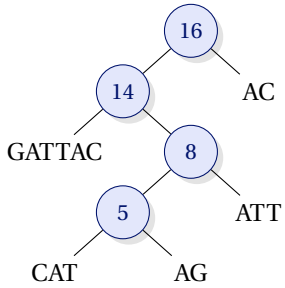
Cordes (d'après X 2008)

1 Structure de corde

La majorité des langages de programmation fournissent une notion primitive de chaînes de caractères. Si ces chaînes s'avèrent adaptées à la manipulation de mots ou de textes relativement courts, elles deviennent généralement inutilisables sur de très grands textes. L'une des raisons de cette inefficacité est la duplication d'un trop grand nombre de caractères lors des opérations de concaténation ou d'extraction d'une sous-chaîne. Or il existe des domaines où la manipulation efficace de grandes chaînes de caractères est essentielle (représentation du génôme en bio-informatique, éditeurs de texte, etc.).

Ce problème aborde une alternative à la notion usuelle de chaîne de caractères connue sous le nom de *corde*. Une corde est tout simplement un arbre binaire strict, dont les feuilles sont des chaînes de caractères usuelles et dont les nœuds internes représentent des concaténations. Ainsi la corde



représente la chaîne de caractères GATTACCATAGATTAC, obtenue par concaténation des cinq chaînes de caractères GATTAC, CAT, AG, ATT et AC. L'intérêt des cordes est d'offrir une concaténation immédiate et un partage possible de caractères entre plusieurs chaînes, au prix d'un accès aux caractères un peu plus coûteux.

Une corde est donc un arbre binaire dont les feuilles sont des chaînes de caractères. Plus précisément, une corde peut être soit une feuille, étiquetée par une chaîne (`string`), soit un noeud représentant la concaténation des chaînes représentées par ses fils gauche et droit. Pour des raisons d'efficacité, on stocke dans les noeuds la longueur de la chaîne représentée par la corde correspondante. On a donc

```
type rope = L of string | N of int * rope * rope;;
```

Pour des raisons pratiques, on interdit d'avoir des feuilles contenant des chaînes vides, à la seule exception d'un arbre réduit à une feuille. La chaîne vide sera donc impérativement représentée par `L ""`. Cela garantit par ailleurs que les entiers associés à chaque nœud sont tous strictement positifs. Toute corde fournie à une fonction vérifiera cet invariant, les cordes retournées par vos fonctions doivent également respecter cet invariant.

On rappelle que, pour `s` une chaine de caractères de longueur n,

- `String.length s` retourne le nombre n de caractères dans la chaîne `s` (en $\Theta(1)$) ;
- si `i` est un entier vérifiant $0 \leqslant i < n$, alors `s.[i]` fournit le caractère en position `i` (indexation commençant à zéro) de la chaîne (en $\Theta(1)$), et lève une exception sinon ;
- si `i` et `m` sont deux entiers vérifiant $0 \leqslant i < i+m \leqslant n$, alors `String.sub s i m` est la sous-chaîne de longueur m contenant les caractères aux positions d'index i à $i+m-1$ inclus (en $\Theta(m)$), et lève une exception sinon.

1. Écrire une fonction `length` (`rope -> int`) prenant en argument une corde et renvoyant, en temps constant, la longueur de la chaîne de caractères qu'elle représente.

2. Écrire une fonction `create` (`string -> rope`) prenant en argument une chaîne de caractère et renvoyant, en temps constant, une corde (simple) représentant cette chaîne.

3. Écrire une fonction `concat` (`rope -> rope -> rope`) prenant en argument deux cordes et renvoyant, en temps constant, une corde représentant la concaténation des chaînes que représentent les deux cordes passées en argument.

4. Écrire une fonction `nth` (`int -> rope -> char`) prenant en argument un entier `i` et une corde représentant la chaîne de caractères $a_0 a_1 \dots a_{n-1}$ et renvoyant le caractère a_i. On supposera $0 \leqslant i < n$ (la fonction peut avoir n'importe quel comportement si cette condition n'est pas vérifiée).

5. Écrire une fonction `index` (`char -> rope -> int`) prenant en argument un caractère `c` et une corde et renvoyant l'index de la première occurrence de `c` dans la chaîne de caractère représentée par la corde si `c` s'y trouve, et -1 sinon.

6. Écrire une fonction `prefix` (`int -> rope -> rope`) prenant en argument un entier `p` et une corde représantant la chaîne de caractères $a_0 a_1 \dots a_{n-1}$ et renvoyant une cordre représentant la chaîne de caractères $a_0 a_1 \dots a_{p-1}$. On supposera $0 \leqslant p \leqslant n$. On s'attachera, dans cette question et les suivantes, à réutiliser autant de sous-arbres de la corde en argument que possible.

7. Écrire une fonction `suffix` (`int -> rope -> rope`) prenant en argument un entier `p` et une corde représantant la chaîne de caractères $a_0 a_1 \dots a_{n-1}$ et renvoyant une cordre représentant la chaîne de caractères $a_p a_{p+1} \dots a_{n-1}$. On supposera $0 \leqslant p < n$.

8. Écrire une fonction `sub` (`int -> int -> rope -> rope`) prenant en argument un entier `i`, un entier `m` et une corde représantant la chaîne de caractères $a_0 a_1 \dots a_{n-1}$ et renvoyant une cordre représentant la chaîne de caractères $a_i a_{i+1} \dots a_{i+m-1}$. On supposera $0 \leqslant i < i+m \leqslant n$.

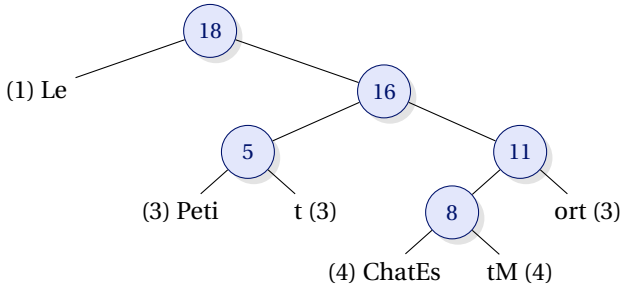
9. Quelle est la complexité de la fonction `sub` ? On pourra exprimer cette complexité en fonction de la longueur n de la chaîne de caractères représentée par la corde, le nombre de feuilles, et/ou la hauteur de l'arbre, et on justifiera la réponse.

2 Équilibrage

Le hasard des concaténations peut amener une corde à se retrouver déséquilibrée, c'est-à-dire à avoir certaines de ses feuilles très éloignées de la racine et donc d'accès plus coûteux. Nous allons à présent tâcher de rééquilibrer une corde déséquilibrée.

Considérons une corde `r` composée de $k+1$ feuilles, et donc de k nœuds internes. Notons ces $k+1$ feuilles $x_0, x_1, \ldots, x_k$ lorsqu'on les considère de la gauche vers la droite, si bien que la corde c représente la chaîne de caractères obtenue par concaténation des feuilles (dans la suite les x_i désignent tant les feuilles que les chaînes qui les étiquètent), soit $x_0 x_1 \ldots x_k$.

La profondeur d'une feuille x_i est notée prof(x_i) et est définie comme la distance de x_i à la racine. Voici un exemple de corde pour $k = 5$ où la profondeur de chaque feuille est indiquée entre parenthèses :



Le coût de l'accès à un caractère de la feuille x_i est défini comme la profondeur de cette feuille dans `r`, soit prof(x_i) (on ne considère donc pas le coût de l'accès dans le mot x_i lui-même).

Le coût total d'une corde est alors défini comme la somme des coûts d'accès à tous ses caractères, et vaut donc

$$\text{coût}(r) = \sum_{i=0}^{k} \text{longueur}(x_i) \times \text{prof}(x_i)$$

10. Quel est le coût associé à la corde proposé en exemple ci-dessus ?

11. Proposer une fonction `cost` (`rope -> int`), prenant en argument une corde et renvoyant, en coût linéaire en la taille de l'arbre constituant la corde, le coût associé à cette corde.

Un rééquilibrage consiste à construire une corde différente, dont les feuilles sont $x_0, x_1, \ldots, x_k$ dans le même ordre (le mot représenté ne doit pas changer) et dont le coût est, éventuellement, meilleur. L'algorithme proposé est le suivant.

On considère un tableau (global) nommé `file` de cordes dans lequel les feuilles de `r` vont être successivement insérées dans le sens des indices croissants. Les cases d'indices 0 et 1 ne sont pas utilisées. La case d'indice i contient soit la corde vide (`L ""`), soit une corde de hauteur inférieure ou égale à $i-2$ et dont la longueur est comprise dans l'intervalle $[F_i, F_{i+1}[$ où F_i désigne le i^e terme de la suite de Fibonacci.

On rappelle que la suite de Fibonacci est définie par :

$$\begin{cases} F_0 = 0 \\ F_1 = 1 \\ F_{n+2} = F_n + F_{n+1} \qquad \text{pour } n \geqslant 0 \end{cases}$$

La hauteur d'une corde `r`, notée hauteur(r), est la profondeur maximale de ses feuilles, c'est-à-dire :

$$\begin{cases} \text{hauteur}(\text{L } s) = 0 \\ \text{hauteur}(\text{N } (n, r_1, r_2)) = 1 + \max(\text{hauteur}(r_1), \text{hauteur}(r_2)) \end{cases}$$

Pour équilibrer la corde `r` dont les feuilles sont $x_0, x_1, \ldots, x_k$, dans cet ordre, on procède ainsi de la façon suivante.

① On insère successivement chaque feuille x_j dans le tableau `file` à partir de la case 2 ; l'insertion d'une feuille, et plus généralement d'une corde, à partir de la case d'indice i se fait ainsi :

- la corde à insérer est concaténée à droite de la corde se trouvant dans la case i ; soit r_0 le résultat ; si la longueur de r_0 est comprise dans l'intervalle $[F_i, F_{i+1}[$, on affecte r_0 à la case i et on a terminé l'insertion de cette corde ;
- sinon, on affecte une corde vide à la case i et on revient à l'étape précédente pour effectuer l'insertion de r_0 à partir de la case d'indice $i+1$.

l'objectif est de garantir l'invariant suivant : après l'insertion de la feuille x_j, la concaténation successive de toutes les cordes contenues dans les cases de `file`, considérées dans le sens des indices décroissants, est égale à une corde représentant le mot $x_0, x_1, \ldots, x_j$.

② Le résultat est alors la corde résultant de la concaténation successive de toutes les cordes de file, considérées dans le sens des indices décroissants.

12. Calculer le résultat de cet algorithme sur la corde de l'exemple précédent.

Afin d'éviter tout débordement arithmétique en calculant F_n, on limite la taille du tableau `file` à 44 cases, indexées de 0 à 43 (les cases 0 et 1 ne servent pas).

On introduit la constante `max_size = 44` et on calcule les valeurs de F_n une fois pour toutes pour $0 \leqslant n \leqslant 44$ dans un tableau (global) `fib` ainsi déclaré :

```
let max_size = 44;;

let fib = Array.make (max_size+1) 0;;
```

13. Écrire la fonction `init_Fib` (`unit -> unit`) qui initialise le tableau `fib` avec les valeurs F_n de la suite de Fibonacci en temps linéaire.

Le tableau `file` est déclaré comme un tableau global contenant des cordes :

```
let file = Array.make max_size (L "");;
```

14. Écrire la fonction `insert` (`rope -> int -> unit`) qui prend en arguments une corde `r` et un entier `i`, tels que $2 \leqslant i <$`max_size`, hauteur$(r) \leqslant i-2$ et longueur$(r) \geqslant F_i$, et réalise l'insertion de `r` dans le tableau `file` à partir de la case d'indice `i`.

15. Montrer que l'invariant « hauteur$(r_i) \leqslant i-2$ et longueur$(r_i) \geqslant F_i$ » est préservé par cette fonction pour toutes les valeurs r_i dand les cases non vides du tableau `file`.

16. Écrire la fonction `rebalance` (`rope -> rope`) qui réalise l'équilibrage d'une corde par l'algorithme ci-dessus, et laisse `L ""` dans toutes les cases du tableau `file`.

17. Soit `r` une corde non vide renvoyée par la fonction `rebalance` ci-dessus. Soit n sa longueur et h sa hauteur. Montrer que l'on a $n \geqslant F_{h+1}$.

18. En déduire qu'il existe une constante K (indépendante de n) et une fonction f telles que

$$\text{coût}(r) \leqslant n\left(\log_\phi f(n) + K\right)$$

où ϕ est le nombre d'or. On admettra que l'on a $F_{i+1} \geqslant \phi^i/\sqrt{5}$ pour tout $i \geqslant 0$.

3 Équilibrage optimal

Bien que satisfaisant en pratique, l'équilibrage précédent n'est pas optimal. Le but de cette dernière partie est d'étudier une stratégie optimale de rééquilibrage (algorithme de Garcia-Wachs).

L'algorithme procède en deux temps : il commence par construire une corde c_1 ayant les mêmes feuilles que c, mais pas nécessairement dans le même ordre, et dont le coût est égal au coût d'un équilibrage optimal de c. Puis, dans un second temps, il transforme cette corde c_1 en une autre corde c_2 de même coût, dans laquelle les feuilles sont dans le même ordre que c.

La première partie opère sur une liste de cordes $q = \langle q_1, q_2, \ldots, q_p \rangle$, et procède de la façon suivante :

- initialement, la liste est la liste $q = \langle x_0, x_1, \ldots, x_k \rangle$ des $k+1$ feuilles de c (considérées de gauche à droite) ;
- tant que la liste contient au moins deux éléments, on effectue l'opération suivante :
  - on détermine le plus petit indice i vérifiant longueur$(q_{i-1}) \leqslant$ longueur(q_{i+1}) s'il existe de tels indices, et on pose $i = p$ sinon ;
  - on ôte q_{i-1} et q_i de la liste q et on forme leur concaténation ; soit c' la corde obtenue ;
  - on détermine le plus grand indice $j < i$ vérifiant longueur$(q_{j-1}) \geqslant$ longueur(c') s'il en existe, et on pose $j = 0$ sinon ;
  - on insère c' dans la liste juste après q_{j-1} (au début de la liste si $j = 0$).
- le résultat est la dernière corde restant dans la liste à l'issue de la boucle.

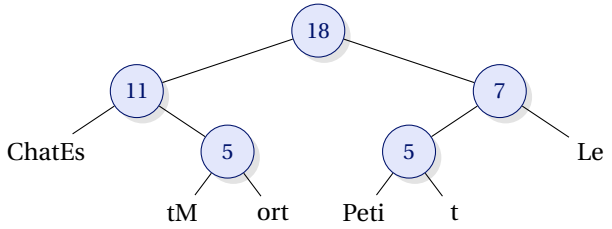
Il est clair que le résultat est une corde ayant les mêmes feuilles que c, mais pas dans le même ordre. On admettra le résultat suivant : l'arbre ainsi obtenu est de coût égal au coût d'une corde après un rééquilibrage optimal. On fournit une implémentation de cette première étape, une fonction `transform` (`rope list -> rope`) et de coût quadratique en la longueur de la liste :

```
let rec transform = function
  | [r] -> r
  | lst -> let rec zipper = function
             | [] -> failwith "impossible"
             | a::b::t when t=[] || length a <= length (List.hd t)
                 -> t, concat a b
             | h::t -> match zipper t with
                         | rst, (N _ as r) when length r <= length h
                             -> h::r::rst, L ""
                         | rst, r -> h::rst, r
           in let rst, r = zipper lst
              in transform (if r = L "" then rst else r::rst)
```

Pour les feuilles de la corde prise précédemment en exemple, cela donne :

```
# transform [L "Le"; L "Peti"; L "t"; L "ChatEs"; L "tM"; L "ort"];;
- : rope =
N (18, N (11, L "ChatEs", N (5, L "tM", L "ort")),
 N (7, N (5, L "Peti", L "t"), L "Le"))
```

Soit la corde c_1 suivante :



19. Avant d'utiliser la fonction précédente, il est nécessaire de construire la liste $\langle x_0, x_1, \ldots, x_k \rangle$ des $k+1$ feuilles de c, considérées de gauche à droite. Proposer une fonction `leaves` (`rope -> rope list`) prenant en argument une corde et retournant la liste de ses $k+1$ feuilles (prises de gauche à droite) en temps linéaire en k. Pour la corde en exemple,

cela donnerait :

```
# leaves (N (18, L "Le", N (16, N (5, L "Peti", L "t"),
            N (11, N (8, L "ChatEs", L "tM"), L "ort"))));;
- : rope list = [L "Le"; L "Peti"; L "t"; L "ChatEs"; L "tM"; L "ort"]
```

La seconde étape de l'algorithme fonctionne ainsi : soit c_1 la corde obtenue à l'issue de la première étape de l'algorithme, après appel de la fonction `transform` sur la liste de feuilles de c. Chaque feuille x_i de c se trouve dans c_1 à une certaine profondeur. Notons p_i cette profondeur. On admet la propriété suivante : il existe une corde c_2 dont les feuilles sont exactement $x_0, x_1, \ldots, x_k$ dans cet ordre, où la profondeur de chaque x_i est exactement p_i. On peut alors construire c_2 en ne connaissant que les p_i, et c_2 est alors un rééquilibrage optimal de c.

20. Proposer une fonction `depths` (`rope list -> rope -> int list`) prenant en argument la liste des feuilles de c et la corde c_1 et retournant une liste d'entiers de taille $k+1$ telle que dans l'entier à la position i dans la liste corresponde à la profondeur p_i de la feuille x_i dans c_1. **On admettra que l'on peut comparer** [1] **les feuilles de c et de c_2 avec l'opérateur `=`**. Cette fonction, appelée sur les cordes c et c_1 prises en exemple, doit retourner la liste `[2; 3; 3; 2; 3; 3]`.

21. Écrire une fonction `reconstruct` (`rope list -> int list -> rope`) prenant en argument la liste des feuilles x_i de c et la liste de leurs profondeurs dans c_1 et renvoyant la corde c_2. Notez qu'il n'existe pas nécessairement une corde où chaque x_i est à la profondeur p_i pour n'importe quelle liste de p_i, on demande simplement un algorithme qui fonctionne uniquement sur l'*hypothèse* qu'une telle corde existe (ce qui est le cas ici). Votre algorithme doit être expliqué *en détail*, et une meilleure complexité sera valorisée. Sur l'exemple choisi, cela donne :

```
# reconstruct [L "Le"; L "Peti"; L "t"; L "ChatEs"; L "tM"; L "ort"]
              [2; 3; 3; 2; 3; 3];;
- : rope = N (18, N (7, L "Le", N (5, L "Peti", L "t")),
             N (11, L "ChatEs", N (5, L "tM", L "ort")))
```

22. En déduire une fonction `rebalance_opt` (`rope -> rope`) prenant en argument une corde et retournant une corde représentant la même chaîne de caractères mais de coût optimal.

1. Cela suppose que les feuilles correspondent toutes à des chaînes de caractères différentes, ce que l'on peut s'arranger pour garantir si nécessaire.