Algorithmique et structures de données 2
TP4

am@up8.edu

2022-2023

1 Consignes de rendu

TP à rendre pour le mercredi 08/03 20h (deadline fixe)

Un projet qui ne compile pas, qui termine sur une erreur de segmentation ou qui ne respecte pas les consignes ci-dessous ne sera pas corrigé.

- fichiers à rendre :
  - au minimum, fichiers `structure`.c et `structure`.h où vous définissez les structures et implémentez les fonctions demandées dans le tp. Si vous le souhaitez, vous pouvez créer un module pour l'implémentation sous forme de liste chaînée, un autre pour l'implémentation sous forme de table de hachage.
  - fichier `tp4.c` où vous testez les fonctions du module ci-dessus. Répondez aux questions du TP en commentaire en haut de votre fichier.
  - Makefile (cibles `all`, `clean`, `dist`)
- dépôt : archive `tp4_votre_nom.zip`
- vous pouvez ajouter un fichier `README.txt` au besoin (ne m'envoyez pas d'information complémentaire par mail / *via* mattermost).
- implémentez les fonctions nécessaire pour ne pas avoir de fuite mémoire (elles ne sont pas détaillées dans l'énoncé).

2 Annuaire sous forme de liste chaînée

2.1 Structure `contact`

Question 1 : définissez la structure

```
1 struct contact_s;
```

permettant d'implémenter l'annuaire sous forme de liste chaînée. Chaque objet doit stocker :

- le nom du contact ;
- le numéro de téléphone du contact ;

— un pointeur vers le contact suivant.

Vous pourrez utiliser **typedef** pour alléger votre code :

```
typedef struct contact_s contact;
```

2.2 Interaction avec la liste

Question 1 : implémentez la fonction

```
contact* ajouter_liste(contact *liste, char* nom, char* telephone);
```

Qui ajoute un contact en tête la liste et la renvoie.

La fonction `ajout_db_liste` est déjà définie et fait appel à la fonction `ajouter_liste`.

```
contact* ajout_db_liste(char *file_name, contact* liste);
```

Cette fonction permet d'ajouter la liste des contacts présents dans un fichier csv. Si votre exécutable est `a.out`, vous devez spécifier le fichier en argument de votre programme, par exemple :

```
$./a.out tp4_db.csv
```

Question 2 : Implémentez la fonction

```
void rechercher_liste(contact* c);
```

qui :

1. demande à l'utilisateur de saisir le prénom et le nom d'un contact,
2. recherche le contact dans la liste chaînée et renvoie son adresse de stockage en mémoire les informations du contact si il est présent dans la liste, affiche `"le contact n'est pas dans la liste"`, sinon.

2.3 Temps de recherche

Question 1 : Dans la fonction `main`, ajoutez-vous à une liste de contacts vide **puis** importez la base de données à l'aide de la fonction `ajout_db_liste`. Faites afficher le temps de recherche pour votre contact.

Question 2 : Répétez l'opération mais cette fois-ci commencez par importer la base **puis** ajoutez-vous à la liste. Comment évolue le temps de recherche pour votre contact ?

3 Annuaire sous forme de table de hachage

3.1 Structure `repertoire`

Question 1 : définissez la structure

```c
struct repertoire_s;
```

permettant d'implémenter l'annuaire sous forme de table de hachage. Chaque objet doit stocker :

— la taille de la table `taille_table` ;
— un tableau de listes chaînées

Vous pourrez utiliser **`typedef`** pour alléger votre code :

```c
typedef struct contact_s contact;
```

Question 2 : Implémentez la fonction `hash` qui calcule le hachage d'une chaîne de caractère selon la méthode proposée par Daniel J. Bernstein et connue sous le nom de `djb2` :

```c
unsigned long hash(char *str)
{
    unsigned long hash = 5381;
    int c;

    while (c = *str++)
        hash = ((hash << 5) + hash) + c; // hash * 33 + c

    return hash;
}
```

Question 3 : Implémentez la fonction `get_index` qui prend en paramètre la clé et la taille de la table de hachage et renvoie l'indice auquel l'élément doit être stocké, c'est à dire la valeur du hachage de la clé modulo la taille de la table.

3.2 Intéraction avec le répertoire

Question 1 : Implémentez la fonction

```c
repertoire* create_rep(int len);
```

qui initialise un répertoire. Le nombre de contact est initialisé à 0, la taille de la table à `len`, et l'espace mémoire est réservé pour le tableau de taille `len`.

Question 2 : Implémentez la fonction

```c
repertoire* ajouter_repertoire(repertoire *rep, char* nom, char*
    telephone);
```

qui ajoute un contact au répertoire.

Question 3 : En vous inspirant de la fonction `rechercher_liste`, implémentez la fonction :

```
void rechercher_rep(repertoire* rep, int length) {
```

qui se comporte comme `rechercher_liste` à la différence près que la recherche est effectuée dans le répertoire.

Question 4 : En vous inspirant de la fonction `ajout_db_liste`, implémentez la fonction :

```
repertoire* ajout_db_rep(char *file_name, repertoire* rep);
```

Question 4 : Implémentez une fonction d'affichage vous permettant de visualiser le contenu de la table de hachage.

3.3 Temps de recherche

Question 1 : Ajoutez vous à un répertoire vide de taille 10 et importez la base de contacts à votre répertoire. Quel est le temps de calcul de recherche de votre contact ? Que pensez-vous de ce temps par rapport au temps de recherche dans une liste chaînée ?

Question 2 : Testez différentes valeurs de taille de répertoire et commentez l'impact sur le temps de recherche.

3.4 Table de taille dynamique

Question 1 : Définissez une nouvelle structure permettant d'ajuster la taille de la table est sous-dimensionnée par rapport au nombre de contacts.

```
struct repertoire_dyn_s;
```

contenant également le nombre de contacts présents dans la table de hachage.

Question 2 : Implémentez les fonctions nécessaires pour permettre le comportement suivant : lorsque le nombre de contacts atteint 80% de la taille de la table de hachage, alors la taille de celle-ci est multipliée par deux.

Attention : la position des contacts déjà présents dans la table doit être recalculée !

Quelle est la taille de la structure après import de la base de données ?