I Représentation par des listes

Question 1. L'ajout est ici faisable en temps constant, comme l'ajout d'un élément en tête de liste.
Question 2. On commence par écrire une fonction qui calcule la taille d'un mot. Ensuite, on utilise une fonction auxiliaire qui garde en mémoire la taille du mot le plus long et le mot en question. On compare alors avec la taille du mot courant et on change les arguments si nécessaire.

```
let rec taille_mot = function
   | [] -> 0
   | x :: q -> 1 + taille_mot q;;

let plus_long d =
   let rec aux n u = function
      | [] -> u
      | (v, _) :: q -> let m = taille_mot v in
                  if m > n then aux m v q
                  else aux n u q in
   aux 0 [] d;;
```

Question 3. On compare les éléments un par un. On s'arrête dès que le premier mot est vide :

```
let rec prefixe u v = match u, v with
   | [], _ -> true
   | x :: q, y :: r when x = y -> prefixe q r
   | _ -> false;;
```

Question 4. Même idée que précédemment, mais il faut que les deux soient vides :

```
let rec egaux u v = match u, v with
   | [], [] -> true
   | x :: q, y :: r when x = y -> egaux q r
   | _ -> false;;
```

Question 5. On parcourt le dictionnaire jusqu'à trouver un mot qui est le même. On renvoie alors le numéro associé :

```
let rec recherche u = function
   | [] -> -1
   | (v, n) :: q when egaux u v -> n
   | _ :: q -> recherche u q;;
```

Question 6. La preuve se fait ici par induction sur la taille du dictionnaire. Elle diminue de 1 à chaque appel récursif.
Question 7. On doit d'abord réfléchir à la complexité maximale de la fonction `egaux`. Elle est atteinte lors que le mot cherché est un préfixe du mot auquel on le compare, et alors de l'ordre de $O(k)$. Ensuite, la complexité de `recherche` est maximale lorsque le mot cherché n'est pas dans le dictionnaire. Au total, la complexité est de l'ordre de $O(kn)$.
Question 8. On parcourt les éléments jusqu'à trouver le mot à supprimer. On renvoie alors la queue du dictionnaire.

```
let rec suppression u = function
   | [] -> []
   | (v, _) :: q when egaux u v -> q
   | x :: q -> x :: (suppression u q);;
```

II Arbres lexicographiques et Patricia

Question 9. Il suffit de compter le nombre de nœud dont le numéro est différent de -1. On utilise pour cela une fonctionnelle, car on doit relancer un appel sur chaque fils de l'arbre. L'argument initial du `fold_left` vaut soit 0 soit 1 selon que le nœud est terminal ou non.

```
let rec taille = function N(_, n, l) ->
   List.fold_left (fun x y -> x + taille y) (if n = -1 then 0 else 1) l;;
```

Question 10. On utilise une fonctionnelle `map` qui applique une concaténation à chaque premier élément d'un couple.

```
let prefixage u d = List.map (fun (x, y) -> (u@x, y)) d;;
```

Question 11. L'idée est de partir des feuilles et de préfixer au fur et à mesure, en créant de nouvelles entrées chaque fois qu'on a un nœud terminal. Pour cela, on calcule les dictionnaires associés aux fils, puis on les préfixe par le mot de la racine, et on rajoute ce dernier si le numéro n'est pas -1.

```
let rec dico_patricia = function N(u, n, l) ->
    let a = if n = -1 then [] else [u, n] in
    a @ (List.fold_left (fun x y -> x @ (prefixe u (dico_patricia y))) [] l);;
```

Question 12. On applique exactement ce qui est décrit :

```
let compare u v = match u, v with
   | (x :: q), (y :: r) -> x < y
   | _ -> false;;
```

Question 13. On utilise ici deux fonctions auxiliaires : `aux1` sert à vérifier qu'une feuille doit être terminale. Si un nœud n'est pas une feuille, on lance `aux2` sur ses fils. Cette deuxième fonction sert à vérifier que les éléments sont bien dans l'ordre croissant des premières lettres. On traite le cas de la liste vide et des singletons à part.

```
let est_patricia (N(u, n, l)) =
        let rec aux1 (N(u, n, l)) = match l with
      | [] -> n <> -1
      | _ -> aux2 l
   and aux2 = function
      | [] -> true
      | [a] -> aux1 a
      | (N(v, m, q)) :: (N(w, p, r)) :: s -> (compare v w) && aux1 (N(v, m, q)) &&
                                             aux2 ((N(w, p, r)) :: s) in
   n = -1 && aux1 (aux1 (N(u, n, l)) || l = [];;
```

Question 14. On commence par écrire une fonction qui nous servira pour toute la suite : la fonction `prefixe_commun` prend en arguments deux mots u et v et renvoie trois mots w, x et y tels que w est le plus grand préfixe commun à u et v et $u = wx$ et $v = wy$. Elle nous permettra de continuer les recherches avec les bons suffixes. Cette fonction continue la recherche jusqu'à ce que l'un des deux mots soit vide ou que leurs premières lettres diffèrent.

```
let rec prefixe_commun u v = match u, v with
   | [], _ -> [], [], v
   | _, [] -> [], u, []
   | x :: q, y :: r when y = x -> let w, z, t = prefixe_commun q r in
                                 x :: w, z, t
   | x :: q, y :: r -> [], u, v;;
```

Ensuite, la fonction de recherche se décompose en deux parties : la fonction `recherche` elle-même qui compare le mot cherché au mot de la racine. S'ils sont égaux, on a trouvé le bon mot, sinon si le mot de la racine est un préfixe du mot cherché, on cherche le suffixe parmi tous les fils, sinon le mot n'est pas dans le dictionnaire. La fonction `cherche_liste` cherche un mot dans une liste d'arbres. On aurait pu utiliser ici une fonctionnelle, mais elle aurait nécessité l'écriture d'une fonction d'agrégation compliquée et n'aurait pas facilité la lecture. L'idée est de chercher un arbre dont la racine commence par la même lettre que le mot cherché et de relancer un appel à `recherche`.

```
let rec recherche u a = match a with
   | N([], _, q) -> cherche_liste u q
   | N(v, m, q) -> begin match prefixe_commun u v with
                   | _, [], [] -> m
                   | _, y, [] -> cherche_liste y q
                   | _ -> -1
                  end
and cherche_liste u = function
   | [] -> -1
   | (N(v, m, q)) :: r when compare u v -> -1
   | (N(v, m, q)) :: r when compare v u -> cherche_liste u r
   | a :: r -> recherche u a;;
```

Question 15. On a ici une complexité qui dépend de la hauteur de l'arbre h et de la taille du mot n. De plus, elle dépend éventuellement du nombre de fils d'un nœud, qui au maximum est de la taille k (qu'on peut supposer

constante) de l'alphabet utilisé. Pour un nœud donné, on fera un appel à `prefixe_commun` dont la complexité est $O(m)$ où m est la taille du préfixe commun. On a :

- Si $h = 0$, alors $C(n, 0) = O(1)$. De plus, $C(0, h) = O(1)$.
- Sinon, $C(n, h) = O(m) + O(k) + C(n - m, h - 1)$.

On montre facilement que la complexité totale est $O(k \times \min(n, h))$.

Question 16. On renvoie juste un arbre avec un seul fils :

```
let creation (u, n) = N([], -1, [N(u, n, [])]);;
```

Question 17. Un peu dans la même idée que la fonction de recherche, on écrit deux fonctions (une pour un nœud, une pour une liste). Il faut ici envisager les différents cas possibles : si le mot à ajouter u est égal au mot du nœud v, alors on se contente de modifier le numéro. Si v est préfixe de u, On ajoute le suffixe au bon fils. Si l'inverse est vrai, on remplace v par u et on fait descendre v d'un niveau de profondeur. Sinon, on doit scinder le nœud.

```
let rec ajout (u, n) = function
   | N([], _, []) -> creation (u, n)
   | N(v, m, q) -> begin match prefixe_commun u v with
                   | w, [], [] -> N(u, n, q)
                   | w, x, [] -> N(v, m, ajoute_liste (x, n) q)
                   | w, [], y -> N(u, n, [N(y, m, q)])
                   | w, x, y -> N(w, -1, ajoute_liste (x, n) [N(y, m, q)])
                  end
and ajoute_liste (u, n) l = match l with
   | [] -> [N(u, n, [])]
   | (N(v, m, q)) :: r when compare u v -> (N(u, n, [])) :: l
   | (N(v, m, q)) :: r when compare v u -> (N(v, m, q)) :: (ajoute_liste (u, n) r)
   | x :: r -> (ajout (u, n) x) :: r;;
```

Question 18. On se contente de faire des ajouts successifs en réutilisant l'arbre obtenu :

```
let patricia_dico = List.fold_left (fun x y -> ajout y x) (N([], -1, []));;
```

Question 19. Toujours en deux parties. Il faut ici traiter à part les cas où on supprime un nœud qui a un seul fils (on remonte d'un niveau le fils), ou un nœud qui a un seul frère et un père non terminal (le père et le frère sont alors fusionnés).

```
let rec suppression u = function
   | N([], _, []) -> N([], -1, [])
   | N([], _, [N(v, m, [])]) when egaux u v -> N([], -1, [])
   | N(v, m, [N(w, p, q)]) when egaux u v -> N(v@w, p, q)
   | N(v, m, q) when egaux u v -> N(v, -1, q)
   | N(v, -1, [N(w, p, []);N(x, k, r)]) when egaux u (v@w) -> N(v@x, k, r)
   | N(v, -1, [N(w, p, q);N(x, k, [])]) when egaux u (v@x) -> N(v@w, p, q)
   | N(v, m, q) -> let _, y, _ = prefixe_commun u v in
                   N(v, m, supprime_liste y q)
and supprime_liste u l = match l with
   | [] -> []
   | (N(v, m, q)) :: r when compare u v -> l
   | (N(v, m, q)) :: r when compare v u -> (N(v, m, q)) :: (supprime_liste u r)
   | (N(v, m, [])) :: r when egaux u v -> r
   | (N(v, m, q)) :: r -> (suppression u (N(v, m, q))) :: r;;
```

Question 20. Les premiers cas sont les cas où les nœuds fusionnés sont préfixes l'un de l'autre. On se contente alors de repousser la fusion au niveau inférieur. Sinon, on doit scinder un nœud (c'est le dernier cas de filtrage). La deuxième fonction effectue la fusion de deux listes d'arbres. Il faut bien réfléchir à l'ordre des éléments comparés.

```
let rec fusion (N(u, m, q)) (N(v, n, r)) =
   match prefixe_commun u v with
      | _, [], [] -> N(u, max m n, fusion_liste q r)
      | _, [], z -> N(u, m, fusion_liste q [N(z, n, r)])
      | _, y, [] -> N(v, n, fusion_liste [N(y, m, q)] r)
      | x, y, z -> N(x, -1, [fusion (N(y, m, q)) (N(z, n, r))])
and fusion_liste l1 l2 = match l1, l2 with
   | _, [] -> l1
   | [], _ -> l2
   | (N(u, m, q)) :: qq, (N(v, n, r)) :: rr when compare u v ->
                (N(u, m, q)) :: (fusion_liste qq l2)
   | (N(u, m, q)) :: qq, (N(v, n, r)) :: rr when compare v u ->
                (N(v, n, r)) :: (fusion_liste l1 rr)
   | (N(u, m, q)) :: qq, (N(v, n, r)) :: rr ->
                (fusion (N(u, m, q)) (N(v, n, r))) :: (fusion_liste qq rr);;
```