

TP 6 : Langage C : structures récursives en C

On utilisera la ligne de commande suivante pour compiler :

```
1 gcc -Wall -Wextra -Werror -Wvla -fsanitize=address,undefined -o tp6.exe tp6.c
```

Shell

Ce TP comporte cinq exercices :

- Les Exercices 1 et 2 sont obligatoires et portent sur la manipulation de listes chaînées et d'arbres binaires en C.
- Les Exercices 3 à 5 sont facultatifs et portent respectivement sur les listes chaînées circulaires, les arbres binaires de recherche et les arbres avec pointeur vers le père.

Exercice 1 – Listes chaînées en C

Pour représenter une collection d'éléments en C, on sait déjà utiliser les tableaux : ils permettent de représenter une collection de taille fixe, avec un accès immédiat à chaque élément. Cependant, on peut sacrifier cette seconde propriété (accès immédiat) pour relâcher la première (taille fixe) en utilisant une structure de donnée appelée *liste chaînée*.

Une liste chaînée est une structure de donnée définie par induction :

- une liste vide est une liste chaînée,
- si t est un élément et q une liste chaînée, alors la structure $t :: q$ (lire « t cons q ») est une liste chaînée dont la tête est t et la queue est q.

Cette définition ressemble à celle de « chaîne » décrite en cours : on y a ajouté le fait que dans chaque « maillon » de notre chaîne, on stocke une valeur t (la tête) et un pointeur vers le maillon suivant q (la queue). Ainsi, une liste chaînée est soit vide, soit un élément suivi d'une liste chaînée. L'objectif de cet exercice est d'implémenter la structure de donnée de liste chaînée par le type suivant en C :

```
1 struct Maillon {
2     double tete;
3     struct Maillon* queue;
4 };
5 typedef struct Maillon liste;
```

C

Dans la suite, on manipulera toujours une liste comme un pointeur de type `liste*`. Une liste vide est alors représenté par le pointeur `NULL`. Par exemple, pour représenter la liste [1.; 10.; 5.], on le représentera comme suis :

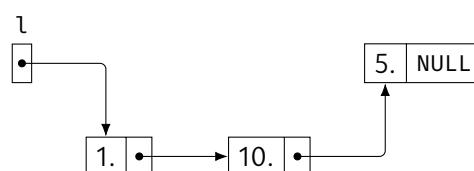


FIGURE 1 : Représentation en mémoire de la liste [1.; 10.; 5.]

► **Question 1** Implémenter la fonction `liste* cons(double t, liste* q)` qui crée sur le tas une nouvelle liste dont la tête est t et la queue est q, et renvoie un pointeur vers le premier maillon de

cette liste. Par exemple, l'appel `cons(1., cons(10., cons(5., NULL)))` doit renvoyer un pointeur vers une liste représentant [1.; 10.; 5.]. On n'oubliera pas de vérifier que les allocations mémoire se sont bien passées.

► **Question 2** Proposer à l'écrit un schéma représentant en mémoire les listes `l = [1.]`, `cons(2., l)` et `cons(3., l)`. Combien de cases mémoires sont utilisées pour représenter ces trois listes ?

★ **Question 3** En déduire une fonction `liste* depuis_tableau(double* tab, int n)` qui crée une liste contenant les éléments du tableau `[tab[0], ..., tab[n - 1]]` dans le même ordre.

► **Question 4** Implémenter les fonctions `hd` et `tl` renvoyant respectivement la tête et la queue d'une liste. On utilisera une assertion pour lever une exception si la liste en entrée est vide.

► **Question 5** Écrire une fonction `void liberer_liste(liste* l)` qui libère toute la mémoire utilisée par `l`.

★ **Question 6** Supposons que l'on dispose d'une fonction de prototype `void f(double x)` qui effectue une opération sur un `double` et ne renvoie rien. Écrire une fonction `void iter_liste(liste* l)` qui exécute dans l'ordre de la `liste*` la fonction `f` sur les éléments de `l`. *On pourra la tester avec une fonction f qui affiche x.*

► **Question 7** Écrire une fonction `int longueur(liste* l)` qui calcule la longueur d'une liste.

► **Question 8** Écrire une fonction `double somme(liste* l)` calculant la somme des éléments de `l`.

★ **Question 9** En déduire une fonction `double* vers_tableau(liste* l)` qui stocke dans un tableau les éléments de `l`.

► **Question 10** Écrire une fonction `bool sont_egales(liste* l1, liste* l2)` testant si deux listes contiennent les mêmes éléments.

Les fonctions suivantes demandent un soin particulier et d'être testées avec attention.

★ **Question 11** Écrire une fonction `bool est_triee(liste* l)` vérifiant qu'une liste est triée.

★ **Question 12** Écrire une fonction `liste* insere(double x, liste* l)` qui, pour une liste triée `l`, renvoie une liste triée contenant les mêmes éléments que `l` plus une occurrence de `x`. On n'utilisera qu'une seule allocation mémoire supplémentaire.

★ **Question 13** Faire un schéma de l'état de la mémoire avant et après une insertion `l1 = insere(x, l)`, dans les cas suivants :

- Si la liste `l` est vide,
- Si la liste `l` est non vide et qu'on insère l'élément en première position,
- Si la liste `l` est non vide et qu'on insère l'élément à l'intérieur de `l`.

★ **Question 14** En déduire un comportement potentiellement non désiré de la fonction précédente.

★ **Question 15** Implémenter le tri par insertion sur le type `liste`. Elle ne doit pas modifier la liste initiale, mais renvoyer une nouvelle liste.

Exercice 2 – Manipulation d'arbres binaires en C

On peut manipuler des arbres binaires en C grâce un type enregistrement de la forme suivante. Ici, on chaque nœud porte une étiquette de type `int`.

```

1 struct Noeud {
2     int valeur;
3     struct Noeud* gauche;
4     struct Noeud* droit;
5 };
6 typedef struct Noeud arbre;
```

C

Comme pour les listes chaînées, on manipulera toujours des arbres sous forme de pointeur de type `arbre*`. Ainsi, un arbre vide sera représenté par `NULL`. Pour commencer à tester ses fonctions, on peut retirer temporairement l'option `-fsanitize=address` de la commande de compilation.

► **Question 1** Écrire une fonction `arbre* construire_arbre(int valeur, arbre* gauche, arbre* droit)` qui renvoie un arbre binaire de racine `valeur` et de fils `gauche` et `droit` droit.

► **Question 2** Écrire une fonction `int hauteur(arbre* a)` qui renvoie la hauteur de l'arbre `t`. De même, écrire une fonction `int taille(arbre* a)` qui renvoie le nombre de nœuds de l'arbre `t`.

► **Question 3** Écrire une fonction `int est_feuille(arbre* a)` qui renvoie 1 si `t` est une feuille et 0 sinon.

► **Question 4** Écrire une fonction `int nb_feuilles(arbre* a)` qui renvoie le nombre de feuilles de l'arbre `t`.

► **Question 5** Écrire une fonction `int somme_feuilles(arbre* a)` qui renvoie la somme des étiquettes des feuilles de l'arbre `t`.

► **Question 6** Écrire une fonction `bool est_strict(arbre* a)` qui renvoie `true` si `t` est un arbre binaire strict et `false` sinon.

► **Question 7** Écrire une fonction `bool est_dans_arbre(int valeur, arbre* a)` qui renvoie `true` si `valeur` est dans l'arbre `t` et `false` sinon.

► **Question 8** Écrire une fonction `void ajoute_valeur(int valeur, arbre* a)` qui ajoute `valeur` à l'étiquette de chaque nœud de l'arbre `t`.

► **Question 9** Écrire une fonction `void miroir(arbre* a)` qui modifie en place l'arbre `t` en échangeant les fils gauches et droits de chaque nœud.

► **Question 10** Écrire une fonction `void liberer(arbre* a)` qui libère la mémoire allouée pour l'arbre `t`.

Exercice 3 – ★ Listes chaînées circulaires

Pour cet exercice seulement, on ignorera les fuites mémoires. Je vous conseille cependant de garder l'option `-fsanitize=address` pour la compilation.

Puisque chaque champ `queue` des maillons d'une liste chaînée est modifiable, on peut commencer à

faire des bêtises de ce genre :

```
1 liste* l = cons(1, cons(2, cons(3, NULL)));
2 l->queue->queue->queue = l;
```

C

► **Question 1** Représenter graphiquement l'état de la liste `l` en mémoire après l'exécution de ces instructions.

► **Question 2** Que se passe-t-il si on exécute `longueur(l)` sur cette liste ?

► **Question 3** Écrire une fonction `bool est_circulaire(liste* l)` qui renvoie `true` si la liste `l` est circulaire (c'est-à-dire que l'un de ses maillons pointe vers le maillon de départ) et `false` sinon (si la liste est finie et se termine par un pointeur `NULL`).

► **Question 4** Écrire une fonction `void longueur_circulaire(liste* l)` qui renvoie la longueur d'une liste circulaire.

Supposons qu'après les deux instructions ci-dessus, on exécute l'instruction suivante :

```
1 l = cons(-1, cons(0, l));
```

C

► **Question 5** Représenter graphiquement l'état de la liste `l` en mémoire après l'exécution de cette instruction.

► **Question 6** Que se passe-t-il si on exécute `longueur(l)` sur cette liste ?

Pour détecter si une liste est ultimement circulaire (c'est-à-dire que son « dernier » maillon pointe vers un maillon précédent, mais pas forcément le premier), on peut utiliser l'algorithme de Floyd (aussi appelé « algorithme du lièvre et de la tortue »). L'idée est d'utiliser deux pointeurs `tortue` et `lievre` qui parcourent la liste, le premier avançant d'un maillon à la fois, le second de deux maillons à la fois. Si la liste est finie, le lièvre atteindra `NULL` avant la tortue. Si la liste est circulaire, le lièvre finira par rattraper la tortue.

► **Question 7** Implémenter cet algorithme dans une fonction `bool est_ultimement_circulaire(liste* l)`.

★ Revenons aux listes circulaires. On peut utiliser cette structure de donnée pour représenter une file, c'est-à-dire permettant de stocker une collection d'éléments où l'on peut appliquer les opérations suivantes :

- `file* creer_file()` : crée une file vide,
- `void enfiler(file* f, double x)` : ajoute l'élément `x` à la fin de la file `f`,
- `double defiler(file* f)` : enlève et renvoie l'élément en tête de la file `f`,
- `bool est_vide(file* f)` : renvoie `true` si la file `f` est vide, `false` sinon.
- `void liberer_file(file* f)` : libère la mémoire allouée pour la file `f`.

Pour cela, l'idée est de représenter une file par un pointeur vers le dernier maillon d'une liste circulaire. Ainsi, le maillon suivant de ce pointeur sera la tête de la file. Une file vide sera représentée par le pointeur `NULL`.

► **Question 8** Définir le type `file` en C et écrire les fonctions ci-dessus.

Exercice 4 – ★ Implémentation des ABR en C

Les arbres binaires de recherche^a peuvent être manipulés en C. On utilisera le type suivant :

```

1 struct abr {
2     int val;
3     struct abr *fg;
4     struct abr *fd;
5 };
6 typedef struct abr abr;
```

C

► **Question 1** Écrire une fonction `abr* nœud(int val, abr *fg, abr *fd)` qui renvoie un pointeur vers un nouveau nœud d'ABR.

► **Question 2** Écrire une fonction `bool est_abr(abr *a)` qui renvoie `true` si a est un arbre binaire de recherche, `false` sinon.

► **Question 3** Écrire une fonction `bool recherche(int x, abr *a)` qui renvoie `true` si x est présent dans l'ABR a, `false` sinon.

► **Question 4** Écrire une fonction `void liberer_abr(abr *a)` qui libère la mémoire allouée sur le tas pour l'arbre a.

► **Question 5** Écrire une fonction `abr* insere(int x, abr *a)` qui insère l'entier x dans l'ABR a si x n'y est pas déjà présent et renvoie un pointeur vers la racine de l'arbre modifié^b. Si x est déjà présent, on renvoie simplement un pointeur vers la racine de l'arbre.

★ **Question 6** Proposer une stratégie pour supprimer un entier x d'un ABR a si x y est présent.

★ **Question 7** Implémenter la recherche, l'insertion et la suppression impérativement.

a. https://fr.wikipedia.org/wiki/Arbre_binaire_de_recherche pour ceux qui n'auraient pas fait de NSI en terminale, ou ceux qui voudraient une piqûre de rappel.

b. qui sera donc a sauf s'il est `NULL`

Exercice 5 – ★ Arbres à trois pointeurs

Dans les ?? et Exercice 2, on a vu deux représentations des arbres en C identiques en mémoire (une valeur et deux pointeurs par nœud) permettant, selon leur interprétation, de représenter les arbres binaires ou les arbres généraux. Dans les deux cas, on peut compléter les types avec un troisième pointeur indiquant l'adresse du père du nœud en mémoire (le père de la racine étant `NULL`). On obtient alors les types suivants pour les arbres binaires :

```

1 typedef struct arbre_avec_pere {
2     int valeur;
3     struct arbre_avec_pere* gauche;
4     struct arbre_avec_pere* droit;
5     struct arbre_avec_pere* pere;
6 } arbre_avec_pere;
```

C

L'un des avantages de cette représentation est de permettre de parcourir l'arbre purement itérativement, sans utiliser de pile.

► **Question 1** Écrire une fonction `int taille(arbre_avec_pere* t)` qui renvoie la taille de l'arbre `t` sans utiliser d'appels récursifs.

► **Question 2** Écrire une fonction `int hauteur(arbre_avec_pere* t)` qui renvoie la hauteur de l'arbre `t` sans utiliser d'appels récursifs.

► **Question 3** Écrire une fonction `void parcours_prefixe(arbre_avec_pere* t)` qui affiche les étiquettes des nœuds de l'arbre `t` en parcours préfixe sans utiliser d'appels récursifs.

Évidemment, comparé à la représentation à deux pointeurs par nœud, ce pointeur supplémentaire par nœud peut être considéré comme un espace auxiliaire linéaire en la taille de l'arbre, ce qui est identique à la complexité mémoire de la pile d'appel utilisée pour les parcours récursifs. La constante associée sera cependant plus faible. De plus, cette représentation permet de manipuler des nœuds à l'intérieur d'un arbre sans oublier leur contexte (le fait qu'ils ne sont qu'un sous-nœud d'un plus grand arbre), alors que les représentations précédentes permettaient de raisonner sur chaque nœud comme un sous-arbre enraciné en ce nœud.
