TP8 - Allocation dynamique, structure d'arbre binaire

Le langage C gère automatiquement la mémoire pour les variables statiques et automatiques. Par le biais des pointeurs, il est également possible de manipuler directement l'allocation et la libération d'espace mémoire, en utilisant des fonctions prédéfinies de gestion dynamique de la mémoire.

Intérêt de la gestion dynamique

Les objets du C (déclarés par l'intermédiaire de variables) peuvent se classer en trois catégories :

1. les objets statiques,
2. les objets automatiques,
3. les objets dynamiques.

Les objets statiques occupent un emplacement parfaitement défini à la compilation.

Les objets automatiques n'ont pas d'emplacement défini *a priori*, ils sont créés lors de l'entrée dans le bloc (ou sous programme) où ils sont définis et détruits à la sortie. Ils sont souvent gérés en utilisant une pile qui croît et décroît selon les besoin du programme.

Les objets dynamiques n'ont pas non plus d'emplacement *a priori*, mais leur création (allocation) ou leur destruction (libération) dépend de demandes explicites gérées par le programmeur.

L'emploi d'objets statiques est transparente au programmeur qui n'a pas à se soucier de leur gestion en mémoire. Cependant, elle ne permet pas de manipulation efficace de tableau de dimension variable ou de structure de données perpétuellement changentes et donc la taille est impossible à prédire (liste chaînées, arbre binaire, ...).

Exemple introductif

Considérons le programme C suivant :

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

void main()
{
  double *nombre; /* nombre est un POINTEUR vers un flottant double */

  *nombre=1.254789;  /* il reçoit une valeur... */
```

```
  *nombre = 8 * *nombre + 256.4587; /* que l'on modifie... */
  printf("Nombre = %f\n", *nombre); /* et que l'on imprime */
}
```

L'exécution de ce programme produit : **Bus error**. En effet l'espace mémoire référencé par un pointeur est par essence dynamique. Le pointeur doit recevoir l'adresse d'un espace déjà existant, une variable statique par exemple, ou l'adresse d'un nouvel espace mémoire demandé à l'aide d'une fonction prédéfinie : la fonction **malloc**. Modifions le programme précédent :

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

void main()
{
  double *nombre; /* nombre est un POINTEUR vers un flottant double */
  nombre = malloc(sizeof(double)); /* demande explicite d'allocation */
  *nombre=1.254789;  /* il reçoit une valeur... */
  *nombre = 8 * *nombre + 256.4587; /* que l'on modifie... */
  printf("Nombre = %f\n", *nombre); /* et que l'on imprime */
}
```

Le programme produit : **Nombre = 266.497012**

La fonction **malloc** alloue dans la mémoire libre de la machine un emplacement ayant la taille précisée par son argument et renvoie l'adresse de cet espace. La fonction sizeof appliquée à un type prédéfini ou créé par le programmeur donne la taille correspondante en octets. Cela assure la portabilité, en effet dans le cas de l'exemple, la taille du type double peut varier selon les architectures ou les compilateurs. Cet emplacement mémoire alloué par malloc ne sera déalloué que par un appel à la fonction **free** ou la fin du programme.

Le type renvoyé par **malloc** est **void ***. Certain compilateurs vont signaler que l'affectation :

```
nombre = malloc(sizeof(double))
```

engendre une conversion implicite de type. On peut préciser cette convertion dans le programme par :

```
nombre = (double *) malloc(sizeof(double));
```

Tests à réaliser

Tapez et testez les différents programmes donnés ci-dessus.

1 Arbre binaire

Nous allons utiliser maintenant la gestion dynamique de la mémoire pour impléménter un arbre binaire en C. Le premier point à aborder est celui de la définition de la structure de données pour mettre en oeuvre cet objet.

Type récursif : arbre binaire de recherche

Un arbre binaire est constitué de :

1. une information : int pour simplifier,
2. un sous arbre gauche qui est lui-même un arbre binaire,
3. un sous-arbre droit, lui aussi étant un arbre binaire.

Ce concept peut se concrétiser en C par la déclaration de structure suivante :

```
struct tabr
{
  int element;
  struct tabr *sag;
  struct tabr *sad;
};
```

Pour pouvoir manipuler aisément cette structure on crée un type :

```
typedef struct tabr *ABR;
```

Ces déclarations seront écrite dans un fichier header (ex. abr.h) avec les autres #**define** pouvant être utiles à l'ensemble du programme (pour TRUE et FALSE par exemple). Un autre fichier (ex. abr.c) contiendra le code des sous-programmes chargés de créer, initialiser, modifier un arbre binaire de type ABR. On suppose que les arbres manipulés dans ce TP sont des arbres binaires de recherche. C'est à dire que les informations des sommets du sous-arbre gauche sont toujours inférieures ou égale à celle du sommet père, et que les informations du sous-arbre droit sont toujours strictement supérieures à celle du sommet père.

Exemple de sous programmes utilisant ABR

On crée donc un fichier abr.c avec le code suivant :

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

/* constructeur d'arbre */

ABR CreerArbre(void)
{
  return NULL; /* pointeur vide prédéfini */
}

/* test arbre vide */

int EstVide(ABR arbre)
{
  return (arbre == NULL);
}

/* on ajoute une information */
```

```
ABR Ajouter(ABR arbre,int elt)
{
  if (EstVide(arbre))
    {
      arbre = (ABR) malloc(sizeof(struct tabr)); /* ALLOCATION */
      arbre->element=elt;
      arbre->sag = NULL;
      arbre->sad = NULL;
      return arbre;
    }
  else
    {
      ABR pere,auxp;

      auxp=arbre;pere=NULL;
      while (auxp!=NULL)
{
  if (auxp->element ==elt)
    break;
  else
    {pere=auxp;
    if (auxp->element > elt)
      auxp=auxp->sag;
    else  /* auxp->element<elt */
      auxp=auxp->sad;
    }/*  end else */
}/* end while */
      if (auxp == NULL)
{
  auxp=(ABR) malloc(sizeof(struct tabr)); /* on se taille une nouvelle cellule */
  auxp->element=elt;      /* on y case l'élément */
  auxp->sag=NULL;
  auxp->sad=NULL;
  if (pere->element>elt)
    pere->sag=auxp;
  else
    pere->sad=auxp;
}
      return arbre;
    }
}/* Ajouter */

/* impression infixe de l'arbre */

void Imprimer(ABR arbre)
{
  if (arbre!=NULL)
    {
      Imprimer(arbre->sag);
```

```
      printf("%d\n",arbre->element);
      Imprimer(arbre->sad);
    }
}
```

On crée maintenant un fichier main.c :

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

main()
{
  ABR MonArbre;

  MonArbre = CreerArbre();
  if (EstVide(MonArbre))
    printf("Cet arbre est vide\n");
  else printf("Cet arbre n\'est pas vide\n");

  MonArbre= Ajouter(MonArbre,12);
  MonArbre= Ajouter(MonArbre,45);
  MonArbre= Ajouter(MonArbre,30);

  if (EstVide(MonArbre))
    printf("Cet arbre est vide\n");
  else printf("Cet arbre n\'est pas vide\n");

  Imprimer(MonArbre);
}
```

Le point le plus délicat du code précédent est l'ajout d'une information. C'est là qu'intervient l'allocation dynamique d'un espace mémoire correspondant à un nouveau sommet de l'arbre, et sa connection à l'arbre existant.

Si l'arbre est vide (arbre === NULL), on demande l'allocation d'un espace de la taille de la structure tabr :

```
arbre = (ABR) malloc(sizeof(struct tabr)); /* ALLOCATION */
```

Ce nouvel espace doit être initialisé : ses sous-arbres droit et gauche sont vides (NULL) et son information est celle que l'on veut ajouter.

Si l'arbre n'est pas vide, on doit rechercher la place de cet élément dans la structure d'arbre existante, ce qui suppose un parcours de l'arbre. Quand cet emplacement est trouvé il faut allouer un espace pour un nouveau sommet, l'initialiser comme précédemment, et insérer le sommet à sa place en modifiant les pointeurs.

Travail à réaliser

Q1 Tapez abr.h, abr.c et main.c, testez-les,

Q2 Créez un Makefile pour gérer la compilation de l'ensemble du programme,

Q3 Ajoutez un sous-programme de suppression d'une information.