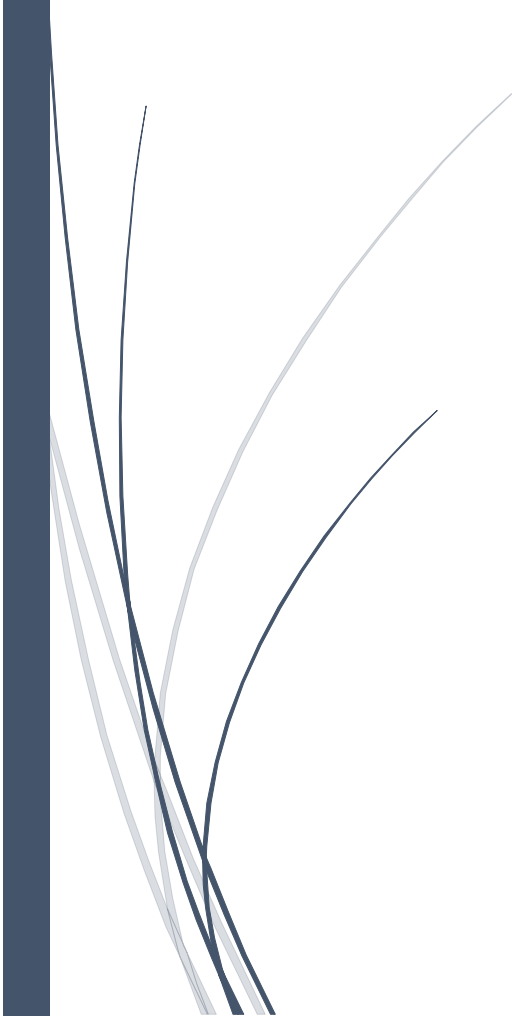
24/01/2021

Labo4

Table de hachage et correcteur orthographique



Pellissier David, Ruckstuhl Michael, Sauge Ryan
HEIG-VD

Table des matières

1 Introduction

Il était une fois un village nommé ASD où vivait 5 grandes familles : Les Linear probing, Les Separate Chaining , les TST, les AVL et la très vaste famille des conteneurs STL dont faisait notamment partie les célèbres Vector et Unordered_set. Ces différentes familles vivaient en paix et en harmonie, mais un jour, un voyageur venu des contrées lointaines de l'HEIG demanda à savoir quelle était la plus valeureuse, la plus brave, la plus rapide et surtout la plus forte des familles. Les grandes familles ne s'étaient jamais posé la question et la belle entente fut rompue. Pour éviter un bain de sang, 3 juges furent appelés en renfort. Ce rapport est le compte-rendu de leur aventure.

- La première partie se concentre sur le mode de fonctionnement et l'implémentation des familles Linear Probing et Separate Chaining, ainsi que de la concurrence régnante entre les différents membres des Conteneurs à travers la rivalité entre Set et Unordred_set.
- La deuxième partie se concentre sur le tournoi où pour se départager les différentes familles s'affrontèrent lors de la terrible épreuve du Correcteur Orthographique. Chaque famille était testée sur leur rapidité et sur la justesse de leur résultat à trouver des fautes d'orthographes dans un texte et à présenter des corrections.

2 Partie 1 – Table de hachages

2.1 Choix d'implémentation

2.1.1 Tests effectués

Nous avons testé HashMapLinearProbing et HashMapSeparateChaining avec des entiers et des strings. Les test sont identiques et communs entre les 2 structures.

1. Ces tests ci-dessous sont génériques(int, String, double, etc.). Pou utiliser ces tests, l'utilisateur doit passer en argument les vecteurs avec les valeurs nécessaire au déroulement des tests. Dans nos tests, nous avons utilisés des size_t et des string.
   - Test que la hashmap est vide
   - Contains() et erase() sur des clés qui ne sont pas dans la hashmap
   - Insertion d'une clé et vérification que la clé se trouve dans la hashmap
   - Suppression d'une clé
   - Redimensionnement des tables de hachage
   - Insère des éléments de keys, supprime les clés spécifiées par nomes, compte et vérifie le résultat.
2. Ces tests sont disponibles qu'avec des valeurs numériques(int, double, etc) car les valeurs utilisées sont générées dans les fonctions de tests, et non passé en arguments. Ils permettent de tester avec un plus grand nombre de valeurs.
   - Insertion d'une valeur sur deux
   - Suppression
   - Suppression d'une valeur sur deux

L'insertion et la suppression sont également testées dans les tests du point 1 mais avec de plus petites valeurs car les fonctions de la partie 1 était initialement destiné aux strings.

2.2 Question 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.

Il est préférable d'utiliser un set :

- Si on souhaite afficher de manière triée, par exemple par ordre alphabétique, tous les éléments contenus dans la structure. Les éléments étant déjà ordonnés dans un set, aucune modification du contenu n'est nécessaire, on a alors une complexité en O(N) qui correspond au parcours du conteneur en entier.
- Toute opération nécessitant un tri sera plus efficace dans un set, par exemple, si on souhaite l'élément qui suit/précède (par rapport à un tri) un élément donné. On peut également utiliser les fonctions lower_bound() et upper_bound() sur un set.

<u>Remarque :</u>

Nous n'avons pas comparé les complexités entre unorderd_set et set parce que qu'un unordered_set n'est pas fait pour réaliser des opérations sur une structure nécessitant d'être triées.

2.3 Question 1.2.

A) Comment est-ce que les collisions sont gérées dans l'implémentation des tables de hachage de la STL (std::unordered_set) ?

Unordered_set gère les collisions par chaînage. Les éléments sont organisés en compartiment(bucket) en fonction de leur valeur de hachage. Au sein d'un même compartiment, il n'y aura que des éléments ayant produit le même hash.

Sources : http://www.cplusplus.com/reference/unordered_set/unordered_set/

B) A quoi correspond le max_load_factor de l'implémentation de la STL, et quelle est sa valeur par défaut ?

Le load_factor est le ratio entre le nombre d'éléments(size) dans le container et le nombre de compartiment(bucket_count)

La valeur par défaut du max_load_factor est de 1.0:

Source : http://www.cplusplus.com/reference/unordered_set/unordered_set/max_load_factor/

2.4 Question 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 ?

Un nombre premier par sa définition n'a que 2 diviseurs: 1 et lui-même.

En prenant un nombre premier, on réduit fortement le risque de collision suivant le calcul utilisé lors de la fonction de hachage.

Par exemple, pour la fonction h(K) = (ak + b) modulo m.

Le nombre de collision est réduite si a et m ne partage pas de facteur commun, ce qui a de très fortes chances d'arriver vu que m est un nombre premier.

2.4.1 Démonstration

Pour vérifier ça, nous avons rempli un tableau de 1000 éléments de 0 à 1000 puis nous avons ensuite haché chaque élément dans une fonction de hachage de type h(k) = (ak + b) mod m avec une valeur de a = 4 et b = 7.

Nous avons testé pour différentes valeurs de m avec des nombres premiers et non premiers.

Premier : {13, 17, 19, 29}

Non premier : {12, 16, 25, 33}

2.4.1.1 Fonctions

```
int hashTest(int k, int m){
    int a = 4;
    int b = 7;
    return (a * k + b) % m;
}
```

2.4.1.2 Remplissage Tableau

Code utilisé pour remplir le tableau des éléments

```
const int TAILLE = 1000;
int donnee[TAILLE];
std::iota (donnee, donnee + TAILLE, 0);
```

2.4.1.3 Programme de test

- **Nombre Premier**

```
cout << "Premier" << endl;
 const int PREMIER_TEST[] = {13, 17, 19, 29};
 cout << "Avec 13, 17, 19, 29 " << endl;
 for(int m : PREMIER_TEST){
     cout << "Pour m = " << m << endl;
     set<int> conteneur;
     for (int& k : donnee){

        conteneur.insert(hashTest(k, m));
     }
     cout <<"Emplacements utilises : ";
     for(int k : conteneur){
         cout << k << " ";
     }
     cout << endl;
     if(conteneur.size() == m){
         cout << "Tous les emplacements sont utilises" << endl;
     }
     cout << endl;
 }
```

- **Nombre Non premier**

```
const int NON_PREMIER_TEST[] = {12, 16, 25, 33};
 cout << endl;
 cout << "Avec 12, 16, 25, 33" << endl;
 for(int m : NON_PREMIER_TEST){
     cout << "Pour m = " << m << endl;
     set<int> conteneur;
     for (int& k : donnee){
         conteneur.insert(hashTest(k, m));
     }
     cout <<"Emplacements utilises : ";
     for(int k : conteneur){
         cout << k << " ";
     }
     cout << endl;
     if(conteneur.size() == m){
         cout << "Tous les emplacements sont utilises" << endl;
     }else{
         cout << "Sur " << m << " emplacements disponibles, seuls " <<
conteneur.size() << " sont utilises" << endl;
         cout << "Il y a " << m - conteneur.size() << " emplacements non
utilises" << endl;
     }
```

```
    cout << endl;
}
```

2.4.1.4 Résultat

- Nombre premier

```
Premier
Avec 13, 17, 19, 29
Pour m = 13
Emplacements utilises : 0 1 2 3 4 5 6 7 8 9 10 11 12
Tous les emplacements sont utilises

Pour m = 17
Emplacements utilises : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
Tous les emplacements sont utilises

Pour m = 19
Emplacements utilises : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
Tous les emplacements sont utilises

Pour m = 29
Emplacements utilises : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
Tous les emplacements sont utilises
```

- Nombre non premier

```
Avec 12, 16, 25, 33
Pour m = 12
Emplacements utilises : 3 7 11
Sur 12 emplacements disponibles, seuls 3 sont utilises
Il y a 9 emplacements non utilises

Pour m = 16
Emplacements utilises : 3 7 11 15
Sur 16 emplacements disponibles, seuls 4 sont utilises
Il y a 12 emplacements non utilises

Pour m = 25
Emplacements utilises : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
Tous les emplacements sont utilises

Pour m = 33
Emplacements utilises : 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Tous les emplacements sont utilises
```

2.4.1.5 Conclusion

Avec les 4 nombres premiers testés, l'ensemble des emplacements possibles sont obtenus

Avec les 4 nombres non premiers, on peut remarquer qu'avec 12 et 16 qui sont des multiples de a (a=4) on obtient peu d'emplacements possibles.

Avec 33 et 25, on a bien tous les emplacements possibles qui sont utilisés car ils n'ont pas de facteur commun avec a.

2.5 1.4 - Analyse de la classe Person et sa fonction de hachage

- 3 attributs caractérisant une personne ne sont pas pris en compte dans le calcul du hash(firstname, gender et birthday), ce qui fait beaucoup.
- p.name provoque beaucoup de collisions car les personnes de la même famille ont toutes le même nom. On pourrait utiliser à la place :

```
namespace std {
    template<>
    struct hash<Person> {
        std::size_t operator()(const Person& p) {
            using std::size_t;
            using std::hash;
            using std::string;
            size_t hashval = 17;
            hashval = 31*hashval + hash<string>()(p.name);
            hashval = 31*hashval + hash<string>()(p.firstname);
            hashval = 31*hashval + hash<string>()(p.gender);
            hashval = 31*hashval + hash<string>()(p.birthday);
            return hashval;
        }
    };
}
```

Pour des raisons d'optimisations, plutôt que d'utiliser p.name, p.firstname, p.gender et p.birthday pour calculer le hash, on pourrait utiliser un identifiant numérique, qui serait unique pour chaque personne.

3 Partie 2 : Correcteur orthographique

3.1 Choix d'implémentation

3.1.1 DictionaryContainer

Ce wrapper sert à définir une interface sur une structure de données qui puisse être utilisée par le dictionnaire. Il est donc possible d'utiliser n'importe quelle structure en dans un dictionnaire sans devoir redéfinir la classe Dictionary.

La classe définit 3 fonctions publiques qui doivent lui être passées par lambda à sa création : contains(), insert() et erase().

La structure de données ne peut pas être stockée à l'intérieur de la classe, car dans les lambdas il n'est pas possible d'accéder aux attributs du Container qui est en train d'être créé.

Exemple de création d'un Container:

```
unordered_set<string> test;
Container<string> cw (
    [&test](const string& KEY){ return test.find(KEY) != test.end();},
    [&test](const string& KEY){ test.insert(KEY);},
    [&test](const string& KEY){ test.erase(KEY);}       );
Dictionary<Container<string>> dico(cw, DICTIONARY_FILE);
```

3.1.2 Dictionary

Un dictionnaire a pour but de contenir une base de mots corrects afin d'être utilisé pour la correction orthographique.

On doit lui passer en paramètre un Container et un nom de fichier facultatif si on veut que le dictionnaire lise et remplisse son Container.

Avant d'ajouter ou de tester un mot, le mot est passé dans une fonction "sanitizeWord" qui enlève les caractères non alpha, les apostrophes aux extrémités et met les caractères restants en minuscule.

3.1.3 SpellChecker

Cette classe vérifie si chaque mot dans un fichier donné existe dans le dictionnaire. Si le mot n'existe pas, il va stocker des suggestions en créant un nouvel objet de classe Suggestions, qui ira chercher dans le dictionnaire des propositions comme décrites dans la donnée de ce labo.

3.1.4 TernarySearchTree

Dans le ternary search tree, les fonctions insert, size, contains et erase ont été créée comme demandé dans la consigne. L'implémentation de celles-ci a été grandement inspirée des diapositives 52Tries.key de Robert Sedgewick et Kevin Wayne. Pour l'équilibrage la majorité des fonctions écrites en C++ sur les diapositives du cours d'ASD2 sur les AVL ont pu être reprises sans changement.

Une fonction isBalanced a été rajoutée en public afin de pouvoir faire des tests sur chaque nœud pour savoir si l'arbre et chaque sous arbres sont équilibrés. La fonction insert demande une valeur. Ce n'est

pas nécessaire dans ce laboratoire, mais ceci est utile si une fonction get doit être implémentée par la suite (cf. 52Tries.key).

Les tests catch2 de cette class se trouvent dans TST_Test.cpp.

3.2 Question 2.1

Quels sont les avantages et les inconvénients de la structure Ternary Search Trie par rapport à une structure de table de hachage ?

	Table de hachage	Ternary Search Tree(TST)	Conclusion
Généricité	Générique(Strings, int, Objet)	Fonctionne uniquement sur les strings	Désavantage TST
Rechercher, insérer, supprimer une clé	-Doit examiner l'entièreté de la clé -"Complexité linéaire" L'insertion ainsi que la recherche des succès ou des échecs ont le même coût, c'est-à-dire linéaire	Permet d'examiner uniquement les caractères nécessaires de la clé. La recherche des "échecs" ne prend que quelques caractères	Avantage TST
Opérations sur les tables de symbole ordrés	Pas supportée	Supportée	Avantage TST

Divers :

Pour les tables de hachage, la performance dépend de la fonction de hash, ce qui peut être un avantage ou un désavantage suivant la fonction utilisée.

3.3 Question 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 ?

Non, car la TST est bien plus efficace qu'une fonction de hachage cryptographique (voir question 2.1)

De plus, les fonctions de hachage cryptographique sont onéreuses en temps de calcul.

L'avantage de ces fonctions c'est qu'il s'agit de fonction unique avec très peu de collisions, voire pas du tout (si on prend des bonnes fonctions de hachage). Elles sont particulièrement bien adaptées pour hacher le mot de passe des utilisateurs.

3.4 Question 2.3

A) Pourquoi l'utilisation d'un tableau (std::vector) n'est pas recommandée comme structure pour notre correcteur orthographique ?

- Il y a trop d'opérations inutiles (parcours de tout le tableau dans le pire cas, caractères stockés autant de fois qu'ils apparaissent dans des mots)
- Il est difficile de détecter les erreurs et proposer des mots semblables.

B) 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 ?

Nous pouvons trier le tableau avec la fonction std::sort après avoir inséré les données du dictionnaire avec push_back, ce qui nous laisse utiliser un binary_search pour chercher rapidement un élément.

C) Quelles en seraient les conséquences au niveau de la complexité ?

Résultat obtenu :

```
Dictionary generated in 954ms
Spellcheck done in 640ms
Total time : 1594ms
```

- du chargement + traitement du dictionnaire

Insertion des clés dans le vecteur : fonction push_back, complexité en O(1) par conséquent pour tout insérer, on a une complexité en O(N)

Une fois toutes les clés insérées dans le dictionnaire, on effectue un tri avec sort(), complexité : $N*\log_2(N)$

La complexité totale est de : O(N) + O(N*log2(N)) = O(N *Log2(N))

- de la correction de textes

Pour rechercher un mot, nous utilisons la fonction binary_search ayant une complexité en O(log(n))

4 Comparaison des temps de traitement de texte

4.1 Complexités théoriques

	Linear Probing	Separate Chaining	Unordered Set	Sorted Vector*	TST
insert()	O(1)	O(1)	O(1) (log(n) dans le pire des cas)	O(log(n)*n) **ou** O(1)	O(log(n))
contains()	O(1)	O(1)	O(1)	O(log(n))	O(log(n))
realloc	O(n)	O(n)	O(n)	O(n)	-
balance	-	-	-	-	O(1) car on stocke la hauteur d'un noeud

* Pour l'insertion de Sorted Vector, on utilise un BinarySearch pour trouver l'endroit où insérer, c'est pourquoi on a log(n) *n.
Néanmoins, dans nos tests on remplit le dictionnaire en utilisant push_back et en triant le tout à la fin, ce qui fait qu'on peut avoir également O(1) pour l'insertion.

4.2 Valeurs observées [milisecondes]

Il s'agit d'une moyenne effectuée sur 5 tests.

	Linear Probing	Separate Chaining	Unordered Set	Sorted Vector	TST
Chargement dictionnaire	909	1586	854	811	1123
Correction texte	325	570	424	586	434

Testé sur le fichier "input_sh.txt".

Classement :

Création du dictionnaire: Sorted Vector < Unordered Set < Linear Probing < TST < Separate Chaining

Correction du texte: Linear Probing < Unordered Set < TST < Separate Chaining < Sorted Vector

4.3 Commentaires

Au niveau du chargement du dictionnaire, on peut voir que le Sorted Vector est le plus rapide de tous. Cela est dû au fait que l'on insère d'abord en O(1) les valeurs puis on trie le tout en $N*\log_2(N)$. La réallocation du vecteur est en O(n) et ne demande pas de calculer de hash, ce qui explique la différence avec les autres structures utilisant le hachage. Le TST est plus lent car l'insertion est en log(N).

Les résultats du Separate Chaining est surprenant, mais il est dû en partie au parcours des buckets lorsqu'on essaie de chercher et d'insérer des valeurs dans la hashmap. Cependant unordred_set est basé aussi sur un chainage mais lui possède de meilleurs résultats. Cette différence pourrait notamment provenir que Unordred_set ne double pas la taille des compartiments, mais utilise des nombres premiers.

5 Conclusion

Le tournoi étant fini (voir introduction), il est temps aux juges de rendre leur verdict :

- L'utilisation des wrappers a permis de mettre en commun des tests et comportements identiques, par exemple entre le sondage linéaire et le chaînage.
- La partie 1 nous a permis de voir la différence entre les 2 façons de régler les collisions :
    - Le sondage linéaire présente une structure plus simple avec un seul conteneur, néanmoins nécessite d'utiliser des pointeurs pour stocker les éléments afin de déterminer si une case est libre ou occupée
    - Le chainage présente quant à lui une structure plus importante car à 2 "niveaux" avec un conteneur contenant différents compartiments(buckets)
- La partie 2, le tournoi, a vu s'affronté différentes familles avec une concurrence très rude.
    - La famille Separate Chaining se retrouve bonne dernière du tournoi.
    - Pour l'épreuve de la génération du dictionnaire, on remarque des résultats proches pour le trio de tête Unorderd_set, Linear et Vector. Le TST prend lui nettement plus de temps(insertion en log2(n)).
    - Pour l'épreuve du Correcteur Orthographique, on retrouve Linear Probing en tête suivi de Unordered_set. Le podium est complété cette fois par le TST. Vector qui avait fait des bons résultats lors de la génération du dictionnaire ne tient pas la cadence et recule au classement.