Cordes (d'après X 2008)

1 Structure de corde

1. Si c'est une feuille, on prendra la longueur de la chaîne, si c'est un nœud, l'information est directement disponible incluse :

```
let length = function
  | L s -> String.length s
  | N (n, _, _) -> n
```

2. Il suffit d'utiliser le constructeur `L` !

```
let create s = L s
```

3. Au tour du constructeur `N`, en prenant soin de déterminer la longueur du résultat, **mais aussi de respecter l'invariant imposé : `L ""` ne doit pas se retrouver comme feuille d'une corde non vide !**

```
let concat s1 s2 =
  if s1 = L "" then s2 else
  if s2 = L "" then s1 else
    N (length s1+length s2, s1, s2)
```

Cette fonction sera fort utile dans la suite, on n'utilisera le constructeur `N` que si l'on est *certain* que les deux sous-arbres sont non-vides, sinon `concat` nous permettra d'assurer l'invariant sans effort.

4. Si c'est un nœud, selon la taille du sous-arbre gauche, on décide de descendre d'un côté ou de l'autre. Si c'est une feuille, on cherche directement dans la chaîne.

```
let rec nth i = function
  | L s -> s.[i]
  | N (_, lc, _) when i < length lc -> nth i lc
  | N (_, lc, rc) -> nth (i - length lc) rc
```

Attention au caractère strict de la comparaison : les caractères de `lc` se trouvent à des index variant entre 0 et $n_l - 1$ inclus, où n_l est la longueur de `lc`. Le caractère d'index n_l est le *premier* de `rc`.

Notons qu'en général on évite de faire plusieurs fois le même appel avec les mêmes paramètres. Ici, ce n'est pas gênant exceptionnellement pour « `length lc` » car la fonction est en temps constant et que le résultat est déterministe.

5. Un peu plus difficile. On peut commencer par écrire une fonction qui fait la même chose sur une chaîne de caractères [1] :

```
let index_string c s =
  let rec search = function
    | i when i = String.length s -> -1
    | i when s.[i] = c -> i
    | i -> search (i+1)
  in search 0
```

Puis l'utiliser pour les feuilles d'une corde, et si on a affaire à un nœud, on essaie d'abord le côté gauche, et en cas d'échec le côté droit :

```
let rec index c = function
  | L s -> index_string c s
  | N (_, lc, rc) -> match index c lc with
                     | -1 -> (match index c rc with
                              | -1 -> -1
                              | i -> i + length lc)
                     | i -> i
```

On évitera d'utiliser `nth` pour des valeurs croissantes d'index jusqu'à trouver un caactère égal à `c`, si la solution fonctionne, elle a une complexité bien moins bonne (dans le pire des cas produit de la hauteur et de la longueur de la corde !) Il n'est *pas toujours* attendu que la question n serve dans la question $n+1$, il faut toujours avoir du recul sur les coûts.

6. Si on a affaire à un nœud, on regarde si l'on a besoin d'au moins un caractère du côté droit. Attention encore ici à ne pas se retrouver avec des feuilles `L ""` !

```
let rec prefix p = function
  | L s -> L (String.sub s 0 p)
  | N (_, lc, _) when p <= length lc -> prefix p lc
  | N (_, lc, rc) -> N(p, lc, prefix (p - length lc) rc)
```

Ici, on peut utiliser sans crainte `N` et non `concat` pour le dernier cas car `p - length lc > 0`. On peut envisager un cas supplémentaire, à placer entre le premier et le second cas :

```
  | N (_, lc, _) when p = length lc -> lc
```

1. Ou utiliser `String.index`, mais attention, elle lève une exception en cas d'absence, qu'il faudra gérer correctement.

On évite de poursuivre la descente dans l'arbre, mais cette situation ne se présentera pas si fréquemment, et le résultat obtenu est de toute façon le même, même si l'on ne réutilise pas certains nœuds interne. De même, on peut ajouter en tête

```
  | L s when String.length s = p -> L s
```

ce qui évite techniquement de copier une chaîne (et respecte l'envie du sujet de récupérer le maximum de feuilles originales). Mais ce n'était pas réellement attendu ici.

7. La fonction `suffix` est fort similaire à la précédente, en échangeant le rôle des deux côtés. Mais attention à bien lire le sujet, l'argument `p` n'est pas la longueur du suffixe demandé, mais l'index de son premier caractère ! Comme précédemment, on s'assurera que les sous-arbres sont non vides. On prendra garde aussi au fait que le troisième paramètre de `String.sub` est une longueur et non un index de fin. Cela donne par exemple :

```
let rec suffix p = function
  | L s -> L (String.sub s p (String.length s - p))
  | N (s, lc, rc) when p<length lc -> N (s-p, suffix p lc, rc)
  | N (_, lc, rc) -> suffix (p-length lc) rc
```

8. Pour la fonction sub, on utilise les deux fonctions précédentes [2] :

```
let sub i m r = prefix m (suffix i r)
```

9. Les fonctions `prefix` et `suffix` ont une complexité en temps dans le pire des cas en la hauteur de la corde `r` fournie en argument ($O(\text{hauteur}(r))$), plus le coût d'un (unique) appel à `String.sub`. Dans le pire des cas, cet appel coût $O(p)$ pour `prefix`, $O(n-p)$ pour `suffix`.

La complexité de sub est donc $O(\text{hauteur}(r)+(n-i)+m)$ où n est la longueur de la corde. Fort heureusement, dans la plupart des situations, les feuilles à « couper » sont des chaînes de caractères de longueur bien plus petite que la corde, et la complexité, en $O\big(\text{hauteur}(r)+\text{longueur}(f_1)+\text{longueur}(f_2)\big)$ où f_1 et f_2 sont les feuilles sur lesquelles agissent `String.sub`, sera bien moindre.

2 Équilibrage

10. On a un coût $2\times1+4\times3+1\times3+6\times4+2\times4+3\times3=58$.

11. Le coût d'une corde réduite à une feuille est nul. Le coût d'une corde qui n'est pas une corde est égal à sa longueur plus le coût de ses sous-cordes droite et gauche (en effet, chaque caractère s'éloigne de 1 de la racine à chaque concaténation).

On a donc, fort simplement :

```
let rec cost = function
  | L _ -> 0
  | N (n, lc, rc) -> n + (cost lc + cost rc)
```

Une solution peut-être plus facile à trouver utilise une fonction auxiliaire prenant en premier argument la profondeur de la racine du sous-arbre qu'on lui passe en second argument :

```
let cost r =
  let rec cost_aux depth = function
    | L s -> depth * String.length s
    | N (_, lc, rc) -> cost_aux (d+1) lc + cost_aux (d+1) rc
  in cost_aux 0 r
```

12. Comme souvent, le sujet vous invite à effectuer un essai de l'algorithme à la main. C'est l'occasion de s'assurer que l'on a bien compris son fonctionnement, et de clarifier les détails (d'autant que dans le sujet original, l'algorithme était décrit de façon moins précise, notamment concernant l'ordre dans lequel les concaténations sont faites).

La première feuille fusionne avec la corde vide de la case 2, et comme elle est trop grande pour y rester, fusionne avec la corde vide de la case 3 et s'y installe :

0	1	2	3	4	5	6	7	8	9
			Le						

La seconde feuille progresse de la même façon, fusionne avec la première feuille, et l'algorithme conduit à la formation d'un arbre dans la case 5 (attention, il s'agit bien d'un arbre, la concaténation étant à prendre au sens des concaténations de cordes) :

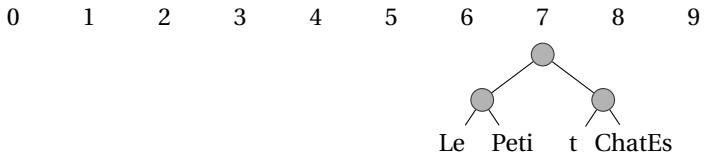
0	1	2	3	4	5	6	7	8	9
									

La troisième feuille se glisse dans la case 2 :

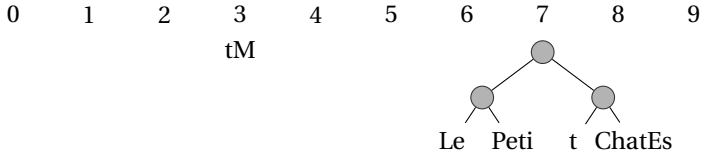
0	1	2	3	4	5	6	7	8	9
		t							

2. Le sujet original demandait sub directement sans demander les deux fonctions `prefix` et `suffix`. Cela étant dit, tenter d'écrire la fonction directement va nécessiter bien plus de cas, et peut vite conduire à des erreurs, et il était de toute façon préférable de découper cette fonction en plusieurs étapes, cela ne change de toute façon pas la complexité.

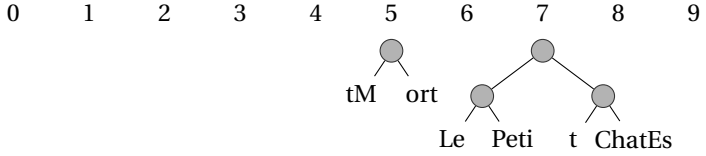
La quatrième conduit, après plusieurs fusions, à la formation d'un arbre dans la case 7 :



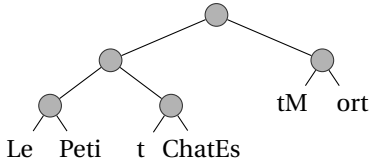
La cinquième feuille finit dans la case 3 :



La sixième feuille produit un arbre dans la case 5 :



Enfin, on reconstruit l'intégralité de l'arbre, ce qui donne :



On remarque que le nouvel arbre possède bien les mêmes six feuilles que l'arbre original (ce qui était le but), et qu'il correspond à la même chaîne de caractères, les feuilles ayant juste été arrangées spatialements pour que les caractères soient plus proches de la racine en moyenne.

13. On utilise une simple boucle (on peut omettre la première ligne car le tableau est initialisé avec des 0) :

```
let init_Fib () =
  fib.(0) <- 0;   (* peut être omis *)
  fib.(1) <- 1;
  for i = 2 to max_size do
    fib.(i) <- fib.(i-1) + fib.(i-2)
  done
```

14. On applique simplement l'algorithme proposé à la lettre. Attention de bien utiliser `concat` pour éviter des feuilles avec `L ""`, et au sens des concaténations.

```
let rec insert r i =
  let c = concat file.(i) r in
  if length c < fib.(i) then file.(i) <- c
                        else (file.(i) <- L ""; insert c (i+1))
```

15. Si l'on regarde attentivement ce qui se passe dans l'algorithme proposé pour la fonction `insert`, on remarque que l'on ne peut mettre que deux choses dans une case d'index i du tableau `file` :

- une feuille `L ""`
- une corde dont la longueur est dans l'intervalle $[F_i, F_{i+1}[$.

Par conséquent, puisque le tableau ne contient que des feuilles vides initialement, après un nombre quelconque d'appels à la fonction `insert`, dans toute case d'index i, on trouve nécessairement soit une feuille vide, soit une corde dont la longueur est dans l'intervalle $[F_i, F_{i+1}[$, puisque la fonction `insert` ne place dans les cases du tableau que des `L ""` ou des cordes vérifiant cette propriété.

Le second invariant, sur la hauteur, est beaucoup plus délicat à justifier. On tente d'insérer une des feuilles de la corde à équilibrer en commençant par la case 2. Supposons que l'on finisse par réussir à placer une corde dans la case d'index j. Toutes les cases d'index 2 à $j-1$ contiennent alors `L ""`, ce qui vérifie l'invariant. Toutes les cases d'index strictement supérieur à j n'ont pas vu leur contenu changer, donc respectent l'invariant. Il faut donc juste vérifier que la hauteur de la corde qui finit en j est compatible avec l'invariant, donc a une hauteur majorée par $j-2$.

La corde que l'on essaie successivement de faire entrer dans chaque case k devient progressivement de plus en plus haute. Toutefois, on peut vérifier, par récurrence, que la corde obtenue après la concaténation avec le contenu de la case k est de hauteur au plus $k-1$.

Pour l'initialisation de la récurrence, notons que l'on part bien d'une corde de hauteur au plus 1 après la fusion éventuelle avec le contenu de la case 2. En effet il s'agit

- soit de la feuille initiale que l'on cherche à insérer si la case 2 contenait `L ""` et donc une corde de hauteur 0 ;
- soit du résultat de la concaténation avec la corde de la case 2, laquelle ne peut être qu'une feuille, car seule une feuille peut être de longueur 1, la seule longueur permise dans la case 2, donc nécessairement une corde de hauteur 1.

Pour la récurrence proprement dite, constatons que la corde que l'on tente d'insérer dans la case k est le résultat de la concaténation de la corde de hauteur au plus $k-2$ qui se trouvait précédemment dans cette case k et de la corde de hauteur au plus $(k-1)-1$ qu'on a tenté, sans succès, d'insérer dans la case $k-1$. En comptant le nœud supplémentaire, cela donnera bien une corde de hauteur au plus $k-1$.

Par récurrence donc, la corde que l'on tente de placer dans une case d'index k quelconque a une hauteur d'au plus $k-1$.

Seulement, une majoration par $k-1$ de la hauteur n'est pas suffisante pour garantir l'invariant lors de l'insertion finale dans la case j (on obtiendrait $j-1$ alors qu'il nous faut $j-2$). Il nous faut donc un argument supplémentaire : si la corde termine dans la case d'index j, alors la case d'index j contenait nécessairement `L ""`. Dans ce cas, l'invariant sera bien garanti puisque sans cette dernière concaténation, la hauteur de la corde sera bien inférieure à $(j-1)-1$ (la majoration de sa hauteur lorsqu'on a tenté de la placer dans la case précédente), soit $j-2$.

Pourquoi la case d'index j contenait nécessairement `L ""` ? Tout simplement parce que si elle contenait préalablement une corde non vide, la longueur de cette corde serait au moins F_j. Et puisque la corde que l'on tente d'insérer n'a pas pu être placée dans la case $j-1$, elle a également une longueur d'au moins F_j. Donc le résultat de la concaténation serait de longueur supérieure ou égale à $2F_j$, et $2F_j > F_j + F_{j-1} = F_{j+1}$ selon les propriétés de la suite de Fibonacci (croissance et définition), et elle ne pourrait donc pas terminer dans la case j !

16. On insère toutes les feuilles (grâce à un parcours en profondeur) avec la fonction `insert`, puis on concatène le contenu des cases, en partant d'une corde vide. À chaque concaténation, le coût augmente, comme nous l'avons vu, de la taille du résultat. Donc on préférera commencer par concaténer les cordes de plus petite taille d'abord. On considère donc les cases **de la gauche vers la droite**, MAIS la concaténation des cordes dans le tableau, elle, doit être considérée de la droite vers la gauche : vu la façon dont le tableau est rempli, les morceaux de cordes représentant la fin de la chaîne se trouvent en effet nécessairement à gauche de ceux représentant le début.

```
let rebalance r =
  (* Insertion des feuilles dans le bon ordre *)
  let rec dfs = function
    | N (_, lc, rc) -> dfs lc; dfs rc
    | leaf -> insert leaf 2
  in dfs r;
  (* Concaténation des cases de droite à gauche *)
  let res = ref (L "") in
  for i = 2 to max_size-1 do
    res := concat file.(i) !res;
    file.(i) <- L ""   (* Pour que le tableau soit vide *)
  done;
  !res
```

Le sujet original était assez flou sur la question, et la version réécrite proposée conservait ce flou, a dessein. Il arrivera que vous ayez à comprendre l'intention du sujet à travers des descriptions parfois fort imprécises.

17. Soit j l'index le plus grand parmi les index des cases non-vides à l'issue de toutes les insertions. Comme toute case i contient une corde de hauteur au plus $i-2$, La concaténation de toutes les cases non-vides (**sous réserve de le faire dans le bon sens !**) donne nécessairement une corde de hauteur au plus $j-1$, soit $h \leqslant j-1$, donc $j \geqslant h+1$.

Par ailleurs, la longueur de la corde est au moins égale à la longueur de la corde dans la case j, donc au moins $n \geqslant F_j$.

Les F_i formant une suite croissante, $j \geqslant h+1$ implique $F_j \geqslant F_{h+1}$, et donc $n \geqslant F_{h+1}$.

18. $n \geqslant F_{h+1}$ conduit à $n \geqslant \Phi^h/\sqrt{5}$, soit $\log(n\sqrt{5}) \geqslant h \log \Phi$.

3 Équilibrage optimal

19. Reprendre le cours au besoin sur la question de la concaténation de listes dans un arbre. On peut par exemple écrire :

```
let leaves tree =
  let rec to_list l = function
    | N (_, lc, rc) -> to_list (to_list l rc) lc
    | leaf -> leaf::l
  in to_list [] tree
```

20. On dispose ici d'un peu de souplesse pour l'implémentation. Comme les feuilles sont toutes distinctes (puisque l'on peut les identifier avec `=`), on peut associer une chaîne et sa profondeur dans c_1.

On va donc faire un parcours en profondeur dans c_1, et mémoriser ces profondeurs dans un dictionnaire. Il ne restera alors qu'à prendre les feuilles de la liste des feuilles de c, et à les remplacer, via un `List.map`, par leur profondeur, extraite du dictionnaire. Cela donne :

```
let depths leaves_c c1 =
  let dict = Hashtbl.create 42 in
  let rec dfs d = function
    | N (_, lc, rc) -> dfs (d+1) lc; dfs (d+1) rc
    | L s -> Hashtbl.add dict s d
  in dfs 0 c1;
     List.map (function | L s -> Hashtbl.find dict s
                        | _ -> failwith "impossible")
              leaves_c
```

On peut également remarquer que la première partie de l'algorithme étant quadratique en la taille de la corde, on peut bien se permettre ici un algorithme quadratique (le précédent est linéaire), en cherchant les feuilles une à une.

Une première fonction détermine la profondeur d'une feuille, retournant -1 si elle ne la

trouve pas et la profondeur si elle la trouve :

```
let rec depth c = function
  | L s -> if s = c then 0 else -1
  | N (_, lc, rc) -> match depth c lc with
                     | -1 -> (match depth c rc with
                              | -1 -> -1
                              | d -> d+1)
                     | d -> d+1
```

Puis on l'applique à la liste :

```
let depths leaves_c c1 =
  List.map (fun c -> depth c c1) leaves_c
```

21. C'est une question qui n'est pas simple à faire efficacement. Il existe de nombreuses solutions à ce problème, de complexité variées, et il est facile d'écrire des fonctions qui ne marchent pas, et vous êtes encouragés à *tester* vos idées.

Une solution consiste à utiliser une pile de couples : des sous-arbres de la corde que l'on cherche à construire, et la profondeur à laquelle on veut qu'ils se trouvent.

On va ajouter dans la pile les feuilles une par une (avec leur profondeur), puis après chaque ajout, tant que les deux éléments au sommet de la pile sont à la même profondeur souhaitée, on les concatène avec le constructeur `N`. Le sous-arbre plus grand doit se trouver à une profondeur décrémentée de un. Cette boucle « tant que » est assurée par la fonction `update`.

S'il y a une solution, l'algorithme termine avec un seul arbre dans la pile, associé à la profondeur souhaitée 0.

Plutôt que d'utiliser une pile (pour laquelle il est difficile d'accéder au *second* élément au sommet, s'il existe), on travaille avec une simple liste passée en paramètre dans la fonction récursive `traite` qui effectue la boucle que l'on vient de décrire.

Cette fonction `iterate` (`rope list * int list * (rope * int) list -> rope`) prend donc en argument un triplet (la liste des feuilles restant à traiter, la liste des profondeurs restant à traiter, et l'état actuel de la pile) et retourne la corde c_2 souhaitée. On remarquera que les deux listes ayant la même longueur, elles doivent se vider au même moment.

La fonction `update` (`(rope * int) list -> (rope * int) list`) prend donc la pile de couples corde/profondeur souhaitée et effectue les concaténations souhaitées, comme décrit précédemment.

On peut montrer que cette approche a une complexité linéaire en la longueur des listes manipulées. Compte tenu de la complexité de la transformation proposée[3], une solution quadratique resterait acceptable ici.

Cela donne finalement :

```
let reconstruct list_x list_p =
  let rec iterate = function
    | [], [], [t] -> t
    | x::tx, p::tp, stack
        -> let rec update = function
             | (r, p)::(r', p')::t when p = q'
                 -> update ((concat r' r, p-1)::t)
             | lst -> lst
           in iterate (tx, tp, update ((x, p)::stack))
    | _ -> failwith "Impossible"
  in fst (iterate (list_x, list_p, []))
```

22. Il ne reste plus qu'à assembler les morceaux :

```
let rebalance_opt c =
  let leaves_c = leaves c in
  let c1 = transform leaves_c in
  let d = depths leaves_c c1 in
  reconstruct leaves_c d
```

3. Une implémentation optimale pourrait donner une complexité quasi-linéaire, cependant.