Structures de données
TD4 Tables de hachage

1 Exercice 1 Empreintes digitales

Pour stocker les empreintes on utilise une association clé-valeur où la clé est l'empreinte et la valeur une structure contenant les informations de l'individu. La structure de données adaptée aux associations clé-valeur est la table de hachage.

On sait qu'il y a au plus 10^5 empreintes dans le fichier, on doit donc choisir une fonction de hachage retournant une valeur modulo 10^5.

On ne peut pas multiplier toutes les composantes du vecteur de l'empreinte entre elles car si une composante est nulle alors le résultat de la fonction sera nul lui aussi, toutes les empreintes contenant un 0 seraient donc stockées dans la case 0. Pour contouner ce problème, on peut ajouter 1 (ou autre mais au moins 1) à chaque composante et les multiplifier entre elles ensuite.

Des clés différentes peuvent avoir la même valeur de hachage, c'est ce qu'on appelle une collision.

2 Exercice 2 Gestion des collisions

Le résultat de la fonction de hachage $g(k) = k \mod 16$ est les 16 derniers bits de k correspondant donc à la dernière lettre. Exemple : $g(f(\text{"}LE\text{"})) = g(FF2E) = E = 14$

On peut résoudre les collisions de plusieurs façons dont :

- **par chaînage** : on utilise une liste chaînée par case
- **par adressage ouvert avec probing** Le probing consiste à tenter d'insérer à une position donnée par le résultat d'une fonction à deux paramètres : la clé et le numéro de tentative.
  - **probing linéaire** : $h(k, i) = g(k) + i$ on inssère à la première case disponible à parti de $g(k)$
  - **probing quadratique** $h(k, i) = g(k) + \frac{i}{2} + \frac{i^2}{2}$

  S'il n'y a plus de case libre alors on ne peut pas insérer.

	Chaînage	Probing linéaire	Probing quadratique
Case 0			13*
Case 1	5	5	5
Case 2			
Case 3	3	3	3
Case 4	7 → 8 → 13	7	7
Case 5	4 → 11 → 12	4	4
Case 6	9	8*	9
Case 7		9*	8*
Case 8	2 → 6 → 10	2	2
Case 9		6*	6*
Case 10		10*	12*
Case 11		11*	10*
Case 12		12*	
Case 13		13*	
Case 14	1	1	1
Case 15			11*

FIGURE 1 – Résolution des conflits avec les trois techniques. * indique qu'on a du faire plusieurs tentatives pour placer cette valeur

3 Exercice 3 Fonction de hachage

— f1 : le problème c'est qu'on n'utilise que le dernier chiffre de la clé
— f2 : aucun chiffre n'est pas pris compte (multiplication par 10 du coup le résultat sera toujours 0)
— f3 : on multiplie par 2 qui est un diviseur de la taille de la table donc on aura une case sur deux d'utilisée, dans nos exemples on utilise que la 4 et 6 case.
— f4 : c'est la bonne fonction car elle tient compte de tous les chiffres de la clé

4 Exercice 4 Table de hachage avec chaînage

```
typedef struct _hashTableEntry {
    int key;                      // la cle
    int value;                    // la valeur
    struct _hashTableEntry* next; // liste chainee des collisions
} HashTableEntry;

typedef struct {
    HashTableEntry** tab; // tableau
    int capacity;         // taille du tableau
    int size;             // nombre d'elements contenus
} HashTable;

HashTable* hashtable_create(int capacity)
{
    HashTable* new = malloc(...);
    new->tab       = malloc(...);
    new->capacity  = capacity;
    new->size      = 0;
    for(int i=0;i<capacity;i++) new-tab[i] = NULL;

    return new;
}

void hashtable_add(HashTable* ht, int k, int v)
{
    int hash = g(k);

    HashTableEntry* entry = malloc(...);
    entry->key   = k;
    entry->value = v;

    entry->next = ht[hash];
    ht[hash] = entry;
}
```

Si on souhaite connaître le nombre d'éléments contenus dans la table de hachage, il est plus interessant d'utiliser un compteur d'éléments que l'on place dans la structure (comme *size* dans la structure HashTable) et qui est mis à jour à chaque ajout et suppression plutôt que de calculer cette information lorsqu'elle est demandée.

La suppression d'un élément d'une liste chaînée est dans le pire cas en

O(n). Pour améliorer notre table de hachage on pourrait remplacer la liste chaînée par un arbre binaire de recherche qui permet de trouver et supprimer un élément en O(log(n)).