Nom :
Prénom :

Examen d'algorithmique

EPITA ING1 2012 S1 RATTRAPAGE; A. DURET-LUTZ

Durée : 1 heure 30

Rattrapage

Consignes

– Cet examen se déroule **sans document** et **sans calculatrice**.
– Répondez sur le sujet dans les cadres prévus à cet effet.
– Il y a cinq pages d'énoncé, et une page d'annexe.
Rappelez votre nom 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 26.

1 Notation Θ (3 points)

1. **(2 points)** Prouvez rigoureusement que pour trois constantes a, b, c strictements positives, on a $a\sqrt{bn+c} = \Theta(\sqrt{n})$

Réponse :

2. **(1 point)** Laquelle ou lesquelles des constantes a, b et c peuvent être prises nulles sans invalider l'égalité ci-dessus ?

Réponse :

2 Tableaux dynamiques (10 points)

On considère un tableau dynamique dans lequel les opérations suivantes sont définies :
– Access(A,i) retourne l'élément d'indice i du tableau A, en $\Theta(1)$ opérations.
– InsertBack(A,v) ajoute la valeur v en queue du tableau A, en agrandissant le tableau si nécessaire.
– DeleteBack(A) supprime la dernière valeur du tableau, dans un premier temps sans changer la taille du tableau.

2.1 Insertion en queue

Considérons l'algorithme suivant, pour l'insertion en queue du tableau dynamique :

Si le tableau n'est pas plein
  insérer l'élément dans la première cellule vide.
Sinon (si le tableau est plein)
  allouer un nouveau tableau deux fois plus grand,
  y copier tous les éléments du tableau original,
  libérer l'ancien tableau,
  puis enfin insérer l'élément dans la première cellule vide.

La taille du tableau ainsi que la position de la première cellule vide sont bien sûr stockées dans deux variables.

Nous avons vu en cours pourquoi doubler la taille du tableau dynamique lors des réallocations permettait que l'insertion en queue ait un coût *amorti* constant. Pour retrouver ce résultat, considérons une insertion dans un tableau de n éléments. La complexité de cette insertion est évidement en $\Theta(1)$ si le tableau n'est pas plein. Si le tableau est plein, le coût de copier les n éléments dans un nouveau tableau domine celui de l'ajout de l'élément lui-même, et la complexité totale de l'insertion est en $\Theta(n)$. Cependant, nous savons que la taille du tableau double à chaque fois, donc si une insertion dans un tableau de taille n a demandé une réallocation, alors les $n/2-1$ insertions qui l'ont précédée se sont faites en temps constant. Nous pouvons donc étaler le coût de la réallocation sur les $n/2$ dernières insertions (qui ont eu lieu après la réallocation précédente) pour obtenir un coût amorti de

$$\frac{(n/2-1)\Theta(1)+\Theta(n)}{n/2}=\frac{\Theta(n)}{\Theta(n)}=\Theta(1)$$

1. **(2 point)** Quelle serait la complexité amortie de l'insertion si au lieu de doubler la taille du tableau on multipliait la taille par $5/4$? Justifiez votre réponse.

Réponse :

2. **(4 points)** Quelle serait la complexité amortie de l'insertion si au lieu de doubler la taille du tableau on *augmentait* la taille du tableau de $\sqrt{n}$ cellules. (Par exemple si un tableau de taille 16 est plein, on réalloue un tableau de taille $16+\sqrt{16}=20$.) Justifiez votre réponse sans vous inquiéter des arrondis sur la racine carrée.
   À toutes fins utiles : si $x=y+\sqrt{y}$ alors $y=x-\frac{1}{2}\sqrt{4x+1}+\frac{1}{2}$.

Réponse :

2.2 Suppression en queue

Revenons au scénario dans lequel l'insertion dans un tableau plein double la taille du tableau. Considérons maintenant, DeleteBack(A), l'opération de suppression de l'élément en queue du tableau A.

Supposons qu'on cherche à économiser la mémoire et qu'on modifie cette fonction de façon à libérer de la mémoire de façon symétrique à l'insertion. C'est-à-dire que si un tableau de taille n ne contient que $n/2$ éléments, alors le tableau est réalloué et sa taille est divisée par deux.

1. **(2 points)** Proposez une séquence d'insertions et de suppressions dans laquelle les opérations demandent de nombreuses réallocations consécutives (i.e., le coût des réallocations n'est plus amorti).

Réponse :

2. **(2 point)** Comment proposez-vous de corriger *DeleteBack* pour que deux réallocations soient toujours séparées par un nombre suffisant d'opérations (insertion ou suppression) en $\Theta(1)$ de façon à amortir le coût des réallocations ?

Réponse :

3 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.

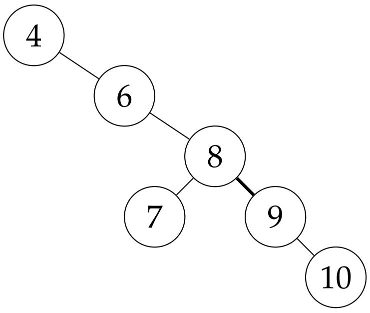
Réponse :

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

Réponse :

4 Rotation d'arbre binaire de recherche (2 point)

Redessinez l'arbre binaire de recherche suivant après avoir effectué une rotation gauche de l'arc 8—9.



Réponse :

5 Arbres quaternaires (6 points)

On considère ici des arbres quaternaires, c'est-à-dire des arbres dont chaque nœud possède soit 4 fils (on parle alors de nœud interne) soit aucun fils (on parle alors de feuille).

La *profondeur* d'un nœud dans un arbre est la longueur du chemin qui le relie à la racine. La racine est à la profondeur 0, ses fils à la profondeur 1, ses petits-fils à la profondeur 2, etc. La *hauteur* d'un arbre est la profondeur de sa feuille la plus profonde.

Soyez précis dans vos réponses aux questions qui suivent. En particulier, si vous utilisez une partie entière choisissez bien entre partie entière $\lceil$supérieure$\rceil$ ou $\lfloor$inférieure$\rfloor$.

1. **(1 point)** Quel est le nombre maximal de feuilles que peut posséder un arbre quaternaire de hauteur $h \geq 0$?

Réponse :

2. **(2 point)** Quel est le nombre maximal de nœuds que peut posséder un arbre quaternaire de hauteur $h \geq 0$?

Réponse :

3. **(2 point)** Quelle est la hauteur minimale que peut atteindre un arbre quaternaire de $n > 0$ nœuds ?

Réponse :

4. **(1 point)** Quelle est la hauteur maximale que peut atteindre d'un arbre quaternaire de $n > 0$ nœuds ?

Réponse :

Notations asymptotiques

$$\mathrm{O}(g(n)) = \{f(n) \mid \exists c \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq f(n) \leq cg(n)\}$$
$$\Omega(g(n)) = \{f(n) \mid \exists c \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq cg(n) \leq f(n)\}$$
$$\Theta(g(n)) = \{f(n) \mid \exists c_1 \in \mathbb{R}^{+\star}, \exists c_2 \in \mathbb{R}^{+\star}, \exists n_0 \in \mathbb{N}, \forall n \geq n_0, 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \iff g(n) \in \mathrm{O}(f(n))$ et $f(n) \notin \mathrm{O}(g(n))$ $\quad f(n) \in \mathrm{O}(g(n)) \iff g(n) \in \Omega(f(n))$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \iff f(n) \in \mathrm{O}(g(n))$ et $g(n) \notin \mathrm{O}(f(n))$ $\quad f(n) \in \Theta(g(n)) \iff \begin{cases} f(n) \in \Omega(g(n)) \\ g(n) \in \Omega(f(n)) \end{cases}$

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in \mathbb{R}^{+\star} \iff f(n) \in \Theta(g(n))$

Ordres de grandeurs

constante	$\Theta(1)$	
logarithmique	$\Theta(\log n)$	
polylogarith.	$\Theta((\log n)^c)$	$c > 1$
	$\Theta(\sqrt{n})$	
linéaire	$\Theta(n)$	
	$\Theta(n \log n)$	
quadratique	$\Theta(n^2)$	
	$\Theta(n^c)$	$c > 2$
exponentielle	$\Theta(c^n)$	$c > 1$
factorielle	$\Theta(n!)$	
	$\Theta(n^n)$	

Identités utiles

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2}$$
$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1} \qquad \text{si } x \neq 1$$
$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \qquad \text{si } |x| < 1$$
$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2} \qquad \text{si } |x| < 1$$
$$\sum_{k=1}^{n} \frac{1}{k} = \Theta(\log n)$$
$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

Définitions diverses

La complexité d'un problème est celle de l'algorithme le plus efficace pour le résoudre.

Un tri stable préserve l'ordre relatif de deux éléments égaux (au sens de la relation de comparaison utilisée pour le tri).

Un tri en place utilise une mémoire temporaire de taille constante (indépendante de n).

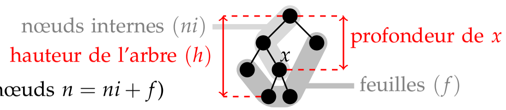
Théorème général

Soit à résoudre $T(n) = aT(n/b) + f(n)$ avec $a \geq 1$, $b > 1$

- Si $f(n) = \mathrm{O}(n^{\log_b a - \varepsilon})$ pour un $\varepsilon > 0$, alors $T(n) = \Theta(n^{\log_b a})$.
- Si $f(n) = \Theta(n^{\log_b a})$, alors $T(n) = \Theta(n^{\log_b a} \log n)$.
- Si $f(n) = \Omega(n^{\log_b a + \varepsilon})$ pour un $\varepsilon > 0$, et si $af(n/b) \leq cf(n)$ pour un $c < 1$ et tous les n suffisamment grands, alors $T(n) = \Theta(f(n))$.

(Note : il est possible de n'être dans aucun de ces trois cas.)

Arbres



(nœuds $n = ni + f$)

Pour tout arbre binaire :

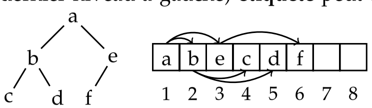
$n \leq 2^{h+1} - 1 \qquad h \geq \lceil \log_2(n+1) - 1 \rceil = \lfloor \log_2 n \rfloor$ si $n > 0$

$f \leq 2^h \qquad h \geq \lceil \log_2 f \rceil$ si $f > 0$

$f = ni + 1$ (si l'arbre est *complet* = les nœud internes ont tous 2 fils)

Dans un arbre binaire équilibré une feuille est soit à la profondeur $\lfloor \log_2(n+1) - 1 \rfloor$ soit à la profondeur $\lceil \log_2(n+1) - 1 \rceil = \lfloor \log_2 n \rfloor$. Pour ces arbres $h = \lfloor \log_2 n \rfloor$.

Un arbre parfait (= complet, équilibré, avec toutes les feuilles du dernier niveau à gauche) **étiqueté** peut être représenté par un tableau.



Les indices sont reliés par :

$\text{Père}(y) = \lfloor y/2 \rfloor$

$\text{FilsG}(y) = y \times 2$

$\text{FilsD}(y) = y \times 2 + 1$

Rappels de probabilités

Espérance d'une variable aléatoire X **:** C'est sa valeur attendue, ou moyenne. $\mathrm{E}[X] = \sum_x \Pr\{X = x\}$

Variance : $\mathrm{Var}[X] = \mathrm{E}[(X - \mathrm{E}[X])^2] = \mathrm{E}[X^2] - \mathrm{E}^2[X]$

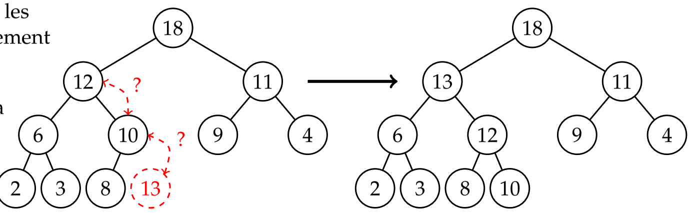
Loi binomiale : On lance n ballons dans r paniers. Les chutes dans les paniers sont équiprobables ($p = 1/r$). On note X_i le nombre de ballons dans le panier i. On a $\Pr\{X_i = k\} = C_n^k p^k (1-p)^{n-k}$. On peut montrer $E[X_i] = np$ et $\mathrm{Var}[X_i] = np(1-p)$.

Tas

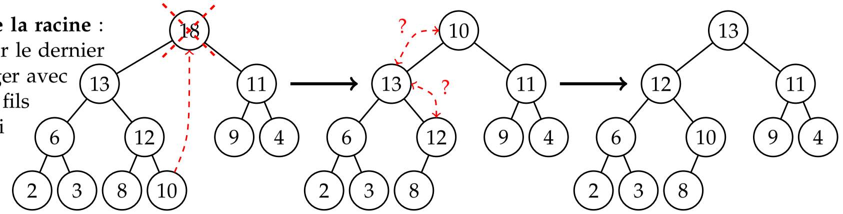
Un tas est un arbre parfait partiellement ordonné : l'étiquette d'un nœud est supérieure à celles de ses fils.

Dans les opérations qui suivent les arbres parfaits sont plus efficacement représentés par des tableaux.

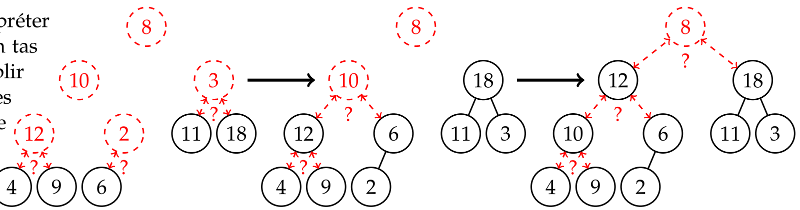
Insertion : ajouter l'élément à la fin du tas, l'échanger avec son père tant qu'il lui est inférieur (en remontant vers la racine). $T_{\text{insert}} = \mathrm{O}(\log n)$



Suppression de la racine : la remplacer par le dernier nœud ; l'échanger avec son plus grand fils tant que celui-ci est plus grand. $T_{\text{rem}} = \mathrm{O}(\log n)$



Construction : interpréter le tableau comme un tas (incorrect) puis rétablir l'ordre en partant des feuilles (vues comme des tas corrects). $T_{\text{build}} = \Theta(n)$

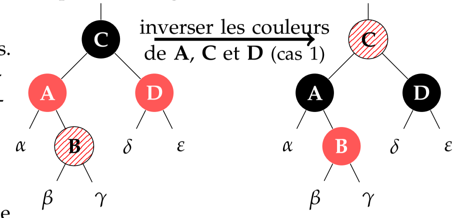


Arbres Rouge et Noir

Les ARN sont des arbres binaires de recherche dans lesquels : (1) un nœud est rouge ou noir, (2) racine et feuilles (NIL) sont noires, (3) les fils d'un nœud rouge sont noirs, et (4) tous les chemins reliant un nœud à une feuille (de ses descendants) contiennent le même nombre de nœuds noirs (= la *hauteur noire*). Ces propriétés interdisent un trop fort déséquilibre de l'arbre, sa hauteur reste en $\Theta(\log n)$.

Insertion d'une valeur : insérer le nœud avec la couleur rouge à la position qu'il aurait dans un arbre binaire de recherche classique, puis, si le père est rouge, considérer les trois cas suivants dans l'ordre.

Cas 1 : Le père et l'oncle du nœud considéré sont tous les deux rouges. **Répéter** cette transformation à partir du grand-père si l'arrière grand-père est aussi rouge.



(Dans tous ces cas, les lettres grecques représentent des sous-arbres avec la même hauteur noire.)

Cas 2 : Si le père est rouge, l'oncle noir, et que le nœud courant n'est pas dans l'axe père–grand-père, une rotation permet d'aligner fils, père, et grand-père.

Cas 3 : Si le père est rouge, l'oncle noir, et que le nœud courant est dans l'axe père–grand-père, un rotation et une inversion de couleurs rétablissent les propriétés des ARN.