Algorithmique Avancée — Examen Réparti 2

Les seuls documents autorisés sont les polys de cours, ainsi que la copie personnelle. Le barème est indicatif.

Exercice 1 *: Tournoi binomial et file binomiale [7 points]*

On a un ensemble de n clés (qui sont des entiers distincts) à stocker dans une file binomiale.

Question 1 Donner un algorithme afin de construire une file binomiale contenant toutes les clés. Pour ce faire, on va s'appuyer sur l'approche permettant de construire l'arbre optimal de Huffman (dans le contexte de la compression) : on va fusionner les plus petits tournois de même taille 2 par 2 et itérer ce processus.
On ne pourra utiliser que les primitives suivantes :

- `ConsTB0` : qui prend une clé et renvoie un tournoi de taille 1 contenant l'entier.
- `Union2Tid` : qui prend en entrée 2 tournois de même taille et qui contruit leur union.
- `AjoutMin` : qui prend un tournoi et une file, de telle sorte que le degré du tournoi est strictement plus petit que le tournoi de degré minimal de la file et qui renvoie la file augmentée de ce tournoi.

Question 2 Indiquer quelles sont la mesure de complexité et la notion de complexité, (complexité au pire cas, en moyenne ou amortie), adéquates. Quelle est la complexité de l'algorithme dans ce contexte ?

Question 3 En supposant que le nombre de clés est une puissance de 2 c'est-à-dire $n = 2^r$, que peut-on en déduire sur la forme de la file construite à la Question 1 ?

Question 4 Donner un algorithme, de type diviser pour régner, qui à chaque étape sépare les données en deux ensembles de taille identique, prenant en paramètre un tournoi binomial et permettant de trier dans l'ordre décroissant l'ensemble des clés du tournoi.
On ne pourra utiliser que la primitive : `SepareT` : qui prend un tournoi T en entrée qui renvoie un couple (T', T'') tel que T' est l'enfant gauche de T et T'' est T dans lequel on a supprimé T'.

Question 5 Indiquer quelles sont la mesure de complexité et la notion de complexité, (complexité au pire cas, en moyenne ou amortie), adéquates. Quelle est la complexité de l'algorithme dans ce contexte ?

Exercice 2 *: Hachage coucou [9 points]*

Le *hachage coucou* est une technique de hachage qui utilise deux fonctions de hachage h_1 et h_2. Ces deux fonctions sont définies sur un univers de n clés et sont à valeur dans $\{1, \ldots, r\}$, avec $r > n$. On suppose que :

- h_1 et h_2 répartissent uniformément les clés, *i.e.*
  pour tout $i \in \{1, \ldots, r\}$ on a $\mathbb{P}(h_1 = i) = \mathbb{P}(h_2 = i) = \frac{1}{r}$;
- h_1 et h_2 sont indépendantes, *i.e.*
  pour tous $i, j \in \{1, \ldots, r\}$, on a $\mathbb{P}(h_1 = i, h_2 = j) = \mathbb{P}(h_1 = i) \times \mathbb{P}(h_2 = j)$.

Les clés sont réparties dans une table de hachage $T[1..r]$, **chaque clé x peut être placée à la position $h_1(x)$ ou à la position $h_2(x)$ dans la table T.**

Pour insérer une clé x dans la table T, on calcule sa position $i = h_1(x)$. Si la case $T[i]$ est vide on y met la clé x, sinon on éjecte la clé y déjà présente et on la remplace par x. Il faut alors placer la clé y à nouveau dans la table. Pour cela, on calcule l'autre position j de y (si $i = h_1(y)$ alors $j = h_2(y)$ sinon $j = h_1(y)$). On recommence avec y ce qu'on a fait avec x (si $T[j]$ est vide on y met y, sinon...). Le processus s'arrête lorsqu'on tombe sur une case vide ou lorsqu'on a atteint le nombre maximal d'itérations, que l'on a fixé au préalable (égal au nombre n de clés). Dans ce dernier cas, deux nouvelles fonctions de hachage sont choisies et on reconstruit toute la table (on dit qu'il y a un *re-hachage*). Il est possible que ce re-hachage n'aboutisse pas non plus, auquel cas on a recours à un deuxième re-hachage (et éventuellement à un troisième, etc).

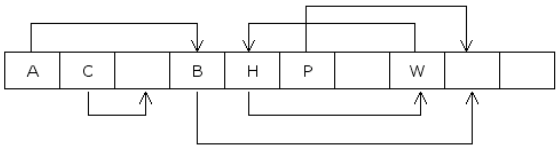
À gauche voici un pseudo-code pour la procédure d'insertion d'une clé x dans une table T.

```
def Inserer(T, x):
    """ Table * cle -> Nonetype
    Renvoie la table T apres insertion de x."""

    if T[h1(x)] != x and T[h2(x)] != x:
        pos = h1(x)
        for i in range(n):
            if estVide(T[pos]):
                T[pos] = x
                return None
            else:
                tmp = x
                x = T[pos]
                T[pos] = tmp

                if pos == h1(x):
                    pos = h2(x)
                else:
                    pos = h1(x)
        Re-Hacher(T)
        Inserer(T, x)
    return None
```

```
# != signifie "différent de"
# EstVide prédicat indiquant si une case est vide
```



On représente le hachage coucou des clés A, B, C, H, P, W dans une table $T[1..10]$. Chaque clé x est placée à une position correspondant à l'une de ses deux valeurs de hachage et une flèche indique l'autre position possible de x dans la table (correspondant à l'autre valeur de hachage de x).

Remarque : Après avoir vérifié que x n'est pas présente dans la table, la procédure n'examine pas les deux positions $h_1(x)$ et $h_2(x)$ de la clé à insérer, mais seulement la position $h_1(x)$. La position $h_2(x)$ sera éventuellement examinée si l'on retombe sur la position $h_1(x)$ lors du passage dans la boucle `for`. Cette situation se présentera dans les exemples.

Question 1 Réaliser l'insertion (à partir de la table T à droite du pseudo-code) de la clé Z ayant comme valeurs de hachage $h_1(Z) = 5$ et $h_2(Z) = 1$. Justifier la réponse en exhibant les étapes intermédiaires.

Question 2 Peut-on insérer dans la table T précédente, une clé V ayant comme valeurs de hachage $h_1(V) = 5$ et $h_2(V) = 8$? Justifier la réponse.

Si x est un entier, on désigne par $b_0(x), b_1(x), \ldots, b_k(x)$ les bits de x dans l'écriture binaire de x. Autrement dit, $x = b_k(x)2^k + \ldots b_1(x)2^1 + b_0(x)2^0$, avec $b_i(x) = 0$ ou 1. On veut réaliser le hachage coucou de clés entières en utilisant les opérations & et `rot` ainsi définies :

- si x et y sont deux entiers naturels alors $x\&y$ est l'entier z tel que $b_i(z) = 1$ ssi $b_i(x) = 1$ et $b_i(y) = 1$.
- si x est un entier naturel alors $\texttt{rot}(x, j)$ est l'entier obtenu en faisant une rotation circulaire de j bits vers la droite dans la représentation binaire de x. Autrement dit, si $x = a_k 2^k + \ldots + a_1 2^1 + a_0 2^0$ et si $z = \texttt{rot}(x, j)$, avec $j \leq k$, alors $z = a_{j-1}2^k + \ldots + a_0 2^{k-j+1} + a_k 2^{k-j} + \ldots + a_j 2^0$.

Question 3 Donner les écritures binaires des entiers $13, 23$ et 100. Calculer 13 & 23 et $\texttt{rot}(100, 4)$.

Question 4 Étant donné $x, k \in \mathbb{N}$ avec $k > 0$, prouver que x & $(2^k - 1)$ est le reste de la division de x par 2^k.

Question 5 On considère les deux fonctions de hachage suivantes, à valeurs dans $\{1, \ldots, 8\}$:

$$h_1(x) = [(x^2 \bmod 17) \ \& \ 7] + 1 \qquad h_2(x) = [(\texttt{rot}(x, 4) \bmod 33) \ \& \ 7] + 1$$

Effectuer le hachage coucou des clés 14, 100, 1000, 31, 117, dans cet ordre.
Rappel : $14^2 \bmod 17 = 9$, $100^2 \bmod 17 = 4$,$1000^2 \bmod 17 = 9$, $31^2 \bmod 17 = 9$, $117^2 \bmod 17 = 4$.

Exercice 3 *: Arbres quasi-équilibrés [6 points]*

On rappelle qu'un arbre binaire strict est soit réduit à une feuille (1 nœud externe), soit possède une racine (un nœud interne) et deux enfants qui sont eux-même des arbres binaires stricts. **Dans cet exercice, tous les arbres sont binaires stricts.** La *hauteur h d'un arbre T* est telle que : si T est une feuille alors $h(T) = 0$, et sinon $h(T) = 1 + \max\{h(T1), h(T2)\}$, où $T1$ et $T2$ sont les enfants de la racine de T.

Soit $\mathcal{P}_c$ la propriété suivante définie sur les arbres : *Un arbre T vérifie la propriété $\mathcal{P}_c$ si et seulement si il existe une constante c telle que pour tout nœud interne ν de T, les hauteurs des enfants de ν diffèrent au plus de c.*
Autrement dit, si $T1_\nu$ et $T2_\nu$ sont les enfants de ν, alors $|h(T1_\nu) - h(T2_\nu)| \leq c$.

Question 1 Caractériser la famille des arbres qui vérifient $\mathcal{P}_0$ et celle des arbres qui vérifient $\mathcal{P}_1$.

Question 2 Donner un exemple d'arbre vérifiant $\mathcal{P}_2$ mais pas $\mathcal{P}_1$ et un exemple d'arbre ne vérifiant pas $\mathcal{P}_2$.

Question 3 Montrer que pour tout $c \in \mathbb{N}\setminus\{0\}$, il existe $\alpha_0 \in]0, 1]$, tel que pour tout $\alpha \in]0, \alpha_0]$ on ait $\alpha(1+\alpha)^c \leq 1$.

Question 4 Soit $c \in \mathbb{N} \setminus \{0\}$, montrer qu'il existe α vérifiant $0 < \alpha \leq 1$ tel que pour tout arbre vérifiant $\mathcal{P}_c$, de taille n et de hauteur h, on ait $n \geq (1 + \alpha)^h - 1$.

Question 5 En déduire que tout arbre, qui vérifie la propriété $\mathcal{P}_c$, a une hauteur en $O(\log n)$.