TD #12 — Tas (avec solutions)

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 :

```
1 let _parent i = (i - 1)/2
2 let _swap d i j = let ti = d.(i) in d.(i) <- d.(j) ; d.(j) <- ti
3
4 let rec _siftUp ({dat ; gt} as h) (i:int) : unit =
5     if i = 0 then () else
6     let ip = _parent i in
7     if gt dat.(i) dat.(ip)
8         then ( _swap dat i ip ; _siftUp h ip )
```

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

La terminaison peut par exemple se prouver en utilisant `i` comme variant récursif : il est minoré par 0 par la ligne 5 et le fait que c'est un indice valide de `dat`, et strictement décroissant à chaque appel récursif puisqu'un tel appel est conditionné à `i > 0`.
On prouve ensuite que `_siftUp` ne fait que des accès valides au tableau `dat` et qu'à l'issue de son exécution son contenu est une permutation du contenu initial. On procède par récurrence (plus ou moins explicite, par exemple sur la profondeur des appels).
Les seuls accès à `dat` marginalement effectués par `_siftUp` (hors appel récursif) le sont aux indices `i` et `ip` aux lignes 7 & 8 (par l'intermédiaire de l'appel à `_swap dat i ip` pour cette dernière). Par hypothèse `i` est un indice valide, et par calcul `ip` est égal à `(i-1)/2` $\in [\![0, i]\!]$, et est donc également valide. Par hypothèse de récurrence, l'appel récursif à `_siftUp` ne fait lui-même que des accès valides, ce qui permet de conclure.
De même, les seules modifications marginales de `dat` sont faites par le même appel `_swap dat i ip`. En exécutant le code de `_swap` pas à pas, on constate qu'à l'issue de son appel le contenu de `dat` est identique à son état initial, si ce n'est que les éléments d'indices `i` et `ip` (indices éventuellement identiques) ont été échangés : c'est donc une permutation de l'état initial. Par hypothèse de récurrence, le contenu de `dat` après appel récursif à `_siftUp` est aussi une permutation de son état initial, ce qui permet de conclure.
On prouve le reste par récurrence sur la profondeur du nœud d'indice `i`.
Cas de base : `i` est de profondeur zéro, ce qui coïncide avec `i = 0`. Dans ce cas par les hypothèses sur `h` celui-ci est déjà un tas et il n'y a rien à faire, ce que fait correctement la ligne 5.
Conservation : on suppose `_siftUP` correcte pour `i` de profondeur d, et l'on considère un appel avec `i` de profondeur $d+1$. Si le test à la ligne 7 échoue, alors `h` est un tas et il n'y a rien à faire, ce qui est correctement fait par le `else ()` implicite. Sinon par l'hypothèse initiale de l'appel à `_siftUp`, la seule valeur du sous-arbre de racine `ip` pouvant être plus grande (pour `gt`) que `dat.(ip)` était `dat.(i)`. Il s'ensuit qu'après appel à `_swap`, le sous-arbre de racine `i` est un tas, et `h` tout entier est donc un tas sauf éventuellement pour le fait que la (nouvelle) valeur du nœud d'indice `ip` peut être plus grand que celle de son parent. Donc `h` satisfait les conditions d'appel à `_siftUp h ip`. Puisque `ip` est de profondeur d, cet appel est correct par hypothèse de récurrence, ce qui permet de conclure.

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 $\lfloor N/2 \rfloor - 1$.

Par définition, un arbre binaire de N nœuds est quasi-complet ssi. ses nœuds sont de numéros $0, \dots, N-1$.
Si N est pair, on pose $x = N/2-1$. L'arbre étant quasi-complet, un nœud est présent au numéro $2x+1 = N-1$ et x n'est pas une feuille ; inversement, pour tout nœud $y = x+i$ avec $i > 0$, aucun nœud n'est présent au numéro $2y+1 = N-1+2i$.
Si N est impair, on pose $N = 2d+1$ et $x = d-1$. Par les mêmes arguments que ci-dessus, un nœud est présent au numéro $2x+2 = 2d = N-1$; inversement, pour tout nœud $y = x+i$ avec $i > 0$, aucun nœud n'est présent au numéro $2y+1 = 2d-2+2i+1 = N+2(i-1)$.

2. Prouvez la correction de `heapify`.

On utilise l'invariant suivant : au début (resp. à la fin) de chaque itération de la boucle de la ligne 3, les arbres de racine $i+1, \dots, n-1$ (resp. $i, \dots, n-1$) respectent tous la condition de tas (et à la fin de l'itération, contiennent les mêmes éléments qu'au début de l'itération). Cet invariant est vrai avant la première entrée dans la boucle, car les arbres de racines plus grandes que `n/2 - 1` sont tous de profondeur 0 et donc des tas. Soit une itération de la boucle, l'appel à `_siftDown h i` vérifie les conditions d'entrée de cette fonction puisque par l'invariant d'entrée de boucle les sous-arbres du nœud de numéro `i` ont des numéros tous strictement supérieurs à `i` et sont des tas ; on a donc en sortie que l'arbre de racine `i` est un tas de mêmes éléments qu'auparavant, et l'invariant de sortie de boucle est vérifié ; enfin la décrémentation de `i` n'invalide pas l'invariant d'entrée de boucle à l'itération suivante. La correction totale suit maintenant simplement de l'invariant à la fin de la dernière itération pour $i = 0$, et le fait que tous les appels à `_siftDown` terminent et qu'il en va donc de même pour `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 : $S := \sum_{i=1}^{\infty} i/2^i = \sum_{i=1}^{\infty} 1/2^i + \sum_{i=2}^{\infty} 1/2^i + \dots = \sum_{j\geq 1} \sum_{i\geq j} 1/2^i$, 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.

On utilise le fait qu'un arbre quasi-complet à N éléments a au plus 2^d nœuds de profondeur d avec $d \leq d_{max} := \lfloor \log(N) \rfloor$ et donc au plus 2^d nœuds racines d'arbres de hauteur $d_{max} - d$. Soit c_h le coût pire cas de `_siftDown` pour un nœud racine d'arbre de hauteur h, on a que $c_h = O(h)$. Le coût de `heapify` est alors majoré par :

$$\sum_{d=0}^{d_{max}} 2^d \times c_{d_{max}-d} = \sum_{d=0}^{d_{max}} 2^{d_{max}-d} \times c_d = 2^{d_{max}} \sum_{d=0}^{d_{max}} \frac{c_d}{2^d} = 2^{d_{max}} O\left(\sum_{d=0}^{d_{max}} \frac{d}{2^d}\right)$$

Les majorations $2^{d_{max}} < n$ et $\sum_{d=0}^{d_{max}} d/2^d \leq 2$ donnent le résultat voulu.

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 $N_h-2^h,\dots,N_h-1$.

On procède par récurrence sur la hauteur. La proposition est vraie pour l'arbre de hauteur 0 et d'unique nœud sa racine. En la supposant vraie pour un arbre complet de hauteur h, un arbre complet de hauteur $h+1$ a $N_{h+1} = N_h + 2^{h+1} = 2N_h + 1$ nœuds et ses feuilles sont de numéros $2\times(N_h-2^h)+1 = 2N_h - 2^{h+1} + 1 = N_{h+1} - 2^{h+1}$ à $2\times(N_h-1)+2 = 2N_h = N_{h+1}-1$.

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.

D'après la question précédente, un nœud de profondeur h a un numéro supérieur à tout nœud de profondeur inférieure : ceci est vrai pour un arbre complet, et reste vrai pour un arbre non-complet, qu'on peut construire depuis un arbre complet en supprimant des nœuds sans que cela ne change la numérotation des nœuds restants. On considère maintenant le cas de nœuds de même profondeur, en raisonnant par récurrence sur la profondeur des nœuds. La proposition est vraie à profondeur 1 puisque l'enfant gauche de la racine est parcouru avant, et a un numéro inférieur à l'enfant droit. Supposons maintenant la propriété vraie à profondeur h, et considérons u, v deux nœuds de profondeur h, avec u de numéro x_u visité avant v de numéro $x_v \geq x_u + 1$. Par la propriété d'un parcours en largeur, les enfants de x_u (s'ils existent) sont visités avant ceux de x_v, et ils sont de numéros $\leq 2x_u+2$, tandis que ceux de x_v sont de numéros $\geq 2(x_u+1)+1$. On conclut par la remarque que tous les nœuds de profondeur $h+1$ sont enfants de nœuds de profondeur h.

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 $\nu(n_L) < \nu(n_R)$.

On raisonne encore une fois par récurrence sur la profondeur. Soit n le nœud dont n_L et n_R sont des descendants, par la définition de ν la proposition est vraie dans le cas où n_L & n_R sont enfants immédiats de n. On suppose maintenant la proposition vraie pour tous les descendants de n à profondeur h, et considère n_L & n_R à profondeur $h+1$. Par hypothèse, les parents n'_L & n'_R de n_L & n_R appartiennent respectivement au sous-arbre gauche et droit de n ; par hypothèse de récurrence on a donc $x'_L := \nu(n'_L) < x'_R := \nu(n'_R)$. On a alors bien $\nu(n_R) \geq 2x'_R + 1 \geq 2(x'_L+1)+1$ et $\nu(n_L) \leq 2x'_L + 2 < 2(x'_L+1)+1$.

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.

On raisonne toujours par récurrence sur la profondeur. La proposition est vraie pour n_L, n_R enfants du même nœud. Supposons-la vraie pour tous nœuds de plus proche ancêtre commun à distance au plus h et considérons n_L, n_R de plus proche ancêtre commun n_A à distance $h+1>1$, avec $\nu(n_L) < \nu(n_R)$. Soit n'_L & n'_R respectivement parent de n_L & n_R, par propriété de ν et l'hypothèse $\nu(n_L) < \nu(n_R)$, on a $\nu(n'_L) < \nu(n'_R)$; par hypothèse de récurrence on a donc que n'_L (resp. n'_R) appartient au sous-arbre de gauche (resp. droite) de n_A, et il en va donc de même pour n_L & n_R.

Pour conclure :

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

Montrons que les N nœuds d'un arbre quasi-complet A de hauteur $h+1$ ont une numérotation consécutive pour ν. Par la première question et l'hypothèse que A est quasi-complet, les nœuds de numéros $0,\dots,N_h-1$ sont présents dans A et il suffit de montrer que ceux de numéros $N_h,\dots,N-1$ le sont également. Supposons par l'absurde que A ne possède aucun nœud au numéro $N_h \leq x \leq N-1$. Puisque A est quasi-complet, il possède un nœud n_P au numéro « parent » $(x-1)/2$; soit maintenant y le numéro maximum d'un nœud de A et n_Q son parent, on distingue les cas :

— $n_P = n_Q$: dans ce cas l'unique enfant de n_P est son enfant droit et A n'est pas quasi-complet

— $n_P < n_Q$ (seul cas restant) : par la question précédente, n_P (resp. n_Q) appartient au sous-arbre gauche (resp. droit) de son plus proche ancêtre commun avec n_Q (resp. n_P), et n_P est donc visité avant n_Q dans un parcours en profondeur gauche. Que n_P soit une feuille ou ne possède qu'un enfant est alors en contradiction avec le fait que A est quasi-complet

Montrons maintenant que A dont les N nœuds sont numérotés $0, \ldots, N_h - 1, \ldots, N-1$ est quasi complet. Soit deux feuilles n_A, n_B de profondeur respectivement h et $h+1$, supposons par l'absurde que n_A soit visité avant n_B dans un parcours en profondeur. Soit $n'_B \neq n_A$ le parent de n_B, et n_C le plus proche ancêtre commun de n_A & n_B (ou n'_B). Par hypothèse n_A (resp. n'_B) appartient au sous-arbre de gauche (resp. droite) de n_C, et par la question précédente on a donc $\nu(n_A) < \nu(n'_B)$. Il s'ensuit que $\nu(n_B) > 2 \times \nu(n_A) + 1$, ce qui est une contradiction puisque ce dernier numéro ne correspond à aucun nœud.
On raisonne de même pour le cas où n_A de profondeur h ne possède qu'un enfant qui n'est pas le dernier visité dans un parcours en profondeur.

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 ».