Devoir maison n°1

Corrigé

⋆ ⋆ ⋆

Arbres et tas binomiaux

Question 1 On peut envisager différentes manières d'écrire une telle fonction sur les arbres d'arité quelconque. On propose ici trois versions :

– Fonction directement récursive :

```
let rec taille (N(x, lst)) = match lst with
    | [] -> 1
    | a :: q -> taille a + taille (N(x, q))
```

– Avec une fonction auxiliaire qui renvoie la somme des tailles des arbres dans une liste :

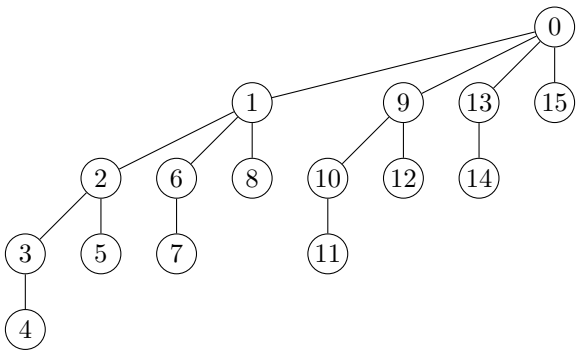
```
let taille a =
    let rec somme_tailles = function
        | [] -> 0
        | N(x, lst) :: q -> 1 + somme_tailles lst + somme_tailles q
    in
    somme_tailles [a]
```

– Avec l'utilisation de fonctionnelles OCaml (on calcule récursivement les tailles des enfants, et on les ajoute à 1) :

```
let rec taille (N(x, lst)) =
    List.fold_left (+) 1 (List.map taille lst)
```

1 Arbres binomiaux

Question 2 On obtient l'arbre suivant :



Question 3 On montre par récurrence forte sur $k \in \mathbb{N}$ que $|\mathcal{B}_k| = 2^k$:

– pour $k = 0$, $|\mathcal{B}_0| = 1 = 2^0$;

– supposons le résultat établi pour tout $0 \leqslant i < k$, $k \in \mathbb{N}^*$ fixé. Alors $|\mathcal{B}_k| = 1 + \sum_{i=0}^{k-1} |\mathcal{B}_i| = 1 + \sum_{i=0}^{k-1} 2^i = 2^k$.

On conclut par récurrence.

En notant $f(\mathcal{T})$ le nombre de feuilles de $\mathcal{T}$, on obtient que $f(\mathcal{B}_k) = \sum_{i=0}^{k-1} f(\mathcal{B}_i)$. Sachant que $f(\mathcal{B}_0) = f(\mathcal{B}_1) = 1$, on en déduit que $f(\mathcal{B}_k) = 2^{k-1}$ pour $k \geqslant 1$ (cette expression vérifie la même relation de récurrence et les mêmes conditions initiales).

Question 4 On peut définir un arbre binomial de la façon suivante :

– $\mathcal{B}_0$ est réduit à sa racine ;
– pour $k \geqslant 0$, si $\mathcal{T}_k = (r_k, \mathcal{L}_k)$ est un arbre binomial d'ordre k, alors $(r_{k+1}, \mathcal{T}_k :: \mathcal{L}_k)$ est un arbre binomial d'ordre $k+1$, où $x :: q$ désigne la liste q à laquelle on a rajouté l'élément x.

En effet, dans la définition récursive de l'énoncé, on remarque que $(r_k, (\mathcal{T}_{k-2}, \ldots, \mathcal{T}_0))$ désigne un arbre binomial d'ordre $k-1$, au même titre que $\mathcal{T}_{k-1}$.

Question 5 On écrit une fonction récursive. On décale la racine et on applique la fonction de décalage à chaque enfant. Même si la fonction semble ne pas avoir de cas de base, il existe quand même, car si `lst` est vide, l'appel à `List.map` renvoie une liste vide.

```
let rec decale n (N(r, lst)) =
    N(r + n, List.map (decale n) lst);;
```

Question 6 L'idée est ici d'utiliser la définition de la question 3 et de décaler de la bonne valeur les numéros concernés. Pour connaître cette valeur, il faut pouvoir calculer la taille d'un arbre binomial, en utilisant la formule de la question 2. Le premier enfant sera décalé de 1 (car la racine sera 0, donc la racine du premier enfant sera 1), et le deuxième arbre binomial calculé sera décalé de la taille du premier. On n'oublie pas de remettre la racine à 0.

```
let rec bk = function
    | 0 -> N(0, [])
    | k ->
        let b = bk (k - 1) in
        let N(r, lst) = decale (1 lsl (k - 1)) b in
        N(0, decale 1 b :: lst);;
```

Question 7 On vérifie que $h(\mathcal{B}_k) = k$. En effet, la relation est vérifiée pour $k = 0$. Si elle est vraie pour tous les $0 \leqslant i \leqslant k$, alors $h(\mathcal{B}_{k+1}) = 1 + \max_{0 \leqslant i \leqslant k} h(\mathcal{B}_i) = 1 + k$. On conclut par récurrence forte.

Soient deux nœuds x et y tels que $d(x, y)$ est maximal. Notons z le plus proche (celui de profondeur maximale) ancêtre commun à x et y. Alors $d(x, y) = d(z, x) + d(z, y) \leqslant prof(x) + prof(y) \leqslant 2h(\mathcal{T})$. Comme il n'y a qu'une seule feuille de profondeur maximale dans $\mathcal{B}_k$ (cela se montre par récurrence), la longueur maximale d'un chemin entre deux nœuds est au plus $2h(\mathcal{B}_k) - 1$, qui est atteinte entre la feuille la plus profonde du premier enfant $(\mathcal{B}_{k-1})$ et la feuille la plus profonde du deuxième enfant $(\mathcal{B}_{k-2})$. On en déduit que cette longueur est $2k - 1$.

Notons enfin que si $k = 0$, cette longueur est 0 (un seul nœud) et pour $k = 1$, cette longueur est 1 (donc la formule convient).

Question 8 Notons $p(k, \ell)$ le nombre de nœuds à profondeur ℓ dans $\mathcal{B}_k$. Montrons que $p(k, \ell) = \binom{k}{\ell}$ (d'où le nom d'arbre binomial). En effet, cette formule convient pour $k = 0$ et $k = \ell$. De plus, en utilisant la définition de la question 3, on a $p(k+1, \ell) = p(k, \ell) + p(k, \ell - 1)$. On reconnait ici la formule de Pascal, qui nous assure bien de l'égalité annoncée.

2 Tas binomiaux

Question 9 Comme les tailles des arbres binomiaux sont des puissances de 2 toutes distinctes, on en déduit par l'unicité de la décomposition en binaire d'un entier que le nombre d'arbres binomiaux dans un tas de taille n est le nombre de 1 dans la décomposition binaire de n, et les ordres sont les indices des 1 dans la décomposition en binaire de n (0 étant l'indice du bit de poids le plus faible).

Plus formellement, si $n = \sum_{i=0}^{m} b_i 2^i$ avec $b_i \in \{0, 1\}$, alors le nombre d'arbres binomiaux est $\sum_{i=0}^{m} b_i$ et les ordres sont les $\{i \in [\![0, m]\!] \mid b_i \neq 0\}$.

Question 10 On se contente de comparer les racines et de rajouter l'un des deux arbres comme enfant de l'autre.

```
let fusion_arbre (k1, a1) (k2, a2) =
    assert (k1 = k2);
    let N(x1, l1) = a1 and N(x2, l2) = a2 in
    if x1 >= x2 then (k1 + 1, N(x1, a2 :: l1))
    else (k2 + 1, N(x2, a1 :: l2));;
```

Question 11 L'idée est de procéder comme pour la fusion du tri fusion : on ajoute à la nouvelle liste l'arbre d'ordre le plus petit. Dans le cas d'égalité, pour conserver la propriété d'ordres tous différents, on fusionne les deux arbres grâce à la fonction précédente, et on rajoute l'arbre obtenu en tête du premier tas. Notons que le premier cas permet de faire des fusions successives dans le premier tas pour éviter les doublons.

```
let rec fusion t1 t2 = match t1, t2 with
    | (k, a1) :: (k', a2) :: q1, t2 when k = k' ->
        fusion ((fusion_arbre (k, a1) (k', a2)) :: q1) t2
    | [], t | t, [] -> t
    | (k1, a1) :: q1, (k2, a2) :: q2 ->
        if k1 < k2 then (k1, a1) :: fusion q1 t2
        else if k1 > k2 then (k2, a2) :: fusion t1 q2
        else fusion ((k1, a1) :: (k2, a2) :: q1) q2;;
```

Question 12 C'est naïf, mais on se contente de fusionner un tas à un élément avec l'autre tas.

```
let insertion t x =
    fusion [(1, N(x, []))] t;;
```

Question 13 On remarque que lors d'un appel à `insertion`, les appels à `fusion` se feront toujours avec une première liste de taille au plus 1. Cette liste deviendra vide lorsqu'on rencontrera le premier 0 dans la décomposition en binaire de n, la taille du tas (car alors la « propagation de retenue » s'arrêtera). Le nombre d'appels récursifs sera alors le nombre de 1 consécutifs à droite de l'écriture en binaire de n. On remarque alors que le nombre d'entiers de $[\![0, 2^m - 1]\!]$ dont le nombre de 1 consécutifs à droite de leur décomposition binaire est p est 2^{m-p}. La complexité totale après 2^m insertion est donc de l'ordre de (pour calculer la somme, on considère la dérivée de $\sum x^p$) :

$$\sum_{p=0}^{m} p 2^{m-p} = 2^{m-1} \sum_{p=0}^{m} p \left(\frac{1}{2}\right)^{p-1} = m - 2(m+1) + 2^{m+1} = \mathcal{O}(2^m)$$

En divisant par le nombre total d'insertion, on obtient le résultat attendu.

Question 14 On procède récursivement : le cas de base est le tas à un seul élément. S'il y a au moins

deux arbres, on calcule récursivement l'extraction de la queue, et on compare sa racine avec celle de la tête. On reconstitue le reste en fonction.

```
let rec extraire_arbre = function
    | [] -> failwith "Tas vide"
    | [a] -> a, []
    | a :: q ->
        let b, lst = extraire_arbre q in
        let (_, N(x, _)) = a and (_, N(y, _)) = b in
        if x >= y then a, q
        else b, a :: lst;;
```

Question 15 L'idée est la suivante : on extrait l'arbre qui contient le maximum, on transforme la liste de ses enfants en tas binomial, puis on fusionne ce nouveau tas avec le tas restant de l'extraction. On renvoie la valeur du maximum et le résultat de la fusion.

Pour transformer la liste des enfants en tas, on profite du fait que les ordres sont consécutifs et tous présents (pour ne pas avoir à calculer les tailles des tas). On fait attention à renverser la liste des enfants pour que les arbres soit par ordre croissants.

```
let enfants_vers_tas lst =
    let rec ajoute_ordre k = function
        | [] -> []
        | a :: q -> (k, a) :: ajoute_ordre (k + 1) q in
    ajoute_ordre 0 (List.rev lst);;
```

Dès lors, on obtient :

```
let extraire_max t =
    let (k, N(x, lst)), t' = extraire_arbre t in
    let t_lst = enfants_vers_tas lst in
    x, fusion t_lst t';;
```

Question 16 D'après la question 9, le nombre d'arbres dans un tas est le nombre de 1 dans l'écriture binaire de n. En particulier, il est inférieur à $1 + \log_2 n$. On en déduit que le calcul de `maximum` et `extraire_arbre` se sont en $\mathcal{O}(\log n)$.

De plus, l'ordre maximal d'un arbre binomial dans le tas est également $\log_2 n$. On en déduit que la conversion des enfants en tas se fait en $\mathcal{O}(\log n)$.

Finalement, la fusion se fait entre deux tas contenant au plus $\log_2 n$ arbres, donc la complexité totale de `extraire_max` est en $\mathcal{O}(\log n)$, ce qui correspond à la complexité des tas binaires.

⋆ ⋆ ⋆