Contrôle Continu n°1

Analyse d'Algorithmes et Programmation

16 mars 2018

Durée : 2 heures

Les seuls documents autorisés sont les notes de cours.
L'utilisation d'un appareil électronique est proscrit pendant toute la durée de l'épreuve.
Le barème est indicatif.

Exercice 1 — Structures de données [7 points]

Dans cet exercice, les questions sont indépendantes.

Question 1.1 [1.5 points]
Réécrire les procédures ENFILER et DÉFILER pour détecter les débordements (normaux et négatifs) de file.

Solution:

Il suffit d'ajouter à ENFILER la condition suivante :

1: **if** F.queue + 1 == F.tête **or** (F.tête == 1 **and** F.queue == n) **then**
2: **error** "débordement"
3: **end if**

Pour DÉFILER c'est encore plus facile :

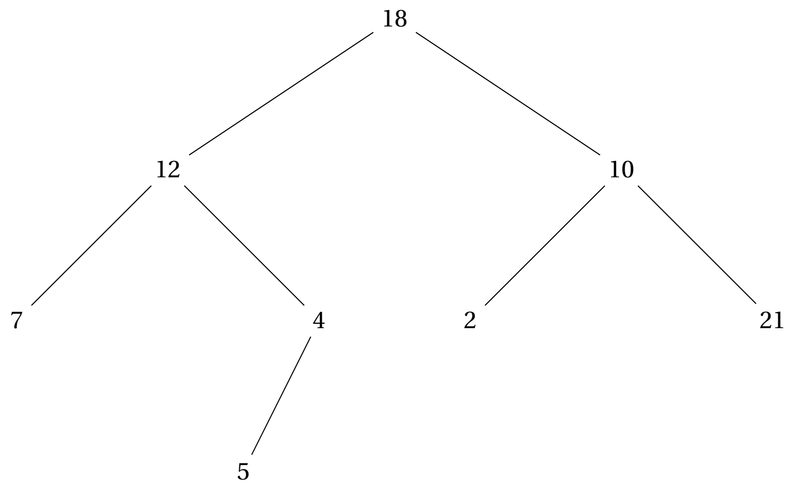
1: **if** F.tête == F.queue **then**
2: **error** "débordement négatif"
3: **end if**

Question 1.2 [1.5 points]
Dessiner l'arbre binaire dont l'indice de la racine est 6 et qui est représenté par les champs suivants.

indice	clé	gauche	droite
1	12	7	3
2	15	8	NIL
3	4	10	NIL
4	10	5	9
5	2	NIL	NIL
6	18	1	4
7	7	NIL	NIL
8	14	6	2
9	21	NIL	NIL
10	5	NIL	NIL

Solution:

Voici l'arbre construit :



Notons que les indices 2 et 8 n'apparaissent pas dans l'arbre.

Question 1.3 [1 point]

Écrire des versions récursives des procédures MINIMUM et MAXIMUM dans le cas d'un arbre binaire de recherche.

Solution:

MINIMUM(x)

if x.gauche == NIL **then**
    return x
end if
return MINIMUM(x.gauche)

MAXIMUM(x)

if x.droite == NIL **then**
    return x
end if
return MAXIMUM(x.droite)

Question 1.4 [1.5 points]

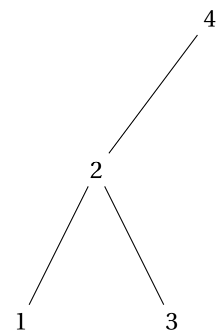
Votre camarade pense avoir découvert une remarquable propriété des arbres binaires de recherche. Supposez que la recherche d'une clé k dans un arbre binaire de recherche se termine sur une feuille. On considère trois ensembles :

- A, l'ensemble des clés situées à gauche du chemin de recherche;
- B, l'ensemble des clés situées sur le chemin de recherche;
- et C, l'ensemble des clés situées à droite du chemin de recherche.

Votre camarade affirme que, étant donnés trois clés $a \in A$, $b \in B$ et $c \in C$ quelconques, elles doivent satisfaire $a \leq b \leq c$. Donner un contre-exemple qui soit le plus petit possible.

Solution:

On recherche 1 dans l'arbre suivant :



On a alors $A = \emptyset$, $B = \{1, 2, 4\}$, $C = \{3\}$ et $3 < 4$.

Question 1.5 [1.5 points]

En vous basant sur la procédure ROTATION-GAUCHE (RG) vue dans le cadre des arbres rouge–noir, écrire le pseudo code de ROTATION-DROITE(T, y).

Solution:

```
ROTATION-DROITE(T, y)
x = y.gauche
y.gauche = x.droite
if x.droite ≠ NIL then
    x.droite.p = y
end if
x.p = y.p
if y.p == NIL then
    T.racine = x
else
    if y == y.p.gauche then
        y.p.gauche = x
    else
        y.p.droite = x
    end if
end if
x.droite = y
y.p = x
```

Exercice 2 — Validité du partitionnement de Hoare pour le tri rapide [7 points]

La version de PARTITION donnée dans le cours n'est pas l'algorithme de partitionnement originel. Voici la version d'origine, due à C.A.R. Hoare :

Algorithm 1 Algorithme de partitionnement de Hoare.

```
HOARE-PARTITION(tab, p, r)
 1: x = tab[p]
 2: i = p − 1
 3: j = r + 1
 4: while true do
 5:     repeat
 6:         j = j − 1
 7:     until tab[j] ≤ x
 8:     repeat
 9:         i = i + 1
10:     until tab[i] ≥ x
11:     if i < j then
12:         échanger tab[i] et tab[j]
13:     else
14:         return j
15:     end if
16: end while
```

Remarque : Une boucle **repeat** est similaire à une boucle **while**, à ceci près que l'on ne vérifie la condition qu'après la première occurence. Ainsi, la série d'instructions est exécutée au moins une fois, quelle que soit la condition.

Question 2.1 [1.5 points]
Montrer le bon fonctionnement de HOARE-PARTITION sur le tableau

$$\mathrm{tab} = [13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21]\,,$$

en donnant les valeurs du tableau et les valeurs auxiliaires (i et j) après chaque itération de la boucle **while** des lignes 4–16.

Solution:
Avant la première itération, x est initialisé à 13, $i = 0$ et $j = 13$.

$$\begin{aligned} j = 11, \quad i = 1, \quad \mathrm{tab} = [6, 19, 9, 5, 12, 8, 7, 4, 11, 2, 13, 21] \\ j = 10, \quad i = 2, \quad \mathrm{tab} = [6, 2, 9, 5, 12, 8, 7, 4, 11, 19, 13, 21] \\ j = 9, \quad i = 10, \quad \mathrm{tab} = [6, 2, 9, 5, 12, 8, 7, 4, 11, 19, 13, 21] \end{aligned}$$

À la fin, l'algorithme retourne $j = 9$.

Les trois questions suivantes vont vous donner l'occasion de démontrer très précisément que la procédure HOARE-PARTITION est correcte. En supposant que le sous-tableau $\mathrm{tab}[p, \dots, r]$ contienne au moins deux éléments, prouver les points suivants :

Question 2.2 [1 point]
Les indices i et j sont tels que l'on n'accède jamais à un élément de tab qui soit en dehors du sous-tableau $\mathrm{tab}[p, \dots, r]$.

Solution:

Les 2 indices i et j sont initialisés à $p-1$ et $r+1$ mais l'utilisation d'une boucle **repeat** fait que $i \geq p$ et $j \leq r$. En outre, ni j, ni i, ne peuvent "ressortir" du tableau puisque celui-ci contient au moins un élément plus petit (ou plus grand) que x : x lui-même.

Question 2.3 [1.5 points]

Quand HOARE-PARTITION se termine, elle retourne une valeur j telle que $p \leq j < r$.

Solution:

On a vu que j restait entre p et r tout au long de l'algorithme. Il reste à savoir pourquoi j est différent de r. Si le premier **repeat** s'arrête à $j = r$, alors il y aura une inversion $\text{tab}[p] \leftrightarrow \text{tab}[r]$ car i s'arrêtera à 1. En effet $\text{tab}[p] = x \geq x$. Puis il y aura une seconde itération de la boucle **while**, ce qui entrainera une décrémentation de j. Donc le j final vérifie $j < r$.

Question 2.4 [1.5 points]

Chaque élément de $\text{tab}[p,\ldots,j]$ est inférieur ou égal à chaque élément de $\text{tab}[j+1,\ldots,r]$ quand HOARE-PARTITION se termine.

Solution: (se montre facilement en utilisant un invariant de boucle.)

La procédure PARTITION vue en cours sépare le pivot (originellement en $\text{tab}[r]$) des deux partitions qu'elle crée. La procédure HOARE-PARTITION, en revanche, place toujours le pivot (originellement en $\text{tab}[p]$) dans l'une des deux partitions $\text{tab}[p,\ldots,j]$ et $\text{tab}[j+1,\ldots,r]$. Comme $p \leq j < r$, ce découpage n'est jamais trivial.

Question 2.5 [1.5 points]

Réécrire la procédure TRI-RAPIDE pour qu'elle utilise HOARE-PARTITION.

Solution:

```
TRI-RAPIDE(tab, p, r)
if p < r then
    j = HOARE-PARTITION(tab, p, r)
    TRI-RAPIDE(tab, p, j)
    TRI-RAPIDE(tab, j + 1, r)
end if
```

Exercice 3 — Conformité de la règle de Horner pour l'évaluation d'un polynôme [6 points]

Le code suivant implémente la règle de Horner relative à l'évaluation d'un polynôme

$$P(x) = \sum_{k=0}^{n} a_k x^k = a_0 + x\,(a_1 + x\,(a_2 + \cdots + x\,(a_{n-1} + x a_n)\cdots)),$$

étant donnés les coefficients $a_0, a_1, \ldots, a_n$ et une valeur de x :

1: $y = 0$
2: **for** $i = n$ **to** 0 **by** -1 **do**
3:     $y = a_i + x \cdot y$
4: **end for**

Question 3.1 [1 point]

Quel est le temps d'exécution asymptotique de ce fragment de code ? (On considère qu'une opération arithmétique se fait en temps constant.)

Solution:

Il y a n additions et n multiplications donc la complexité est en $\Theta(n)$.

Question 3.2 [2 points]

Écrire du pseudo code qui implémente l'algorithme naïf d'évaluation polynomiale, lequel calcule chaque terme du polynôme *ex nihilo*. Quel est le temps d'exécution de cet algorithme ? Quelles sont ses performances, comparées à celles de la règle de Horner ?

Solution:

$y = 0$
for $i = 0$ **to** n **do**
    $z = 1$
    for $j = 1$ **to** i **do**
        $z = z \cdot x$
    end for
    $y = y + a_i \cdot z$
end for

Question 3.3 [2 points]

Montrer que ce qui suit est un invariant de boucle :

Au début de chaque itération de la boucle **for** des lignes 2–4,

$$y = \sum_{k=0}^{n-i-1} a_{k+i+1} x^k.$$

On considérera qu'une sommation sans termes est égale à 0.

Solution:

Avant la première itération, on a $i = n$ et $y = 0$. Puis, au début de l'étape i, $y = \sum_{k=0}^{n-i-1} a_{k+i+1} x^k$, et on calcule

$$y = a_i + x \cdot \sum_{k=0}^{n-i-1} a_{k+i+1} x^k = a_i + \sum_{k=0}^{n-i-1} a_{k+i+1} x^{k+1} = a_i + \sum_{k=1}^{n-i} a_{k+i} x^k = \sum_{k=0}^{n-i} a_{k+i} x^k = \sum_{k=0}^{n-(i-1)-1} a_{k+(i-1)+1} x^k.$$

On retrouve bien la valeur de l'invariant au début de l'itération suivante de la boucle **while**.

Question 3.4 [1 point]

Conclure en démontrant que le fragment de code donné évalue correctement un polynôme caractérisé par les coefficients $a_0, a_1, \dots, a_n$.

Solution:

Grâce à l'invariant de boucle de la question précédente, on sait qu'au début de la dernière itération $y = \sum_{k=0}^{n-1} a_{k+1} x^k$. Alors le résultat retourné vérifie

$$y = a_0 + x \cdot \sum_{k=0}^{n-1} a_{k+1} x^k = a_0 + \sum_{k=0}^{n-1} a_{k+1} x^{k+1} = a_0 + \sum_{k=1}^{n} a_k x^k = \sum_{k=0}^{n} a_k x^k.$$