Algorithmique - Semestre d'Automne 2010/2011

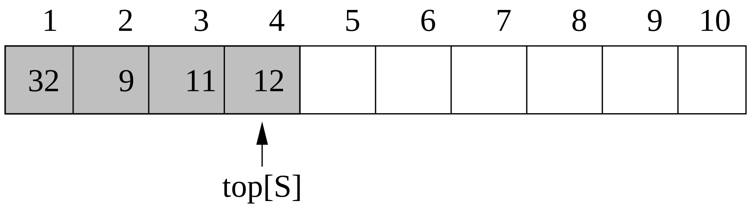
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Section d'Informatique et de Systèmes de Communication

Corrigé de la série 3 10 Octobre 2011

1. Stacks

a) Après ces opérations on a:



b) On voit qu'il y a une parenthèse fermante de trop, l'output attendu est donc "incorrect".

L'algorithme du cours ferait les opérations suivantes:

(1) Push([ )	
(2) Push(( )	
(3) Pop()	l'élément enlevé est "(", il correspond bien à ")" donc nous continuons
(4) Push(( )	
(5) Push(( )	
(6) Pop()	l'élément enlevé est "(", il correspond bien à ")" donc nous continuons
(7) Pop()	l'élément enlevé est "(", il correspond bien à ")" donc nous continuons
(8) Pop()	l'élément enlevé est "[" (le seul qui reste sur le stack), il ne correspond pas à ")" l'algorithme retourne donc "incorrect"

2. Stacks et files d'attente

a) Soient S_1 et S_2 deux stacks. Nous aimerions réaliser une file d'attente Q en utilisant S_1 et S_2.

Nous pouvons réaliser une telle file d'attente qui contient à $top[S_1]$ l'élément le plus récemment ajouté, et la file continue alors jusqu'à la fin de S_1, continue à la fin de S_2 jusqu'à $top[S_2]$ qui contient l'élément le plus ancien.

L'implémentation de `Enqueue`(x) est triviale:

```
Push(S_1, x)
```

L'idée de l'algorithme pour `Dequeue`(x) est d'enlever un élément de S_2. S'il n'y en a plus, on bouge le contenu de S_1 dans S_2 et on ressaye. Le voici:

```
r ← Pop(S_2)
if r = underflow then
    r ← Pop(S_1)
    while r ≠ underflow do
```

```
        Push(S_2, r)
        r ← Pop(S_1)
    r ← Pop(S_2)
return r
```

b) Avec les algorithmes comme présentés sous le point a) ci-dessus, chaque élément est enlevé au plus deux fois est inséré au plus deux fois d'un stack, ce qui fait que la propriété voulue est vérifiée.

3. Permutations avec stacks et files d'attente

a)
- $(1, 2, 3, 5, 4)$ est possible par la suite suivante d'instructions:
  `Push, Pop, Push, Pop, Push, Pop, Push, Push, Pop, Pop.`
- $(2, 3, 5, 4, 1)$ est possible par:
  `Push, Push, Pop, Push, Pop, Push, Push, Pop, Pop, Pop.`
- Mais il n'y a aucune suite d'instructions qui sort $(3, 1, 2, 5, 4)$.

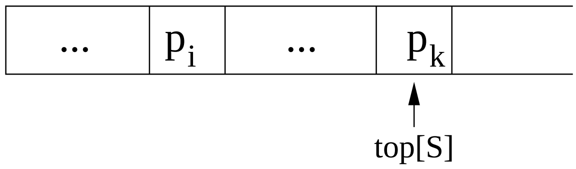
b) Nous devons d'abord fixer l'algorithme qui résout le problème et montrer ensuite qu'il fonctionne bien si la permutation vérifie la propriété exigée. Nous fixons la permutation $(p_1, \ldots, p_n)$ et considérons l'algorithme suivant:

```
1: j ← 1
2: i ← 1
3: for i = 1, ..., n do
4:   while j ≤ p_i do
5:     Push(S, j)
6:     j ← j + 1
7:   Pop(S)
```

L'ordre des push effectué de cet algorithme est bien $\texttt{Push}(S, 1), \texttt{Push}(S, 2), \ldots, \texttt{Push}(S, n)$ parce que j est incrémenté après chaque opération `Push`. Aussi le "while" de l'étape 4 se termine forcément parce que j est incrémenté en chaque itération et testé contre une valeur p_i fixe.

Le seul endroit possible d'erreur de l'algorithme est alors l'instruction `Pop` à l'étape 8. Montrons d'abord qu'il n'est pas possible que `Pop` résulte en un stack underflow: Supposons que nous nous trouvons à l'itération i juste avant le `Pop`. Alors il existe ℓ, $1 \leq \ell \leq i$, tel que $p_\ell \geq i$. Si ce n'était pas le cas, on avait que $\{p_1, \ldots p_i\} \subseteq \{1, \ldots, i-1\}$. Comme les p_k sont distincts, l'ensemble $\{p_1, \ldots, p_i\}$ est de taille i et ne peut donc pas être inclus dans $\{1, \ldots, i-1\}$ qui est de taille $i-1$.

La dernière possibilité d'erreur est finalement donc que `Pop` ne rend pas la valeur p_i attendue. Nous supposerons par la suite que l'itération i est la première itération d'échec de l'algorithme. Comme le "while" nous assure que p_i doit se trouver sur le stack, ceci n'arrive que s'il y a une autre valeur encore au dessus du stack, i.e. qu'on se trouve dans la situation suivante:



Nous avons $i < k$ puisque p_k se trouve encore sur le stack. D'autre part, comme les éléments sur le stack sont stockés de manière croissant, nous avons $p_i < p_k$. Une telle situation sur le stack ne peut arriver que si à une itération précédente $l < i$, la valeur p_l devait être sorti avec $p_l > p_k$. En résumé:

$$l < i < k,$$
$$p_i < p_k < p_l,$$

ce qui correspond à l'énoncé.

c) Pour n'importe quelle suite d'instructions, l'output sera toujours $(1, 2, 3, 4, 5)$. Parce que dans une file d'attente l'élément enlevé est l'élément le plus vieux. Alors, 1 est toujours le premier élément qui sort la file, 2 est toujours le deuxième élément qui sort la file, etc.

4. Arbres

a) On obtient les suites suivantes pour les différents parcours:

Parcours	Suite
Preorder	I, D, A, C, B, G, E, F, H, O, M, K, J, L, N
Inorder	A, B, C, D, E, F, G, H, I, J, K, L, M, N, O
Postorder	B, C, A, F, E, H, G, D, J, L, K, N, M, O, I

b) Il s'agit de voir que pour chaque sommet r la propriété $g < r < d$ est vérifié pour tout sommet g dans le sous-arbre gauche et d dans le sous-arbre droit. Ceci est équivalent à voir que la suite obtenue par le parcours inorder est croissante (cf. point suivant). Ceci est manifestement vrai.

c) " $\Longleftarrow$ ". Nous montrons qu'un arbre de recherche résulte en une suite croissante si parcouru inorder.

Nous prouvons ce résultat par induction (forte) sur les arbres à n sommets. L'arbre vide ($n = 0$), correspond à la suite vide, qui est clairement croissante. Soit maintenant T un arbre avec racine x, sous-arbre gauche L et sous-arbre droit R. La suite obtenue par parcours inorder est

$$[\text{suite du parcours de } L], x, [\text{suite du parcours de } R].$$

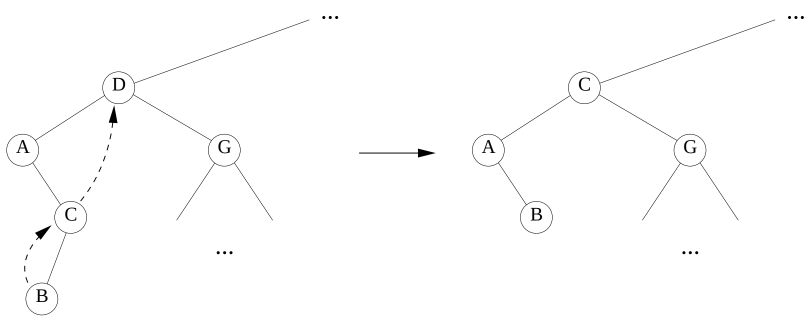
Par hypothèse d'induction, la partie L est croissante, et la partie R aussi. Il suffit donc de montrer que x est supérieur à son prédécesseur et inférieur à son succésseur. Comme T est un arbre de recherche, n'importe quel élément dans L est inférieur à x, donc aussi le dernier de la suite. De manière analogue, il n'y a pas non plus d'inversion entre x et son successeur.

" $\Longrightarrow$ ". Nous montrons que si le parcours inorder est croissant, alors l'arbre en question est un arbre de recherche. Soit x n'importe quel sommet dans l'arbre. Nous montrons que l'opération FIND(x) retourne bien le sommet x, ce qui permet de conclure.

Si x est racine de l'arbre de recherche, il sera clairement trouvé. Sinon, soit z n'importe quel sommet sur le chemin de la racine à x. Supposons sans perdre de généralité que le chemin descend dans le sous-arbre gauche de z, l'autre cas étant analogue. Alors x est parcouru avant z dans la traversée inorder, et donc $x < z$ par la croissance de la suite. Donc à l'étape où FIND se trouve en z, FIND descend aussi dans le sous-arbre gauche.

Comme cet argument s'applique à n'importe quel z sur le chemin vers x, FIND finit par trouver x.

d) On utilise l'algorithme du cours, i.e. on cherche dans le sous-arbre de gauche du sommet D le plus grand membre et on trouve C. Ce sommet ne peut pas avoir de sous-arbre droit (car ceci contredirait sa maximalité), ensuite, on remplace D par C et l'ancien sous-arbre gauche de C est mis à l'ancienne place de C. Schématiquement:



e) L'algorithme effectue plusieurs pas:

[1] Rechercher le sommet x a effacer
[2.0] S'il n'a pas de sous-arbre gauche, alors
  Effacer le sommet et mettre son sous-arbre droit a la place. stop.
[2.1] Trouver le plus grand element du sous-arbre gauche de x:
  $y \leftarrow$ `left`$[x]$
  Tant que y a un sous-arbre droite
    $y \leftarrow$ `right`$[y]$
[2.2] Enlever y de l'arbre et mettre son sous-arbre gauche a sa place
[2.3] Enlever x de l'arbre et mettre y a sa place.

L'algorithme fonctionne parce que le sommet y vérifie la propriété que tous les autres sommets du sous-arbre gauche sont plus petits; ceci garantit que la propriété de l'arbre de recherche est présérvée. (Voir aussi page 72 des notes de cours).

f) La preuve se fait par induction sur h. Écrivons N_h le nombre maximal de sommets que peut avoir un arbre. Un arbre binaire de hauteur h ne peut avoir qu'un seul sommet (la racine), donc $N_0 = 1 = 2^{0+1} - 1$.

Supposons maintenant l'affirmation prouvée pour h et montrons la pour $h+1$. Notons $\mathcal{T}_h$ l'ensemble d'arbres binaires de hauteur h. Alors comme tout arbre $T \in \mathcal{T}_{h+1}$ est formé d'une racine et de deux sous-arbres $L, R \in \mathcal{T}_h$ nous avons

$$\begin{aligned}
N_{h+1} &= \max_{T \in \mathcal{T}_{h+1}} \{\#(\text{sommets de } T)\} \\
&= \max_{L,R \in \mathcal{T}_h} \{1 + \#(\text{sommets de } L) + \#(\text{sommets de } R)\} \\
&= 1 + \max_{L \in \mathcal{T}_h} \{\#(\text{sommets de } L)\} + \max_{R \in \mathcal{T}_h} \{\#(\text{sommets de } R)\} \\
&= 1 + (2^{h+1} - 1) + (2^{h+1} - 1) \\
&= 2^{h+2} - 1,
\end{aligned}$$

ce qui termine la preuve.