Cours 2

Implémentation des SDA

Implémenter une SDA, cela veut dire : écrire des fonctions que l'on pourra utiliser ensuite (soit en les recopiant, soit comme des fonctions de bibliothèque) dans des programmes qui ont besoin de ces structures. L'implémentation des SDA dans un langage typé comme C ou Java est paramétrée par le type de données. On utilisera pour l'exemple des float. Dans certains exemples, ce n'est pas le plus réaliste, mais pour commencer c'est plus simple. La technique serait la même avec n'importe quel type de taille fixe. Quand il s'agit de chaînes de caractères, tous les langages utilisent (en le masquant ou pas) des pointeurs de taille fixe sur des chaînes rangées ailleurs.

Pour implémenter, il faut faire des choix de programmation. Ils sont regroupés autour de trois idées différentes :
- la représentation des données
- le style de programmation pour utiliser les SDA
- la gestion de la mémoire

1 Implémentations dans des tableaux élémentaires

1.1 Implémentation des piles

On donne l'implémentation en C d'une pile dans un fichier à part. Cette technique permet de limiter la visibilité des variables globales au fichier, en les déclarant 'static' – elles ont des propriétés analogues aux variables membres privées des classes java.

1.1.1 Implémentation en taille bornée

```
#define MAXTAILLE 100
static float pile[MAXTAILLE];
static int sommet = -1 ;// le dernier écrit
void empiler(float f) {
    if(sommet == MAXTAILLE - 1){perror("dépassement de capacité de la
    pile") ; exit 1 ;}
    sommet++ ; pile[sommet]=f ;
    }
float dépiler(void) {
    if(est_vide()) {perror("essai de dépiler la file vide") ; exit 2 ;}
    sommet-- ; return (pile[sommet + 1]) ;
    }
int est_vide(void) {
    return(sommet == -1) ;
 }
```

1.1.2 Implémentation en taille variable

Si le langage permet de demander de la mémoire pendant l'exécution du programme (allocation dynamique), on peut éviter l'erreur "dépassement de capacité" en demandant de l'espace pour créer un tableau plus grand. En Java, l'espace est alloué par **new**. En C, on fait un **malloc()** : la fonction `void * malloc(int nbbyte)` renvoie un pointeur (sans type) sur une zone de *nbbyte* octets qui était libre et est maintenant attribuée au programme. En standard, si l'on veut un tableau tabfloat de nbfloat nombres de type float, on écrit : tabfloat = (float *) malloc(nbfloat * sizeof(float)).

(la taille du tableau, en octets, est le nombre de données (nbfloat) multiplié par la taille d'une donnée (sizeof(float)) ; le (float *) devant indique que les données dans le tableau seront des float, ce que le calcul de la taille n'impliquait pas obligatoirement). Il faut alors utiliser une fonction pour initialiser la pile.

```
#define MAXTAILLE 100
static float *pile ;
static int sommet = -1 ;// le dernier écrit
static int taille = MAXTAILLE ;
void init(void) {
    pile=(float*) malloc(taille * sizeof(float)) ;
    }
void empiler(float f) {
    if(sommet == taille - 1){ // double la taille de la pile
            float *newpile ; int i ;
            newpile=(float*) malloc(taille * 2 * sizeof(float)) ;
            for(i=0 ; i<taille ; i++) newpile[i] = pile[i] ;
            free(pile) ; pile = newpile ; taille = taille * 2 ;
    }
    sommet++ ; pile[sommet]=f ;
    }
float dépiler(void) {
    if(est_vide()) {perror("essai de dépiler la file vide") ; exit 2 ;}
    sommet-- ; return (pile[sommet + 1]) ;
    }
int est_vide(void) {
    return(sommet == -1) ;
    }
```

1.2 Implémentation des files

L'implémentation des files dans un tableau se fait avec deux variables qui indiquent la **tète** (là où on stocke) et la **queue** (la où on prélève) de la file. Au fur et à mesure que l'on stocke et que l'on prélève, la partie occupée du tableau avance (à la manière des vers que l'on dessine à l'écran). Quand on est à la fin du tableau, la tête du vers retourne au début, à condition que celui-ci ait été libéré par des opérations prélever(). Dans un tableau de taille MAXTAILLE, cela s'écrit tete = (tete + 1)%MAXTAILLE. Une file implémentée ainsi s'appelle une **file circulaire**.

```
/*  Fichier........: file.c
 * Description..: implémentation d'une file circulaire en C
 * tete et queue sont les adresses ou placer / prendre la prochaine
 * donnee a stocker / prélever. La file est vide quand tete == queue
 * et est pleine quand tete + 1 = queue  (en stockant un
 * élément, on retrouverait tete == queue, i.e. la file vide ...)
*/
#define MAXTAILLE 100
static float file[MAXTAILLE] ;
static int tete = 0, queue = 0 ;

void stocker(float f)  {int nextete ;
  nextete = (tete + 1)%MAXTAILLE ;
  if(nextete == queue) {perror("file pleine") ; exit(1) ; }
  file[tete] = f ; tete = nextete ;
} /* fin de stocker() */


float prelever()  { float f ; int nextqueue ;
  nextqueue = (queue + 1)%MAXTAILLE ;
```

```
    if(queue == tete) {perror("file vide") ; exit(2) ;}
    f = file[queue] ; queue = nextqueue ; return(f) ;
  } /* fin de prelever() */

  int est_vide() {
    return(tete == queue) ;
  }/* fin de est_vide() */
```

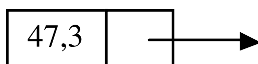
1.3 Implémentation naïve des listes

On peut toujours implémenter une liste dans un tableau élémentaire, et insérer en décalant les éléments qui suivent. C'est une méthode peu efficace (les opérations sont très lentes), mais qui permet d'utiliser de petites listes sans effort de programmation. Par exemple pour implémenter une liste de floats :

```
/* nanf() (nan = not a number) renvoie le code d'un float impossible */
#include <math.h>
#define MAXTAILLE 100
static float NULL ;
static int taille = 0 ;
static int curseur = -1 ;
static float liste[MAXTAILLE] ;

void rembobiner() {
  curseur = -1 ;  NULL = nanf("") ;
} /* fin de rembobiner() */

float voir_suivant() {
  if(curseur == taille - 1) return NULL ;
  return(liste[curseur + 1]) ;
} /* fin de voir_suivant() */

float suivant() {
  if(curseur == taille - 1) return NULL ;
  curseur++ ;   return(liste[curseur]) ;
} /* fin de suivant() */

void ajouter(float f)  {  int i ;
 if(taille==MAXTAILLE) {perror("plus de place dans la file") ; exit(1) ;}
  for(i = taille-1 ; i > curseur ; i--) liste[taille + 1=liste[taille] ;
  liste[curseur + 1] = f ; taille++ ;
} /* fin de ajouter() */

void supprimer()  { int i ;
  if(voir_suivant() == NULL) {
    perror("suppression aprËs la fin de liste") ; exit(2) ;}
  for(i=curseur + 1 ; i <taille-1 ; i++) {liste[i] = liste[i+1] ; }
  taille-- ;
} /* fin de supprimer() */
```

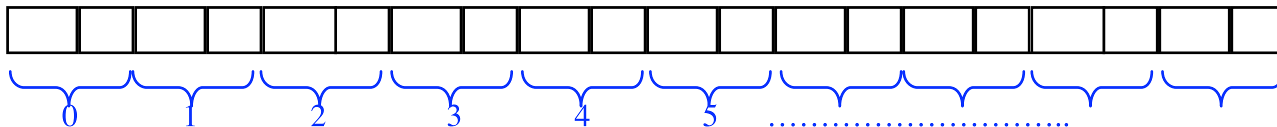
2 La représentation des listes chaînées

On suppose que l'on veut implémenter des listes de nombres à virgule (float). Le premier problème à résoudre est de représenter les cellules, c'est à dire une donnée plus un accès au suivant.



2.1 Tableau de données composées

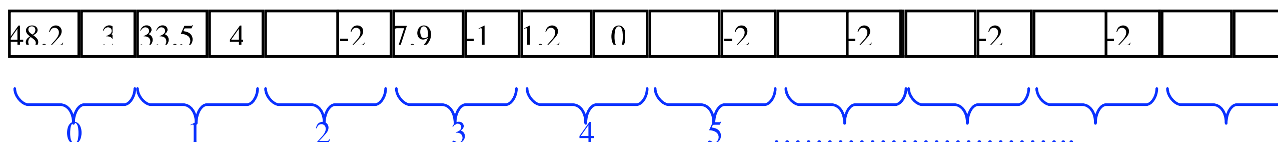
Choix 1 : on prend un langage capable de créer des types de données composées, par ex. C avec la construction **struct**. On utilisera donc une structure ***cell*** à 2 champs, *donnee* et *suivant*.
Choix 2 : on peut donner un maximum raisonnable sur le nombre de cellules à utiliser, et on réserve au lancement du programme l'espace nécessaire en un seul morceau (donc c'est un tableau)



Conséquence : on peut utiliser la position (l'indice) dans le tableau comme adresse. Donc dans cette implémentation, l'adresse est de type entier.

```
#define MAXCELL 3000
typedef int ADRCELL ;
typedef struct cell {float donnee ; ADRCELL suivant } CELL ;
CELL heapcel [MAXCELL] ;
```

Convention : la fin de liste (pas de suivant) est indiquée par une adresse impossible, ici -1
Exemple de liste :



Pour trouver le premier :

```
ADRCELL liste1 = 1 ; // la première cellule est à l'indice 1
```

Au démarrage du programme, toutes les cellules sont libres. Au fur et à mesure qu'on crée des listes, on prend des cellules libres ; on rend les cellules dont on n'a plus besoin.
Convention : on marque une cellule libre (inutilisée) en mettant son champs *suivant* à –2

Cette convention permet de placer plusieurs listes dans le même tableau.

2.2 Tableaux parallèles de données élémentaires

Quand on ne dispose que d'un langage très pauvre, sans types de données composées, la solution précédente n'est plus possible, mais il y a un palliatif : on crée deux tableaux parallèles, un pour la donnée, l'autre pour l'adresse. On continue à repérer la cellule par son indice, mais ses champs sont dans deux tableaux différents.

```
#define MAXCELL 3000
typedef int ADRCELL ;
float donnee[MAXCELL];
ADRCELL suivant[MAXCELL];
```

On utilise les mêmes conventions pour fin de liste et libre. Ci-dessous la même liste que dans 1.1, mais représentée autrement :

	0	1	2	3	4	5	6	7	8
suivant	3	4	-2	-1	0	-2	-2	-2	-2
donnée	48,2	33,5		7,9	1,2				

Pour trouver le premier :

```
ADRCELL liste1 = 1 ; // la première cellule est à l'indice 1
```

2.3 Langages avec de l'allocation dynamique

Ces langages ont des types de données composées et des adresses gérées par le système. Dans ce cas, on utilise comme adresse une adresse système (quand on la voit, c'est un nombre sur 4 octets, ininterprétable), et on demande de l'espace au fur et à mesure qu'on crée des cellules.

2.3.1 En C

Ex. on crée une liste qui contient des nombres positifs lus au clavier (on ne sait pas combien). L'algorithme consiste à répéter insérer_après(dernier) jusqu'à ce qu'on lise un nombre négatif – avec un cas particulier pour la première insertion.
(Rappel : en C, quand on a le **nom** d'une structure, on accède à ses champs par nom.champs; quand on a son **adresse**, on accède à ses champs par adresse->champs)

```
struct cell {float donnee; struct cell*suivant;};
typedef struct cell CELL, *ADRCELL ;
ADRCELL initListe(void) {
  ADRCELL premier, dernier, nouveau ;
  float lu  ;
  premier = NULL ; /* liste vide */
  while (1) {
    scanf("%f",&lu) ;
    if (lu < 0) break ; // lu <0 veut dire : c'est fini
    nouveau = (ADRCELL) malloc(sizeof(CELL)) ;
    nouveau->donnee = lu;
    if(premier == NULL)
      premier = nouveau ; /* c'est la première lecture */
    else
      dernier->suivant = nouveau ; /* dernier est le nouveau de la
                lecture precedente */
    //fin du if
    dernier = nouveau ; /* on change dernier pour la lecture suivante */
  }// fin du while
  dernier->suivant = NULL ; /* ferme la liste */
  return (premier) ;
}//fin de initListe()
```

2.3.2 En Java

Rappel : Java ne gère pas l'espace de la même manière pour les types primitifs et pour les objets.

- La déclaration d'une variable ou d'un tableau **de type primitif** (int, float, char) réserve l'espace mémoire nécessaire à cette variable (ce tableau).
- Pour les **objets**, java procède autrement : la déclaration ne réserve que l'espace nécessaire à leur **adresse**. Pour obtenir l'espace nécessaire à l'objet lui-même, il faut faire un **new**, et chaque new créé un nouvel objet (qui ne subsiste que tant qu'on garde son adresse).

Les types de données composées sont les classes. Ci dessous un schéma d'implémentation avec des données de type *Élément* (schéma, parce que on peut remplacer *Élément* par n'importe quel type, y compris primitif). Seules les méthodes de services sont traitées.

```
private class CellElem{
    Élément donnee ;
    CellElem suivant ;

    CellElem() {
    this(null) ;
    }

    CellElem(Élément val) {
    donnee = val ;
    suivant = null ;
    }

    void setDonnee(Élément val) {
    donnee = val ;
    }

    void setSuivant ( CellElem suiv) {
    suivant = suiv ;
    }
    // etc. …
    }
```

Le même programme de création de liste avec des cellules de type CellFloat s'écrit :

```
CellFloat premier, dernier, nouveau ;
float lu  ;
premier = null ; /* liste vide */
while(true) {
    lu = Console.readFloat() ; // par exemple
    if (lu < 0) break ;
    nouveau = new CellFloat() ;
    nouveau.setDonnee(lu);
    if(premier == null)
          premier = nouveau ; /* c'est la première lecture */
    else
          dernier.setSuivant(nouveau) ; /* dernier est le nouveau de la
                                           lecture précédente */
    //fin du if
    dernier = nouveau ; /* on change dernier pour la lecture suivante */
}// fin du while
/* dernier.setSuivant(null) ; a déjà été fait par le constructeur */
```

Notez bien que **new** en java est pour la création l'équivalent du **malloc()** en C : nouveau contient successivement les adresses de toutes les cellules créées – même si Java ne permet pas de lire cette adresse ou de faire des opérations dessus.

3 Le style de programmation visé

Les algorithmes que nous avons écrits pour insérer ou supprimer un élément laissent de coté une question importante : quel est le rapport entre l'ancienne liste et la nouvelle liste ?
Autrement dit, est-ce que l'ancienne liste est perdue, remplacée par la nouvelle (ce qu'on appelle une modification physique), ou bien est-ce que l'ancienne version subsiste, et la nouvelle est créée à partir de cellules physiquement différentes ?
Les deux sont possibles, et on n'écrira pas les programmes de la même façon suivant ce qu'on a choisi. Comme c'est une source d'erreurs, nous allons prendre le temps d'expliquer.

3.1 Copie et partage de données en Java

Le problème existe en Java en dehors des listes. Quand une affectation ou une fonction porte sur un **type primitif**, java travaille par **copie**. On ne peut donc pas modifier l'original en modifiant la copie. Ex :

```
int x = 3, y ;
y = x ; // c'est une copie
y = y + 4 ; // x n'est pas changé
```

Par contre, quand une affectation porte sur un **objet**, java ne le copie pas, mais utilise son **adresse**. Ex :

```
Component comp1 = new Component(), comp2 ;
comp2 = comp1 ;
comp2.add(myPopupMenu) ; // comp1 est changé aussi
```

comp1 et comp2 sont deux noms pour le même objet (en fait, deux emplacements mémoire qui contiennent la même adresse), Toute modification faite sur l'un modifie aussi l'autre. Donc comp1 a le même menu que comp2. Quand une instruction a un effet qui n'est pas décelable à la seule lecture de cette instruction – comme ici l'ajout d'un menu à comp1 par la 3eme ligne – on parle d'**effet de bord**.

3.2 Implémentation des listes et effets de bord

Cette propriété des objets explique la distinction que fait Java pour les objets entre == (il n'y a qu'un seul objet, une seule adresse mémoire) et equal() (les objets occupent deux zones mémoire différentes mais leurs variables ont les mêmes valeurs). Le problème existe dans tous les langages pour les données complexes, et se pose pour les cellules et les listes.

Supposons qu'on a le code suivant (en C. concat est une fonction qui ajoute les listes l'une derrière l'autre) :

```
LISTE l1, l2, l3 ;
/* remplissage de l1 et l2 */
l3 = concat(l1, l2) ;
```

Si on a implémenté concat() comme une opération physique sans recopie, on a attaché l2 à la fin de l1, donc : l1 est modifiée, on a l1 == l3 et toute modification de l2 après cette instruction modifie l1 et l3 par effet de bord.

Les effets de bord sont considérés comme une source d'erreurs très difficiles à déceler. Une option est de les supprimer en dupliquant automatiquement les listes – dans ce cas, concat(l1,l2) renvoie la concaténation d'une copie de l1 et d'une copie de l2. C'est une solution sûre, mais qui peut devenir coûteuse si le programmeur n'est pas conscient de cette duplication. Si il initialise une liste par

```
LISTE maListe = NULL, l1 ;
for(i=0 ; i < 500 ; i ++) {
    lu = lireFloat() ;
    l1 = créeListe1(lu) ; /* crée une liste à 1 élément */
    maListe = concat(l1, maListe) ;
}
```

il crée successivement 250000 cellules (1 + 2 + 3 + …. + 500) en croyant en créer 500.

Une solution plus efficace consiste à fournir une fonction de copie et à laisser le programmeur autoriser ou interdire les effets de bord selon les cas. Ici aussi, il faut avoir bien compris et

programmer très proprement. En appelant copieListe() la fonction de copie et en gardant le style C de l'exemple, on a trois instructions correctes possibles :

```
l1 = concat(l1, l2) ; //l1 = et non l3 = , parce que l1 est modifiée
l3 = concat(copieListe(l1), l2) ; /* les 3 listes sont distinctes. Il
                        peut y avoir des effets de bord entre l2 et l3 */
l3 = concat(copieListe(l1), copieListe(l2)) ; // protection totale
```

3.3 Type 'liste' et type 'cellule'

Pour parcourir une liste chaînée, on n'a besoin que de sa première cellule – ensuite on répète "passer au suivant". L'implémenteur doit décider s'il utilise un type liste distinct du type cellule. Il y a deux façons cohérentes de faire.

3.3.1 Choix 1 : un seul type

Pour que le code soit lisible, on fera en C un `typedef ADRCELL LISTE`
En java, dans ce choix il vaut mieux créer une seule classe qui s'appelle `ListeElem`, qui remplace celle qui est appelée plus haut `CellElem`, avec les mêmes variables.

Quel que soit le langage, les opérations insérer_après, supprimer_après sont faciles à implémenter puisque ce sont des modifications des adresses dans les cellules. L'opération qui consiste à ajouter un élément devant la tête de liste sera en C une fonction *ajouter_en_tête(maListe,nombre)* ou en Java une méthode *maListe.ajouter_en_tête(nombre).* Dans les deux cas, il faut affecter à maListe sa nouvelle valeur, et on ne peut pas le faire dans la fonction ou la méthode (essayez !). La seule solution est de renvoyer le résultat, et d'écrire maListe = ajouter_en_tête(maListe,nombre). Mais alors il faut utiliser le même style partout.

Bilan : il y a un seul type, qui comporte une donnée et un suivant. Toutes les fonctions de modification renvoient une liste résultat du type liste sur lequel on travaille(ListeFloat, ListeElem, etc.). Ces fonctions sont toujours appelées en affectant ce résultat à la liste modifiée. Ex. en Java :

```
maListe  = maListe.ajouter_en_tête(nombre) ;
maListe  = maListe.ajouter_après(nombre) ;
maListe  = maListe.supprimer_après(indice) ;
```

etc.

3.3.2 Choix 2 : liste est un type différent de cellule

En passant une liste comme argument de fonction, on pourra modifier l'adresse de la cellule de tète, donc ajouter en tète dans la fonction. En C :

```
typedef struct {ADRCELL tete } LISTE ;
void ajouter_en_tete(LISTE *maListe, float nombre) {
   nouveau = (ADRCELL) malloc(sizeof(CELL)) ;
   nouveau->donnee = lu;
   nouveau->suivant = maListe->tete ;
   maListe->tete = nouveau ;
}
```

En Java, on a le schéma d'implémentation suivant :

```
public class ListeElem {
   CellElem premier ;
   ListeElem() {
      this(null) ; }// fin du constructeur
```

```
    ListeElem(CellElem c) {
       premier = c ; }// fin du constructeur
    ListElem(Elem ea[]) {
       for(int i = ea.length - 1 ; i >=0 ; i—) {
       ajouterEnTete(ea[i]);
       }// fin du for
    }// fin du constructeur

    public void ajouterEnTete(Elem e){
      CellElem cell = new CellElem(e) ;
      cell.setSuivant(premier) ;
      premier = cell ;
    }

    public Element supprimerPremier () {
    …
    }

  // etc. …
  }
```

Bilan : il y a un type Liste et un type Cellule. Toutes les fonctions de modification sont de type void (autrement dit, elles ne renvoient rien). Elles sont appelées sans affectation, ex. :

```
maListe.ajouter_en_tête(nombre) ;
maListe.ajouter_après(nombre) ;
maListe.supprimer_après(indice) ;
```

etc.

4 Parcours de listes

L'approche la plus simple du parcours de liste consiste à écrire une fonction différente pour chaque traitement. Nous en donnons quelques exemples en C dans le style 'un seul type', pour montrer la variété des combinaisons possibles.

```
struct cell {float donnee; struct cell*suivant;};
typedef struct cell CELL, *ADRCELL ;
typedef ADRCELL LISTE ;

void affiche(LISTE l) {
ADRCELL courant = l ;
while(courant != NULL) {
   printf(" %f  ",courant->donnee) ;
   courant = courant->suivant ;
}}

float max(LISTE l) {
float m ;
ADRCELL courant = l ;
m = 0 ;
while(courant != NULL) {
   if (m < courant->donnee) m = courant->donnee ;
   courant = courant->suivant ;
   }
return(m) ;
}
```

```
float somme(LISTE l) {
float s ;
ADRCELL courant = l ;
s = 0 ;
while(courant != NULL) {
    s = s + courant->donnee ;
    courant = courant->suivant ;
    }
return(s) ;
}

    /* en bonne programmation, on doit voir dans le nom de fonction  si
    elle modifie sa liste argument et peut faire un effet de bord.
    J'emploie rplac (= remplace) pour marquer les fonctions qui font des
    effets de bord */
void rplacDouble(LISTE l) {
ADRCELL courant = l ;
while(courant != NULL) {
    courant->donnee =  2 * courant->donnee ;
    courant = courant->suivant ;
}}

LISTE elemsSup(LISTE l, float n) {
/* construit une nouvelle liste qui contient les éléments supérieurs à n
de la liste argument */
LISTE sliste ;
ADRCELL courant, nouveau,dernier ;
courant = l ; sliste = NULL ;
while(courant != NULL) {
    if(courant->donnée > n) {
          nouveau = (ADRCELL) malloc(sizeof(CELL)) ;
          nouveau->donnee = courant->donnee ;
          if(sliste == NULL)
                sliste = nouveau ; /* c'est la première boucle */
          else
                dernier->suivant = nouveau ; /* dernier est le nouveau de
                                                la boucle precedente */
          //fin du if
          dernier = nouveau ; /* on met à jour dernier */
          courant = courant->suivant ;
    }//fin du if
}// fin du while
  dernier->suivant = NULL ; /* ferme la liste */
  return(sliste) ;
}

LISTE différences(LISTE l) {
  /* construit une nouvelle liste qui contient les différences entre
  éléments successifs de la liste argument */
  LISTE dliste ;
  ADRCELL precedent, courant, nouveau,dernier ;
  precedent = l ; dliste = NULL ;
  if (precedent == NULL) return(dliste) ;// traite à part la liste vide

  courant = precedent->suivant ;
  while(courant != NULL) {
      nouveau = (ADRCELL) malloc(sizeof(CELL)) ;
      nouveau->donnee = courant->donnee - precedent->donnee ;
```

```
    if(dliste == NULL)
      dliste = nouveau ; /* c'est la première boucle */
    else
      dernier->suivant = nouveau ; /* dernier est le nouveau de la
                boucle precedente */
    //fin du if
    dernier = nouveau ; /* on change dernier pour la boucle suivante */
    precedent = courant ;
    courant = courant->suivant ;
  }// fin du while
  dernier->suivant = NULL ; /* ferme la liste */
  return(dliste) ;
}

void main(void) {
LISTE maListe, difListe, ddifliste ;
maListe = initListe() ;
affiche(maListe) ;
printf("\nPlus grand élément : %f ;  Somme : %f",
                      max(maListe), somme(maListe)) ;
printf("\néléments supérieurs à 3 : \n") ;
affiche(elemsSup(maListe,3)) ;
rplacDouble (maListe) ;
difListe = différences(maListe) ;
ddifListe = différences(difListe) ;
printf("\ndifférences secondes doublées: \n") ;
affiche(ddifListe) ;
}
```