Dictionnaires

***Projet à rendre avant le 13 janvier 2020**

Structures de données et algorithmes

1 Objectif

L'objectif du projet est d'écrire un algorithme efficace permettant de calculer l'intersection de deux fichiers (en terme de lignes). La solution à ce problème passe par l'utilisation d'une structure de données de type ensemble. Le projet consistera d'abord à implémenter cette structure de plusieurs manières différentes et à comparer théoriquement et empiriquement ces implémentations. On étudiera ensuite l'utilisation de cette structure pour calculer l'intersection de deux fichiers.

2 Ensemble

Un ensemble est une structure de données dynamique permettant de stocker des éléments, sans ordre particulier entre eux et tel que chaque élément de l'ensemble soit unique (pas de doublons permis au sein d'un ensemble). L'opération principale de cette structure est la possibilité de tester l'appartenance d'un élément à l'ensemble. Dans le contexte de ce projet, on supposera que cette structure fournit les opérations suivantes :

- `createEmptySet()` : crée un ensemble vide.
- `freeSet()` : libère un ensemble.
- `sizeOfSet(`S`)` : renvoie le nombre d'éléments contenus dans l'ensemble S.
- `insertInSet(`S, x`)` : insère l'élément x dans l'ensemble S.
- `contains(`S, x`)` : renvoie $True$ si l'élément x est contenu dans l'ensemble S, $False$ sinon.
- `setIntersection(`S_1, S_2`)` : renvoie un tableau contenant les éléments communs aux deux ensembles S_1 et S_2.

2.1 Implémentation

On vous demande d'implémenter une structure de type ensemble de deux manières différentes, à partir d'une table de hachage et à partir d'un arbre binaire de recherche (une troisième implémentation par liste liée vous est fournie). Pour simplifier l'implémentation et étant donné notre objectif, ces deux structures seront particularisées pour le stockage de chaînes de caractères.

Les fichiers suivants vous sont fournis :

- `Set.h` : le *header* définissant l'interface publique du `Set`.
- `ListSet.c` : une implémentation partielle du `Set` basée sur des listes chaînées. Cette implémentation est fournie à titre d'exemple.

*Adapté de P. Geurts, J.M. Begon, R. Mormont.

- `StringArray.h` et `StringArray.c` : une petite bibliothèque pour gérer des tableaux dynamiques (extensibles) de chaînes de caractères. Le résultat de l'intersection de deux ensembles devra être stocké dans une structure de ce type.

On vous demande d'implémenter :

1. l'interface du `Set` (définie dans `Set.h`) à l'aide d'une table de hachage dans un fichier `HashSet.c`.
2. l'interface du `Set` à l'aide d'un arbre binaire de recherche dans un fichier `TreeSet.c`.

Pour la table de hachage, vous pouvez choisir le système de gestion de collision que vous souhaitez. Votre programme doit rester *efficace* quel que soit le nombre d'éléments qui seront stockés dans l'ensemble et ce nombre ne pourra pas supposer être connu à l'avance. Cela implique qu'une stratégie de rehachage devra être implémentée.

L'utilisation d'arbres binaires de recherche équilibré (e.g. AVL) n'est pas obligatoire et sera considéré comme bonus.

2.1.1 Analyse

Dans le rapport, on vous demande de répondre aux questions suivantes :

1. Pour les deux implémentations :
   a) Expliquez brièvement vos choix d'implémentation.
   b) Expliquez le principe de la fonction d'intersection, en donnant son pseudo-code dans le cas de l'arbre binaire de recherche.
   c) Donnez et justifiez la complexité au pire cas de la fonction `setIntersection` en fonction des tailles N_A et N_B des deux ensembles (en supposant que $N_A \leq N_B$).
2. Pour vos deux implémentations d'ensemble, arbres binaires de recherches et tables de hachage, ainsi que pour l'implémentation sur base de listes chaînées qui vous est fournie :
   - Calculez les temps (moyens) nécessaires à l'insertion de N valeurs dans l'ensemble pour des valeurs de N croissantes.
   - Calculez les temps (moyens) nécessaires à une recherche dans un ensemble contenant N éléments pour des valeurs de N croissantes. Reportez les résultats sur deux graphes séparés. Vous êtes libres de choisir l'outil de génération de graphe qui vous semble approprié (e.g. gnuplot).
3. Discutez ces résultats par rapport aux résultats théoriques attendus.

Suggestion : Pour générer les courbes demandées au point 2, nous vous suggérons de procéder de la manière suivante, pour un N fixé :

- Calculez le temps nécessaire pour insérer les N premières chaînes du fichier `French.txt` fourni.
- Calculez le temps nécessaire pour rechercher les chaînes du fichier `English.txt` dans l'ensemble obtenu à l'étape précédente (qui est donc bien de taille N).

Pour obtenir des courbes plus stables, vous pouvez faire des moyennes des temps obtenus en répétant l'expérience précédente en mélangeant les éléments du fichier `French.txt`. Une fonction `shuffleArray` vous est fournie à cet effet dans le fichier `StringArray.c`.

3 Intersection de deux fichiers

En se basant sur une structure d'ensemble, on aimerait maintenant implémenter un programme permettant d'afficher les lignes communes à deux fichiers passés en argument. Par exemple, si les fichiers `A.txt` et `B.txt` ont les contenus suivants :

A.txt:	B.txt:
eee	eee
bbb	fff
ddd	bbb
ccc	
aaa	

Le résultat de l'application du programme affichera les lignes `bbb` et `eee`. L'ordre d'affichage des lignes communes n'a pas d'importance.

3.1 Implémentation

Les fichiers suivants vous sont fournis :

- `StringArray`.h et `StringArray.c` : une petite bibliothèque pour gérer les tableaux de chaînes de caractères.
- `main.c` : le main d'un programme, prenant deux paramètres en entrées : les deux fichiers à comparer, et calculant l'intersection de ceux-ci.
- Deux fichiers de tests réels `French.txt` et `English.txt`.

En vous basant sur ces fichiers ainsi que sur vos implémentations d'ensemble, on vous demande d'implémenter l'algorithme générique d'intersection dans un fichier `Intersection.c`. Il correspond à la fonction `getIntersection` définie dans `StringArray.h`. Cette fonction suppose que les chaînes contenues dans les fichiers à comparer ont été préalablement chargées dans deux tableaux de chaînes de caractères, cette étape étant prise en charge pour vous par le fonction main fournie. Votre choix d'implémentation de cette fonction devra découler de l'analyse théorique qui vous est demandée au point 3.2 ci-dessous.

Une fois la fonction `getIntersection` implémentée, vous pourrez tester l'implémentation par liste liée à l'aide des commandes :

```
gcc main.c StringArray.c ListSet.c Intersection.c --std=c99 -o intersection
./intersection A.txt B.txt
```

3.2 Analyse

Sans perte de généralité, supposons que les fichiers aient été chargés dans deux tableaux A et B, respectivement de longueurs N_A et N_B et que $N_A \leq N_B$. Supposons également que chacun des tableaux A et B ne contienne aucun doublon.

1. Quelle est la complexité théorique minimale pour calculer l'intersection de A et B, en fonction de N_A et N_B, quelle que soit la solution adoptée ? (sans faire d'autres hypothèses sur le contenu de A et B). Justifiez votre réponse.
2. On considérera deux approches possibles :
   - La première consiste à lire et à stocker dans un ensemble S_A les éléments de A et à ensuite parcourir les éléments de B en ne retenant que ceux qui se trouvent dans l'ensemble S_A.
   - La seconde consiste à stocker les éléments de A et les éléments de B dans deux ensembles séparés S_A et S_B et ensuite à appeler la fonction `setIntersection` sur ces deux ensembles pour calculer leur intersection.

a) Donnez et justifiez la complexité au pire cas de ces deux approches pour vos deux implémentations d'ensemble, par arbre binaire de recherche et par table de hachage.

b) Sur base de cette analyse, concluez sur la pertinence relative des deux approches proposées et sur l'implémentation d'ensemble la plus appropriée.

3. Si on pouvait supposer que A et B sont triés par ordre alphabétique, comment procéderiez-vous et quelle serait la complexité de votre solution aux meilleur et pire cas ?

4 Deadline et soumission

Le projet est à réaliser individuellement ou en groupes de 3 étudiants maximum pour le **13 janvier 2020, 23h59 au plus tard**. Il doit être soumis via l'ENT, sur la page du cours :

`https://moodle.univ-paris13.fr/course/view.php?id=963`

Le projet doit être rendu sous la forme d'une archive `tar.gz` contenant :

- Votre rapport (5 pages maximum) au format PDF. Soyez bref mais précis et respectez bien la numération des (sous-)questions.
- Les fichiers mentionnés `Intersection.c`, `HashSet.c` et `TreeSet.c`.

Vos fichiers seront évalués avec les commandes :

```
gcc main.c StringArray.c HashSet.c Intersection.c --std=c99 --pedantic -Wall -
    Wextra -Wmissing-prototypes -o intersection

gcc main.c StringArray.c TreeSet.c Intersection.c --std=c99 --pedantic -Wall -
    Wextra -Wmissing-prototypes -o intersection
```

Ceci implique que :

- Les noms des fichiers doivent être respectés.
- Le projet doit être réalisé dans le standard C99.
- La présence de *warnings* impactera négativement la note finale.
- Un projet qui ne compile pas avec ces commandes recevra une note nulle pour la partie code du projet.

Un projet non rendu à temps recevra une pénalité. En cas de plagiat avéré, l'étudiant se verra affecter une note nulle.