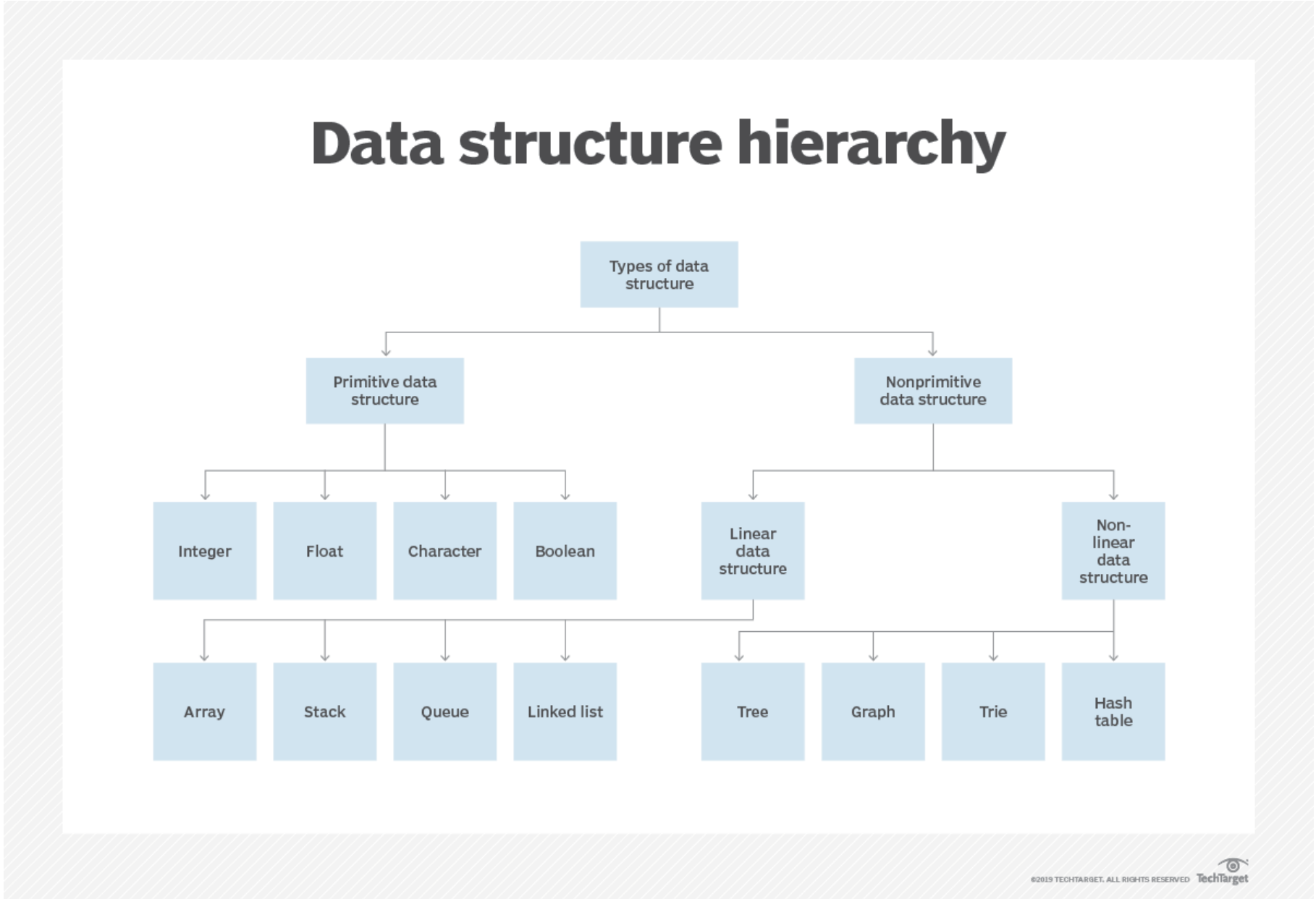
Structure de Données

Mohamed Maachaoui

Structure de données

- une **structure de données** est une manière d'organiser les données pour les traiter plus facilement. Une structure de données est une mise en œuvre concrète d'un type abstrait.
- Les structures de données sont souvent classées d’après leurs caractéristiques :
  - Linéaires ou non linéaires
  - Homogènes ou non homogènes
  - Statiques ou dynamiques
- Exemples :
- Tableau, Pile, File, Liste chaînée, Arbre, Graphe,…

Structure de données



Les Types énumérés

Questions

1. Comment définir un type mois ?

En algorithmique :

```
T_Mois = 1..12
```

1..12 est un intervalle qui correspond aux entiers 1 à 12, numérotation conventionnelle des mois de l'année.

Les Types énumérés

Questions

2. Comment définir un type jours ?

```
T_Jours = 1..7
```

ou

```
T_Jours = 0..6
```

Il n'y a pas de consensus comme pour les mois :

- Faut-il numéroter les jours de 0 à 6 ou de 1 à 7 ?
- La semaine démarre-t-elle à dimanche ou à lundi ?

Les Types énumérés

Type énuméré

Définir un type énuméré qui définit toutes les valeurs possibles :

```
Type
        Jour = ( LUNDI, MARDI, MERCREDI,
                    JEUDI, VENDREDI, SAMEDI, DIMANCHE )
        Mois = (JAN, FEV, MAR, AVR, MAI, JUIN, JUIL,
                    AOUT, SEPT, OCT, NOV, DEC )
```

- Chaque valeur est nommée par un identifiant
- Convention : identifiant en majuscules
- Chaque identifiant est une constante symbolique initialisée par le compilateur

  LUNDI vaut 0, **MARDI** vaut 1, **MERCREDI** vaut 2, etc ...

Les Types énumérés en C

- Type avec un nombre de valeur déterminé à l'avance

```
enum jour
     {lundi, mardi, mercredi, jeudi,
     vendredi,samedi, dimanche};

enum jour j = samedi;
```

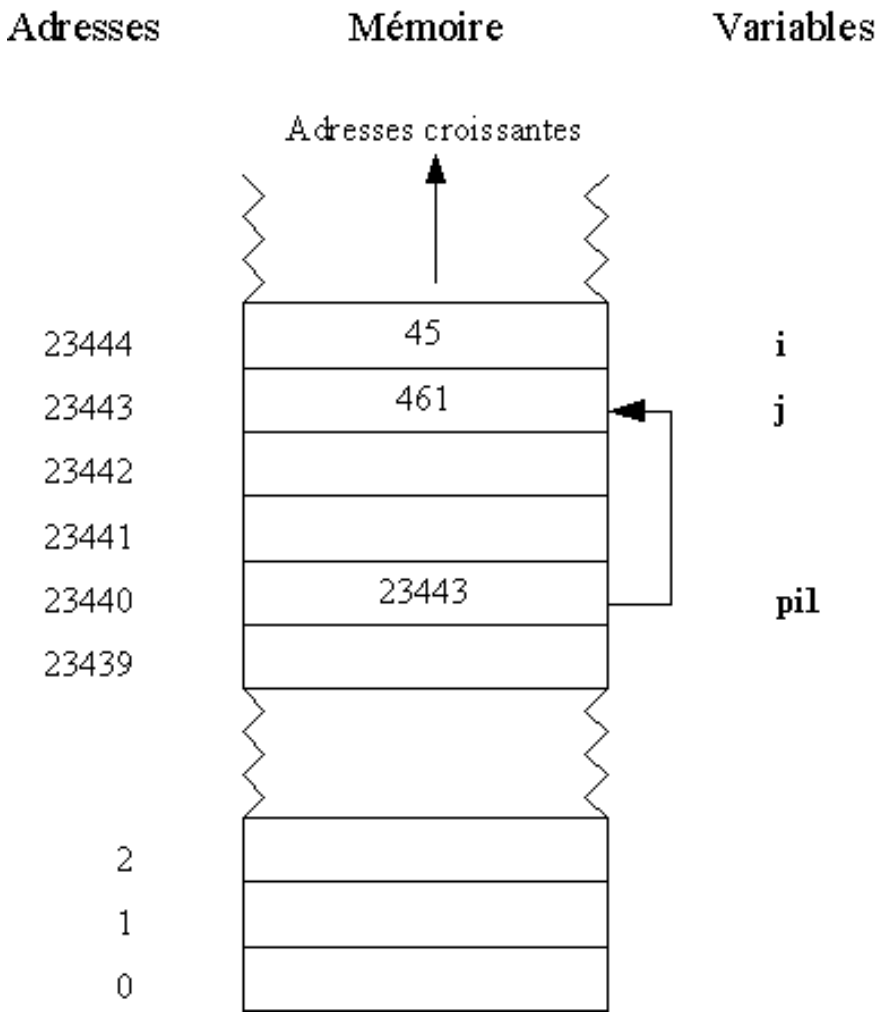
Les Types énumérés

Remarques

- `enum X {...};` définit un type `enum X` et non un type `X`
- les variables de type *enum* sont implicitement converties en int, le premier identifiant valant 0, le suivant 1, etc. . .
- historiquement nommées "constantes énumérées"

Les pointeurs

- Variable qui contient l'adresse



Les pointeurs

- Deux types de pointeur
    - Non typé
    - Typé

```
void * p;     /*Contient l'adresse de
              n'importe quel type de
              variable*/

int * pi;     /*Contient l'adresse
              d'uniquement un int*/
```

Les pointeurs

```
void * p;
int * pi;

int i;

pi = &i;
p = &i;

*pi = 10;                    /* ok */
*p = 10;                     /* erreur ! */
*((int *)p) = 10;            /* ok */
```

Les pointeurs

```
void * p;
int * pi;

int i;

pi = &i;
p = &i;

*pi = 10;                    /* ok */
*p = 10;                     /* erreur ! */
*((int *)p) = 10;            /* ok */
```

Les structures (Enregistrement)

- Type enregistrement
- Un enregistrement est le produit cartésien de plusieurs types T_1, … T_n
- A chaque composante de type T_i est associé un identificateur choisi par le programmeur. Cet identifiant permet d'accéder à la valeur de la composante.
- Le couple (identificateur, Type) est appelé champ ou attribut de l'enregistrement
- Notation en Algorithmique

```
Type
      T = -- Le nom du type (significatif !)
            Enregistrement -- sa définition
                  nomChamp1 : T1 -- un champ, son nom, type et rôle
                  nomChampn : Tn -- dernier champ, son nom, type et rôle
            FinEnregistrement
```

Les structures (Enregistrement)

- Exemple d'enregistrement

```
Type
     Date = -- Type enregistrement Date
          Enregistrement
               leJour   : Jour   -- numéro du Jour
               leMois   : Mois   -- mois de type énuméré
               lAnnee   : Année  -- année
          FinEnregistrement
```

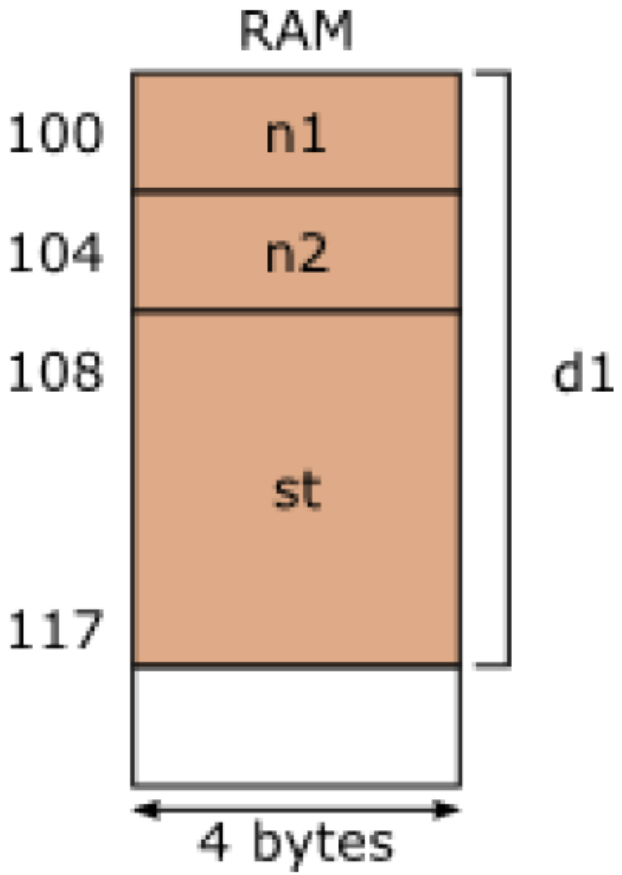
Attention

- Les T_i sont de n'importe quel type : type scalaire, type énuméré et aussi enregistrement.
- *Seule contrainte* : T ne peut avoir un membre de type T

Les structures

- Ensemble de variables définit à l'avance de types différents rangés en mémoire

```
struct contact
{
     int n1;
     float n2;
     char st[10];
} d1;
```



Les structures

```c
struct Point
{
    char n; // Nom
    int x; // Abscisses
    int y; // Ordonnées
};

struct Point p1;

p1 /* Adresse du 1ér élément */

p1.n = 'A';
p1.x = '10';

&(p1.x) /* Adresse de x */
```

Les structures

- **Remarques :**
- **`struct X {...};`** définit un type **`struct X`** et non un type **`X`**
- ne pas oublier le **;** après le **}**
- Un membre du type struct X ne peut être du type struct X.
- Par contre, si `struct X1` est défini avant `struct X2`, un membre de `struct X2` peut être de type `struct X1`.

sizeof ()

opérateur donnant la taille en octets de son opérande

```c
#inlude<stdio.h>
#inlude<stdlib.h>

struct my_struct{
  int a;
  size_t b;};

int main() {
  int a;
  size_t s;
  struct my_struct ms;

  printf("sizeof(a)=%lu\n",
         sizeof(a));
  printf("sizeof(s)=%lu\n",
         sizeof(s));
  printf("sizeof(ms)=%lu\n",
         sizeof(ms));
  printf("sizeof(int)=%lu\n",
         sizeof(int));
  printf("sizeof(size_t)=%lu\n",
         sizeof(size_t));
  printf("sizeof(struct␣my...)=%lu\n",
         sizeof(struct my_struct));

  return EXIT_SUCCESS;
}
```

Résultats :

sizeof ()

opérateur donnant la taille en octets de son opérande

```c
#inlude<stdio.h>
#inlude<stdlib.h>

struct my_struct{
  int a;
  size_t b;};

int main() {
  int a;
  size_t s;
  struct my_struct ms;

  printf("sizeof(a)=%lu\n",
         sizeof(a));
  printf("sizeof(s)=%lu\n",
         sizeof(s));
  printf("sizeof(ms)=%lu\n",
         sizeof(ms));
  printf("sizeof(int)=%lu\n",
         sizeof(int));
  printf("sizeof(size_t)=%lu\n",
         sizeof(size_t));
  printf("sizeof(struct␣my...)=%lu\n",
         sizeof(struct my_struct));

  return EXIT_SUCCESS;
}
```

Résultats :

```
sizeof(a)=4
sizeof(s)=8
sizeof(ms)=16
sizeof(int)=4
sizeof(size_t)=8
sizeof(struct my_struct)=16
```

sizeof ()

opérateur donnant la taille en octets de son opérande

```c
#inlude<stdio.h>
#inlude<stdlib.h>

int main() {
  int a;
  int * pa;
  int tab[7];
  int * ptab;

  pa = &a;
  ptab = tab

  printf("sizeof(a)=%lu\n",
         sizeof(a));
  printf("sizeof(pa)=%lu\n",
         sizeof(pa));
  printf("sizeof(tab)=%lu\n",
         sizeof(tab));
  printf("sizeof(ptab)=%lu\n",
         sizeof(ptab));

  return EXIT_SUCCESS;
}
```

Résultats :

sizeof ()

opérateur donnant la taille en octets de son opérande

```c
#inlude<stdio.h>
#inlude<stdlib.h>

int main() {
  int a;
  int * pa;
  int tab[7];
  int * ptab;

  pa = &a;
  ptab = tab

  printf("sizeof(a)=%lu\n",
         sizeof(a));
  printf("sizeof(pa)=%lu\n",
         sizeof(pa));
  printf("sizeof(tab)=%lu\n",
         sizeof(tab));
  printf("sizeof(ptab)=%lu\n",
         sizeof(ptab));

  return EXIT_SUCCESS;
}
```

Résultats :

```
sizeof(a)=4
sizeof(pa)=8
sizeof(tab)=28
sizeof(ptab)=8
```

malloc et free

- Allocation de mémoire dynamique `#include <stdlib.h>`

malloc

fonction d'allocation de mémoire dynamique (allocation dans le segment mémoire du TAS)

```
void *malloc( size_t numbytes );
```

free

libération de la mémoire allouée dynamiquement

```
int free( void *region );
```

malloc et free

- Allocation de mémoire dynamique #include <stdlib.h>

```
#inlude < stdio.h>
#inlude < stdlib.h>

int main (){
      int *pv;

      pv = (int *) malloc ( sizeof (int ) );
      if(pv == NULL )
            return EXIT_FAILURE ;
      ...

      free (pv)
      return EXIT_SUCCESS ;
}
```

Tableaux

Définition

Un tableau un type de données qui permet de regrouper un nombre fini d'éléments, ayant tous le même type.

Conséquence

Un tableau est défini par :

- le type des éléments qu'il contient
- sa taille, i.e. le nombre d'éléments qu'il peut contenir
- le moyen d'accéder à un élément via sa position dans le tableau (son index)

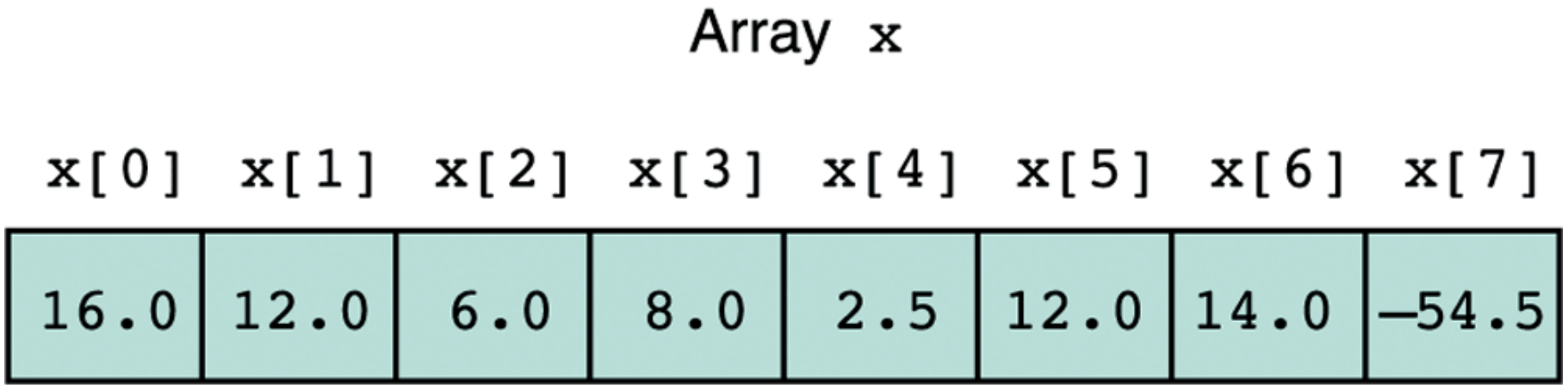
Exemple en algorithmique de variables

```
Variable
    monTab : Tableau [10] De Entier -- tableau de 10 entiers
    autreTab : Tableau [2] De Réels -- tableau de 2 réels
```

Les tableaux

- Ensemble de variable du même type rangé successivement en mémoire
- Accès aux éléments par l'adresse en mémoire du premier élément du tableau

```
double x[8];
```



Tableaux à plusieurs dimensions

Définition

Un tableau à plusieurs dimensions est un tableau pour lequel il faut au moins deux indices pour accéder à un élément.

Principe

Pour accéder à un élément, il faut préciser la valeur de tous les indices.

Exemple

```
Constante
    NB LIGNES = 10
    NB COLONNES = 5
Type
    Matrice = Tableau [NB LIGNES, NB COLONNES] De Réel
Variable
    m1 : Matrice
Début
    m1[0, 0] < - 0.7
    m1[9, 0] < - 30.9
Fin
```

Tableaux à plusieurs dimensions

Occupation mémoire

- Les éléments en mémoire sont généralement contigus (comme un tableau à 1 dimension)
- Ainsi l'accès à l'élément i se fait par la recherche de l'élément à la position

i * NB LIGNES + j

Tableaux de tableaux

Isomorphes à des tableaux à plusieurs dimensions

```
Type
   Vecteur = Tableau [NB COLONNES] De Réel
   Matrice2 = Tableau [NB LIGNES] De Vecteur
Variable
   m2 : Matrice2
Début
   m2[0][0] <- 0.7 -- équivalent à m1[0,0]
   m2[9]           -- est un vecteur
Fin
```

Les tableaux (statiques) en C

Point fondamental

On ne peut pas changer la taille d'un tableau au cours de l'exécution du programme. On doit donc pouvoir dimensionner le tableau à priori.

Modes d'utilisation

- On connait sa taille et on utilise toutes les cases du tableau.
- On ne connait pas sa taille et on sur-dimensionne la capacité du tableau. On n'utilise qu'une partie des cases du tableau. Pour cela, il faut se souvenir du nombre de cases utilisées dans le programme.

Attention, dans les deux cas, il faut contrôler dans le programme que tous les indices sont valides (i.e. sont dans la limite de capacité du tableau)

Les tableaux (statiques) en C

Déclaration

- Déclaration d'un tableau à une dimension de *type-element* :

  type-element nom[taille];

- Déclaration tableau à N dimensions :

  type-element nom[taille-d1]...[taille-dN];

Déclaration d'un synonyme de ce type **syn**

typedef type-element syn[taille-d1]...[taille-dN];

Exemples :

```
#define CAPA 6
int leTableau [CAPA];
double uneMatrice [CAPA][CAPA];
typedef int HyperCube [2][2][2];
```

Pointeur et Tableaux

Rappel et précisions

En C, un tableau est géré en interne à l'aide des pointeurs :

type-element nomTab[taille]

- nomTab est un pointeur constant sur le premier élément du tableau :
- On a donc ***nomTab = nomTab[0]**
- Les "taille – 1" autres valeurs sont stockées dans les cases mémoires consécutives à nomTab[0].
- L'élément numéro 5 du tableau (nomTab[5]) est traduit par le compilateur en un chemin d'accès utilisant le nom du tableau.
- Notation équivalente à nomTab[i] en utilisant le pointeur nomTab :

nomTab[i] == *(nomTab+i)

- Il existe donc une arithmétique des pointeurs pour les tableaux

Les tableaux

```
int tab[5];

tab             /*Adresse du 1èr élément */

*tab            /*Valeur  du 1èr élément */

tab + 1         /*Adresse du 2ème élément */
*(tab + 1)      /*Valeur  du 2ème élément */

&tab[1]         /*Adresse du 2ème élément */
tab[1]          /*Valeur  du 2ème élément */
```

Tableaux (dynamiques) en C : Création

```c
# inlude < stdio.h>
# inlude < stdlib.h>

int main (){
      size_t size = 200; // taille du tableau
      double * tab ; // pointeur sur la premiere case du tableau

      tab = ( double *) malloc ( sizeof ( double ) * size );
      if(tab == NULL )
            return EXIT_FAILURE ;
      ...

      free ( tab )
      return EXIT_SUCCESS ;
}
```

Tableaux en C : Lecture / Ecriture

```
void arraySet ( int i, double v, double * tab , size_t
size ){
    if(i < size )
        tab [i] = v;
}

double arrayGet (int i, double * tab , size_t size ){
    if(i < size )
        return tab [i];
    return 0;
}
```

Tableaux en C : Insertion

```
void arrayPushBack ( double v, double ** tab , size_t * size ){
	int i;

	// Allocation du nouveau tableau
	double * tempTab = ( double *) malloc ( sizeof ( double ) * (* size +
1));

	// Recopie du tableau
	for (i=0 ; i < * size ; i ++)
		tempTab [i] = (* tab )[i];

	tempTab [* size ] = v; // insertion
	free (* tab ); // Liberation de l'ancien

	// mise a jour des pointeurs
	(* size )++;
	* tab = tempTab ;
}
```

Tableaux en C : Suppression

```
void arrayDeleteElem ( int i, double ** tab , size_t *
size ){

    ......

}
```

Liste chaînée : définition

- Une liste linéaire est définie par une suite de cellules.
- Chaque cellule est associée à une adresse dans la mémoire.
- Une cellule est définie par deux champs :
  - le premier champs indique le contenu de la cellule,
  - le deuxième champs est une référence (pointeur) vers une autre cellule.

- TYPE Cellule de T = structure

  info : T

  suivant : RefCellule de T

- TYPE Refcellule de T = ↑Cellule de T

Liste chaînée : Gestion dynamique de la mémoire

- On dispose des procédures suivantes :
- **nouveau(L)** crée une cellule et stocke une référence de type RefCellule de T dans L. Cette cellule va contenir un élément de type T. Le champs suivant est Affecté à NULL par default.
- **laisser(L)** détruit une cellule dont la référence est donnée dans L.

- Il suffit d'avoir la référence de la première cellule de la liste pour pouvoir manipuler la liste entière.

Liste chaînée en C : Création

```
typedef struct { double data ; node * next ;} node ;

int main (){
        node * head ; // pointeur sur la tete

        head = NULL ; // Initialisation de la tete

        // Insertion d'un noeud en tete
        node * tmp = ( node *) malloc ( sizeof ( node ) );
        linkedListInsertBefore (& head , tmp );

        // Insertion d'un noeud en queue
        tmp = ( node *) malloc ( sizeof ( node ) );
        node * queue = linkedListGetTail ( head );
        linkedListInsertAfter (queue , tmp );

        linkedListFree ( head );
        return EXIT_SUCCESS ;
}
```

Liste chaînée en C : Lecture / Ecriture

```
node * linkedListGetNode ( int i, node * head ){
    ..........
}

node * linkedListGetTail ( node * head ){
    .....
}
```

Liste chaînée en C : Insertion

```
void linkedListInsertBefore ( node ** thisNode , node * newNode )
{
     .....
}

void linkedListInsertAfter ( node * thisNode , node * newNode )
{
     .....
}
```

Pile : Définition

- Structure qui permet d’empiler des éléments dans un container.
- La pile est dite vide si elle ne contient aucun élément.
- Un nouveau élément est toujours ajouté au sommet.
- Pour supprimer un élément d’une pile on supprime toujours le sommet.

Pile : Primitives de traitement de pile

- **initPileVide :** est une procédure qui permet d'initialiser une pile, à la suite de cette opération la pile est vide (elle ne contient aucun élément).
- **pilevide :** est une fonction booléenne qui permet de tester si une pile est vide ou non.
- **empiler :** est une procédure qui permet d'ajouter un élément au sommet de la pile.
- **sommet :** est une fonction qui permet d'accéder à l'élément au sommet de pile
- **dépiler :** est une procédure qui permet de supprimer l'élément qui est au sommet de la pile si la pile n'est pas vide.

Pile : Représentation contiguë

- Un tableau appelé pile pour représenter une pile.
- La variable sommet contiendra le rang de l'élément qui est au sommet.
- Quand la pile est vide sommet vaut 0.
- dimpile est une variable contenant la taille maximale

```
type TPILE de T = structure
        pile     : TABLEAU de T
        sommet   : ENTIER
        dimpile  : ENTIER
```

Pile : Représentation contiguë

PROCEDURE CreerPile(ES p : TPILE de T, E n :ENTIER)
  $p.dimpile \leftarrow n$

PROCEDURE initPileVide(ES p : TPILE de T)
  $p.sommet \leftarrow 0$

FONCTION pilePleine(p : TPILE de T) :BOOLEEN
RETOURNER p.sommet=p.dimpile

FONCTION sommetPile(p : TPILE de T) :T
RETOURNER p.pile[p.sommet]

Pile : Représentation contiguë

```
PROCEDURE empiler(ES p : TPILE de T, E val :T, S
    possible :BOOLEEN)
    SI pilePleine(p) ALORS
        possible ← FAUX
    SINON
        p.sommet ← p.sommet + 1
        p.pile[p.sommet] ← val
        possible ← VRAI
    FIN SI
```

Pile : Représentation contiguë

```
PROCEDURE depiler(ES p : TPILE de T, S
    possible :BOOLEEN)
    SI pileVide(p) ALORS
        possible ← FAUX
    SINON
        p.sommet ← p.sommet − 1
        possible ← VRAI
    FIN SI
```

Pile : Représentation chaînée

- Représenter une pile à l’aide d’une liste chaînée.
- Pas de limite maximale sur longueur de la liste.
- La variable pile contient la référence de la première cellule dans la liste.

Pile : Représentation chaînée

PROCEDURE initPileVide(ES p : RefCellule de T)
  $p \leftarrow nil$

FONCTION pileVide(p : RefCellule de T) :BOOLEEN
RETOURNER p=nil

FONCTION sommetPile(p : RefCellule de T) :T
RETOURNER $p\uparrow .info$

Pile : Représentation chaînée

```
PROCEDURE empiler(ES p : RefCellule de T, E val :T)
    insererTete(p,val)
```

```
PROCEDURE depiler(ES p : RefCellule de T, S
    possible :BOOLEEN)
    SI pileVide(p) ALORS
        possible ← FAUX
    SINON
        suppTete(p)
        possible ← VRAI
    FIN SI
```

Pile : Représentation chaînée

En tête

PROCEDURE inserer Tete (ES liste : RefCellule de T, E V : T)
VARIABLES l : RefCellule de T

nouveau(*l*)
l ↑ *.info* ← *V*
l ↑ *.suivant* ← *liste*
liste ← *l*

Pile : Représentation chaînée

De la tête

PROCEDURE supptête (ES liste :RefCellule de T)
VARIABLES l : RefCellule de T
    l ← *liste*
    liste ← *l* ↑ *.suivant*
    laisser(l)

File d’attente

- Une file d’attente est un container d’éléments qui obéit au comportement suivant : le premier arrivé est le premier servi.
- Un exemple d’une file d’attente sera la queue devant un guichet.
- Les primitives de traitement d’une file sont :
  - **initFileVide** est une procédure qui permet d’initialiser une file, à la suite de cette opération la file est vide.
  - **filevide** est une fonction booléenne qui permet de tester si une file est vide ou non.
  - **premierFile** est une fonction qui retourne le premier élément de la file.
  - **ajouter** est une procédure qui permet d’ajouter un élément à la fin de la file.
  - **supprimer** est une procédure qui permet de supprimer le premier élément de la file si la file n’est pas vide.

Exercice

- Écrire une procédure qui effectue la Suppression de l'élément *elem* d'une liste linéaire.
- Nous supposons l'existence d'une fonction *refEA* qui prend en entrée deux paramètres qui sont la *liste* en question et l'élément *elem* est une fonction qui renvoie la référence de la cellule qui se trouve juste avant celle contenant l'élément recherché.

```
PROCEDURE suppressionE (liste : RefCellule de T, elem : T)
```

Exercice : Solution

```
PROCEDURE suppressionE (ES liste : RefCellule de T, E elem : T)
	VAR
		l : RefCellule de T
		e : RefCellule de T

	l <- refEA(liste, elem)
	SI (l <> nil) ALORS
		SI (l = liste ET l^.elem = elem)
			suppTete (liste, elem)
		SINON
			e <- l^.suivant
			l^.suivant <- e^.suivant
			laisser(e)
		FIN SI
	FIN SI
FIN PROCEDURE
```

Structure de Données

Les arbres

Arbre

1. **Définition, Notions et représentation**
2. **Primitives d’accès pour les arbres binaires**
3. **Niveau, Taille et Hauteur**
4. **Algorithmes de parcours**
5. **Recherche, Insertion, suppression**
6. **Arbres binaires complets et tas**
7. **Tri par tas**

Arbre : Définition

- Structure de données arborescente acyclique orientée possédant une
- unique racine
- Un arbre est un ensemble de nœuds tels que :
  - Chaque **nœud** possède un certain nombre de fils.
  - Lorsqu'un nœud n'a pas de fils il est appelé **nœud terminal** ou **feuille**.
  - Un arbre contient un et un seul nœud qui n'a pas de père c'est la **racine**.
  - L'arc qui relie deux nœuds est appelé branche.
  - Tous les nœuds (sauf la racine) ont un seul père.

Arbre : Définition

- Lorsqu'un arbre admet, pour chaque nœud, au plus n fils l'arbre est appelé n-aire.
- Pour les arbres binaires on parlera de fils gauche et de fils droit ainsi que de sous-arbre gauche et de sous-arbre droit.

On peut définir un arbre d'une façon récursive :

- un arbre est soit vide,
- soit il est constitué d'un élément auquel sont chaînés un ou plusieurs arbres.

Représentation d'un arbre binaire

- type nœudB de T= structure

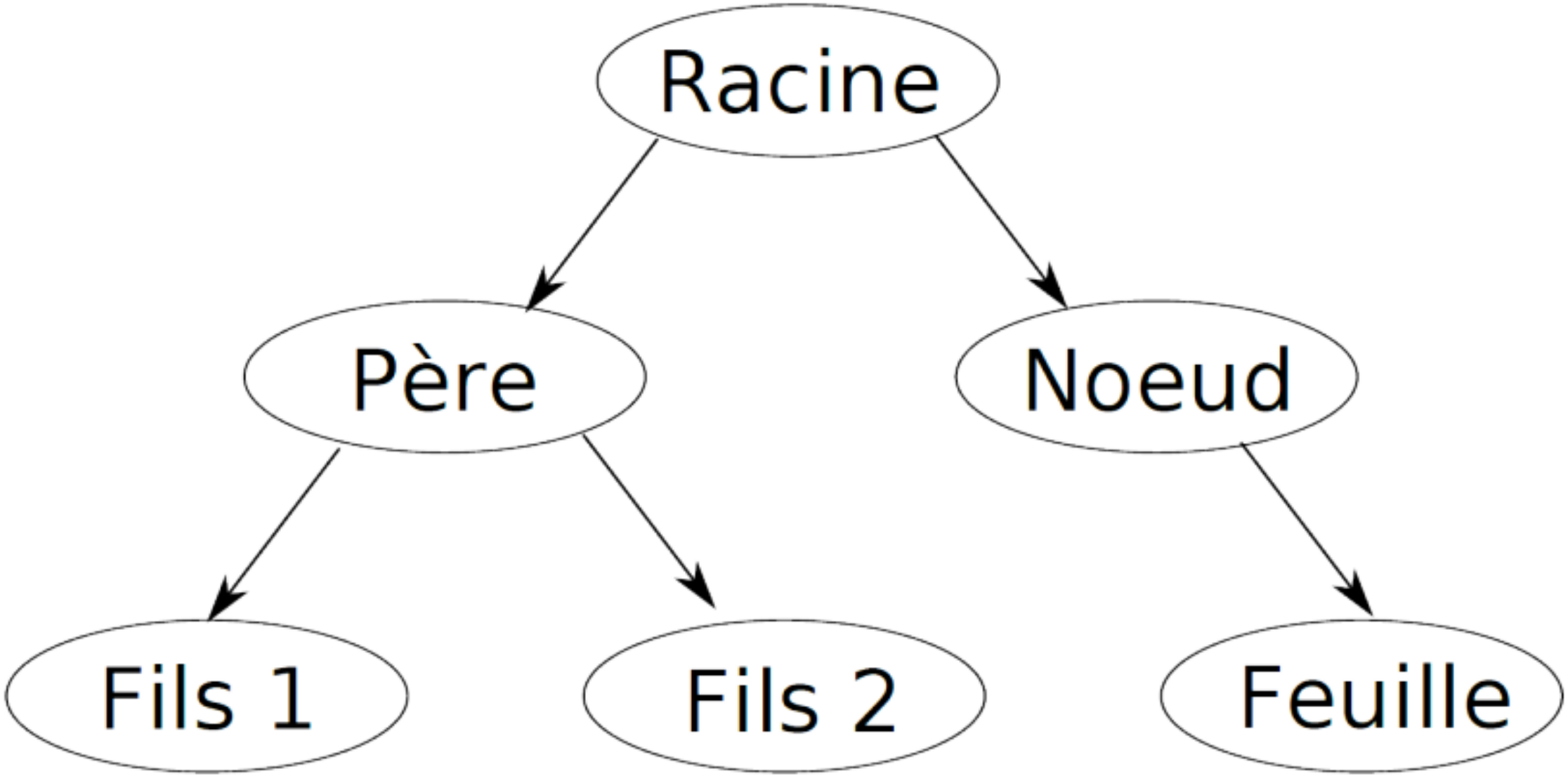
```
        info : T

        gauche : RefCellule de nœudB de T

        droite : RefCellule de nœudB de T
```

- type arbreB de T= RefCellule de nœudB de T

Représentation d'un arbre binaire



Primitives d’accès

- **CreerA(T)** T étant un type donné, creerA(T) crée une cellule contenant trois champs, et retourne une valeur de type arbreB de T.
- **DetruireA(R**) détruit la cellule. R est de type arbreB de T.
- **Affectinfo(R,V)** R est de type arbreB de T et V est de type T.
- **AffectG(R,G)** affecte le fils gauche, R et G sont de types arbreB de T.
- **AffectD(R,D)** R et D sont de types arbreB de T
- **info(R)** récupère le contenu d’un nœud et retourne une valeur de type T.
- **gauche(R)** retourne la référence du fils gauche (une valeur de type arbreB de T).
- **droite(R)** retourne la référence du fils droit.

Arbre : Quelques définition

Niveau d’un nœud

- Le niveau de la racine est égal à 1.
- Le niveau de chaque nœud est égal au niveau de son père plus 1

Taille d’un arbre

- La taille de l’arbre est défini par le nombre de ses nœuds

Hauteur d’un arbre

- La hauteur d’un arbre est égale au maximum des niveaux des feuilles.

Arbres binaires complets et équilibrés

Arbre binaire parfait

- Si chaque nœud non terminal admet deux descendants et si toutes les feuilles sont au même niveau on dit que l'arbre est complet.
- La taille d'un arbre complet est $2^k - 1$ où k est le niveau des feuilles.

Arbre binaire équilibrés

- Le facteur d'équilibre d'un nœud est égal à la hauteur du sous arbre gauche moins la hauteur du sous arbre droit.
- Le facteur d'équilibre de chaque sous-arbre est associé à sa racine.
- Un arbre est dit équilibré si pour chaque nœud p nous avons : $|$ *facteurEquilibre(p)* $| \leq 1$

Arbres dégénérés et ordonnés

Arbre dégénérés

- Un arbre est dit dégénéré si tous les nœuds de cet arbre ont au plus un descendant. Un arbre dégénéré est équivalent à une liste chaînée

Arbre binaire ordonné

- Soit ≤ une relation d'ordre complet sur l'ensemble des éléments de type T :
- On dit qu'un arbre binaire est ordonné si pour tout nœud *p* de l'arbre :
  - tous les éléments du sous-arbre gauche sont strictement inférieurs à l'élément contenu dans *p*,
  - tous les éléments du sous arbre droit sont supérieurs ou égaux à l'élément contenu dans *p*

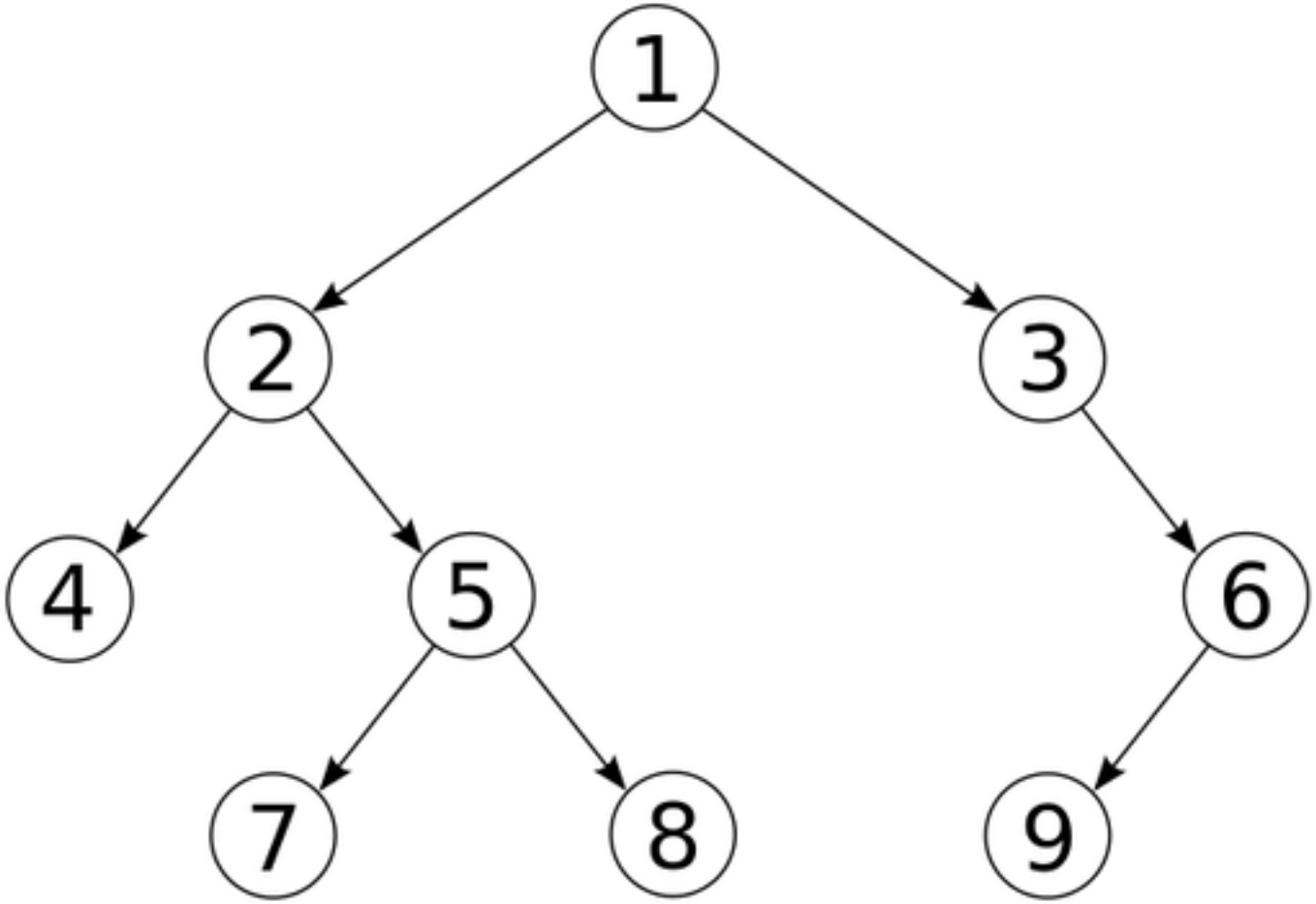
Algorithmes de parcours

```
PROCEDURE prefixe(racine : arbreB de T)
  SI racine <> nil ALORS
    traiter(info(racine))
    prefixe(gauche(racine))
    prefixe(droite(racine))
  FIN SI
```

```
PROCEDURE infixe(racine : arbreB de T)
  SI racine <> nil ALORS
    infixe(gauche(racine))
    traiter(info(racine))
    infixe(droite(racine))
  FIN SI
```

```
PROCEDURE postfixe(racine : arbreB de T)
  SI racine <> nil ALORS
    postfixe(gauche(racine))
    postfixe(droite(racine))
    traiter(info(racine))
  FIN SI
```

Algorithmes de parcours



Dans cet arbre binaire,

- Rendu du parcours préfixe : 1, 2, 4, 5, 7, 8, 3, 6, 9
- Rendu du parcours postfixe : 4, 7, 8, 5, 2, 9, 6, 3, 1
- Rendu du parcours infixe : 4, 2, 7, 5, 8, 1, 3, 9, 6

Quelques algorithmes

Taille d'un arbre

```
FONCTION taille( racine : arbreB de T) : ENTIER
   SI racine = nil ALORS
RETOURNER    0
   SINON
RETOURNER    1+taille(gauche(rac))+taille (droite(rac))
   FIN SI
```

Vérifier si un nœud est une feuille

```
FONCTION feuille(racine : arbreB de T) : BOOLEEN
RETOURNER gauche(racine)=nil ET droite(racine)=nil
```

Recherche dans un arbre binaire

```
FONCTION rech(racine : arbreB de T, val :T) : BOOLEEN
VARIABLES tmp : BOOLEEN
   SI racine=nil ALORS
      tmp ← FAUX
   SINON
      SI info(racine)=val ALORS
         tmp ← VRAI
      SINON
         tmp ← FAUX
         rech(gauche(racine), val)OUrech(droite(racine), val)
      FIN SI
   FIN SI
RETOURNER tmp
```

Création d'une feuille

```
FONCTION creerFeuille(E elem :T) : arbreB de T
VARIABLES tmp : arbreB de T
    tmp ← CreerA(T)
    Affectinfo(tmp,E)
    AffectG(tmp,nil)
    AffectD(tmp,nil)
RETOURNER tmp
```

Insertion dans un arbre binaire ordonné

```
PROCEDURE inserer(ES racine : arbreB de T, E elem : T)
  SI racine<>nil ALORS
    SI elem <info(racine) ALORS
      SI gauche(racine)=nil ALORS
        AffectG(racine,creerFeuille(elem))
      SINON
        inserer(gauche(racine),elem)
      FIN SI
    SINON
      SI droite(racine)=nil ALORS
        AffectD(racine,creerFeuille(elem))
      SINON
        inserer(droite(racine),elem)
      FIN SI
    FIN SI
  FIN SI
```

Suppression de la racine dans un arbre binaire ordonné

```
PROCEDURE suppnoeud(ES n : arbreB de T)
VARIABLES p,q : arbreB de T
   p ← n
   SI droite(n) <> nil ALORS
      q ← droite(n)
      TANTQUE gauche(q) <> nil FAIRE
         q ← gauche(q)
      FIN TANTQUE
      AffectG(q,gauche(p))
      n ← droite(n)
   SINON
      n ← gauche(n)
   FIN SI
   DetruireA(p)
```

Représentation contiguë d'un arbre binaire complet

Un arbre binaire complet peut être représente par un tableau de taille $2^k - 1$ où k étant la hauteur de l'arbre. Le tableau vérifie les propriétés suivantes :

- `tab[1]` est l'information contenue dans la racine.
- `tab[2 * i]` est l'information associée au fils gauche du nœud *i*.
- `tab[2 * i + 1]` est l'information associée au fils droit du nœud *i*

Parcours préfixe, représentation contiguë

```
PROCEDURE prefixe (E tab : TABLEAU de T, E n, racine :
   ENTIER)
   SI racine < n  ALORS
      afficher(tab[racine])
      prefixe (tab,n,2*racine)
      prefixe (tav,n,2*racine+1)
   FIN SI
```

Définition d'un tas

Un tas est un arbre binaire dans le quel tout nœud non terminal n vérifie la propriété suivante :

$$\text{info}(n) \geq \max(\text{info}(\text{gauche}(n)), \text{info}(\text{droite}(n)))$$

Si l'arbre est représenté d'une façon contiguë, on obtient :

- $\text{tab}[i] \geq \text{tab}[2 * i]$ ssi $2*i \leq n$
- $\text{tab}[i] \geq \text{tab}[2 * i + 1]$ ssi $2*i + 1 \leq n$

Propriété d'un tas

- Dans un tas, la valeur contenue dans la racine correspond au maximum des valeurs contenues dans l'arbre.

Tri par tas

- Transformation d'un tableau en tas,
- Utilisation du tas pour trier le tableau
- Complexité de $O(NlogN)$

Construction d'un tas

```
PROCEDURE construiretas (ES tab : TABLEAU de T, E n :
   ENTIER)
VARIABLES i : ENTIER
   i ← n div 2
   TANTQUE i> 0 FAIRE
      constas(tab,i,n)
      i ← i − 1
   FIN TANTQUE
```

Construction d'un tas

Construire un tas à partir d'un nœud donné

```
PROCEDURE constas(ES tab : TABLEAU de T, E i, n : ENTIER)
VARIABLES filsmax : ENTIER
  filsmax ← 2 * i
  SI filsmax ≤ n ALORS
    SI filsmax < n ALORS
      SI tab[filsmax + 1] > tab[filsmax] ALORS
        filsmax ← filsmax + 1
      FIN SI
    FIN SI
    SI tab[filsmax] > tab[i] ALORS
      permut(tab, filsmax,i)
      constas(tab,filsmax,n)
    FIN SI
  FIN SI
```

Heap Sort

Heap sort

```
PROCEDURE tritas (ES tab : TABLEAU de T, E n : ENTIER)
VARIABLES nb : ENTIER
    construiretas(tab,n)
    permut(tab,1,n)
    nb ← n − 1
    TANTQUE nb > 1 FAIRE
        constas(tab,1,nb)
        permut(tab,1,nb)
        nb ← nb − 1
    FIN TANTQUE
```

Exercices

1. Écrire une fonction qui calcule le nombre de feuilles dans un arbre binaire.

2. Écrire une fonction booléenne qui recherche une valeur donnée de type T dans un arbre binaire ordonné.

3. Écrire une procédure qui effectue la suppression d'une valeur donnée de types T d'un arbre binaire ordonné

Exercices : Nombre de Feuilles

```
FONCTION nbFeuille(racine : arbreB de T) : ENTIER
    VARIABLES tmp : ENTIER

    SI racine = nil ALORS
        tmp <- 0
    SINON
        SI feuille(racine) ALORS
            tmp 1
        SINON
            tmp nbF euille(gauche(racine))
                + nbFeuille(droite(racine))
        FIN SI
    FIN SI
RETOURNER tmp
```

Exercices : Recherche

```
FONCTION rechO(racine : arbreB de T, val :T) : BOOLEEN
      VARIABLES tmp : BOOLEEN

      SI racine=nil ALORS
            tmp <- FAUX
      SINON
      SI info(racine)=val ALORS
            tmp <- VRAI
      SINON
            SI info(racine) > val ALORS
                  tmp <- rechO(gauche(racine))
            SINON
                  tmp <- rech(droite(racine))
            FIN SI
      FIN SI
      FIN SI
RETOURNER tmp
```

Exercices : Suppression

```
PROCEDURE supprim (ES racine : arbreB de T, val :T)
     VAR r : arbreB de T
     SI racine <> nil ALORS
          SI info(racine)=val ALORS
               suppnoeud(racine)  // Voir cours
          SINON
            SI info(racine)>val ALORS
               r <- gauche(racine)
               supprim(r,val)
               AffectG(racine,r)
            SINON
               r <- droite(racine)
               supprim(r,val)
               AffectD(racine,r)
            FIN SI
          FIN SI
     FIN SI
```

Structure de Données

Les fichiers séquentiels

Définition d’un fichier séquentiel

- Un fichier séquentiel sur un ensemble V est une suite finie d’éléments de V , munie de certaines propriétés.
- Le fichier correspond à un moyen de stockage de l’information.
- Pour manipuler un fichier, nous manipulons deux entités

le nom de fichier qui sera représenté par une valeur de type chaîne de caractère.

La variable représentant la tête de la lecture/ou écriture cette variable contient à un moment donné une copie de l’élément de fichier où pointe la tête de lecture ou d’écriture. Cette variable doit être de type **flot de T**

Primitives d'accès

- **`fdf(f)`** est une fonction qui prend la valeur VRAI si on atteint la marque de fin de fichier. f est une variable de type flot de T.
- **`contenu(f)`** est une fonction qui retourne une valeur de type T correspondant au contenu du flot. Dans le cas où on a atteint la fin de fichier le contenu de f est indéfini.
- **`affectF(f,V)`** est une procédure qui affecte le contenu d'un flot f de type T par la valeur V de type T.
- **`relire(nom de fichier,f)`** rend possible l'accès au premier élément du fichier en lecture. A la suite de cette opération, f contiendra une copie du premier élément du fichier.

Primitives d'accès

- **`prendre(f)`** fait avancer la tête de lecture et copie l'élément pointé par la tête de lecture dans f.
- **`fermer(f)`** libère la variable f.
- **`reecrire(nom de fichier,f)`** ouvre un fichier en écriture, `fdf(f)` est VRAI.
- **`mettre(f)`** mettre le contenu de f dans le fichier est avance la tête d'écriture.
- **`reajouter(nom de fichier,f)`** ouvre un fichier en écriture et la tête d'écriture pointe sur le fin de fichier.

Primitives d’accès

Contraintes d’utilisation des primitives

- **R1**

On peut écrire dans un fichier seulement si il est ouvert avec `reecrire` ou `reajouter`.

- **R2**
- On peut lire d’un fichier seulement s’il est ouvert avec `relire`.

Exemple d'algorithme

La somme des entiers contenus dans un fichier

```
FONCTION somme (nom : CHAINE de CARACTERES) :
   ENTIER
VARIABLES s : ENTIER
   f : flot de ENTIER
   relire(nom,f)
   s ← 0
   TANTQUE non fdf(f) FAIRE
      s ← s + contenu(f)
      prendre(f)
   FIN TANTQUE
   fermer(f)
```

Construction d'un ficher à partir d'un tableau

```
PROCEDURE tableautofic (E nom : CHAINE de CARACTERES,
   E tab : TABLEAU de T, E n : ENTIER )
VARIABLES i : ENTIER
   f : flot de T
   reecrire(nom,f)
   i ← 1
   TANTQUE i ≤n FAIRE
      affectF(f,tab[i])
      mettre(f)
      i ← i + 1
   FIN TANTQUE
   fermer(f)
```

Accès à un élément dans un fichier

Par la position k

```
PROCEDURE accesK (E nom : CHAINE de CARACTERES, E k :ENTIER, S valk : T, S trouve : BOOLEEN)
VARIABLES f : flot de T
   i : ENTIER
   relire(nom,f)
   i ← 1
   trouve ← FAUX
   TANTQUE non fdf(f) ET i < k FAIRE
      prendre(f)
      i ← i + 1
   FIN TANTQUE
   SI non fdf(f) ALORS
      valk ← contenu(f)
      trouve ← VRAI
   FIN SI
   fermer(f)
```

Accès à un élément dans un fichier

Par contenu

```
PROCEDURE acces (E nom : CHAINE de CARACTERES, E val : T, S trouve : BOOLEEN)
VARIABLES f : flot de T
   c : T
   relire(nom,f)
   c ← contenu(f)
   trouve ← FAUX
   TANTQUE non fdf(f) ET c <> val FAIRE
      prendre(f)
      c ← contenu(f)
   FIN TANTQUE
   fermer(f)
   trouve ← c = val
```

Eclatement d'un fichier d'entiers en deux fichiers

Eclatement d'un fichier d'entiers en deux fichiers : un fichier contenant les entiers pairs et un autre contenant les entiers impairs

```
PROCEDURE eclatementC (E nom,pair,impair : CHAINE de CARACTERES)
VARIABLES f,fpair,fimpair : flot de ENTIER
   relire(nom,f)
   reecrire(pair,f1)
   reecrire(impair,f2)
   TANTQUE non fdf(f) FAIRE
      SI pair(contenu(f)) ALORS
         affectF(f1,contenu(f))
         mettre(f1)
      SINON
         affectF(f2,contenu(f))
         mettre(f2)
      FIN SI
      prendre(f)
   FIN TANTQUE
   fermer(f)
   fermer(f1)
   fermer(f2)
```

Fusion de deux fichiers triés

```
PROCEDURE fusion (E fic,fic1,fic2 : CHAINE de CARACTERES)
VARIABLES f,f1,f2 : flot de T
  reecrire(fic,f)
  relire(fic1,f1)
  relire(fic2,f2)
  TANTQUE non fdf(f1) ET non fdf(f2) FAIRE
    SI contenu(f1) < contenu(f2) ALORS
      affectF(f,contenu(f1))
      mettre(f)
      prendre(f1)
    SINON
      affectF(f,contenu(f2))
      mettre(f)
      prendre(f2)
    FIN SI
  FIN TANTQUE
  TANTQUE non fdf(f1) FAIRE
    affectF(f,contenu(f1)) mettre(f) prendre(f1)
  FIN TANTQUE
  TANTQUE non fdf(f2) FAIRE
    affectF(f,contenu(f2)) mettre(f) prendre(f2)
  FIN TANTQUE
  fermer(f) fermer(f1) fermer(f2)
```

Copie d'une liste linéaire dans un fichier

```
PROCEDURE listToFic (E fic : CHAINE de CARACTERES, E l :
   RefCellule de T)
VARIABLES f : FLOT de T
   l1 : RefCellule de T
   l1 ← l
   reecrire(fic,f)
   TANTQUE l1 <> nil FAIRE
      AffectF(f,l1 ↑ .info)
      mettre(f)
      l1 ← l1 ↑ .suivant
   FIN TANTQUE
   fermer(f)
```

Copie d'un fichier dans une liste linéaire

```
PROCEDURE fictoList (E fic : CHAINE de CARACTERES, S l : RefCellule de T)
VARIABLES f : FLOT de T
   l1,prec : RefCellule de T
   l1 ← nil
   l ← nil
   relire(fic,f)
   SI non fdf(f) ALORS
      nouveau(l)
      l ↑ .info ← contenu(f)
      l ↑ .suivant ← nil
      prec ← l
      prendre(f)
   FIN SI
   TANTQUE non fdf(f) FAIRE
      nouveau(l1)
      l1 ↑ .info ← contenu(f)
      l1 ↑ .suivant ← nil
      prec ↑ .suivant ← l1
      prec ← l1
      prendre(f)
   FIN TANTQUE
   fermer(f)
```

Insertion d'un élément dans un fichier

En utilisant une liste linéaire

```
PROCEDURE FLInsertK (E fic : CHAINE de CARACTERES, E
   k : ENTIER, E elem : T, S possible : BOOLEEN )
VARIABLES l : RefCellule de T
   fictoList(fic,l)
   insererK(l,k,elem,possible)
   SI possible ALORS
      listtoFic(l,fic)
   FIN SI
   DetruireL(l)
```

DetruireL

```
PROCEDURE DetruireL(ES l : RefCellule de T)
VARIABLES C : RefCellule de T
   TANTQUE l<>nil FAIRE
      C ← l
      l ← l↑.suivant
      laisser(C)
   FIN TANTQUE
```

Rappel de InsererK

Inserer elem dans la liste linéaire à la position k

```
PROCEDURE insererK (ES liste : RefCellule de T, E k : ENTIER, E V : T, S possible : BOOLEEN)
VARIABLES l,preced : RefCellule de T
   possible ← FAUX
   SI k=1 ALORS
      insererTete(liste,V)
      possible ← VRAI
   SINON
      preced ← refk(liste, k − 1)
      SI preced <> nil ALORS
         nouveau(l)
         l ↑ .info ← V
         l ↑ .suivant ← preced ↑ .suivant
         preced ↑ .suivant ← l
         possible ← VRAI
      FIN SI
   FIN SI
```

Suppression d'un élément d'un fichier

Suppression d'un élément en utilisant un autre fichier

```
PROCEDURE FsuppE (E fic,ficS : CHAINE de CARACTERES, E elem : T, S possible : BOOLEEN )
VARIABLES f1,f2 : FLOT de T
   relire(fic,f1)
   reecrire(ficS,f2)
   possible ← FAUX
   TANTQUE non fdf(f1) FAIRE
      SI contenu(f1)=elem ET non possible ALORS
         possible ← VRAI
      SINON
         AffectF(f2, contenu(f1))
         mettre(f2)
      FIN SI
      prendre(f1)
   FIN TANTQUE
   fermer(f1)
   fermer(f2)
```

Implémentation d'une file avec deux piles

- Une file de taille n à l'aide de deux piles de taille n.
- Le sommet de la première pile correspond au premier de la file,
- Le sommet de la seconde pile correspond au dernier de la file.

Rappel de la représentation contiguë d'une pile

- type TPILE de T = structure

  pile : TABLEAU de T

  sommet : ENTIER

  dimpile : ENTIER

Implémentation d'une file avec deux piles

Structure File2Piles

- type File2Piles de T = structure
  tete : TPILE de T
  dernier : TPILE de T

```
PROCEDURE copieFinFile ( ES f :File2Piles de de T)
  TANTQUE non pileVide(f.dernier) FAIRE
    empiler(f.premier,sommetPile(f.dernier),possible),
    depiler(f.dernier)
  FIN TANTQUE
```

Implémentation des primitives

```
PROCEDURE CreerFile(ES f : TFILE de T, E n :ENTIER)
    CreerPile(f.tete)
    CreerPile(f.dernier)
PROCEDURE initFileVide(ES f : File2Piles de T)
    initPileVide(f.tete)
    initPileVide(f.dernier)
FONCTION fileVide(E f : File2Piles de T) :BOOLEEN
RETOURNER pileVide(f.tete) ET pileVide(f.dernier)
FONCTION filePleine (E f : File2Piles de T) :BOOLEEN
RETOURNER pilePleine(f.dernier)
```

Implémentation des primitives

```
FONCTION premierFile(ES f : File2Piles de T) :T
   SI pileVide(f.tete) ALORS
      copieFinFile(f)
   FIN SI
RETOURNER sommetPile(f.tete)
```

```
PROCEDURE ajouter(ES File2Piles de T, E val :T, S
   possible :BOOLEEN)
   SI filePleine(f) ALORS
      possible ← FAUX
   SINON
      empiler(f.dernier,val ,possible)
   FIN SI
```

Implémentation des primitives

```
PROCEDURE supprimer(ES File2Piles de T, S
    possible :BOOLEEN)
VARIABLES
    SI pileVide(f.tete) ALORS
        copieFinFile(f)
    FIN SI
    depiler(f.tete,possible)
```

Exercices

Écrire une fonction qui permet d'évaluer une expression postfixée lue dans un vecteur de chaînes de caractères qui se termine par "**#**". Nous disposons des fonctions suivantes :

– la fonction booléenne `operation` qui retourne vraie si la chaîne de caractères en question est une opération par exemple `operation("+")= vrai`,

– la fonction `oper` qui prend en argument un réel, une chaîne de caractères et un réel et qui rend le réel résultant de l'opération par exemple `oper(3.0,"+",4.2)` renvoie le réel `7.2`,

– la fonction `convertirNum` qui convertit une chaîne de caractères en réel par exemple `convertirNum("3.9")` renvoie le réel `3.9`.

La fonction que vous devez écrire doit prendre en paramètre un vecteur de caractères et doit retourner un réel.

Par exemple le résultat que doit renvoyer cette fonction pour le vecteur

`V=("3", "5", "+", "7","12","+","*","3,"+","#")` est `155`.

Exercices : Solution

```
FONCTION evalPost(tab : TABLEAU de Chaines de caractères) : REEL
        VARIABLES           P : RefCellule de TPILE de REEL
                            i : ENTIER
                            droite,gauche,num : REEL
                            possible : BOOLEEN
        initPileVide(P)
        i <- 1
        TANTQUE tab[i]<>'#' FAIRE
                SI non operation(tab[i]) ALORS
                            convertirNum(tab[i],num)
                            empiler(P,num)
                SINON
                            droite sommetPile(P)
                            depiler(P,possible)
                            gauche sommetPile(P)
                            depiler(P,possible)
                            empiler(P,oper(gauche,tab[i],droite))
                FIN SI
                i <- i + 1
        FIN TANTQUE
        num <- sommetPile(P)
        depiler(P,possible)
        RETOURNER num
```

Exercices

Écrire un algorithme qui permet d'éclater un fichier en deux autres fichiers selon les rangs des éléments : les éléments ayant des rangs pairs seront dans un fichier et ceux ayant des rangs impairs seront dans un autre.

Exercices : Solution

```
PROCEDURE eclatementR (E nom,pair,impair : CHAINE de CARACTERES)
        VARIABLES          f,fpair,fimpair : flot de T          rangpair : BOOLEEN
        relire(nom,f)      reecrire(pair,fpair)      reecrire(impair,fimpair)
        rangpair <- FAUX
        TANTQUE non fdf(f) FAIRE
                SI rangpair ALORS
                        affectF(fpair,contenu(f))      mettre(fpair)      rangpair <- FAUX
                SINON
                        affectF(fimpair,contenu(f))    mettre(fimpair)    rangpair <- VRAI
                FIN SI
                prendre(f)
        FIN TANTQUE
        fermer(f)          fermer(fpair)          fermer(fimpair)
```