TD #11 — Arbres binaires (avec solutions)

Exercice 1. *Parcours en profondeur en C*

On considère le type :

```
struct btree {
    struct btree *left; struct btree *right; int data;
};
```

1. Écrivez une fonction C de signature `void dfsi(struct btree *t)` qui ne fait rien si son argument est un pointeur nul, et sinon effectue un parcours en profondeur infixe de l'arbre représenté par l'objet référencé par son argument afin d'afficher les contenus `.data` de ses nœuds.

   On propose :

   ```
   void dfsi(struct btree *t)
   {
       if (t == NULL) { return; }

       dfsi(t->left);
       printf(" %d ", t->data);
       dfsi(t->right);
   }
   ```

2. Donnez un extrait de code construisant un arbre pour lequel la fonction précédente affiche `1 2 3`, et effectue cet appel.

   On propose :

   ```
   struct btree t1 = { .left = NULL, .right = NULL, .data = 1};
   struct btree t3 = { .left = NULL, .right = NULL, .data = 3};
   struct btree t2 = { .left = &t1, .right = &t3, .data = 2};
   dfsi(&t2);
   ```

Exercice 2. *Relation feuille-nœud*

Montrez la *relation feuille-nœud* pour un arbre binaire, c'est à dire que le nombre de feuilles F et de nœuds internes N d'un arbre binaire satisfont l'inégalité $F \leq N + 1$, avec égalité ssi. l'arbre est strict (c'est à dire est tel que chacun de ses nœuds internes a exactement deux enfants).

On procède par récurrence forte sur la hauteur des arbres.
On pose comme hypothèse de récurrence $\mathcal{P}(h)$ que la propriété ci-dessus est vraie pour tous les arbres de hauteur $\leq h$.
Initialisation.

- Un arbre vide a $F = N = 0$ et est non strict.
- Le seul arbre binaire de hauteur 0 est strict, et possède une feuille et zéro nœud interne. On a bien $F = N + 1$.

Conservation. On considère un arbre t de hauteur $h + 1 > 0$ construit comme `N(_, t1, t2)`.
Par définition de la hauteur, les hauteurs de t_1 et t_2 sont $\leq h$. Soit N_1, F_1, N_2, F_2 les nombres de nœuds internes & feuilles de t_1 et t_2, et F et N ceux de t l'on a alors :

$F_1 \leq N_1 + 1$	Hyp. de récurrence forte
$F_2 \leq N_2 + 1$	Hyp. de récurrence forte
$F = F_1 + F_2$	Par construction
$N = N_1 + N_2 + 1$	Par construction
$F \leq N_1 + N_2 + 1 + 1$	Substitution
$\leq N + 1$	Substitution

De plus par hypothèse de récurrence forte les inégalités $F_1 \leq N_1 + 1$ & $F_2 \leq N_2 + 1$ sont des égalités ssi. t_1 et t_2 sont stricts, et t (de hauteur > 0) est strict ssi. t_1 et t_2 sont stricts (et en particulier non vides).
On a donc bien que :

— Si t est strict, $F = N + 1$.
— Sinon (sans perte de généralité) t_1 est non strict, d'où $F_1 < N_1 + 1$ et donc $F < N + 1$.

Exercice 3. *Numérotation par mots binaire*

Dans tout cet exercice, on utilisera le `type tree = E | N of tree * tree` pour représenter les arbres. On rappelle une numérotation par mots binaires des nœuds d'un arbre binaire :

— la racine (non vide) est numérotée par le mot vide ε (élément neutre pour la concaténation « $\cdot$ ») ;
— l'enfant gauche (resp. droit) non vide d'un nœud de numéro ν est numéroté $0 \cdot \nu$ (resp. $1 \cdot \nu$).

1. Soit `t1`, `t2` deux arbres binaires et `t = N(t1, t2)`, montrez que pour tout nœud non vide `t1'` de `t1` (resp. `t2'` de `t2`), soit ν_1 (resp. ν_2) son numéro dans `t1` pour la numérotation ci-dessus, alors son numéro dans `t` est $\nu_1 \cdot 0$ (resp. $\nu_2 \cdot 1$).

On montre cela pour `t1` par récurrence sur la profondeur des nœuds dans `t`.
Cas de base. Si `t1 = E` il n'y a rien à faire. Sinon `t1 = N(_, _)` et est de numéro ε dans `t1` et $0 = \varepsilon \cdot 0$ dans `t = N(t1, t2)`, et la propriété est vérifiée.
Conservation. On suppose la propriété vraie pour tous les nœuds non vides de `t1` de profondeur $\leq d$ dans `t`. Soit `k` un (éventuel) nœud non vide de profondeur $d + 1$, par définition de la profondeur c'est l'enfant (sans perte de généralité) droit d'un nœud `t1' = N(k, _)` de profondeur d. Alors soit $\nu_{1'}$, ν_k, $n_{1'}$, n_k les numérotations de `t1'` et `k` dans `t1` et `t`, l'on a :

$n_{1'} = \nu_{1'} \cdot 0$	Hyp. de récurrence
$\nu_k = 1 \cdot \nu_{1'}$	Définition de la numérotation
$n_k = 1 \cdot n_{1'}$	Définition de la numérotation
$n_k = \nu_k \cdot 0$	Substitution

Ce qui conclut la démonstration dans le cas de `t1`. Le cas de `t2` se traite symétriquement, en remplaçant 0 par 1.

2. Montrez que cette numérotation est injective, c'est à dire que tous les nœuds (non vides) d'un arbre ont un numéro distinct pour cette numérotation.

On montre cela à nouveau par récurrence, cette fois forte sur la hauteur des arbres.
Cas de base. C'est évident pour l'arbre vide, puisqu'il ne possède aucun nœud.
Conservation. On suppose la propriété vraie pour tous les arbres de hauteur $\leq h$. Soit `t = N(t1, t2)` un arbre de hauteur $h + 1$, par définition de la hauteur `t1` et `t2` sont tous deux de hauteur $\leq h$. Par hypothèse de récurrence forte, les nœuds de `t1` ont tous un numéro distinct, et il en va de même pour les nœuds de `t2`. Par la question précédente, soit ν_1 (resp. ν_2) le numéro d'un nœud de `t1` (resp. `t2`), le numéro du même nœud dans `t` est $\nu_1 \cdot 0$ (resp. $\nu_2 \cdot 1$). Il s'ensuit que les numéros des nœuds de `t1` (resp. `t2`) dans `t` sont tous distincts, et également distincts de ceux de `t2` (resp. `t1`) dans `t`, et également distincts de celui de la racine de `t` de numéro ε.

3. Proposez (informellement) un algorithme récursif efficace qui étant donnée (la racine d') un arbre binaire `t` et un numéro ν accède au nœud de numéro correspondant dans `t` (s'il existe).

Le cas de base de la récursion est quand $\nu = \varepsilon$: le nœud à accéder est la racine même de `t`. Sinon, l'on sait par les questions précédentes que si $\nu = \nu' \cdot 0$ (resp. $\nu = \nu' \cdot 1$), alors le nœud à accéder se trouve dans le sous-arbre de gauche (resp. droite) de `t`, et l'on peut résoudre le problème récursivement en effectuant un appel sur le sous-arbre approprié (si existant) avec ν'. Le nombre d'appels récursifs, et donc le coût de l'algorithme, est majoré par la longueur de ν, qui est aussi la profondeur du nœud à accéder ; c'est plutôt efficace (on ne peut pas faire mieux pour la représentation d'arbre utilisée ici).

Exercice 4. *Parcours en largeur*

Un *parcours en largeur* d'un arbre binaire est un algorithme qui visite tous les nœuds de l'arbre par niveau de profondeur croissant, c'est à dire qu'il visitera d'abord la racine (de profondeur 0), puis tous les enfants de la racine (de profondeur 1), puis tous les petits-enfants (de profondeur 2), etc.
On souhaite montrer dans cet exercice qu'un tel parcours peut être réalisé efficacement de la façon suivante : on initialise une *file* avec la racine de l'arbre, puis tant que la file est non vide on défile un élément, le visite, puis enfile ses (éventuels) enfants.

1. Implémentez l'algorithme esquissé ci-dessus en OCaml par une fonction :
   `bfs : ('a -> unit) -> 'a tree -> unit`
   pour le `type 'a tree = E | N of 'a * 'a tree * 'a tree`. Cette implémentation *devra* être écrite en style impératif et utiliser l'implémentation de file (impérative) fournie par le module `Queue`, notamment les fonctions :
   - `Queue.create : unit -> 'a queue`
   - `Queue.is_empty : 'a queue -> bool`
   - `Queue.push : 'a -> 'a queue -> unit`
   - `Queue.pop : 'a queue -> 'a`

   On propose :
```
1 let bfs f t =
2     let q = Queue.create () in
3     let () = Queue.push t q  in
4     while not (Queue.is_empty q) do
5         match Queue.pop q with
6         | E -> ()
7         | N (x, t1, t2) -> f x ; Queue.push t1 q ; Queue.push t2 q
8     done
```

2. Montrez que si la file contient uniquement des arbres vides `E`, la boucle `while` n'ajoute aucun nouvel élément à la file, et termine.

   Par la ligne 5 chaque itération défile un élément. Par hypothèse, cet élément est toujours un arbre vide, et donc par la ligne 6 l'itération n'enfile aucun élément. Le nombre d'éléments dans la file décroît donc strictement à chaque itération, et comme il est minoré par 0 c'est un variant de boucle, et celle-ci termine.

3. Montrez que la boucle `while` de votre fonction satisfait l'invariant de boucle suivant : la file est ou bien vide, ou bien contient uniquement des arbres vides, ou bien contient (en plus d'éventuels arbres vides) des nœuds de profondeurs toutes égales, ou bien des nœuds de profondeur d et $d+1$, où tous les nœuds de profondeur d ont été insérés avant ceux de profondeur $d+1$.

   Initialisation. Par les lignes 2 & 3, avant entrée dans la boucle la file contient uniquement `t`, qui est ou vide ou un nœud de profondeur zéro.
   Conservation. On suppose la propriété vraie au début d'une itération ; par la condition d'entrée dans la boucle, la file est également non vide. Par l'invariant, elle peut être des formes suivantes (représentées comme des listes, où l'on note les éléments par ordre d'insertion) :
   - `E::xs`. Dans ce cas par la ligne 6 et les spécifications d'une file sont état à la fin de l'itération est `xs`. Par hypothèse `xs` satisfait elle-même les propriétés de l'invariant, et celui-ci est donc conservé.
   - `N(_,t1,t2)::xs`. Soit d la profondeur de la tête de file, on a par hypothèse que les éventuels éléments non vides de `xs` sont tous de profondeur d, ou de profondeur (par ordre d'insertion) $d, \ldots, (d+1), \ldots$.
     Par la ligne 7 et les spécifications d'une file, son état à la fin de l'itération est `xs @ [t1] @ [t2]`, avec `t1` et `t2` ou bien vides ou bien de profondeur $d+1$, et la propriété est conservée.

4. Montrez que lors d'une itération de la boucle `while`, l'état successeur d'une file qui (en ignorant les arbres vides) contient uniquement des nœuds de profondeur d est une file contenant ou bien uniquement des arbres vide, ou bien contenant des nœuds de profondeur d et $d+1$, ou bien contenant des nœuds de profondeur $d+1$.

   La preuve est identique à l'analyse de la question précédente.

5. Montrez que si `t` est non vide, le premier élément à être défilé est le nœud de profondeur zéro (la racine) de `t`.

Par la ligne 3, la file contient uniquement `t` avant la première itération de la boucle. Par hypothèse `t` est non vide et par le type `'a tree` est aussi un nœud représentant la racine de l'arbre. Enfin `t` est défilé dès la première itération par la ligne 5.

6. Montrez que la suite des profondeurs des nœuds (on ignore donc les arbres vides) extraits de la file est croissante, et si non vide contient tous les entiers $\in [\![0, h]\!]$ pour un certain h.

Supposons que cette suite contienne un « trou » ; elle est donc sans perte de généralité de la forme $\ldots, d, d+2, \ldots$. En considérant les deux itérations successives produisant ces deux extractions, l'on a deux cas possibles :

— ou bien la file contient deux nœuds successifs de profondeur d et $d+2$ avant la première de ces itérations, mais c'est impossible pas l'invariant de la question 3 ;
— ou bien la file contient un unique nœud de profondeur d avant la première de ces itérations, puis un nœud de profondeur $d+2$ (qui doit alors nécessairement avoir été inséré à la ligne 7), mais c'est impossible par la question 4.

Supposons maintenant que cette suite ne soit pas croissante, alors elle est (sans perte de généralité) de la forme $\ldots, d, d-1, \ldots$. On conclut également que ce cas est impossible, par la même analyse que ci-dessus *mutatis mutandis*. Enfin, si non vide la profondeur du premier nœud est zéro par la question précédente, et l'arbre étant fini les profondeurs des nœuds le sont également.

7. Montrez que tout nœud est visité exactement une fois par ce parcours.

Par la ligne 7, la visite d'un nœud de profondeur d ne peut que causer l'ajout dans la file de nœuds de profondeur strictement supérieure (ou d'au plus deux arbres vides). Par la question précédente, un même nœud ne peut donc être visité qu'au plus une fois.
De cela, la question 2 et le fait que pour tout d le nombre de nœuds à profondeur d est fini on déduit que la boucle `while` termine.
Il suffit alors pour conclure de montrer que tout nœud a été inséré au moins une fois dans la file, ce que l'on fait par récurrence sur leur profondeur.
Initialisation. Si `t` est non vide, sa racine est insérée à la ligne 3.
Conservation. Par hypothèse de récurrence les nœuds de profondeur d ont été insérés dans la file. Par la ligne 5 et la finitude du nombre d'itérations de la boucle `while`, tous ces nœuds sont extraits de la file. Par définition, tout nœud de profondeur $d+1$ est enfant de son parent, nœud de profondeur d, et est donc inséré dans la file par la ligne 7 après extraction de ce dernier.

8. Conclure.

La question précédente montre que `bfs` termine et est un parcours, et la question 6 montre que celui-ci est en largeur.