Tri Lisse

Samy Jaziri

Sujet de khôlle inspiré du sujet du Concours Centrale Option Informatique 2016

Préambule

Tous les algorithmes de ce sujet devront être implémentés en ***C****.*

Pour vous mettre dans les conditions du concours, vous n'avez pas le droit d'accéder aux ressources en ligne.

Vous indiquerez vos réponses sur la fiche réponse qui vous a été fournie en utilisant, en entrée de vos programmes, le numéro u_0 inscrit sur cette fiche. Vous remettrez cette fiche à l'examinateur en fin de séance. Une fiche réponse est mise à disposition à la fin du sujet en tant qu'exemple des réponses attendues pour un $\widetilde{u_0}$ particulier.
En ce qui concerne les questions orales de la khôlle, lorsque la description d'un algorithme est demandée, vous devez présenter son fonctionnement de façon schématique, courte et précise. Vous ne devez pas expliquer votre code ligne par ligne! Quand la complexité d'un algorithme est demandée en temps ou en mémoire en fonction d'un paramètre n, on demande l'ordre de grandeur en fonction du paramètre, donné en notation de Landau ($\mathcal{O}(n), \mathcal{O}(\log(n)), \dots$). Prenez des notes lorsque vous préparez une question orale pour retrouver plus rapidement les grandes lignes de votre explication lorsque l'examinateur passe vous voir.
Il est recommandé de **tester vos programmes sur des petits exemples** que vous aurez résolus préalablement à la main ou bien à l'aide de la fiche réponse type fournie en annexe.
Il vous est demandé d'aborder les questions dans l'ordre et de noter vos difficultés à répondre à une question avant de passer à la suivante. Vous pourrez alors les aborder avec l'examinateur.
Attention, une attention particulière sera portée aux problèmes de fuite de mémoire lorsque vous utiliserez l'allocation dynamique

1 Introduction

Motivation

Trier des données est un problème récurrent dans tous les systèmes d'information. Dans de nombreuses applications réelles, l'étude de la complexité au cas pire est importante. Or les données à trier ne sont souvent pas quelconques. Il est fréquent que les données soient déjà presque triées. Pourtant, de nombreux algorithmes atteignent leur complexité maximale lorsque les données sont déjà triées.
Le but de ce problème est d'étudier l'algorithme du *tri lisse* (smoothsort), proche du tri par tas, mais présentant des performances intéressantes lorsque les données qu'il reçoit sont presque triées.
Dans tout le problème, on triera, par ordre croissant, des valeurs entières.
Dans toutes les questions de complexité en temps, la mesure de complexité à considérer est le nombre de comparaisons par la relation d'ordre $\leq$.

Préliminaires

On tiendra compte dans la suite que la structure à trier n'est pas un ensemble, car le même élément peut-être répété plusieurs fois. Une même valeur peut donc apparaître plusieurs fois dans une liste ou un arbre. De manière générale lorsqu'on dit que deux structures contiennent les mêmes éléments, ce sera toujours en tenant compte des répétitions.

Tri par insertion

Question 1

En utilisant une liste simplement chaînée, implémentez le type `liste` représentant une liste d'entiers positifs et supportant les opérations suivantes :

- `liste* creer_liste()` renvoie un pointeur vers une liste vide allouée dynamiquement.
- `void libere_liste(liste*)` libère la mémoire utilisée par une liste.
- `int nth(liste*, int i)` renvoie l'élément d'indice `i` de la liste en $\mathcal{O}(\texttt{i})$ où `i` sera considéré comme étant un indice valide de la liste.

Question 2

Implémenter une fonction `void insere(liste* l, int x);` qui prend en argument une liste triée `l` et insère `x` dans `l` en maintenant `l` triée.

Question 3

Implémenter une fonction `liste* tri_insertion(liste* l);` qui utilise la fonction précédente pour renvoyé une liste triée ayant les mêmes éléments que la liste reçu en argument.

Préparer une réponse à donner à l'oral

Quelle est la complexité dans le pire cas de l'appel de `tri_insertion` ? Quelle est la complexité de l'appel de `tri_insertion` sur une liste triée ?

Listes aléatoires

Considérons $(u_k)_{k\geq 0}$ la suite d'entiers définie par :

$$u_k = \begin{cases} u_0 \text{ (sur votre fiche réponse)} & \text{si } k = 0 \\ 15091 \times u_{k-1} \mod 64007 & \text{si } k > 0 \end{cases}$$

Pour tout $n \in \mathbb{N}$, on défini l'ensemble

$$E_n = \{u_i, i \in [\![0, n]\!]\}$$

Question 4

Pour chaque couple d'ensemble et d'entier (E_n, i), si on considère que $e_0, e_1, \ldots, e_n$ sont les éléments de E_n *rangés dans l'ordre croissant*, donnez la valeur de e_i.
a) $(E_{100}, 21)$ **b)** $(E_{1000}, 144)$ **c)** $(E_{10000}, 1212)$

2 Tas binaires

Type arbre

Question 5

Implémenter le type `arbre` représentant un arbre binaire étiqueté par des entiers positifs et supportant les opérations suivantes :

- `arbre* creer_arbre(int x,arbre* fg, arbre* fd)` renvoie un pointeur vers un arbre alloué dynamiquement dont la racine est étiquetée par `x` et ayant l'arbre pointé par `fg` pour fils gauche et celui pointé par `fd` pour fils droit. La mémoire allouée pour `fg` et `fd` qui n'est pas utilisée dans le nouvelle arbre doit être libérée.
- `void libere_arbre(arbre*)` libere la mémoire utilisée par un arbre.

Un *tas binaire parfait* est un arbre vide ou un arbre binaire parfait dont la valeur de chaque nœud est inférieure ou égale à celle de ses fils. Un *quasi-tas* est un arbre non vide dont le fils gauche et le fils droit sont des *tas binaires parfaits* de même taille. Aucune contrainte n'est donnée sur l'étiquette de la racine.
Soit A un arbre, on note $\mathcal{H}(A)$ sa hauteur (l'arbre vide est de hauteur 0) et $|A|$ sa taille (nombre de nœuds). On note $\min(A)$ le minimum des étiquettes de A. On note enfin m_k la taille d'un arbre binaire parfait de hauteur $k \geq 0$.

Préparer une réponse à donner à l'oral

Déterminez m_k. On attend une preuve rigoureuse.

Question 6

Implémenter une fonction `int min_tas(arbre* a);` renvoyant $\min($`a`$)$ en $\mathcal{O}(1)$ pour `a` un tas binaire parfait considéré non vide.

Question 7

Implémenter une fonction `int min_quasi(arbre* a);` renvoyant min(`a`) en $\mathcal{O}(1)$ pour `a` un quasi-tas.

Transformation d'un quasi-tas en tas

Question 8

Implémenter une fonction `void percole(arbre* a);` qui ne fait rien si `a` est vide et transforme `a` en un tas binaire parfait ayant les mêmes éléments si `a` est un quasi-tas.

Préparer une réponse à donner à l'oral

Quelle est la complexité au pire cas de `percole` en fonction de la hauteur de l'arbre passé en argument ? Au meilleur cas ?

3 Création d'une liste de tas

On admettra dans la suite que pour tout entier naturel n, il existe un unique entier naturel r et un unique r-uplet $(k_1, \dots, k_r)$ tels que

- $n = m_{k_1} + \cdots + m_{k_2}$
- $(k_1, \dots, k_r)$ est *quasi strictement croissant* (QSC), *i.e* $k_1 \leq k_2 < k_3 < \cdots < k_r$.

Cette décomposition s'appelle décomposition parfaite de n.
Exemples : $1 = m_1$, $2 = m_1 + m_1$, $3 = m_2$, $4 = m_1 + m_2$, $5 = m_1 + m_1 + m_2$.
Soit $n = m_{k_1} + \cdots + m_{k_r}$ un entier naturel et sa décomposition parfaite. On admettra la formule suivante :

$$(\star)\ n+1 = \begin{cases} m_{k_1+1} + m_{k_3} + \cdots + m_{k_r} & \text{si } r \geq 2 \text{ et } k_1 = k_2 \\ m_1 + m_{k_1} + \cdots + m_{k_r} & \text{sinon} \end{cases}$$

Type liste de tas

On appelle *liste de tas* une liste de couples de la forme (A, t) où a désigne un arbre binaire parfait et $t = |A|$.

Question 9

En utilisant une liste doublement chaînée, implémentez le type `liste_tas` représentant une liste de tas et supportant les opérations suivantes :

- `liste_tas* creer_liste_tas()` renvoie un pointeur vers une liste de tas vide allouée dynamiquement.
- `void libere_liste_tas(liste*)` libère la mémoire utilisée par une liste de tas.
- `void insere_tas(liste_tas*, arbre*, int)` insère un couple en tête de liste.
- `void extrait_tas(liste_tas* lt, arbre** a, int* p)` prend en argument une liste de tas qu'on considérera **non vide** et supprime le premier élément – (A, t) – de la liste. La fonction sauvegarde l'adresse de A à l'adresse pointée par `a` et t à l'adresse pointée par `p`. La mémoire utilisée par le noeud de la liste supprimé est libérée, mais attention à ne pas libérer la mémoire réservée pour A et t.

Soit h une liste de tas, on défini :

- $\mathcal{L}(h)$, la longueur de la liste h.
- $|h|$, la taille de h, par la somme des tailles des tas de la liste.
- $\mathcal{H}(h)$, la hauteur de h, par le maximum des hauteurs des tas de la liste.
- $\min(h)$, le minimum de h, par le minimum des valeurs apparaissant dans un tas quelconque de la liste.

On dit que h *vérifie la condition* TC (tas croissant) si la liste des tailles des tas de h *forment une décomposition parfaite* de $|h|$, *i.e* si $h = [(a_1, t_1), \dots, (a_r, t_r)]$,

- $\forall 1 \leq i \leq r, \exists k_i, t_i = m_{k_i}$,
- (k_1,...,k_r)$ est QSC
- $|h| = t_1 + \cdots + t_r$.

Préparer une réponse à donner à l'oral

Montrez que pour toute liste de tas h vérifiant TC,

- $\mathcal{L}(h) = \mathcal{O}(\log(|h|))$
- $\mathcal{H}(h) = \mathcal{O}(\log(|h|))$

Ajout dans une liste de tas

On cherche à ajouter une valeur dans une liste de tas. On forme un arbre restreint à un seul nœud étiqueté par cette valeur. Lorsque l'on ajoute cet arbre en tête de la liste, la condition TC n'est pas nécessairement conservée.

Préparer une réponse à donner à l'oral

Dans quels cas la condition TC est-elle perdue ? Décrivez une méthode pour transformer la liste de tas obtenue en ajoutant une valeur en tête de la liste, en une liste de tas contenant les mêmes valeurs et respectant la condition TC. La complexité au pire devra être en $\mathcal{O}(k)$, k étant la hauteur du premier tas de la liste.
Inspirez vous du résultat de la formule $(\star)$.

Question 10

Implémenter une fonction `void ajoute(liste_tas* h, int x);` qui prend en argument une liste de tas `h` vérifiant la condition TC, un entier `x` et transforme `h` en une liste de tas vérifiant la condition TC et contenant les mêmes éléments que `h`, auxquels s'ajoute `x`.

Transformation d'une liste en une liste de tas

Question 11

Implémenter une fonction `liste_tas* constr_liste_tas(liste* l);` qui renvoie un pointeur vers une liste de tas vérifiant la condition TC et contenant les mêmes éléments que `l`.

Préparer une réponse à donner à l'oral

Montrez que la complexité au pire cas de `constr_liste_tas(l)` est en $\mathcal{O}(|l| \log(|l|))$ et que la complexité si `l` est triée est $\mathcal{O}(|l|)$.

On note H_n la liste de tas obtenu depuis E_n par l'appel de la fonction `constr_liste_tas`.

Question 12

Pour chaque liste de tas ci-dessous, donnez pour chaque tas de la liste, dans l'ordre, l'étiquette de sa racine et la taille du tas.
a) H_{100} **b)** H_{1000} **c)** H_{10000}

4 Tri des racines

On dit qu'une liste de tas $h = [(a_1, t_1), \dots, (a_r, t_r)]$ vérifie la condition RO (pour *racines ordonnées*) si $\min(a_1) \leq \cdots \leq \min(a_r)$.
On cherche dans cette section un algorithme permettant de transformer une liste de tas vérifiant la condition TC en une liste de tas vérifiant TC et RO et ayant les mêmes éléments.

Question 13

Implémenter une fonction `void echange_racine(arbre*, arbre*);` qui prend deux arbres considérés non vide en argument et échange leurs racines en $\mathcal{O}(1)$.

Soit $h = [(a_1, t_1), \dots, (a_r, t_r)]$ une liste de tas vérifiant RO et a un quasi-tas de taille t. On remarque que :

- si $\min(a) \leq \min(a_1)$ alors $[(\texttt{percole(a)}, t), (a_1, t_1), \ldots, (a_r, t_r)]$ est une liste de tas vérifiant RO.
- si $\min(a) > \min(a_1)$ alors, après l'appel de `echange_racine(a,a_1)`, a est un tas binaire parfait, a_1 est un quasi-tas et $\min(a) \leq \min(a_1)$.

Question 14

Implémenter une fonction `void insere_tas_ro(liste_tas*, arbre*, int);` qui prend en argument une liste de tas `h` vérifiant RO, un tas binaire parfait `a` et sa taille et qui ajoute `(a,t)` en tête de `h` et permute les étiquettes de cette liste en utilisant les remarque ci-dessus pour lui faire vérifier la condition RO.

Préparer une réponse à donner à l'oral

Quelle est la complexité au pire cas de `insere_tas_ro` ? Quelle est la complexité de `insere_tas_ro` dans le cas où le minimum du tas inséré est plus petit que toutes les valeurs apparaissant dans la liste de tas ?

On utilise cette fonction pour réaliser un tri des racines à la manière du tri par insertion.

Question 15

Implémenter une fonction `void tri_racine(liste_tas*);` qui prend en argument une liste de tas vérifiant TC et la transforme en une liste de tas ayant les mêmes éléments et vérifiant TC et RO.

Préparer une réponse à donner à l'oral

Montrer que `tri_racine` appliquée à une liste de tas h vérifiant TC, a une complexité au pire cas en $\mathcal{O}(\log(|h|)^2)$.

Question 16

Pour chaque liste de tas ci-dessous, donnez, après tri des racines, la racine du dernier tas de la liste.
a) H_{100} **b)** H_{1000} **c)** H_{10000}

5 Extraction des éléments d'une liste de tas.

On souhaite dans cette section récupérer une liste d'étiquettes à partir d'une liste de tas vérifiant TC et RO. Supposons que le premier tas de la liste de tas h possède une racine étiquetée par x, un fils gauche a_1 de taille t_1 et un fils droit a_2 de taille t_2. Pour supprimer x de h en maintenant les propriétés RO et TC nous admettrons qu'il suffit de supprimer le premier nœud de h puis d'appeler `insert_tas_ro(insert_tas_ro(h,a_2,t_2),a_1,t_1)`. On précise que t_1 et t_2 peuvent êtres calculées en temps constant.

Question 17

Implémenter la fonction `int extraire(liste_tas*)` qui renvoie la racine du premier élément d'une liste de tas tout en préservant les conditions RO et TC.

Préparer une réponse à donner à l'oral

Montrez que la fonction a une complexité en $\mathcal{O}(\log(|h|))$. Montrez que si les éléments du premier tas sont tous inférieurs aux autres éléments apparaissant dans la liste de tas, alors la complexité est en $\mathcal{O}(1)$.

6 Tri lisse

Question 18

Implémenter une fonction `liste* tri_lisse(liste*)` qui trie une liste en construisant une liste de tas intermédiaire vérifiant RO et TC avant d'en extraire les éléments.

Préparer une réponse à donner à l'oral

Montrez que la complexité au pire du tri lisse est en $\mathcal{O}(|l| \log(|l|))$. Montrez que la complexité du tri lisse appelé sur une liste triée est $\mathcal{O}(|l|)$.

Fiche réponse : Tri Lisse

Nom, prénom : DOUZ Nadine $\widetilde{u_0}$: 12

Question 4:

a) 13285

b) 9825

c) 7651

Question 12:

a) [(57904, 7) (35416, 31) (12, 63)]

b) [(63392, 7) (63052, 7) (60987, 31) (56935, 63) (47990, 127) (32817, 255) (12, 511)]

[(64003, 3) (63976, 3) (63894, 15) (62387, 255) (59237, 511) (52491, 1023) (12, 8191)]

Question 16:

c) 1131

d) 599

e) 50