DM1: Dictionnaires

MPSI Lycée Pierre de Fermat

Ce devoir est à rendre pour le mercredi 3 juin / jeudi 4 juin. Il est à rédiger sur papier, mais il est fortement conseillé de travailler aussi sur ordinateur, afin de pouvoir tester et valider les différentes fonctions que vous écrirez.

1. Structure abstraite

On fixe deux ensembles K et V. On appelle K l'ensemble des **clés** et V l'ensemble des **valeurs**. Un dictionnaire est une structure abstraite de données servant à représenter des **associations** de la forme `clé` $\mapsto$ `valeur`. Un dictionnaire contient plusieurs clés, et à chacune est associée une unique valeur. De nombreux langages, comme Python, permettent aux utilisateurs d'utiliser nativement des dictionnaires, sans avoir à utiliser de librairie externe:

```
1  # création d'un dictionnaire associant à chaque animal son cri
2  d = dict() # créer un dictionnaire vide
3  d["chat"] = "Miaou" # ajoute à d l'association (chat -> Miaou)
4  d["chien"] = "Ouaf"
5  d["carpe"] = ""
6  x = d["chien"] # x vaut "Ouaf"
7  # on peut directement définir d en une ligne:
8  d = {"chat": "Miaou", "chien": "Ouaf", "carpe": ""}
9  # associer une valeur à une clé déjà présente écrase l'ancienne valeur
10 d["chat"] = "Miau" # "chat" n'est plus associé à "Miaou" mais à "Miau"
```

Définition 1. Un dictionnaire stocke des couples $(k, v) \in K \times V$, de telle sorte que pour une clé $k \in K$ donné, il y a au plus un couple dans le dictionnaire dont k est la clé. On peut donc chercher dans un dictionnaire l'existence d'une clé particulière, modifier la valeur associée, et supprimer une clé et sa valeur associée:

- **dict**() crée un dictionnaire vide;
- **ajout**(D, k, v) associe la valeur v à la clé k dans le dictionnaire D. Si la clé k est déjà associée à une autre valeur v', cette ancienne association est supprimée et remplacée par l'association (k, v).
- **contient**(D, k) renvoie un booléen déterminant si la clé k est dans le dictionnaire D.
- **recherche**(D, k) renvoie la valeur associée à la clé k dans le dictionnaire D. k doit être une clé valable de D.
- **supprime**(D, k) supprime la clé k du dictionnaire D, ainsi que la valeur qui y était associée.

Q1. Pour chacune des cinq opérations présentées ci-dessus, dire si c'est un constructeur, un accesseur, ou un transformateur.

L'objectif de ce devoir est d'implémenter les dictionnaires de deux façons. On s'intéresse dans une première partie aux **arbres binaires de recherche**, qui permettent de conserver de l'ordre parmi les clés du dictionnaire, et donc de réaliser certaines opérations plus efficacement. Puis, on étudie les **tables de hachage**, des structures particulièrement efficaces utilisées pour implémenter les dictionnaires dans de nombreux langages, dont Python et OCaml.

2. Arbres binaires de recherche

Soit E un ensemble muni d'un ordre total. On considère l'ensemble $\mathcal{A}_E$ des arbres binaires étiquetés par E:

- $V \in \mathcal{A}_E$, c'est l'arbre vide, de hauteur $h(V) = -1$;
- Pour $G, D \in \mathcal{A}_E, N(x, G, D) \in \mathcal{A}_E$, c'est un arbre de hauteur $1 + \mathbf{max}(h(G), h(D))$.

Définition 2. Un arbre $A \in \mathcal{A}_E$ étiqueté par E est un ***arbre binaire de recherche*** s'il est vide, ou si il est de la forme $N(x, G, D)$ avec:

- pour **toute** étiquette x_g dans G, $x_g < x$;
- pour **toute** étiquette x_d dans D, $x_d > x$.
- G, D sont des arbres binaires de recherche;

Dans la suite, on notera **ABR** pour "arbre binaire de recherche".

Q2. Les arbres A_1 et A_2 suivants sont-ils des ABR (pour l'ordre alphabétique standard) ?

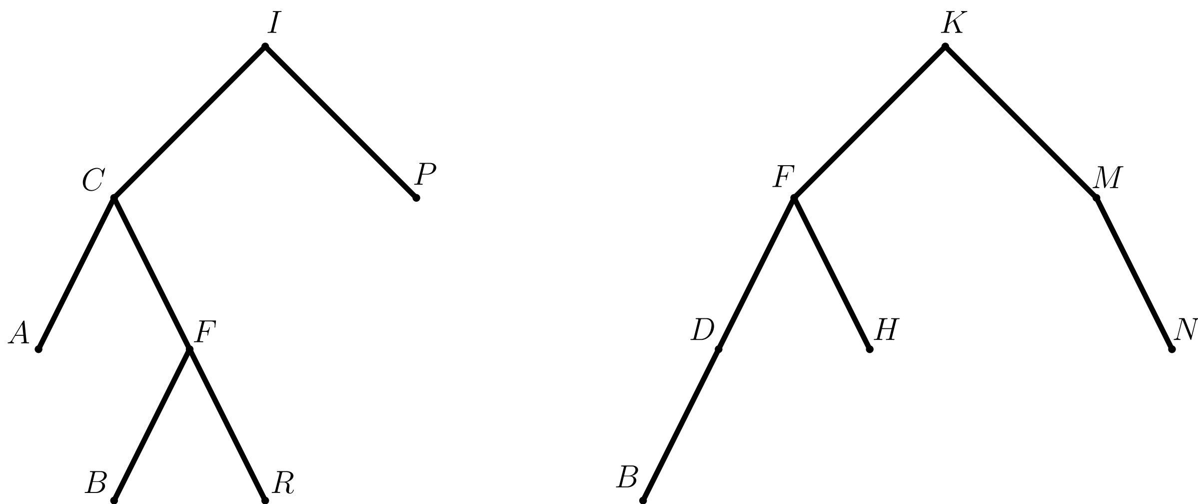


Figure 1: Arbres A_1 (à gauche) et A_2 (à droite).

On considère dans la suite le type OCaml suivant:

```
1 type 'a abr = V | N of 'a * 'a abr * 'a abr
```

Q3. Écrire une fonction `min_abr: 'a abr -> 'a` renvoyant le minimum d'un ABR non-vide. *On pourra remarquer que le minimum d'un ABR se trouve le plus à gauche possible.*

Q4. Donner la complexité de la fonction précédente en fonction de la hauteur h de l'arbre. On justifiera brièvement.

On suppose avoir écrit une fonction `max_abr` analogue à celle de la question précédente, permettant d'obtenir le maximum d'un ABR non vide. On propose le code suivant pour déterminer si un arbre binaire de recherche est valide:

```
(* renvoie true si a est un ABR valide, false sinon. *)
let rec est_abr (a: 'a abr) : bool =
  match a with
  | V -> true
  | N (x, V, V) -> true
  | N (x, g, V) -> est_abr g && max g < x
  | N (x, V, d) -> est_abr d && x < min d
  | N (x, g, d) -> est_abr g && est_abr d &&
                   max g < x && x < min d
```

On considère la famille d'arbres $(P_n)_{n\in\mathbb{N}}$ définie comme suit:

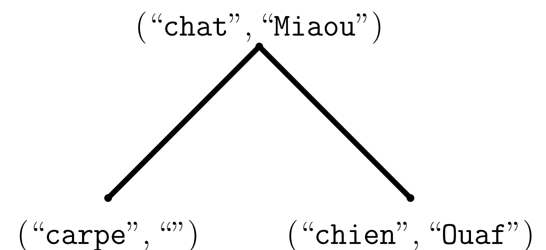
- $P_0 = \mathbf{V}$
- $P_{n+1} = \mathbf{N}(n, P_n, \mathbf{V})$ pour $n \in \mathbb{N}$

On appelle P_n le **peigne gauche** d'ordre n.

Q5. Montrer que le temps d'exécution de `est_abr` sur P_n est quadratique en n.

Q6. Montrer par induction structurelle qu'un arbre est un **ABR** si et seulement si son parcours infixe est strictement croissant, et en déduire une version linéaire de `est_abr`.

On utilise les ABR pour implémenter les dictionnaires, en stockant aux noeuds les couples (clé, valeur), et en considérant l'ordre des clés. Par exemple, le dictionnaire créé dans le code Python au début de la partie 2 pourra être représenté par l'ABR ci-contre.



On utilise donc le type suivant:

```
type ('a, 'b) dict = ('a * 'b) abr

let dict_vide = V
```

L'utilité des arbres binaires de recherche est que ses noeuds servent d'aiguillage pour trouver une valeur. Par exemple, la fonction `dict_contient` peut s'écrire comme suit:

```
let rec dict_contient (a: ('a * 'b) abr) (k: 'a) : 'b =
  match a with
  | V -> false
  | N ((k', v), g, d) ->
    if k = k' then true else
    if k < k' then dict_contient g k
    else           dict_contient d k
```

En effet, lorsque l'on cherche la clé k dans un arbre binaire A dont la racine est un couple (k', v), avec $k < k'$, alors les conditions des ABR assurent que k ne se trouve pas dans le sous-arbre droit de A, et inversement si $k > k'$, alors k ne se trouve pas dans le sous-arbre gauche.

Q7. Justifier que la complexité de `dict_contient` est $\mathcal{O}(h)$, avec h la hauteur de l'arbre.

Q8. Selon le même principe, implémenter `dict_recherche`, en $\mathcal{O}(h)$.

Q9. Implémenter la fonction d'ajout `dict_ajoute`, en $\mathcal{O}(h)$ également. On rappelle que l'ajout d'un couple (k, v) doit **remplacer** un éventuel couple dont k est la clé et qui serait déjà présent dans le dictionnaire.

On s'intéresse maintenant à la suppression. On remarque que dans un ABR non vide $A = N(x, G, D)$, si l'on veut supprimer l'élément à la racine, x, on peut toujours le remplacer par le plus grand élément de G. Par exemple:

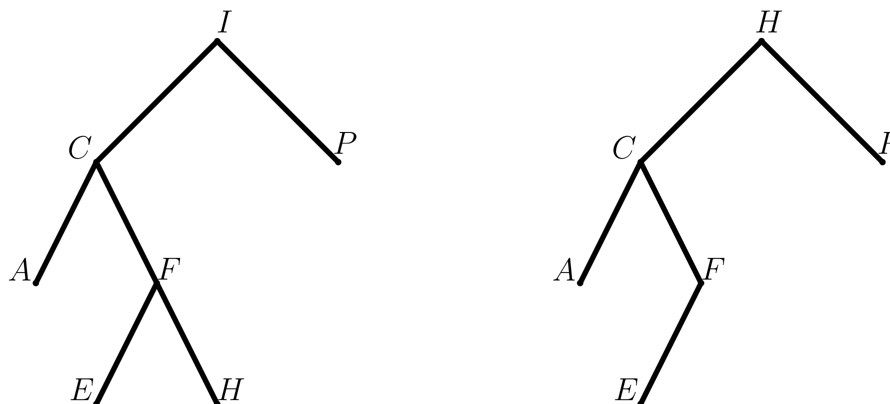


Figure 2: Arbre A_3 avant et après suppression de l'étiquette I.

Q10. En notant A'_3 l'arbre obtenu après suppression dans la Fig.2, dessiner l'arbre obtenu après avoir supprimé H de $A'3$.

La suppression va passer par l'écriture d'une fonction auxiliaire ayant la spécification suivante:

```
(* extraire_max a renvoie un couple (a', m) avec
   - m le maximum de a
   - a' l'ABR obtenu en ayant retiré m de a
   Préconditions: a doit être un ABR non vide. *)
extraire_max: 'a abr -> ('a abr * 'a)
```

Q11. Implémenter `extraire_max`.

Q12. En utilisant `extraire_max`, implémenter la fonction `dict_supprime`.

Q13. Donner les complexités de `extraire_max` et `dict_supprime` en fonction de h la hauteur de l'arbre en entrée.

3. Efficacité des ABR

Les complexités des opérations sur les ABR dépendent non pas de leur taille mais de leur hauteur.

Q14. Soit $n \in \mathbb{N}$. Proposer deux ABR A_n et B_n dont les étiquettes sont $1, 2, \ldots n$, avec A_n de hauteur $n - 1$ et B_n de hauteur au plus $\lfloor \log_2(n + 1) \rfloor$.

En pratique, il existe plusieurs méthodes permettant d'équilibrer les ABR, afin de garantir que leur hauteur reste proportionnelle à $\log_2(n)$ *au fil des insertions et suppressions. Les deux*

méthodes les plus connues sont les arbres rouge-noir et les arbres AVL.

Un avantage des ABR est que grâce à l'ordre de stockage des clés, certaines opérations sont réalisables de manière très efficace. Par exemple, il est possible de récupérer toutes les clés se situant dans un certain **intervalle**.

Q15. Écrire une fonction `range_query` prenant en entrée un dictionnaire A, et deux bornes i, s, et renvoyant l'ensemble des clés k de A vérifiant $i \leq k \leq s$.

Q16. Évaluer la complexité de la fonction `range_query`, en fonction de h la hauteur de l'arbre et de p le nombre d'éléments dans le résultat de la requête.

Q17. Étant donné un dictionnaire A, et une clé k pas forcément présente dans A, proposer un algorithme permettant de trouver la clé de A la **plus proche** de k, et en donner la complexité.

4. Tables de hachage

.

Les tables de hachage sont une alternative aux ABR pour implémenter les dictionnaires. On se propose de les utiliser pour créer une structure **mutable**, ayant l'interface suivante:

```
(* dictionnaire dont les clés sont 'a et les valeurs 'b *)
type ('a, 'b) hashtbl

(* Renvoie un dictionnaire vide *)
hashtbl_vide : unit -> ('a, 'b) hashtbl

(* hashtbl_ajoute d k v modifie d pour y ajouter l'association k -> v *)
hashtbl_ajoute : ('a, 'b) hashtbl -> 'a -> 'b -> unit

(* hashtbl_contient d k renvoie true si d contient k, false sinon *)
hashtbl_contient : ('a, 'b) hashtbl -> 'a -> bool

(* hashtbl_recherche d k renvoie la clé associée à k dans d.
   Précondition: d contient k *)
hashtbl recherche: ('a, 'b) hashtbl -> 'a -> 'b

(* hashtbl_supprime d k supprime la clé k de d *)
hashtbl_supprime: ('a, 'b) hashtbl -> 'a -> unit
```

Notons la différence avec la structure immuable implémentée dans la première partie: on peut **modifier** une table de hachage en changeant son état. Voici le même programme, écrit à gauche avec un dictionnaire immuable, à droite avec un dictionnaire mutable:

```
(* IMMUABLE *)
let d = dict_vide;;
let d = dict_ajoute d "chat" "miaou";;
let d = dict_ajoute d "chien" "ouaf";;
assert (dict_contient d "chat");;
assert (dict_recherche d "chat" = "miaou"
    );;
let d = dict_supprime d "chat";;
assert (not(dict_contient d "chat"));;
```

```
(* MUTABLE *)
let d = hashtbl_vide ();;
hashtbl_ajoute d "chat" "miaou";;
hashtbl_ajoute d "chien" "ouaf";;
assert (hashtbl_contient d "chat");;
assert (hashtbl_recherche d "chat" = "
    miaou");;
hashtbl_supprime d "chat";;
assert (not(hashtbl_contient d "chat"));;
```

Pour $a, b \in \mathbb{N}$, on note $[\![a, b]\!] = \{a, a+1, \ldots, b\}$.
On rappelle que l'on note K l'ensemble des clés et V l'ensemble des valeurs. Soit $m \in \mathbb{N}$, et supposons que l'on dispose d'une fonction $h : K \to [\![0, m-1]\!]$, appelée **fonction de hachage**.

Une table de hachage est un **tableau** T de m cases, et, pour une clé $k \in K$, on stocke k et sa valeur associée dans la case $T[h(k)]$.
Par exemple, si K est l'ensemble des prénoms, et que l'on veut représenter une partie de jeu où 3 joueurs ont chacun un score:

```
score = {"Antoine": 8, "Vincent": 11, "Guillaume": 10}
```

Prenons $m = 26$, et h la fonction qui à un prénom associe le rang dans l'alphabet de sa première lettre (en commençant à 0 pour A). On a:

$$\begin{array}{lcl} h(\text{Antoine}) & = & 0 \\ h(\text{Vincent}) & = & 21 \\ h(\text{Guillaume}) & = & 6 \end{array}$$

Alors, la table de hachage représentant le dictionnaire des scores de la partie serait:

i	0	1	...	6	...	21	...	25
$T[i]$	(Antoine, 8)	(vide)	...	(Guillaume, 10)	...	(Vincent, 11)	...	(vide)

Pour modifier le score de Vincent, on change le contenu de la case numéro $h(\text{Vincent}) = 21$.
Un problème survient si un nouveau joueur appelé Alain rejoint la partie: on a $h(\text{Alain}) = 0 = h(\text{Antoine})$, la table de hachage doit donc stocker les informations des deux joueurs dans la case 0. On dit qu'il y a **collision** de la fonction de hachage.
Une manière de gérer le collisions, dite par **chainage**, consiste à stocker dans chaque case du tableau non pas un unique couple clé-valeur, mais la **liste** de tous les couples clé-valeur qui doivent y être stockés.
En OCaml, on utilisera donc le type suivant:

```
type ('a, 'b) hashtbl = ('a, 'b) list array
```

Une table de hachage est donc un **tableau de listes de couples**. En reprenant l'exemple précédent, où Alain a rejoint la partie, voici la table de hachage pour $m = 26$ et la fonction de hachage décrite plus haut:

```
let table_scores = [|
  [("Antoine", 8); ("Alain", 0)]; (* case 0 *)
  []; (* case 1 *)
  ...
  [("Guillaume", 10)]; (* case 6 *)
  ...
  [("Vincent", 11)]; (* case 21 *)
  ...
  [] (* case 25 *)
|]
```

Q18. On pose $m = 4$, et $h : K \to [\![0, 3]\!]$ qui à un prénom $k \in K$ associe son nombre de lettres, modulo 4. Donner la nouvelle table de hachage associée pour la partie à 4 joueurs ci-dessus.

On suppose dans la suite avoir fixé $m \in \mathbb{N}$, de telle sorte que toutes les tables de hachage auront m cases distinctes, chacune pouvant contenir une liste. En OCaml, on disposera d'une variable globale `m`.
Dans la suite, nous allons réimplémenter ce système de tables de hachage, qui existe déjà en OCaml via le module `Hashtbl`. La seule partie de ce module que l'on ne réimplémentera pas est la fonction de hachage `Hashtabl.hash: 'a -> int`, capable de produire un entier quelconque entre 0 et $2^{62} - 1$ à partir de n'importe quelle valeur, peu importe son type. Cette fonction dispose de plusieurs propriétés utiles qui en font une **bonne** fonction de hachage. Notamment,

il est difficile de trouver des collisions, et tout aussi difficile de trouver des préimages. On ne s'intéressera pas à la construction d'une telle fonction dans ce sujet, mais sachez qu'il existe de **très** nombreuses fonctions de hachage, et qu'outre l'implémentation de dictionnaires, elles peuvent notamment servir en cryptographie.

Dans la suite, on suppose avoir renommé la fonction de hachage native d'OCaml `h` afin de simplifier le code:

```
1 let h = Hashtbl.hash
```

Afin de ramener le résultat de cette fonction dans l'intervalle $[\![0, m-1]\!]$, on prend simplement ce résultat modulo m. Ainsi, si l'on dispose d'une table de hachage `t`, alors chaque case `t.(i)` est la **liste** de tous les couples d'associations (k, v) pour lesquels `(h k) mod m = i`.
Par exemple, la fonction `hashtbl_contient` s'écrira ainsi:

```
1 let hashtbl_contient(t: ('a, 'b) hashtbl) (k: 'a) : bool =
2   let m = Array.length t in
3   let i = (h k) mod m in
4   let l = t.(i) in
5   (* vérifie si t.(i) contient au moins un couple (k', v') avec k' = k,
6      c'est à dire vérifie si t.(i) contient la clé k. *)
7   List.exists (fun (k', v') -> k = k') l
```

Q19. En vous aidant de la fonction `List.assoc`, implémenter la fonction `hashtbl_recherche`.

Q20. Écrire le reste des fonctions de l'interface:

```
1 (* Renvoie un dictionnaire vide *)
2 hashtbl_vide : unit -> ('a, 'b) hashtbl
3
4 (* hashtbl_ajoute d k v modifie d pour y ajouter l'association k -> v *)
5 hashtbl_ajoute : ('a, 'b) hashtbl -> 'a -> 'b -> unit
6
7 (* hashtbl_supprime d k supprime la clé k de d *)
8 hashtbl_supprime: ('a, 'b) hashtbl -> 'a -> unit
```

Attention à ne pas créer de doublons de clés à l'ajout !

Q21. Écrire une fonction `hashtbl_keys: ('a, 'b) hashtbl -> 'a list` renvoyant la liste des clés d'une table de hachage.

Q22. Comparer le coût de la fonction `hashtbl_keys` avec celui de la même opération pour les ABR.

Intéressons-nous à la complexité des opérations des dictionnaires dans cette implémentation. On considère une table de hachage T, stockant n clés distinctes.

Q23. Quelle serait la **pire** fonction de hachage possible pour la complexité de `hashtbl_contient` ? Que serait alors la complexité atteinte, en fonction de n ?

On suppose maintenant que la fonction de hachage répartir les clés de manière totalement uniforme, de sorte que si k est une variable aléatoire uniforme sur K, on a $\mathbb{P}(h(k) = i) = \frac{1}{m}$ pour tout $i \in [\![0, m-1]\!]$.

Q24. En justifiant brièvement, quelle est la longueur moyenne attendue des listes de la table, en fonction de n et m ?

En pratique, les tables de hachage ne fixent pas m à l'avance, elles implémentent un mécanisme de **redimensionnement** qui permet de garder le ratio $\alpha = \frac{n}{m}$ entre deux valeurs α_0 et α_1. C'est le cas des dictionnaires Python par exemple, qui maintiennent toujours α entre 0.1 et 0.5.

Q25. Quel sont les avantages et inconvénients de prendre α petit / grand du point de vue des performances ?

`-- FIN DU SUJET --`

Annexe

Module Array

```
(* Array.length t renvoie la longueur de t *)
Array.length : 'a array -> int

(* Array.make n x renvoie un tableau de taille n
   dont chaque case vaut x *)
Array.make : int -> 'a -> 'a array
```

Module List

```
(* List.iter f l exécute f sur chaque élément de l
   dans l'ordre. *)
List.iter : ('a -> unit) -> 'a list -> unit

(* List.mem x l renvoie true si x est dans l, false sinon *)
List.mem : 'a -> 'a list -> bool

(* List.exists f l renvoie true s'il existe x dans l tel que
   f x = true, false sinon. *)
List.exists : ('a -> bool) -> 'a list -> bool

(* List.assoc x l renvoie y tel que (x, y) est dans l.
   Lève une exception Not_found si aucun couple de l
   n'a x comme première composante. *)
List.assoc : 'a -> ('a * 'b) list -> 'b

(* List.map f l applique f sur chaque élément de l et renvoie la
   liste obtenue. *)
List.map : ('a -> 'b) -> 'a list -> 'b list

(* List.fold_left f a [x1; x2; ...; xn] renvoie
   f( f ... f (f (f a x1) x2) x3....) xn *)
List.fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```