Algorithmique et structures de données

Structures de données dynamiques

Samar MOUCHAWRAB, PhD Eng

2A Cycle Préparatoire – Semestre 1
2017/2018

Introduction

Une structure de données dynamique est une structure qui permet d'**enregistrer les données** en mémoire dans le tas. **La gestion des données s'effectue à l'aide de pointeurs**.

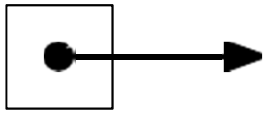
Les avantages de telles implantations sont les suivants :
- le tas a une taille plus grande que la zone de mémoire statique. On peut ainsi stocker plus de données.
- La gestion de la mémoire occupée par les données est contrôlée.

Ce contrôle se fait par programmation : on active des réservations mémoire correspondant à la taille des données. On active des libérations de l'espace mémoire précédemment alloué quand les données ne sont plus utiles. On n'est donc pas obligé de prévoir une taille maximale (constante max de la version statique).

Définir ce type de structures de données se fait à l'aide des pointeurs.

Pointeurs

Un pointeur est une variable “pointant” vers un emplacement en mémoire.



Un pointeur est une variable qui contient l’adresse mémoire d’une variable.

Pointeur = adresse mémoire (entier).

L’espace mémoire pointé peut être alloué :

- de façon statique : int i = 3; ou int *p;
- ou dynamiquement : int *q = (int *) malloc(sizeof(int));

Pointeurs

Il existe deux opérateurs spécifiques :

- **Référencement : &** donne l'adresse d'une variable, (un pointeur sur cette variable) :

```
char c = 'a';
char *p = &c;        // p est de type (char *)
```

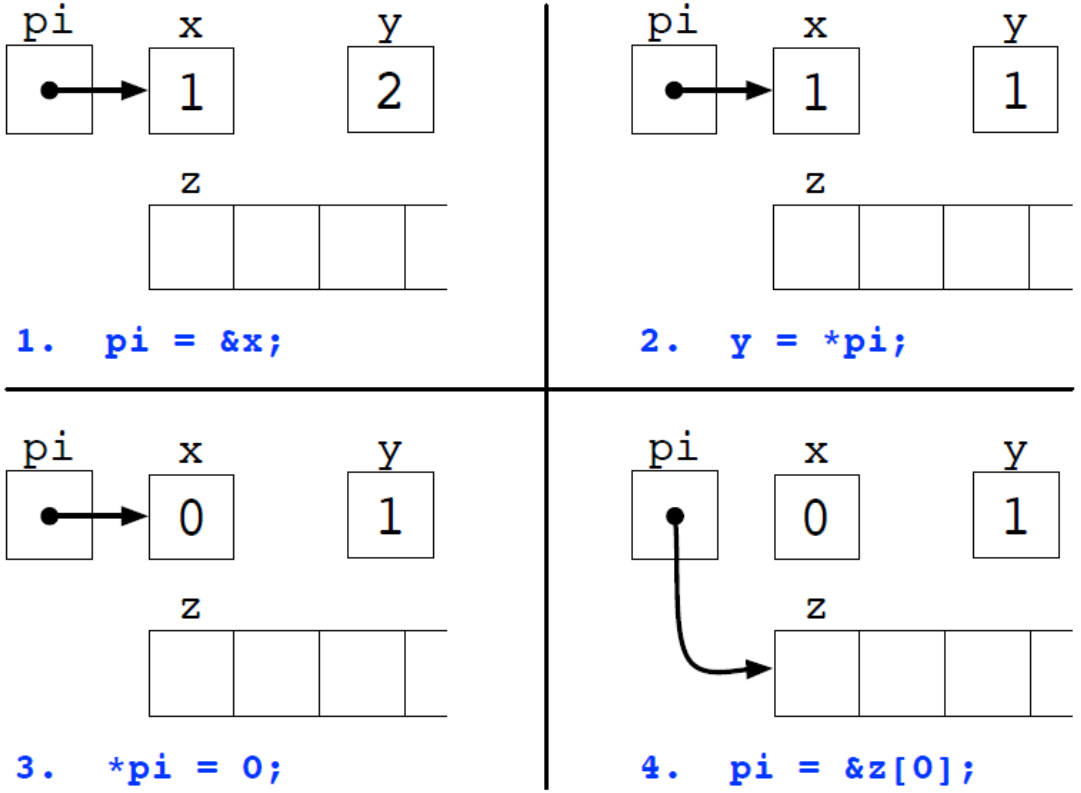
- **Déréférencement, Indirection : *** donne la valeur pointée par un pointeur (la valeur à l'adresse correspondante) :

```
printf("%c", *p);    /* affiche a */
```

Pointeurs

Exemples :

```
int x = 1, y = 2, z[10]; int *pi;
pi = &x;
y = *pi;     // y=*(&x)
*pi = 0;
pi = &z[0]; // pi=z
```



L'opérateur sizeof

L'opérateur sizeof() renvoie la taille en octets de l'argument (soit un type, soit une variable ou une expression).

Exemple:

```
short A;     char B[5][10];

sizeof A → 2

sizeof B → 50

sizeof 4.25 → 8

sizeof "abcd" → 5

sizeof(float) → 4

sizeof(double) → 8
```

L'opérateur sizeof

Exemple:

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

void fnc (int tab[]){
        printf(" %d,  %d \n", sizeof(tab), sizeof(tab[0]));
}

void main (){
        int t1[]={1,2,3,4,5};
        int *t2= (int *)malloc(20*sizeof(int));

        printf(" %d,  %d \n", sizeof(t1), sizeof(t1[0]));
        printf(" %d,  %d \n", sizeof(t2), sizeof(t2[0]));
        fnc(t1);
}
Affiche :     20 , 4
              8 , 4
              8 , 4
```

Exercice

Ecrire un programme qui retourne la taille en octets d'un char, d'un short, d'un int, d'un unsigned int, d'un float, et d'un double

```
#include <stdio.h>
#include <stdlib.h>
void main (){
    printf(" sizeof(char) = %d \n", sizeof(char));
    printf(" sizeof(short) = %d \n", sizeof(short));
    printf(" sizeof(int) = %d \n", sizeof(int));
    printf(" sizeof(unsigned int) = %d \n", sizeof(unsigned int));
    printf(" sizeof(float) = %d \n", sizeof(float));
    printf(" sizeof(double) = %d \n", sizeof(double));
}
```

```
Résultat :
 sizeof(char) = 1
 sizeof(short) = 2
 sizeof(int) = 4
 sizeof(unsigned int) = 4
 sizeof(float) = 4
 sizeof(double) = 8
```

La fonction malloc

Pour allouer la mémoire dynamiquement.

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

- Alloue size octets ;
- Renvoie un pointeur sur la mémoire allouée ;
- La zone de mémoire n'est pas initialisée.

```
long *a =(long *)malloc( sizeof(long) );
int n = 100;
long *tab = (long *) malloc(n * sizeof(long));
struct z {int f,g;};
struct z *p;
p=(struct z *)malloc(sizeof(struct z)); /* ptr sur struct z */
```

Les fonctions calloc et realloc

`void *calloc (size t nb, size_t size);`

Alloue et remplit de 0 la mémoire nécessaire pour nb éléments de size octets et renvoie un pointeur sur la zone allouée.

```
int n = 100;
long *tab = (long *) calloc(n, sizeof(long));  /* n fois 0 */
```

`void *realloc (void *p, size_t size);`

Réduit (ou augmente) la taille du bloc de mémoire pointé par p à une taille de size octets et conserve les size premiers octets à l'adresse p. Le reste de la nouvelle zone n'est pas initialisé.

```
int * tab;
tab = (int *) calloc ( 2, sizeof(int) );
tab[0] = 33;
tab[1] = 55;
tab = (int *)realloc(tab, 3 * sizeof(int) );
tab[2] = 77;
```

Vérification du retour NULL

Attention, il se peut se produire des erreurs lors de l'allocation dynamique de mémoire : il faut toujours vérifier que le pointeur retourné lors de l'allocation n'est pas NULL !

```
int *tab1, *tab2;

tab1 = (int *) malloc( 1000*sizeof(int) );
if( tab1 == NULL ){
        fprintf(stderr,"allocation ratee !");
        exit(EXIT_FAILURE);
}
if((tab2 = (int *)malloc(1000*sizeof(int)))== NULL )  {
        fprintf(stderr,"allocation ratee !");
        exit(EXIT_FAILURE);
}
```

Libérer la mémoire

La mémoire n'étant pas infinie, lorsqu'un emplacement mémoire n'est plus utilisé, il est important de libérer cet espace.

```
void free( void *p )
```

- libère l'espace mémoire pointé par p
- qui a été obtenu lors d'un appel à malloc, calloc ou realloc
- Sinon, ou si il a déjà été libéré avec free(), le comportement est indéterminé
- Attention, free ne met pas le pointeur p à NULL.

Exemple

Quel est le résultat ?

```
#include <stdio.h>
int main(){
   int* pc;
   int c;
   c = 22;
   printf("Addresse of c:%d\n",&c);
   printf("Value of c:%d\n\n",c);
   pc = &c;
   printf("Address of pointer pc:%d\n", pc);
   printf("Content of pointer pc:%d\n\n",*pc);
   c = 11;
   printf("Address of pointer pc:%d\n", pc);
   printf("Content of pointer pc:%d\n\n",*pc);
   *pc = 2;
   printf("Address of c:%d\n",&c);
   printf("Value of c:%d\n\n",c);
   return 0;
}
```

Exemple

```
Address of c: 2686784
Value of c: 22

Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
Content of pointer pc: 11

Address of c: 2686784
Value of c: 2
```

Exercice 1

Ecrire une fonction en C qui échange deux entiers. Utiliser un appel par référence et non pas par valeur.

Tester avec un appel de la fonction à partir de main

Solution

```
#include <stdio.h>
void swap(int *a,int *b);
int main(){
  int num1,num2;
  printf("Entrer deux numeros : ");
  scanf("%d %d", &num1, &num2);
  printf("Avant l'echange: \n");
  printf("Numero1 = %d\n",num1);
  printf("Numero2 = %d\n",num2);
  swap(&num1,&num2);
  printf("Apres l'echange: \n");
  printf("Numero1 = %d\n",num1);
  printf("Numero2 = %d",num2);
  return 0;
}
void swap(int *a,int *b){
  int temp;
  temp=*a;
  *a=*b;
  *b=temp;
}
```

Exercice 2

Ecrire un programme en C qui trouve la somme de n éléments saisis par l'utilisateur en utilisant des pointeurs et de la réservation de mémoire en nombre de n saisi par l'utilisateur.

Après avoir terminé le calcul, libérer la mémoire réservée.

Attention ! Vérifier le cas où la réservation de mémoire a généré une erreur.

Solution

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

int main(){
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int));

    if(ptr==NULL) {
        printf("Error! memory not allocated.");
        exit(0);
    }

    printf("Enter elements : ");
    for(i=0;i<n;++i) {
        scanf("%d",ptr+i);
        sum += *(ptr+i);
    }

    printf("Sum=%d", sum);
    free(ptr);
    return 0;
}
```

Exercice 3

Ecrire un programme en C qui trouve le plus grand nombe d'une série de n nombres saisis par l'utilisateur en utilisant des pointeurs et de la réservation de mémoire en nombre de n saisi par l'utilisateur.

Après avoir terminé le calcul, libérer la mémoire réservée.

Attention ! Vérifier le cas où la réservation de mémoire a généré une erreur.

Solution

```
#include <stdio.h>
#include <stdlib.h>
void main() {
    int i,n;
    int *data;
    printf("Enter total number of elements(1 to 100): ");
    scanf("%d",&n);
    data=(int*)calloc(n,sizeof(int));
    if(data==NULL) {
        printf("Error!!! memory not allocated.");
        exit(0);
    }
    printf("\n");
    for(i=0;i<n;++i) {
       printf("Enter Number %d: ",i+1);
       scanf("%d",data+i);
    }
    for(i=1;i<n;++i) {
       if(*data<*(data+i))
           *data=*(data+i);
    }
    printf("Largest element = %d",*data);
    free(data);
}
```

Les listes

Les listes

Une liste est un ensemble fini d'éléments notée $L = e_1, e_2, ..., e_n$ où e_1 est le premier élément, e_2 le deuxième, etc...

Lorsque n = 0 on dit que la liste est vide.

Les listes servent à gérer un ensemble de données, un peu comme les tableaux. Elles sont cependant plus efficaces pour réaliser des opérations comme l'insertion et la suppression d'éléments. Elles utilisent par ailleurs l'allocation dynamique de mémoire et peuvent avoir une taille qui varie pendant l'exécution. L'allocation (ou la libération) se fait élément par élément.

Les listes peuvent être:
– simplement chaînées
– doublement chaînées
– circulaires (chaînage simple ou double)

Les opérations sur les listes

Les opérations sur une liste peuvent être:

- Créer une liste
- Supprimer une liste
- Rechercher un élément particulier
- Insérer un élément (en début, en fin ou au milieu)
- Supprimer un élément particulier
- Permuter deux éléments
- Concaténer deux listes
- ...

La notion de maillon

L'élément de base d'une liste chaînée s'appelle le maillon. Il est constitué :

– d'un champ de données ;
– d'un pointeur vers un maillon.

Le champ pointeur vers un maillon pointe vers le maillon suivant de la liste. S'il n'y a pas de maillon suivant, le pointeur vaut NULL.

Une liste vide est une liste qui ne contient pas de maillon. Elle a donc la valeur NULL.

La terminologie suivante est généralement employée :

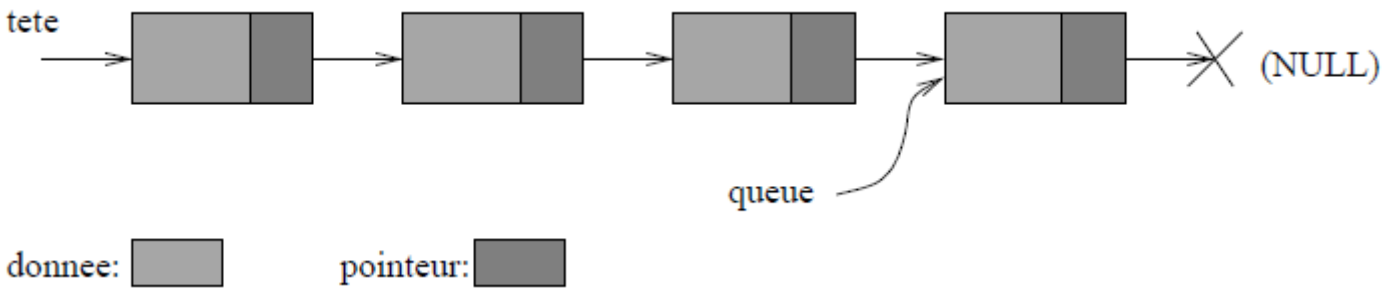
– le premier maillon de la liste est appelé tête ;
– le dernier maillon de la liste est appelé queue.

Les listes simplement chaînées

Une liste simplement chaînée est composée d'éléments distincts liés par un simple pointeur.

Chaque élément d'une liste simplement chaînée est formé de deux parties:
– un champ contenant la donnée (ou un pointeur vers celle-ci)
– un pointeur vers l'élément suivant de la liste.

Le premier élément d'une liste est sa tête, le dernier sa queue. Le pointeur du dernier élément est initialisé à la valeur NULL en C.



Les listes simplement chaînées

Pour accéder à un élément d'une liste simplement chaînée, on part de la tête et on passe d'un élément à l'autre à l'aide du pointeur suivant associé à chaque élément.

En pratique, les éléments étant créés par allocation dynamique, ne sont pas contigus en mémoire contrairement à un tableau. La suppression d'un élément sans précaution ne permet plus d'accéder aux éléments suivants. D'autre part, une liste simplement chaînée ne peut être parcourue que dans un sens (de la tête vers la queue).

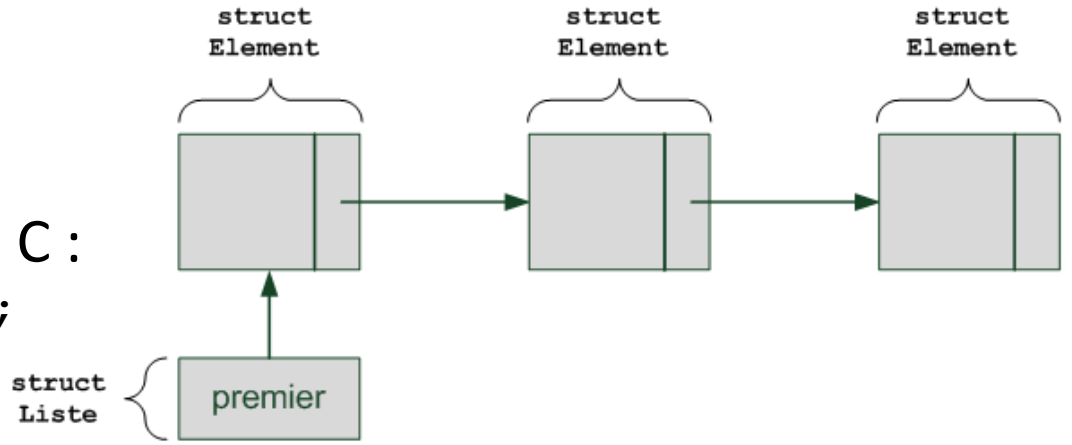
Les listes simplement chaînées

Implémentation de l'élément en C qui correspond à un élément de la liste et que l'on peut dupliquer autant de fois que nécessaire :

```
typedef struct Element Element;
struct Element {
    int nombre;
    Element *suivant;
};
```

Implémentation de la liste en C :

```
typedef struct Liste Liste;
struct Liste {
    Element *premier;
} ;
```



Cette structure Liste contient un pointeur vers le premier élément de la liste. En effet, il faut conserver l'adresse du premier élément pour savoir où commence la liste. Si on connaît le premier élément, on peut retrouver tous les autres en passant d'élément en élément à l'aide des pointeurs suivants.

Initialiser une liste chaînée

La fonction d'initialisation crée la structure de contrôle (la liste) et le premier élément de la liste. Dans cet exemple, le premier élément est initialisé à 0. La liste contient un seul élément alors le pointeur suivant du premier élément pointe à NULL :

```
Liste * initialisation() {
    Liste *liste = (Liste *)malloc(sizeof(Liste));
    Element *element = (Element *)malloc(sizeof(Element));

    if (liste == NULL || element == NULL) {
        exit(EXIT_FAILURE);
    }

    element->nombre = 0;
    element->suivant = NULL;
    liste->premier = element;

    return liste;
}
```

Ajouter un élément au début de la liste

La fonction insertion prend comme paramètres d'entrée une liste et un nombre à ajouter au début de la liste. Après avoir réservé l'espace mémoire pour le nouvel élément, on met à jour le membre nombre de l'élément. L'insertion se fait par un jeu de pointeurs. Le nouvel élément pointera en suivant vers le premier élément de la liste en entrée et la liste change de premier élément.

```
void insertion(Liste *liste, int nvNombre) {
    /* Création du nouvel élément */
    Element *nouveau = malloc(sizeof(Element));
    if (liste == NULL || nouveau == NULL) {
        exit(EXIT_FAILURE);
    }
    nouveau->nombre = nvNombre;

    /* Insertion de l'élément au début de la liste */
    nouveau->suivant = liste->premier;
    liste->premier = nouveau;
}
```

Supprimer un élément du début de la liste

La fonction suppression prend comme paramètres d'entrée une liste. Après avoir vérifié que la liste n'est pas vide, la suppression se fait par un jeu de pointeurs. L'élément à supprimer (le premier) est gardé dans une variable temporaire, la liste pointe dorénavant sur le deuxième élément qui est le suivant du premier élément. A la fin, on libère la mémoire réservée pour le premier élément qui est supprimé.

```
void suppression (Liste *liste) {
    if (liste == NULL) {
        exit(EXIT_FAILURE);
    }

    if (liste->premier != NULL) {
        Element *aSupprimer = liste->premier;
        liste->premier = liste->premier->suivant;
        free(aSupprimer);
    }
}
```

Afficher le contenu d'une liste chaînée

La fonction afficherListe prend comme paramètre d'entrée une liste. Après avoir vérifié que la liste n'est pas vide, on initialise une variable avec le premier élément, et dans une boucle les éléments seront imprimés un à un. On passe d'un élément à un autre en utilisant le pointeur suivant.

```
void afficherListe(Liste *liste) {
    if (liste == NULL) {
        exit(EXIT_FAILURE);
    }

    Element *actuel = liste->premier;

    while (actuel != NULL) {
        printf("%d -> ", actuel->nombre);
        actuel = actuel->suivant;
    }
    printf("NULL\n");
}
```

Exemple

Modifier la fonction Initialisation pour accepter en entrée le nombre à ajouter.

```
Liste * initialisation(int Nb) {
    Liste *liste = malloc(sizeof(*liste));
    Element *element = malloc(sizeof(*element));

    if (liste == NULL || element == NULL) {
        exit(EXIT_FAILURE);
    }

    element->nombre = Nb;
    element->suivant = NULL;
    liste->premier = element;

    return liste;
}
```

Exemple

Etant donné les fonctions présentées ci-avant, créer un programme qui utilise ces fonctions pour créer une liste en se basant sur des entrées de l'utilisateur puis affiche la liste.

```
Liste * initialisation(int Nb);
void insertion(Liste *liste, int nvNombre);
void afficherListe(Liste *liste);
```

Exemple

```
void main() {
    int i,n, nb_a_ajouter;
    Liste *liste;
    printf("Enter total number of elements(1 to 100): ");
    scanf("%d",&n);

       printf("Enter element : ");
       scanf("%d", &nb_a_ajouter);

       liste = initialisation(nb_a_ajouter);

       for (i=1 ; i <n; i++) {
              printf("Enter element : ");
              scanf("%d", &nb_a_ajouter);
              insertion(liste, nb_a_ajouter);
       }

       afficherListe(liste);
       free(liste);
}
```

Exercice

Ecrire une fonction qui retourne la taille de la liste (nombre d'éléments dans la liste).

Exercice

```
int taille_liste (Liste * liste){
    int count = 0;
    if (liste == NULL) {
        exit(EXIT_FAILURE);
    }

    Element *actuel = liste->premier;

    while (actuel != NULL) {
        count++;
        actuel = actuel->suivant;
    }
    return count;
}
```

Exercice

Ecrire une fonction qui ajoute un entier dans une liste triée (ordre ascendant).

Exercice

```
void insertionListeTriee(Liste *liste, int nvNombre) {
        int ajoute = 0;
        Element *nouveau = malloc(sizeof(Element));
        Element *actuel = liste->premier;
        Element *suivant = actuel->suivant;
        if (liste == NULL || nouveau == NULL) {
                exit(EXIT_FAILURE);
        }
        nouveau->nombre = nvNombre;

        if (nvNombre < actuel->nombre) {
                nouveau->suivant = actuel;
                liste->premier = nouveau;
                ajoute = 1;
        } else {
                while (!ajoute && suivant != NULL) {
                        if (nvNombre < suivant->nombre ) {
                                actuel->suivant = nouveau;
                                nouveau->suivant = suivant;
                                ajoute = 1;
                        }
```

Exercice

```
                        else {
                                actuel = suivant;
                                suivant = actuel->suivant;
                        }
                }
                if (!ajoute) {
                        actuel->suivant = nouveau;
                        nouveau->suivant = NULL;
                }
        }
}
```

Exercice

Ecrire un programme pour trier une liste chaînée en utilisant le tri à bulles

Exercice

```
void liste_trier(Liste * liste) {
    int i, permutation = 1;
    Element *actuel, *prec, *suiv;
    // cas liste vide
    if (liste == NULL || liste->premier == NULL) {
        exit(EXIT_FAILURE);
    }
    // cas liste d'un seul element
    if (liste->premier->suivant == NULL) {
        return;
    }
    // cas liste a deux elements ou plus
    while (permutation == 1) {
            permutation = 0;
            actuel = liste->premier;
            suiv = actuel->suivant;
            // traitement de la comparaison des deux premiers elements
            if (actuel->nombre > suiv->nombre) {
                /* Il faut une permutation */
                actuel->suivant = suiv->suivant;
                suiv->suivant = actuel;
                liste->premier = suiv;
                permutation = 1;
            }
```

Exercice

```
            // continuons a avancer
            prec = actuel;
            actuel = suiv;
            suiv = suiv->suivant;

            // repeter la comparaison entre chaque deux elements
            // jusqu'a la fin de la liste
            while (suiv != NULL) {

                if (actuel->nombre > suiv->nombre) {
                    /* Il faut une permutation */
                    actuel->suivant = suiv->suivant;
                    suiv->suivant = actuel;
                    prec->suivant = suiv;
                    permutation = 1;
                }
                // continuons a avancer
                prec = actuel;
                actuel = suiv;
                suiv = suiv->suivant;
            }
        }
    }
```

Exercice

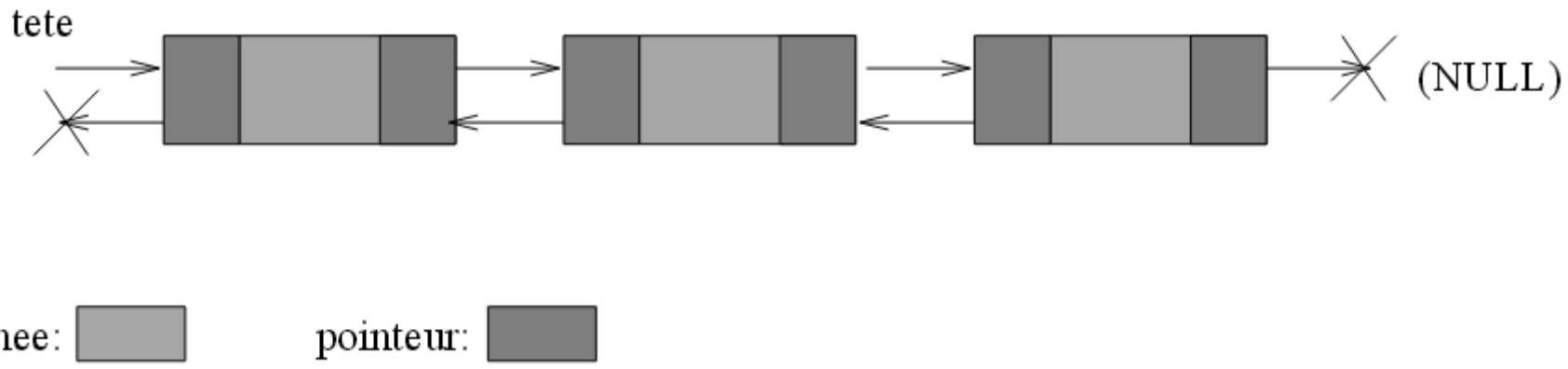
```
void main() {
    int i,n, nb_a_ajouter;
    Liste *liste;
    printf("Enter total number of elements(1 to 100): ");
    scanf("%d",&n);

    printf("Enter element : ");
    scanf("%d", &nb_a_ajouter);
    //initialiser la liste avec un seul element
    liste = initialisation(nb_a_ajouter);
    //inserer des elements dans la liste
    for (i=1 ; i <n; i++) {
        printf("Enter element : ");
        scanf("%d", &nb_a_ajouter);
        insertion(liste, nb_a_ajouter);
    }
    //afficher la liste avant tri
    afficherListe(liste);
    liste_trier(liste); // trier la liste
    //afficher la liste apres tri
    afficherListe(liste);
    free(liste); //liberer mémoire reservee pour la liste
}
```

Listes doublement chaînées

Les listes doublement chaînées sont constituées d'éléments comportant trois champs:

- Un champ contenant la donnée (ou un pointeur vers celle-ci)
- Un pointeur vers l'élément suivant de la liste
- Un pointeur vers l'élément précédent de la liste.

Elles peuvent donc être parcourues dans les deux sens.



Listes doublement chaînées

Implémentation de l'élément en C qui correspond à un élément de la liste doublement chaînée et que l'on peut dupliquer autant de fois que nécessaire :

```
typedef struct ElementD ElementD;
struct ElementD {
    int nombre;
    ElementD *suivant;
    ElementD *precedent;
};
```

Implémentation de la liste en C :

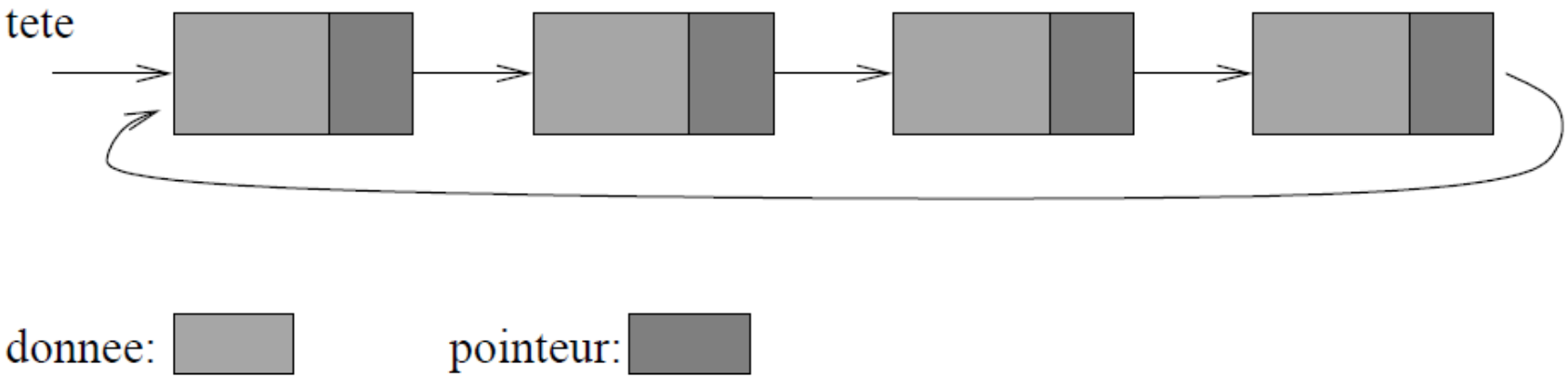
```
typedef struct ListeD ListeD;
struct ListeD {
    ElementD *premier;
} ;
```

Listes circulaires

Une liste circulaire peut être simplement ou doublement chaînée.

Sa particularité est de ne pas comporter une queue.

Le dernier élément de la liste pointe vers le premier. Un élément possède donc toujours un suivant.



Exercice

Ecrire les structures Liste et Element d'une liste circulaire

Ecrire les fonctions suivantes pour les listes circulaires:

1. Initialisation avec un nombre
2. Insertion au début de la liste
3. Suppression du début de la liste
4. Calcul de la taille d'une liste
5. Affichage d'une liste

Exercice

```
typedef struct Element Element;
struct Element {
    int nombre;
    Element *suivant;
};

typedef struct Liste Liste;
struct Liste {
    Element *premier;
} ;

Liste * initialisation(int Nb) {
    Liste *liste = malloc(sizeof(*liste));
    Element *element = malloc(sizeof(*element));

    if (liste == NULL || element == NULL) {
        exit(EXIT_FAILURE);
    }
    element->nombre = Nb;
    element->suivant = element;
    liste->premier = element;

    return liste;
}
```

Exercice

```
void insertion(Liste *liste, int nvNombre) {
    Element *nouveau = malloc(sizeof(Element));
    if (liste == NULL || nouveau == NULL) {
        exit(EXIT_FAILURE);
    }
    nouveau->nombre = nvNombre;

    // cas de liste vide
    if (liste->premier == NULL) {
        nouveau->suivant = nouveau;
        liste->premier = nouveau;
        return;
    }

    // cas de liste non vide
    // Insertion de l'élément au début de la liste
    nouveau->suivant = liste->premier;
    Element *actuel = liste->premier;
    //chercher le dernier élément
    while (actuel->suivant != liste->premier)
        actuel = actuel->suivant;
    actuel->suivant = nouveau;
    liste->premier = nouveau;
}
```

Exercice

```
void suppression(Liste *liste) {
    if (liste == NULL) {
        exit(EXIT_FAILURE);
    }

    // cas de liste non vide
    if (liste->premier != NULL) {
        Element *aSupprimer = liste->premier;

        // cas de liste contient 1 seul element
        if (liste->premier->suivant == liste->premier)
            liste->premier = NULL;
        // cas de liste contient plus qu'un element
        else {
            //chercher le dernier élément
            Element *actuel = liste->premier;
            while (actuel->suivant != liste->premier)
                actuel = actuel->suivant;
                actuel->suivant = liste->premier->suivant;
                liste->premier = liste->premier->suivant;
        }
        free(aSupprimer);
    }
}
```