ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Section d'Informatique et de Systèmes de Communication

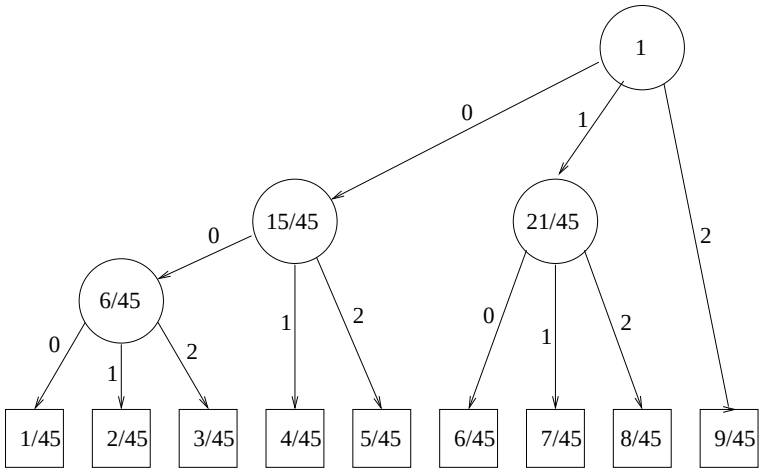
Corrigé de la série 8 16 Novembre 2009

1. Codes Huffman ternaires

a) La construction de codes de Huffman ternaires est similaire à celle de codes de Huffman binaires: on utilise la même procédure, sauf qu'on combine à chaque pas trois sous-arbres au lieu de deux.

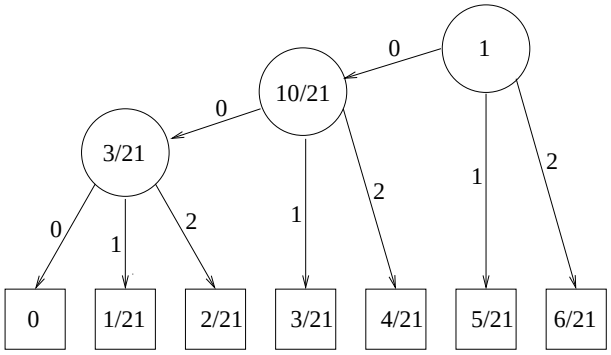
Cependant, il y a une subtilité à laquelle il faut faire attention: Il faut s'assurer qu'on puisse toujours combiner trois sous-arbres jusqu'à la fin (i.e., jusqu'à ce qu'il ne reste qu'un seul arbre). Il faut donc qu'il reste dans la dernière itération trois arbres et non deux. Pour s'en assurer de, il suffit d'ajouter une lettre de fréquence 0 à l'alphabet au début si le nombre de sommets est pair et d'ainsi s'assurer que l'alphabet est toujours de taille impaire: en combinant trois sous-arbres en un, on réduit à chaque pas le nombre d'objets à traiter par deux, et le fait que le nombre d'objets qu'on traite est impair est alors présérvé pendant que l'algorithme itère.

b) Le arbre suivant est l'arbre d'un code de Huffman C_1 ternaire avec les fréquences données.

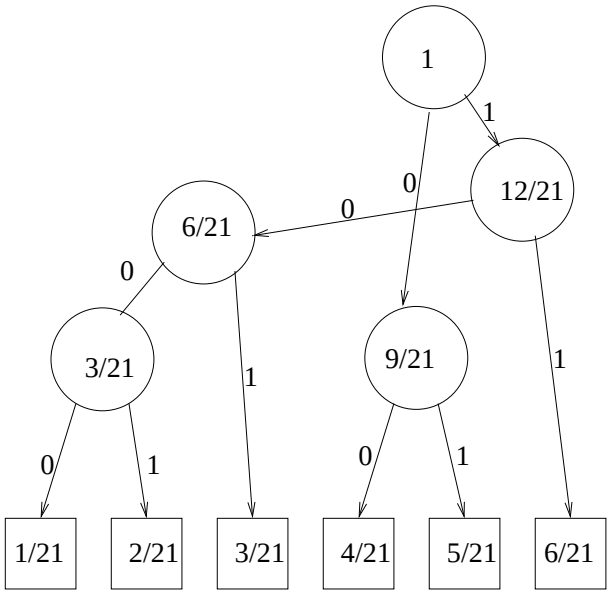


Les longueurs des mots du code sont $(3, 3, 3, 2, 2, 2, 2, 2, 1)$.

c) C'est un des cas pour lesquels il faut ajouter une lettre avec la fréquence 0 dans l'alphabet, comme mentionné au point a) ci-dessus. Appelons ce code C_2. Voici l'arbre:



Appelons C_3 le code de Huffman binaire pour les fréquences données. L'arbre correspondant:



d) On calcule la longueur moyenne de C_1, C_2, et C_3.

$$\begin{aligned}
\bar{L}_1 &= 1 \cdot \tfrac{9}{45} + 2 \cdot (\tfrac{8}{45} + \tfrac{7}{45} + \tfrac{6}{45} + \tfrac{5}{45} + \tfrac{4}{45}) + 3 \cdot (\tfrac{3}{45} + \tfrac{2}{45} + \tfrac{1}{45}) = \tfrac{29}{15} \\
\bar{L}_2 &= 1 \cdot (\tfrac{6}{21} + \tfrac{5}{21}) + 2 \cdot (\tfrac{4}{21} + \tfrac{3}{21}) + 3 \cdot (\tfrac{2}{21} + \tfrac{1}{21}) = \tfrac{34}{21} \\
\bar{L}_3 &= 2 \cdot (\tfrac{6}{21} + \tfrac{5}{21} + \tfrac{4}{21}) + 3 \cdot \tfrac{3}{21} + 4 \cdot (\tfrac{2}{21} + \tfrac{1}{21}) = \tfrac{51}{21}
\end{aligned}$$

2. Priority Queues

a) Ce sont les 15 arrangement suivants:

$$\begin{array}{lll}
(4,2,5,1,3), & (4,1,5,3,2), & (4,3,5,1,2), \\
(5,3,4,1,2), & (5,2,4,1,3), & (5,1,4,3,2), \\
(3,5,4,1,2), & (1,5,4,3,2), & (2,5,4,1,3), \\
(1,3,4,5,2), & (2,1,4,5,3), & (3,1,4,5,2), \\
(1,2,4,3,5), & (3,2,4,1,5), & (2,3,4,1,5)
\end{array}$$

b) L'idée est de mémoriser dans la queue de priorité les non-premiers et de ainsi simuler un crible d'Erastosthène. On peut alors successivement tester si un nombre donné est premier en testant s'il est le prochain élément dans la queue de priorité. Une première version de cet algorithme serait alors la suivante. (Dans l'algorithme suivant, H est une queue de priorité stockant des entiers, et tel que l'élément prioritaire $\top[H]$ est toujours l'élément le plus petit.

Input: Un entier N indiquant la fin de la queue de priorité
Output: Une suite croissante de tous les premiers entre 1 et N

```
H ← {4, 6, 8, ..., ⌊N/2⌋ · 2}
print 2
i ← 3
while i ≤ N do
  x ← ⊤[H]
  if x > i then
    for j = 2 · i, 3 · i, ..., ⌊N/i⌋ · i do
      H ← H ∪ {j}
    print i
  else
    while x = i do
      pop(H)
      x ← ⊤[H]
  i ← i + 1
```

Le désavantage de cette solution est qu'il est très couteux de stocker tous les multiples d'un premier dans la queue de priorité. Une petite refléxion montre qu'il suffit de stocker le prochain multiple d'un premier donné si l'on sait de quel premier c'est un multiple: à chaque fois qu'un tel nombre est enlevé de la queue de priorité, il suffit de rajouter le prochain multiple du même premier dans la queue. Pour pouvoir le faire, il faut stocker des couples (x, p), où p est le premier en question est x est son prochain multiple. L'algorithme suivant réalise cet approche avec deux modifications en plus: Les nombres pairs ne sont pas considérés, et les premiers $> \sqrt{N}$ ne sont pas inserés dans la queue puisque les nombres composés $\leq N$ ont toujours un facteur $\leq \sqrt{N}$.

Input: Un entier N indiquant la fin de la queue de priorité
Output: Une suite croissante de tous les premiers entre 1 et N

```
H ← {(9, 3)}
print 2
print 3
for i = 3, 5, 7, 9, ..., ⌊N/2⌋ · 2 do
  (x, p) ← ⊤[H]
  if x > i then
    print i
    if i ≤ √N then
      H ← H ∪ {(3 · i, i)}
  else
    while x = i do
      pop(H)
      H ← H ∪ {(i + 2p, p)}
```

$(x, p) \leftarrow \top[H]$

c) L'opération SiftUp réalise le cœur d'une insertion: il suffit d'ajouter l'élément à la fin et de faire un SiftUp par la suite:

Input: Un heap (a, N), où a est un tableau, et N le nombre d'éléments dans le heap. Un élément e à insérer.
Output: (a, N) modifiés.
$a[N] \leftarrow e$
$N \leftarrow N + 1$
SiftUp$(a, N - 1)$

De manière similaire, on peut aussi effacer un élément en utilisant les opérations SiftUp et SiftDown. Néanmoins, la procédure correcte est légèrement plus compliquée que pour une insertion.

Input: Un heap (a, N) et un indice k d'un élément à enlever.
Output: (a, N) modifiés.
$a[k] \leftarrow a[N - 1]$
$N \leftarrow N - 1$
if $k < N$ **then**
  if $a[\lfloor k/2 \rfloor] < a[k]$ **then**
    SiftUp(a, k)
  else
    SiftDown(a, k)