Université Paris 7 - Licence d'informatique - L2

EA4 – Corrigé de l'examen du 30 Mai 2012

Les réponses sont en *italique.*

Exercice 1 (4 points).

Soit la fonction :

```
int f(int n) {
  int m = abs(n); int res = 0;
  tantque  (m>1) faire
     si (m est pair) alors m:=m/2 sinon m:=m-8;
     res=res+1;
  ftq;
  retourne(res);
}
```

1. Quel est le temps de calcul de f relativement à n dans le meilleur cas ? (Donner la quantité exacte, pas le Grand-O).

   On a le meilleur cas quand m est toujours divisé par 2. Dans ce cas, la boucle est exécutée $\lceil \log_2 |n| \rceil$ fois. Le temps de calcul est donc $c\lceil \log_2 |n| \rceil + d$ où c est le coût d'une itération de la boucle et d celui des opérations hors de la boucle.

2. Quel est le temps de calcul de f relativement à n dans le pire des cas ? (Donner la quantité exacte, pas le Grand-O).

   On a le pire des cas quand on retranche toujours 8 de m. Dans ce cas, la boucle est exécutée $\lceil \frac{|n|}{8} \rceil$ fois. Le temps de calcul est donc $c\lceil \frac{|n|}{8} \rceil + d$ où c est le coût d'une itération de la boucle et d celui des opérations hors de la boucle.

3. Donner une valeur de n qui réalise le meilleur cas et une qui réalise le pire des cas.

   Les valeurs de n qui réalisent le meilleurs cas sont les puissances de 2, puisque dans ce cas m sera toujours pair. Par exemple : 128.

   Les valeurs de n qui réalisent le meilleurs cas sont les nombres impairs, puisque dans ce cas m sera toujours impair. Par exemple : 371.

4. Quelle est la probabilité d'avoir le meilleur cas ?

   Si `MaxInt` *est le plus grand* `int` *représentable en machine, il y a exactement* $\lceil \log_2(\texttt{MaxInt}) \rceil$ *nombres qui sont puissances de 2. La probabilité est donc* $\frac{\lceil \log_2(\texttt{MaxInt}) \rceil}{\texttt{MaxInt}}$.

5. Et celle d'avoir le pire des cas ?

   Un `int` *sur deux est impair. La probabilité est donc* $\frac{1}{2}$.

Exercice 2 (10 points).

Les fonctions demandées dans cet exercice prennent en paramètre un arbre binaire donné par l'adresse de sa racine, qui est un objet de type :

```
struct noeud {int val ; struct noeud * g ; struct noeud * d} ;
```

Elles doivent toutes avoir une complexité en temps en $O(N)$ où N est le nombre de noeuds de l'arbre. Vous expliquerez les raisonnements derrière vos algorithmes et vous justifierez leur complexité. Veuillez noter que des solutions élégantes et succinctes existent, qui comprennent environ dix lignes de pseudo-code pour les deux premières questions, et moins de vingt pour la troisième.

1. Une fonction booléenne qui retourne VRAI si et seulement dans l'arbre binaire donné tout noeud interne a exactement deux fils. (3 points)

   Un arbre vide satisfait la condition (il n'y a pas de noeud interne).
   Un arbre constitué seulement de sa racine satisfait la condition (il n'y a pas de noeud interne).
   Si seulement l'un de deux fils de la racine est `NULL`*, l'arbre ne satisfait pas la condition.*
   Dans tous les autres cas, on teste récursivement, si le sous-arbre gauche ET le sous-arbre droit satisfont la condition.

```
Algo EstBinaireStrict (A : Noeud *) : booléen {
    Si (A = NULL) alors retourner (VRAI);
    Si ((A -> fg = NULL) ET (A -> fd = NULL)) alors retourner (VRAI):
    Si ((A -> fg = NULL) OU (A -> fd = NULL)) alors retourner (FALSE);
    Retourner (EstBinaireStrict (A -> fg) ET EstBinaireStrict (A -> fd));
)
```

2. Une fonction qui, pour un entier h, retourne h si dans l'arbre binaire donné tout noeud interne a exactement deux fils et toutes les feuilles se trouvent à la même profondeur h. Retourne -2 sinon. (Pourquoi on ne fait pas retourner -1 ?) (3 points)

   On demandait simplement de tester si l'arbre binaire donné est complet et de retourner la hauteur de l'arbre dans ce cas (et -2 *sinon). On pouvait donc raisonner ainsi :*
   Un arbre vide est complet et de hauteur -1*. (C'est pourquoi on retourne* -2 *et non* -1 *en cas d'échec.)*
   Un arbre constitué seulement de sa racine est complet et de hauteur 0*.*
   Si seulement l'un de deux fils de la racine est `NULL`*, l'arbre n'est pas complet.*
   Sinon, si les deux fils de la racine ne sont pas `NULL`*, on teste récursivement si le sous-arbre gauche ET le sous-arbre droit sont complets. Si les deux le sont et s'ils ont la même hauteur* $hg = hd$ *(et* hg *n'est pas* -2*) on retourne* $hg + 1$*.*
   Dans tous les autres cas, on retourne -2*.*

```
Algo EstBinaireComplet (A : Noeud *) : int {
    Si (A = NULL) alors retourner (-1);
    Si ((A -> fg = NULL) ET (A -> fd = NULL)) alors retourner(0):
    Si ((A -> fg = NULL) OU (A -> fd = NULL)) alors retourner (-2);
    int hg := EstBinaireComplet (A -> fg);
    int hd := EstBinaireComplet (A -> fd);
    Si ((hg = hd) ET (hg != -2)) alors retourner (hg+1)
    Sinon retourner (-2)
}
```

   Pour ceux qui avaient plutôt compris que l'entier h était sensé faire partie des paramètres de la fonction, on pouvait appeler la fonction ci-dessus dans la suivante :

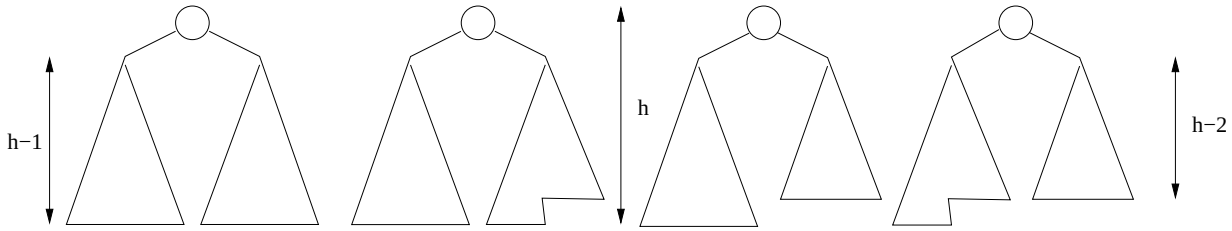
```
Algo EstBinaireCompletAux (A : Noeud *, h: int) : int {
    int k := EstBinaireComplet (A);
    Si (k = h) alors retourner (k)
    Sinon retourner (-2)
}
```

3. Une fonction qui, pour un entier h, retourne h si l'arbre binaire donné a la forme d'un tas de hauteur h (tous les niveaux sont "complets", sauf éventuellement le dernier de profondeur h, dans lequel les noeuds "occupés" sont ceux les plus à gauche). Retourne -2 sinon. (4 points)

Une solution récursive (voir plus loin pour la solution itérative) utilise des objets de type `couple`*, où un* `couple` *possède une première composante de type* `typeforme` *correspondant à l'ensemble de valeurs* {`Complet`, `Tas`, `Autre`} *et une seconde composante* `int` *correspondante à la hauteur de l'arbre. Elle est basée sur le principe suivant :*
Un arbre vide est complet (et, comme cas particulier, un tas) de hauteur -1*.*
Un arbre non vide est un tas de hauteur h *dans l'un des quatre cas suivants :*

- *le sous arbre gauche est* ***complet*** *de hauteur* $h-1$ *et le sous arbre droit est* ***complet*** *de hauteur* $h-1$ *(on a ainsi un arbre complet, qui est un cas particulier de tas) ;*
- *le sous arbre gauche est* ***complet*** *de hauteur* $h-1$ *et le sous arbre droit est un* ***tas*** *de hauteur* $h-1$ *(correspondant au cas où le dernier niveau est rempli pour plus de la moitié) ;*
- *le sous arbre gauche est* ***complet*** *de hauteur* $h-1$ *et le sous arbre droit est* ***complet*** *de hauteur* $h-2$ *(correspondant au cas où le dernier niveau est rempli exactement à moitié) ;*
- *le sous arbre gauche est un* ***tas*** *de hauteur* $h-1$ *et le sous arbre droit est* ***complet*** *de hauteur* $h-2$ *(correspondant au cas où le dernier niveau est rempli pour moins de la moitié) ;*

Les quatre cas sont illustrés dans la figure suivante :



D'où l'algorithme :

```
Algo Test (A : Noeud *): (typeforme, int) {
    Si (A = NULL) alors retourner  (Complet, -1);
    (formeG, hG)=Test(A -> fg);
    (formeD, hD)=Test(A -> fd);
    Si (hG = hD) alors {
        si ((formeG = formeD) ET (formeG = Complet) alors retourne (Complet, hG+1);
        si ((formeG = Complet) ET (formeD = Tas) alors retourne (Tas, hG+1);
    }
    Si (hG = hD+1) alors {
            Si  (((formeG = Tas) OU (formeG = Complet)) & (formeD = Complet))
            alors  retourne (Tas,hG+1)
    };
    retourne (Autre, max(hG, hD)+1)
}

Algo EstTas(A : Noeud *) : int {
    (forme, hauteur) := Test(A);
    Si ((forme = Complet) OU (forme = Tas)) retourne (hauteur))
    Sinon retourne (-2);
}
```

Une solution itérative, plus compliqué à écrire, est basée sur l'idée suivante.
On fait un parcours par niveaux (en largeur) de l'arbre à l'aide d'une ***file*** *(queue).*
Tant qu'on rencontre un noeud X *ayant deux fils, on ajoute les deux fils à la file et on sort* X *de la file. Quand on arrive au premier noeud* X *qui a moins que deux fils, on vérifie si* X *a seulement un fils droit (dans ce cas, on a échec et on retourne* -2*) ou s'il a seulement un*

fils gauche (dans ce cas, on ajoute ce fils unique à la file pour le traiter ensuite).
Enfin, il ne reste qu'à vérifier que tous les noeuds en attente de traitement dans la file sont tous des feuilles. Un compteur sert à compter les noeuds de l'arbre et à retourner la bonne hauteur, puisque dans un tas on a toujours $h = \lfloor \log_2 n \rfloor$.

```
Algo EstTasIter (A : Noeud *) : int {
    Si (A == NULL) alors retourner (-1);
    Si ( (A -> fg == NULL) ET (A -> fd == NULL) ) alors retourner(0):
    F : File de Noeud*;
    X : Noeud *;
    compt : int ;
    X := A;
    compt := 1;
    Tantque ((X -> fg != NULL) ET (X -> fd != NULL) Faire
        Enfiler (F, X -> fg);
        Enfiler (F, X -> fd);
        compt := comp +2;
        X := defiler (F);
    ftq;
    Si (X -> fd) != NULL) alors retourner (-2);
    Si (X -> fg) != NULL) alors {Enfiler (F, X -> fg); compt++}
    tantque ( ! EstVide(F) ) faire
        X := defiler (F);
        Si (! EstFeuille (X)) alors retourner (-2);
    ftq;
    retourner (log_2(compt))
}
```

Complexité. Tous les algorithmes récursifs sont en $O(N)$ puisque ils effectuent au plus un appel par noeud de l'arbre ***et puisque chaque appel a un coût constant****. L'algorithme itératif est en $O(N)$ puisque les deux boucles* `tantque` *effectuent au plus une itération pour chaque noeud de l'arbre et le coût de chaque itération est constant.*

Exercice 3 (6 points).

Proposer un algorithme pour trier des valeurs entières stockées dans un tableau qui utilise comme structure auxiliaire un ABR. Les fonctions de gestion des ABR ne devront pas être réécrites.

On commence par insérer un par un les entiers du tableau dans un ABR initialement vide. Ensuite on a deux options :
- *Soit on supprime successivement le min de l'ABR et on l'insère à sa bonne position dans le tableau en le remplissant de gauche à droite.*
- *Soit on effectue un parcours préfixe de l'ABR et on réinsère les valeurs dans le tableau dans l'ordre dans lequel on les rencontre pendant le parcours.*

La deuxième option est plus efficace (plus rapide) que la première, mais plus difficile à écrire. Toutefois, la complexité asymptotique totale (le grand-O) est la même dans les deux options.

Première option :

```
Algo TriParABR (T : int [ ]) : int [ ]{
    A : ABR;
```

```
    A:= NouveauABR();
    Pour i de 1 à taille(T) faire InsertionABR(A, T[i]);
    Pour i de 1 à taille(T) faire T[i] := SuppriMinABR(A);
    retourner (T);
}
```

Deuxième option :

```
Algo TriParABR (T : int [ ]) : int [ ] {
    A : ABR;
    A:= NouveauABR();
    Pour i de 1 à taille(T) faire InsertionABR (A, T[i]);
    ParcoursPrefixe (A, T, 1);
    retourner (T);
}

Algo ParcoursPrefixe(A : ABR, T : int [ ], i int) {
    Si A != Null alors {
         i := ParcoursPrefixe(A -> fg, T, i);
         T[i] := A -> val;
         i++;
         i := ParcoursPrefixe(A -> fd, T, i);
     }
}
```

1. Quel est le pire des cas de votre algorithme et quelle est sa complexité en temps ?
   Tout d'abord on rappelle que les opérations `InsertionABR` *et* `SuppriMinABR` *s'effectuent en temps $O(h)$ où h est la hauteur de l'arbre. Le parcours préfixe s'effectue en $O(n)$ où n est le nombre de noeuds de l'arbre.*
   On a le pire des cas quand on obtient un ABR dégénéré (chaque noeud a seulement un fils). A une constante multiplicative près, les n insertions de la première boucle `Pour` *coutent donc :*
   $$1 + 2 + 3 + \ldots + (n-1) + n = \frac{n(n+1)}{2} \in O(n^2)$$
   puisque la hauteur augmente d'une unité après chaque insertion.
   Ceci arrive ***en particulier*** *si T est déjà trié* ***soit en ordre croissant soit en ordre décroissant****. Dans le premier cas, chaque noeud a seulement un fils droit, dans le second, seulement un fils gauche.*
   Ensuite, si on choisit la première option, le coût de la deuxième boucle est aussi $O(n^2)$ (pour la même raison), pour un coût total en $O(n^2)$.
   Si on choisit la deuxième option, le coût de ParcoursPrefixe est en $O(n)$, mais ce gain est tout de même rendu négligeable par le coût en $O(n^2)$ de la boucle qui fait l'insertion.
2. Quelle est la complexité en temps du meilleur des cas de votre algorithme ?
   Dans le meilleur cas, l'ABR est équilibré et sa hauteur se mantient en $O(\log n)$. Le coût total sera donc en $O(n \log n)$ (pour les deux options).
3. Quelle complexité en moyenne vous attendez pour votre algorithme ? Pourquoi ?
   La complexité en moyenne attendue est $O(n \log n)$ car en moyenne une insertion coûte $O(\log n)$.
4. Que peut-on dire des complexités si on utilise un AVL à la place d'un ABR ?
   Les AVL sont des ABR équilibrés. Leur hauteur est toujours en $O(\log n)$. La complexité sera donc en $O(n \log n)$ dans tous les cas (pire, meilleur, moyen).