

Algorithmes et structures de données 2

Laboratoire n°4

Tables de hachage et correcteur orthographique

02.11.2020

Introduction

Ce laboratoire un peu particulier se déroulera par étapes et sera évalué comme note de contrôle continu. Il vous permettra d'étudier et de travailler avec les tables de hachage dans un premier temps, puis de vous persuader que l'utilisation de structures de données adaptées est d'une importance capitale pour résoudre un problème donné. Pour cela vous appliquerez ce que vous aurez vu dans les derniers chapitres du cours, à savoir les arbres de recherche.

Ce laboratoire est composé de deux étapes distinctes :

1. La première consiste à implémenter vos propres tables de hachage.
2. La deuxième où vous devrez implémenter un correcteur orthographique, en utilisant différentes structures de données.

Durée

- 12 périodes (jusqu'à la fin du semestre)
- A rendre le dimanche **24.01.2021** à **23h59** au plus tard

Environnement de travail

- Un template de base vous est fourni afin d'effectuer vos tests avec *catch2*. Vous êtes totalement libres pour la structure de votre projet. Vos classes créées doivent toutefois fournir les méthodes demandées et votre programme doit respecter la sortie attendue.
- Les fichiers suivants vous sont fournis pour vous inspirer lors de l'implémentation des Ternary Search Tries :
 - `BinarySearchTree.h` : Arbre binaire de recherche
 - `AVLTree.h` : Arbre binaire de recherche équilibré

Ces fichiers ne sont pas à rendre avec votre programme.

Première partie : Tables de hachage

Les tables de hachage sont disponibles sous plusieurs implémentations dans la librairie *stl* en C++, notamment à travers la structure `std::unordered_set`¹, qui a besoin d'une fonction de hachage pour fonctionner, mais qui a également besoin de maintenir un état interne cohérent pour stocker les différentes valeurs hachées et s'adapter aux besoins d'ajouter ou supprimer des valeurs.

Fonctionnement des tables de hachage

Les tables de hachages fonctionnent en interne en utilisant un tableau où l'on fait correspondre les indices aux différentes clés hachées. Il existe plusieurs solutions pour gérer les collisions entre différentes clés qui seraient *mappées* sur un même indice, nous avons vu en cours les techniques de résolution de collisions par *chaînage* et par *sondage linéaire*.

Fonctions de hachage

Les tables de hachage, en fonction des types qu'elles contiennent, ont besoin de fonctions de hachage pour fonctionner. Dans la librairie standard, il y a 2 possibilités de les définir : en les passant comme second argument template de `std::unordered_set` ou en spécialisant l'opérateur `()` de `std::hash()` pour le type que nous souhaitons mettre dans la table. Il est aussi nécessaire de définir une fonction d'égalité selon les mêmes principes ou encore en surchargeant l'opérateur `==`. La librairie STL fournit déjà les implémentations des spécialisations de `std::hash()` pour les types les plus courants.

A faire

Dans cette première partie nous vous demandons d'implémenter deux structures de table de hachage **sans doublon**, la première avec résolution des collisions par *chaînage*, et la deuxième par *sondage linéaire*. Vos tables de hachage doivent être génériques et donc pouvoir stocker n'importe quel type de données (pour autant que la fonction de hachage ait été définie pour ce type).

Ces deux structures doivent offrir au minimum les méthodes suivantes au client, dont leurs complexités doivent toutes être en $O(1)$ amorti :

- `size` Le nombre d'éléments dans la table de hachage
- `insert` Permet d'insérer une clé
- `contains` Vérifie si une clé est présente dans la table de hachage
- `erase` Supprime la clé de la table de hachage

Au niveau du redimensionnement de vos tables de hachage, nous vous demandons d'appliquer les règles suivantes :

- **Chaînage**
 - Doubler la taille du tableau quand $N / M \geq 8$
 - Réduire le tableau à la moitié quand $N / M \leq 2$

¹ Documentation : http://www.cplusplus.com/reference/unordered_set/unordered_set/

- **Sondage linéaire**

- Doubler la taille du tableau quand $N / M \geq \frac{1}{2}$
- Réduire le tableau à la moitié quand $N / M \leq \frac{1}{8}$

Où N est le nombre de clés dans le tableau et M la taille du tableau.

Nous vous demanderons aussi de tester vos implémentations à l'aide de la librairie [catch2](#) sous la forme de tests unitaires. Un projet d'exemple vous est fourni. Nous vous demandons au minimum de tester chaque méthode de chaque implémentation (maximisez la réutilisation de code) avec deux types que vous connaissez (comme `size_t` et les strings par exemple). Vous testerez également que le redimensionnement de vos tables de hachage corresponde bien au comportement attendu.

Pour votre implémentation, nous n'attendons aucune autre explication que votre implémentation de votre table de hachage. Les commentaires sont par contre bienvenus pour nous expliquer vos choix d'implémentations. Vous prêterez également une attention particulière à la redondance dans votre code.

Questions

Vous répondrez aux questions suivantes dans votre rapport :

- 1.1. Donnez et détaillez un exemple de cas d'utilisation concret où un `std::set` serait préférable à utiliser qu'un `std::unordered_set`, en comparant les complexités de chacune des deux structures pour ce cas d'utilisation.
- 1.2. Comment est-ce que les collisions sont gérées dans l'implémentation des tables de hachage de la STL (`std::unordered_set`) ? A quoi correspond le `max_load_factor` de l'implémentation de la STL, et quelle est sa valeur par défaut ?
- 1.3. L'implémentation des tables de hachage de la STL (`std::unordered_set`) ne double pas exactement la taille du tableau lorsque nécessaire, mais utilise des nombres premiers (la taille augmente par exemple de 2->5->11->23->47->97->etc...). Quel est l'avantage de cette approche ?

1.4. Soit la classe Person suivante :

```
class Person {
public:
    std::string name;
    std::string firstname;
    std::string gender;
    std::string birthday;

    bool operator ==(const Person &other) const {
        return  this->name      == other.name      &&
               this->firstname == other.firstname &&
               this->gender    == other.gender    &&
               this->birthday  == other.birthday;
    }
};
```

Et soit la fonction de hachage suivante :

```
namespace std
{
    template <>
    struct hash<Person>
    {
        size_t operator() (const Person& p) const
        {
            return hash<string>() (p.name);
        }
    };
}
```

Veuillez analyser les problèmes de l'implémentation de cette fonction de hachage. Vous proposerez également une nouvelle version améliorée de cette fonction de hachage, en discutant de vos démarches et réflexions.

Deuxième partie : Correcteur orthographique


Dans cette seconde partie vous devrez implémenter un correcteur orthographique en anglais permettant de trouver les fautes d'orthographe dans les mots composant un texte donné à l'aide d'un dictionnaire. Cette partie s'inspire d'un ancien cours de programmation Ada donné à l'EPFL².

Un mot est considéré comme correctement orthographié, s'il se trouve dans le dictionnaire de référence. Si un mot n'est pas dans le dictionnaire, votre programme devra proposer un ensemble de corrections possibles et les valider à l'aide du dictionnaire.

A faire

- Vous trouverez les structures et exemples fournis sur la page *CyberLearn* du cours.
- Votre programme devra lire 2 formats de fichiers différents :
 - Le **dictionnaire** : fichier texte, encodé en UTF-8 sans caractères accentués, comporte un mot par ligne.
 - Un **document texte** : fichier texte, encodé en UTF-8, comporte plusieurs lignes de plusieurs mots. Vous devrez être en mesure d'accéder aux mots un par un.

Dans les deux cas, vous convertirez toutes les lettres en minuscules et supprimerez tous les caractères qui ne sont pas une lettre (a-z) ou une apostrophe (') en milieu de mot.

1. Votre solution devra comporter au minimum trois implémentations différentes et il doit être aisé de passer de l'une à l'autre (en les exécutant à la suite ou en changeant une variable par exemple) :
 - Une solution utilisant vos deux tables de hachage développées dans la partie 1.
 - Une solution où vous testerez au minimum deux structures de la *STL*. Vous ferez particulièrement attention à la complexité des structures choisies. (Vous testerez séparément chaque structure). 
 - Une solution utilisant un Ternary Search Trie (TST), que vous devez implémenter vous-même librement. De la documentation est disponible dans le dossier du laboratoire sur *CyberLearn*. Pour des raisons de performances, nous vous demanderons d'équilibrer votre *Ternary Search Trie*. Votre TST doit fournir la même API que vos tables de hachage (méthodes `size`, `insert`, `contains`, `erase`). Nous vous demandons également de tester votre implémentation avec `catch2`.
2. Vous afficherez dans la console le temps chargement du dictionnaire ainsi que le temps de correction du texte.
3. Si le temps de chargement du dictionnaire en mémoire est supérieur à 1 minute (prétraitement compris) alors nous vous suggérons d'utiliser une version prétraitée du dictionnaire. Dans ce cas, vous devrez également nous remettre la version prétraitée du dictionnaire.
4. Nous vous demandons de comparer et analyser le temps de correction des différentes structures pour au minimum le fichier texte le plus long (`input_sh.txt`) dans votre rapport. Vous comparerez les différentes structures entre elles en vous appuyant sur leurs complexités théoriques respectives. N'hésitez pas à vous appuyer sur des tableaux/graphiques pour vos comparaisons.

² Jörg Kienzle, *Programming course: spellchecker project*, EPFL, 2002

Détection d'erreurs

Si un mot du texte n'est pas présent dans le dictionnaire, il sera considéré comme mal orthographié. Le logiciel générera 4 ensembles de propositions de corrections sur la base des 4 hypothèses suivantes :

1. L'utilisateur a tapé une lettre supplémentaire : **a**queux → aqueux (il y a un c en trop) ;
2. L'utilisateur a oublié de taper une lettre : a**q**eux → aque**u** (il manque la lettre u) ;
3. L'utilisateur a mal tapé une lettre : a**w**ueux → aq**u**eux (il y a un w à la place du q) ;
4. L'utilisateur a échangé 2 lettres consécutives : a**uq**eux → aq**ue**ux (u et q intervertis).

Ces propositions seront vérifiées avec le dictionnaire et seules celles qui s'y trouvent seront proposées à l'utilisateur.

Pour chaque document pour lequel vous vérifierez l'orthographe, vous générerez un **fichier texte** comportant les mots mal orthographiés (préfixés d'une étoile) suivi immédiatement des propositions de correction vérifiées (préfixées du numéro de l'hypothèse). Les mots doivent rester dans l'ordre dans lequel ils apparaissent dans le texte, en cas de répétition d'un mot les propositions de correction devront à nouveau être présentées. Voir un exemple de la sortie attendue sur la Figure 1.

```
*lates
1:late
2:plates
2:latest
3:fates
3:gates
4:altes
...
*motsuivant
...
```

Exemple de fichier de sortie

Nous utiliserons un script pour vérifier vos résultats, vous serez donc pénalisés si vous ne respectez pas le format demandé.

Questions

- 2.1. Quels sont les avantages et les inconvénients de la structure Ternary Search Trie par rapport à une structure de table de hachage ?
- 2.2. Quels sont les avantages et les inconvénients des fonctions de hachage cryptographiques ? L'utilisation d'une fonction de hachage cryptographique est-elle adaptée pour notre correcteur orthographique ?
- 2.3. Pourquoi l'utilisation d'un tableau (`std::vector`) n'est pas recommandée comme structure pour notre correcteur orthographique ? Si toutefois l'utilisation d'un tableau était forcée, quel traitement pourrions-nous appliquer sur notre tableau (notre dictionnaire) afin d'améliorer par la suite la complexité de corrections de textes ? Quelles en seraient les conséquences au niveau de la complexité :
 - du chargement + traitement du dictionnaire
 - de la correction de textes

Rendu / Évaluation

Ce laboratoire est relativement conséquent, son évaluation sera considérée comme une note de contrôle continu.

En plus de vos implémentations des deux parties nous vous demanderons un rapport pour répondre aux différentes questions et choix d'implémentations, ainsi que l'analyse du temps de correction entre les différentes structures demandées (partie 2, point4). Votre rapport contiendra également un titre, vos noms, une petite introduction et conclusion.

Vous devrez également apporter une attention particulière aux commentaires dans votre code qui devront nous permettre d'identifier les différentes étapes principales de vos implémentations, choix que vous aurez effectués et les justifications correspondantes.

Bonne chance !