

> Certaines fonctions usuelles sont prédéfinies : **MOD**, **MIN**, **MAX** et **EXP**.

> Le langage est l'ensemble des algorithmes abstraits, écrits à base de modèles ou machines abstraites.

> On définit ainsi des machines abstraites sur
- les structures ou objets composés
- les vecteurs
- les listes mono directionnelles
- les listes bidirectionnelles
- les piles
- les files d'attente
- les arbres de recherche binaire
- les arbres de recherche m-aire
- les fichiers

> Le langage permet les types composés du genre **PILE de FILES de LISTES** ... dont la dernière est de type scalaire ou structure simple.

> Le langage peut être étendu avec d'autres machines abstraites dans les futures versions.

> Le langage est doté des opérations de haut niveau permettant de construire des listes, des arbres, des files, ... à partir d'un ensemble de valeurs ( expressions entières )

> Le langage offre deux fonctions très utiles permettant de générer aléatoirement des chaînes de caractères (**ALEACHAINE**) et des entiers (**ALEANOMBRE**).

> Le langage offre également deux fonctions pour la manipulation des chaînes de caractères (**LONGCHAINE** et **CARACT**).

> Le langage permet la lecture et l'écriture de scalaires, de vecteurs de n'importe quelle dimension et des structures simples ou complexes.

> Le format d'écriture est libre.

## Structure d'un Z-algorithme

**SOIENT**

Objets locaux et globaux
Annonce des modules

**DEBUT**

Instructions

**FIN**

Module 1
Module 2
...
Modules n

Chaque module est soit une action composée soit une fonction.

## Structures de contrôle

La boucle TANTQUE

**TANTQUE** Exp[ : ]

Instructions

**FINTANTQUE**

La boucle POUR

**POUR** V : = Exp1, Exp2 [,Exp3] [ : ]

Instructions

**FINPOUR**

Exp3 désigne l'incrément du pas. Par défaut, sa valeur est 1.

La conditionnelle

**SI** Exp [ : ]
Instructions
**FINSI**

L'alternative

**SI** Exp [ : ]

Instructions

**SINON**

Instructions

**FINSI**

## Actions élémentaires

Affectation

V := Exp

Affecte la valeur d'une expression à une variable.

Lecture

**LIRE** ( V1, V2, .... )

Introduit des données dans les variables V1, V2,...

Ecriture

**ECRIRE** ( Exp1, Exp2, ... )

Restitue les expressions Exp1, Exp2,....
(Expressions entières ou chaînes de caractères)

## Opérations de haut niveau

Elles permettent de remplir une structure de données (ou initialiser une machine) à partir d'un ensemble d'expressions.

**INIT_VECT** ( T, [Exp1, Exp2, ....] )
**INIT_STRUCT** ( S, [Exp1, Exp2, ....] )
**CREER_LISTE** ( L, [Exp1, Exp2, ....] )
**CREER_LISTEBI** ( LB, [Exp1, Exp2, ....] )
**CREER_ARB** ( A, [Exp1, Exp2, ....] )
**CREER_ARM** ( M, [Exp1, Exp2, ....] )
**CREER_FILE** ( F, [Exp1, Exp2, ....] )
**CREER_PILE** ( P, [Exp1, Exp2, ....] )

Exp1, Exp2, .... sont des expressions scalaires ou des structures simples.

Exemple
**CREER_LISTE** ( L, [12, 34, I, I+J, 45] )

Crée la liste linéaire chaînée L avec les valeurs entre crochets dans l'ordre indiqué.

### Fonctions standards

**MOD** (A, B)
Reste de la division entière de A par B.

**MAX**(A, B)
Maximum entre A et B.

**MIN**(A, B)
Minimum entre A et B.

**EXP**(A, B)
A exposant B

### Fonctions de génération aléatoire

**ALEACHAINE** ( N )
Fournit une chaîne aléatoire de N caractères parmi les caractères alphabétiques majuscule et minuscule.

**ALEAENTIER** ( N )
Fournit un nombre aléatoire entre 0 et N

### Fonctions sur chaînes de caractères

**LONGCHAINE** ( C )
Donne la longueur de la chaîne C

**CARACT**(C, I)
Fournit le I-ième caractère de la chaîne C

### Définition d'une action composée

**ACTION** Nom ( P1, P2, ...Pn )

Objets locaux et paramètres

**DEBUT**

Instructions

**FIN**

Les paramètres sont appelés par "référence", ce qui implique que les paramètres d'entrée ne sont pas protégés par l'action.
Une action composée doit être annoncée dans le module principal.

L'appel à une action se fait par
**APPEL** nom de l'action (paramètres réels)

### Définition d'une fonction

**FONCTION** Nom ( P1, P2, ...Pn ) : type

Objets locaux et paramètres

**DEBUT**

Instructions

**FIN**

Les paramètres sont appelés par "référence", ce qui implique que les paramètres d'entrée ne sont pas protégés par l'action.

Une fonction utilisateur doit être annoncée dans le module principal en précisant son type.

Il doit y exister dans le corps d'une fonction une affectation du genre
Nom := Expression.

### Scalaires

Quatre types standards sont autorisés : **ENTIER**, **CARACTERE**, **CHAINE**, **BOOLEEN**.

Définition des scalaires

[**SOIT**/**SOIENT**] <li> Sep Type

<Li> : Liste d'identificateurs séparés par des virgules.
Sep dans { **:**, **UN**, **UNE**, **DES** }
Type est un type standard.

Exemples

A, B, Trouv : **BOOLEENS** ;
X, Y, Z **DES BOOLEENS** ;
A : **ENTIERS** ;
X, Y, Z **DES CHAINES** ;

### Objets "Pointeurs"

On définit des variables de type "Pointeur" pour la manipulation des structures de données.

[**SOIT**/**SOIENT**] <li> Sep
**POINTEUR VERS** [sep] [ **DE** Typec **DE** Typec **DE** ....] [ **DE** Types]

<Li> : Liste d'identificateurs séparés par des virgules.
Sep dans { :, **UN**, **UNE**, **DES** }
Typec dans { **VECTEUR**, **PILE**, **LISTE**, **FILE**, **PILE**, **ARB**, **LISTEBI**, **ARM** }
Types est un scalaire ou une structure simple.

Remarque

On peut s'en passer de ce type. En effet, les déclarations
"**SOIT** L **UNE LISTE**" et
"**SOIT** L **UN POINTEUR VERS UNE LISTE**" sont équivalentes.

Exemples

```
P1 : POINTEUR VERS LISTE DE PILE;
P2, P2 DES POINTEURS VERS DES ARB;
```

## Expressions Z

Comme dans les langages de programmation.

Expressions arithmétiques : + , - , / , *
Expressions logiques : **ET**, **OU**, **NON**
Expressions sur chaînes de caractères : +
Expressions relationnelles : <, <=, >, >=, =, <> (ou #)

Constantes logiques : **VRAI**, **FAUX**
Constante pointeur : **NIL**

Exemples

```
B+C / F
NON Trouv
(X # 5) ET NON TROUV
F(X) <> 5
P = Nil
```

## Commentaires

Les commentaires peuvent être insérés dans tout endroit oŚ on peut avoir un blanc.
Les commentaires sont entre { et } ou entre /* et */.

## Exemple d'un Z-algorithme

```
SOIENT
  L1, L2 DES LISTES;
  Rech, Tous : FONCTION(BOOLEEN);
DEBUT
  CREER_LISTE(L1, [2, 5, 9, 8, 3, 6 ]);
  CREER_LISTE(L2, [12, 5, 19, 8, 3, 6, 2,9]);
  ECRIRE( Tous(L1, L2) )
FIN
/* Recherche de la valeur Val dans la liste L */
FONCTION Rech ( L, Val ) : BOOLEEN
```

```
SOIENT
  L UNE LISTE;
  Val UN ENTIER;
DEBUT
  SI L = NIL :  Rech := FAUX
  SINON
    SI VALEUR(L) = Val
      Rech := VRAI
    SINON
      Rech := Rech(SUIVANT(L), Val )
    FSI
  FSI
FIN

/* Détermine si tous les éléments de L1 sont dans L2 */
FONCTION Tous ( L1, L2 ) : BOOLEEN
SOIENT
  L1, L2 DES LISTES;
DEBUT
  SI L1 = NIL
    Tous := VRAI
  SINON
    SI NON Rech(L2, VALEUR(L1) )
      Tous := FAUX
    SINON
      Tous :=  Tous(SUIVANT(L1), L2)
    FSI
  FSI
FIN
```

## 4. Langage Z (Structures de données)

### Structures

Une structure est un ensemble d'éléments hétérogènes.
Une structure peut être simple, c'est à dire composée uniquement de scalaires.
Une structure peut être complexe, c'est à dire composée de scalaires et/ou de vecteurs à une dimension de scalaires.
Une structure peut être statique ou dynamique.

Définition des structures

[**SOIT**/**SOIENT**] <Li> Sep
[**STRUCTURE**] ( Type1, type2, ... ) [**DYNAMIQUE**]

<Li> : Liste d'identificateurs séparés par des virgules.
Sep dans { :, **UN**, **UNE**, **DES** }
Typei est soit un type scalaire soit un tableau à une dimension de scalaires.

Exemples

S1 : (**ENTIER**, **CHAINE**) ;
S2 **UNE STRUCTURE** ( **CHAINE**, **ENTIER**, **BOOLEEN**) ;
S3 **UN** ( **ENTIER**, **VECTEUR**(5) **DE CHAINE**) **DYNAMIQUE**;

### Tableaux

Un tableau est un ensemble d'éléments homogènes.
Un tableau ( ou vecteur) peut être simple, c'est à dire composé uniquement de scalaires.
Un tableau peut être complexe, c'est à dire composé de structures simples.
Un tableau peut être statique ou dynamique.

Définition des tableaux

[**SOIT**/**SOIENT**] <Li> sep
**VECTEUR**(Dim1, Dim2, ... )
**DE** Typec **DE** Typec **DE** .... **DE** Types [**DYNAMIQUE**]

<Li> : Liste d'identificateurs séparés par des virgules.
Sep dans {:, **UN**, **UNE**, **DES** }
Typec dans { **VECTEUR, PILE, LISTE, FILE, PILE, ARB, LISTEBI, ARM** }
Types est un scalaire ou une structure simple.

Exemples

V1 **UN TABLEAU** (5) **DYNAMIQUE**;
V2, V3 **DES VECTEURS** (3, 8) **DE CHAINES** ;

### Listes linéaires chaînées

Une liste linéaire chaînée est un ensemble de maillons alloués dynamiquement.
Un élément possède deux champs : Valeur et Adresse.

Le champ 'Valeur' peut être quelconque.

Définition des listes

[**SOIT/SOIENT**] <Li> Sep **LISTE** [ **DE** Typec **DE** Typec **DE** .... ] [ **DE** Types ]

<Li> : Liste d'identificateurs séparés par des virgules.
Sep dans {:, **UN**, **UNE**, **DES** }
Typec dans { **VECTEUR**, **PILE**, **LISTE**, **FILE**, **PILE**, **ARB**, **LISTEBI**, **ARM** }
Types est un scalaire ou une structure simple.

Exemples

L1 **UNE LISTE DE** (**CHAINE**, **ENTIER**);
L2 **UNE LISTE DE PILE DE CHAINE** ;
L3 **UNE LISTE DE CHAINES**;

## Listes linéaires chaînées bilatérales

Une liste linéaire chaînée bilatérale est un ensemble de maillons alloués dynamiquement qui peut être parcourue dans les deux sens.

Un élément possède trois champs : Valeur, adresse gauche, adresse droite.

Le champ 'Valeur' peut être quelconque.

Définition des listes bilatérales

[**SOIT/SOIENT**] <Li> Sep **LISTEBI** [**DE** Typec **DE** Typec **DE** .... ][ **DE** Types]

<Li> : Liste d'identificateurs séparés par des virgules.
Sep dans { **:**, **UN**, **UNE**, **DES** }
Typec dans { **VECTEUR, PILE, LISTE, FILE, PILE, ARB, LISTEBI, ARM** }
Types est un scalaire ou une structure simple.

Exemples

Lb1 **UNE LISTEBI DE** (**CHAINE**, **ENTIER**);
Lb2 **UNE LISTEBI DE PILE DE CHAINE** ;
Lb3 **UNE LISTEBI DE CHAINES**;

## Files d'attente

Une file d'attente est une collection d'éléments dans laquelle tout nouvel élément est inséré à la fin et tout retrait se fait du début. C'est le principe FIFO : First In First Out
Un élément peut être quelconque

Définition des files d'attente

[**SOIT/SOIENT**] <Li> Sep **FILE** [**DE** Typec **DE** Typec **DE** .... ] [**DE** Types]
<Li> : Liste d'identificateurs séparés par des virgules.
Sep dans { **:**, **UN**, **UNE**, **DES** }

Typec dans { **VECTEUR**, **PILE**, **LISTE**, **FILE**, **PILE**, **ARB**, **LISTEBI**, **ARM** }
Types est un scalaire ou une structure simple.

Exemples

F1 **UNE FILE DE** (**CHAINE**, **ENTIER**);
F2 **UNE FILE DE PILE DE CHAINE** ;
F3 **UNE FILE DE CHAINES**;

## Piles

Une pile est une collection d'éléments dans laquelle tout nouveau élément est inséré à la fin et tout retrait se fait également de la fin. C'est le principe LIFO : Last In First Out.

Un élément peut être quelconque.

Définition des piles

[**SOIT**/**SOIENT**] <Li> Sep **PILE** [**DE** Typec **DE** Typec **DE** .... ][ **DE** Types]

<Li> : Liste d'identificateurs séparés par des virgules.
Sep dans { **:**, **UN**, **UNE**, **DES** }
Typec dans { **VECTEUR**, **PILE**, **LISTE**, **FILE**, **PILE**, **ARB**, **LISTEBI**, **ARM** }
Types est un scalaire ou une structure simple.

Exemples

P1 **UNE PILE DE** (**CHAINE**, **ENTIER**);
P2 **UNE PILE DE PILE DE CHAINE** ;
P3 **UNE PILE DE CHAINES**;

## Arbres de recherche binaire

Un arbre de recherche binaire est une structure de données généralement dynamique non linéaire.

Un nœud d'un arbre de recherche binaire contient une information et deux fils.

Structure d'un nœud : (a1, V1, a2)
Toutes les données rangées dans le sous arbre a1 sont strictement inférieures à V1.
Toutes les données rangées dans le sous arbre a2 sont strictement supérieures à V1.
Un élément (nœud) peut être quelconque.

Définition des arbres de recherche binaire

[**SOIT**/**SOIENT**] <Li> Sep **ARB** [**DE** Typec **DE** Typec **DE** .... ] [**DE** Types]

<Li> : Liste d'identificateurs séparés par des virgules.
Sep dans { **:**, **UN**, **UNE**, **DES** }
Typec dans { **VECTEUR**, **PILE**, **LISTE**, **FILE**, **PILE**, **ARB**, **LISTEBI**, **ARM** }
Types est un scalaire ou une structure simple.

Exemples

A1 **UN ARB DE** (**CHAINE**, **ENTIER**);
A2 **UN ARB DE PILE DE CHAINE** ;
A3 **UN ARB DE CHAINES**;

## Arbres de recherche m-aire

Un arbre de recherche m-aire est une structure de données généralement dynamique non linéaire.
C'est une généralisation de l'arbre de recherche binaire.
Un nœud d'un arbre de recherche m-aire d'ordre p contient (p-1) informations et p fils.

Structure d'un nœud : (a1, V1, a2, V2, .....,Vp-1, ap)

Toutes les données rangées dans le sous arbre a1 sont strictement inférieures à V1.
Toutes les données rangées dans le sous arbre ap sont strictement supérieures à Vp-1.
Toutes les données rangées dans le sous arbre ai ( 1 < i < p ) sont strictement inférieures à Vi et strictement supérieures à Vi+1.

Un élément (nœud) peut être quelconque.

Définition des arbres de recherche m-aire

[**SOIT**/**SOIENT**] <Li> Sep **ARM** (degre) [ **DE** Typec **DE** Typec **DE** ....] [ **DE** Types]

<Li> : Liste d'identificateurs séparés par des virgules.
Sep dans { **:**, **UN**, **UNE**, **DES** }
Typec dans { **VECTEUR, PILE, LISTE, FILE, PILE, ARB, LISTEBI, ARM** }
Types est un scalaire ou une structure simple.

Exemples

M1 **UN ARM**(4) **DE** (**CHAINE**, **ENTIER**);
M2 **UN ARM**(2) **DE PILE DE CHAINE** ;
M3 **UN ARM**(3) **DE CHAINES**;

## Fichiers

Un fichier est un ensemble d'enregistrements ( ou structures ) rangés généralement sur un disque.
Les enregistrements peuvent être des articles pour un utilisateur.
Les enregistrements peuvent être des blocs pour un concepteur.
Le fichier renferme une partie indispensable ( En-tête )pour la conception de structures de fichiers.

Définition des fichiers

[**SOIT**/**SOIENT**] <Li> sep **FICHIER DE** type
**BUFFER** <Li>
[**ENTETE** ( Type1, Type2, .... ) ]
<Li> : Liste d'identificateurs séparés par des virgules.
Sep dans { :, **UN**, **UNE**, **DES** }

Une définition de fichier comporte 3 parties :

La première partie (**FICHIER**) précise la nature des éléments du fichier.
Un élément (article ou bloc) du fichier peut être
- un scalaire
- un vecteur à une dimension de scalaires
- une structure complexe ( pouvant contenir des scalaires et/ou des vecteurs à une dimension de scalaires)

La deuxième partie (**BUFFER**) définit les variables Tampon utilisées dans les opérations de lecture/écriture.

La troisième partie (**ENTETE**) définit les caractéristiques du fichier en précisant le type de chacune d'entre elles. Cette partie est facultative et est surtout utilisée pour la création de structures de fichiers. Elle sert à mémoriser toutes les informations utiles pour l'exploitation du fichier.

Exemples

```
F1 UN FICHIER DE CHAINES BUFFER V1, V2;
F2 UN FICHIER DE VECTEUR(5) DE ENTIER BUFFER V ;
F3 UN FICHIER DE (ENTIER, VECTEUR(3) DE CAR) BUFFER V
ENTETE(ENTIER,ENTIER) ;
F4 UN FICHIER DE CAR BUFFER V ENTETE (ENTIER, CHAINE, BOOLEEN) ;
```

## 5. Machines abstraites

Sur chaque structure de données, une machine abstraite est définie avec son ensemble d'opérations.

[**SOIT**/**SOIENT**] <li> Sep <Machine abstraite>

<Li> : Liste d'identificateurs séparés par des virgules.
Sep dans {:, **UN**, **UNE**, **DES** }

Exemples

L1, L2 **DES LISTES**;
F **UNE FILE**;
V1 **UN VECTEUR**(10, 60);
Y **UNE LISTE DE PILES DE VECTEUR**(5);

### Machine abstraite sur les vecteurs

**ELEMENT** ( T [i, j, ...] )

Accès à l'élément T[i, j, ...]du vecteur T.

**AFF_ELEMENT** ( T [I, J, ...], Val )

Affecter à l'élément T[i, j, ...] la valeur Val.

**ALLOC_TAB** ( T )

Allocation d'un tableau de taille spécifiée par la définition de T.L'adresse est rendue dans la variable T.

**LIBER_TAB** ( T )

Libération de l'espace mémoire pointé par T.

### Machine abstraite sur les structures

**STRUCT** ( S, i)

Accès au i-ème champ de la structure S

**AFF_STRUCT** ( S, i, Exp )

Affecter à la structure S dans son i-ème champ l'expression Exp.

**ALLOC_STRUCT** ( S )

Allocation d'un espace mémoire de taille spécifiée par la définition de S. L'adresse est rendue dans la variable S.

**LIBER_STRUCT** ( S )

Libération de l'espace mémoire pointé par S.

### Machine abstraite sur les listes linéaires chaînées

**ALLOUER** ( P )

Crée un maillon et retourne son adresse dans P.

**LIBERER** ( P )

Libère le nœud d'adresse P.

**SUIVANT** ( P )

Accès au champ 'Adresse' du nœud référencé par P.

**VALEUR** ( P )

Accès au champ 'Valeur' du nœud référencé par P.

**AFF_ADR** ( P, Q )

Affecter au champ 'Adresse' du nœud référencé par P, l'adresse Q.

**AFF_VAL**( P, Val )

Affecter au champ 'Valeur' du nœud référencé par P, la valeur Val.

### Machine abstraite sur les listes bidirectionnelles

**ALLOUER** ( P )

Crée un maillon et retourne son adresse dans P.

**LIBERER** ( P )

Libère le nœud d'adresse P.

**SUIVANT** ( P )

Accès au champ 'Adresse droite' du nœud référencé par P.

**PRECEDENT** ( P )

Accès au champ 'Adresse gauche' du nœud référencé par P.

**VALEUR** ( P )

Accès au champ 'Valeur' du nœud référencé par P.

**AFF_ADRD** ( P, Q )

Affecter au champ 'Adresse droite' du nœud référencé par P, l'adresse Q

**AFF_ADRG** ( P, Q )

Affecter au champ 'Adresse gauche' du nœud référencé par P, l'adresse Q.

**AFF_VAL**( P, Val )

Affecter au champ 'Valeur' du nœud référencé par P, la valeur Val.

### Machine abstraite sur les files d'attente

**CREERFILE** ( F )

Crée une file d'attente vide.

**FILEVIDE** ( F)

Teste si une file d'attente est vide.

**ENFILER** ( F, Val )

Enfiler (rajouter en queue) la valeur Val dans la file d'attente F.

**DEFILER** ( F, Val )

Défiler ( récupérer de la tête ) une valeur pour la  mettre dans Val.

### Machine abstraite sur les piles

**CREERPILE** ( P )

Crée une pile vide.

**PILEVIDE** ( P )

Teste si une pile est vide.

**EMPILER** ( P, Val )

Empiler (rajouter au sommet) la valeur Val dans la pile P.

**DEPILER** ( P, Val )

Dépiler ( récupérer du sommet ) une valeur pour la  mettre dans Val.

### Machine abstraite sur les arbres de recherche binaire

**CREERNOEUD** ( Val )

Crée un nœud avec l'information Valet retourne l'adresse du nœud. Les autres champs sont à NIL.

**LIBERERNOEUD** ( P )

Libère le nœud d'adresse P.

**FG** ( P )

Accès au champ Fils gauche du nœud référencé par P.

**FD** ( P )

Accès au champ Fils droit du nœud référencé par P.

**PERE** ( P )

Accès au champ Père du nœud référencé par P.

**INFO** ( P )

Accès au champ Info du nœud référencé par P.

**AFF_FG** ( P, Q )

Affecter au champ Fils gauche du nœud référencé par p, l'adresse Q

**AFF_FD** ( P, Q )

Affecter au champ Fils droit du nœud référencé par p, l'adresse Q

**AFF_PERE** ( P, Q )

Affecter au champ Père du nœud référencé par p, l'adresse Q

**AFF_INFO**( P, Val )

Affecter au champ Info du nœud référencé par p, la valeur Val

## Machine abstraite sur les arbres de recherche m-aire

**CREERNOEUD** ( Val )

Crée un nœud avec l'information Val et retourne l'adresse du nœud. Les autres champs sont à NIL.

**LIBERERNOEUD** ( P )

Libère le nœud d'adresse P.

**FILS** ( P, I )

Accès au champ I-ième fils du nœud référencé par P.

**PERE** ( P )

Accès au champ Père du nœud référencé par P.

**INFOR** ( P, I )

Accès au I-ième champ Info du nœud référencé par P.

**AFF_FILS** ( P, I, Q )

Affecter au champ I-ième fils du nœud référencé par P, l'adresse Q.

**AFF_PERE** ( P, Q )

Affecter au champ Père du nœud référencé par P, l'adresse Q.

**AFF_INFOR**( P, I, Val )

Affecter au I-ième champ Information du nœud référencé par P, la valeur Val.

## Machine abstraite sur les fichiers

**OUVRIR** (Fl, Fp, Mode)

Ouvrir le fichier logique Fl et l'associer au fichier physique Fp en précisant le mode( fichier nouveau('N') ou ancien 'A') )

**FERMER** (Fl)

Fermer le fichier Fl.

**LIRESEQ** (Fl, V)

Lire dans la variable tampon V le bloc (ou l'article)se trouvant à la position courante.

**ECRIRESEQ** (Fl, V)

Ecrire le contenu de la variable tampon V à la position courante du fichier Fl.

**LIREDIR** (Fl, V, N) :

Lire le N-ième bloc (ou article) du fichier Fl dans la variable tampon V.

**ECRIREDIR** (Fl, V, N)

Ecrire le contenu de la variable tampon V à la N-ième position du fichier Fl.

**RAJOUTER**(Fl, V)

Ecrire le contenu de la variable tampon à la fin du fichier Fl.

**FINFICH**(Fl)

Prédicat égal à vrai si la fin du fichier Fl est rencontrée, faux sinon.

**ALLOC_BLOC**(Fl)

Fournit un bloc (ou article) du fichier dans lequel on pourra écrire.

**ENTETE**(Fl, I)

Récupérer la I-ième caractéristique du fichier Fl.

**AFF_ENTETE**(Fl, I, Exp)

Affecter Exp comme la I-ème caractéristique du fichier.

## 6. Passage de Z vers PASCAL

### Déclaration des variables

Une déclaration de variables PASCAL se fait par
VAR <Li> : Type;
où <Li> désigne une liste d'identificateurs.

**SOIT** (**SOIENT**) se traduit par VAR.

Les objets simples

Equivalents des objets Z --> PASCAL

| Z | PASCAL |
|---|---|
| **ENTIER** | INTEGER |
| **BOOLEEN** | BOOLEAN |

### Boucle "TANTQUE"

-------Z---------
**TANTQUE** Cond :

instructions

**FINTANTQUE**

-------PASCAL------
WHILE ( Cond ) DO
BEGIN
Instructions
END

### Boucle "POUR"

----------Z----------
**POUR** V:= Exp1, Exp2 [, Exp3] :
Instructions
**FINPOUR**

Si Exp3 est absent ou égale à 1

Se traduit par :

--------PASCAL---------
FOR V:= Exp1 TO Exp2 DO
BEGIN
Instructions
END

Si Exp3 <> 1 :

```
----------PASCAL------
V := Exp1;
WHILE ( V <= Exp2 ) DO
  BEGIN
     Instructions  ;
    V := V + Exp3
  END;
```

### L'alternative "SI"

```
----------Z----------
SI Cond :

     Instructions

[SINON

     Instruction  ]
FSI
```

Se traduit par :

```
-------PASCAL---------
IF Cond
THEN
  BEGIN
    Instructions
  END
[ELSE
   BEGIN
     Instructions
   END]
```

### Z-expressions

La grammaire des Z-expressions est incluse dans la grammaire PASCAL.

### Lecture

```
--------Z--------
   LIRE(V1, V2, ...)
```

Se traduit par :

```
   ------PASCAL------
   READLN(V1, V2, ...)
```

### Ecriture

```
----------Z----------
ECRIRE(Exp1, Exp2, ...)
```

Se traduit par

```
--------PASCAL----------
WRITELN(Exp1, Exp2, ...)
```

### Affectation

Même syntaxe

### Action composée

```
----------Z----------------
ACTION Nom ( P1, P2, ...)
SOIENT
     Définition des objets locaux
      et des paramètres
DEBUT
     Instructions
FIN
```

Se traduit par :

```
--------------------PASCAL--------------------
PROCEDURE Nom ( VAR P1: typ; VAR P2:typ, ...);
VAR
      Définition des objets locaux
BEGIN
      Instructions
END
```

### Fonctions

```
----------------Z----------------
FONCTION Nom ( P1, P2, ...) : Type
SOIENT
     Définition des objets locaux
      et des paramètres
DEBUT
     Instructions
FIN
```

Se traduit par :

```
---------------------PASCAL-------------------------
FUNCTION Nom ( VAR P1: typ; VAR P2:typ, ...) : Type;
VAR
      Définition des objets locaux
```

```
BEGIN
      Instructions
END
```

## Fonctions prédéfinies

**MOD** (a, b)

```
FUNCTION Mod (a, b : INTEGER) : INTEGER;
 BEGIN
   Mod := a Mod b
 END;
```

**MIN** (a, b)

```
FUNCTION Min (a, b: INTEGER) : INTEGER;
 BEGIN
   Min := a; IF b < a THEN Min := b;
 END;
```

**MAX** ( a, b )

```
FUNCTION Max (a, b: INTEGER) : INTEGER;
 BEGIN
   Max := a; IF b > a THEN Max := b;
 END;
```

**EXP** ( a, b )

```
FUNCTION Exp (a, b: INTEGER) : INTEGER;
 VAR I : INTEGER;
 BEGIN
   Exp := 1;
   FOR I:= 1 TO b DO Exp := Exp * a
 END;
```

**ALEAENTIER** ( N )

```
FUNCTION Aleaentier (N: INTEGER) : INTEGER;
 BEGIN
   Aleaentier := Random( N );
 END;
```

**ALEACHAINE** ( N )

```
FUNCTION Aleachaine(N: INTEGER) : STRING;
 VAR
   K : BYTE;
   Chaine : STRING;
 BEGIN
   Chaine := '';
   FOR K:=1 TO N DO
```

```
    CASE Random(2) OF
    0 : Chaine := Chaine + CHR(97+Random(26) ) ;
    1 : Chaine := Chaine + CHR(65+Random(26) )
    END;
   Aleachaine := Chaine;
 END;
```

**LONGCHAINE** ( C )

```
FUNCTION Longchaine(C : STRING): INTEGER;
 BEGIN
   Min := a; IF b < a THEN Min := b;
 END;
```

**Algorithme**

```
 ----------Z----------
 SOIENT

    Objets locaux et globaux
    Annonce des modules

 DEBUT

    Instructions

 FIN

 Module 1
 Module 2
 ...
 Modules n
```

Se traduit par :

```
 ----------PASCAL----------
 PROGRAM  Pascal;
  VAR
      Objets locaux et globaux

  { Définition des modules }
  Module 1
  Module 2
  ...
  Module n

  BEGIN
      Instructions
  END.
```

## 7. Implémentation des machines Z en PASCAL

### Implémentation des vecteurs en PASCAL / Statique

```
CONST Max =100;  Taille arbitraire
TYPE
  Typeqq =  { type d'un élément du tableau } ;
  Typevect = ARRAY[1..Max] OF Typeqq;

    ELEMENT ( V, I )

FUNCTION Element ( VAR V:Typevect; I: INTEGER ) : Typeqq;
  BEGIN
    Element :=  V[I];
  END;

    AFF_ELEMENT ( V, I, Val )

PROCEDURE Aff_element ( VAR V :Typevect; I:INTEGER; Val : Typeqq );
  BEGIN
    V[I] := Val;
  END;
```

### Implémentation des vecteurs en PASCAL / Exemple

```
VAR
  I : INTEGER;
  V : Typevect;

BEGIN
  Aff_element(V, 1, 34);
  Aff_element(V, 2, 56);
  Aff_element(V, 3, 89);
  Aff_element(V, 4, 38);
  Aff_element(V, 5, 156);

  FOR I:=1 TO 5 DO
    WRITELN(Element( V, I)  );

  Aff_element(V,3, 99);

  FOR I:= 1 TO 5 DO
    WRITELN(Element( V, I) );
          END.
```

### Implémentation des vecteurs en PASCAL /  Dynamique

```
CONST Max =100;  Taille arbitraire
TYPE
  Typeqq = { type d'un élément du tableau } ;
  Typevect = ARRAY[1..Max] OF Typeqq;
  Typevect_dyn = ^typevect;
```

```
   ALLOC_TAB ( T )

PROCEDURE Alloc_tab ( var T : Typevect_dyn );
 BEGIN
   NEW(T)
 END;

   LIBER_TAB ( T  )

PROCEDURE Liber_tab ( var T : Typevect_dyn );
 BEGIN
   DISPOSE(T)
 END;

   ELEMENT ( V, I )

FUNCTION Element ( V:Typevect_dyn; I: INTEGER ) : Typeqq;
 BEGIN
  Element :=  V^[I];
 END;

   AFF_ELEMENT ( V, I, Val )

PROCEDURE Aff_element ( VAR V :Typevect_dyn; I:INTEGER; Val : Typeqq );
 BEGIN
  V^[I] := Val;
 END;
```

**Implémentation des vecteurs en PASCAL / Exemple**

```
VAR
 I : INTEGER;
 V : Typevect_dyn;

BEGIN
 Alloc_tab(V);
 Aff_element(V, 1, 34);
 Aff_element(V, 2, 56);
 Aff_element(V, 3, 89);
 Aff_element(V, 4, 38);
 Aff_element(V, 5, 156);

 FOR I:=1 TO 5 DO
   WRITELN(Element( V, I)  );

 Aff_element(V,3, 99);

 FOR I:= 1 TO 5 DO
   WRITELN(Element( V, I) );
 Liber_tab(V);
END.
```

## Implémentation des structures en PASCAL / Statique

```
TYPE
 Type1 =  type du champ1;
 Type2 =  type du champ2;
 ...
 Typen =  type du champn;

 Typestruct = record
   Champ1 : Type1;
   Champ2 : Type2;
   ....
   ....
   Champ2 : Typen;
  END;

 VAR S : Typestruct;
```

STRUCT (S, I )

Si le type du champ I est un scalaire, STRUCT se traduit par une fonction comme suit :

```
FUNCTION STRUCTI ( S:Typestruct ) : TypeI;
  BEGIN
    StructI :=  S.champI;
  END;
```

Il y a donc autant de fonctions STRUCT que de champs scalaires dans la structure. Si le type du champ I est n'est pas scalaire, c'est à dire un vecteur à une dimension de scalaires, STRUCT se traduit par une procédure comme suit :

```
PROCEDURE STRUCTI ( S:Typestruct; VAR Result : TypeI);
  BEGIN
    Result :=  S.champI;
  END;
```

AFF_STRUCT (S, I, Exp)

```
PROCEDURE AFF_STRUCTI ( VAR S :Typestruct; Val : TypeI );
  BEGIN
    S.champI := Val;
  END;
```

Il y a donc autant de fonctions Aff_struct que de champs scalaire dans la structure.

## Implémentation des structures en PASCAL / Exemple

Le Z-algorithme suivant :

```
SOIENT
  S UNE STRUCTURE (ENTIER, VECTEUR(5) DE CHAINES);
  V1, V2 DES VECTEURS(5) DE CHAINES;
```

```
  I UN ENTIER ;
DEBUT
  AFF_ELEMENT(V1[1],'1');
  AFF_ELEMENT(V1[2],'2');
  AFF_ELEMENT(V1[3],'3');
  AFF_ELEMENT(V1[4],'4');
  AFF_ELEMENT(V1[5],'5');
  AFF_STRUCT(S, 1, 5);
  AFF_STRUCT2(S, 2,  V1);
  ECRIRE('champ1 = ', STRUCT(S, 1) );
  ECRIRE('champ2 =');
  V2 = STRUCT(S, 2);
  POUR I := 1, 5  ECRIRE( V2[I])  FPOUR
FIN
```

se traduit en PASCAL comme suit :

```
TYPE
  Type1 = INTEGER;
  Type2 = ARRAY[1..5] OF STRING;

  Typestruct = RECORD
    Champ1 : Type1;
    Champ2 : Type2
  END;

VAR  S : Typestruct;

FUNCTION Struct1 ( S:Typestruct ) : Type1;
  BEGIN
    Struct1 :=  S.Champ1;
  END;

PROCEDURE Struct2 ( S:Typestruct ; VAR Result : Type2);
  BEGIN
    Result :=  S.Champ2;
  END;

PROCEDURE Aff_struct1 ( VAR S :Typestruct; VAL : Type1 );
  BEGIN
    S.Champ1 := VAL;
  END;

PROCEDURE Aff_struct2 ( VAR S :Typestruct; VAL : Type2 );
  BEGIN
    S.Champ2 := VAL;
  END;

VAR
  I : INTEGER;
  V1, V2 : Type2;
BEGIN
  V1[1] := '1';    V1[2] :='2';    V1[3] := '3';
```

```
  V1[4] := '4';    V1[5] := '5';
  Aff_struct1(S, 5);
  Aff_struct2(S, V1);
  WRITELN('champ1 = ', Struct1(S) );
  WRITELN('champ2 =');
  Struct2(S, V2);
  FOR I := 1 TO 5 DO
  WRITELN(V2[I])
END.
```

**Implémentation des structures en PASCAL / Dynamique**

```
TYPE
 Type1 =  type du champ1;
 Type2 =  type du champ2;
 ...
 Typen =  type du champn;

 Typestruct = ^type_structure
 type_structure = record
    Champ1 : Type1;
    Champ2 : Type2;
    ....
    ....
    Champ2 : Typen;
  END;

 VAR S : Typestruct;

 PROCEDURE Alloc_struct( VAR S : Typestruct) ;
  BEGIN
    New(s);
  end;

 PROCEDURE Liber_struct( VAR S : Typestruct) ;
  BEGIN
    Dispose(S);
  end;
```

STRUCT (S, I )

Si le type du champ I est un scalaire, STRUCT se traduit par une fonction comme suit :

```
 FUNCTION STRUCTI ( S:Typestruct ) : TypeI;
  BEGIN
    StructI :=  S^.champI;
  END;
```

Il y a donc autant de fonctions STRUCT que de champs scalaires dans la structure.

Si le type du champ I est n'est pas scalaire, c'est à dire un vecteur à une dimension de scalaires, STRUCT se traduit par une procédure comme suit :

```
 PROCEDURE STRUCTI ( S:Typestruct; VAR Result : TypeI);
```

```
  BEGIN
    Result :=  S^.champI;
  END;
```

AFF_STRUCT (S, I )

```
PROCEDURE AFF_STRUCTI ( VAR S :Typestruct; Val : TypeI );
  BEGIN
    S^.champI := Val;
  END;
```

Il y a donc autant de fonction Aff_structi que de champs scalaire dans la structure.

## Implémentation des structures en PASCAL / Exemple

Le Z-algorithme suivant :

```
SOIENT
  S UNE STRUCTURE (ENTIER, VECTEUR(5) DE CHAINES) DYNAMIQUE;
  V1, V2 DES VECTEURS(5) DE CHAINES;
  I UN ENTIER ;
DEBUT
  AFF_ELEMENT(V1[1],'1');
  AFF_ELEMENT(V1[2],'2');
  AFF_ELEMENT(V1[3],'3');
  AFF_ELEMENT(V1[4],'4');
  AFF_ELEMENT(V1[5],'5');
  ALLOC_STRUCT(S);
  AFF_STRUCT(S, 1, 5);
  AFF_STRUCT(S, 2,  V1);
  ECRIRE('champ1 = ', STRUCT(S, 1) );
  ECRIRE('champ2 =');
  V2 = STRUCT(S, 2);
  POUR I := 1, 5  ECRIRE( V2[I])  FPOUR;
  LIBER_STRUCT(s);
FIN
```

se traduit en PASCAL comme suit :

```
TYPE
  Type1 = INTEGER;
  Type2 = ARRAY[1..5] OF STRING;

  Typestruct = ^type_structure;
  Type_structure = RECORD
    Champ1 : Type1;
    Champ2 : Type2
  END;
VAR
   S : Typestruct;
```

```
FUNCTION Struct1 ( S:Typestruct ) : Type1;
  BEGIN
    Struct1 :=  S^.Champ1;
  END;

PROCEDURE Struct2 ( S:Typestruct ; VAR Result : Type2);
  BEGIN
    Result :=  S^.Champ2;
  END;

PROCEDURE Alloc_struct( VAR S : Typestruct) ;
  BEGIN
    New(s);
  END;

PROCEDURE Liber_struct( VAR S : Typestruct) ;
  BEGIN
    Dispose(s);
  END;

PROCEDURE Aff_struct1 ( VAR S :Typestruct; VAL : Type1 );
  BEGIN
    S^.Champ1 := VAL;
  END;

PROCEDURE Aff_struct2 ( VAR S :Typestruct; VAL : Type2 );
  BEGIN
    S^.Champ2 := VAL;
  END;
VAR
  I : INTEGER;
  V1, V2 : Type2;

BEGIN
  V1[1] := '1';    V1[2] :='2';    V1[3] := '3';
  V1[4] := '4';    V1[5] := '5';
  Alloc_struct(s);
  Aff_struct1(S, 5);
  Aff_struct2(S, V1);
  WRITELN('champ1 = ', Struct1(S) );
  WRITELN('champ2 =');
  Struct2(S, V2);
  FOR I := 1 TO 5 DO
    WRITELN(V2[I]);
  Liber_struct(s);
END.
```

## Implémentation des listes linéaires chaînées en PASCAL / Dynamique

```
TYPE
  Typeelem = INTEGER;  { type du champ 'Valeur'  }
```

```
  Pointeur = ^Maillon; { type du champ 'Adresse' }
  Maillon = RECORD
      Val : Typeelem;
      Suiv : Pointeur
  END;

 { Opérations du modèle }

 PROCEDURE Allouer ( VAR P : Pointeur ) ;
   BEGIN  NEW(P)  END;

 PROCEDURE Liberer ( P : Pointeur ) ;
   BEGIN  DISPOSE(P)  END;

 PROCEDURE Aff_val(P : Pointeur; Val : Typeelem );
   BEGIN P^.Val := Val  END;

 FUNCTION Valeur (P : Pointeur) : Typeelem;
   BEGIN   Valeur := P^.Val  END;

 FUNCTION Suivant( P : Pointeur) : Pointeur;
   BEGIN  Suivant := P^.Suiv END;

 PROCEDURE Aff_adr( P, Q : Pointeur ) ;
   BEGIN P^.Suiv := Q    END;
```

## Implémentation des listes linéaires chaînées en PASCAL / Statique

Plusieurs listes dans un même tableau. Le tableau est un ensemble de triplets Element, Suivant, Occupe). Le champ "Occupe" est nécessaire pour les opérations Allouer et Libérer. Une phase d'initialisation est obligatoire avant l'utilisation de ce tableau. Donc le tableau est global. Une liste est définie par l'indice de son premier élément

```
  CONST Max = 100; { Taille arbitraire pour le tableau }

  TYPE Typeqq = INTEGER;
  TYPE Typeliste = RECORD
    Element: Typeqq ;
    Suivant : INTEGER;
    Occupe : BOOLEAN
  END;

  { Le tableau }
  VAR
    Liste : ARRAY[1..Max ] OF Typeliste;

  { initialisation }
  PROCEDURE Init;
    VAR
      I : INTEGER;
    BEGIN
```

```
   FOR I:= 1 TO Max DO
     Liste[I].Occupe := FALSE;
  END;

PROCEDURE Allouer ( VAR I: INTEGER  );
  VAR
    Trouv :BOOLEAN;
  BEGIN
    I:= 1;
    Trouv := FALSE;
    WHILE ( (I <= Max) AND  NOT Trouv ) DO
      IF Liste[I].Occupe
      THEN  I := I + 1
      ELSE  Trouv := TRUE;

    IF  NOT Trouv THEN I := -1;
  END;

PROCEDURE Liberer ( I:INTEGER );
  BEGIN
    Liste[I].Occupe := FALSE ;
  END;

FUNCTION Valeur ( I:INTEGER ) : Typeqq;
  BEGIN
    Valeur := Liste[I].Element;
  END;

FUNCTION Suivant ( I:INTEGER ) : INTEGER;
  BEGIN
    Suivant :=  Liste[I].Suivant  ;
  END;

PROCEDURE Aff_val ( I:INTEGER; Val :Typeqq );
  BEGIN
    Liste[I].Element := Val;
  END;

PROCEDURE Aff_adr (I:INTEGER; J: INTEGER);
  BEGIN
    Liste[I].Suivant := J;
  END ;
```

**Implémentation des listes linéaires chaînées en** PASCAL / Exemple

```
VAR
  E : INTEGER;
BEGIN
  Init;
  Allouer (E);
  IF E <> -1
  THEN
    BEGIN
```

```
      Aff_val (E, 25);
      Aff_adr(E, -1);
    END
  ELSE
    WRITELN('Pas d''espace ');
END.
```

## Implémentation des listes bilatérales en PASCAL / Dynamique

```
TYPE
  Typeelem = INTEGER;  % type du champ 'Valeur'  %
  Pointeur = ^Maillon; % type du champ 'Adresse' %
  Maillon = RECORD
      Val : Typeelem;
      Suiv : Pointeur;
      Prec : Pointeur
  END;

 { Opérations du modèle }

 PROCEDURE Allouer ( VAR P : Pointeur ) ;
  BEGIN  NEW(P)  END;

 PROCEDURE Liberer ( P : Pointeur ) ;
  BEGIN  DISPOSE(P)  END;

 PROCEDURE Aff_val(P : Pointeur; Val : Typeelem );
  BEGIN P^.Val := Val  END;

 FUNCTION Valeur (P : Pointeur) : Typeelem;
  BEGIN   Valeur := P^.Val  END;

 FUNCTION Suivant( P : Pointeur) : Pointeur;
  BEGIN  Suivant := P^.Suiv END;

 FUNCTION Precedent( P : Pointeur) : Pointeur;
  BEGIN  Precedent := P^.Prec END;

 PROCEDURE Aff_adrd( P, Q : Pointeur ) ;
  BEGIN P^.Suiv := Q    END;

 PROCEDURE Aff_adrg( P, Q : Pointeur ) ;
  BEGIN P^.Prec := Q    END;
```

## Implémentation des listes bilatérales en PASCAL /  Statique

Plusieurs listes dans un même tableau. Le tableau est un ensemble de quadruplé (Element, Suivant, Precedent, Occupe). Le champ "Occupe" est nécessaire pour les opérations Allouer et Libérer.  Une phase d'initialisation est obligatoire avant l'utilisation de ce tableau. Donc le tableau est global. Une liste est définie par l'indice de son premier élément.

```
 CONST Max = 100; { Taille arbitraire du tableau }
```

```
TYPE Typeqq = INTEGER;
TYPE Typelistebi = RECORD
  Element: Typeqq ;
  Suivant : INTEGER;
  Precedent : INTEGER;
  Occupe : BOOLEAN
END;

{ Le tableau }
VAR
  Listebi : ARRAY[1..Max ] OF Typelistebi;

{ Initialisation }
PROCEDURE Init;
  VAR
    I : INTEGER;
  BEGIN
    FOR I:= 1 TO Max DO
      Listebi[I].Occupe := FALSE;
  END;

PROCEDURE Allouer ( VAR I: INTEGER  );
  VAR
    Trouv :BOOLEAN;
  BEGIN
    I:= 1;
    Trouv := FALSE;
    WHILE ( (I <= Max) AND  NOT Trouv ) DO
      IF Listebi[I].Occupe
      THEN  I := I + 1
      ELSE  Trouv := TRUE;

    IF  NOT Trouv THEN I := -1;
  END;

PROCEDURE Liberer ( I:INTEGER );
  BEGIN
    Listebi[I].Occupe := FALSE ;
  END;

FUNCTION Valeur ( I:INTEGER ) : Typeqq;
  BEGIN
    Valeur := Listebi[I].Element;
  END;

FUNCTION Suivant ( I:INTEGER ) : INTEGER;
  BEGIN
    Suivant :=  Listebi[I].Suivant  ;
  END;

FUNCTION Precedent ( I:INTEGER ) : INTEGER;
  BEGIN
```

```
    Precedent :=  Listebi[I].Precedent ;
  END;

PROCEDURE Aff_val ( I:INTEGER; Val :Typeqq );
  BEGIN
    Listebi[I].Element := Val;
  END;

 PROCEDURE Aff_adrd (I:INTEGER; J: INTEGER);
  BEGIN
    Listebi[I].Suivant := J;
  END;
 PROCEDURE Aff_adrg (I:INTEGER; J: INTEGER);
  BEGIN
    Listebi[I].Precedent := J;
  END;
```

## Implémentation des listes bilatérales en PASCAL / Exemple

```
 VAR
  E : INTEGER;
 BEGIN
  Init;
  Allouer (E);
  IF E <> -1
  THEN
    BEGIN
      Aff_val (E, 25);
      Aff_adrg(E, -1);
      Aff_adrd(E, -1);
    END
  ELSE
    WRITELN('Pas d"espace ');
 END.
```

## Implémentation des arbres de recherche binaire en PASCAL / Dynamique

```
 TYPE
  T = ^Noeud;
  Noeud = RECORD
    Element : INTEGER;
    Fg, Fd, Pere  : 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;
```

```
FUNCTION Pere( P : T) : T;
  BEGIN  Pere := P^.Pere  END;

PROCEDURE Aff_info ( 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;

PROCEDURE Aff_pere( VAR P : T; Q : T);
  BEGIN   P^.pere :=  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;

PROCEDURE Liberernoeud( P : T);
  BEGIN
    DISPOSE ( P )
  END;
```

## Implémentation des arbres de recherche binaire  en PASCAL /  Statique

Plusieurs arbres de recherche binaire dans un même tableau. Le tableau est un ensemble de 5-uplets (Info, Fg, Fd, Pere, Occupe). Le champ "Occupe" est nécessaire pour les opérations Creernoeud et Libérernoeud. Une phase d'initialisation est obligatoire avant l'utilisation de ce tableau. Donc le tableau est global. Un arbre de recherche binaire est défini par l'indice de son premier élément.

```
CONST Max = 100; { Taille arbitraire du tableau }
TYPE Typeqq = INTEGER;

TYPE  Typearb = RECORD
  Info : Typeqq ;
  Fg, Fd, Pere : INTEGER;
  Occupe : BOOLEAN;
END;

{ Le tableau }
VAR
  Arb : ARRAY[1..Max ] OF Typearb;

{ Initialisation }
```

```
PROCEDURE  Init;
  VAR
    I : INTEGER;
  BEGIN
    FOR I :=1 TO Max DO
      Arb[I].Occupe := FALSE;
  END;

PROCEDURE Creernoeud ( VAR I : INTEGER );
  VAR
    Trouv : BOOLEAN;
  BEGIN
    I := 0;
    Trouv := FALSE;
    WHILE ( (I <= Max) AND NOT Trouv ) DO
      IF ( Arb[I].Occupe )
      THEN I := I + 1
      ELSE Trouv := TRUE;

    IF  NOT Trouv THEN I := -1;
  END;

PROCEDURE Liberernoeud ( I: INTEGER );
  BEGIN
    Arb[I].Occupe := FALSE ;
  END;

FUNCTION Info ( I:INTEGER ) : Typeqq;
  BEGIN
    Info :=  Arb[I].Info
  END;

FUNCTION Fd (  I: INTEGER ) : INTEGER;
  BEGIN
    Fd := Arb[I].Fd
  END;

FUNCTION Fg (  I: INTEGER ) : INTEGER;
  BEGIN
    Fg := Arb[I].Fg
  END;

FUNCTION Pere (  I: INTEGER ) : INTEGER;
  BEGIN
Pere := Arb[I].Pere
  END;

PROCEDURE Aff_info ( I:INTEGER; Val : Typeqq );
  BEGIN
    Arb[I].Info := Val;
  END;

PROCEDURE  Aff_fd ( I:INTEGER; J : INTEGER);
```

```
  BEGIN
    Arb[I].Fd := J;
  END;

 PROCEDURE  Aff_fg ( I:INTEGER; J : INTEGER);
  BEGIN
    Arb[I].Fg := J;
  END;

 PROCEDURE  Aff_pere ( I:INTEGER; J : INTEGER);
  BEGIN
    Arb[I].Pere := J;
  END;
```

**Implémentation des arbres de recherche binaire en PASCAL /  Exemple**

```
 VAR
   E : INTEGER;
 BEGIN
   Init;
   Creernoeud (E);
   IF ( E <> -1 )
   THEN
     BEGIN
       Aff_info (E, 25);
       Aff_fg(E, -1);
       Aff_fd(E, -1);
     END
   ELSE
     WRITELN('Pas d"espace');
 END.
```

**Implémentation des arbres de recherche m-aire en PASCAL / Dynamique**

```
 TYPE
   T = ^Noeud;
   Noeud = RECORD
     Infor : ARRAY[1..Max] of INTEGER;
     Fils : ARRAY[1..Max] of T;
     Degre : Byte ;
     Pere : T
   END;

 FUNCTION Infor(P : T; I: INTEGER) : INTEGER;
   BEGIN  Infor := P^.Infor[I]   END;

 FUNCTION Fils( P : T; I : INTEGER) : T;
   BEGIN  Fils := P^.Fils[I]  END;

 FUNCTION Pere( P : T) : T;
   BEGIN  Pere := P^.Pere  END;

 PROCEDURE Aff_infor ( VAR P : T; I:INTEGER; Val : INTEGER);
```

```
  BEGIN   P^.Infor[I] := Val  END;

PROCEDURE Aff_fils( VAR P : T; I:INTEGER; Q : T);
  BEGIN   P^.Fils[I] :=  Q  END;

PROCEDURE Aff_pere( VAR P : T; Q : T);
  BEGIN   P^.pere :=  Q  END;

FUNCTION Creernoeud(  Val : INTEGER) : T;
  VAR
    P : T;
    I : BYTE;
  BEGIN
    NEW ( P ) ;
    Creernoeud := P ;
    For I:=1 TO Max Do P^.Fils[I] := NIL;
    P.degre := 0
  END;

FUNCTION Degre ( P : T ) : BYTE;
  BEGIN
    Degre := P^.Degre
  END

PROCEDURE Aff_Degre ( VAR P : T; N : BYTE);
  BEGIN
    P^.Degre := N
  END;

PROCEDURE Liberernoeud( P : T);
  BEGIN    DISPOSE ( P )   END;
```

## Implémentation des arbres de recherche m-aire en PASCAL / Statique

Plusieurs arbres de recherche m-aire dans un même tableau.Le tableau est un ensemble de quadruplé (Info, Fils, Degre, Occupe).Info est un tableau de (Ordre-1) valeurs.Fils est un tableau de (Ordre) indices. Le champ Degre contient le nombre courant de valeurs dans le noeud. Le champ "Occupe" est nécessaire pour les opéations Creernoeud et Libérernoeud.Une phase d'initialisation est obligatoire avant l'utilisation de ce tableau. Donc le tableau est global.Un arbre de recherche m-aire est défini par l'indice de son premier élément.

```
  CONST Max = 100; { Taille arbitraire du tableau }
  CONST Ordre = 8; { Ordre arbitraire }
  TYPE Typeqq= INTEGER;
  TYPE Typearm = RECORD
    Fils : ARRAY[1..Ordre] OF INTEGER;
    Info : ARRAY[1..Ordre-1] OF Typeqq;
    Degre : BYTE;
    Occupe : BOOLEAN;
  END;

  { Le tableau }
  VAR
```

```
  Arm : ARRAY [1.. Max ] OF Typearm;

 { Initialisation }
 PROCEDURE Init;
  VAR I : INTEGER;
  BEGIN
    FOR I:=1 TO Max DO
      Arm[I].Occupe := FALSE;
  END;

 PROCEDURE Creernoeud ( VAR P : INTEGER );
  VAR Trouv : BOOLEAN;
  BEGIN
    P := 1;
    Trouv := FALSE;
    WHILE ( (P <= Max) AND NOT Trouv ) DO
      IF  Arm[P].Occupe
      THEN P := P+ 1
      ELSE Trouv := TRUE;

    IF NOT Trouv THEN P := -1;
  END;

 PROCEDURE Liberernoeud ( P: INTEGER );
  BEGIN
    Arm[P].Occupe := FALSE ;
  END;

 FUNCTION  Infor ( P:INTEGER;I:INTEGER ) : Typeqq;
  BEGIN
    Infor :=  Arm[P].Info[I] ;
  END;

 FUNCTION Fils (  P, I : INTEGER ) : INTEGER;
  BEGIN
    Fils :=  Arm[P].Fils[I]  ;
  END;

 PROCEDURE Aff_infor ( P, I : INTEGER; Val :Typeqq );
  BEGIN
    Arm[P].Info[I] := Val;
  END;

 PROCEDURE Aff_fils (  P, I, J : INTEGER);
  BEGIN
            Arm[P].Fils[I] := J;

END;

 FUNCTION Degre ( P: INTEGER ) : BYTE;
  BEGIN
    Degre :=  Arm[P].Degre ;
  END;
```

```
PROCEDURE Aff_degre ( P : INTEGER; I : BYTE );
  BEGIN
    Arm[P].Degre := I ;
  END;
```

## Implémentation des arbres de recherche m-aire en PASCAL / Exemple

```
VAR
  E : INTEGER;
BEGIN
  Init;
  Creernoeud (E);
  IF ( E <> -1 )
  THEN
    BEGIN
      Aff_infor (E, 1, 25);
      Aff_fils(E, 2, -1);
    END
  ELSE
    WRITELN('Pas D''Espace ');
END.
```

## Implémentation des piles en PASCAL / Dynamique

```
TYPE
  Typeqq = INTEGER;
  Typepile = ^Maillon;
  Maillon = RECORD
    Valeur : Typeqq;
    Suivant : Typepile
  END;

PROCEDURE Creerpile( VAR P : Typepile );
  BEGIN
    P := NIL;
  END;

FUNCTION Pilevide ( P : Typepile ) : BOOLEAN;
  BEGIN
    Pilevide := ( P = NIL )
  END;

PROCEDURE Empiler ( VAR P : Typepile; Val : Typeqq );
  VAR
    Q : Typepile;
  BEGIN
    NEW(Q);
    Q^.Valeur := Val;
    Q^.Suivant := P;
    P := Q;
```

```
  END;

PROCEDURE Depiler ( VAR P : Typepile; VAR V :Typeqq );
  VAR Sauv : Typepile;
  BEGIN
   IF NOT Pilevide (P)
   THEN
     BEGIN
       V := P^.Valeur;
       Sauv := P;
       P := P^.Suivant;
       DISPOSE(Sauv);
     END
   ELSE WRITELN('Pile Vide');
            END;
```

**Implémentation des piles en PASCAL / Statique**

```
TYPE
  Typepile = RECORD
    Som : INTEGER;
    Tab : ARRAY(.1..50.) OF T ;
  END;

PROCEDURE Creerpile ( VAR P : Typepile);
  BEGIN P.Som := 0 END;

FUNCTION Pilevide( P : Typepile) : BOOLEAN;
  BEGIN Pilevide := (P.Som = 0) END;

PROCEDURE Empiler (VAR P : Typepile ; Val : T);
  BEGIN
    IF NOT Pilepleine(P)
    THEN
      BEGIN
        P.Som := P.Som + 1 ;
        P.Tab(.P.Som.) := Val ;
      END
    ELSE
      BEGIN
        WRITELN(' Pile saturée');
        HALT;
      END
  END;

PROCEDURE Depiler (VAR P: Typepile; VAR Val  : T);
  BEGIN
    IF NOT Pilevide(P)
    THEN
      BEGIN
        Val := P.Tab(.P.Som.);
        P.Som := P.Som - 1
      END
```

```
    ELSE
      BEGIN
        WRITELN(' Pile saturée');
        HALT;
      END
  END;
```

### Implémentation des piles en PASCAL / Exemple

```
 VAR
   Pile : Typepile;
   V : Typeqq;
 BEGIN
   Creerpile(Pile);
   Empiler (Pile, 25);
   Empiler (Pile, 35);
   Empiler (Pile, 45);
   Depiler (Pile, V );
   WRITELN(V);
END.
```

### Implémentation des files d'attente en PASCAL / Dynamique

```
 TYPE
   Typelement=INTEGER;
   T1 = ^Elm ;
   Elm = RECORD
     Val : Typelement;
     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 : Typelement );
   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 : Typelement );
 BEGIN
  IF NOT Filevide(Fil)
  THEN
   BEGIN
    Val := Fil.Tete^.Val;
    Fil.Tete := Fil.Tete^.Suiv;
   END
  ELSE WRITELN(' File Vide ');
 END;
```

## Implémentation des files d'attente en PASCAL / Statique

```
CONST Max = 100;
TYPE Typeqq= INTEGER;
TYPE Typefile = RECORD
 Elements : ARRAY[1..Max] OF Typeqq;
 Tete, Queue : INTEGER
END;

PROCEDURE Creerfile ( VAR F :Typefile );
 BEGIN
  F.Tete := Max ;
  F.Queue := Max ;
 END;

FUNCTION Filevide ( F : Typefile  ) : BOOLEAN;
 BEGIN
  Filevide := ( F.Tete = F.Queue );
 END;

FUNCTION Filepleine ( F : Typefile  ) : BOOLEAN;
 BEGIN
  Filepleine := ( F.Tete = F.Queue MOD Max + 1 );
 END;

PROCEDURE Enfiler ( VAR F : Typefile ; Val : Typeqq  );
 BEGIN
  IF  NOT Filepleine(F)
  THEN
   BEGIN
    F.Queue := F.Queue MOD Max  + 1;
    F.Elements[F.Queue] := Val;
   END
  ELSE
   WRITELN('File Pleine');
 END;

PROCEDURE Defiler ( VAR F : Typefile ; VAR Val :Typeqq  );
 BEGIN
  IF NOT Filevide(F)
```

```
  THEN
    BEGIN
      F.Tete := F.Tete MOD Max + 1;
      Val := F.Elements[F.Tete];
    END
  ELSE
    WRITELN('File Vide');
END;
```

## Implémentation des files d'attente en PASCAL / Exemple

```
VAR F : Typefile;
V : Typeqq ;

BEGIN
  Creerfile (F);
  Enfiler(F, 5);
  Enfiler(F, 18);
  Enfiler(F, 22);
  Defiler(F, V);
  WRITELN(V);
END.
```

## Implémentation des fichiers en PASCAL

```
TYPE
  Type1 = Type du champ1 de la structure du bloc (ou article);
  Type2 = Type du champ2 de la structure du bloc (ou article);
  ...
  Typen = Type du champn de la structure du bloc (ou article);

  Typecaract1 = Type de la première caractéristique du fichier;
  Typecaract2 = Type de la deuxième caractéristique du fichier;
  ...
  Typecaractm = Type de la m-ième caractéristique du fichier;

  { Définition d'un bloc du fichier }
  Sorte = (Caract, Art );
  Typestruct = RECORD
    CASE Id : Sorte OF
      Caract : (
               Champ1 : Type1;
               Champ2 : Type2;
               ...
               Champn : Typen;
              );
      Art :
             (
              Caract1 : Typecaract1;
              Caract2 : Typecaract2;
              ...
```

```
            Caractm : Typecaractm;
           )
    END;
  Typefile =   File OF Typestruct;

VAR
  F  : Typefile;            { Fichier }
  Buf_caract : Typestruct;   { Buffer des caractéristiques }

 { Machine abstraite sur les fichiers }

PROCEDURE Ouvrir (VAR Fl : Typefile ; Fp, Mode : STRING );
  BEGIN
    ASSIGN(Fl, Fp);
    IF Mode = 'A'
    THEN
      BEGIN
        RESET(Fl);
        READ(Fl, Buf_caract);
      END
    ELSE
      BEGIN
        REWRITE(Fl);
        Buf_caract.Id := Caract;
        WRITE(Fl, Buf_caract)
      END;
  END;

PROCEDURE Fermer ( VAR Fl : Typefile);
  BEGIN
    Buf_caract.Id := Caract;
    SEEK(Fl, 0);
    WRITE(Fl, Buf_caract);
    CLOSE( Fl)
  END;

   ENTETE  (Fl, i )

FUNCTION Entete1(  VAR Fl : Typefile): Typecaract1;
  BEGIN
    Entete1 := Buf_caract.Caract1;
  END;
```

Il y a donc autant de fonctions ENTETEi que de types scalaires définis dans la partie 'ENTETE'.

```
   AFF_ENTETE  ( Fl, i, Exp )

PROCEDURE Aff_entete1 ( VAR Fl: Typefile; VAL : Typecaract1);
  BEGIN
    Buf_caract.Caract1 := VAL
  END;
```

Il y a donc autant de fonctions AFF_ENTETEi que de types scalaires définis dans la partie 'ENTETE'.

```
PROCEDURE Ecrireseq ( VAR Fl: Typefile;  Buf : Typestruct );
 BEGIN
   Buf.Id := Art;
   WRITE(Fl, Buf)
 END;

PROCEDURE Ecriredir ( VAR Fl: Typefile;  Buf : Typestruct;  N: INTEGER );
 BEGIN
   Buf.Id := Art;
   SEEK(Fl, N);
   WRITE(Fl, Buf)
 END;

PROCEDURE Lireseq ( VAR Fl: Typefile; VAR Buf : Typestruct );
 BEGIN
   READ(Fl, Buf)
 END;

PROCEDURE Liredir ( VAR Fl: Typefile; VAR Buf : Typestruct;  N: INTEGER );
 BEGIN
   SEEK(Fl, N);
   READ(Fl, Buf)
 END;

FUNCTION Finfich ( VAR Fl : Typefile): BOOLEAN;
 BEGIN
   Finfich := EOF(Fl)
 END;

FUNCTION Alloc_bloc ( VAR Fl : Typefile) : INTEGER;
 BEGIN
   Alloc_bloc := FILESIZE(Fl) ;
 END;
```

## Implémentation des fichiers en PASCAL / Exemple

Le Z-algorithme suivant :

```
SOIENT
 F UN FICHIER DE ( ENTIER , VECTEUR ( 5 ) DE CHAINES )
 BUFFER B1 ENTETE ( ENTIER , ENTIER ) ;
  { fichier de blocs contenant le nombre d'articles et un tableau d'articles}
  { entete : nombre d'articles, nombre de blocs}
 Creer , Imprimer DES ACTIONS ;
DEBUT
 APPEL Creer ;
 APPEL Imprimer ;
FIN

/*****      Chargement de n articles avec un chargement _ 100% ****/
```

```
ACTION Creer ;
SOIENT
 I , K , N , Nbblocs DES ENTIERS ;
DEBUT
 OUVRIR ( F , 'f.pas' , 'N' ) ;
 I := 0 ;
 Nbblocs := 0 ;
 N := 500 ;
 AFF_ENTETE ( F , 1 , N ) ;
 TQ I < N :
   K := 0 ;
   TQ ( K < 5 ) ET ( I < N )
     K := K + 1 ;
     I := I + 1 ;
     AFF_ELEMENT ( STRUCT ( B1 , 2 ) [ K ] , ALEACHAINE )
   FTQ ;
   AFF_STRUCT ( B1 , 1 , K ) ;
   Nbblocs := Nbblocs + 1 ;
   ECRIRESEQ ( F , B1 ) ;
 FTQ ;
 AFF_ENTETE ( F , 2 , Nbblocs ) ;
 FERMER ( F ) ;
FIN

/*****       Impression des articles du fichier ****/
ACTION Imprimer ;
SOIENT
 I , K DES ENTIERS ;
DEBUT
 I := 0 ;
 OUVRIR ( F , 'f.pas' , 'A' ) ;
 TQ NON FINFICH ( F )
   I := I + 1 ;
   ECRIRE ( 'B  L  O  C     n° ' , I ) ;
   LIRESEQ ( F , B1 ) ;
   POUR K := 1 , STRUCT ( B1 , 1 )
     ECRIRE ( ELEMENT ( STRUCT ( B1 , 2 ) [ K ] ) )
   FPOUR
 FTQ;
 FERMER(f);
FIN
```

se traduit en PASCAL Comme suit :

```
 TYPE
   Type1 = INTEGER;
   Type2 = ARRAY[1..5] OF STRING;
   Typecaract1 = INTEGER;
   Typecaract2 = INTEGER;
   Sorte = (Caract, Art );


Typestruct = RECORD
```

```
  CASE Id : Sorte OF
    Caract : (
              Champ1 : Type1;
              Champ2 : Type2;
             );
    Art :
            (
             Caract1 : Typecaract1;
             Caract2 : Typecaract2
            )
  END;

 Typefile =   File OF Typestruct;

VAR
 F  : Typefile;
 { Fichier de blocs contenant le nombre d'articles et un tableau d'articles}
 B1 : Typestruct;
 Buf_caract : Typestruct;

 { Machine abstraite sur les vecteurs }
 FUNCTION Element ( VAR V:Type2; I: INTEGER ) : STRING;
  BEGIN
    Element :=  V[I];
  END;

 PROCEDURE Aff_element ( VAR V :Type2; I:INTEGER; VAL : STRING);
  BEGIN
    V[I] := VAL;
  END;

 { Machine abstaite sur les structures }
 FUNCTION Struct1(  S : Typestruct): Type1;
  BEGIN
    Struct1 := S.Champ1;
  END;

 PROCEDURE Struct2(S: Typestruct; VAR Result : Type2);
  BEGIN
    Result := S.Champ2;
  END;

 PROCEDURE Aff_struct1( VAR S : Typestruct; VAL : Type1);
  BEGIN
    S.Champ1 := VAL;
  END;

 PROCEDURE Aff_struct2( VAR S : Typestruct; VAL : Type2);
  BEGIN   S.Champ2 := VAL;       END;

 { Machine abstraite sur les fichiers }
 PROCEDURE Ouvrir (VAR Fl : Typefile ; Fp, Mode : STRING );
  BEGIN
```

```
   ASSIGN(Fl, Fp);
   IF Mode = 'A'
   THEN
     BEGIN
       RESET(Fl);
       READ(Fl, Buf_caract);
     END
   ELSE
      BEGIN
        REWRITE(Fl);
        Buf_caract.Id := Caract;
        WRITE(Fl, Buf_caract)
      END;
 END;


PROCEDURE Fermer ( VAR Fl : Typefile);
 BEGIN
   Buf_caract.Id := Caract;
   SEEK(Fl, 0);
   WRITE(Fl, Buf_caract);
   CLOSE( Fl)
 END;

FUNCTION Entete1(  VAR Fl : Typefile): Typecaract1;
 BEGIN
   Entete1 := Buf_caract.Caract1;
 END;

FUNCTION Entete2(  VAR Fl : Typefile): Typecaract2;
 BEGIN
   Entete2 := Buf_caract.Caract2;
 END;

PROCEDURE Aff_entete1 ( VAR Fl: Typefile; VAL : Typecaract1);
 BEGIN
   Buf_caract.Caract1 := VAL
 END;

PROCEDURE Aff_entete2 ( VAR Fl: Typefile; VAL : Typecaract2);
 BEGIN
   Buf_caract.Caract2 := VAL
 END;

PROCEDURE Ecrireseq ( VAR Fl: Typefile;  Buf : Typestruct );
 BEGIN
   Buf.Id := Art;
   WRITE(Fl, Buf)
 END;

PROCEDURE Ecriredir ( VAR Fl: Typefile;  Buf : Typestruct;  N: INTEGER );
 BEGIN
   Buf.Id := Art;
```

```
   SEEK(Fl, N);
   WRITE(Fl, Buf)
  END;

PROCEDURE Lireseq ( VAR Fl: Typefile; VAR Buf : Typestruct );
  BEGIN
   READ(Fl, Buf)
  END;

PROCEDURE Liredir ( VAR Fl: Typefile; VAR Buf : Typestruct;  N: INTEGER );
  BEGIN
   SEEK(Fl, N);
   READ(Fl, Buf)
  END;

FUNCTION Finfich ( VAR Fl : Typefile): BOOLEAN;
  BEGIN
   Finfich := EOF(Fl)
  END;

FUNCTION Aleachaine : STRING;
  VAR
   K : BYTE;
   Chaine : STRING;
  BEGIN
   Chaine := '';
   FOR K:=1 TO 4 DO
    CASE Random(2) OF
      0 : Chaine := Chaine + CHR(97+Random(26) ) ;
      1 : Chaine := Chaine + CHR(65+Random(26) )
    END;
   Aleachaine := Chaine;
  END;

FUNCTION Alloc_bloc ( VAR Fl : Typefile) : INTEGER;
  BEGIN
     Alloc_bloc := FILESIZE(Fl) ;
  END;

/*****    Chargement de 50 blocs à raison de 60% ****/
PROCEDURE Creer ;
  VAR
   I , K  : INTEGER ;
   V : Type2;
  BEGIN
   Ouvrir ( F , 'f.pas' , 'N' ) ;
   FOR I := 1 TO  10 DO
    BEGIN
      FOR  K := 1 TO 3 DO
       Aff_element ( V , K  , ALEACHAINE );
      Aff_struct2( B1, V);
      Aff_struct1 ( B1 , 3 ) ;
      Ecrireseq ( F , B1 ) ;
```

```
      END;
    Fermer ( F ) ;
  END;

 /*****    Impression des articles du fichier ****/
 PROCEDURE Imprimer ;
  VAR
    K  : INTEGER ;
    V : Type2;
  BEGIN
    Ouvrir ( F , 'f.pas' , 'A' ) ;
    WHILE NOT Finfich ( F ) DO
      BEGIN
        Lireseq ( F , B1 ) ;
        Struct2(B1, V);
        FOR  K := 1 TO  Struct1 ( B1 ) DO
          WRITELN ( Element ( V , K  ) );
        READLN;
      END;
  END;

BEGIN
  Creer ;
  Imprimer ;
END.
```

## 8. Passage de Z vers C

### Déclaration des variables

Une déclaration de variables en C se fait par
Type <Li>;
où <Li> désigne une liste d'identificateurs.

Les objets simples

Equivalents des objets Z --> C

**ENTIER** se traduit par INT.
**CAR** se traduit par CHAR.
Au type "**CHAINE**" on associe le type construit Chaine que l'on définit par
typedef char Chaine[256]

En C le type Booléen n'existe pas.
Pour continuer à travailler toujours avec ce type, il suffit de rajouter au début du programme

typdef int Booleen

et de définir les valeurs True et False comme suit :

```
#define true 1
#define false 0
```

Les objet "structures"

Pour définir une structure S en C il faut choisir une implémentation.
Implémenter, c'est choisir une représentation mémoire ( généralement statique ou dynamique ) et traduire les opérations de la machine abstraite dans cette représentation.

Il suffit de remplacer la structure S par Pointeur et rajouter au niveau de l'en-tête du programme C l'implémentation désirée où l'on définira le type Pointeur.

### Boucle "TANTQUE"

```
----------Z----------
TANTQUE  Cond :
    Instructions
FINTANTQUE
```

se traduit par :

```
-------------C-----------
WHILE ( Cond )
  {
   Instructions
  }
```

### Boucle "POUR"

```
----------Z----------
POUR V:= Exp1, Exp2 [, Exp3] [:]
    Instructions
FINPOUR
```

se traduit par ‖

```
----------C----------
for (V=Exp1; V<=Exp2; V=V+Exp3)
  {
    Instructions
  }
```

### L'alternative "SI"

```
----------Z----------
SI Cond :
    Instructions
[SINON
    Instruction  ]
FSI
```

se traduit par :

```
----------C----------
IF ( Cond )
 {
  Instructions
 }
[ELSE
 {Instructions } ]
```

### Z-expressions

La grammaire des Z-expressions est incluse dans la grammaire C.

### Lecture

```
--------Z--------
LIRE(V1, V2, ...)
```

se traduit par :

```
----------C----------
scanf("...", &V1, &V2, ...)
```

### Ecriture

```
--------Z----------
ECRIRE(E1, E2, ...)
```

se traduit par :

```
--------C---------
printf(E1, E2, ...)
```

### Affectation

Même syntaxe avec le '=' à la place de ':='.

### Action composée

```
------------Z-------------
ACTION Nom ( P1, P2, ...);
SOIENT
     Définition des objets locaux
      et des paramètres
DEBUT
     Instructions
FIN
```

se traduit par :

```
-----------------C----------------
void Nom ( typ1 P1 , typ2 P2, ...)
  {
     Définition des objets locaux
     Instructions
  }
```

### Fonctions

```
------------------Z------------------
FONCTION  Nom (  P1, P2, ...) : type;
SOIENT
     Définition des objets locaux
      et des paramètres
DEBUT
     Instructions
FIN
```

se traduit par

```
------------------C------------------
type Nom ( typ1 P1, typ2 P2, ...)
  {
     Définition des objets locaux
     Instructions
  }
```

**Algorithme**

----------Z----------
**SOIENT**

Objets locaux et globaux
Annonce des modules

**DEBUT**

Instructions

**FIN**

Module 1
Module 2
...
Modules n

se traduit par :

----------C----------
Objets locaux et globaux

Définition des modules
Module 1
Module 2
...
Module n

```
main()
{
    Instructions
}
```

## 9. Implémentation des machines Z en C

### Implémentation des vecteurs en C / Statique

```
typedef % type d'un élément du tableau % Typeqq;
typedef Typeqq Typevect[10];

   ELEMENT ( V, I )

int Element (Typevect V , int I )
  {
   return ( V[I] );
  }

   AFF_ELEMENT ( V, I, Val )

void Aff_element (Typevect  V,  int  I, int Val )
  {
   V[I] = Val;
  }
```

### Implémentation des vecteurs en C / Exemple

```
int I;
Typevect V ;

main()
  {
   Aff_element(V, 0, 189);
   Aff_element(V, 1, 34);
   Aff_element(V, 2, 56);
   Aff_element(V, 3, 89);
   Aff_element(V, 4, 38);
   Aff_element(V, 5, 156);

   for (I= 0; I< 6; I++)
      printf("%d\n",Element( V, I) );

   Aff_element(V,3, 99);
   for (I= 0; I< 6; I++)
     printf("%d\n",Element( V, I) );
  }
```

### Implémentation des vecteurs en C / Dynamique

```
typedef % type d'un élément du tableau %    Typeqq    ;
typedef Typeqq Typevect[10] ;
typedef Typevect  *Typevect_dyn ;
```

```
  ALLOC_TAB ( T )

void Alloc_tab( Typevect_dyn *T)
 {
  *T = (Typevect_dyn) malloc ( sizeof ( Typevect) );
 }

  LIBER_TAB ( T  )

void Liber_tab( Typevect_dyn T)
 {
  free (T);
 }

  ELEMENT ( V, I )

int Element (Typevect_dyn T , int I )
 {
  return ( *T[I] );
 }

  AFF_ELEMENT ( V, I, Val )

void Aff_element ( Typevect_dyn T,  int  I, int Val )
 {
  *T[I] = Val;
 };
```

**Implémentation des vecteurs / Exemple**

```
#include <alloc.h>

int I;
Typevect_dyn V ;

main()
 {
  Alloc_tab(&V);
  Aff_element(V, 0, 189);
  Aff_element(V, 1, 34);
  Aff_element(V, 2, 56);
  Aff_element(V, 3, 89);
  Aff_element(V, 4, 38);
  Aff_element(V, 5, 156);

  for (I= 0; I< 6; I++)
      printf("%d\n",Element( V, I) );

  Aff_element(V,3, 99);
  for (I= 0; I< 6; I++)
   printf("%d\n",Element( V, I) );
```

```
  Liber_tab(V);
 }
```

## Implémentation des structures en C / Statique

```
typedef %type du premier champs% Type1;
typedef %type du deuxième champs% Type2;
...
typedef %type du n-ième champs% Typen;

struct Typestruct
 {
  Type1  Champ1;
  Type2  Champ2 ;
 }  ;

struct Typestruct S;
```

STRUCT (S, I )

Si le type du champ I est un scalaire, STRUCT se traduit par une fonction comme suit :

```
Typei Structi ( struct Typestruct S )
 {
  return(S.Champi) ;
 }
```

Il ya donc autant de fonctions STRUCT que de champs scalaires dans la structure.

Si le type du champ I est n'est pas scalaire, c'est à dire un vecteur à une dimension de scalaires, STRUCT se traduit par une procédure comme suit :

```
void Struct2 ( struct Typestruct S , Type2 Result)
 {
  int K;
  for (K=0; K<5; K++)
    Result[K] = S.Champ2[K] ;
 }
```

S'il s'agit d'un tableau de chaînes l'affectation se fait par la fonction "strcpy".

AFF_STRUCT (S, I, Exp)

```
void Aff_struct1 ( struct Typestruct *S,  Type1 Val )
 {
  struct Typestruct P;
  P = *S;
  P.Champ1 = Val;
  *S = P;
 }
```

Il y a donc autant de fonctions Aff_struct que de champs scalaire dans la structure.

## Implémentation des structures en C / Exemple

Le Z-algorithme suivant :

```
SOIENT
  S UNE STRUCTURE (ENTIER, VECTEUR(5) DE CHAINES);
  V1, V2 DES VECTEURS(5) DE CHAINES;
  I UN ENTIER ;
DEBUT
  AFF_ELEMENT(V1[1],'1');
  AFF_ELEMENT(V1[2],'2');
  AFF_ELEMENT(V1[3],'3');
  AFF_ELEMENT(V1[4],'4');
  AFF_ELEMENT(V1[5],'5');
  AFF_STRUCT(S, 1, 5);
  AFF_STRUCT(S, 2,  V1);
  ECRIRE('champ1 = ', STRUCT(S, 1) );
  ECRIRE('champ2 =');
  V2 = STRUCT(S, 2);
  POUR I := 1, 5  ECRIRE( V2[I])  FPOUR
FIN
```

se traduit en C comme suit :

```
typedef int Type1  ;
typedef char String[255];
typedef String  Type2 [5]  ;

struct Typestruct
  {
   Type1  Champ1;
   Type2  Champ2 ;
  }  ;

struct Typestruct S;
int I ;
Type2 V1, V2 ;

Type1 Struct1 ( struct Typestruct S )
   { return(S.Champ1) ;}


void Struct2 ( struct Typestruct S , Type2 Result)
  {
   int K;
   for (K=0; K<5; K++)
   strcpy(Result[K], S.Champ2[K] );
  }

void Aff_struct1 ( struct Typestruct *S,  Type1 Val )
  {
   struct Typestruct P;
   P = *S;
```

```
    P.Champ1 = Val;
    *S = P;
   }


void Aff_struct2 ( struct Typestruct *S,  Type2 Val )
   {
    int K;
    struct Typestruct P;
    P = *S;
    for (K=0; K<5; K++)
      strcpy(P.Champ2[K], Val[K]);
    *S = P;
   }

 main()
  {
   strcpy(V1[0], "1");
   strcpy(V1[1], "2");
   strcpy(V1[2], "3");
   strcpy(V1[3], "4");
   strcpy(V1[4], "5");
   Aff_struct1(&S, 5);
   Aff_struct2(&S, V1);
   printf("champ1 = %d \n", Struct1(S) );
   printf("champ2 = \n");
   Struct2(S, V2);
   for ( I= 0; I<5; I++) printf("%s \n",V2[I])  ;
  }
```

## Implémentation des structures en C / Dynamique

```
 typedef type du premier champs Type1;
 typedef type du deuxième champs Type2;
 ...
 typedef type du n-ième champs Typen;

 struct Typestruct
  {
    Type1  Champ1;
    Type2  Champ2 ;
  }   ;

 struct Typestruct *S;

    ALLOC_STRUCT (S)

 void Alloc_struct( struct Typestruct *S)
  {
   S = (struct Typestruct *) malloc ( sizeof (struct Typestruct));
  }
```

```
   LIBER_STRUCT (S)
void Liber_struct( struct Typestruct *S)
  {
   free (S);
  }
```

STRUCT (S, I )

Si le type du champ I est un scalaire, STRUCT se traduit par une fonction comme suit :

```
Typei Structi ( struct Typestruct S )
  {
   return(S->Champi) ;
  }
```

Il y a donc autant de fonctions STRUCT que de champs scalaires dans la structure.

Si le type du champ I est n'est pas scalaire, c'est à dire un vecteur à une dimension de scalaires, STRUCT se traduit par une procédure comme suit :

```
void Struct2 ( struct Typestruct *S , Type2 Result)
  {
   int K;
   for (K=0; K<5; K++)
     Result[K] = S->Champ2[K] ;
  }
```

S'il s'agit d'un tableau de chaînes l'affectation se fait par la fonction "strcpy".

AFF_STRUCT (S, I, Exp)

```
void Aff_struct1 ( struct Typestruct *S,  Type1 Val )
  {
   struct Typestruct P;
   P = *S;
   P.Champ1 = Val;
   *S = P;
  }
```

Il y a donc autant de fonction Aff_struct que de champs scalaire dans la structure.

## Implémentation des structures en C / Exemple

Le Z-algorithme suivant :

```
SOIENT
  S UNE STRUCTURE (ENTIER, VECTEUR(5) DE CHAINES) DYNAMIQUE;
  V1, V2 DES VECTEURS(5) DE CHAINES;
  I UN ENTIER ;
DEBUT
  AFF_ELEMENT(V1[1],'1');
  AFF_ELEMENT(V1[2],'2');
  AFF_ELEMENT(V1[3],'3');
```

```
  AFF_ELEMENT(V1[4],'4');
  AFF_ELEMENT(V1[5],'5');
  ALLOC_STRUCT(S);
  AFF_STRUCT(S, 1, 5);
  AFF_STRUCT(S, 2,  V1);
  ECRIRE('champ1 = ', STRUCT(S, 1) );
  ECRIRE('champ2 =');
  V2 = STRUCT(S, 2);
  POUR I := 1, 5  ECRIRE( V2[I])  FPOUR;
  LIBER_STRUCT(S);
FIN
```

se traduit en C comme suit :

```
#include <alloc.h>
typedef int Type1  ;
typedef char String[255];
typedef String  Type2 [5]  ;

struct Typestruct
  {
   Type1  Champ1;
   Type2  Champ2 ;
  } ;

struct Typestruct *S;
int I ;
Type2 V1, V2 ;

void Alloc_struct( struct Typestruct *S)
  {
   S = (struct Typestruct *) malloc ( sizeof (struct Typestruct));
  }

void Liber_struct( struct Typestruct *S)
  {
   free (S);
  }

Type1 Struct1 ( struct Typestruct *S )
  {
   return(S->Champ1) ;
  }

void Struct2 ( struct Typestruct *S , Type2 Result)
  {
   int K;
   for (K=0; K<5; K++)
     strcpy(Result[K], S->Champ2[K] );
  }

void Aff_struct1 ( struct Typestruct *S,  Type1 Val )
  {
   struct Typestruct P;
```

```
   P = *S;
   P.Champ1 = Val;
   *S = P;
  }

void Aff_struct2 ( struct Typestruct *S,  Type2 Val )
  {
   int K;
   struct Typestruct P;
   P = *S;
   for (K=0; K<5; K++)
     strcpy(P.Champ2[K], Val[K]);
   *S = P;
  }

main()
   {
   strcpy(V1[0], "1");
   strcpy(V1[1], "2");
   strcpy(V1[2], "3");
   strcpy(V1[3], "4");
   strcpy(V1[4], "5");
   Alloc_struct(S);
   Aff_struct1(S, 5);
   Aff_struct2(S, V1);
   printf("champ1 = %d \n", Struct1(S) );
   printf("champ2 = \n");
   Struct2(S, V2);
   for ( I= 0; I<5; I++) printf("%s \n",V2[I])  ;
   Liber_struct(S);
  }
```

## Implémentation des listes linéaires chaînées en C / Dynamique

```
/* Maillon*/
struct Maillon
  {
   int  Val ;
   struct Maillon *Suiv ;
   } ;

/* Opérations */
struct Maillon *Allouer ( )
  {
   return  (  (struct Maillon *) malloc( sizeof(struct Maillon))   );}


void Aff_val(struct Maillon *P, int V)
 {  P->Val =V; }

void Aff_adr( 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) ; }
```

**Implémentation des listes linéaires chaînées en C / Statique**

Plusieurs listes dans un même tableau. Le tableau est un ensemble de triplets (Element, Suivant, Occupe). Le champ "Occupe" est nécessaire pour les opérations Allouer et Libérer. Une phase d'initialisation est obligatoire avant l'utilisation de ce tableau. Donc le tableau est global.Une liste est définie par l'indice de son premier élément.

```
#define Max 100
#define True 1
#define False 0
#define Nil -1

typedef int Bool;
typedef int Typeqq;

struct Typeliste
  {
   Typeqq Element ;
    int Suivant;
    Bool Occupe;
 } ;

/* Initialisation */
void Init()
  {
    int I;
    for (I=0; I<Max; I++)
      Liste[I].Occupe = False;
  }

void Allouer ( int *I )
  {
    Bool Trouv;
    *I = 0;
    Trouv = False;
    while ( *I < Max && !Trouv )
      if ( Liste[*I].Occupe )
        *I++  ;
      else
        Trouv = True;

      if  ( !Trouv )  *I = -1;
  }

void Liberer ( int I )
  {
```

```
    Liste[I].Occupe = False ;
  }

 Typeqq Valeur ( int I )
  {
   return( Liste[I].Element );
  }

 int Suivant ( int I )
  {
   return ( Liste[I].Suivant ) ;
  }
 void Aff_val ( int I, Typeqq Val)
  {
   Liste[I].Element = Val;
  }

 void Aff_adr ( int I, int J)
  {
   Liste[I].Suivant = J;
  }
```

## Implémentation des listes linéaires chaînées en C / Exemple

```
 struct Typeliste Liste[ Max ];
 int E;
 main()
  {
   Init();
   Allouer (&E);
   if ( E != Nil )
     {
      Aff_val (E, 25);
      Aff_adr(E, Nil);
      printf("Ok");
     }
   else
     printf(" Pas d"espace ");
  }
```

## Implémentation des listes bidirectionnelles en C / Dynamique

```
 /* structure du maillon */

 struct Maillon
  {
   int  Val ;
   struct Maillon *Suiv ;
   struct Maillon *Prec ;
  } ;

 /* Opérations */
```

```
struct Maillon *Allouer ( )

   { return  (  (struct Maillon *) malloc( sizeof(struct Maillon))   );}


void Aff_val(struct Maillon *P, int V)
  { P->Val =V; }

void Aff_adrd( struct Maillon *P,  struct Maillon *Q)
  {  P->Suiv = Q; }

void Aff_adrg( struct Maillon *P,  struct Maillon *Q)
  {  P->Prec = Q; }
struct Maillon *Suivant(  struct Maillon *P)
  { return( P->Suiv );  }

struct Maillon *Precedent(  struct Maillon *P)
  { return( P->Prec );  }

int Valeur( struct Maillon *P)
   {return( P->Val) ; }
```

## Implémentation des listes bidirectionnelles en C / Statique

Plusieurs listes dans un même tableau. Le tableau est un ensemble de quadruplé (Element, Suivant, Precedent, Occupe). Le champ "Occupe" est nécessaire pour les opérations Allouer et Libérer. Une phase d'initialisation est obligatoire avant l'utilisation de ce tableau.Donc le tableau est global. Une liste est définie par l'indice de son premier élément.

```
#define Max 100
#define True 1
#define False 0
#define Nil -1

typedef int Bool;
typedef int Typeqq;

struct Typeliste
  {
    Typeqq Element ;
    int Suivant;
    int Precedent;
    Bool Occupe;
  };

/* Initialisation */
void Init()
  {
    int I;
    for (I=0; I<Max; I++)
      Listebi[I].Occupe = False;
  }
```

```
void Allouer ( int *I )
  {
    Bool Trouv;
    *I = 0;
    Trouv = False;
    while ( *I < Max && !Trouv )
     if ( Listebi[*I].Occupe )
        *I++  ;
     else
        Trouv = True;

    if  ( !Trouv )  *I = -1;
  }
void Liberer ( int I )
  {
    Listebi[I].Occupe = False ;
  }

Typeqq Valeur ( int I )
  {
    return( Listebi[I].Element );
  }

int Suivant ( int I )
  {
    return ( Listebi[I].Suivant ) ;
  }

int Precedent ( int I )
  {
    return ( Listebi[I].Precedent ) ;
  }

void Aff_val ( int I, Typeqq Val)
  {

    Listebi[I].Element = Val;
  }

void Aff_adrd ( int I, int J)
  {
    Listebi[I].Suivant = J;
  }

void Aff_adrg ( int I, int J)
  {
    Listebi[I].Precedent = J;
  }
```

## Implémentation des listes bidirectionnelles en C / Exemple

```
struct Typeliste Listebi[ Max ];
int E;
main()
  {
   Init();
   Allouer (&E);
   if ( E != Nil )
     {
      Aff_val (E, 25);
      Aff_adrg(E, Nil);
      Aff_adrd(E, Nil);
     }
   else
     printf(" Pas d"espace ");
  }
```

## Implémentation des arbres de recherche binaire en C / Dynamique

```
struct Noeud
  {
   int  Element ;
   struct Noeud *Fg ;
   struct Noeud *Fd ;
  } ;

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; }
```

## Implémentation des arbres de recherche binaire en C / Statique

Plusieurs arbres de recherche binaire dans un même tableau. Le tableau est un ensemble de 5-uplets(Info, Fg, Fd, Pere, Occupe). Le champ "Occupe" est nécessaire pour les opérations Creernoeud et Liberernoeud. Une phase d'initialisation est obligatoire avant l'utilisation de ce tableau.
Donc le tableau est global. Un arbre de recherche binaire est défini par l'indice de son premier élément.

```
#define Max 100
#define True 1
#define False 0
#define Nil -1

typedef int Bool;
typedef int Typeqq;

struct Typearb
  {
   Typeqq Info ;
   int Fg;
   int Fd;
   Bool Occupe;
  };

/* Initialisation */
void Init()
  {
   int I;
   for (I=0; I<Max; I++)
     Arb[I].Occupe = False;
  }

void Creernoeud ( int *I )
  {
   Bool Trouv;
   *I = 0;
   Trouv = False;
   while ( *I < Max && !Trouv )
    if ( Arb[*I].Occupe )
      *I++  ;
    else
      Trouv = True;
   if  ( !Trouv )  *I = -1;
  }

void Liberernoeud ( int I )
  {
   Arb[I].Occupe = False ;
  }
```

```
Typeqq Info ( int I )
  {
   return( Arb[I].Info );
  }

int Fd ( int I )
  {
   return ( Arb[I].Fd ) ;
  }

int Fg ( int I )
  {
   return ( Arb[I].Fg ) ;
  }

void Aff_info ( int I, Typeqq Val)
  {
   Arb[I].Info = Val;
  }


void Aff_fd ( int I, int J)
  {
   Arb[I].Fd = J;
  }

void Aff_fg ( int I, int J)
  {
   Arb[I].Fg = J;
  }
```

## Implémentation des arbres de recherche binaire en C / Exemple

```
struct Typearb Arb[ Max ];
int E;
main()
  {
   Init();
   Creernoeud (&E);
   if ( E != Nil )
     {
      Aff_info (E, 25);
      Aff_fg(E, Nil);
      Aff_fd(E, Nil);
     }
   else
     printf(" Pas d"espace ");
  }
```

## Implémentation des arbres de recherche m-aire en C / Dynamique

```
#include <Alloc.H>
```

```
#include <Stdio.H>

#define Ordre 5
#define True  1
#define False 0

typedef int Bool;
typedef int Typeqq;

struct Noeud
  {
   Typeqq  Info[Ordre-1] ;
   struct Noeud *Fils[Ordre] ;
   short Degre;
  } ;

void Creernoeud ( struct Noeud *P)
  {
   int I;
   P =   (struct Noeud *) malloc( sizeof(struct Noeud))   ;
   for (I=1; I<Ordre ; I++)
     P->Fils[I] = NULL;
  }
struct Noeud *Fils ( struct Noeud *P, short I)
  {       return ( P->Fils[I] );  }

Typeqq Infor ( struct Noeud *P, short I)
  {       return ( P->Info[I] ); }

void Aff_infor(struct Noeud *P, short I, Typeqq  V)
   {    P->Info[I] =V; }

void Aff_fils(struct Noeud *P, short I, struct Noeud *Q)
   {    P->Fils[I] = Q; }

short Degre ( struct Noeud *P)
  {
   return ( P->Degre );
  }

void Aff_degre ( struct Noeud *P, short I)
  {
   P->Degre = I ;
            }
```

**Implémentation des arbres de recherche m-aire en C / Exemple**

```
struct Noeud *A;
int I;

main()
  {
   Creernoeud (A);
```

```
  for (I=0; I < 5; I++ )
    {
      Aff_infor(A, I, 3*I);
    };
  for (I=0; I < 5; I++ )
    printf("%d  ", Infor(A, I) );
}
```

## Implémentation des arbres de recherche m-aire en C / Statique

Plusieurs arbres de recherche m-aire dans un même tableau.Le tableau est un ensemble de quadruplé (Info, Fils, Degre, Occupe).Info est un tableau de (Ordre-1) valeurs. Fils est un tableau de (Ordre) indices. Le champ degre contient le nombre courant de valeur dans le noeud. Le champ "Occupe" est nécessaire pour les opérations Creernoeud et Liberernoeud. Une phase d'initialisation est obligatoire avant l'utilisation de ce tableau. Donc le tableau est global. Un arbre de recherche m-aire est défini par l'indice de son premier élément.

```
#define Max 100
#define Ordre 8
#define True 1
#define False 0
#define Nil -1

typedef int Bool;
typedef int Typeqq;

struct Typearm
  {
    int Fils[Ordre];
    int Info[Ordre-1];
    short Degre;
    Bool Occupe;
  };

/* Initialisation */
void Init()
  {
    int I;
    for (I=0; I<Max; I++)
      Arm[I].Occupe = False;
  }

void Creernoeud ( int *P )
  {
    Bool Trouv;
    *P = 0;
    Trouv = False;
    while ( *P < Max && !Trouv )
      if ( Arm[*P].Occupe )
         *P++  ;
      else
         Trouv = True;
```

```
   if  ( !Trouv )  *P = -1;
  }

void Liberernoeud ( int P )
  {
   Arm[P].Occupe = False ;
  }

Typeqq Infor ( int P, int I )
  {
   return( Arm[P].Info[I-1] );
  }

int Fils ( int P, int I )
  {
   return ( Arm[P].Fils[I-1] ) ;
  }

void Aff_infor ( int P, int I, Typeqq Val)
  {
   Arm[P].Info[I-1] = Val;
  }

void Aff_fils ( int P, int I, int J)
  {
   Arm[P].Fils[I-1] = J;
  }

short Degre ( int P )
  {
   return ( Arm[P].Degre );
  }
void Aff_degre ( int P, short I )
  {
   Arm[P].Degre = I ;
  }
```

**Implémentation des arbres de recherche m-aire en C / Exemple**

```
struct Typearm Arm[ Max ];
int E;
main()
  {
   Init();
   Creernoeud (&E);
   if ( E != Nil )
     {
      Aff_infor (E, 1, 25);
      Aff_fils(E, 2, Nil);
     }
   else
     printf(" Pas d"espace ");
  }
```

## Implémentation des piles en C / Dynamique

```
#include <Alloc.H>
#include <Stdio.h>

typedef int Bool;
typedef int Typeqq;

struct Maillon
  {
   int  Valeur ;
   struct Maillon *Suivant ;
  } ;
void Creerpile( struct Maillon ** P )
  {
   struct Maillon *Pil;
   Pil = *P;
   Pil = NULL;
   *P = Pil;
  }

Bool Pilevide ( struct Maillon *P )
  {
   struct Maillon *Pil;
   Pil = P;
   return ( Pil == NULL ) ;
  }

void Empiler ( struct Maillon **P, Typeqq Val )
  {
   struct Maillon *Q;
   Q = (struct Maillon *) malloc ( sizeof(struct Maillon));
   Q->Valeur = Val;
   Q->Suivant = *P;
   *P = Q;
  }

void Depiler ( struct Maillon **P, Typeqq *V )
  {
   struct Maillon *Sauv, *Pil;
   Pil = *P;
   if ( ! Pilevide ( Pil ) )
     {
      *V = Pil->Valeur;
      Sauv = Pil;
      Pil = Pil->Suivant;
      *P = Pil;
     }
   else
     printf(" Pile vide ");
}
```

## Implémentation des piles en C / Statique

```
#define True  1
#define False 0
typedef int Bool;

struct Pile
  {
   int Som ;
   struct Noeud *Tab[50];
  };

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( "Pile satur馥\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( "Pile vide\n"); exit(0) ;
  }
```

## Implémentation des piles  en C / Exemple

```
struct Maillon *Pile;
Typeqq V ;

main()
  {
   Creerpile(&Pile);
   Empiler ( &Pile, 25);
   Empiler ( &Pile, 35);
   Empiler ( &Pile, 45);
   Depiler ( &Pile, &V );
   printf( " %d ", V);
  }
```

## Implémentation des files d'attente en C / Dynamique

```
#define True  1
#define False 0
typedef int Bool;
Typedef int typelment;

struct Filedattente
  {
   struct Maillon *Tete, *Queue;
  };
struct Maillon
  {
   typelement Val ;
   struct Maillon *Suiv ;
  } ;

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, typelement 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, typelement *Val)
 {
  struct Filedattente Q;
  Q = *F;
  if (  !Filevide(*F)  )
    {
     *Val = Valeur(Q.Tete)  ;
     Q.Tete = Suivant(Q.Tete);
     *F = Q ;
    }
  else
     fprintf("File Vide\n"); exit(0) ;
 }
```

## Implémentation des files d'attente en C / Statique

```
#define Max 100
typedef int Bool;
typedef int Typeqq;

struct Typefile
 {
  Typeqq Elements[Max];
  int Tete, Queue;
 };

void Creerfile ( struct Typefile *F )
 {
  struct Typefile Fil;
  Fil = *F;
  Fil.Tete = Max - 1;
  Fil.Queue = Max - 1;
  *F = Fil;
 }

Bool Filevide ( struct Typefile F )
 {
  return ( F.Tete == F.Queue );
 }

Bool Filepleine ( struct Typefile F )
 {
  return ( F.Tete == F.Queue % (Max-1) + 1 );
 }
```

```
void Enfiler ( struct Typefile *F, Typeqq Val )
  {
   struct Typefile Fil;
   Fil = *F;
   if ( ! Filepleine(Fil) )
     {
      if (Fil.Queue == Max-1 )
        Fil.Queue = 0;
      else
        Fil.Queue =  Fil.Queue + 1 ;
      Fil.Elements[Fil.Queue] = Val;
      *F = Fil;
     }
   else
     printf(" File pleine");
  }
void Defiler ( struct Typefile *F, Typeqq *Val )
  {
   struct Typefile Fil;
   Fil = *F;
   if ( ! Filevide(Fil) )
     {
      if ( Fil.Tete == Max - 1)
        Fil.Tete = 0;
      else
        Fil.Tete = Fil.Tete + 1;

      *Val = Fil.Elements[Fil.Tete];
      *F = Fil ;
     }
   else
     printf(" File vide");
  }
```

## Implémentation des files d'attente en C / Exemple

```
struct Typefile F;
Typeqq V;
main()
  {
   Creerfile (&F);
   Enfiler(&F, 5);
   Enfiler(&F, 18);
   Enfiler(&F, 22);
   Defiler(&F, &V);
   printf("%d\n", V);
  }
```

## Implémentation des fichiers en C

```
/* Types des champs formant le bloc du fichier */
typedef type du premier champ du bloc du fichier  Type1  ;
typedef type du deuxième champ du bloc du fichier  Type2  ;
...
typedef type du n-ième champ du bloc du fichier  Typen  ;

/* Types des champs formant l'entête du fichier */
typedef type du premier champ des caractéristiques du fichier Typecaract1 ;
typedef du deuxième champ des caractéristiques du fichier Typecaract2 ;
...
typedef du n-ième champ des caractéristiques du fichier Typecaractn ;

/* Types de travail */
typedef int Boolean;
typedef char String[255];

/* Type du bloc de données du fichier */
typedef struct
  {
   Type1 Champ1 ;
   Type2 Champ2;
   ...
   Typen Champn;
   }  Typestruct1;

/* Type du bloc des caractéristiques du fichier */
typedef struct
  {
   Typecaract1 Caract1;
   Typecaract2 Caract2 ;
   ...
   Typecaractm Caractm ;
   }Typestruct2 ;

/* Type du bloc du fichier */
typedef union
 {
   Typestruct1 Buf_Donnees;
   Typestruct2 Buf_Caract;
 }Typebloc;


FILE *F;                /* Le fichier */
Typebloc Bloc_Caract;    /* Le bloc des caractéristiques du fichier */

/* Machine abstaite sur les fichiers */

void Ouvrir ( FILE **Fl , String Fp , String Mode )
  {
    if ( strcmp(Mode,"A") == 0)
       {
```

```
        *Fl = fopen(Fp, "r+b");
        fread(&Bloc_Caract, sizeof(Typebloc), 1, *Fl);
       }
     else
       {
        *Fl = fopen(Fp, "w+b");
        fwrite(&Bloc_Caract, sizeof(Typebloc), 1, *Fl)  ;
       }
  }

void Fermer ( FILE **Fl )
  {
   fseek(*Fl, 0, 0);
   fwrite(&Bloc_Caract, sizeof( Typebloc), 1, *Fl)  ;
   fclose( *Fl) ;
  }
```

ENTETE ( Fl, i )

```
Typecaract Entetei(  FILE *Fl)
  {
   return(Bloc_Caract.Buf_Caract.Caracti) ;
  }

/* Il y a donc autant de fonctions ENTETEi que de types scalaires définis dans la partie 'ENTETE'. */
```

AFF_ENTETE ( Fl, i, Exp )

```
void Aff_entetei ( FILE *Fl, Typecaracti Val)
  {
   Bloc_Caract.Buf_Caract.Caracti = Val ;
  }

 /* Il y a donc autant de fonctions AFF_ENTETEi que de types scalaires définis dans la partie
'ENTETE'. */

void Ecrireseq ( FILE **Fl,  Typebloc Buf )
  {
   fwrite(&Buf, sizeof( Typebloc), 1, *Fl)  ;
  }

void Ecriredir ( FILE **Fl,  Typebloc Buf, int N )
  {
   fseek(*Fl, 0, (long) (N-1)* sizeof(  Typebloc) /*SEEK_SET*/);
   fwrite(&Buf, sizeof( Typebloc), 1, *Fl)  ;
  }

void Lireseq ( FILE **Fl,  Typebloc *Buf )
  {
   fread(&*Buf, sizeof( Typebloc), 1, *Fl);
  }

void Liredir ( FILE **Fl, Typebloc *Buf, int N )
```

```
  {
   fseek(*Fl, 0, (long) (N-1)* sizeof(  Typebloc) /*SEEK_SET*/ );
   fread(&*Buf, sizeof( Typebloc), 1, *Fl);
  }

Boolean Finfich ( FILE **Fl)
  {
   if ( feof(*Fl) == 0 )
     return(0)   ;
   else
     return(1)    ;
  }

int  Alloc_bloc ( FILE *Fl )
  {
   int K;
   fseek(Fl, 0, 2); /* Fin du fichier */
   K = ftell(Fl);   /* position _ partir du d饕ut */
   K = K / sizeof (Typebloc);
   K ++;
   return(K);
 }
```

**Implémentation des fichiers en C / Exemple**

Le Z-algorithme suivant :

```
SOIENT
  F UN FICHIER DE ( ENTIER , VECTEUR ( 5 ) DE ENTIERS )
  BUFFER Bloc
  ENTETE ( ENTIER , ENTIER ) ;
   { fichier de blocs contenant le nombre d'articles et un tableau d'articles}
   { entete : nombre d'articles, nombre de blocs}
  Creer , Ajouter, Imprimer DES ACTIONS ;
DEBUT
  APPEL Creer ;
  APPEL Imprimer ;
  APPEL Ajouter ;
  APPEL Imprimer;
 FIN

/******    Chargement de 5 blocs _ raison de 60% ****/
ACTION Creer ;
SOIENT
  I , K DES ENTIERS ;
DEBUT
  OUVRIR ( F , 'f.pas' , 'N' ) ;
  POUR I:=1, 5 :
    POUR K:=1, 3 :
      AFF_ELEMENT ( STRUCT ( Bloc , 2 ) [ K ] , ALEAENTIER )
    FPOUR
    AFF_STRUCT ( Bloc , 1 , K ) ;
    ECRIRESEQ ( F , Bloc ) ;
  FINPOUR ;
```

```
  AFF_ENTETE ( F , 1 , 5 ) ;
  AFF_ENTETE ( F , 2 , 3*5 ) ;
  FERMER ( F ) ;
 FIN

/*  Réouvre le fichier, rajoute un bloc en fin de fichier puis le ferme */

 ACTION Ajouter
  DEBUT
   OUVRIR(F, 'f.c' , 'A' ) ;
   POUR  K := 1, 4
    AFF_ELEMENT ( STRUCT ( Bloc , 2 ) [ K ], ALEAENTIER );
       FPOUR:
   AFF_STRUCT ( Bloc, 1 , 4 ) ;

   ECRIREDIR(F, Bloc, ALLOC_BLOC(F) );
   FERMER(F);
  FIN

 /*****       Impression des articles du fichier ****/
 ACTION Imprimer ;
 SOIENT
  I , K DES ENTIERS ;
 DEBUT
  I := 0 ;
  OUVRIR ( F , 'f.pas' , 'A' ) ;
  ECRIRE('Caractéristiques du fichier');
  ECRIRE('- Nombre de blocs : ', Entete(F, 1) );
  ECRIRE('- Nombre d''articles : ', Entete(F, 2) );
  LIRESEQ ( F , Bloc ) ;
  TQ NON FINFICH ( F )
    I := I + 1 ;
    ECRIRE ( 'B  L  O  C    ' , I ) ;
    LIRESEQ ( F , Bloc ) ;
    POUR K := 1 , STRUCT ( Bloc , 1 )
ECRIRE ( ELEMENT ( STRUCT ( Bloc , 2 ) [ K ] ) )
    FPOUR
    LIRESEQ ( F , B1 ) ;
  FTQ;
  FERMER(F);
 FIN
```

se traduit en C Comme suit :

```
 #include "stdio.h"
 #include "string.h"
 #include "stdlib.h"

 typedef int Type1  ;
 typedef int  Type2 [5]  ;

 typedef char String[255];

 typedef int Typecaract1 ;
```

```
typedef int Typecaract2 ;

typedef int Boolean;

typedef struct
  {
   Type1 Champ1 ;
   Type2 Champ2;
   }  Typestruct1;

typedef struct
  {
   Typecaract1 Caract1;
   Typecaract2 Caract2 ;
   }Typestruct2 ;

typedef union
 {
   Typestruct1 Buf_Donnees;
   Typestruct2 Buf_Caract;
 } Typebloc;

/*fichier de blocs contenant le nombre d'articles et un tableau d'articles */
FILE *F;

Typebloc
   Bloc ,          /* pour les lectures/écritures */
   Bloc_Caract ;   /* pour la sauvegarde des caractéristiques */

/* Machine abstraite sur les vecteurs */
int Element ( Type2 V, int I )
  {
    return( V[I] );
  }

void Aff_element ( Type2 V, int I, int Val)
  {
    V[I] = Val;
  }
/* Machine abstraite sur les structures */
Type1 Struct1 ( Typestruct1 S )
  {
    return(S.Champ1) ;
  }

void Struct2 ( Typestruct1 S , Type2 Result)
  {
    int K;
    for (K=0; K<5; K++)
      Result[K] = S.Champ2[K] ;
  }
```

```
void Aff_struct1 ( Typestruct1 *S,  Type1 Val )
  {
   Typestruct1 P;
   P = *S;
   P.Champ1 = Val;
   *S = P;
  }

void Aff_struct2 ( Typestruct1 *S,  Type2 Val )
  {
   int K;
   Typestruct1 P;
   P = *S;
   for (K=0; K<5; K++)
     P.Champ2[K] = Val[K] ;
   *S = P;
  }

/* Machine abstraite sur les fichiers */
void Ouvrir ( FILE **Fl , String Fp , String Mode )
  {
   if ( strcmp(Mode,"A") == 0)
     {
      *Fl = fopen(Fp, "r+b");
      fread(&Bloc_Caract, sizeof(Typebloc), 1, *Fl);
     }
   else
     {
      *Fl = fopen(Fp, "w+b");
      fwrite(&Bloc_Caract, sizeof(Typebloc), 1, *Fl)  ;
     }
  }

void Fermer ( FILE **Fl )
  {
   fseek(*Fl, 0, 0);
   fwrite(&Bloc_Caract, sizeof( Typebloc), 1, *Fl)  ;
   fclose( *Fl) ;
  }

Typecaract1 Entete1(  FILE *Fl)
  {
   return(Bloc_Caract.Buf_Caract.Caract1) ;
  }
Typecaract2 Entete2(  FILE *Fl )
  {
   return (Bloc_Caract.Buf_Caract.Caract2);
  }

void Aff_entete1 ( FILE *Fl, Typecaract1 Val)
  {
   Bloc_Caract.Buf_Caract.Caract1 = Val ;
  }
```

```
void Aff_entete2 ( FILE *Fl, Typecaract2 Val)
  {
   Bloc_Caract.Buf_Caract.Caract2 = Val ;
  }
void Ecrireseq ( FILE **Fl,  Typebloc Buf )
  {
   fwrite(&Buf, sizeof( Typebloc), 1, *Fl)  ;
  }
void Ecriredir ( FILE **Fl,  Typebloc Buf, int N )
  {
   fseek(*Fl, (long) (N-1)* sizeof(  Typebloc), 0 /*SEEK_SET*/);
   fwrite(&Buf, sizeof( Typebloc), 1, *Fl)  ;
  }
void Lireseq ( FILE **Fl,  Typebloc *Buf )
  {
   fread(&*Buf, sizeof( Typebloc), 1, *Fl);
  }
void Liredir ( FILE **Fl, Typebloc *Buf, int N )
  {
   fseek(*Fl, (long) (N-1)* sizeof(  Typebloc), 0 /*SEEK_SET*/ );
   fread(&*Buf, sizeof( Typebloc), 1, *Fl);
  }
Boolean Finfich ( FILE **Fl)
  {
   if ( feof(*Fl) == 0 )
     return(0)   ;
   else
     return(1)    ;
  }
int Aleaentier ()
  {
   return( random(10000) );
  }

int  Alloc_bloc ( FILE *Fl )
  {
   int K;
   fseek(Fl, 0, 2);
   K = ftell(Fl); /* position _ partir du d饕ut */
   K = K / sizeof (Typebloc);
   K ++;
   return(K);
  }

/******    Chargement de 5 blocs à raison de 60% ****/
void Creer  ()
  {
   int I , K  ;
   Type2 V;
   Ouvrir ( &F , "f.c" , "N" ) ;
   for ( I = 1; I<=5; I++)
     {
      for ( K = 0; K<3; K++)
```

```
          Aff_element ( V , K  , Aleaentier() );
        Aff_struct2( &Bloc.Buf_Donnees, V);
        Aff_struct1 ( &Bloc.Buf_Donnees , 3 ) ;
        Ecrireseq ( &F , Bloc ) ;
       } ;
     Aff_entete1(F, 5);  /* Nombre de blocs   */
     Aff_entete2(F, 3*5); /* Nombre d'articles */
     Fermer ( &F ) ;
    }

  void Ajouter()
    {
     int K;
     Type2 V;
     Ouvrir ( &F , "f.c" , "A" ) ;
     for ( K = 0; K<4; K++)
       Aff_element ( V , K  , Aleaentier() );
     Aff_struct2( &Bloc.Buf_Donnees, V);
Aff_struct1 ( &Bloc.Buf_Donnees , 4 ) ;
     Ecriredir(&F, Bloc, Alloc_bloc(F) );
     Fermer(&F);
    }
  /*****    Impression des articles du fichier ****/
  void Imprimer ()
    {
     int I=0;
     int  K;
     Type2 V;
     Ouvrir ( &F , "f.c" , "A" ) ;
     printf("Caractéristiques du fichier : \n");
     printf("-Nombre de bloc    : %d \n", Entete1(F) );
     printf("-Nombre d'articles : %d \n", Entete2(F) );
     Ouvrir ( &F , "f.c" , "A" ) ;
     Lireseq ( &F , &Bloc ) ;
     while ( ! Finfich ( &F )  )
       {
        I++;
        printf("\n Bloc %d: \n ", I);
        Struct2(Bloc.Buf_Donnees, V);
        for ( K=0; K < Struct1 ( Bloc.Buf_Donnees ); K++)
          printf("Element : %d \n", Element ( V , K  ) );
        Lireseq ( &F , &Bloc ) ;
       }
     Fermer(&F);
    }

  main ()
    {
     randomize();    Creer() ;      Imprimer() ;
     Ajouter();      Imprimer();
    }
```

## 10. Index des mot-clés Z