Correction du DS 5

Julien Reichert

Questions de cours

Question de cours 1 : [Réponse dans le cas des tas-min, l'adaptation se fait facilement pour les tas-max.]
Au vu des propriétés des tas et des ABR, si l'arbre est non vide, alors la plus petite valeur est à la racine. Dans ce cas, toutes les autres valeurs seront dans le sous-arbre droit, qui doit lui-même à la fois être un tas et un ABR. Par une induction structurelle évidente, on conclut que notre ABR qui est un tas est un peigne droit. Si de plus l'arbre doit être bicolore, le peigne droit ne peut pas avoir de nœud noir dans le sous-arbre droit de la racine, donc comme le fils d'un nœud rouge ne peut pas être rouge, il reste les cas suivants : arbre vide, arbre réduit à une racine (quelle que soit la couleur), arbre réduit à une racine noire avec un fils droit rouge (en particulier de clé supérieure).
Sans l'hypothèse que toutes les valeurs étaient différentes deux à deux, on n'aurait pas forcément un peigne mais toutes les clés du sous-arbre gauche de tout nœud devraient être égales à la clé du nœud en question. La hauteur des arbres bicolores ne serait alors plus limitée et la forme des ABR pourrait être quelconque.

Question de cours 2 :
Ma suggestion : un arbre PATRICIA est une variante d'un trie dont les nœuds sont étiquetés par des chaînes de caractères (vide pour la racine), avec les autres propriétés du trie : il s'agit d'un arbre dont les nœuds sont aussi étiquetés par des booléens, de sorte qu'une branche finissant par un booléen vrai corresponde à un mot obtenu en fusionnant les chaînes de caractères rencontrées sur les nœuds de la branche.

Question de cours 3 :
La matrice d'adjacence d'un graphe non orienté complet a des 1 partout, sauf sur la diagonale qui ne peut contenir que des 0 pour des raisons structurelles.

Question de cours 4 :
Un graphe sans circuit est un graphe orienté dont aucun sommet n'est relié à lui-même par un circuit de taille au moins un. Une composante fortement connexe d'un graphe orienté G est un sous-graphe G' de G maximal pour l'inclusion tel que pour deux sommets s et t de G' il existe un chemin de s à t en restant dans G' (et quitte à échanger les rôles de s et t il existe aussi un chemin dans l'autre sens). Soit une composante fortement connexe d'un graphe sans circuit. Si elle contient au moins deux sommets, alors il existe un chemin de l'un à l'autre puis de l'autre à l'un, et ces deux chemins collés bout à bout donnent un circuit de taille au moins deux. Donc une composante fortement connexe d'un graphe sans circuit est réduite à un sommet, voire à $\emptyset$ si le graphe est vide.

Question de cours 5 : [Pas de dessin, les exemples du cours suffisent.]
Si on ajoute un nouvel arc, il est possible qu'il n'y ait plus de chemin eulérien si l'arc relie deux sommets qui étaient initialement isolés. Si on retire un arc, comme le départ d'un circuit peut être où l'on souhaite, on peut imaginer qu'il s'agissait du dernier arc d'un circuit eulérien et donc en le perdant on a toujours un chemin qui passe une et une seule fois par les arcs restants, donc un chemin eulérien.

Question de cours 6 :

```
type 'a abr = Vide | Noeud of 'a abr * 'a * 'a abr;;

let rec minimum a = match a with
| Vide -> failwith "Arbre vide"
| Noeud(g, x, d) -> if g = Vide then x, d else let m, reste = minimum g in m, Noeud(reste, n, d);;
```

Les appels récursifs imbriqués portent sur un sous-arbre, donc la hauteur (entier minoré) est strictement décroissante à chaque appel, d'où la terminaison.

Concernant la correction, on peut envisager une induction structurelle en respectant la définition mathématique :
- Le minimum d'un arbre vide n'existe pas.
- Si un ABR a un sous-arbre gauche vide, alors son minimum est à la racine et le reste est le sous-arbre droit.
- Sinon, le minimum est dans le sous-arbre gauche, et c'est même son minimum. Le sous-arbre gauche sans son minimum est alors le sous-arbre gauche de l'ABR de départ sans son minimum, en laissant la racine et le sous-arbre droit inchangés.

Question de cours 7 :

Soit un trie créé à partir de trois mots. La racine doit avoir deux fils d'après l'énoncé. Cela veut dire qu'il y aura au moins un mot dans les deux sous-arbres, donc au plus deux.

Cas possible pour un fils de la racine :
- Pas de fils, donc c'est un mot.
- Un seul fils, donc c'est un mot.
- Deux fils, mais ce n'est pas un mot (sinon il y aurait trois mots dans le sous-arbre).

Dans le troisième cas, chacun des deux fils du fils produira un mot et un seul, pour des raisons d'effectif. Ce mot ne peut alors être produit que si le fils du fils en question n'a pas d'enfant.

Dans le deuxième cas, le fils du fils produira un mot et un seul, pour le même résultat.

On en déduit qu'un fils de la racine doit être dans le premier cas et l'autre dans le deuxième ou le troisième.

Cela donne un nombre de forme d'arbres de 2 (choix du fils qui est dans le premier cas) fois 3 (choix du cas pour l'autre fils, sachant que le deuxième cas a deux sous-cas pour déterminer où est le fils).

En utilisant le deuxième cas, on n'obtient pas un code préfixe. Un exemple est `0, 1, 10`. Il y a donc deux codes préfixes : `0, 10, 11` et `00, 01, 1`. Les quatre autres codes sont ambigus.

Question de cours 8 :

```
let floyd_warshall graphe =
  let n = Array.length graphe in
    let dist = Array.make_matrix n n max_int in
    let chemins = Array.make_matrix n n [] in
      for i = 0 to n-1 do
        List.iter (fun (s, p) -> dist.(i).(s) <- p; chemins.(i).(s) <- [i; s]) graphe.(i)
      done;
      for i = 0 to n-1 do if dist.(i).(i) > 0 then dist.(i).(i) <- 0;
      chemins.(i).(i) <- [i] done;
      for k = 0 to n-1 do
        for i = 0 to n-1 do
          for j = 0 to n-1 do
            if max dist.(i).(k) dist.(k).(j) < max_int && dist.(i).(k) + dist.(k).(j) < dist.(i).(j)
              then begin
                dist.(i).(j) <- dist.(i).(k) + dist.(k).(j);
                chemins.(i).(j) <- chemins.(i).(k) @ (List.tl chemins.(k).(j))
              end
          done;
          if dist.(i).(i) < 0 then failwith "circuit de poids négatif détecté"
        done
      done; chemins;;
```

Exercices

Exercice 1 :

Un trie est valide si tout nœud n'ayant aucun enfant est associé au booléen vrai et si aucun nœud n'a deux enfants étiquetés par la même lettre.

```
type trie = N of char * bool * trie list;;

let est_prefixe trie mot =
  let n = String.length mot in
  let rec aux i noeud =
    i = n || match noeud with
    | N(_, _, []) -> false
    | N(_, _, N(c, b, l)::_) when c = mot.[i] -> aux (i+1) (N(c, b, l))
    | N(osefc, osefb, N(c, _, _)::q) -> aux i (N(osefc, osefb, q))
  in aux 0 trie;;

let tous_differents l =
  let tbl = Hashtbl.create 8 in
  let rec aux liste = match liste with
  | [] -> true
  | a::_ when Hashtbl.mem tbl a -> false
  | a::q -> Hashtbl.add tbl a 0; aux q
  in aux l;;

let etiquettes l = List.map (fun (N(c, _, _)) -> c) l;;

let rec est_valide trie = match trie with
| N(_, b, l) -> (b || l <> []) && tous_differents (etiquettes l) && List.for_all est_valide l;;
```

Exercice 2 :

```
struct abit { int etiquette; int taille; struct abit* fg; struct abit* fd; };
typedef struct abit iabt;

void ajouter(iabt* a, int x)
{
  a->taille += 1;
  if (a->fg == NULL)
  {
    iabt* nv_noeud = malloc(sizeof(iabt);
    nv_noeud->etiquette = x; nv_noeud->taille = 1;
    nv_noeud->fg = NULL; nv_noeud->fd = NULL;
    a->fg = nv_noeud;
  }
  else if (a->fd == NULL)
  {
    iabt* nv_noeud = malloc(sizeof(iabt);
    nv_noeud->etiquette = x; nv_noeud->taille = 1;
    nv_noeud->fg = NULL; nv_noeud->fd = NULL;
    a->fd = nv_noeud;
  }
  else if (a->fg->taille < a->fd->taille) ajouter(a->fg, x);
  else ajouter(a->fd, x);
}
```

La différence entre la taille des deux sous-arbres de chaque nœud est nulle ou d'un en faveur du sous-arbre gauche. Ceci est un invariant de structure respecté par la fonction d'ajout.

Quand la taille est une puissance de deux moins un, cela donne un arbre vide ou une racine et deux enfants dont les tailles sont deux fois la même puissance de deux moins un, ce qui permet de prouver par induction structurelle que l'arbre dans sa globalité est complet.

Exercice 3 :

On représente ici un graphe non orienté comme un graphe orienté avec les arcs dans les deux sens. La représentation retenue est la liste d'adjacence (`int list array`).

```
let parcours_largeur graphe origine =
  let n = Array.length graphe in
  let fermes = Hashtbl.create 8 in Hashtbl.add fermes origine 0;
  let f = Array.make n 0 in (* file en tant que tableau non redimensionnable de capacité suffisante *)
  f.(0) <- origine;
  let ou_ecrire = ref 1 and ou_lire = ref 0 in
  while !ou_lire < !ou_ecrire do
    let sommet = f.(!ou_lire) in incr ou_lire;
    let mouline successeur =
      if not (Hashtbl.mem fermes successeur) then
        begin
          Hashtbl.add fermes successeur (Hashtbl.find fermes sommet + 1);
          f.(!ou_ecrire) <- successeur;
          incr ou_ecrire
        end
    in List.iter mouline graphe.(sommet)
  done; fermes;;

let diametre graphe =
  let n = Array.length graphe in
  let rep = ref 0 in
  for i = 0 to n - 1 do
    let tbl = parcours_largeur graphe i in
    let nb_fermes = ref 0 in
    Hashtbl.iter (fun _ valeur -> if valeur > !rep then rep := valeur; incr nb_fermes) tbl;
    if !nb_fermes < n then failwith "Pas connexe !"
  done; !rep;;
```

La complexité pour un graphe à n sommets et p arcs est n fois celle du parcours en largeur (plus un résidu négligeable pour parcourir les tables de hachage), soit $\mathcal{O}(n(n+p))$.

Exercice 4 :

On suppose que les sommets sont les premiers entiers naturels et on décide arbitrairement de représenter une arête $\{s,t\}$ (avec $s<t$ sans perte de généralité) en mettant t dans la liste d'adjacence de s uniquement. Le test d'existence d'une arête se fera dans la bonne liste.

```
type gno = int list array;;

let existe_hamiltonienne_complet (g:gno) = g <> [||];;

let existe_eulerienne_complet (g:gno) = let n = Array.length g in n = 2 || n mod 2 = 1;;
```

Oui, cet exercice était du troll.

Exercice 5 :

Ici, un algorithme glouton convient.

```
exception Gagne;;

let contient s m =
  let ns = String.length s in
  let nm = String.length m in
  try
    for i = 0 to ns - nm do
      let j = ref 0 in
      while !j < nm && s.[i + !j] = m.[!j] do incr j done;
      if !j = nm then raise Gagne
    done; false
  with Gagne -> true;;

let decoupages_mini journal chaine =
  let n = String.length chaine in
  let nb_decoupages = ref 0 in
  let ind_deb = ref 0 in
  let taille = ref 1 in
  while !ind_deb < n do
    if !ind_deb + !taille > n
    then begin
      incr nb_decoupages;
      ind_deb := n
    end
    else if contient journal (String.sub chaine !ind_deb !taille) then incr taille
    else begin
      if !taille = 1 then failwith "Un caractère de la chaine n'est pas dans le journal !";
      incr nb_decoupages;
      ind_deb := !ind_deb + !taille - 1;
      taille := 1
    end
  done; !nb_decoupages;;
```

Exercice 6 :

Les séquences seront toutes des listes ici.

```
let trouver_station reseau lgn =
  let lignes = Hashtbl.create 8 in
  for i = Array.length reseau - 1 downto 0 do
  (* Pour avoir le même ordre que lgn ! *)
    let mouline station =
      match Hashtbl.find_opt lignes station with
      | None -> Hashtbl.add lignes station [i]
      | Some l -> Hashtbl.replace lignes station (i::l)
    in List.iter mouline reseau.(i)
  done;
  Hashtbl.fold (fun station liste accu -> if liste = lgn then station::accu else accu) lignes [];;
```

Oui, je n'ai pas documenté `fold` mais ça existe et c'est classe. Au pire on peut utiliser `iter` pour muter une référence de liste...

Exercice 7 :

```
bool content(int codes[], int taille) // On va refaire un peu de C, pour changer !
{
  int defaites_a_rattraper = 0;
  int e_f_a_rattraper = 0;
  int serie_victoires = 0;
  int serie_bonifiees = 0;
  int serie_defaites = 0;
  int serie_epic_fail = 0;
  int serie_max_defaites = 0;
  int serie_max_epic_fail = 0;
  for (int i = 0 ; i < taille ; i += 1)
  {
    if (codes[i] > 0)
    {
      if (codes[i] == 2)
      {
        serie_bonifiees += 1;
        if (e_f_a_rattraper > 0) e_f_a_rattraper -= 1;
      }
      else
      {
        if (serie_bonifiees >= serie_max_epic_fail) serie_max_epic_fail = 0;
        serie_bonifiees = 0;
      }
      if (defaites_a_rattraper > -1) defaites_a_rattraper -= 1;
      serie_victoires += 1;
      if (serie_defaites > serie_max_defaites) serie_max_defaites = serie_defaites;
      serie_defaites = 0;
      if (serie_epic_fail > serie_max_epic_fail) serie_max_epic_fail = serie_epic_fail;
      serie_epic_fail = 0;
      if (serie_victoires >= serie_max_defaites) serie_max_defaites = 0;
    }
    else
    {
      if (codes[i] == -2)
      {
        e_f_a_rattraper += 1;
        serie_epic_fail += 1;
      }
      else
      {
        if (serie_epic_fail > serie_max_epic_fail) serie_max_epic_fail = serie_epic_fail;
      }
      if (defaites_a_rattraper == -1) defaites_a_rattraper = 1;
      else defaites_a_rattraper += 1;
      serie_victoires = 0;
      serie_bonifiees = 0;
      serie_defaites += 1;
    }
  }
  return defaites_a_rattraper == -1 && e_f_a_rattraper == 0
    && serie_max_defaites <= serie_victoires && serie_max_epic_fail <= serie_bonifiees;
}
```

Exercice 8 :

```
type 'a arbre = N of 'a * 'a arbre list;;

let rec inserer_noeud position element liste = match position, liste with
| 0, _ -> N(element, []) :: liste
| _, [] -> failwith "Liste trop courte"
| _, a::q -> a::(inserer_noeud (position-1) element q);;

let rec separer position liste = match position, liste with
| _, [] -> failwith "Liste trop courte"
| 0, a::q -> [], a, q
| _, a::q -> let avant, ici, apres = separer (position-1) q in a::avant, ici, apres;;

let rec insertion arbre element liste = match arbre, liste with
| _, [] -> failwith "Liste vide"
| N(e, l), [i] -> N(e, inserer_noeud i element l)
| N(e, l), i::q ->
  let avant, ici, apres = separer i l in
  N(e, avant @ ((insertion ici element q) :: apres));;

let rec retirer_noeud position liste = match position, liste with
| _, [] -> failwith "Liste trop courte"
| 0, N(_, [])::q -> q
| 0, _::q -> failwith "Pas une feuille"
| _, a::q -> a::(retirer_noeud (position-1) q);;

let rec retrait arbre liste = match arbre, liste with
| _, [] -> failwith "Liste vide"
| N(e, l), [i] -> N(e, retirer_noeud i l)
| N(e, l), i::q -> let avant, ici, apres = separer i l in N(e, avant @ ((retrait ici q) :: apres));;
```

Exercice 9 :

```
let bellman_ford graphe origine =
  let n = Array.length graphe in
  let dist_avant = Array.make n max_int in
  let dist = Array.make n max_int in dist.(origine) <- 0;
  let k = ref 0 in
  while !k <= n && dist <> dist_avant do
    let a_traiter = Array.make n false in
    for sommet = 0 to n-1 do
      if dist.(sommet) <> dist_avant.(sommet)
      then begin dist_avant.(sommet) <- dist.(sommet); a_traiter.(sommet) <- true end
    done;
    for sommet = 0 to n-1 do
      if a_traiter.(sommet) then
        for isucc = 0 to Array.length graphe.(sommet)-1 do
          let succ, poids = graphe.(sommet).(isucc) in
          dist.(succ) <- min dist.(succ) (dist_avant.(sommet) + poids)
        done
    done;
    incr k
  done;
  if !k > n then failwith "Circuit de poids strictement négatif !";
  dist;;
```

Exercice 10 : La fonction prend en argument un pointeur de pointeur d'entier, qui s'avèrera être l'adresse d'un tableau, ainsi qu'un tableau et sa taille. Elle calcule le nombre d'éléments positifs du tableau qu'elle renverra, après avoir recopié tous ces éléments positifs dans un nouveau tableau où le pointeur sera redirigé.

Ainsi, il est possible pour une fonction de « produire » un tableau sans avoir besoin de mettre sa taille en première position. Avec cette méthode, les tableaux pourront contenir autre chose que des entiers en pratique, ce qui n'était pas possible avec la méthode classique.

Problème 1

```
type ('a, 'b) abr = V | N of ('a, 'b) abr * ('a * (int * 'b)) * ('a, 'b) abr;;

let creer_abr () = V;;

let rec dans_abr cle a = match a with
| V -> false
| N(g, (cl, _), d) -> cle = cl || cle < cl && dans_abr cle g || cle > cl && dans_abr cle d;;

let rec consulter_abr cle a = match a with
| V -> failwith "Introuvable, ce n'est pas possible !"
| N(_, (cl, (_, elt)), _) when cle = cl -> elt
| N(g, (cl, _), _) when cle < cl -> consulter_abr cle g
| N(_, _, d) -> consulter_abr cle d;;

let rec inserer_abr cle cpl a = match a with
| V -> N(V, (cle, cpl), V)
| N(g, (cl, cp), d) when cle < cl -> N(inserer_abr cle cpl g, (cl, cp), d)
| N(g, e, d) -> N(g, e, inserer_abr cle cpl d);;

let rec modifier_elt_abr cle nv_elt a = match a with
| V -> failwith "Introuvable, ce n'est pas possible !"
| N(g, (cl, (pos, elt)), d) when cle = cl -> N(g, (cl, (pos, nv_elt)), d)
| N(g, (cl, cp), d) when cle < cl -> N(modifier_elt_abr cle nv_elt g, (cl, cp), d)
| N(g, e, d) -> N(g, e, modifier_elt_abr cle nv_elt d);;

let rec modifier_pos_abr cle nv_pos a = match a with
| V -> failwith "Introuvable, ce n'est pas possible !"
| N(g, (cl, (pos, elt)), d) when cle = cl -> N(g, (cl, (nv_pos, elt)), d)
| N(g, (cl, cp), d) when cle < cl -> N(modifier_pos_abr cle nv_pos g, (cl, cp), d)
| N(g, e, d) -> N(g, e, modifier_pos_abr cle nv_pos d);;

let rec minimum_abr a = match a with (* Le retour de la question de cours numéro 6 *)
| V -> failwith "Cas impossible"
| N(V, e, d) -> e, d
| N(g, e, d) -> let (m, gsansm) = minimum_abr g in (m, N(gsansm, e, d));;

let rec supprimer_abr cle a = match a with
| V -> failwith "Introuvable, ce n'est pas possible !"
| N(g, (cl, cp), V) when cle = cl -> cp, g
| N(V, (cl, cp), d) when cle = cl -> cp, d (* On profite d'une accélération gratuite ! *)
| N(g, (cl, cp), d) when cle = cl -> let (m, dsansm) = minimum_abr d in cp, N(g, m, dsansm)
| N(g, (cl, cp), d) when cle < cl -> let (rep, gsansrep) = supprimer_abr cle g in (rep, N(gsansrep, (cl, c
| N(g, e, d) -> let (rep, dsansrep) = supprimer_abr cle d in (rep, N(g, e, dsansrep));;

type 'a fp = { mutable taille : int ; mutable donnees : (int * 'a) array };;

type ('a, 'b) dict_ordre = { mutable dico : ('a, 'b) abr ; mutable file : 'a fp ; mutable nb_inser : int }

let creer_d_o init =
  { dico = creer_abr (); file = { taille = 0 ; donnees = Array.make 8 (0, init) }; nb_inser = 0 };;

let consulter_d_o cle d_o = consulter_abr cle d_o.dico;;

let modifier_d_o cle nv_elt d_o = d_o.dico <- modifier_elt_abr cle nv_elt d_o.dico;;
```

```
let inserer_d_o cle elt d_o =
  if dans_abr cle d_o.dico then failwith "Doublon !";
  let f = d_o.file in
  let prio = d_o.nb_inser in d_o.nb_inser <- d_o.nb_inser + 1;
  if f.taille = Array.length f.donnees then
  begin
    let nv_donnees = Array.init (2 * f.taille) (fun i -> f.donnees.(i mod f.taille)) in
    f.donnees <- nv_donnees
  end;
  f.donnees.(f.taille) <- (prio, cle);
  d_o.dico <- inserer_abr cle (f.taille, elt) d_o.dico;
  f.taille <- f.taille + 1;;

let supprimer_d_o cle d_o =
  let (pos, _), reste = supprimer_abr cle d_o.dico in
  d_o.dico <- reste;
  let f = d_o.file in
  let n = f.taille - 1 in
  f.donnees.(pos) <- f.donnees.(n);
  f.taille <- n;
  let prio, clebis = f.donnees.(pos) in
  d_o.dico <- modifier_pos_abr clebis pos d_o.dico;
  let position = ref pos in
  (* Remonter éventuellement *)
  while !position > 0 && fst f.donnees.(!position) < fst f.donnees.((!position-1) / 2) do
    let cle_pere = snd f.donnees.((!position-1) / 2) in
    f.donnees.(!position) <- f.donnees.((!position-1) / 2);
    d_o.dico <- modifier_pos_abr cle_pere !position d_o.dico;
    f.donnees.((!position-1) / 2) <- (prio, clebis);
    position := !position / 2
  done;
  (* Redescendre éventuellement *)
  while 2 * !position + 1 < n do
    if 2 * !position + 2 < n
      && fst f.donnees.(2 * !position + 2) < fst f.donnees.(2 * !position + 1)
      && fst f.donnees.(2 * !position + 2) < prio then
    begin
      let cle_fd = snd f.donnees.(2 * !position + 2) in
      f.donnees.(!position) <- f.donnees.(2 * !position + 2);
      d_o.dico <- modifier_pos_abr cle_fd !position d_o.dico;
      f.donnees.(2 * !position + 2) <- (prio, clebis);
      position := 2 * !position + 2
    end
    else if fst f.donnees.(2 * !position + 1) < prio then
    begin
      let cle_fg = snd f.donnees.(2 * !position + 1) in
      f.donnees.(!position) <- f.donnees.(2 * !position + 1);
      d_o.dico <- modifier_pos_abr cle_fg !position d_o.dico;
      f.donnees.(2 * !position + 1) <- (prio, clebis);
      position := 2 * !position + 1
    end
    else position := n
  done;
  d_o.dico <- modifier_pos_abr clebis !position d_o.dico;;
```

```
(*
Pour le retrait, au lieu de passer par la racine du tas implémentant la file de priorité,
on va échanger l'élément à la position connue grâce au dictionnaire avec le dernier élément,
celui-ci devant être remonté ou redescendu dans le tas si nécessaire.
Ensuite, la taille est annoncée comme baissée d'un cran, ce qui correspond au retrait proprement dit.
Il n'y a rien à retourner pour la suppression d'un élément dans un dictionnaire.
*)

let parcourir_d_o d_o =
  let les_cles = Array.sub d_o.file.donnees 0 d_o.file.taille in
  Array.init d_o.file.taille (fun i -> (snd les_cles.(i), consulter_abr (snd les_cles.(i)) d_o.dico));;
```

Problème 2

Question P1 :

Pour faire joli, on ne mettra pas de caractère à la racine. Cela donne un deuxième type.

```
type noeud_trie_plus = N of char * bool * int * noeud_trie_plus list;;

type trie_plus = R of int * noeud_trie_plus list;;
```

Question P2 :

```
let creer_trie_plus () = R (0, []);;

let char_associe (N(c, _, _, _)) = c;;
(* Oui, cette syntaxe est légitime, mais rare, avertissement s'il y a plusieurs constructeurs ! *)

let char_present c l = List.exists (fun noeud -> char_associe noeud = c) l;;

let rec tester_noeud mot pos (N(_, b, _, l)) =
  if pos = String.length mot then b
  else List.exists (fun fils -> char_associe fils = mot.[pos] && tester_noeud mot (pos + 1) fils) l;;

let tester_trie_plus mot trie =
  if mot = "" then failwith "Mot vide";
  match trie with R(_, liste) ->
  List.exists (fun fils -> char_associe fils = mot.[0] && tester_noeud mot 1 fils) liste;;

let rec chaine_mot mot pos =
  if pos = String.length mot - 1 then N(mot.[pos], true, 1, [])
  else N(mot.[pos], false, 1, [chaine_mot mot (pos+1)]);;

let rec ajouter_noeud mot pos (N(c, b, n, l)) =
  if pos = String.length mot then N(c, true, n+1, l)
  else if char_present mot.[pos] l
  then
    let f = fun noeud ->
      if char_associe noeud = mot.[pos]
      then ajouter_noeud mot (pos+1) noeud
      else noeud
    in N(c, b, n+1, List.map f l)
  else N(c, b, n+1, (chaine_mot mot pos)::l);;

let ajouter_trie_plus mot trie =
  if mot = "" then failwith "Mot vide";
  if tester_trie_plus mot trie then failwith "Doublon";
  (* Test pour éviter d'ajouter un alors qu'il y aura une erreur ! *)
  match trie with R(n, liste) ->
    if char_present mot.[0] liste
    then
      let f = fun noeud ->
        if char_associe noeud = mot.[0]
        then ajouter_noeud mot 1 noeud
        else noeud
      in R(n+1, List.map f liste)
    else R(n+1, (chaine_mot mot 0)::liste);;
```

Question P3 :

```
let rec plus_petit_prefixe_noeud mot taille (N(_, _, n, l)) =
 if n = 1 then String.sub mot 0 taille else if taille = String.length mot then mot
 else let rec mouline liste = match liste with
  | [] -> failwith "On avait dit que le mot y était"
  | noeud::_ when char_associe noeud = mot.[taille] -> plus_petit_prefixe_noeud mot (taille+1) noeud
  | _::reste -> mouline reste
  in mouline l;;

let rec plus_petit_prefixe mot (R(_, l)) =
  if mot = "" then failwith "Mot vide";
  let rec mouline liste = match liste with
  | [] -> failwith "On avait dit que le mot y était"
  | noeud::_ when char_associe noeud = mot.[0] -> plus_petit_prefixe_noeud mot 1 noeud
  | _::reste -> mouline reste
  in mouline l;;
```

Question P4 :

```
let raccourcir tab =
  let n = Array.length tab in
  let rec construit_trie i =
  if i = n then creer_trie_plus ()
  else ajouter_trie_plus tab.(i) (construit_trie (i+1))
  in let trie = construit_trie 0
  in Array.init n (fun i -> plus_petit_prefixe tab.(i) trie);;
```

Question P5 :

Avec la restriction demandée, oui. Dans l'exemple des jours de la semaine, il y aura `"Ma"` et `"Me"` dans le résultat. Si le raisonnement avait pu se faire par élimination, on aurait pu imaginer que la présence de `"Ma"` permette de laisser `"M"` sans risque de confusion, ou vice-versa.

Question P6 :

On pourrait utiliser simplement deux tries indépendants, un pour le premier mot de chaque couple et un pour le deuxième.

Seulement, les calculs seraient a priori aussi indépendants et les quatre couples `("AAAAA", "CCCCC")`, `("AAAAA", "DDDDD")`, `(BBBBB", "CCCCC")` et `("BBBBB", "DDDDD")` ne pourraient pas être raccourcis.

En utilisant par exemple un trie où chaque nœud a deux listes de descendants, les nœuds où on ajoute un caractère du premier mot et les nœuds où on ajoute un caractère du deuxième mot, on pourra observer qu'une fois qu'on a ajouté le premier `'A'` et le premier `'C'`, il n'y a plus d'ambigüité.

En contrepartie, chaque couple serait ajouté un nombre exponentiel de fois, car avant l'ajout de chaque lettre on peut partir dans une liste ou l'autre (le nombre exact est la taille d'un mot parmi la somme des tailles des deux mots). Il faut absolument procéder à une confluence afin que le nombre d'ajouts soit le produit des tailles.

En pratique, on va plutôt utiliser un dictionnaire indexé par des couples de mots et dont les valeurs seront la liste des couples de mots (qu'on pourrait représenter par une position du couple dans le tableau si le dictionnaire est local à la fonction de la question suivante) qui peuvent s'obtenir en complétant le couple de mots qui constitue la clé.

Ici, la complexité sera polynomiale en temps et en espace, et il semble difficile de faire mieux.

Question P7 :

```
let creer_dict () = Hashtbl.create 8;; (* Juste pour le cas où on change d'avis... *)

let ajouter_dict dico (mot1, mot2) =
  for i = 0 to String.length mot1 do
    let pref1 = String.sub mot1 0 i in
    (* On pourrait accélérer en partant de "" et en ajoutant une lettre à chaque tour,
    mais la complexité de ^ n'est pas non plus géniale a priori. *)
    for j = 0 to String.length mot2 do
      let pref2 = String.sub mot2 0 j in
      match Hashtbl.find_opt dico (pref1, pref2) with
      | None -> Hashtbl.add dico (pref1, pref2) [(mot1, mot2)]
      | Some l -> Hashtbl.replace dico (pref1, pref2) ((mot1, mot2)::l)
    done
  done;;

exception Continuer;;

let raccourcir_couples tab =
  let n = Array.length tab in
  let dico = creer_dict () in
  for ind = 0 to n-1 do ajouter_dict dico tab.(ind) done;
  let rep = Array.make n ("", "") in
  for ind = 0 to n-1 do
    let taille1 = String.length (fst tab.(ind)) in
    let taille2 = String.length (snd tab.(ind)) in
    try
      for taille = 0 to taille1 + taille2 do
        for i = 0 to taille do
          if i <= taille1 && taille - i <= taille2 then
            let sous_mot1 = String.sub (fst tab.(ind)) 0 i
            and sous_mot2 = String.sub (snd tab.(ind)) 0 (taille - i) in
            let possibilites = Hashtbl.find dico (sous_mot1, sous_mot2) in
            if List.length possibilites = 1 then
            begin
              rep.(ind) <- (sous_mot1, sous_mot2);
              raise Continuer
            end
        done
      done; failwith "On dirait qu'il y a un doublon !"
    with Continuer -> ()
  done;
  rep;;
```

Question P8 :

Soient les couples `("AA", "CC")` et `("AB", "CB")`. On peut raccourcir le premier couple en `("AA", "C")` ou en `("A", "CC")`, ce qui contredit l'unicité. Idem pour le deuxième couple.