Algorithmique Avancée **Devoir de Programmation**

---

*Devoir à faire en **binôme**. Soutenance en séance de TD/TME 10 (entre le 11/12/23 et le 14/12/23). Rapport (de format pdf, d'une dizaine de pages), Code Source, et transparents de Présentation à déposer sur Moodle au plus tard le 15/12 à 23h59, dans une archive nommée Nom1_Nom2_ALGAV.zip (où Nom1 et Nom2 sont vos noms de famille).*
Langage de programmation libre ; le code source doit contenir un makefile ou un script permettant à l'enseignant de reproduire vos tests.

1 Présentation

Le but du problème consiste à visualiser graphiquement les temps d'exécution sur des données réelles des algorithmes et structures de données que nous avons introduits dans les chapitres 1, 2 et 3 du module.
Il est attendu un soin particulier concernant la réflexion et la mise en place concernant les expérimentations dans ce devoir.

1.1 Échauffement

Dans tout le devoir, les clés sont des entiers codés sur 128 bits : on peut, par exemple, utiliser un quadruplet de *4 entiers non signés de 32 bits* pour les représenter. On n'autorise pas l'utilisation des entiers de taille arbitraire de votre langage de programmation.

Question 1.1 Choisir une représentation d'une clé 128 bits et l'expliquer.

Question 1.2 Étant donné 2 clés 128 bits, écrire un prédicat `inf` (une fonction renvoyant un booléen) permettant de déterminer si `cle1` est strictement inférieure à `cle2`.

Question 1.3 Étant donné 2 clés 128 bits, écrire un prédicat `eg` permettant de déterminer si `cle1` et `cle2` sont égales.

Par la suite, un *jeu de données aléatoires* est disponible sur Moodle. Il y a 40 listes de clés de 128 bits encodées en hexadécimal (une clé par ligne dans chaque fichier) contenant chacune 1000, 5000, 10000, 20000, 50000, 80000, 120000, 200000 clés.

2 Structure 1 : Tas priorité *min*

En cours, nous avons introduit la structure de données de tas *min*. Nous allons la représenter en mémoire avec deux structures distinctes : via un arbre binaire et via un tableau.

Question 2.4 Implémenter les 3 fonctions fondamentales d'un tas *min* :
`SupprMin, Ajout, AjoutsIteratifs`.

Ces fonctions permettent respectivement de supprimer l'élément de clé minimale de la structure, d'ajouter un élément à un tas. La fonction `AjoutsIteratifs` prend en argument une liste de clés et construit le tas correspondant par ajouts simples dans un tas (on fait appel à `Ajout`).

Question 2.5 La fonction `Construction` permet de construire de façon plus efficace un tas à partir d'une liste de clés (toutes distinctes). Pour ce faire on insère tout d'abord les clés dans une structure arborescente qui a la bonne forme (ou dans un tableau) puis on effectue des remontées afin d'obtenir la croissance le long des branches (ou la croissance adéquate dans le tableau). Ces remontées doivent être effectués de manière non naïves pour assurer la construction en temps linéaire du tas à partir d'une

liste de clés.
Donner le pseudo code de la fonction avec des explications puis l'implémenter.

Question 2.6 La fonction `Union` prend en arguments deux tas ne partageant aucune clé commune, et construit un tas qui contient l'ensemble de toutes les clés. Proposer un pseudo-code expliqué afin de garantir la complexité linéaire en la somme des tailles des deux tas, puis l'implémenter.

Question 2.7 Prouver que les complexités (au pire cas) présentées dans le cours sont vérifiées dans vos 5 implémentations (on sera particulièrement méticuleux pour l'étude de la complexité de `Construction`).

Pour chacune des deux structures effectuer les expérimentations suivantes.

Question 2.8 Utiliser les *jeux de données aléatoires* afin de vérifier graphiquement les complexités temporelles des fonctions `AjoutsIteratifs, Construction`. Pour chaque taille de listes, calculer le temps de construction en mémoire de la structure de données et en faire la moyenne. Puis représenter sur un graphique les moyennes pour chacune des tailles. Interpréter ce graphique vis-à-vis des complexités théoriques. Si les données ne permettent pas de conclure, proposer d'autres de jeux de données plus probants.

Question 2.9 Utiliser les *jeux de données aléatoires* afin de vérifier graphiquement la complexité temporelle de la fonction `Union`.

3 Structure 2 : Files binomiales

Question 3.10 Définir et encoder les primitives de base concernant les files binomiales.

Question 3.11 Implémenter les 4 fonctions fondamentales d'une files binomiale :
`SupprMin, Ajout,Construction, Union`. La construction se fait par ajouts itératifs (cf. cours).

Question 3.12 Utiliser les *jeux de données aléatoires* afin de vérifier graphiquement la complexité temporelle de la fonction `Construction`.

Question 3.13 Utiliser les *jeux de données aléatoires* afin de vérifier graphiquement la complexité temporelle de la fonction `Union`. Reprendre la même stratégie que celle de la Question 2.8.

Pour les deux dernières questions, si les jeux de données fournis ne permettent pas de visualiser l'ordre de grandeur de la complexité, augmenter les expérimentations en indiquant comment vous les avez définies.

4 Fonction de hachage

Pour cette section, définir et implémenter la fonction de hachage classique nommée *Message Digest 5*, ou `MD5`. Implémenter l'algorithme dont le pseudo-code est donné sur la page wikipedia de l'algorithme : `https://fr.wikipedia.org/wiki/MD5`.

5 Arbre de Recherche

Implémenter une structure arborescente de recherche permettant, en moyenne, de savoir si un élément est contenu dans la structure de données en $O(\log n)$ comparaisons, où n est le nombre de clés stockées. On rappelle que les clés seront codées sur 128 bits.

6 Étude expérimentale

La dernière partie du devoir consiste à comparer expérimentalement les structures, sur des données réelles. On utilisera les mots de l'œuvre de Shakespeare [1]. Les mots que nous considérons sont construits sur l'alphabet du code ASCII. Celui-ci est composé de 128 caractères, chacun étant encodé sur 8 bits.

Question 6.14 Stocker dans une structure arborescente de recherche, le haché `MD5` de chaque mot, c'est-à-dire le résultat de l'application de la fonction `MD5` sur chaque mot. En parallèle, construire une liste des mots de l'œuvre de Shakespeare où chaque mot n'apparaît qu'une seule fois.
L'ordre d'apparition des mots dans le résultat doit correspondre à l'ordre induit par la première occurrence de chaque mot des fichiers pris en entrée. On veut une seule liste de mots pour l'ensemble de tous les fichiers de l'archive.

Question 6.15 Quels sont les ensembles des mots différents de l'œuvre de Shakespeare qui sont en collision pour *MD5* ?

Question 6.16 Comparer graphiquement les temps d'exécution des algorithmes `SupprMin, Ajout, Construction, Union` pour les deux types de structure de données : tas *min* et files binomiales sur les données extraites de la question 6.14.
Pour l'algorithme `Union` expliquer quelles données sont utilisées.

1. Une archive de chaque œuvre est disponible sur la page moodle de l'UE.