IFT339 - Exercices sur les tables de hachage et les graphes

Note: dans ces exercices, on suppose qu'une table de hachage peut faire insertion/recherche/suppression en temps $O(1)$, en ignorant l'aspect amorti et avec forte probabilité.

Exercice 1.
Soit T une table de hachage de capacité m. On a vu dans l'adressage ouvert que si on veut ajouter une clé c à une position $p = h(c)$, où ici h est notre fonction de hachage, on doit parcourir les alvéoles p, $p+1$, $p+2, \ldots$ jusqu'à ce qu'on trouve une alvéole vide. Un effet indésirable de cette idée est que les valeurs ont tendance à être regroupées en blocs alors qu'on voudrait plutôt disperser au maximum.

Une idée est de changer le pas du parcours. C'est-à-dire, on définit un k et si on veut insérer c à la position $p = h(c)$, on parcourt les alvéoles dans l'ordre p, $p+k$, $p+2k$, $p+3k, \ldots$ jusqu'à ce qu'on trouve une cellule vide.

a Insérez dans une table de hachage de taille 12 (donc avec des positions de 0 à 11) avec un pas de $k = 3$ les valeurs suivantes : 3, 18, 24, 42, 45. Prenez comme fonction de hachage $h(i) = i\%12$. Donnez l'état de la table après la dernière insertion. Quel est le problème? Notez que seules deux des valeurs insérées ont la même valeur après application de h.

b Nous avons le théorème suivant, qui nous provient de la théorie des groupes (une conséquence du théorème de Lagrange):

Théorème : Soit m un nombre premier. Alors pour tout entiers $p < m$ et $k < m$, les valeurs
p,
$(p+k)\%m$,
$(p+2k)\%m$,
$(p+3k)\%m$,
$\ldots$,
$(p+(m-1)k)\%m)$
sont toutes différentes.

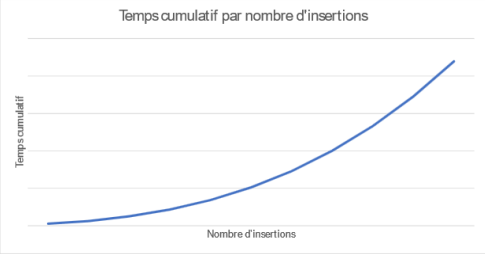
Utilisez le théorème pour justifier qu'il est préférable d'utiliser un nombre premier pour la capacité m de table de hachage en adressage ouvert.

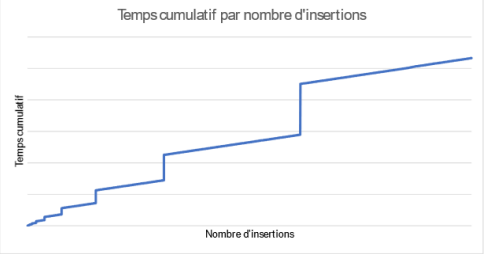
Exercice 2.
On dit souvent que la complexité des opérations d'une table de hachage est $O(1)$, mais ceci suppose que la fonction de hachage est calculable en temps $O(1)$. Ceci n'est pas toujours vrai.

Supposons que vous stockez des chaînes de caractères de longueur au plus m dans une table de hachage en adressage chaîné. Donnez les complexités pour l'insertion/recherche/suppression, dans le pire cas et dans le meilleur cas (qui est lorsque chaque alvéole a un nombre $O(\alpha)$ d'éléments).

Exercice 3.
Dans le célèbre problème de la double-somme (two-sum en anglais), on a en entrée un vecteur d'entier `vec` et un entier `t`. Il faut déterminer s'il existe deux indices `i` et `j` tels que `vec[i]` + `vec[j]` = `t`. Donnez un code en temps $O(n)$ pour ce problème (avec n la taille de `vec`).

Exercice 4.
Considérez les graphiques suivants, qui montrent le temps cumulatif pour effectuer n insertions dans un arbre binaire de recherche balancé, et dans une table de hachage. Dites lequel des graphiques correspond à quelle structure de données.





Exercice 5.
Vous avez une liste chaînée programmée par un stagiaire et vous suspectez que ses pointeurs peuvent être mal organisés. Il se peut qu'une liste chaîne ait un cycle, qui a la forme d'une séquence de noeuds $v1 \to v2 \to \ldots \to vk \to v1$. La notation $vi \to vj$ veut dire que $vi.next = vj$.

Montrez comment on peut détecter si une liste chaînée contient un cycle ou non en temps $O(n)$. Vous pouvez utiliser la classe suivante.

```
template <typename TYPE>
class list{
    struct Noeud{
        TYPE val;
        Noeud* next;
    }
    Noeud* debut;

    bool contient_cycle();
}
```

Exercice 6.
En bioinformatique. les séquences d'ADN sont représentées par des string sur alphabet $\{A, C, G, T\}$. Étant donné une séquence S, un k-mer est une sous-chaîne de S de longueur k (donc un segment consécutif de k caractères). Par exemple, si $S = ACTACTG$ et $k = 3$, l'ensemble des k-mers est $\{ACT, CTA, TAC, CTG\}$ (notez que l'on n'a pas listé ACT deux fois car on parle d'un ensemble ici).

a On veut stocker les k-mers d'une séquence S dans une table de hachage. Donnez d'abord une bonne fonction de hachage pour accomplir cette tâche. Cette fonction devrait être indépendante de k et de la longueur de S, et bien fonctionner peut importe la capacité de la table.

b Soit deux séquences S_1 et S_2. La distance de k-mers entre S_1 et S_2 consiste en le nombre de k-mers présents dans S_1 mais pas dans S_2, plus le nombre de k-mers présents dans S_2 mais pas dans S_1. Dites comment on peut calculer cette distance efficacement, puis donnez la complexité en fonction de la taille de S_1 et S_2, et de k.

Exercice 7.
Une façon alternative de stocker un graphe est d'entreposer toutes les arêtes dans une table de hachage. Donc on ne stocke jamais les sommets individuels – que les arêtes. On suppose que l'on a implémenté une fonction de hachage appropriée pour la structure `pair<int, int>`. Chaque paire représente une arête et on ne fait que stocker chaque arête dans un `unordered_set< pair <int, int> >`. Donnez la complexité des opérations sur les graphes.

- ajouter un sommet
- supprimer un sommet
- ajouter une arête
- supprimer une arête
- obtenir le nombre de voisins d'un sommet
- déterminer si une arête est présente

Exercice 8.
Considérez un graphe représenté par une liste d'adjacence, comme dans la classe ci-bas.

a Étant donné un indice i, donnez le code d'une fonction qui retourne l'ensemble de tous les sommets que i peut atteindre. On dit que i peut atteindre j s'il existe un chemin de i à j, c'est-à-dire, une liste de sommets $i - a_1 - a_2 - \ldots - a_k - j$ telle que tous les sommets qui se suivent dans la séquence partagent une arête.

Ne visez pas une complexité optimale, mais évitez un algorithme en temps exponentiel ou pire.

b Étant donné deux indices de sommets i et j, donnez le code d'une fonction qui d retourne un *chemin* de i à j, s'il en existe un.

```
class graphe{
    //adj[i] = voisins de i
    vector<int, vector<int> > adj;

    vector<int>
}
```