DS MP2I : Arbres

Solution. □

Toujours indiquer le type et l'objectif des fonctions auxiliaires. Tout code long doit être expliqué.

1 Tableau associatif

Le haché *DJB2 (Dan Bernstein)* est un algorithme simple de hachage de chaîne de caractères datant de 1991. Il associe un entier positif à toute chaîne de caractères.

On donne, pour l'implémenter, le code suivant :

```
let hash_djb2 (s : string) : int =
  let h = ref 5381 in
  String.iter (fun c ->
    h := (!h lsl 5) + !h + Char.code c
    )
    s;
  !h land max_int;;
```

Dans ce code, `Char.code c` désigne le code ASCII du caractère `c` (ainsi, `Char.code 'a'` vaut 97).

```
# hash_djb2 "toto";;
- : int = 6385734411
```

Question 1.

L'appel `hash_djb2 s` effectue AVANT la dernière ligne un calcul $P(x)$ où P est un certain polynôme dont le dégré dépend de n (la longueur de s) et x une valeur fixée. La première partie du code est donc une implémentation de l'algorithme de Horner.

Identifier le polynôme P et la valeur de x.

Solution.

$$P(x) = 5381x^{|s|} + \sum_{i=0}^{|s|-1} \text{code}(s_i)x^{|s|-1-i}$$

Ici $x = 33 = 2^5 + 1$. car

```
!h lsl 5  + !h = !h * (1 lsl 5 + 1)
```

□

Question 2.

Expliquer la dernière ligne : on peut la comprendre comme l'ajout d'une certaine valeur à `!h` si celui-ci est négatif et d'une autre valeur si `!h` est positif. Mais lesquelles ?

Solution. On obtient un nombre positif : on ne garde que les bits de `!h` après le bit de poids fort : le bit de signe de `max_int` est 0 et tous les autres valent 1.

Comme les entiers OCaml sont codés sur 63 bits (le 64ème étant réservé pour stocker des métas-informations par OCaml), l'opération finale revient à ajouter 2^{62} à `!h` lorsqu'il est négatif. Et à ne rien faire (ou à ajouter 0) sinon. □

On donne les types suivants :

```
type key = string
 (*Un bucket est la collection d'éléments associés à un même indice.*)
 type 'v bucket = (key * 'v) list
 type 'v sht = {
     mutable buckets : 'v bucket array;
     hash : key -> int;
     mutable size : int;  (* nombre de seaux *)
   }
 (* Seuil de charge à partir duquel on redimensionne *)
 let max_load_factor = 2.0 /. 3.0
```

Question 3.

Écrire la fonction
`create (hash : 'a -> int ) (initial_capacity : int) :'a sht`. Elle prend en paramètre une fonction `hash` de hachage et une indication de la capacité du tableau de stockage. Elle renvoie un dictionnaire vide. Si le second paramètre est inférieur à 1, on n'en tient pas compte et le tableau de stockage prend une taille de 1.

Solution. Code

```
let create hash initial_capacity =
    let cap = max 1 initial_capacity in
    { buckets = Array.make cap []; size = 0; hash }
```

□

Lorsque la table devient trop pleine, on double la capacité, puis on réinsère chaque élément dans la nouvelle table. La constante `max_load_factor` est une variable globale telle que le redimensionnement est déclenché si le ratio `nombre d'associations / capacité` est supérieur à cette quantité. On prend $\frac{2}{3}$ comme en `Python`[1] pour cette valeur.

Question 4.

Écrire la fonction `resize (tbl : 'a sht) : unit` qui double la taille du tableau de stockage et réinsère les associations **dans l'ordre où elles étaient auparavant.**.

Solution. Code

```
(* Redimensionnement : double la capacité et réinsère tous les
     éléments, en préservant l'ordre de chaque bucket *)
  let resize tbl =
    let old = tbl.buckets in
    let new_cap = max 1 (Array.length old * 2) in
    let new_buckets = Array.make new_cap [] in
    Array.iter (fun bucket ->
        (* On parcourt chaque bucket à l'envers pour conserver l'ordre
           original lors des réinsertions *)
        List.iter (fun (k, v) ->
            let idx = (tbl.hash k) mod new_cap in
            new_buckets.(idx) <- (k, v) :: new_buckets.(idx)
          ) (List.rev bucket)
      ) old;
    tbl.buckets <- new_buckets
```

□

1. En OCaml c'est plutôt $\frac{1}{2}$.

Question 5.

Écrire la fonction `add (tbl : 'a sht) (k : key) (v: 'a) : unit` qui ajoute une association au dictionnaire. Éventuellement, si le ratio taille/capacité est trop élevé, on effectue d'abord un redimensionnement. La fonction `add` se contente d'ajouter une association `(k,v)`. Il est possible qu'une autre association de même clé se trouve plus loin dans le seau.

Solution. Code

```
(* Ajout d'une paire clé/valeur, avec redimensionnement si
    nécessaire *)
 let add tbl key value =
   let cap = Array.length tbl.buckets in
   if float tbl.size >= max_load_factor *. float cap then resize tbl;
   let idx = (tbl.hash key) mod (Array.length tbl.buckets) in
   tbl.buckets.(idx) <- (key, value) :: tbl.buckets.(idx);
   tbl.size <- tbl.size + 1
```

□

Question 6.

Écrire la fonction `find (tbl : 'a sht) (k:key) : 'a` qui renvoie la valeur associée à la clé passée en paramètre et soulève `Not_found` sinon. Si plusieurs associations de même clé se trouvent dans le seau, alors `find` renvoie la valeur de la première rencontrée (donc la plus récemment ajoutée) sans se préoccuper des autres (celles qui sont plus loin dans le seau).

Solution. Code

```
(* Recherche d'une valeur par clé, lève Not_found si absente *)
  let find tbl key =
    let idx = (tbl.hash key) mod (Array.length tbl.buckets) in
    List.assoc key tbl.buckets.(idx)
```

□

2 Peignes

On définit le type arbre de la manière suivante :

```
type arbre =
   |Feuille of int
   |Noeud of arbre * arbre ;;
```

On dit qu'un arbre est un *peigne* si tous les nœuds internes (à l'exception éventuelle de la racine) ont au moins une feuille pour fils. On dit qu'un peigne est un peigne *strict* si sa racine a au moins une feuille pour fils, ou s'il est réduit à une feuille. On dit qu'un peigne est *rangé* si le fils droit d'un nœud interne est toujours une feuille. Un arbre réduit à une feuille est considéré comme un peigne rangé.

Question 7.

Représentation.

1. Dessiner un peigne rangé à 5 feuilles ;
2. puis donner son code OCaml.

Solution. On a

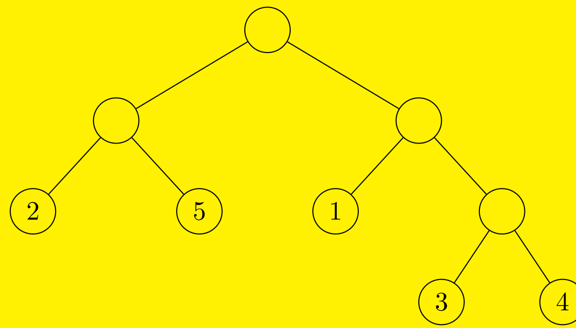
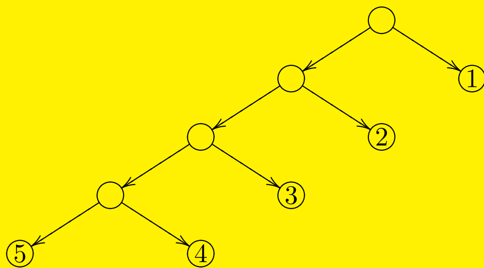


Figure 1 – Un peigne à cinq feuilles



Et voici son codage en OCAML :

```
let r = Noeud(Noeud(Noeud(Noeud(Feuille 5,Feuille 4),
      Feuille 3),Feuille 2),Feuille 1);;
```

□

Question 8.

Quelle est la hauteur d'un peigne rangé à n feuilles ? On justifiera la réponse.

Solution.

Remarque. Dans vos rédactions, indiquez bien à quel endroit vous appliquez l'hypothèse d'induction ou de récurrence.

Par Induction Pour un arbre A on note $h(A)$ sa hauteur et $f(A)$ son nombre de feuilles.
Cas de base pour l'arbre feuille : OK.
Soit A un peigne rangé dont le fils gauche G vérifie l'hypothèse d'induction.
Alors $h(G) = f(G) - 1$ (par HI). Or $f(A) = f(G) + 1$ et

$$h(A) = \max(h(G), 1) + 1 = h(G) + 1 = f(G) - 1 + 1 = f(G) = f(A) - 1$$

□

Question 9.

Ecrire une fonction `est_range : arbre → bool` qui renvoie `true` si l'arbre donné en argument est un peigne rangé.

Solution. Code

```
let rec est_range a =
  match a with
  | Feuille _ -> true
  | Noeud(g, Feuille _) -> est_range g
  | _ -> false;;
```

□

Question 10.

Caractérisation des peignes.

1. Écrire une fonction `est_peigne_strict : arbre → bool` qui renvoie `true` si l'arbre donné en argument est un peigne strict.
2. En déduire une fonction `est_peigne : arbre → bool` qui renvoie `true` si l'arbre donné en argument est un peigne.

Solution. Voici

```
let rec est_peigne_srict a =
  match a with
  | Feuille _ -> true
  | Noeud(g, Feuille _) -> est_peigne_srict g
  | Noeud(Feuille _, d) -> est_peigne_srict d
  | _ -> false;;

let est_peigne a =
  match a with
  | Feuille _ -> true
  | Noeud(g, d) -> est_peigne_srict g && est_peigne_srict d;;
```

□

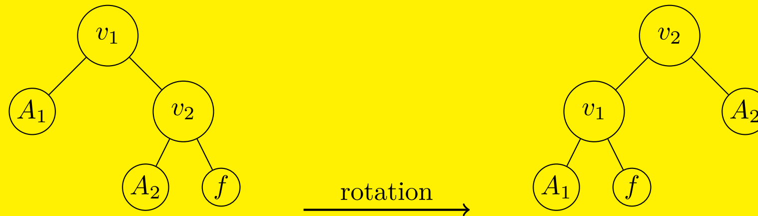


Figure 2 – Une rotation

On souhaite maintenant ranger un peigne donné. Supposons que le fils droit D de sa racine ne soit pas une feuille. Notons A_1 le sous-arbre gauche de la racine, f l'une des feuilles du noeud D et A_2 l'autre sous-arbre du noeud D. On va utiliser l'opération de *rotation* qui construit un nouveau peigne où

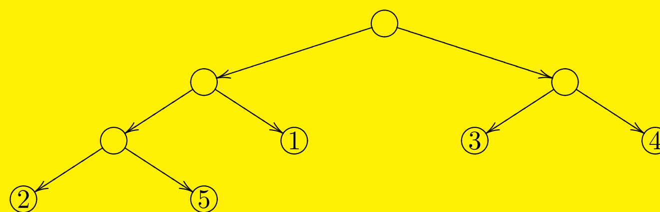
- le fils droit de la racine est le sous-arbre A_2 ;
- le fils gauche de la racine est un noeud de sous-arbre gauche A_1 et de sous-arbre droit la feuille f.

Question 11.

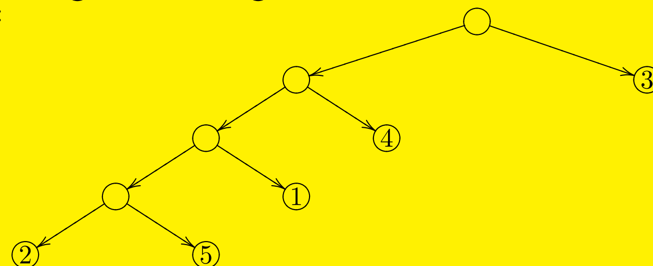
Application de rotations.

1. Dessiner le résultat d'une rotation sur l'arbre de la figure 1
2. Appliquer encore une rotation.

Solution. Après une rotation :



Après deux rotations : □



Question 12.

Ecrire une fonction `rotation : arbre → arbre` qui effectue une rotation à la racine de l'arbre. La fonction renverra l'arbre initial si une rotation n'est pas possible.

Solution. Pas de difficulté, mais il faut faire un filtrage un peu détaillé selon que le fils droit de la racine à une feuille à droite ou à gauche :

```
let rotation a =
  match a with
  | Noeud (a1, Noeud (a2, Feuille x)) ->
     Noeud(Noeud (a1, Feuille x),a2)
  | Noeud (a1, Noeud (Feuille x, a2)) ->
     Noeud(Noeud (a1, Feuille x),a2)
  | _ -> a ;;
```

□

Question 13.

Ecrire une fonction `rangement : arbre → arbre` qui range un peigne donné en argument, c'est à dire qu'il renvoie un peigne rangé ayant les mêmes feuilles que celui donné en argument. La fonction renverra l'arbre initial si celui-ci n'est pas un peigne.

Solution. Code

```
let rangement a =
   (* pour prévenir les boucles infinies *)
   if not (est_peigne a) then a
   else
     let rec _ranger a = match a with
       | Feuille _ -> a
       | Noeud (g, (Feuille _ as d)) -> Noeud (_ranger g, d)
       | _ -> _ranger (rotation a)
     in
     _ranger a;;
```

□

3 Centre et diamètres

On donne les types

```
   (* Définition de l'arbre binaire *)
type 'a tree =
  | Empty
  | Node of 'a * 'a tree * 'a tree

(* Pour représenter le centre de l'arbre *)
type 'a center =
  | Single of 'a         (* centre unique *)
  | Double of 'a * 'a      (* deux centres *)
```

Dans cet exercice, on fixe la *hauteur* du nœud vide à 0. La *hauteur* d'un nœud est la longueur (exprimée en nombre de nœuds) du chemin le plus long entre ce nœud et les feuilles qui en descendent.

Pour définir un *chemin*, on considère les arbres comme des graphes non orientés : un chemin se parcourt dans les deux sens (il n'y a plus de père ni de fils seulement des *voisins*). Dans ce contexte, un *diamètre* d'un arbre est un plus long chemin entre deux nœuds de l'arbre.

Question 14.

Établir qu'un arbre peut avoir plusieurs diamètres. C'est à dire qu'il existe deux chemins n'empruntant pas les même sommets et dont la longueur est maximale.

Solution. Dans un arbre parfait de hauteur ≥ 2, les chemins des deux feuilles les plus à gauche vers les deux feuilles les plus à droite sont des diamètres. Il y a donc au moins 4 diamètres □

Question 15.

Établir qu'il existe un arbre dont aucun diamètre ne passe par la racine.

Solution. Ajoutons une racine qui aurait pour fils gauche l'arbre de la figure 3 et pas de fils droit.
Alors aucun diamètre ne passe par la racine. □

Un *centre* d'un arbre est un nœud tel que la distance maximale (en nombre de nœuds) entre ce nœud et tout autre nœud de l'arbre est minimal (cf figure 3).

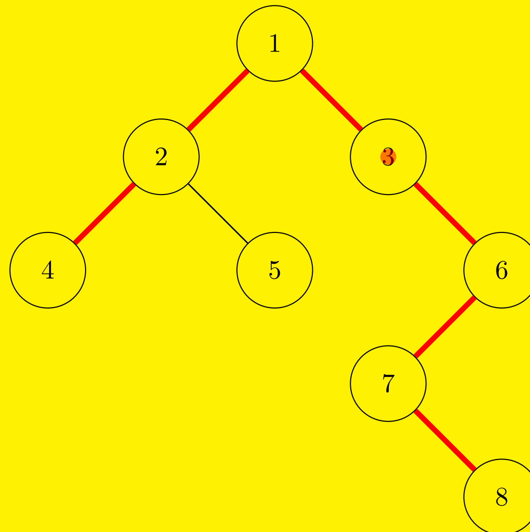


Figure 3 – Le diamètre est 6, le centre est le nœud d'étiquette 3

Listing 1 – Implémentation de l'arbre de la figure 3

```
let example_tree =
  Node (1,
    Node (2,
      Node (4, Empty, Empty),
      Node (5, Empty, Empty)
    ),
    Node (3,
      Empty,
      Node (6, Node (7, Empty, Node (8, Empty, Empty)), Empty)
    )
  );;
```

Une propriété bien connue est que tout arbre possède soit un centre unique (si la longueur d'un diamètre est impaire), soit exactement deux centres adjacents (si la longueur d'un diamètre est paire). Selon la parité de la longueur, le centre est le point au milieu d'un diamètre ou les deux points au milieu.

Remarque. Cette propriété se prouve en considérant qu'un arbre est un graphe non orienté connexe et acyclique et en considérant la suite des graphes obtenus en partant de l'arbre initial et en supprimant toutes les feuilles du graphe précédent.

Une approche classique pour trouver le centre consiste à :

— Calculer le diamètre de l'arbre, c'est-à-dire le chemin le plus long (en nombre de nœuds) reliant deux nœuds quelconques.

— Le ou les centres se trouvent alors au milieu de ce chemin. Plus précisément, si le chemin comporte un nombre impair de nœuds, le centre est unique (le nœud médian) ; s'il comporte un nombre pair de nœuds, il y a deux centres (les deux nœuds centraux).

Pour le calcul du diamètre, on utilise une approche par programmation dynamique qui, pour chaque nœud, calcule :

— La longueur (en nombre de nœuds) de la plus longue branche descendant du nœud (la hauteur) et le chemin correspondant.

— Un diamètre et sa longueur, ce diamètre pouvant être soit dans le sous-arbre gauche, soit dans le sous-arbre droit, soit passer par le nœud courant (en combinant la plus grande branche gauche et la plus grande branche droite).

Question 16.

Écrire une fonction `diameter (t: 'a tree) : int * 'a list * int * 'a list` qui prend en paramètre un arbre et renvoie un quadruplet `(height, height_path, diam, diam_path)` où :

— `height` désigne la hauteur (nombre de nœuds) du chemin descendant le plus long partant de t ;

— `height_path` est le chemin correspondant (du nœud courant jusqu'à la feuille la plus éloignée) donné sous forme de liste d'étiquettes ;

— `diam` désigne le diamètre (en nombre de nœuds) de l'arbre t ;

— `diam_path` est le chemin correspondant au diamètre donné sous forme de liste d'étiquettes.

```
# diameter example_tree;;
- : int * int list * int * int list =
(5, [1; 3; 6; 7; 8], 7, [4; 2; 1; 3; 6; 7; 8])
```

On privilégiera une approche EN UN SEUL PARCOURS.

Solution. Code

```
let rec diameter t =
  match t with
  | Empty -> (0, [], 0, [])
  | Node(v, left, right) ->
      let (hl, hpath_left, diam_left, diam_path_left) = diameter left in
      let (hr, hpath_right, diam_right, diam_path_right) = diameter right in
      (* Calcul de la hauteur et du chemin descendant
         le plus long depuis le noeud courant *)
      let (h, hpath) =
        if hl >= hr then (hl + 1, v :: hpath_left)
        else (hr + 1, v :: hpath_right)
      in
      (* Candidat pour un diamètre passant par le noeud courant *)
      let candidate_diam = hl + hr + 1 in
      let candidate_path =
        let left_path = if left = Empty then [] else List.rev hpath_left in
        let right_path = if right = Empty then [] else hpath_right in
        left_path @ [v] @ right_path
      in
      (* Choix du meilleur diamètre *)
      let (best_diam, best_path) =
        if candidate_diam >= diam_left && candidate_diam >= diam_right then
          (candidate_diam, candidate_path)
        else if diam_left >= diam_right then
          (diam_left, diam_path_left)
        else
          (diam_right, diam_path_right)
      in
      (h, hpath, best_diam, best_path);;
```

□

Question 17.

Écrire une fonction `center (t: 'a tree) : int list` qui, étant donné un arbre binaire, retourne l'étiquette (ou le couple d'étiquettes) du centre (ou des deux centres) sous la forme d'une valeur de type `centre`.

```
# center example_tree;;
- : int center = Single 3
```

Solution. Code

```


let rec tronque k l =(*enlève les k premiez éléments*)
  match l, k with
  | l, 0 -> l
  | [], _ -> []
  | _ :: q, k when k > 0 -> tronque (k - 1) q
  | _ -> invalid_arg "tronque"
;;


(*V1*)
let center t =
  let (_, _, n, diam_path) = diameter t in
  if n = 0 then failwith "Arbre vide"
  else
    match n mod 2, tronque ((n - 1) / 2) diam_path with
    | 1, x :: _ -> Single x
    | 0, c1 :: c2 :: _ -> Double (c1, c2)
    | _ -> failwith "center"
;;
(*
paramètre de tronque :
si n impair, (n - 1) / 2 = n / 2
si n pair, (n - 1) / 2 = (n / 2) - 1
*)


(*V2 en parcourant le diamètre et son inverse simultanément*)
let center t =
  let (_, _, _, diam_path) = diameter t in
  let rev_diam_path = List.rev diam_path in

  let rec aux l1 l2 =
    match l1, l2 with
    | [], [] -> failwith "Arbre vide"
    | [x], [x'] when x = x' ->
        Single x
    | x :: y :: q, x' :: y' :: q' ->
        if x = x' then
          Single x
        else if x = y' then
          Double (x, y)
        else
          aux (y :: q) (y' :: q')
    | _ ->
        failwith "diamètre mal formé"
  in
  aux diam_path rev_diam_path
;;
```

□