TP2 - Tables de hachage

Ce qu'on voit lors de ce TP

- Les listes simplement chaînées (rappel)
- Comment implémenter une table de hachage (implémentation du type abstrait *"Ensemble"*)
- Comment on interface une structure de données en fonctions d'une autre
- L'utilité des ensembles dans certains problèmes algorithmiques

Commencez par télécharger les fichiers `liste_chainee.h` et `ensemble.h`. Déplacez-le dans un répertoire personnel, ainsi que vos deux fichiers `liste.c` et `liste.h`.

I) Listes simplements chaînées

Créez un fichier `liste_chainee.c` et codez toutes les fonctions dont le prototype se trouve dans `liste_chainee.h`.

C'est votre responsabilité maintenant de tester vos fonction au fur et à mesure ! Vous pouvez procéder comme la séance précédente : avec un fichier `main.c` où vous écrirez quelques tests unitaires et un `makefile`.

II) Ensembles

Faites de même avec `ensemble.h` : créez un fichier `ensemble.c` et codez toutes les fonctions dont le prototype se trouve dans le fichier entête.

III) Application : nombres amoureux

On va considérer une liste remplie d'entiers, par exemple : `[3,-1,8,4,18,0,9,16,-2]`. On dira qu'un nombre de la liste est *amoureux* si sa moitié est dans le tableau. Dans l'exemple du dessus, il y a 4 entiers amoureux :

- `-2` (il y a -1 dans le tableau)
- `0` (il y a 0 dans le tableau)
- `8` (il y a 4 dans le tableau)
- `16` (il y a 8 dans le tableau)

Par contre `9` n'est pas amoureux : 4 ne compte pas pour sa moitié.

Écrire une fonction qui étant donnée une liste, compte le nombre d'entiers amoureux dans cette liste.

Le but c'est de trouver un algorithme qui s'exécute en $O(n)$, où n est la taille de la liste.

Une fois que vous avez écrit votre programme, sauriez-vous justifier sa complexité ?

Je vous ai écrit un petit programme pour tester si votre fonction marche correctement :

```
#include <assert.h>
#include <time.h>
#include <limits.h>

/**
 * @brief Permet de vérifier le bon fonctionnement et la rapidité de
 * votre fonction qui compte le nombre d'entiers amoureux.
 * Une liste de la bonne taille est tirée aléatoirement.
 * @param votre_fonction la fonction que vous avez écrit
 * (elle doit prendre une Liste en entrée et un size_t) en sortie
 * @param taille la taille de la liste tirée aléatoirement pour le test.
 */
void test_nombre_amoureux(size_t (*votre_fonction) (Liste), size_t taille ){

    Liste l = liste_vide();

    struct timespec start, end;

    int base = -taille/2 + (taille/2 % 2) + 1;
    int x = base;
    size_t sol = 0;

    ajouter_en_fin(l,x);

    for(size_t i = 1; i < taille ; i++){

        if ( rand() % 2 == 0 || abs(x) > INT_MAX / 8 ) {
            base += 2*(1 + rand() % 3);
            x = base;
            if (rand() % 2 == 0 && abs(x) < INT_MAX / 8){
                x *= 2;
            }
        }
        else{
            x *= 2;
```

```
            sol++;
        }

        size_t j = rand() % (i+1);
        ajouter_en_fin(l,x);
        modifier(l,i,element(l,j));
        modifier(l,j,x);
    }
    clock_gettime(CLOCK_MONOTONIC, &start);

    // Je teste votre fonction ici :
    assert( votre_fonction(l) == sol );

    clock_gettime(CLOCK_MONOTONIC, &end);
    long int elapsed_time_ns = (end.tv_sec - start.tv_sec) * 1000000000 + (e
    printf("Temps d'exécution (pour une taille %ld): %f secondes \n", taille

    liberer_liste(l);
}
```

Il suffira juste d'écrire l'instruction `test_nombre_amoureux(votre_fonction,1000);` par exemple.

Jusqu'à quelle taille arrivez-vous à résoudre le problème ? Est-ce que votre fonction est plus rapide avec ou sans ensemble ?