TD #12 — Tas

Exercice 1. *Dessiner des tas*

Soit l'ensemble $\mathcal{S} = \{0, 1, 20, 54, 61, 87, 89, 97, 99\}$:

1. Dessinez (sous forme d'arbre) deux tas *min* binaires contenant les éléments de $\mathcal{S}$.
2. Donnez pour chacun sa représentation sous forme de tableau.
3. Dessinez un arbre binaire croissant contenant les éléments de $\mathcal{S}$ qui ne soit pas un tas binaire.
4. Dessinez un arbre binaire quasi-complet contenant les éléments de $\mathcal{S}$ qui ne soit pas un tas binaire.

Exercice 2. *Correction de* `_siftUp`

On rappelle les spécifications de `_siftUp` : soit `h` un arbre binaire quasi-complet représenté par un tableau, `i` un numéro de nœud valide et tel que `h` est un tas sauf éventuellement pour le fait que `i` peut être plus grand que son parent, on veut modifier `h` pour qu'il contienne les mêmes éléments qu'initialement (que le tableau soit une permutation de l'état initial) et soit un tas (max).

On propose l'implémentation OCaml suivante pour `_siftUp`, où le tas est représenté par le même `type 'a heap = { dat : 'a array ; gt : 'a -> 'a -> bool ; mutable cnt : int }` que dans le cours :

```
let _parent i = (i - 1)/2
let _swap d i j = let ti = d.(i) in d.(i) <- d.(j) ; d.(j) <- ti

let rec _siftUp ({dat ; gt} as h) (i:int) : unit =
    if i = 0 then () else
    let ip = _parent i in
    if gt dat.(i) dat.(ip)
        then ( _swap dat i ip ; _siftUp h ip )
```

1. Prouvez la correction de cette implémentation relativement à ses spécifications.

Exercice 3. *Analyse de* `heapify`

On utilise le même `type 'a heap` que dans le cours et l'exercice précédent, et l'on rapelle la fonction `heapify` vue en cours. Celle-ci prend en entrée une valeur `h:'a heap` telle que `h.dat` ne vérifie pas forcément la condition de tas pour `h.gt`, et elle modifie `h.dat` en place de façon à ce ce qu'il satisfasse cette condition.

```
let heapify (h:'a heap) : unit =
    let ni = h.cnt/2 - 1 in
    for i = ni downto 0 do _siftDown h i done
```

On utilise la spécification suivante de `_siftDown h i`, un peu plus forte que celle donnée en cours (mais également valide) : soit `h.dat` représentant un arbre sous forme de tableau tel que s'ils existent, les sous-arbres du nœud de numéro `i` respectent la condition de tas pour `h.gt`, alors `h.dat` est modifié pour que l'arbre de racine le nœud de numéro `i` contienne les mêmes éléments et soit de la même forme qu'auparavant et respecte la condition de tas.

1. Montrez que pour tout arbre binaire quasi-complet à N éléments, le numéro (pour la notation vue en cours) du premier nœud qui n'est pas une feuille est $\lfloor N/2 \rfloor - 1$.
2. Prouvez la correction de `heapify`.
3. Montrez que le coût pire-cas de `heapify` est un $O(n)$. On pourra utiliser le fait que le coût pire-cas de `_siftDown h i` est un $O(t_i)$, où t_i est la hauteur de l'arbre de racine le nœud de numéro `i`, et $\sum_{i=1}^{\infty} i/2^i = 2$ ceci peut se montrer de la façon suivante : $S := \sum_{i=1}^{\infty} i/2^i = \sum_{i=1}^{\infty} 1/2^i + \sum_{i=2}^{\infty} 1/2^i + \cdots = \sum_{j\geq 1} \sum_{i\geq j} 1/2^i$, et $\sum_{i\geq j} 1/2^i = \sum_{i\geq 0} 1/2^i - \sum_{i=0}^{j-1} 1/2^i = 2 - (2 - 2^{-j+1}) = 2^{-j+1}$, d'où $S = \sum_{j\geq 1} 2^{-j+1} = 2$, ou plus simplement : converge.

Exercice 4. *Numérotation dans un ABQC*

On rappelle qu'un arbre binaire de hauteur $h+1$ est dit *quasi-complet* si :

— ses feuilles sont toutes à profondeur h ou $h+1$;
— les feuilles de profondeur $h+1$ sont visitées avant celles de profondeur h dans un parcours en profondeur gauche (elles sont « les plus à gauche ») ;
— si un nœud de profondeur h a un unique enfant, celui-ci est nécessairement à gauche et est le dernier nœud de profondeur $h+1$ à être visité dans un parcours en profondeur gauche.

On définit inductivement la numérotation ν suivante pour les nœuds d'un arbre binaire (enraciné) :

— la racine a pour numéro 0 ;
— l'enfant gauche (resp. droit) du nœud de numéro x, s'il existe, a pour numéro $2x+1$ (resp. $2x+2$).

1. Montrez qu'un arbre binaire *complet* de hauteur h possède $N_h := 2^{h+1}-1$ nœuds de numéros $0,\dots,N_h-1$, dont 2^h feuilles de numéros $N_h-2^h,\dots,N_h-1$.

Dans les deux questions suivantes, on considère un arbre binaire quelconque.

2. Montrez qu'un parcours en largeur gauche visite les nœuds de l'arbre parcouru par numéro croissant.

3. **i.** Soit n_L, n_R nœuds *de même profondeur* appartenant respectivement au sous-arbre gauche et droit d'un même nœud, montrez que $\nu(n_L) < \nu(n_R)$.

ii. Réciproquement, soit n_L, n_R de même profondeur et t.q. $\nu(n_L) < \nu(n_R)$, montrez que n_L (resp. n_R) appartient au sous-arbre de gauche (resp. de droite) de leur *plus proche ancêtre commun*.

C'est à dire du nœud n_A le plus profond tel que n_L et n_R sont tous deux éléments de sous-arbres de n_A.

Pour conclure :

4. Montrez qu'un arbre binaire est quasi-complet ssi. ses nœuds ont une numérotation consécutive (« sans trous ») par ν.

Exercice 5. *Conversion de représentation & vérification de la propriété de tas*

On se dote du même type `'a heap` que dans l'exercice précédent.

1. Rappelez les définitions des fonctions `_left_kid`, `_right_kid` et `_parent` qui étant donné comme argument le numéro `i` du nœud d'un arbre binaire, calculent respectivement les numéros des éventuels enfants gauche, enfant droit et parent de ce nœud.

2. Écrivez une fonction :

```
is_heap : 'a heap -> bool
```

telle que `is_heap h` s'évalue à `true` si `h` représente un tas (autrement dit, si `h.dat` représente un arbre binaire quasi-complet qui vérifie la condition de tas pour `h.gt`), et `false` sinon.

Il est en principe envisageable de représenter l'arbre binaire sous-jacent d'un tas de façon « usuelle ». C'est par exemple ce que fait le `type 'a heapt` ci-dessous :

```
type 'a tree = E | N of 'a * 'a tree * 'a tree
type 'a heapt = { datt : 'a tree ; gt : 'a -> 'a -> bool }
```

3. Écrivez une fonction :

```
is_heapt : 'a heapt -> bool
```

telle que `is_heapt h` s'évalue à `true` si `h` (dont l'arbre est maintenant représenté comme un `'a tree`) représente un tas, et `false` sinon. (Il peut être utile de d'abord écrire une fonction `is_qc` qui teste si son argument est un arbre binaire quasi-complet. Essayez d'utiliser pour cela une définition inductive des arbres quasi-complets *d'une certaine hauteur*.)

4. Écrivez une fonction

```
heap2heapt : 'a heap -> 'a heapt
```

telle que `heap2heapt h` échoue sur une erreur si `h` n'est pas un tas valide, et sinon construit et renvoie une nouvelle représentation de `h` utilisant le type `'a heapt`.

5. Écrivez de même une fonction

```
heapt2heap : 'a heapt -> 'a heap
```

telle que `heapt2heap h` échoue sur une erreur si `h` n'est pas un tas valide, et sinon construit et renvoie une nouvelle représentation de `h` utilisant le type `'a heap`. On s'efforcera pour cette nouvelle représentation d'utiliser un tableau « pas trop grand ».