Université de Reims Champagne-Ardenne
Département d'Informatique

Info0501 – 2022-2023
Pierre Delisle

Travaux pratiques 6

Analyse de texte

L'objectif de ce TP est d'implémenter un programme qui compte le nombre de mots différents dans un texte à l'aide de différentes structures de données (liste chaînée, table de hachage et arbre binaire de recherche) et de comparer leur efficacité. Pour ce faire, vous suivrez la procédure suivante :

1. Lire un fichier texte et, pour chaque mot qu'il contient,
    a. Rechercher si le mot est déjà présent dans la structure de données ;
    b. Si le mot n'est pas déjà présent, insérer le mot dans la structure de données ;
2. Calculer et afficher :
    a. Le nombre de mots présents dans la structure de données ;
    b. Le temps d'exécution de toute la procédure, en secondes.

Afin de simplifier le problème de lire les mots dans le fichier, considérez qu'un mot est une chaîne de caractères précédée et suivie d'un blanc (espace ou fin de ligne). Autrement dit, vous pouvez supposer que "Quebec" et "Quebec." sont deux mots différents.

Aussi, il est à noter que pour éviter les problèmes de conversion de caractères en C, les caractères accentués ont été remplacés (é → e, à → a, etc.) et certains caractères spéciaux (°, £, etc.) ont été éliminés. De plus, pour éviter des problèmes de débordement plus loin, ne considérez que les mots de 26 caractères et moins.

Dans un premier temps, effectuez vos tests avec le fichier **texte1.txt** présent sur la page Moodle du cours. Lorsque votre implémentation est validée pour ce fichier, testez votre programme en utilisant les fichiers **texte2.txt**, **dico_france-quebec.txt**, **histoire_quebec.txt**, **dico_ae.txt** et **dico.txt**.

Partie 1 – Liste chaînée

1. Adaptez vos implémentations de **cellule** et de **liste** du TP 1 de sorte à stocker des mots (chaîne de caractères) plutôt que des sommets de graphe.
2. Créez un programme de test permettant de vérifier le fonctionnement de votre implémentation en demandant à l'utilisateur de saisir des mots, en les insérant successivement dans une liste et en affichant la liste ainsi créée. Essayez aussi de supprimer un mot, de rechercher un mot et de détruire la liste.
3. Ouvrez le fichier **texte1.txt** et insérez chaque mot différent qui y figure dans votre liste chaînée. Une fois tout le fichier traité, comptez le nombre d'éléments présents dans la liste en définissant la fonction **compter_liste** et vérifiez la conformité du résultat.
4. Testez votre programme avec les autres fichiers. Affichez, pour chaque fichier, le nombre total de mots présents dans le fichier, le nombre de mots différents et le temps d'exécution.

Partie 2 – Table de hachage

Nous allons expérimenter l'utilisation d'une table de hachage avec gestion des collisions par chaînage. Nous aurons donc besoin d'un tableau de listes chaînées.

1. Créez les fichiers **table_hachage.h** et **table_hachage.c** et implémentez-y une structure adéquate pour une table de hachage ainsi que les fonctions suivantes :
    - **initialiser_table_hachage** : initialise une table de hachage de dimension spécifique ;
    - **detruire_table_hachage** : libère les ressources mémoire d'une table de hachage ;
    - **afficher_table_hachage** : affiche la table de hachage.

Nous devons maintenant implémenter l'insertion de mots dans notre table de hachage. Pour ce faire, nous devons être en mesure de convertir une chaîne de caractères en un entier *k* en s'assurant que des chaînes différentes soient associées à des entiers différents. Pour ce faire, on peut interpréter une chaîne comme un entier exprimé dans une base adaptée :

- on interprète la chaîne de caractères comme un ensemble d'entiers correspondant au code ASCII de chaque caractère (A = 65, B = 66, Z = 90, a = 97, b = 98, 0 = 48, 1 = 49, …) en respectant leur ordre.
  Par exemple, "Abc" → {65, 98, 99} ;
- partant du principe que le codage ASCII de base utilise 128 caractères, on exprime la chaîne en base 128 en additionnant les valeurs de tous les entiers exprimés dans cette base. Ainsi, le dernier entier est multiplié par 128^0, l'avant-dernier par 128^1, et ainsi de suite, puis toutes les valeurs obtenues sont finalement additionnées.
  Par exemple, "Abc" → $65 * 128^2 + 98 * 128^1 + 99 * 128^0 = 1\,064\,960 + 12\,544 + 99 = 1\,077\,603$.

2. Écrivez une fonction **convertir_ch_entier** qui prend une chaîne de caractères et retourne l'entier *k* associé tel que défini ci-haut.
   Note : cette procédure engendrera des débordements. Nous considérons que ces débordements font partie de la fonction de hachage alors pour éviter les nombres négatifs et permettre de grandes valeurs pour *k*, utilisez un entier très long non signé (**unsigned long long**).

Maintenant que nous sommes en mesure de convertir une chaîne de caractères en entier, nous pouvons l'insérer dans notre table de hachage en utilisant une fonction de hachage appropriée.

3. Écrivez une fonction **hachage** qui détermine un indice $h(k)$ à partir d'un *k* donné par la méthode de la division.
4. Complétez l'implémentation de votre table de hachage en définissant les fonctions suivantes :
    - **insérer_hachage** : prend en paramètre un pointeur sur une cellule contenant une chaîne de caractères et l'insère à l'endroit approprié dans une table de hachage ;
    - **rechercher_hachage** : recherche une chaîne de caractères dans une table de hachage et retourne soit un pointeur vers la cellule qui contient la chaîne, soit NULL ;
    - **supprimer_hachage** : prend en paramètre un pointeur sur une cellule d'une table de hachage et supprime cette cellule de la table.
5. Testez votre implémentation sur l'ensemble des fichiers de texte : insérez tous les mots dans une table de hachage où m = 11 et comptez ensuite le nombre de mots présents dans la table en

définissant la fonction **compter_table_hachage**. Vérifiez la similarité des résultats avec votre implémentation en liste et comparez leur temps d'exécution.

Partie 3 – Arbre binaire de recherche

Nous allons finalement expérimenter l'utilisation d'un arbre binaire de recherche.

1. Créez les fichiers **noeud.c** et **noeud.h** implémentant une structure de noeud permettant de stocker un mot et de le relier à un père, un fils gauche et un fils droit de même type. Définissez les procédures de noeud suivantes :
    - **initialiser_noeud** : initialise une cellule à l'aide d'un mot ;
    - **détruire_noeud** : libère (si nécessaire) les ressources mémoire d'un noeud.
2. Créez les fichiers **arbre.c** et **arbre.h** et implémentez-y une structure d'arbre binaire de recherche basée sur l'ordre lexicographique des mots (bobard < bobo < gogo < toto, facilitez-vous la vie en utilisant **strcmp** !) en implémentant les procédures suivantes :
    - **initialiser_arbre** : initialise un arbre vide ;
    - **détruire_arbre** : libère (si nécessaire) les ressources mémoire d'un arbre ;
    - **insérer** : prend en paramètre un pointeur sur un noeud et insère ce nœud à la bonne position dans l'arbre ;
    - **rechercher** : recherche un mot dans un arbre et retourne soit un pointeur sur le noeud qui contient le mot, soit NULL ;
    - **supprimer** : prend en paramètre un pointeur sur un nœud d'un arbre et supprime ce noeud de l'arbre.
    - **afficher** : affiche à l'écran les informations relatives à votre arbre.
3. Créez un programme de test permettant de vérifier le fonctionnement de votre implémentation en demandant à l'utilisateur de saisir des mots, en les insérant successivement dans un arbre et en affichant l'arbre ainsi créée. Essayez aussi de supprimer un mot, de rechercher un mot et de détruire l'arbre.
4. Testez votre implémentation sur l'ensemble des fichiers de texte : insérez tous les mots dans un arbre binaire de recherche et comptez ensuite le nombre de mots présents dans la table en définissant la fonction **compter_arbre**. Vérifiez la similarité des résultats avec vos implémentations précédentes et comparez leur temps d'exécution.