TD 3
Types abstraits de données
Listes

Polytech Marseille - IRM 3ème année
Alexandra Bac

Algorithmique et structures de données
4h TD - 4h TP

1 TD

1.1 Type abstrait de données listes (séance de TD 1)

Dans cette partie, on considère exclusivement le TAD Listes (une liste est définie par son contenu) indépendamment de toute implémentation. Donc deux listes seront considérées comme identiques si elles contiennent les mêmes données et, par exemple, ajouter un élément à une liste consiste à renvoyer une liste contenant une donnée supplémentaire en tête, les autres restant identiques.

Exercice 1 (Fonctions sur les listes)**.** Toute les questions seront implémentées en appelant exclusivement les fonctions du type abstrait (aucuns pointeurs) :

(i) fonction récursive `longueur` calculant la longueur d'une liste

(ii) fonction récursive `rechercher` recherchant si un élément x est présent dans la liste

(iii) fonctions récursives `copier` renvoyant une copie ou une copie inversée de la liste (pour l'inversions la liste $[1; 2; 3; 4]$ donne $[4; 3; 2; 1]$)

(iv) $i$ème élément (pour toutes les fonctions suivantes, vous proposerez une gestion cohérente des erreurs via les exceptions quand c'est nécessaire) :

  (a) fonction récursive `ith` renvoyant le $i$ème élément de la liste

  (b) fonction récursive `insert_ith` ajoutant un élément entre le $i$ème et le $i+1$ème élément de la liste

  (c) (optionnel) fonction récursive `del_ith` supprimant le $i$ème élément de la liste

(v) fonction récursive `positions` renvoyant la liste des indices de position d'un élément x dans une liste

(vi) fonction récursive supprimant un élément x dans une liste :

  (a) `del_1_elt` supprime la première occurrence de x

  (b) `del_all_elt` supprime toutes les occurrence de x

Exercice 2 (Bonus)**.** On supposera ici que nos fonctions peuvent renvoyer plusieurs variables simultanément (pour les codes en C, il faudrait donc travailler sur le passage d'arguments ...).

(i) Ecrire une fonction

```
(liste, liste) moities(liste l1, liste l2)
```

qui, appelée sur une liste `l` (passée deux fois en argument : `moities(l,l)`) renvoie le couples des deux moitiés de `l` et cela en un seul passage récursif. Indication : on passe deux copies de `l` pour pouvoir les parcourir à des vitesses différentes (par exemple 2 éléments par 2 éléments).

(ii) En déduire une fonction

`(bool, liste) palindrome(liste l1, liste l2)`

qui, appelée sur une liste `l` (passée deux fois en argument : `palindrome(l,l)`) teste si la liste contient un palindrome.

1.2 Implémentation par listes chaînées (séance de TD 2)

Exercice 3 (Implémentation du type abstrait des listes par des listes chaînées en C)**.** On utilise la structure de données vue en cours pour implémenter les listes chaînées :

```
struct zMaillon {
   Elt elt ;
   struct zMaillon *suiv ; } ;
typedef struct zMaillon Maillon ;
typedef Maillon * Liste ;
```

(i) Implémenter toutes les opérations du type abstrait de données.

(ii) Réfléchir à la question de la désallocation mémoire ... Quelles fonctions sont naturelles à définir ? Pourquoi sont-elles insuffisantes dans le cas général ?

Exercice 4 (Fonctions sur les listes)**.** Ecrire les fonctions suivantes (cette fois manipulant les pointeurs pour améliorer leur efficacité) :

(i) Ecrire une fonction `ajouter_fin` (quel pointeur cette fonction doit-elle recevoir en argument pour pourvoir s'exécuter en temps constant ?).

(ii) Ecrire une fonction itérative inversant la liste sans créer de nouveaux maillons (simplement en modifiant les chaînages).

2 TP

On implémentera les préconditions et exceptions au moyen de la fonction `assert` du fichier d'entête `assert.h`.

2.1 Séance de TP 1

Implémenter en C ces fonctions sous forme de module (voir notes de cours) en utilisant l'implémentation des listes fournie dans le module `listes` fourni en matériel de TP (implémentation par tableaux avec réallocation automatique).

Vous implémenterez donc un module `liste_ext` contenant toutes vos fonctions et utilisant le module `listes`.

Le fichier `main` fera appel à ces deux modules, les testera et sera compilé de manière séparée.

Pour commencer. Compilez le Makefile fourni. Est-ce que les liste `l` et `l2` correspondent bien à ce que l'on pourrait attendre ?

Ensuite vous implémenterez les fonctions de l'exercice 1 (chrono TP, donc vous travaillez avec votre groupe !)

Bonus. Lisez avec attention le module `listes` :

(i) comment la réallocation automatique est-elle gérée ?

(ii) construire dans le main la liste $[1; 2; 3]$. Quelles sont exactement les copies faites des structures et à quel moment. Que devient le tableau des éléments ? Est-il dupliqué ? Que fait le mécanisme de passage par valeurs aux appels de la fonction ?

(iii) quelle différence entre `tail` et `tail_share` ? Laquelle faut-il utiliser ? Quand ? Donnez un exemple de construction de liste dans lequel on obtiendrait des résultats différents.

(iv) que pensez-vous de la gestion mémoire ?

2.2 Séance de TP 2

Implémenter en C un module de listes `listes` chaînées (qui devra utiliser les prototypes de `listes.h` du TP 1).

L'objectif sera que vous puissiez recompiler votre code du TP1 en substituant votre implémentation par liste chaînées au module fourni au TP 1 **sans rien modifier d'autre**.

Vous implémenterez par ailleurs :

```
Liste copy(Liste) ;
void free_link(Maillon *) ;
void free_list(Liste) ;
```

Pour pouvoir correctement libérer la mémoire, que pensez-vous de déclarations telles que :

```
Liste l1 = add(3, add(2, add(1, create_empty_list()))) ;
Liste l2 = add(4,l1) ;
Liste l3 = add(5, tail(tail(l2))) ;
```

En utilisant la fonction `copy`, écrivez des allocations permettant ensuite une libération "propre" de la mémoire.

2.3 Séance de TP 3 (si vous avez fini les listes ...)

Il est parfois nécessaire de pouvoir circuler de manière bi-directionnelle dans les listes. De telles structures sont appelées des listes doublement chaînées. On peut les définir de la manière suivante :

```
struct zMaillon {
   elt Elt ;
   struct zMaillon *suiv, *prec ; } ;
typedef struct zMaillon Maillon ;
typedef Maillon * Liste ;
```

On conservera donc un seul pointeur (sur l'élément de tête) de la liste. Attention, le cas d'une liste vide devient alors systématiquement un cas particulier (pourquoi ?).

Exercice 5. A partir de votre module de listes, créez un module de listes doublement chaînées. Avant de vous lancer dans l'implémentation, réfléchissez aux questions suivantes :

(i) Comment est représentée la liste vide ? Comment lui ajouter un maillon ?

(ii) Pour renvoyer la queue de la liste, suffit-il de renvoyer un pointeur sur le suivant ? Proposez deux solutions distinctes permettant de faire un retour cohérent.

(iii) Des déclarations telles que l'exemple ci-dessus (`l1`, `l2`, `l3`) sont-elles encore possibles ?

En déduire une implémentation cohérente des listes doublement chaînées et en analyser les limites et les faiblesses.

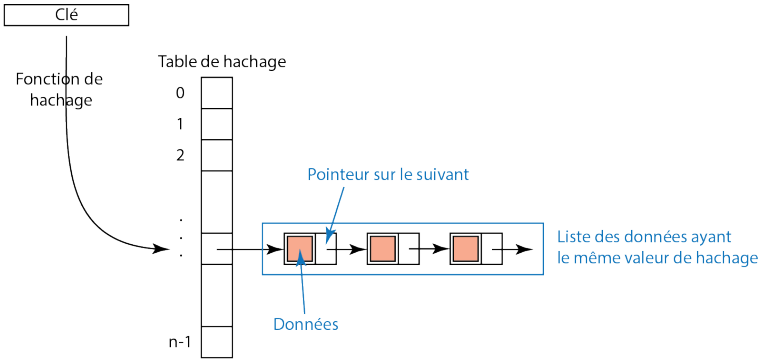
3 Tables de hachage

Exercice 6 (Tables de hachage)**.** Une table de hachage (aussi appelée tableau associatif) peut intuitivement être vue comme un tableau généralisé dans lequel les indices ne sont plus nécessairement des entiers (mais peuvent par exemple être des chaînes de caractères). L'idée est de pouvoir réaliser des opérations du type :

```
T["ceci"] <- data1 ;
T["cela"] <- T["ceci"] ;
```

La clé de hachage (ici une chaîne de caractères) doit être unique.

Bien évidemment, de tels tableaux ne sont pas une structure native, et il faut donc mettre en place une implémentation adaptée. Techniquement, une table de hachage repose sur un tableau standard en interne. Et la conversion "chaîne de caractères" $\rightarrow$ indice entier est réalisée en amont par une fonction de hachage. La figure suivante illustre le fonctionnement :



Si on note f la fonction de hachage, l'opération :

```
T["ceci"] <- data1 ;
```

correspond donc concrètement à l'opération dans le tableau sous-jacent :

```
Tab[f("ceci")] = data1 ;
```

Cependant, il reste encore un problème à régler : l'ensemble des chaînes de caractères est infini et la taille du tableau n ... Donc plusieurs chaînes peuvent avoir la même valeur de hachage (ie. f(s1) = f(s2)). Elles "entrent donc en compétition" sur un même indice du tableau pour stocker leurs données associées ... On appelle cela une **collision**. Afin de gérer ces collisions, le tableau sous-jacent contient en fait des listes de couples (clé,pointeur sur une donnée). Donc `T["ceci"] <- data1` consiste au final à ajouter à la liste `Tab[f("ceci")]` le couple `("ceci", &data1)`.

Dans cet exercice, on va implémenter une table de hachage associant à des chaînes de caractères (clés de noms de villes) des coordonnées GPS. Récupérez le matériel d'exercice challenge qui vous fournit un squelette de code comportant :

- Les types de données
- La fonction de hachage et la fonction `h_to_ind` adaptant la valeur de hachage à la taille du tableau
- La fonction d'accès bas niveau (fonction `raw_acces`) que vous utiliserez pour résoudre le challenge

(i) Que font la fonction de hachage et la fonction `h_to_ind` ? Analyser le code.

(ii) Ecrire les fonctions suivantes :

(a) Fonction d'accès `get` :

```
bool get(char *s, HashTab t, Data *res)
```

renvoyant un booléen pour indiquer si la clé `s` est présente ou non, et stockant dans `res` un pointeur sur la donnée si la clé `s` est présente)

(b) Fonction d'affectation `set` :

```
bool set(char *s, HashTab t, Data *e)
```

réalisant l'affectation `t[s] <- e` (attention si la clé est déjà présente, on écrase la valeur de la "case" s, sinon on ajoute cette association "case" - donnée)

(iii) Puis testez votre code (et son efficacité) avec la fonction `test` fournie dans le main

Bonus. En déduire comment implémenter efficacement :
- Des ensembles (non ordonnés)
- Des matrices creuses
- Un GarbageCollector pour notre implémentation fonctionnelle des listes