Daprès un travail original de A. Reissman.

Tas gaucher (leftist heap)

1 Arbre gaucher

Q.1. La définition récursive est : si $a = N(x, ag, ad)$, alors

$$\boxed{\mathrm{dn}(a) = 1 + \min\left(\mathrm{dn}(ag), \mathrm{dn}(ad)\right)}$$

En effet :

- si ag ou ad est nil, alors a a au plus un fils donc par définition $\mathrm{dn}(a) = 0$. Et grâce à la convention $\mathrm{dn}(\text{nil}) = -1$, on a bien $\mathrm{dn}(a) = 1 + \min\left(\mathrm{dn}(ag), \mathrm{dn}(ad)\right)$.
- sinon, ag et ad sont non vides. Supposons que $\mathrm{dn}(ag) \leq \mathrm{dn}(ad)$ (l'autre cas se traite de manière analogue). Soit $y_1, \ldots, y_p$ un plus court chemin de la racine y_1 de ag vers un nœud y_p ayant au plus un fils. On a donc $p = \mathrm{dn}(ag) = \min(\mathrm{dn}(ag), \mathrm{dn}(ad))$. En ajoutant x, on obtient un chemin $x, y_1, \ldots, y_p$ de la racine de a jusqu'à un nœud ayant au plus un fils. On voit alors facilement que ce chemin est le plus court, et il est de longueur $1 + \min(\mathrm{dn}(ag), \mathrm{dn}(ad))$.

Q.2. Aucune difficulté.

```
let dn a = match a with
  | Nil -> -1
  | N(_,d,_,_) -> d
```

Q.3. La définition récursive d'un arbre gaucher est la suivante : nil est gaucher, et $N(x, ag, ad)$ est gaucher si et seulement si ag et ad le sont, et si $\mathrm{dn}(ag) \geq \mathrm{dn}(ad)$.

```
let rec est_gaucher a = match a with
  | Nil -> true
  | N(_, _, ag, ad) -> (est_gaucher ag)
                       && (est_gaucher ad)
                       && (dn ag >= dn ad)
```

Certains ont supposé que `Nil` n'est pas gaucher...

La définition s'applique-t-elle à nil ?

La définition dit "pour tout nœud de l'arbre". Or, nil ne contient aucun nœud (c'est l'absence d'arbre).

Donc, la condition "pour tout nœud de nil" est vraie (puisqu'il n'y a aucun nœud à vérifier) !!

L'arbre vide est donc gaucher !

Q.4. (a) La définition récursive de la fonction est : si $a = \text{nil}$, $\mathrm{lbd}(a) = -1$, sinon $a = N(x, ag, ad)$ et alors $\mathrm{lbd}(a) = 1 + \mathrm{lbd}(ad)$.

```
let rec lbd a = match a with
  | Nil -> -1
  | N(_,_,_,ad) -> 1 + lbd ad
```

(b) On montre que la propriété $P(a)$: « si a est gaucher alors $\mathrm{lbd}(a) = \mathrm{dn}(a)$ » est vraie pour tout arbre a par induction structurelle :

- $P(\text{nil})$ est évidente.
- Soit a un arbre de la forme $N(x, ag, ad)$ et supposons que $P(ag)$ et $P(ad)$ sont vraies.
  On suppose que a est gaucher, alors par définition ag et ad sont gauchers et $\mathrm{dn}(ag) \geq \mathrm{dn}(ad)$. Donc $\mathrm{dn}(a) = 1 + \min(\mathrm{dn}(ag), \mathrm{dn}(ad)) = 1 + \mathrm{dn}(ad)$. Par l'hypothèse $P(ad)$, on a $\mathrm{dn}(ad) = \mathrm{lbd}(ad)$, donc $\mathrm{dn}(a) = 1 + \mathrm{lbd}(ad) = \mathrm{lbd}(a)$ par la définition récursive de lbd. On a établi $P(a)$.

Q.5. (a) Par induction structurelle : la propriété à montrer est $P(a)$: « si a est gaucher de taille n, alors $n \geq 2^{\mathrm{lbd}(a)+1} - 1$ ».

— L'arbre vide nil a une taille nulle et $\text{lbd}(\text{nil}) = -1$, donc $P(\text{nil})$ est évidente.

— Soit $a = N(x, ag, ad)$, on suppose que $P(ag)$ et $P(ad)$ sont vraies.
Si a est gaucher de taille, alors ag et ad sont gauchers, donc en notant ng et nd leurs tailles, on a $ng \geq 2^{\text{lbd}(ag)+1} - 1$ et $nd \geq 2^{\text{lbd}(ad)+1} - 1$.
On a $n = 1 + ng + nd$, donc $n \geq 1 + 2^{\text{lbd}(ag)+1} - 1 + 2^{\text{lbd}(ad)+1} - 1$ or $\text{lbd}(ag) = \text{dn}(ag)$ et $\text{lbd}(ad) = \text{dn}(ad)$ (par Q.4.(b)) et $\text{dn}(ag) \geq \text{dn}(ad)$ (a est gaucher) ce qui entraîne $n \geq 2 \times 2^{\text{lbd}(ad)+1} - 1 = 2^{(\text{lbd}(ad)+1)+1} - 1 = 2^{\text{lbd}(a)+1} + 1$. On a établi $P(a)$.

(b) On a $2^{\text{lbd}(a)+1} \leq n + 1$ donc $\text{lbd}(a) + 1 \leq \log_2(n+1)$ en particulier $\text{lbd}(a) < \log_2(n+1)$ donc $\boxed{\text{lbd}(a) \leq \lfloor \log_2(n+1) \rfloor}$. Ainsi $\text{lbd}(a)$ est un $O(\log n)$.

2 Arbre croissant

Q.6. La définition récursive est très simple : nil est croissant, et $N(x, ag, ad)$ est croissant si et seulement si ag et ad le sont, et si de plus x est inférieur ou égal aux étiquettes des racines de ag et ad, s'ils ne sont pas vides.

```
let rec est_croissant a = match a with
 | Nil -> true
 | N(x, _, ag, ad) -> match ag, ad with
  | Nil, Nil -> true
  | N(y,_,_,_), Nil -> est_croissant ag && x <= y
  | Nil, N(z,_,_,_) -> est_croissant ad && x <= z
  | N(y,_,_,_), N(z,_,_,_) -> (est_croissant ag) &&
                              (est_croissant ad) &&
                              x <= y && x <= z
```

Q.7. Pour un arbre croissant, le minimum se trouve à la racine.

```
let minimum a = match a with
  | Nil -> failwith "minimum"
  | N(x,_,_,_) -> x
```

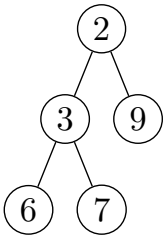
3 Tas gaucher

Q.8. (a) Soit $a_1 = N(2, N(9, \text{nil}, \text{nil}), N(7, \text{nil}, nil))$ et $a_2 = N(3, N(6, \text{nil}, \text{nil}), \text{nil})$ les deux arbres.
Les différentes étapes sont les suivantes :

— On compare leurs racines : $2 \leq 3$ on fusionne donc récursivement $N(7, \text{nil}, \text{nil})$ (le fils droit de a_1) avec a_2.

— Comme $7 > 3$, on échange : on fusionne a_2 avec $N(7, \text{nil}, \text{nil})$. On fusionne récursivement le fils droit (nil) de a_2 avec $N(7, \text{nil}, \text{nil})$, ce qui donne immédiatement $N(7, \text{nil}, \text{nil})$.

— On remonte les appels récursifs : la fusion de a_2 avec $N(7, \text{nil}, \text{nil})$ donne $N(3, N(6, \text{nil}, \text{nil}), N(7, \text{nil}, \text{nil}))$ (on n'échange pas les fils, qui ont la même distance à Nil). Cet arbre a une distance à Nil de 2, alors que le fils gauche de a_1 a une distance à Nil de 1. On les échange, et on obtient finalement comme fusion de a_1 et a_2 l'arbre suivant :



(b) La méthode est la suivante : on montre que la fonction vérifie les propriétés demandées pour les cas de base. Ensuite, on montre que la fonction est correct si les appels récursifs le sont. C'est une preuve par induction structurelle.
Si l'un des deux arbre est vide, la fusion renvoie l'autre donc toutes les propriétés sont évidentes.
Sinon t_1 et t_2 sont de la forme $t_1 = N(x_1, ag_1, ad_1)$ et $t_2 = N(x_2, ag_2, ad_2)$. On raisonne dans le cas où $x_1 \leq x_2$, l'autre se traitant de manière analogue.

L'arbre ad_1 est un tas gaucher. On suppose que l'appel récursif donnant la fusion ad'_1 de ad_1 avec t_2 est bien un tas gaucher, qui contient les étiquettes de ad_1 et t_2.

La fusion de t_1 et t_2 est soit $N(x, ag_1, ad'_1)$, soit $N(x_1, ad'_1, ag_1)$, de façon à ce que la distance à nil à gauche soit au moins égale à celle de droite. Donc la fusion de t_1 et t_2 est un arbre gaucher.

De plus, ad'_1 et t_2 sont croissants, et leurs étiquettes sont celles de ad_1 et t_2 donc sont inférieures à x_1. Donc la fusion de t_1 et t_2 est un arbre croissant.

Enfin, les étiquettes présentes dans la fusion de t_1 et t_2 sont celles de ad'_1 (donc de ad_1 et t_2), celles de ag_1, et x. On a bien toutes les étiquettes de t_1 et t_2 réunies (en comptant les occurences).

(c) Il suffit de traduire l'algorithme décrit dans l'énoncé. On fait juste attention à mettre à jour la distance à nil de la racine.

```
let rec fusion t1 t2 = match t1, t2 with
  | Nil, _ -> t2
  | _, Nil -> t1
  | N(x1, dn1, ag1, ad1), N(x2, dn2, ag2, ad2) ->
      if x1 > x2 then fusion t2 t1
      else let newad1 = fusion ad1 t2 ina
        if dn ag1 >= dn newad1
        then N(x1, dn newad1 + 1, ag1, newad1)
        else N(x1, dn ag1 + 1, newad1, ag1)
```

(d) Pour avoir la complexité, il suffit de compter le nombre d'appels récursifs (car par ailleurs on ne réalise que des opérations en temps constant, du fait que `dn` est en temps constant). Or chaque appel récursif se fait soit sur le sous-arbre droit de t_1, soit sur le sous-arbre droit de t_2. Le nombre d'appels au total est donc en $\text{lbd}(t_1) + \text{lbd}(t_2)$, et puisque ces arbres sont gauchers, on a par la Q.5 une complexité en $O(\log(n_1) + \log(n_2))$.

La complexité de l'appel `fusion t1 t2` est notée $C_{\text{dn(t1)+dn(t2)}}$ et vérifie

$$C_{\text{dn(t1)+dn(t2)}} = \underbrace{1}_{\text{appels à dn}} + C_{\text{dn(t1)+dn(t2)}-1}$$

Q.9. Pour ajouter une valeur x au tas t, il suffit de fusionner t avec le tas réduit à une feuille $N(x, \text{nil}, \text{nil})$.

```
let insere x t = fusion t (N(x, 0, Nil, Nil))
```

Q.10. La complexité est en en $O(\log(n))$, où n est la taille du tas.

Q.11. Étant donné un tas $t = N(x, ag, ad)$, l'étiquette minimale est x qui est à la racine. Pour supprimer x il suffit de fusionner ag et ad, ce qui donne bien un tas gaucher puisque ag et ad en sont.

```
let supprime_min t = match t with
  | Nil -> failwith "supprime_min"
  | N(_,_,ag,ad) -> fusion ag ad
```

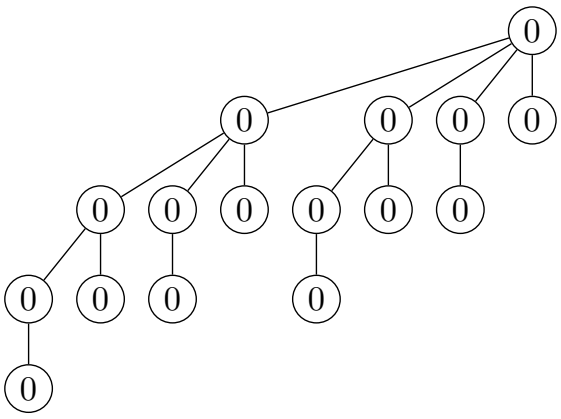
Q.12. En notant ng et nd les tailles de ag et ad, la complexité est $O(\log(ng) + \log(nd))$ donc en $O(\log n)$ puisque n majore ng et nd.

Tas binomial (binomial heap)

---

1 Arbre binomial

Q.13. (a) Un arbre binomial de rang 4 :



(b) Par récurrence (forte) sur n :

— initialisation : un arbre binomial de rang 0 contient $1 = 2^0$ nœud et est de hauteur 0.

— Supposons que l'énoncé est vrai aux rangs $0, 1, \dots, n-1$. Soit a binomial de rang n, et $a_{n-1}, \dots, a_0$ ses fils. Chaque a_k est de taille 2^k et est de hauteur k par les hypothèses de récurrence.

La taille de a est la somme des tailles de ses fils, augmentée de 1 (pour la racine), donc vaut $1 + \sum_{k=0}^{n-1} 2^k = 1 + \frac{2^n - 1}{2 - 1} = 2^n$.

La hauteur de a est le maximum des hauteurs de ses fils, augmentée de 1, donc vaut $n-1+1 = n$.

On a ainsi établi l'hérédité forte pour la propriété.

On a bien montré :

un arbre binomial de rang $n \geq 0$ est de taille 2^n et de hauteur n.

(c) Soit a un arbre binomial, dont la racine a pour fils $a_{n-1}, \dots, a_0$ tels que a_k est binomial de rang k. En retirant a_{n-1}, on obtient une racine ayant pour fils $a_{n-2}, \dots, a_0$, c'est-à-dire un arbre binomial de rang $n-1$.

Ainsi, on peut obtenir un arbre binomial de rang n à partir de deux arbres binomiaux de rang $n-1$: on ajoute l'un des arbres comme premier fils de l'autre.

(d) On raisonne par récurrence sur n.

— Pour $n = 0$: un arbre binomial de rang 0 est réduit à une feuille, donc il y a $1 = \binom{0}{0}$ nœud à la profondeur 0.

— Soit $n > 0$, supposons le résultat vrai pour un arbre binomial de rang $n-1$. On utilise la question précédente : un arbre binomial a de rang n est obtenu à partir de deux arbres binomiaux a_1 et a_2 de rang $n-1$ en ajoutant a_1 comme premier fils de a_2. Pour $k = 0$: il n'y a qu'un $(= \binom{n}{0})$ seul nœud à la profondeur 0, la racine. Pour $k \in [\![1, n]\!]$, les nœuds à la profondeur k dans a sont ceux à la profondeur k dans a_2 et ceux à la profondeur $k-1$ dans a_2. Par hypothèse de récurrence, il y a donc $\binom{n-1}{k} + \binom{n-1}{k-1} = \binom{n}{k}$ nœuds à la profondeur k dans a, ce qui établit l'hérédité.

Q.14. Le rang est dans la structure, la fonction est immédiate :

```
let rang (Node(_, r, _)) = r
```

Q.15. En utilisant la question Q11.(c), on peut écrire la fonction suivante :

```
let  rec verif_binomial (Node(x,r,l)) = match l with
  | [] -> r = 0
  | h::t -> (rang h = r-1)
        && (verif_binomial h)
        && (verif_binomial (Node(x,r-1,t)))
```

2 Forêt binomiale

Q.16. (a) On a vu qu'un arbre binomial de rang k est de taille 2^k.

Étant donné un entier naturel n, il n'y a qu'une seule façon d'écrire n comme somme de puissances de 2, elle est obtenue par l'écriture binaire de n. Si n s'écrit $b_p \cdots b_1 b_0$ en base 2, alors $n = \sum_{i=0}^{p} b_i 2^i$

avec $b_i \in \{0,1\}$. On supprime les b_i qui sont nuls et on numérote les 1 dans l'ordre croissant, on obtient $n = \sum_{k=0}^{q-1} 2^{i_k}$ avec q le nombre de 1 dans l'écriture binaire de n.
Une famille binomiale de taille n est donc consituée d'arbres binomiaux de rangs i_0, $i_1, \cdots, i_{q-1}$.
Par exemple, $n = 6 = 2^1 + 2^2$ s'écrit en binaire 110, une forêt binomiale de taille 6 est donc formée de deux arbres binomiaux de rangs respectifs 1 et 2 comme dans l'exemple figure 6.

(b) Pour une longueur ℓ de forêt donnée, on obtient la plus petite taille possible en ayant des arbres binomiaux de rangs $0, 1, \ldots, \ell-1$. La taille est alors $\sum_{k=0}^{\ell-1} 2^k = 2^\ell - 1$. Dit autrement : le plus petit nombre contenant ℓ chiffres 1 en binaire est $11\cdots1$ qui vaut $2^\ell - 1$.
Si on note n la taille d'une forêt de longueur ℓ, on a donc $n \geq 2^\ell - 1$, donc $2^\ell \leq n+1$, puis $\ell \leq \log(n+1)$ donc $\boxed{\ell \leq \lfloor \log(n+1) \rfloor}$.

3 Tas binomial

3.1 Obtention du minimum

Q.17. Le mimimum du tas est la plus petite racine des arbres qui la composent (car ce sont des arbres croissants). On s'inspire de l'algorithme classique de recherche d'un minimum dans une liste.

```
let rec minimum t = match t with
  | [] -> failwith "minimum"
  | [Node(x,_,_)] -> x
  | Node(x,_,_)::tt -> min x (minimum tt)
```

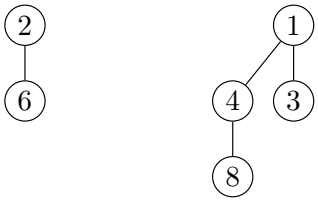
Q.18. Cet algorithme est linéaire en la longueur de la liste, donc logarithmique en la taille du tas par la Q.14.

3.2 Insertion

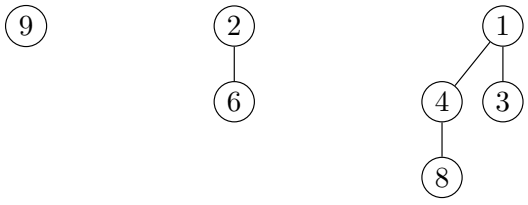
Q.19. On traduit simplement la méthode décrite :

```
let union a1 a2 =
  let Node(x1, r1, l1) = a1 and Node(x2, r2, l2) = a2
  in if x1 <= x2
  then Node(x1, r1+1, a2::l1)
  else Node(x2, r2+1, a1::l2)  (* r1 = r2 *)
```

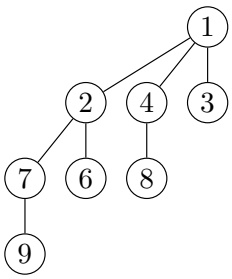
Q.20. (a) — On insère 2 : le premier arbre de t est de rang 0, on réalise son union avec l'arbre de rang 0 contenant 2. L'arbre obtenu est de rang 1, on arrête là l'algorithme et on obtient le tas t_2 :



— On insère ensuite 9, le premier arbre de t_2 étant de rang 1 il n'y a pas d'union à faire. On obtient t_3 :



— On insère 7, on considère l'arbre de rang 0 contenant uniquement la racine d'étiquette 7. On réalise son union avec le premier arbre de t_3 pour obtenir un arbre de rang 1, que l'on réunit avec le deuxième arbre de t_3 pour obtenir un arbre de rang 2, que l'on réunit avec le dernier arbre de t_3. À la fin il ne reste qu'un seul arbre, de rang 3.



(b) On utilise une fonction auxiliaire qui insère un arbre dans le tas, il suffit alors de l'appeler avec l'arbre de rang 0 ne contenant que la racine x.

```
let rec aux_ins ak t = match t with
    | [] -> [ak]
    | b::t2 -> if rang ak <> rang b then ak::t
               else aux_ins (union ak b) t2
let insere x t =
   aux_ins (Node(x,0,[])) t
```

(c) La complexité est linéaire en la longueur du tas, donc logarithmique en la taille.

Q.21. (a) Cette relation repose sur la remarque suivante : à chaque fois que l'on réalise une union lors de l'insertion, on diminue le nombre d'arbre dans le tas. Plus précisémment : si l'on réalise exactement C_i union pour insérer la i-ème valeur, cela signifie que le tas avant insertion commençait par des arbres de rangs $0, 1, \dots, C_i - 1$ et le suivant n'est pas de rang C_i. À la fin de la i-ème insertion, tous les C_i-premiers arbres sont remplacés par un seul arbre. Le nombre d'arbres a donc diminué de $C_i - 1$ lors de cette i-ème insertion. On a donc $L_i - L_{i-1} = -(C_i - 1)$ ou encore $C_i + (L_i - L_{i-1}) = 1$.

(b) On obtient $\sum_{i=1}^{n} C_i + \sum_{i=1}^{n} (L_i - L_{i-1}) = n$ et par télescopage :

$\sum_{i=1}^{n} C_i + L_n - L_0 = n$ et comme $L_0 = 0$, $\boxed{\sum_{i=1}^{n} C_i = n - L_n \leq n}$.

(c) Déjà on peut dire que la complexité d'une insertion est de l'ordre du coût défini dans l'énoncé, puisqu'il y a autant d'appels récursifs dans la fonction `insere` que d'unions d'arbres. On a prouvé que le coût total des n insertions est au plus n, donc la complexité de l'ensemble des n insertions est un $O(n)$. Chacune des insertions est donc en temps amorti constant.

3.3 Fusion de tas

Q.22. On compare les rangs des premiers arbres de chacun des tas. S'ils sont différents, on place celui de rang le plus petit devant dans la fusion, et on continue récursivement. S'ils ont même rang, on réalise leur union, que l'on insère par exemple dans le premier tas, puis on fusionne récursivement.

```
let rec fusion t1 t2 = match t1, t2 with
  | [], _ -> t2
  | _, [] -> t1
  | a1::tt1, a2::tt2 ->
      if rang a1 < rang a2 then a1::(fusion tt1 t2)
      else if rang a2 < rang a1 then a2::(fusion t1 tt2)
      else fusion (aux_ins (union a1 a2) tt1) tt2
```

On compte le nombre de `::` . Il y en a un par appel à `union` .

Notons N_1, N_2 les longueurs (on sait qu'elles sont logarithmiques en fonction de la taille) des tas `t1,t2` .

L'équation de récurrence est de la forme

$$C_{N_1+N_2} = C_{N_1+N_2-1} + 1$$

dans le cas où les rangs sont différents.

Lorsque les rangs sont égaux, on commence par faire une union (1 concaténation) puis on l'insère dans le tas `tt1` de longueur $N_1 - 1$. Il y a ensuite, disons, $k \in [0, N_1 - 1]$ calculs de union (donc autant de `::`) puis un ultime `::` pour ajouter la dernière union au reste du tas. Après insertion, le nouveau tas est de taille $(N_1 - 1) - k + 1$.

Il faut alors faire la fusion de ce nouveau tas avec `tt2` . on obtient une équation de la forme

$$C_{N_1+N_2} = C_{N_1-k+N_2-1} + k + 2$$

Les deux équations de récurrences conduisent à $C_{N_1+N_2} = O(N_1 + N_2)$. la complexité de l'appel à fusion est un $\boxed{O\left(\log(n_1) + \log(n_2)\right)}$ où n_1, n_2 sont les tailles des tas.

3.4 Suppression du minimum

Q.23. (a) Après avoir récupéré l'arbre contenant le minimum `mini` (fonction `minimum`), on refait un parcours du tas à la recherche de l'arbre ayant comme racine `mini` .
La complexité est linéaire en la longueur du tas, donc logarithmique en sa taille.

```
let retire_arbre_min t =
  let mini = minimum t in
  let rec aux tt = match tt with
    | [] -> failwith "retire_arbre"
    | a::tt2 -> let Node(x,_,_) = a in
        if x = mini then a, tt2
        else let aa, newtt = aux tt2 in aa, a::newtt
  in aux t
```

(b) Une fois que l'on supprime la racine de l'arbre *amin* contenant le minimum, on récupère la liste des fils. On remarque et on démontre facilement qu'en renversant la liste de ces fils, on obtient un tas binomial (chaque fils est un arbre binomial croissant, et on rétablit la croissance des rangs en inversant la liste). Il ne reste plus qu'à fusionner cette liste avec le tas t' récupéré par la fonction `retire_arbre_min` .
Si n_1 est la taille de l'arbre *amin* et n_2 la taille du tas t', on a une complexité en $\log(n_1) + \log(n_2)$ et comme la taille n de t majore n_1 et n_2, une complexité logarithmique en n.

```
let supprime_min t =
  let Node(_,_,fils), t2 = retire_arbre_min t in
  fusion (List.rev fils) t2
```