Correction

Examen d'algorithmique

EPITA ING1 2010 Rattrapage, A. DURET-LUTZ

Durée : 1 heure 30

Avril 2008

Consignes

- Tous les documents (sur papier) sont autorisés, livres inclus.
  Les calculatrices, téléphones, PSP et autres engins électroniques ne le sont pas.
- Répondez sur le sujet dans les cadres prévus à cet effet.
- Il y a 7 pages. Rappelez votre UID en haut de chaque feuille au cas où elles se mélangeraient.
- Ne donnez pas trop de détails. Lorsqu'on vous demande des algorithmes, on se moque des points-virgules de fin de commande etc. Écrivez simplement et lisiblement. Des spécifications claires et implémentables sont préférées à du code C ou C++ verbeux.
- Le barème est indicatif et correspond à une note sur 20.

File (2 points)

Dans cette question on considère une file de priorité S implémentée à l'aide d'un tas « max », c'est-à-dire avec la valeur maximale à la racine du tas.

1. S est initialement vide. Donnez l'état du tableau représentant S après y avoir effectué chacune des opérations suivantes :
   - `insert(5)`

5

   - `insert(2)`

5	2

   - `insert(7)`

7	2	5

   - `insert(6)`

7	6	5	2

   - `insert(4)`

7	6	5	2	4

   - `extract_max()`

6	4	5	2

   - `extract_max()`

5	4	2

   - `insert(6)`

6	5	2	4

- `extract_max()`

5	4	2

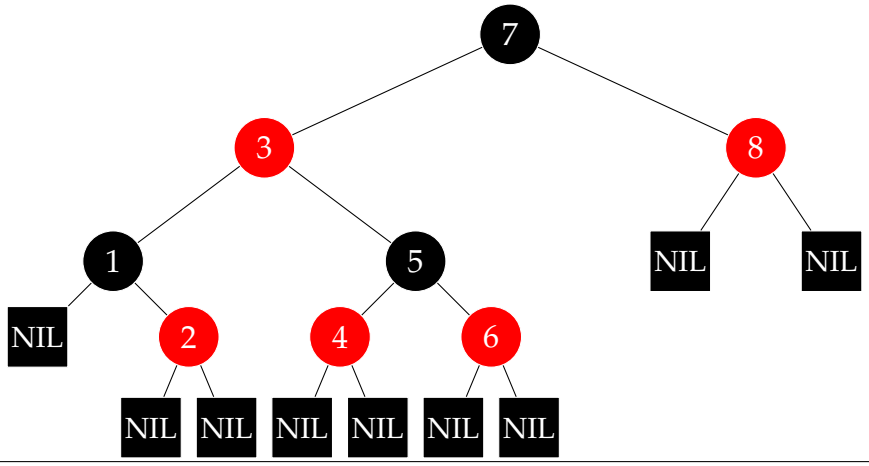
- `extract_max()`

4	2

Identification d'Arbres Rouge et Noir (3 points)

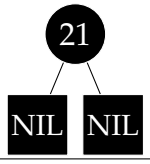
Pour chacun des arbres qui suivent, indiquez si ce sont des arbres binaires de recherche rouge et noir. Si l'arbre n'est pas un arbre rouge et noir, précisez pourquoi. (Les négations non justifiées seront ignorées.)
Pour décoder les photocopies en noir et blanc, voici du rouge et voici du noir.

1.



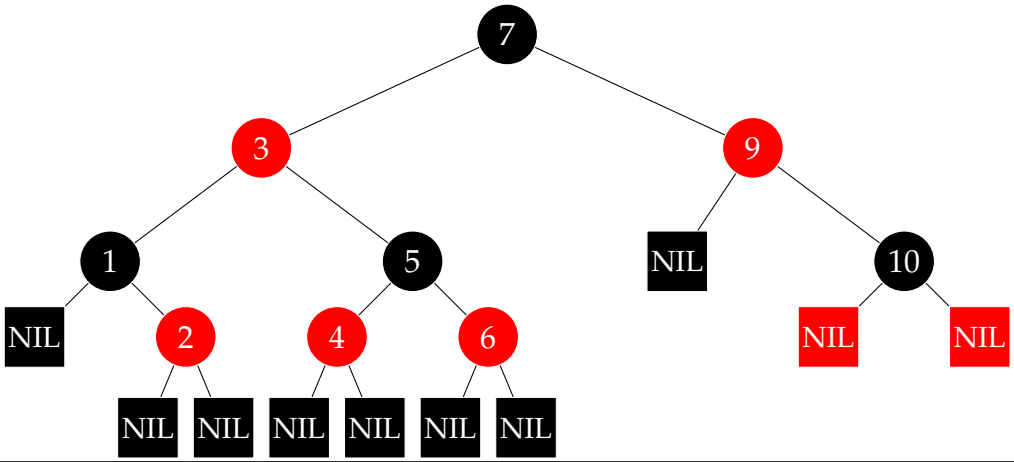
Réponse :
Pas un ARN car la hauteur noire n'est pas constante.

2.



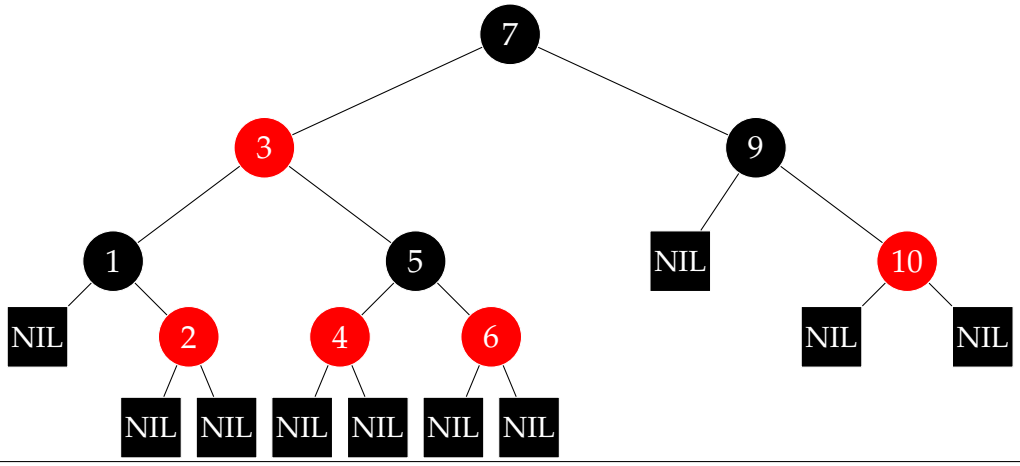
Réponse :
C'est un ARN.

3.



Réponse :
Pas un ARN car (1) la hauteur noire n'est pas constante, (2) il y a des feuilles rouges.

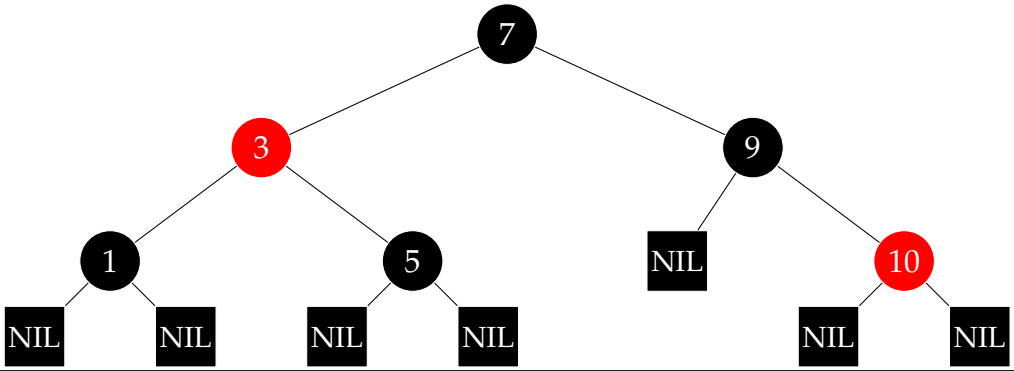
4.



Réponse :

C'est un ARN.

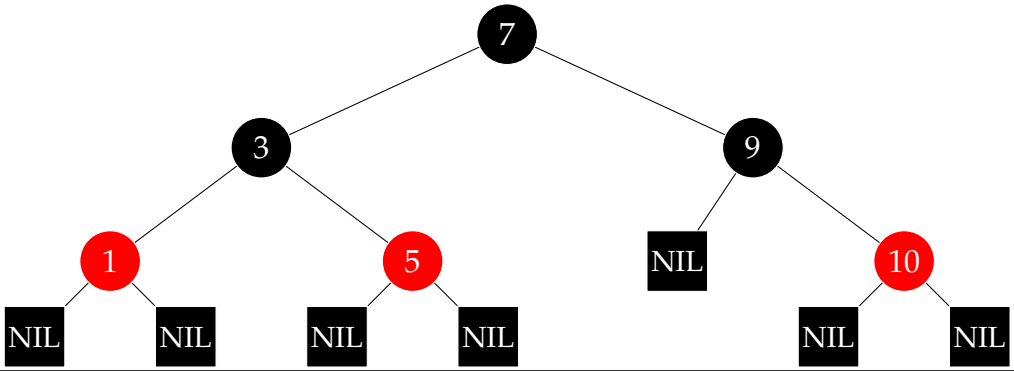
5.



Réponse :

C'est un ARN.

6.

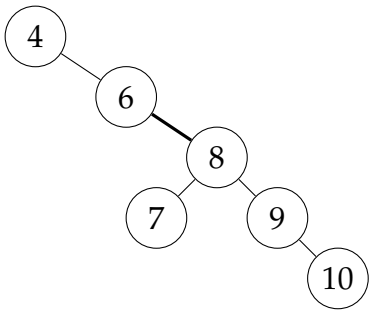


Réponse :

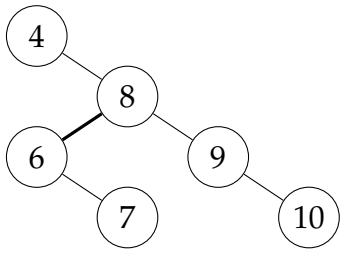
C'est un ARN.

Rotation d'arbre binaire de recherche (1 point)

Redessinez l'arbre binaire de recherche suivant après avoir effectué une rotation gauche de l'arc (6)—(8).



Réponse :



Complexités (3 points)

1. **(1 point)** Ai-je le droit d'écrire $\Theta(n) - \mathrm{O}(n) = \Theta(n)$? Justifiez votre réponse mathématiquement.

   Réponse :

   Non, c'est faux. Par exemple $\underbrace{n}_{\in\Theta(n)} - \underbrace{n}_{\in\mathrm{O}(n)} = \underbrace{0}_{\notin\Theta(n)}$.

2. **(2 point)** La complexité $T(n)$ d'un algorithme est constante pour une entrée de taille $n \leq 2$ et vérifie la relation $T(n) = 2T(n/3) + n\log n$ pour $n > 2$. Que pouvez-vous dire de $T(n)$?

   Réponse :

   On applique le théorème général.
   $a = 2$, $b = 3$. On doit comparer la fonction $f(n) = n\log n$ avec $n^{\log_b a} = n^{\log_3(2)}$. On a clairement $f(n) = \Omega(n) = \Omega(n^{\log_3(2)+\varepsilon})$ avec $\varepsilon = 1 - \log_3(2) > 0$.
   Il nous reste à vérifier qu'il existe une constante $c < 1$ telle que $af(n/b) \leq cf(n)$ pour des n suffisamment grands.
   C'est le cas si on prend $c = \frac{2}{3}$: on a bien $2\frac{n}{3}\log\frac{n}{3} \leq \frac{2}{3}n\log n$ car log est une fonction croissante.
   Le théorème général s'applique donc et on en déduit que $T(n) = \Theta(f(n)) = \Theta(n\log n)$.

Comb sort (5 points)

Le « *comb sort* » ou « tri à peigne » est une amélioration du tri à bulles qui prétend rivaliser en vitesse avec des tris plus sérieux comme le quicksort.

Le tri à bulles fonctionne en plusieurs passes qui parcourent le tableau entièrement de haut en bas : lors de chaque passe, chaque élément du tableau est comparé avec l'élément précédent, ces deux éléments sont éventuellement permutés pour faire remonter les petites valeurs (les bulles) et descendre les grosses. Ces passes se répètent jusqu'à ce qu'aucune permutation ne soit nécessaire. Le problème du tri à bulles et qu'il peut exister des « tortues » : c'est-à-dire des petites valeurs vers la fin du tableau qui vont demander beaucoup de passes pour atteindre leur place finale vers le début du tableau.

Le *comb sort* supprime les tortues en modifiant l'écart entre les éléments comparés. Dans le tri à bulles cet écart est toujours de 1 puisqu'un élément toujours est comparé avec le précédant. Le *comb sort*, en revanche, commence avec un écart aussi grand que la taille du tableau à trier, puis divise cet écart par 1,3 (en arrondissant à l'entier inférieur) après chaque passe. Les grands écarts du début permettent de faire vite remonter les tortues ; à la fin, quand l'écart est devenu 1, le comportement est similaire à un tri à bulles. Le *comb sort* se termine lorsqu'une passe avec un écart de 1 n'a fait aucune comparaison.

Voici une implémentation de l'algorithme, tel qu'il est présenté sur wikipedia :

```
function combsort11(array input)
    gap := input.size //initialize gap size

    loop until gap <= 1 and swaps = 0
        //update the gap value for a next comb
        if gap > 1
            gap := gap / 1.3
            if gap = 10 or gap = 9
                gap := 11
            end if
        end if

        i := 0
        swaps := 0

        //a single "comb" over the input list
        loop until i + gap >= array.size
            if array[i] > array[i+gap]
                swap(array[i], array[i+gap])
                swaps := 1
            end if
            i := i + 1
        end loop
    end loop
end function
```

1. **(4 points)** Donnez et justifiez la complexité temporelle de la fonction `combsort11` dans **le meilleur des cas**, en fonction de la taille n du tableau à trier. Prenez garde au fait que la boucle interne de l'algorithme effectue de plus en plus de comparaisons au fur et à mesure que la variable `gap` décroît.

Réponse :

La boucle principale ne peut s'arrêter qu'une fois que `gap=1`. Comme l'algorithme commence avec $gap = n$ et divise cette valeur par 1.3 à chaque itération, il effectue au moins $I = \lfloor \log_{1.3} n \rfloor = \left\lfloor \frac{\log n}{\log 1.3} \right\rfloor = \Theta(\log n)$ itérations. Dans le meilleur des cas le tableau sera déjà trié après ces I itérations et il ne sera pas nécessaire d'effectuer de passe supplémentaire.

À chaque itération, la boucle interne du tri effectue un nombre croissant de comparaisons des éléments du tableau. À la première itération on a $gap = n/1.3$ donc $n - \lfloor n/1.3 \rfloor$ comparaisons sont effectuées. À la second itération, $gap = n/1.3/1.3$ et $n - \lfloor n/(1.3)^2 \rfloor$ comparaisons sont effectuées, etc.

Dans le meilleur des cas, on effectue ainsi $\sum_{k=1}^{I}(n - \frac{n}{1.3^k})$ comparaisons.

$$\sum_{k=1}^{I}\left(n - \frac{n}{1.3^k}\right) = nI - \frac{n}{1.3}\sum_{k=0}^{I-1}\left(\frac{1}{1.3}\right)^k = nI - \frac{n}{1.3}\left(\frac{1 - \left(\frac{1}{1.3}\right)^I}{1 - \frac{1}{1.3}}\right)$$
$$= nI - n\Theta(1) = n(\Theta(\log n) - \Theta(1)) = \Theta(n \log n)$$

L'algorithme est donc de complexité $\Theta(n \log n)$.

(Pour faire écho à l'exercice précédent, notez que dans ces calculs j'ai le droit de simplifier $\Theta(\log n) - \Theta(1)$ en $\Theta(\log n)$ car toute constante est négligeable devant une fonction de $\Theta(\log n)$.)

2. **(1 point)** Cet algorithme de tri est-il stable ou instable ? Pourquoi ?

Réponse :

Cet algorithme est instable à cause de son utilisation de $gap > 1$. Par exemple si l'on considère une passe sur le tableau $4, 4, 4, 5, 2, 6$ avec $gap = 3$, seul le second 4 va être permuté avec 2 ruinant ainsi l'ordre original des 4.

Le pot (6 points)

L'organigrame de la société ProgDyn est un arbre dont le PDG est la racine. Chaque employé est représenté par un nœud dans cet arbre et, à l'exception du PDG, les employés possèdent tous un supérieur hiérarchique (leur père dans l'arbre) avec lequel ils s'entendent mal.

Pour fêter la découverte d'un nouvel algorithme, la société ProgDyn veut organiser un pot. Pour une meilleure ambiance, un employé ne devra pas y être invité avec son supérieur hiérarchique.

D'autre part chaque employé possède une « note de convivialité » (valeur positive ou nulle).

L'objectif est de trouver l'ensemble des employés à inviter de façon à maximiser la somme des notes de convivialité, sans inviter à la fois un employé et son supérieur.

Notations :
- On note $e(x)$ l'ensemble des employés dont le supérieur est x (les subordonnés immédiats).
- On note $n(x) \geq 0$ la note de convivialité d'un employé x.
- On note p le PDG.

On souhaite résoudre ce problème par programmation dynamique en considérant des sous-arbres de la hiérarchie de taille croissante.

Notons $A[x]$ la somme des notes de convivialité maximale que l'on peut atteindre en choisissant des invités uniquement parmi les subordonnés (immédiats ou non) de x, en invitant x.

Notons $S[x]$ la somme des notes de convivialité maximale que l'on peut atteindre en choisissant des invités uniquement parmi les subordonnés (immédiats ou non) de x, mais sans inviter x.

1. **(2 points)** Donnez des définitions récursives de $A[x]$ et $S[x]$.

Réponse :

$$A[x] = n(x) + \sum_{s \in e(x)} S[s]$$
$$S[x] = \sum_{s \in e(x)} \max(A[s], S[s])$$

2. **(2 points)** En fonction du nombre m d'employés, quelle serait la complexité du calcul de $A[p]$ par programmation dynamique ? (Justifiez votre réponse si vous n'avez pas répondu à la question précédente.)

Réponse :

$\Theta(m)$ car il suffit de remplir les deux tableaux A et S en partant des feuilles de l'arbre et en remontant, nœud après nœud, jusqu'à la racine p. Les calculs pour chaque nœud se font en temps constant et il y a m nœuds (où $m + 1$ si l'on considère que le PDG n'est pas un employé, mais le résultat reste le même).

3. **(2 points)** Une fois que l'on possède un algorithme de programmation dynamique pour calculer $A[p]$, comment le modifier pour obtenir l'ensemble des employés à inviter ?

Réponse :

Il faut garder trace des choix qui ont été faits pour chaque nœud de l'arbre. On se crée un tableau T dans lequel $T[x]$ devra recueillir l'ensemble des subordonnés immédiats de x qui ont été inclus lors du calcul de $S[x]$ (c'est-à-dire les s pour lesquels $S[s]$ était plus petit que $A[x]$). Ce tableau T doit être construit en même temps que S.

Pour construire l'ensemble des employés à inviter, il faut utiliser T récursivement à partir de p. Si une personne x est invitée, on sait qu'on maximisera la somme des notes de convivialité en invitant aussi les « sous-subordonnés » sélectionnés lors du calcul de $S[x]$, c'est-à-dire les employés de l'ensemble $\bigcup_{s \in e(x)} T[s]$. Ces employés étant maintenant invités, on invite récursivement leurs sous-subordonnés de la même façon, etc.

La base tordue (2 points)

Ne lisez et répondez à cette question, calculatoire, que si vous vous ennuyez ferme. Elle ne fait pas partie du barème.

On considère des nombres formés uniquement des chiffres 0 et 1, mais interprétés dans la base $-1+i$ (où i désigne malheureusement ce que vous croyez : l'unité imaginaire dont le carré vaut -1). Appelons cette base tordue la base T. Par exemple le nombre 110 en base T représente $1 \times (-1+i)^2 + 1 \times (-1+i)^1 + 0 \times (-1+i)^0 = -1-i$ en base 10. De même vous pourrez vérifier que le nombre $3+2i$ est représenté par 1001 en base T.

Cette base permet de représenter de façon unique tous les *nombres complexes entiers*, c'est-à-dire les nombres complexes dont les parties imaginaires et réelles sont entières. (Cette affirmation demanderait une preuve, vous devrez me croire sur parole.)

La question est la suivante : **quelle est la représentation de $-2-i$ en base T ?**

Indices :

- Si vous essayez tous les codages binaires dans l'ordre : 0, 1, 10, 11, 100, … cela vous coûtera plus de 200 essais avant de tomber sur la solution.
- Il est préférable d'appliquer un algorithme similaire à celui qu'on utilise habituellement pour afficher un nombre dans une base entière donnée.

Réponse :

$-2-i$ est représenté par 11101011 en base T.
La façon classique d'afficher un nombre N dans la base B est de diviser ce nombre par B jusqu'à obtenir 0 : les restes des divisions successives donnent les chiffres du résultat de droite à gauche.
Ici, pour convertir un nombre complexe $a+ib$ dans la base T, on souhaite diviser $a+ib$ par $-1+i$ de façon à ce qu'il nous reste 0 ou 1, et répéter le processus jusqu'à ce que $a+ib=0$. Autrement dit, on souhaite récrire $a+ib$ sous la forme

$$a+ib=(a'+ib')(-1+i)+r \tag{1}$$

où le reste r vaut 0 ou 1, et où a' et b' sont des entiers.
En sépartant les parties réelles et imaginaires de cette équation on trouve

$$a'=\frac{b-a+r}{2} \quad \text{et} \quad b'=\frac{-a-b+r}{2} \tag{2}$$

En en déduit que si a et b ont la même parité (sont tous les deux pairs ou impairs), alors il faut choisir $r=0$ pour que a' et b' soient entiers. Inversement, si a et b n'ont pas la même parité il faut choisir $r=1$ pour que a' et b' soient entiers.
L'algorithme de traduction devient donc le suivant :
- Tant que $a+ib\neq 0$ faire :
  - déterminer le reste r en fonction de la parité de a et b
  - empiler r
  - déterminer $a'+ib'$ en fonction de a, b et r, soit d'après l'équation (2) qui donne a et b séparément, soit en reprenant l'équation (1) qui nous donne le nouveau nombre complexe directement :
$$a'+ib'=\frac{a+ib-r}{-1+i}$$
  - préparer l'itération suivante avec $a\leftarrow a'$ et $b\leftarrow b'$
- dépiler tous les restes pour obtenir les chiffres de gauche à droite

Voici l'algorithme déroulé pour $-2-i$:

$-2-i$	reste 1	$\frac{(-2-i-1)(-1-i)}{(-1+i)(-1-i)}=\frac{2+4i}{2}=1+2i$
$1+2i$	reste 1	$\frac{(1+2i-1)(-1-i)}{(-1+i)(-1-i)}=\frac{2-2i}{2}=1-i$
$1-i$	reste 0	$\frac{(1-i)(-1-i)}{(-1+i)(-1-i)}=\frac{-2}{2}=-1$
-1	reste 1	$\frac{(-1-1)(-1-i)}{(-1+i)(-1-i)}=\frac{2+2i}{2}=1+i$
$1+i$	reste 0	$\frac{(1+i)(-1-i)}{(-1+i)(-1-i)}=\frac{-2i}{2}=-i$
$-i$	reste 1	$\frac{(-i-1)(-1-i)}{(-1+i)(-1-i)}=\frac{2i}{2}=i$
i	reste 1	$\frac{(i-1)(-1-i)}{(-1+i)(-1-i)}=\frac{2}{2}=1$
1	reste 1	$\frac{(1-1)(-1-i)}{(-1+i)(-1-i)}=\frac{0}{2}=0$ fin

Si le cœur vous en dit, consultez *Hacker's Delight*, Henry S. Warren, Jr. aux éditions Addison-Wesley pour d'autres bases tordues.