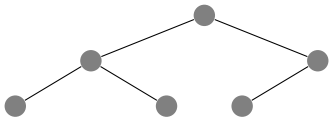
Informatique - MPI

Question 1.

□ **1.1.** Il n'existe qu'un seul arbre presque-parfait de taille 6.



□ **1.2.** La structure de tas binaire, qui permet la mise en œuvre de files de priorité, peut s'implémenter avec des arbres presque-parfaits.

□ **1.3.** L'énoncé définit un *arbre parfait* à partir d'un *arbre strict*. Désignons par d_1 cette première définition. On peut également proposer la définition inductive suivante, désignée par d_2 dans la suite.

Un arbre parfait de hauteur h est :

- *soit l'arbre parfait de hauteur 0 réduit à un seul nœud ;*
- *soit, pour $h > 0$, l'arbre parfait de hauteur h constitué d'un nœud et de deux arbres parfaits de hauteur $h-1$, enfants gauche et droit.*

Montrons par récurrence que d_1 implique d_2.

- Si $h = 0$, alors un arbre au sens de d_1 ne contient qu'un seul nœud. Il est parfait au sens de d_2.
- Suppsons $h > 0$ fixée, supposons la propriété établie jusqu'à la valeur h. Soit a un arbre parfait de hauteur $h+1$ au sens de d_1. Comme a est binaire strict, il possède deux enfants, chacun de hauteur h puisque toutes les feuilles sont à la même profondeur. Ces enfants sont donc des arbres parfaits de hauteur h, donc des arbres parfaits de hauteur h, au sens de d_2, par hypothèse de récurrence. On en conclut que a est bien un arbre parfait de hauteur $h+1$ au sens de d_2.

On conclut cette partie par récurrence.

Montrons à présent par induction que d_2 implique d_1.

- Comme précédemment, le cas de base est assuré.
- Soit a un arbre parfait de hauteur h au sens de d_2. Alors l'arbre (a, a) reste binaire strict et possède toutes ses feuilles à même profondeur. Il est donc parfait au sens de d_1.

Question 2.

□ **2.1.** Avec le type `arbre` fourni par l'énoncé, on peut proposer la fonction `int hauteur(arbre* a)` suivante.

```
1  int max(int a, int b) {
2    return (a < b) ? b : a;
3  }
4
5  int hauteur(arbre *a) {
6    if (a == NULL)
7      return -1;
8    int hg = hauteur(a->gauche);
9    int hd = hauteur(a->droite);
10   return 1 + max(hg, hd);
11 }
```

Pour un arbre a non vide de taille n, désignons par n_g (resp. n_d) le nombre de nœuds de son sous-arbre gauche (resp. droit). Si $C(n)$ désigne la complexité temporelle de la fonction `hauteur` qui reçoit un arbre de taille n, alors :

$$C_h(n) = C_h(n_g) + C_h(n_d) + \Theta(1)$$

Par récurrence, sachant $n = n_g + n_d + 1$, on établit $C_h(n) = \Theta(n)$.

□ **2.2.** La `fonction bool est_parfait(arbre* a)` peut être définie comme suit.

```
1  bool est_parfait(arbre *a) {
2    if (a == NULL)
3      return true;
4    int hg = hauteur(a->gauche);
5    int hd = hauteur(a->droite);
6    return hg == hd && est_parfait(a->gauche) && est_parfait(a->droite);
7  }
```

Désignons par $C_p(n)$ la complexité temporelle de la fonction `est_parfait` qui reçoit un arbre de taille n. Alors, en reprenant les notations de la questions précédentes

$$C_p(n) = C_p(n_g) + C_p(n_d) + O(n)$$

le dernier terme étant lié au coût des appels à la fonction `hauteur`. Pour un arbre parfait, cette relation prend la forme :

$$C_p(n) = 2 \times C_p\left(\frac{n}{2}\right) + O(n)$$

Il vient alors $C_p(n) = O(n \log n)$.

□ **2.3.** Pour écrire une fonction efficace, commençons par remarquer qu'un arbre presque-parfait de hauteur $h > 0$ a pour enfants gauche et droit :

- soit un arbre parfait de hauteur $h-1$ et un arbre presque-parfait de hauteur $h-1$;
- soit un arbre presque-parfait de hauteur $h-1$ et un arbre parfait de hauteur $h-2$.

Pour écrire la fonction demandée, on va d'abord écrire une fonction auxiliaire qui calcule, pour chaque nœud, un quadruplet contenant :

- la taille du sous-arbre ;
- la hauteur du sous-arbre ;
- un booléen qui détermine si le sous-arbre est parfait ;
- un booléen qui détermine si le sous-arbre est presque-parfait.

La connaissance de ces informations pour les deux enfants d'un nœud permet le calcul de celle de ce même nœud, en temps constant. Ce qui fournit une solution de complexité linéaire en la taille de l'arbre.

```
1  struct Quad{
2    int t;
3    int h;
4    bool parf;
5    bool pparf;
6  };
7  typedef struct Quad quad;
8
9  quad pgpp_aux(arbre* a, arbre** best, int* tbest) {
10   if (a == NULL) {
11     quad q = {.t = 0, .h = -1, .parf = true, .pparf = true};
12     return q;
13   }
14   else {
15     quad qg = pgpp_aux(a->gauche, best, tbest);
16     quad qd = pgpp_aux(a->droite, best, tbest);
17     quad q = {
18       .t = 1 + qg.t + qd.t,
19       .h = 1 + ((qg.h<qd.h)?qd.h:qg.h),
20       .parf = qg.h == qd.h && qg.parf && qd.parf,
21       .pparf = (qg.h == qd.h && qg.parf && qd.pparf) ||
22                (qg.h == qd.h + 1 && qg.pparf && qd.parf)
23     };
24     if (q.pparf && q.t > *tbest){
25       *best = a;
26       *tbest = q.t;
27     }
28     return q;
29   }
30 }
31
32 arbre* plus_grand_presque_parfait(arbre* a) {
33   arbre* best = NULL;
34   int tbest = 0;
35   pgpp_aux(a, &best, &tbest);
36   return best;
37 }
```