Informatique de base-IV

10 septembre 2014

1 Algorithmes et fonctions récursifs

Nous étudions ci-dessous la récursion comme technique algorithmique et son instanciation en langage C. Pour cela nous revenons d'abord sur la notion de fonction.

1.1 Rappel sur les fonctions et leurs appels

Une fonction se définit en utilisant la construction :

```
typeRetour nomFonction (type1 arg1, type2 arg2n, ... , typen argn){
        type1 objl1 ;

         instructions ;

        return expressionRetour ;
}
```

Elle est appelée depuis une autre fonction avec un appel de la forme :

```
nomFonction ( expr1, expr2, ..., exprn );
```

Le résultat de cet appel est l'évaluation d'une expression et la valeur est du type de la valeur de retour de la fonction. Les arguments sont transmis par valeur, ces valeurs sont celles des expressions passées comme arguments lors de l'appel. Plus précisément le déroulement des opérations est le suivant :

1. Pour chaque argument *expri*, l'expression est calculée.
2. La fonction *nomFonction*, ou plutôt une *instanciation* de celle-ci, est empilée sur la pile des appels de fonctions.
3. Les n variables arguments de la fonction sont créées et initialisées avec les valeurs calculées au 1.
4. Le corps de la fonction est alors exécuté.
5. Lorsque la fonction rencontre l'instruction *return* les variables locales disparaissent (c'est-à-dire qu'elles ne sont plus accessibles) et la valeur *expressionRetour* est retournée.

6. L'appel à la fonction est alors "dépilé" et l'exécution reprend là où elle avait été arrêtée. Si une valeur de retour a été donnée, cette valeur sert de valeur pour calculer l'expression où apparaît l'appel de fonction.

Remarquons que les variables locales incluent les variables *argi* contenant les valeurs des expressions *expri* ainsi que les variables locales déclarées dans la définition de la fonction comme *objl1*.

1.2 Appel récursif

Une fonction peut s'appeler elle-même, soit directement soit dans un appel à une autre fonction. Un algorithme utilisant des fonctions récursives est un algorithme récursif. Il peut y avoir plusieurs *appels emboîtés* lors de l'exécution. La liste des appels emboîtés est gérée par une pile. La gestion des variables et des arguments suit le modèle de l'appel non-récursif : chaque appel crée ses propres variables arguments et ses propres variables locales.

1.3 Exemple de programme récursif

On veut calculer $a \times b$, a et b étant deux entiers positifs ou nuls, en utilisant la formule :

$$\begin{aligned} a \text{ fois } 0 &= 0 \\ a \text{ fois } b &= (a \text{ fois } (b-1)) + a \end{aligned}$$

On définit la fonction suivante :

```
int fois (int a, int b){
        // renvoie a*b si a et b sont positifs ou nuls
         if (b==0)
                return 0;
        return fois (a,b-1)+a;
}
```

appelée par la fonction principale suivante :

```
int main(void){
        printf (" %d fois %d = %d \n", 3, 2, fois (3,2));
        return 0;
}
```

Simulation de fois(3,2)

Nous simulons ci-dessous l'exécution de la fonction principale :

```
{fois(3, 2} ? }
        {a = 3, b = 2}
        {b == 0 est faux}
        {fois (3, 1} ? }
                {a = 3, b = 1}
                {b == 0 est faux}
```

```
                {fois (3, 0} ? }
                        {a = 3, b = 0}
                        {b = 0 est vrai}
                        {retourne 0}
                {= 0 }
                {retourne 0 +  a = 3}
        {= 3 }
        {retourne 3 + a = 6}
{ =6 }
{ Affichage de 6}
```

1.4 Ecriture d'une fonction récursive

Nous avons vu ci-dessus *comment* s'exécute une fonction récursive, cependant l'écriture d'une fonction récursive ne se fait pas en simulant mentalement son exécution, ce qui s'avère très vite impraticable, mais en acceptant l'appel, dans le corps de la fonction, à une autre instance de cette même fonction. L'écriture de fonctions récursives est très proche de la conception de *preuves par récurrence*, et de fait en écrivant une fonction on fait une preuve de sa correction et de sa terminaison. Par *correction* on entend ici que le résultat est le résultat voulu et par *terminaison* qu'un appel à la fonction se termine toujours après un nombre fini d'appels. Voici ci-dessous les étapes principales de cette preuve :

1. Les appels de la fonction qui ne conduisent pas à d'autres appels (qu'on dira récursifs) sont corrects.
2. Si on suppose que les appels récursifs sont corrects alors l'appel de la fonction est correct.
3. Dans tous les cas un appel à la fonction aboutit, en fin de compte, à des appels non récursifs.

Si nous reprenons le cas de la fonction `fois` les trois étapes s'écrivent de la manière suivante :

1. `fois(a,0)` renvoie 0 ce qui est correct.
2. Si on suppose que `fois (a,b-1)` renvoie $a \times (b-1)$ alors `fois(a,b)` renvoie $a \times (b-1) + a = a \times b$ ce qui est correct.
3. Comme a et b sont des entiers positifs ou nuls, alors un appel à `fois(a,b)` conduira à l'appel de `fois(a,b-1)` qui conduira à l'appel `fois(a,b-2)` etc ...et finalement à l'appel non récursif `fois(a,0)`

Remarquons que la question des *spécifications*, c'est à dire des conditions dans lesquelles la fonction est correcte et se termine, est cruciale. Dans cet exemple précis, la fonction ne se termine pas si b est un entier négatif bien que les appels récursif soient corrects : supposons que nous appelions `fois(3,-1)` alors si on suppose que `fois(3,-2)` renvoie -6 alors (`fois(3,-2) + 3`) a comme valeur $-6+3=-3$ ce qui correspond bien à 3×-1. Cependant clairement la fonction ici ne se termine pas puisque `fois(3,-2)` va faire appel à `fois(3,-3)` etc ...Cependant notre fonction

est correcte et se termine dans les conditions prévues dans ses spécifications, c'est-à- dire lorsque a et b sont positifs ou nuls. Remarquons également qu'ils se trouve qu'elle donne également le résultat de $a \times b$ lorsque a est négatif, mais cela n'étant pas prévu dans ses spécifications (voir le commentaire dans le corps de la fonction) le programmeur ne nous donne pas cette garantie et donc nous n'utiliserons pas la fonction pour des valeurs négatives de a.

1.5 Mémorisation implicite

La programmation récursive utilise un mécanisme d'empilement des appels à une même fonction. Cela crée autant de "variables-arguments" et de "variables locales" qu'il y a d'appels et on a alors un mécanisme de mémorisation implicite.

Exemple : Fonction Envers

Supposons que nous désirions écrire un programme qui lit depuis le clavier un certain nombre de caractères jusqu'à ce que l'utilisateur entre le caractère "#". Le programme doit alors afficher les caractères lus mais à l'envers (et sans le dernier caractère #). Par exemple, si l'utilisateur tape la suite de caractères `ABCDEF#`, le programme devra afficher `FEDCBA`. Voici une version récursive de cette fonction :

```
void envers(void){
        char ch;
        scanf("%c",&ch){
        if (ch!='#'){
                envers();
                printf("%c",ch);
        }
}
```

Simulation avec `ABC#`

```
{envers() }
{lecture de ch = 'A'}
          {c !='#' est vrai}
          {envers() }
          {lecture de ch = 'B'}
                    {c !='#' est vrai}
                    {envers() }
                            {lecture de ch = 'C'}
                            {c !='#' est vrai}
                            {envers }
                                    {lecture de c = '#'}
                                    {c !='#' est faux}
                            {}
                            {écriture de c ='C'}
                  {}
```

```
                {écriture de c ='B'}
        {}
        {écriture de c ='A'}
{}
```

Ci-dessous l'exécution du programme depuis un terminal. L'utilisateur entre le mot `ABC` puis un `#`.

```
sol > ./essaiEnvers
ABC#CBA
```

Ainsi, l'utilisation d'une variable locale (ici `ch`) par chaque instance de la fonction `envers` a permis de mémoriser implicitement les caractères lus depuis le clavier.

Quelle est ici *l'idée récursive* ? Nous avons simplement remarqué que pour afficher un mot à l'envers il suffit d'afficher à l'envers le mot privé de sa première lettre, puis d'afficher celle-ci. Autrement dit, la fonction récursive ne fait qu'exprimer un procédé en mentionnant dans sa description ce même procédé mais appliqué à une instance plus *simple* du problème à résoudre. Quantité de procédés s'expriment ainsi, par exemple chercher un objet dans un appartement peut se faire en cherchant l'objet dans une pièce puis, si on ne l'a pas trouvé, dans le reste de l'appartement. L'utilisation de la récursion suppose toujours de trouver d'abord une idée récursive permettant de résoudre le problème à traiter.

1.6 Appels récursifs multiples

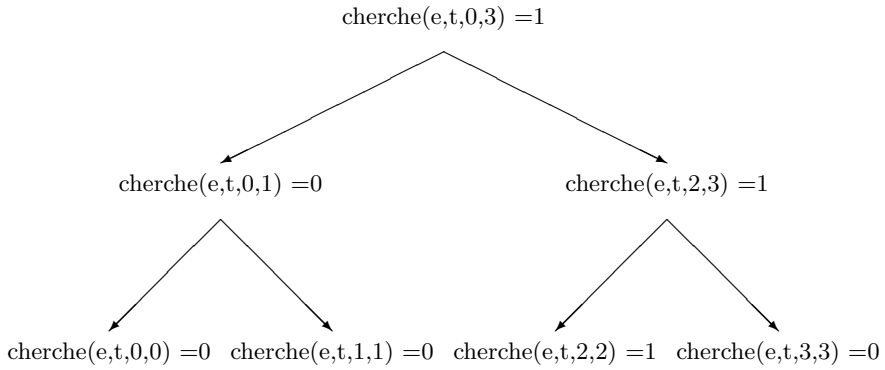
Il peut y avoir plus d'un appel récursif à l'intérieur d'une fonction. Supposons, par exemple que nous désirions chercher si un élément se trouve dans un tableau. Une méthode consiste, lorsque le tableau comporte au moins 2 éléments, à le couper en deux et à rechercher l'élément successivement dans les deux sous-tableaux. Cela donne la fonction suivante :

```
int cherche( char e, char t [], int ideb, int ifin ){
        // renvoie 1 si e est présent dans le tableau t entre ideb et ifin
        // et 0 sinon
        int milieu ;
        if (ideb>ifin ) // la partie à explorer est vide
                return 0;
        if (ideb==ifin) // il n'y a qu'un élement entre ideb et ifin
                return (t[ideb]==e);
        milieu=(ideb + ifin )/2;
        return cherche(e, t, ideb, milieu ) || cherche(e, t, milieu+1, ifin );
}
```

Simulation pour e='G' et t = 'TCGT'

Considérons l'exemple suivant : e = 'G' A = 'TCTG'
Nous donnons ci-dessous l'arbre des appels :

Arbre des appels successifs



Si l'on suit les flèches on obtient l'ordre des appels récursifs. Cet ordre correspond à un parcours de cet arbre en profondeur d'abord et à gauche d'abord. Par convention à un même niveau le premier appel récursif est à gauche. Ainsi pour obtenir le résultat de *cherche ('G',t, 0, 3)* on appelle dabord *cherche ('G', 0, 1)* puis quand on a le résultat de ce dernier appel on appelle *cherche ('G',t, 2, 3)*. Cependant l'ordre d'exécution est lui-même récursif : on n'exécute *Cherche ('G',t, 2, 3)* que lorsque les appels à *cherche ('G',t, 0, 0)* et *cherche ('G', t, 1, 1)* nécessaires pour obtenir *cherche ('G',t, 0, 1)* ont été eux-mêmes exécutés.

Remarquons que la fonction est correcte (et se termine) lorsque `ideb` et `ifin` sont des indices autorisés du tableau, c'est-à-dire entre 0 et la taille du tableau -1 :

1. Les appels sont non récursifs dans deux situations : i) lorsque `ideb > ifin` la fonction renvoie 0, ce qui est correct puisque la région où chercher e est vide ii) lorsque `ideb== ifin` soit `tab[ideb)==e` et dans ce cas la fonction renvoie 1 (on a trouvé e) ce qui est correct également.
2. Si on suppose que les appels à `cherche (e,t, ideb, imilieu)` et `cherche (e,t, imilieu+1, ifin)` sont corrects alors le OU entre les deux appels est vrai si et seulement si e a été trouvé dans une des deux régions au moins, c'est-à-dire entre `ideb` et `ifin`, ce qui est conforme à la spécification.
3. Le seul cas où il y a des appels récursif est celui où on a `ideb < ifin`. Les appels récursifs se font sur deux régions strictement incluses dans celles-ci, donc après une cascade d'appels on arrivera à une région soit vide, soit contenant un seul élément.

1.7 Fonctions récursives sur les listes doublement chaînées : posMin et triFusion

Nous allons utiliser ici la récursion pour effectuer différentes tâches sur les listes, ici doublement chaînées.

– Pour chercher la position du plus petit élément d'une liste, nous allons utiliser une fonction `posMinC` qui fait appel à la fonction récursive `posMin`. La fonction `posMinC` est la fonction qui sera appelée, ici par la fonction principale,

pour chercher la position du plus petit élément de la liste l, sous la forme `posMinC(l)`. Nous dirons de `posMinC` que c'est la fonction *chapeau* de la fonction récursive `posMin`.

– Pour trier une liste par la méthode du *tri fusion* nous allons écrire plusieurs fonctions, dont une fonction d'interclassement de deux listes triées, une fonction qui calcule la taille d'une liste, et une fonction de tri, toutes récursives. Le calcul de la taille utilisera aussi ici une fonction chapeau.

Nous allons tester ces fonctions dans un programme principal, utilisant les sources vues précédemment sur l'implémentation des listes doublement chaînées. Dans la source principale, nous allons ajouter les prototypes des nouvelles fonctions, puis, après la fonction principale, les définitions de ces nouvelles fonctions.

Nous avons aussi ajouté à `listesDoublementChainees.h`, la déclaration suivante :

```
#define HORS_LISTE NULL
```

Dans les fonctions ci-dessous nous utiliserons cette constante `HORS_LISTE`, pour différencier le cas d'une liste vide, initialisée à `NULL`, de celui de la position hors-liste de la liste. Ainsi, pour ajouter un élément en queue de liste, nous l'ajouterons à cette position `HORS_LISTE`.

L'idée ici est que l'implémentation confond une liste et la position d'un élément : dans les deux cas on utilise un pointeur sur une cellule. C'est pour cette raison que nous avons défini un alias `listeD` de `cellule*`. `listeD` est utilisé lorsqu'on se réfère à une liste, et `cellule*` lorsqu'il s'agit de la position d'un élément.

Ainsi pour ajouter un élément en tête de liste nous l'ajouterons à la position `premier(l)`, alors que le compilateur (qui ne différencie pas un type de son alias) nous autorise à utiliser `l` : dans les deux cas il s'agit de l'adresse du premier élément de la liste.

Pour revenir sur le rôle des fonctions chapeau, elles permettent au minimum de ne donner à une fonction comme `posMinC` que les arguments nécessaires : on fera un appel à `posMinC(l)` pour obtenir la position du plus petit élément de la liste, alors que l'appel de la fonction récursive, `posMin(premier(l),l)`, nécessiterait de préciser que c'est à partir de son premier élément qu'il faut chercher le plus petit de la liste. Mais elles ont un rôle plus large : il est parfois utile d'exécuter certaines instructions avant de faire l'appel récursif initial. Ce sera le cas de la fonction `tailleListeC`.

PosMin La fonction récursive `posMin` prend en argument la position `pDeb` à partir de laquelle on cherche le plus petit élément de la liste, ainsi que la liste elle-même. L'idée récursive est la suivante : le plus petit élément est soit en `pDeb`, soit dans le reste de la liste, c'est-à-dire à la position `pReste` du plus petit élément à partir du successeur de `pDeb` dans la liste. Il suffit de comparer ces deux éléments pour obtenir le résultat. Voici un exemple :

Soit l= 10 70 50 30 40

Pour trouver le plus petit élément à partir du second il suffit de comparer celui-ci (70) au plus petit élément de la liste à partir du troisième (50) , c'est-à-dire (30). En

pratique on a besoin simplement de la position de celui-ci : ici il s'agit du quatrième élément.

En supposant que la liste est représentée par une liste doublement chainée, les positions sont des adresses de cellules. Nous allons les noter ici selon leur rang : p1(1er), p2(second), etc ....

Ainsi l'appel `posMin(p2,l)` compare l'élément en p2 avec le plus petit à partir de p3, et celui-ci se trouve en p4.

triFusion Le tri fusion repose sur le procédé récursif suivant : on trie séparément la première moitié du tableau et la seconde moitié du tableau, puis on interclasse les éléments pour obtenir le tableau trié. Voici un petit exemple :

```
Soit 1 70 50  3 2 80 60 4 à trier
tri première moitié de t
1 3 50 70
tri seconde moitié de t
2 4 60 80
interclassement
1 2 3 4 50 60 70 80
```

La fonction de tri s'écrit simplement

```
listeD  triFusion ( listeD  l ){
        // Trie la liste l et renvoie la liste triée
        listeD  l1 , l2 ;

        coupe(l,&l1,&l2 );
        if  ( estListeVide ( l1 ))
                return  l2 ;
        if  ( estListeVide ( l2 ))
                return  l1 ;
        l1=triFusion( l1 );
        l2=triFusion( l2 );
        return  interclassement ( l1 , l2 );
}
```

Elle fait appel à une fonction coupe qui sépare en deux la liste `l` en deux listes de même taille. Les deux listes étant le résultat de l'appel, nous devons transmettre leurs adresses `l1` et `l2` : à la fin de l'exécution, à ces adresses nous aurons les deux sous-listes. le procédé est le même que pour l'échange de deux variables, ou l'utilisation d'un `scanf`. Le fait que les listes soient elle-même des adresses ne pose pas de problème.

La fonction coupe fait appel à la fonction `tailleListeC` qui renvoie la taille n de la liste, ce qui permet de ranger les $n/2$ premiers élément dans la liste `l1` d'adresse `&l1` et les autres dans la liste `l2` d'adresse `&l2`.

La fonction `tailleListeC` est une fonction chapeau,. Avant de faire l'appel à `tailleListe`, la fonction `tailleListeC` commence par traiter le cas où la liste est vide et dans ce cas renvoie 0. Ce n'était pas ici strictement nécessaire, on aurait pu

inclure ce test en début de `tailleListe`. De même, il est inutile de tester le cas où la position courante est celle du dernier élément. En effet la fonction `successeur` est ici correcte et ne se contente pas de renvoyer la cellule suivante, mais renvoie la position `HORS_LISTE`, c'est à dire l'adresse `NULL`. Ce qui rend l'écriture suivante correcte :

```
int  tailleListe ( cellule * pDeb, listeD l){
    /* renvoie le nombre d'elements de l à partir de pDeb */
    if (pDeb==NULL){
        return 0;
    }
    return 1+ tailleListe (successeur(pDeb,l), l);
}
```

La fonction `interclassement` est elle-même une fonction récursive reposant sur le principe suivant : pour interclasser deux listes triées, on compare les premiers éléments des deux listes, on garde la valeur v du plus petit des deux, on interclasse les deux listes après suppression de l'élément de valeur v de la liste à laquelle il appartenait, et on ajoute v en tête de la liste interclassée. Voici un petit exemple :

```
l1=1 3 50 70
l2= 2 4 60 80
v=min(1,2)=1
l'interclassement de 3 50 70 et  2 4 60 80 donne
2 3 4 50 60 70 80
on ajoute v=1 en tête, et on obtient
1 2 3 4 50 60 70 80
```

Un programme Nous donnons ci-dessous la source principale d'un programme testant ces différentes fonctions, parfois dans différents cas particuliers. Il est largement commenté, avec pour chaque fonctions ses spécifications. On trouvera à la fin en commentaire une trace d'exécution.

```
/* Récursion et listes doublement chaînées circulaires */
#include <stdlib.h>
#include <stdio.h>

#include "listesDoublementChainees.h"
/* attention HORS_LISTE doit être défini dans le .h précédent */


/*********************************************************
 prototypes des fonctions propres à ce programme */
/*********************************************************/

cellule * posMinC( listeD l );
cellule * posMin( cellule * pDeb, listeD l );
listeD interclassement ( listeD l1 , listeD l2 );
```

```
int   tailleListeC ( listeD  l );
int   tailleListe  ( cellule  * pDeb, listeD  l );
listeD  coupe ( listeD  l ,  listeD  *pl1 ,  listeD  *pl2 );
listeD  triFusion ( listeD  l );

/*********************************/
/* principale */
/*********************************/
int main()
{
    // On cree une liste l de n elements de la forme 1 3 ... 1+(2*n)
    // contenant au milieu un 0
    // on utilise posMin qui recherche la position du plus petit element dans l
        // on ajoute -1 en tête et -2 en queue de l

    // On cree deux listes l1 et l2 triees
    // de la forme 0 2 4 ** 2*(n-1) et 1 3 5 .. 1+(2*n)
    // et on les interclasse
    // on coupe ensuite cette liste en deux sous listes

    // On trie la liste l par tri fusion

    listeD  l , l1 , l2 , lr ,  lr1 , lr2 ;
    cellule * pM;
    int  i , n;
    /************************************/
    /* test de posMin */
    /************************************/
    printf (" Recherche du plus petit element d'une liste \n");
    printf (" Taille de la liste initiale ?\n");
    scanf("%d", &n);
    l=creerListe ();   // l est vide
    for( i=0; i< n; i++){
        l=ajouterEnPosition(1+2*i, HORS_LISTE,l); // ajout en queue
        if  (i==n/2)
            l=ajouterEnPosition(0, HORS_LISTE, l);
    }
    afficherListePrim (l );
    if( ! estListeVide (l )){
        pM=posMinC(l); // la fonction récursive s'appellera posMin
        printf (" min = %d \n", valeurElement(pM,l));
    }

    // on ajoute -1 en tête qui devient le plus petit element
    printf (" On ajoute -1 en tete \n");
    l=ajouterEnPosition(-1, l , l );
    afficherListePrim (l );
    pM=posMinC(l);
    printf (" min = %d \n", valeurElement(pM,l));
    // on ajoute -2 en queue
```

```
    printf(" On ajoute -2 en queue \n");
    l=ajouterEnPosition(-2,HORS_LISTE, l);
    afficherListePrim (l);
    pM=posMinC(l);
    printf(" min = %d \n", valeurElement(pM,l));
    /************************************/
    /* test interclassement */
    /************************************/
    printf(" Creation de deux listes triees a interclasser \n");
    l1=creerListe();
    l2=creerListe();
    for(i=0; i< n; i++){
        l1=ajouterEnPosition(2*i, HORS_LISTE,l1); // ajout en queue
        l2=ajouterEnPosition(1+2*i, HORS_LISTE,l2); // ajout en queue
    }
    l2=ajouterEnPosition(1+2*n, HORS_LISTE, l2);
    afficherListePrim (l1);
    printf(" de taille %d \n", tailleListeC (l1));
    afficherListePrim (l2);
    printf(" de taille %d \n", tailleListeC (l2));
    printf(" Interclassement \n");
    lr=interclassement(l1,l2);
    afficherListePrim (lr);
    printf(" de taille %d \n", tailleListeC (lr));
    /************************************/
    /* test coupe de la liste en deux */
    /************************************/
    printf(" Coupe de la liste en deux sous-listes \n");
    lr=coupe(lr, &lr1, &lr2);
    afficherListePrim (lr1);
    printf(" de taille %d \n", tailleListeC (lr1));
    afficherListePrim (lr2);
    printf(" de taille %d \n", tailleListeC (lr2));
    /************************************/
    /* test tri fusion */
    /************************************/
    printf(" Tri d'une liste \n");
    afficherListePrim (l);
    l=triFusion(l);
    afficherListePrim (l);

    return 0;
}
/*******************************/
/* fonctions */
/*******************************/
cellule * posMinC( listeD l){
    /* l n'est pas vide */
    /* renvoie la position de la plus petite valeur dans l */
    return posMin( premier(l), l);
```

```
}

cellule * posMin( cellule * pDeb, listeD l ){
    /* l n'est pas vide, pDeb doit être dans la liste */
        /* renvoie la position de la plus petite valeur dans l */
        /* à partir de pDeb */
     cellule *pReste;

    if (pDeb==dernier(l)){
        return pDeb;
    }
    pReste= posMin( successeur(pDeb,l),l );
        // pReste pointe sur le plus petit élément de la liste
        // à partir du successeur de pDeb
    if (valeurElement(pDeb,l)<= valeurElement(pReste,l))
        return pDeb;
    return pReste;
}

listeD interclassement ( listeD l1, listeD l2){
    /* interclasse deux listes triees l1 et l2 et renvoie la liste */
    element v1,v2;
    listeD l;
    if ( estListeVide (l1 )){
        return l2;
    }
    if ( estListeVide (l2 )){
        return l1;
    }
    v1=valeurElement(premier(l1 ), l1 );
    v2=valeurElement(premier(l2 ), l2 );
    l=creerListe ();
    if (v1 <= v2){
        l1= supprimerEnPosition(premier(l1 ), l1 );
        l=interclassement(l1 , l2 ); // interclassement du reste de l1 avec l2 en l
        l=ajouterEnPosition(v1,premier( l ), l ); // ajout de v1 en tete de l
    }
    else{
        l2= supprimerEnPosition(premier(l2 ), l2 );
        l=interclassement(l1 , l2 );
        l=ajouterEnPosition(v2,premier( l ), l );
    }
    return l;
}
int tailleListeC ( listeD l){
        /* renvoie la taille de la liste l */
    if ( estListeVide (l ))
        return 0;
    return tailleListe (premier(l ), l );// la liste n'est pas vide
}
```

```
int   tailleListe ( cellule  * pDeb, listeD  l){
    /* pDeb est dans la  liste  l , et l n'est pas vide */
    /* renvoie le nombre d'elements de l à partir de pDeb */
    if (pDeb==dernier(l)){// il y a au moins un element
        return 1;
    }
    // successeur(pDeb,l) est dans la liste (pDeb n'est pas le dernier)
    return 1+ tailleListe (successeur(pDeb,l),  l );

}
listeD  coupe (listeD  l ,  listeD  *pl1 ,  listeD  *pl2){
    /* coupe la  liste  l en deux listes                          */
    /* de tailles  si  possible  identiques                       */
    /* pl1 et pl2 sont les adresses de ces  listes        */
        /* renvoie la  liste  l vide                                          */
    int  i ,n;
    element v;
    listeD  l1 ,  l2 ; // pour nommer les listes  construites

    n= tailleListeC (l );
    // les n/2 premiers elements de l sont ranges dans l1
    l1=creerListe ();
    for( i=0;i<n/2;i++){
        v=valeurElement(premier(l ), l );
        l=supprimerEnPosition(premier(l ), l );
        l1=ajouterEnPosition(v,HORS_LISTE,l1);
    }
    // les  elements restant de l sont ranges dans l2
    l2=creerListe ();
    for( i=n/2;i<n;i++){
        v=valeurElement(premier(l ), l );
        l=supprimerEnPosition(premier(l ), l );
        l2=ajouterEnPosition(v,  HORS_LISTE,l2);
    }
    // les  listes  l1 et l2 sont rangees aux adresse pl1 et pl2
    (*pl1)=l1;
    (*pl2)=l2;
    return NULL;
}


listeD  triFusion ( listeD  l ){
        // Trie la  liste  l et renvoie la  liste  triée
        listeD  l1 , l2 ;

        coupe(l,&l1,&l2 );
        if  ( estListeVide (l1 ))
                return  l2 ;
        if  ( estListeVide (l2 ))
```

```
            return l1;
        l1=triFusion(l1);
        l2=triFusion(l2);
        return interclassement(l1,l2);

}


/*
 monMac:soldano gcc listesDoublementChainees.c essaiListesDSuite.c -o essaiListesDSui-
temonMac :soldano ./essaiListesDSuite  Recherche du plus petit element d'une liste
   Taille de la liste initiale ?
 3
   liste 1 3 0 5
  min = 0
  On ajoute -1 en tete
   liste -1 1 3 0 5
  min = -1
  On ajoute -2 en queue
   liste -1 1 3 0 5 -2
  min = -2
  Creation de deux listes triees a interclasser
   liste 0 2 4
  de taille 3
   liste 1 3 5 7
  de taille 4
  Interclassement
   liste 0 1 2 3 4 5 7
  de taille 7
  Coupe de la liste en deux sous- listes
   liste 0 1 2
  de taille 3
   liste 3 4 5 7
  de taille 4
  Tri d'une liste
   liste -1 1 3 0 5 -2
   liste -2 -1 0 1 3 5
 */
```