CS228 - TP4 (mini TP) Complexités expérimentales.

Grenoble INP – Esisar – année 2023-24

Version du 5 août 2024

Objectif pédagogique

- Pratiquer l'algorithmique et la programmation C.
- Pratiquer/réviser les fiches de syntaxe C "niveau 1 et 2".
- Tracer des courbes de complexité expérimentale.
- Utiliser les hashmap de la glib pour stocker un dictionnaire.

Ni le code ni les réponses ne sont évaluées, l'objectif est de **pratiquer**.

** Partie 1 ou 2 au choix ** (TP court)

Au TP2 nous avons utilisé une bibliothèque dont la documentation est disponible à cette adresse: https://docs.gtk.org/glib/ Dans ce TP il est demandé d'utiliser aussi cette bibliothèque.

Partie 1 : utilisation de la bibliothèque `glib` pour les arbres.

(fichiers fournis)

- De la même façon qu'au TP 2, déclarer et utiliser des arbres (qui sont des arbres binaires équilibrés).

La doc de la fonction insert comporte " The cost of maintaining a balanced tree while inserting new key/value result in a O(n log(n)) operation where most of the other operations are O(log(n))."

L'objectif de cette section est d'expérimentalement vérifier que la documentation ne ment pas.

- Proposer une méthodologie pour réaliser une telle étude.
- Réaliser l'étude au moins pour la recherche.
- On fournit un fichier de configuration `gnuplot`

Partie 2 : tables de hachage avec la `glib`

Une table de hachage est un compromis entre les listes chaînées efficaces sur les ajouts et les tableaux (ou listes contiguës) efficaces sur les accès. Soit un ensemble fini $\mathcal{D}$ de données (ici les mots du dictionnaire) que l'on va stocker dans un tableau `ht`. Pour ranger un élement `x` de $\mathcal{D}$, on calcule son image par une fonction de hachage `hash`, qui donne son indice (appelé "indice de hachage") dans `ht`}, `x` sera donc rangé dans `ht[hash(x)]`. Il reste à trouver une fonction `hash` adéquate:

- l'idéal est de trouver une fonction bijective entre $\mathcal{D}$ et l'intervalle d'indices du tableau, ainsi chaque case du tableau contient 1 élément de $\mathcal{D}$. Mais il est difficile de trouver une telle fonction bijective, ne serait-ce que parce que $\mathcal{D}$ est rarement connu a priori.
- l'utilisation d'une fonction injective (ie qui associe à chaque `x` de $\mathcal{D}$ une case différente de $\mathcal{D}$) mène à un tableau de taille $sup\{hash(x),\ x \in \mathcal{D}\}$, c'est-à-dire à un tableau à trous, ce qui peut générer une perte importante de place.

- le principe consiste alors à prendre une fonction surjective, obtenue généralement par modulo sur une taille limitée de tableau, soit `TABLE_SIZE` $\ll card(\mathcal{D})$. On tombe alors sur un problème de “collisions” parce que plusieurs données peuvent avoir le même indice de hachage. On choisit donc une structure mixte formée d'un tableau statique de listes chaînées contenant les éléments de même indice de hachage (appelées “listes de collisions”). Si possible ces listes sont ordonnées pour optimiser les accès. L'ajout d'un élément `x` revient alors à calculer `hash(x)` de coût quasi constant et insérer `x` dans la liste `ht[hash[x]]`, en $o(n)$ n étant la taille de la liste correspondante. Toute la difficulté est de trouver un bon compromis entre la taille du tableau et la taille des listes. Remarquez que prendre TABLE_SIZE=1 revient à faire une simple liste.

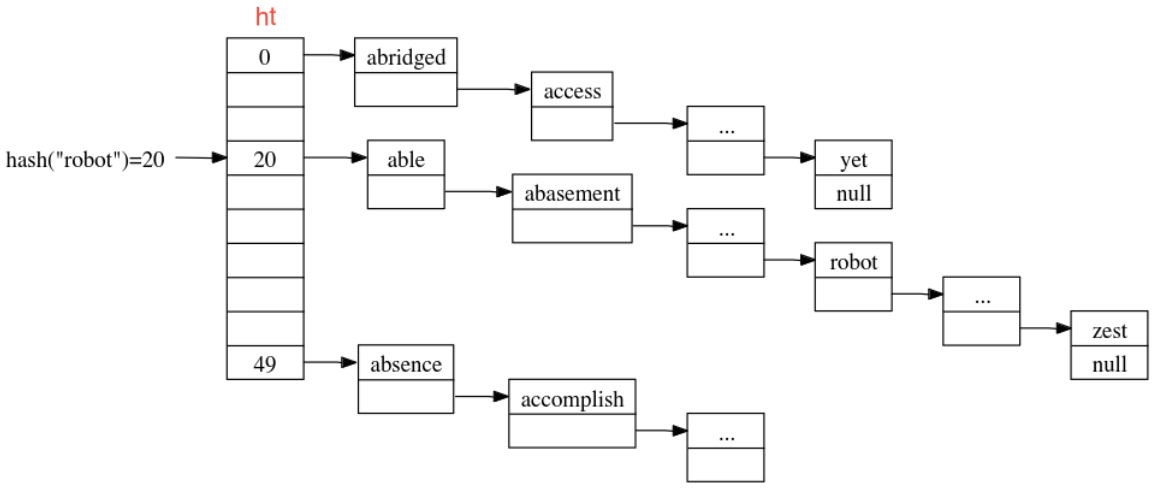

Figure 1: Table `ht` du dictionnaire anglais

Figure 1: Table de hachage schématique avec des mots

à faire:

- déclarer et remplir une hashmap avec la listes des noms français sans accents (cf ce lien).
- chercher si le cout d'une insertion est noté dans la documentation
- valider ce cout expérimentalement.