TD XXV

LISTES À ACCÈS DIRECT

La première partie sert d'inspiration pour la deuxième, même si elles sont techniquement indépendantes. La troisième a un rôle similaire par rapport à la quatrième.

En informatique, on oppose le stockage *à accès direct* (appelé *random access* en anglais) au stockage *séquentiel*. Le premier permet d'accéder directement à n'importe quelle donnée (c'est le cas de la mémoire vive par exemple, appelée aussi RAM pour *random access memory*) alors que le deuxième impose de lire les données dans un ordre fixé au départ (c'est le cas par exemple des bandes magnétiques, très utilisées encore aujourd'hui pour l'archivage de données).

Dans le domaine des structures de données, une distinction similaire peut être faite entre les structures de type tableau qui permettent d'accéder à un élément quelconque en temps constant et les structures de type liste qui peuvent demander un temps linéaire en leur taille pour accéder à un élément.

On s'intéresse dans ce problème à des structures hybrides, dites *listes à accès direct* (ou en anglais *random access lists*) permettant un accès rapide à un élément quelconque tout en gardant les avantages des listes (en particulier, ajout rapide d'un élément en tête). Ces structures seront *purement fonctionnelles* (aucune modification en place). Pour éviter les confusions, on utilisera « liste » pour désigner les structures que nous définirons et `list` quand on voudra parler d'une liste OCaml usuelle.

Plus précisément, on veut définir un type de données `'a t` doté de la signature suivante, dans laquelle toutes les fonctions doivent être au pire en temps logarithmique en le nombre d'éléments :

```
cons : (u : 'a t) -> (x : 'a) -> 'a t
(* renvoie la liste obtenue en rajoutant x en tête de u *)

head : 'a t -> 'a
(* renvoie l'élément de tête d'une liste (ou une exception si la liste est vide) *)

tail : 'a t -> 'a t
(* renvoie la liste obtenue en éliminant l'élément de tête de l'argument *)

get : (u : 'a t) -> (i : int) -> 'a
(* renvoie l'élément d'indice i de u (les indices commencent à 0) *)

set : (u : 'a t) -> (i : int) -> (x : 'a) -> 'a t
(* renvoie la liste obtenue en remplacant l'élément d'indice i de u par x *)
```

▶ **Question 1** Quelle complexité obtiendrait-on pour ces différentes fonctions si l'on utilisait pour `t` le type `list` de OCaml ? *On ne demande pas d'écrire les fonctions, ni de justifier la réponse.*

1 Nombres en binaire

On rappelle que la représentation binaire d'un entier $n \geqslant 1$ est l'unique liste $a_0, \dots, a_p$ d'éléments de $\{0,1\}$ telle que $a_p \neq 0$ et $n = \sum_{i=0}^{p} a_i 2^i$. Le nombre 0 a lui une représentation vide.

On dira que a_i est le chiffre de *rang* i de n et que son poids est 2^i.

On notera $n = \overline{a_0 \dots a_p}^2$ (chiffre le moins significatif en premier, attention).

Un nombre en binaire se représente de manière naturelle en OCaml par la liste de ses chiffres (moins significatif en premier) :

```
type bit = Z | U
type nombre = bit list
```

On veillera à toujours garder des représentations canoniques : 0 est représenté par la liste vide, toute liste non vide de type `nombre` doit se terminer par un `U`. On pourra supposer que les arguments passés aux fonctions vérifient ces contraintes, il faudra faire en sorte que les résultats en fassent de même.

Les fonctions demandées dans cette partie travaillent uniquement sur les représentations en binaire : à aucun moment on n'essaiera de convertir les `nombre` en des `int`.

▶ **Question 2** Écrire une fonction `succ : nombre -> nombre` prenant la représentation d'un entier n et renvoyant la représentation de $n+1$ (le *successeur* de n).
Déterminer sa complexité en fonction de la longueur de son argument ; on caractérisera le pire des cas.

```
# succ [U; Z; U; U];;
- : nombre = [Z; U; U; U]
```

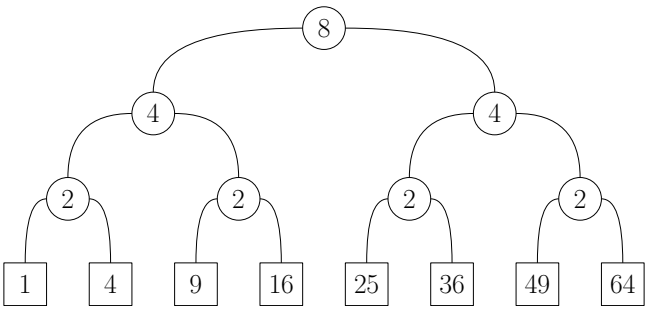
▶ **Question 3** Écrire une fonction `pred : nombre -> nombre` prenant la représentation d'un entier n et renvoyant la représentation de $n-1$ (le *prédécesseur* de n). On renverra une exception *via* `failwith` si l'argument représente 0.
Déterminer sa complexité en fonction de la longueur de son argument ; on caractérisera le pire des cas.

2 Listes binaires à accès direct

On commence par considérer des arbres binaires dont les feuilles sont étiquetées par une valeur de type `'a` et dont les nœuds internes contiennent un champ entier dans lequel on stockera le nombre de feuilles du sous-arbre correspondant (que l'on appellera simplement *taille*).

```
type 'a arbre = F of 'a | N of int * 'a arbre * 'a arbre
```

Les arbres que nous considérerons auront toutes leurs feuilles à la même profondeur : autrement dit, il s'agira d'arbres binaires parfaits, ayant 2^i feuilles si leur hauteur vaut i. À l'intérieur d'un arbre, on considère que les feuilles sont ordonnées de gauche à droite. Ainsi, la liste $(1, 4, 9, 16, 25, 36, 49, 64)$ peut est représentée par l'arbre suivant (de hauteur 3 et de taille $2^3 = 8$) :



Remarque

Pour les fonctions `get` et `set` demandées ci-dessous, on demande des implémentations efficaces, c'est-à-dire tirant partie des informations portées par les nœuds internes pour éviter de parcourir tout l'arbre.

▶ **Question 4** Écrire une fonction `get_arbre : (u : 'a arbre) -> (i : int) -> 'a` qui renvoie l'élément d'indice i de l'arbre u.
Par exemple, si u est l'arbre dessiné ci-dessus, `get_arbre u 2` doit renvoyer 9 et `get_arbre u 7` renvoyer 64.
On renverra une exception `failure "depassement"` (par `failwith "depassement"`) si i n'est pas un indice valide pour u.

▶ **Question 5** Déterminer la complexité de `get_arbre` en fonction de la taille n de u. *On rappelle que* u *est supposé parfait.*

▶ **Question 6** Écrire une fonction `set_arbre : (u : 'a arbre) -> (i : int) -> (x : 'a) -> 'a arbre` qui renvoie l'arbre obtenu en remplaçant l'élément d'indice i de u par x. Donner sa complexité en temps.

▶ **Question 7** Si u est de taille n et si v est le résultat du calcul de `set_arbre u k x` pour un certain k et un certain x, quelle quantité de mémoire supplémentaire faut-il pour stocker v tout en conservant u ? On donnera la réponse en considérant qu'un nœud interne ou une feuille occupe une quantité constante C de mémoire, et l'on fera un schéma.

Une *liste binaire à accès direct* (on écrira simplement *liste binaire*) de longueur n sera constituée d'un arbre du type que nous venons de définir pour chaque chiffre 1 dans la représentation binaire de n. Si le chiffre 1 considéré a un poids de 2^i (*i.e.* s'il s'agit du chiffre de rang i), l'arbre correspondant aura une taille de 2^i. On a donc les types suivants :

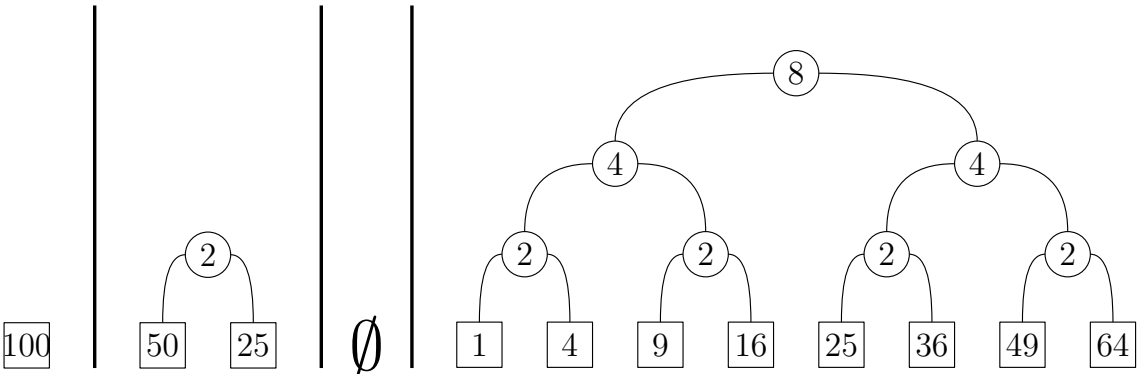
```
type 'a chiffre = Ze | Un of 'a arbre
type 'a liste_binaire = 'a chiffre list
```

À l'intérieur d'une `'a chiffre list`, les arbres seront classés par taille croissante, et l'ordre induit sur les feuilles sera de gauche à droite dans la liste, puis de gauche à droite à l'intérieur de chaque arbre.

Si u est une liste binaire :

- on appellera *taille* de u, et l'on notera $|u|$, la somme des tailles des arbres qui la composent, c'est-à-dire le nombre total de feuilles qu'elle contient ;
- on appellera *longueur* de u, et l'on notera $\texttt{longueur}(u)$, le nombre de chiffres qu'elle contient, c'est-à-dire $\texttt{longueur}(u) =$ `List.length u`.

Par exemple, la liste $(100, 50, 25, 1, 4, 9, 16, 25, 36, 49, 64)$, de longueur $11 = 2^0+2^1+2^3$, sera représentée par la liste binaire suivante (de longueur 4 et de taille 11) :



Cette liste pourrait être définie en OCaml par :

```
(* Les trois arbres : *)
let a = F 100
let b = N(2, F 50, F 25)
let c = N(8,
          N(4, N(2, F 1, F 4), N(2, F 9, F 16)),
          N(4, N (2, F 25, F 36), N(2, F 49, F 64)))
(* et la liste : *)
let li = [Un a; Un b; Ze; Un c]
```

Par commodité, on définit une fonction donnant la taille d'un arbre :

```
let size = function
  | F _ -> 1
  | N (n, _, _) -> n
```

▶ **Question 8** Soit u une liste binaire non vide, $n > 0$ sa taille et k sa longueur. Exprimer k en fonction de n (on justifiera).

▶ **Question 9** Écrire une fonction `get : (li : 'a liste_binaire) -> (i : int) -> 'a` renvoyant l'élément d'indice `i` de `li`. On utilisera la fonction `get_arbre` écrite plus haut.
Avec le `li` défini plus haut, on doit avoir `get li 3 = 1`.

▶ **Question 10** Donner en fonction de $|u|$ la complexité dans le pire des cas de `get u k`. Peut-on garantir l'exécution en temps constant si $k = 0$?

▶ **Question 11**
Écrire une fonction `set : (li : 'a liste_binaire) -> (i : int) -> (x : 'a) -> 'a liste_binaire` renvoyant la liste binaire obtenue en remplaçant l'élément d'indice `i` de `li` par `x`. Donner la complexité de cette fonction.

▶ **Question 12** Écrire une fonction `cons : (li : 'a liste_binaire) -> (x : 'a) -> 'a liste_binaire` renvoyant la liste binaire obtenue en rajoutant `x` en tête de `li`.
On pourra s'aider d'une fonction `cons_arbre : 'a liste_binaire -> 'a arbre -> 'a liste_binaire` inspirée du `succ` de la partie précédente.
Avec le `li` précédent, on doit avoir

```
# cons li 12;;
- : int chiffre list =
[Ze;
 Ze;
 Un (N (4, N (2, F 12, F 100), N (2, F 50, F 25)));
 Un
  (N (8, N (4, N (2, F 1, F 4), N (2, F 9, F 16)),
    N (4, N (2, F 25, F 36), N (2, F 49, F 64))))]
```

▶ **Question 13** Déterminer la complexité de la fonction `cons` dans le pire des cas et dans le meilleur des cas.

▶ **Question 14** Écrire une fonction `uncons : (li : 'a liste_binaire) -> 'a * 'a liste_binaire` qui renvoie l'élément en tête de `li` ainsi que la liste binaire obtenue en retirant cet élément de `li`.
Donner la complexité de cette fonction dans le pire et dans le meilleur des cas.

▶ **Question 15** En déduire les fonctions `head` et `tail` spécifiées au début du problème, ainsi que leur complexité.

3 Nombres en binaire décentré

Dans le système en binaire décentré (*skewed binary* en anglais), les chiffres peuvent prendre les valeurs $0, 1$ et 2, et le chiffre de rang i a pour poids $r_i = 2^{i+1} - 1$. Autrement dit, une représentation décentrée est une liste $(a_0, \dots, a_p)$ d'éléments de $\{0, 1, 2\}$ et le nombre correspondant est $n = \sum_{i=0}^{p} a_i r_i$. On pourra alors noter $n = \overline{a_0 \dots a_p}^d$.

Ainsi, $21 = \overline{0201}^d = \overline{122}^d$. Comme on peut le voir, la représentation décentrée n'est pas unique. Cependant, on peut définir une représentation canonique : une représentation décentrée est dite canonique si elle ne contient pas de 2 ou si son seul chiffre égal à 2 est le chiffre non nul de rang le plus petit. La représentation canonique de 21 est alors $\overline{0201}$ (bien sûr, on impose toujours qu'il n'y ait pas de zéros à la fin de la représentation).

On notera l'identité suivante, utile dans plusieurs questions : $r_{i+1} = 2r_i + 1$.

▶ **Question 16** *Cette question, un peu délicate, n'a pas vraiment de rapport avec le reste du sujet. Je vous encourage plutôt à la chercher chez vous.*
Montrer que tout entier naturel a une unique représentation décentrée canonique.

Dans la suite du problème, on supposera toujours que les représentations utilisées sont canoniques (et il faudra faire en sorte que les représentations crées par les fonctions que l'on demande d'écrire le soient aussi).

L'intérêt principal de la représentation décentrée est que les opérations `succ` et `prec` peuvent se faire sans propagation de retenue.

▶ **Question 17** Donner la représentation canonique des entiers de 6 à 15.

▶ **Question 18** Expliquer comment calculer le successeur et le prédécesseur d'un nombre binaire décentré en ne changeant la valeur que de deux chiffres au maximum.

▶ **Question 19** On suppose à cette question (uniquement) que l'on représente les nombres en binaire décentré à l'aide du type suivant, en stockant le chiffre le moins significatif en tête de liste :

```
type chiffre_decentre = ZZ | UU | DD
type nombre_decentre = chiffre_decentre list
```

Expliquer pourquoi la méthode vue à la question précédente ne permet pas d'écrire des fonctions `succ` et `pred` en temps constant.

Pour éviter le problème soulevé à la question précédente, nous allons utiliser une représentation *creuse* pour les nombres en binaire décentré : au lieu de stocker tous les chiffres, nous allons stocker la liste des poids des chiffres non nuls. Cette liste sera en ordre strictement croissant, sauf que le premier élément pourra être présent deux fois, ce qui sera notre manière de marquer la présence d'un 2.
On définit donc le type `decentre` des représentations décentrées par :

```
type decentre = int list
```

Ainsi, 21 sera représenté par la liste `[3; 3; 15]` alors que 41 sera représenté par `[3; 7; 31]`.

▶ **Question 20** Écrire une fonction `succ_d : decentre -> decentre` calculant le successeur d'un entier décentré **en temps constant.**

▶ **Question 21** Écrire une fonction `pred_d : decentre -> decentre` calculant le prédécesseur d'un entier décentré **en temps constant.** On renverra une exception si l'argument représente 0.

Listes en binaire décentré

Nous avons vu plus haut que la représentation des séquences par des listes binaires ne permet pas de conserver des opérations `cons`, `head` et `tail` en temps constant comme pour les `list` usuelles.
Pour obtenir ces performances, on modifie la structure de liste binaire en utilisant la représentation décentrée creuse. De plus, on stocke à présent les éléments de la séquence dans les nœuds internes aussi bien que dans les feuilles : par conséquent, la taille d'un arbre est désormais le nombre total de nœuds et non plus seulement le nombre de feuilles. On ne stocke plus cette taille à chaque nœud, mais seulement la taille de chacun des arbres constituant la liste en binaire décentré. On obtient donc le type suivant :

```
type 'a arbre_decentre = Fd of 'a | Nd of 'a * 'a arbre_decentre * 'a arbre_decentre
type 'a liste_decentre = (int * 'a arbre_decentre) list
```

Une liste en binaire décentré contenant 21 éléments de type `'a` sera donc de la forme

```
[(3, u); (3, v); (15, w)]
```

où `u`, `v` et `w` sont des `'a arbre_decentre` de tailles respectives 3, 3 et 15.

▶ **Question 22** Pour avoir une chance d'implémenter `head` en temps constant, dans quel ordre faut-il stocker les étiquettes à l'intérieur d'un arbre ? *On admettra que les seuls candidats raisonnables à considérer sont les ordres préfixe, infixe et suffixe.*
Donner explicitement (en dessinant les arbres) la liste en binaire décentré correspondant à la séquence $(1, 2, \ldots, 13)$.

▶ **Question 23** Écrire la fonction `get_d : 'a liste decentre -> int -> 'a` et préciser (sans justification) sa complexité.

▶ **Question 24** Écrire la fonction `cons_d : 'a liste_decentre -> 'a -> 'a liste_decentre`. On exige que cette fonction s'exécute en temps constant.

▶ **Question 25** Écrire la fonction `head_d : 'a liste_decentre -> 'a`. On exige que cette fonction s'exécute en temps constant.

▶ **Question 26** Écrire la fonction `tail_d : 'a liste_decentre -> 'a liste_decentre`. On exige que cette fonction s'exécute en temps constant.

Ce problème est directement inspiré de Purely Functional Data Structures, *Chris Okasaki, Cambridge University Press. La thèse de Chris Okasaki (dont l'ouvrage cité est une version légèrement enrichie) est librement disponible en ligne ; elle porte le même titre et fournira une excellente lecture de vacances aux plus motivé·e·s d'entre vous.*

Solutions

▶ **Question 1** On aurait les complexités temporelles suivantes :

- `cons`, `head` et `tail` en $\mathcal{O}(1)$;
- `get` et `set` en $\mathcal{O}(i)$ (et donc $\mathcal{O}(|u|)$).

Nombres en binaire

▶ **Question 2**

```
let rec succ : nombre -> nombre = function
  | [] -> [U]
  | Z :: xs -> U :: xs
  | U :: xs -> Z :: succ xs
```

On parcourt toute la liste u jusqu'à trouver un `Z` (ou arriver à la fin), la complexité est donc en $\mathcal{O}(|u|)$ dans le pire des cas, qui correspond à une liste constituée uniquement de `U`.

▶ **Question 3**

```
let rec pred : nombre -> nombre = function
  | [] -> failwith "zero moins un"
  | [U] -> []
  | Z :: xs -> U :: pred xs
  | U :: xs -> Z :: xs
```

La situation est symétrique de celle pour `succ` : la complexité est en $\mathcal{O}(|u|)$ et le pire cas correspond à une liste constituée uniquement de `Z` (sauf un `U` en dernière position pour assurer le caractère canonique).

Listes binaires à accès direct

▶ **Question 4**

```
let rec get_arbre u n =
  match u with
  | F x -> if n = 0 then x else failwith "depassement"
  | N (p, ga, dr) ->
     if n < p / 2 then
       get_arbre ga n
     else
       get_arbre dr (n - p / 2)
```

▶ **Question 5** On descend dans l'arbre le long d'un chemin reliant la racine à une feuille. Les opérations effectuées à chaque nœud étant en temps constant, le temps de calcul est proportionnel à la longueur de ce chemin (*i.e.* à la profondeur de la feuille).

L'arbre étant parfait, ses 2^i feuilles sont situées à profondeur i : autrement dit, si n est la taille de l'arbre, alors la longueur du chemin est exactement $\log_2 n$. La complexité de `get_arbre` est donc en $\mathcal{O}(\log n)$.

▶ **Question 6**

```
let rec set_arbre u n x =
  match u with
  | F y -> if n = 0 then F x else failwith "depassement"
  | N (p, ga, dr) ->
     if n < p / 2 then
       N (p, set_arbre ga n x, dr)
     else
       N (p, ga, set_arbre dr (n - p / 2) x)
```

On obtient de même une complexité en $\mathcal{O}(\log n)$.

▶ **Question 7** On copie tous les nœuds situés sur le chemin reliant la racine à la feuille « modifiée », le reste de l'arbre est partagé entre les deux versions. Comme dit plus haut, il y a $\log_2 n$ nœuds sur ce chemin, et la quantité d'espace supplémentaire consommée par la coexistence des deux versions est donc de $C \log_2 n$.

▶ **Question 8** La longueur k de la liste est par définition le nombre de chiffres de l'écriture binaire de n. Or n s'écrit avec k chiffres si et seulement si $2^{k-1} \leqslant n < 2^k$, c'est-à-dire $k - 1 \leqslant \log_2 n < k$ et donc $k = 1 + \lfloor \log_2 n \rfloor$.

▶ **Question 9**

```
let rec lookup liste n =
  match liste with
  | [] -> failwith "depassement"
  | Ze :: xs -> lookup xs n
  | Un t :: xs -> if n < size t then (t, n)
                  else lookup xs (n - size t)

let get liste n =
  let t, i = lookup liste n in
  get_arbre t i
```

▶ **Question 10** Notons n la taille de u et k sa longueur. On commence par faire un appel à `lookup` qui parcourt u (en entier si la valeur cherchée est dans le dernier arbre). Cet appel prend donc au pire un temps proportionnel à k, c'est-à-dire à $\log n$ d'après la question 8.

On fait ensuite un appel à `get_arbre t`, où `t` est l'un des arbres de la liste. Cet appel prend un temps proportionnel à $\log |t|$ d'après 5, or comme $n = \sum_{t \in u} |t|$, on a $\log |t| \leqslant \log n$.

Finalement, la complexité dans le pire des cas de `get u k` est en $\mathcal{O}(\log |u|)$.

On ne peut pas garantir une complexité en $\mathcal{O}(1)$ si $k = 0$ car il faut parcourir la liste jusqu'à trouver un arbre : c'est immédiat si n est impair mais peut prendre un temps proportionnel à $\log n$ si n est une puissance de 2.

▶ **Question 11**

```
let rec set liste n x =
  match liste with
  | [] -> failwith "depassement"
  | Ze :: ts -> Ze :: set ts n x
  | Un t :: ts when size t > n -> Un (set_arbre t n x) :: ts
  | Un t :: ts -> Un t :: set ts (n - size t) x
```

On a à nouveau une complexité en $\mathcal{O}(\log n)$.

▶ **Question 12**

- La fonction `link` fusionne deux arbres supposées de même taille 2^i pour en former un de taille 2^{i+1} (les feuilles héritées du premier arbre se retrouvant à gauche de celles héritées du deuxième).
- La fonction `cons_tree` prend une liste **qui commence au chiffre de poids** 2^i et un arbre de taille 2^i et le « rajoute à la liste » : si le chiffre de poids 2^i vaut zéro, on met l'arbre à cette place, sinon on le fusionne avec l'arbre présent, le chiffre passe à zéro et l'on « propage la retenue ».
- Pour ajouter un élément `x` à une liste, il suffit alors de créer une feuille `F x` et d'appeler `cons_tree`.

```
let link u t =
   N (size u + size t, u, t)

let rec cons_tree u t =
  match u with
  | [] -> [Un t]
  | Ze :: xs -> Un t :: xs
  | Un t' :: xs -> Ze :: cons_tree xs (link t t')

let rec cons u x = cons_tree u (F x)
```

▶ **Question 13** La complexité dépend uniquement du nombre d'appels récursifs à `cons_tree` puisque toutes les opérations (y compris `link`) se font en temps constant. Ce nombre d'appels est égal au nombre de `Un t` présents en début de liste (essentiellement, la question est de savoir combien de temps on doit propager la retenue) :

- dans le meilleur des cas (n est pair, le premier chiffre est nul), l'exécution sera $\boxed{\text{en temps constant}}$;
- dans le pire des cas (n de la forme $2^k - 1$, tous les chiffres valent un), on aura un temps proportionnel à k, donc en $\boxed{\mathcal{O}(\log n).}$

▶ **Question 14**

- La fonction `uncons_tree`, appelée sur une liste binaire dont le premier chiffre est de poids 2^i, renvoie l'arbre de taille 2^i contenant les 2^i premières feuilles de la liste binaire, ainsi que la liste binaire contenant les $n - 2^i$ dernières étiquettes. L'idée est similaire à celle de la fonction `pred`.
- La fonction `uncons` traite le cas particulier de la liste de taille 1 (nécessaire car une liste binaire de taille 0 s'écrit `[]` et non `[Ze]`), et appelle sinon `uncons_tree` pour récupérer la feuille contenant le premier élément et la « queue » de la liste.
- La complexité dépend uniquement du nombre d'appels récursifs à `uncons_tree`, qui est égal au nombre de zéros en tête de la liste.
  Dans le meilleur des cas (n impair, le premier chiffre vaut 1), on aura donc une complexité $\boxed{\text{en } \mathcal{O}(1)}$, et dans le pire ($n$ est une puissance de deux, tous les chiffres sauf le dernier valent zéro) une complexité $\boxed{\text{en } \mathcal{O}(\log n).}$

```
let rec uncons_tree u =
  match u with
  | [] -> failwith "depassement"
  | Un t :: ts -> (t, Ze :: ts)
  | Ze :: ts ->
     let y, ys = uncons_tree ts in
     match y with
     | N (_, ga, dr) -> (ga, Un dr :: ys)
     | _ -> failwith "impossible"

let uncons u =
  match u with
  | [Un (F x)] -> x, []
  | _ ->
     match uncons_tree u with
     | F x, ts -> x, ts
     | _ -> failwith "impossible"
```

▸ **Question 15**

```
let head u = fst (uncons u)
let tail u = snd (uncons u)
```

Nombres en binaire décentré

▸ **Question 16**

Existence On montre par récurrence la propriété suivante : « si $0 \leqslant x < r_i = 2^{i+1} - 1$, alors x a une représentation canonique n'utilisant que des chiffres de rang strictement inférieur à i ».

- Pour $i = 0$, on a $x = 0$ (représentation vide).
- Soit $x < r_{i+1}$, on cherche une représentation canonique sans chiffre de rang supérieur ou égal à $i+1$.
  Si $x < r_i$, alors il a une représentation canonique (dont les chiffres sont de rang $< i$) par hypothèse de récurrence.
  Si $x = r_{i+1} - 1$, alors $x = 2r_i$ d'après la remarque de l'énoncé, ce qui fournit une représentation canonique convenable.
  Sinon, on a $r_i \leqslant x < r_{i+1} - 1$. En posant $y = x - r_i$, on a $0 \leqslant y < r_{i+1} - 1 - r_i$, c'est-à-dire $0 \leqslant y < r_i$. On dispose donc d'une représentation canonique de y sans chiffre de rang $\geqslant i$, ce qui fournit une représentation canonique convenable de x en ajoutant un 1 comme chiffre de rang i.

Unicité Comptons les représentations canoniques n'utilisant que des chiffres de rang $0 \dots i-1$.

- Il y a la représentation vide (aucun chiffre non nul).
- Pour chaque k de $[0 \dots i-1]$, il y a $2 \times 2^{i-1-k}$ représentations dont le premier chiffre non nul est celui de rang k. En effet, $a_k \in \{1, 2\}$ (deux choix) et chacun des a_j avec $k < j < i$ est dans $\{0, 1\}$ (deux choix à chaque fois, donc 2^{i-k-1} au total).

On a donc $1 + \sum_{k=0}^{i-1} 2^{i-k} = 1 + \sum_{j=1}^{i} 2^j = 1 + 2^{i+1} - 2 = r_i$ représentations canoniques n'utilisant que des chiffres de rang strictement inférieurs à i. Or, d'après l'existence, l'ensemble des valeurs correspondantes contient les r_i entiers de $[0, r_i - 1]$. On en déduit l'unicité.

▸ **Question 17** On a :

- $6 = 3 + 3 = \overline{02}^d$
- $7 = 7 = \overline{001}^d$
- $8 = 1 + 7 = \overline{101}^d$
- $9 = 1 + 1 + 7 = \overline{201}^d$
- $10 = 3 + 7 = \overline{011}^d$
- $11 = 1 + 3 + 7 = \overline{111}^d$
- $12 = 1 + 1 + 3 + 7 = \overline{211}^d$
- $13 = 3 + 3 + 7 = \overline{021}^d$
- $14 = 7 + 7 = \overline{002}^d$
- $15 = 1 + 7 + 7 = \overline{0001}^d$

▶ **Question 18**
Zéro n'a pas de prédécesseur et son successeur est 1, il n'y a pas de problème. Soit $x \in \mathbb{N}^*$, et $\overline{a_0 \dots a_k}^d$ sa représentation canonique. Soit également a_j son premier chiffre non nul.

Successeur

- Si $a_j \neq 2$, alors il n'y a aucun 2 dans la représentation (car elle est canonique), et $x+1 = x + r_0 = \overline{b_0 \dots b_k}$ où $b_0 = a_0 + 1$ et $b_i = a_i$ si $i \geqslant 1$. La représentation donnée pour $x+1$ est bien canonique, b_0 étant le seul chiffre potentiellement égal à 2.
- Si $a_j = 2$, alors c'est le seul 2 dans la représentation. Notons y le nombre obtenu à partir de x en remplaçant a_j par $b_j = 0$ et a_{j+1} par $b_{j+1} = 1 + a_{j+1}$.
  — La représentation obtenue pour y est canonique : on avait $a_{j+1} \in \{0,1\}$, donc $b_{j+1} \in \{1,2\}$; les autres chiffres, inchangés, sont dans $\{0,1\}$.
  — $y = x - 2r_i + r_{i+1} = x+1$ d'après la remarque de l'énoncé.

On a donc obtenu le successeur de x en changeant 1 ou 2 chiffres de le représentation.

Prédécesseur À nouveau, on considère le premier chiffre non nul de (la représentation canonique de) x. C'est le seul potentiellement égal à 2.

- S'il s'agit de a_0, on le diminue de 1 : on obtient bien une représentation canonique de $y = x - 1$.
- S'il s'agit d'un a_i avec $i > 0$, on remplace a_i par $b_i = a_i - 1$ et a_{i-1} par $b_{i-1} = 2$. On obtient une représentation canonique (b_{i-1} vaut 2 mais b_i ne peut plus valoir 2) et $y = x + 2r_{i-1} - r_i = x - 1$.

Le prédécesseur s'obtient en changeant 1 ou 2 chiffres.

▶ **Question 19** Les deux méthodes données plus haut nécessiteraient de parcourir la liste pour trouver le premier chiffre non nul, qui peut être arbitrairement loin.

▶ **Question 20**

```
let succ_d :  = function
  | x :: y :: xs when x = y -> (x + y + 1) :: xs
  | xs -> 1 :: xs
```

▶ **Question 21**

```
let pred_d = function
  | [] -> failwith "zero moins un"
  | 1 :: xs -> xs
  | x :: xs -> let y = x / 2 in y :: y :: xs
```

Listes en binaire décentré

▶ **Question 22** On veut pouvoir renvoyer l'élément de rang 0 de la liste en temps constant. Il doit donc se trouver à la racine du premier arbre, ce qui ne sera le cas que si l'on stocke les étiquettes en ordre préfixe.
Pour la liste $(1, \dots, 13)$, on obtiendra

```
[(3, Nd (1, Fd 2, Fd 3));
 (3, Nd (4, Fd 5, Fd 6));
 (7, Nd (7,
         Nd (8, Fd 9, Fd 10),
         Nd (11, Fd 12, Fd 13)))]
```

▶ **Question 23**

```
(* Il faut connaître la taille (nombre total de noeuds) de l'arbre dans
   lequel on fait la recherche, et déterminer si le noeud numéro i dans
   l'ordre préfixe se trouve à la racine, à gauche ou à droite. *)

let rec get_tree_d t i taille =
  match t with
  | Fd x when i = 0 -> x
  | Nd (x, _, _) when i = 0 -> x
  | Nd (_, ga, _) when i <= taille / 2 -> get_tree_d ga (i - 1) (taille / 2)
  | Nd (_, _, dr) -> get_tree_d dr (i - 1 - taille / 2) (taille / 2)
  | _ -> failwith "get_tree_d : depassement"

let rec get_d u i =
  match u with
  | [] -> failwith "get_d : depassement"
  | (taille, t) :: xs ->
    if i < taille then get_tree_d t i taille
    else get_d xs (i - taille)
```

On a une complexité logarithmique en la taille de la liste.

▶ **Question 24**

```
let cons_d liste x =
  match liste with
  | (p1, t1) :: (p2, t2) :: ts when p1 = p2 ->
     (1 + p1 + p2, Nd (x, t1, t2)) :: ts
  | _ -> (1, Fd x) :: liste
```

▶ **Question 25**

```
let head  = function
  | (_, Fd x) :: ts -> x
  | (_, Nd (x, _, _)) :: ts -> x
  | [] -> failwith "head : empty list"
```

▶ **Question 26**

```
let tail = function
  | (1, Fd x) :: ts -> ts
  | (p, Nd (x, t1, t2)) :: ts -> (p / 2, t1) :: (p / 2, t2) :: ts
  | _ -> failwith "tail : empty list"
```