DS MP2I : tas ternaire ; Arbres de Patricia

Solution. □

1 Tas-min ternaires en C

Dans cette partie, les tas sont des tas-min ternaires (un nœud a au plus trois fils). Ils sont implémentés à l'aide d'un tableau contenu dans une structure. Le premier élément du tas est stocké à l'indice 0 du tableau, contrairement à la convention adoptée en cours.

La structure contient également la taille du tas (nombre d'éléments effectivement présents) ainsi que la capacité du tableau (nombre maximal d'éléments pouvant être stockés sans redimensionnement). Si le tableau devient trop petit, il est automatiquement redimensionné.

Remarque. Contrairement au cours, pour un tas `t`, `t->size` est ici la première position libre et non la dernière position occupée.

Les types manipulés sont les suivants :

```c
typedef struct {
    KEY key;//type connu à la compilation ex  : gcc -DKEY=int
    VALUE value;//type connu à la compilation ex  : gcc -DValue=float
} heap_entry_t;

typedef struct {
    size_t  size ;          /* nombre d'éléments actuellement stockés */
    size_t capacity;        /* capacité  totale  du tableau */
    heap_entry_t *data;  /* tableau des éléments */
} ternary_min_heap_t;
```

Les inclusions `#include <stdio.h>`, `#include <assert.h>`, `#include <stdbool.h>` et `#include <stdlib.h>` ont été réalisées.

À toute fin utile, on suppose que la fonction suivante qui libère un tas créé par `create` est disponible :

```c
void tmh_free(ternary_min_heap_t *heap);
```

On ne demande pas de l'écrire.

Question 1.

Écrire la fonction

```c
/* Crée un tas vide de capacité  initiale  capacity.
   Si  capacity vaut 0,  le  tableau interne  est  vide  ;  il  pourra
   être  créé  plus  tard  lors  d'un redimensionnement.
*/
ternary_min_heap_t *tmh_create(size_t capacity);
```

En principe, dans ce sujet, on ne fait pas de programmation défensive : on suppose en particulier que les allocations mémoire réussissent et que les arguments donnés aux fonctions ont les bonnes caractéristiques.

Toutefois, afin de se familiariser avec des pratiques plus proches de celles utilisées en programmation professionnelle, on fera ici une exception (et pour cette fonction uniquement).

Ainsi, si un appel à `malloc` échoue (donc, lorsqu'elle renvoie `NULL`), on affichera un message d'erreur sur la sortie d'erreur standard puis on interrompra le programme avec une commande de sortie.

Solution. Code

```c
ternary_min_heap_t *tmh_create(size_t capacity) {
    ternary_min_heap_t *heap = malloc(sizeof(ternary_min_heap_t));
    if (heap == NULL) {
        fprintf(stderr, "Erreur  : allocation  du tas  impossible.\n");
        exit(EXIT_FAILURE);
    }

    heap->size = 0;
    heap->capacity = capacity;

    if (capacity == 0) {
```

```
        heap->data = NULL;
    } else {
        heap->data = malloc(capacity * sizeof(heap_entry_t));
        if (heap->data == NULL) {
            fprintf(stderr, "Erreur : allocation du tableau impossible.\n");
            free(heap);
            exit(EXIT_FAILURE);
        }
    }

    return heap;
}


```

□

Question 2. Tas vide, racine, père

1. Écrire la fonction

```
bool tmh_is_empty(const ternary_min_heap_t *heap);
```

qui renvoie `true` si le tas est vide et `false` sinon.

Solution. Code

```
bool tmh_is_empty(const ternary_min_heap_t *heap) {
    return heap->size == 0;
}

```

□

2. Écrire la fonction

```
heap_entry_t tmh_top(const ternary_min_heap_t *heap);
```

qui renvoie l'entrée de clé minimale d'un tas min.

Précondition. Le tas n'est pas vide.

On vérifie cette précondition à l'aide d'une assertion.

Solution. Code

```
heap_entry_t tmh_top(const ternary_min_heap_t *heap) {
    assert(heap != NULL);
    assert(heap->size > 0);
    return heap->data[0];
}

```

□

3. Écrire une fonction auxiliaire

```
static size_t father(size_t i);
```

qui renvoie l'indice du père du nœud d'indice `i` dans un tas ternaire.

Solution. Code

```
static inline size_t father(size_t i) {
    assert(i > 0);
    return (i - 1) / 3;
}

```

□

Précondition. L'indice `i` est non nul.

On vérifie cette précondition à l'aide d'une assertion.

Une implémentation complète demanderait l'écriture d'une percolation haute puis de l'insertion et, éventuellement, d'un redimensionnement du tableau de stockage. On ne demande toutefois pas de les écrire. On se concentre sur la suppression de la racine.

Question 3.

Écrire une fonction auxiliaire

```
static size_t smallest_child(const ternary_min_heap_t *heap, size_t i);
```

qui renvoie l'indice du fils de plus petite clé du nœud d'indice `i`.

Précondition. Le nœud d'indice `i` possède au moins un fils.

On vérifie cette précondition à l'aide d'une assertion.

Solution. Code

```
static size_t smallest_child(const ternary_min_heap_t *heap, size_t i) {
    size_t first = 3 * i + 1;
    size_t min = first;

    assert(first < heap->size);

    if (first + 1 < heap->size &&
        heap->data[first + 1].key < heap->data[min].key) {
        min = first + 1;
    }

    if (first + 2 < heap->size &&
        heap->data[first + 2].key < heap->data[min].key) {
        min = first + 2;
    }

    return min;
}

```

□

Question 4.

Écrire une fonction auxiliaire

```
static void percolate_down(ternary_min_heap_t *heap, size_t i);
```

qui, dans un tas presque bien ordonné, rétablit la propriété de tas min en faisant descendre l'élément d'indice `i` jusqu'à sa position correcte.

```
static void percolate_down(ternary_min_heap_t *heap, size_t i) {
    while (3 * i + 1 < heap->size) {
        size_t child = smallest_child(heap, i);

        if (heap->data[i].key <= heap->data[child].key) {
            break;
        }

        heap_entry_t tmp = heap->data[i];
        heap->data[i] = heap->data[child];
        heap->data[child] = tmp;

        i = child;
    }
}

```

Question 5.

Écrire la fonction

```
heap_entry_t tmh_pop(ternary_min_heap_t *heap);
```

qui supprime et renvoie l'entrée de clé minimale du tas.

Précondition. Le tas n'est pas vide.

On vérifie cette précondition à l'aide d'une assertion.

Solution. Code

```c
heap_entry_t tmh_pop(ternary_min_heap_t *heap) {
    heap_entry_t min_entry;

    assert(heap->size > 0);

    min_entry = heap->data[0];
    heap->size--;

    if (heap->size > 0) {
        heap->data[0] = heap->data[heap->size];
        percolate_down(heap, 0);
    }

    return min_entry;
}

```

□

Question 6.

Écrire la fonction

```c
ternary_min_heap_t *tmh_build(const heap_entry_t *array, size_t n);
```

qui construit un tas-min ternaire contenant les `n` entrées du tableau `array` en utilisant la méthode par descente.

Solution. Code

```c
ternary_min_heap_t *tmh_build(const heap_entry_t *array, size_t n) {
    ternary_min_heap_t *heap = tmh_create(n);

    for (size_t i = 0; i < n; i++) {
        heap->data[i] = array[i];
    }
    heap->size = n;

    if (n > 1) {
        size_t i = father(n - 1);
        while (1) {
            percolate_down(heap, i);
            if (i == 0) {
                break;
            }
            i--;
        }
    }

    return heap;
}


```

□

2 Arbres préfixes (ou tries) ; arbres de Patricia

— Toutes les fonctions du module `List` sont utilisables.
— Toutes les fonctions auxiliaires doivent être accompagnées d'une explication ;
— Des rappels sur les enregistrements en langage OCaml et la complexité de certaines opération sur les `string` sont disponible dans l'appendice en section 3.

2.1 Problème de stockage de chaînes

Dans toute la suite, on note $|m|$ la longueur d'un mot m et ε le mot vide.
On souhaite représenter un ensemble fini de mots sur un alphabet donné (par exemple l'alphabet des caractères ASCII).
On désire pouvoir effectuer efficacement les opérations suivantes :
— tester si un mot appartient à l'ensemble ;
— insérer un nouveau mot ;
— éventuellement supprimer un mot.
Une première idée consiste à stocker les mots dans une liste ou un tableau.

Question 7.

On stocke n mots dans une liste. Quelle est la complexité, dans le pire cas, de la recherche d'un mot de longueur ℓ ?

Solution. Dans une liste, la recherche d'un mot consiste à comparer successivement ce mot avec chacun des n mots de la liste.
Dans le pire cas (mot absent ou présent en dernière position), on effectue n comparaisons.
Comparer deux mots de longueur ℓ coûte $O(\ell)$ dans le pire cas (il faut comparer les caractères un à un).
Ainsi, la complexité totale est en $O(n\ell)$. □

Question 8.

On suppose maintenant que les mots sont stockés dans un tableau trié. Quelle complexité peut-on attendre de la recherche d'un mot de longueur ℓ parmi n mots ?

Solution. La recherche dichotomique effectue $O(\log n)$ comparaisons.
Chaque comparaison entre deux mots coûte $O(\ell)$ dans le pire cas.
Ainsi, la complexité totale est en $O(\ell \log n)$. □

Ces approches reposent sur des comparaisons entre mots, ce qui peut être coûteux lorsque les mots sont longs.

Nous voulons une structure de données hiérarchique de données permettant :
— d'éviter les comparaisons complètes entre mots ;
— d'exploiter le fait que deux mots peuvent partager un préfixe commun ;
— d'obtenir des opérations dont le coût dépend principalement de la longueur du mot manipulé.

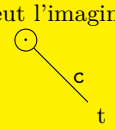
On s'intéresse dans un premier temps à une structure appelée *arbre préfixe* (ou `trie`), qui permet de représenter efficacement un ensemble de mots en factorisant leurs préfixes communs.

2.2 Arbres préfixes

Pour éviter de comparer plusieurs fois des mots entiers, on présente une structure de données qui exploite directement leur décomposition en caractères.
L'idée des *arbres préfixes*, ou *tries*, est la suivante :
— la racine représente le préfixe vide ;
— chaque arête est étiquetée par un caractère ;
— un chemin issu de la racine représente un préfixe ;
— les mots ayant un préfixe commun partagent donc une partie de leur chemin.
Ainsi, au lieu de stocker séparément tous les mots, on factorise leurs préfixes communs. Cette représentation est particulièrement adaptée lorsque de nombreux mots commencent de la même façon.
Par exemple, les mots `car`, `carton` et `chat` auront un début commun correspondant à la lettre `c`, puis des branches distinctes apparaîtront dès que leurs écritures diffèrent.
Pour savoir si un mot appartient à l'ensemble représenté, il suffit alors de parcourir l'arbre en suivant successivement les caractères du mot.
On considère les types suivants :

```
type trie = {
  terminal : bool;
  enfants : branche list
}
and branche = char * trie
```

Une « branche » est un tuple `(c,t)` constitué d'une *valuation* (ou *étiquette*) sous forme de `char` et d'un arbre `trie`. On peut l'imaginer comme une arête valuée par c et pointant vers t.



Dans les implémentations des opérations sur les tries (recherche, insertion), on supposera et on maintiendra l'invariant suivant :

Dans la liste `enfants` d'un nœud, les caractères apparaissant dans les couples sont deux à deux distincts.

Autrement dit, **un nœud ne possède pas deux fils accessibles par la même lettre**.

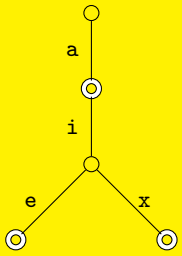
2.2.1 Premiers exemples

Dans un `trie`, chaque nœud représente un préfixe des mots stockés. Le champ `terminal` indique si ce préfixe est lui-même un mot de l'ensemble.

On considère l'ensemble de mots (on dit aussi *langage*) suivant :

$$\mathcal{D} = \{\texttt{a}, \texttt{aie}, \texttt{aix}\}.$$

La figure 1 donne une représentation graphique de l'arbre `trie` correspondant ainsi que son code OCaml. Les nœuds *acceptants* sont représentés par des cercles doubles les autres par de simples cercles.



(a) Trie : le mot vide et `ai` ne sont pas dans le langage.

```
let exemple_aie_aix =
  {
    terminal = false;
    enfants = [
      ('a',
       {
         terminal = true;
         enfants = [
           ('i',
            {
              terminal = false;
              enfants = [
                ('e', {terminal = true; enfants = []});
                ('x', {terminal = true; enfants = []})
              ]
            })
         ]
       })
    ]
  }
```

(b) Code OCaml

FIGURE 1 – Trie et représentation OCaml

Question 9. Une variable globale

Donner un code OCaml de déclaration d'un arbre `trie` nommé `vide` et représentant l'ensemble vide.

Solution. Code

```
let vide = {terminal = false; enfants = []};;
```

□

Dans toute la suite on peut utiliser la variable globale `vide` dans les contextes où elle est pratique.

Question 10. Représentation graphique

Représenter graphiquement l'arbre `trie` correspondant à l'ensemble $\mathcal{D} = \{\varepsilon, \texttt{aa}\}$.

(a) `Trie` associé à $\{\texttt{""}, \texttt{"aa"}\}$

```
let exemple_vide_aa =
  {
    terminal = true;
    enfants = [
      ('a',
       {
         terminal = false;
         enfants = [
           ('a', {terminal = true; enfants = []})
         ]
       })
    ]
  }
```

(b) Valeur OCAML correspondante

FIGURE 2 – Un `trie` représentant le mot vide et le mot `aa`.

Solution. □

2.2.2 Implémentation des opérations

On souhaite maintenant implémenter en OCaml quelques opérations sur les tries.

Question 11.

Écrire une fonction `trouver (c:char)(enfants:branche list): trie option` qui prend en paramètre la liste de tuples (caractère,arbre) `enfants` et renvoie une option contenant le sous-arbre associé au caractère `c` s'il existe, et `None` sinon.

Avec le code de la figure 1, on obtient :

```
# trouver 'a' exemple_aie_aix.enfants;;
- : trie option =
Some
 {terminal = true;
  enfants =
   [('i',
     {terminal = false;
      enfants =
       [('e', {terminal = true; enfants = []});
        ('x', {terminal = true; enfants = []})]})]}
```

Solution. Code

```

let rec trouver (c:char) (enfants:branche list) : trie option =
  match enfants with
  | [] -> None
  | (x, t) :: reste ->
     if x = c then Some t
     else trouver c reste;;
```

□

Question 12.

Écrire une fonction `recherche (t:trie)(s:string): bool` qui teste si le mot `s` appartient au dictionnaire représenté par le trie `t`.

Solution. Code

```
let recherche t s =
  let n = String.length s in
  let rec aux noeud i =
    if i = n then noeud.terminal
    else
      match trouver s.[i] noeud.enfants with
      | None -> false
      | Some fils -> aux fils (i + 1)
  in
  aux t 0
```

□

Question 13.

On note ℓ la longueur du mot recherché et k l'arité maximale pour un nœud. Déterminer la complexité, dans le pire cas, de la fonction `recherche`.

Solution. La fonction parcourt successivement les ℓ caractères du mot.

À chaque étape, la fonction `trouver` recherche le caractère voulu dans la liste des fils du nœud courant. Cette recherche coûte $O(k)$ dans le pire cas.

La complexité totale est donc en $O(\ell k)$.

Si l'alphabet est de taille bornée, alors k est borné, et cette complexité se simplifie en $O(\ell)$. □

Question 14. insertion

Écrire la fonction `insert (t : trie)(m:string): trie` qui insère le mot `m` dans le trie `t`.

On parcourt l'arbre en suivant les caractères de m dans l'ordre. Tant que le préfixe courant de m correspond à un chemin dans le `trie`, on descend dans l'arbre.

Dès que le caractère suivant de m ne correspond plus à un fils du nœud courant, on crée un sous-arbre en forme de chaîne correspondant à la fin du mot m, que l'on rattache à ce nœud.

Il faut bien veiller à ce qu'aucun nœud ne possède deux fils accessibles par la même lettre à l'issue de l'insertion.

```
# insert exemple_aie_aix "aiea";;
- : trie =
{terminal = false;
 enfants =
  [('a',
    {terminal = true;
     enfants =
      [('i',
        {terminal = false;
         enfants =
          [('e',
            {terminal = true;
             enfants = [('a', {terminal = true; enfants = []})]});
           ('x', {terminal = true; enfants = []})]})]})]}
```

Solution. Voici un code possible.

```
let insert t m =
  let n = String.length m in
  let rec aux noeud i =
    if i = n then
      { noeud with terminal = true }
    else
      let c = m.[i] in
      match trouver c noeud.enfants with
      | None ->
          let nouveau = aux vide (i + 1) in
          {
            noeud with
            enfants = (c, nouveau) :: noeud.enfants
          }
      | Some fils ->
          let fils' = aux fils (i + 1) in
          {
            noeud with
            enfants =
              (c, fils') ::
              List.remove_assoc c noeud.enfants
          }
  in
  aux t 0
;;
```

On peut aussi utiliser `List.filter (fun (y,_)-> y<>c)` à la place de `List.remove_assoc` ou écrire une fonction auxiliaire de suppression d'une association. □

Question 15.

Écrire la fonction `build (l: string list)` qui construit un arbre `trie` représentant l'ensemble de mots donné par la liste `l`.

```
 # build [""; "aie"; "aix"];;
- : trie =
{terminal = true;
 enfants =
  [('a',
    {terminal = false;
     enfants =
      [('i',
        {terminal = false;
         enfants =
          [('x', {terminal = true; enfants = []});
           ('e', {terminal = true; enfants = []})]})]})]}
```

Solution. Code

```
  let build = List.fold_left insert vide;;
```

□

Question 16. Type plus efficace

Notre représentation des `trie`s, dans laquelle les `enfants` d'un nœud sont stockés sous la forme d'une liste de couples `(char * trie)`, est simple et purement fonctionnelle. Elle présente toutefois les inconvénients habituels des listes : la recherche d'un fils et sa mise à jour sont linéaires en le nombre de fils du nœud considéré.

Proposer une nouvelle définition OCaml d'un type `trie2` reposant sur le même principe d'enregistrement à deux champs, mais dans laquelle les fils d'un nœud sont stockés dans une structure plus efficace pour la recherche, l'insertion et la suppression.

Solution. On peut utiliser un dictionnaire implanté par table de hachage :

```
type trie2 = {
  terminal2 : bool;
  enfants2 : (char, trie2) Hashtbl.t
};;
```

La recherche, l'insertion et la suppression d'un fils se font alors en temps amorti constant. En revanche, cette représentation n'est plus purement fonctionnelle : les tables de hachage sont des structures modifiables.

On peut aussi utiliser un dictionnaire purement fonctionnel implanté par arbre équilibré :

```
module CharMap = Map.Make(Char);;

type trie3 = {
  terminal3 : bool;
  enfants3 : trie3 CharMap.t
};;
```

Dans ce cas, la recherche, l'insertion et la suppression d'un fils se font en temps logarithmique en fonction du nombre de fils du nœud considéré. La structure reste alors purement fonctionnelle. □

2.3 Arbres de Patricia

Les arbres préfixes permettent d'effectuer des recherches de mots avec un coût acceptable.

Cependant, cette structure présente certains inconvénients.

Question 17.

Soit n, un entier positif. Donner un exemple de langage de cardinal 1 dont l'arbre `trie` correspondant est de hauteur n.

Solution. Considérons par exemple l'ensemble de mots

$$\{\mathtt{a}^n\}.$$

Le `trie` correspondant est essentiellement une chaîne de nœuds, chacun ayant un seul fils. Il est de hauteur n.

Cela entraîne un gaspillage de mémoire, car de nombreux nœuds ne servent qu'à prolonger un chemin sans créer de branchement. Il faut mémoriser, non seulement toutes les lettres mais le poids de la structure elle-même

□

Les arbres de Patricia (pour *Practical Algorithm To Retrieve Information Coded In Alphanumeric*) ont été introduits par Donald R. Morrison en 1968.

Il s'agit d'une variante des arbres `trie` dans laquelle les chemins constitués de nœuds non terminaux ayant un seul fils sont compressés. Cette compression permet de réduire le nombre de nœuds et la hauteur de l'arbre.

Ainsi, dans un arbre de Patricia on maintient l'invariant suivant au fil des ajouts/suppressions :

— chaque arête est étiquetée par une chaîne de caractères non vide ;

— tout nœud interne, sauf éventuellement la racine, possède au moins deux fils (sinon les arêtes correspondantes seraient fusionnées) ;

— les étiquettes des arêtes issues d'un même nœud commencent par des caractères distincts.

On donne le type :

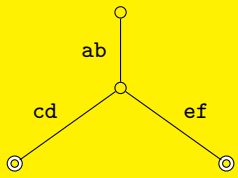
```

type patricia = {
  accepting : bool;
  children : branch list
}
and branch = string * patricia;;
```

Une *branche* est un tuple (mot, arbre de Patricia). Les enfants d'un nœud de Patricia forment donc une liste de tels tuples. Les premiers membres de ces tuples sont les *valuations* (ou *étiquettes*) des arêtes sortantes.

Considérons par exemple le langage :

$$\mathcal{D} = \{\mathtt{abcd}, \mathtt{abef}\}$$

La figure 3 en propose une représentation.



(a) Arbre de Patricia

```
let exemple_pat_abcd_abef =
{
  accepting = false;
  children = [
    ("ab",
     {
       accepting = false;
       children = [
         ("cd", {accepting = true; children = []});
         ("ef", {accepting = true; children = []})
       ]
     })
    ]};;
```

(b) Code OCaml

FIGURE 3 – Arbre de Patricia associé à $\{\texttt{abcd}, \texttt{abef}\}$.

Question 18.

Quelle serait la hauteur d'un arbre `trie` représentant $\mathcal{D} = \{\texttt{abcd}, \texttt{abef}\}$?

Solution. Dans un `trie` dont les arêtes sont étiquetées par des caractères, chaque arête correspond à la lecture d'un caractère. La hauteur de l'arbre est donc égale à la longueur du plus long mot représenté.
Ici, les mots `abcd` et `abef` sont de longueur 4, donc la hauteur d'un tel `trie` serait égale à 4. □

2.3.1 Test d'appartenance

Question 19.

Écrire une fonction `prefixe (p:string)(s:string)(i:int): bool` qui teste si `p` est un préfixe de `s` à partir de la position `i`.
FFF on aurait pu dire aussi « qui teste si `p` est un facteur de `s` démarrant à la position `i` »

```
# prefixe "def" "abcde" 3, prefixe "def" "abcdef" 3, prefixe "def" "abcdefg" 3;;
- : bool * bool * bool = (false, true, true)
```

Solution. Code

```
let prefixe p s i =
  let lp = String.length p in
  try
    for k = 0 to lp-1 do
      if p.[k] <> s.[i+k] then invalid_arg "different chars";
    done;
    true
  with Invalid_argument _ -> false;;
```

□

Question 20. Recherche

On écrit deux fonctions de recherche :

1. Écrire `trouver_pat (s:string)(children:branch list)(i:int): (string * patricia)option` qui renvoie `None` ou bien `Some (x,t)` tel que x est l'unique préfixe de s à partir de la position i présent dans la branche `enfants`.
2. Écrire `recherche_pat (pat:patricia)(s:string)` qui cherche si un mot `s` est présent dans l'ensemble représenté par l'arbre de Patricia `pat`.

```
# let exemple_abcd_abef_x =
  {
    accepting = false;
    children = [
      ("ab",
       {
         accepting = false;
         children = [
           ("cd", {accepting = true; children = []});
           ("ef", {accepting = true; children = []})
         ]
       });
      ("x",
       {
         accepting = true;
         children = []
       })
     ]}
in
recherche_pat exemple_abcd_abef_x "abcd",
recherche_pat exemple_abcd_abef_x "abcx",
recherche_pat exemple_abcd_abef_x "abcde",
recherche_pat exemple_abcd_abef_x "x";;
- : bool * bool * bool * bool = (true, false, false, true)
```

Solution. Code

```
let rec trouver_pat (s:string) (enfants:branch list) (i:int) :
          (string * patricia) option =
  match enfants with
  | [] -> None
  | (x, t) :: reste ->
     if prefixe x s i then Some (x,t)
     else trouver_pat s reste i;;

let recherche_pat (pat:patricia) (s:string) =
  let n = String.length s in
  let rec aux noeud i =
    if i = n then noeud.accepting
    else
      match trouver_pat s noeud.children i with
      | None -> false
      | Some (x,fils) -> aux fils (i + String.length x)
  in
  aux pat 0;;
```

□

2.3.2 Insertion dans un arbre de Patricia

Question 21. Préfixe commun

Écrire une fonction `split_common (s1:string)(s2:string): string * string * string` qui renvoie un triplet `(p, r1, r2)` tel que :

- `p` est le plus long préfixe commun à `s1` et `s2` ;
- `s1 = p ^ r1` ;
- `s2 = p ^ r2`.

On veillera à ce que la complexité de cette fonction soit linéaire en la longueur du plus long préfixe commun de `s1` et `s2`.

```
# split_common "abcd" "abcd";;
- : string * string * string = ("abcd", "", "")
# split_common "abcd" "abce";;
- : string * string * string = ("abc", "d", "e")
# split_common "abcd" "abcde";;
- : string * string * string = ("abcd", "", "e")
# split_common "abcd" "abc";;
- : string * string * string = ("abc", "d", "")
```

Solution. Voici un code possible.

```
let split_common s1 s2 =
  let n1 = String.length s1 in
  let n2 = String.length s2 in
  let rec aux i =
    if i < n1 && i < n2 && s1.[i] = s2.[i]
    then aux (i + 1)
    else i
  in
  let k = aux 0 in
  let p  = String.sub s1 0 k in
  let r1 = String.sub s1 k (n1 - k) in
  let r2 = String.sub s2 k (n2 - k) in
  (p, r1, r2)
;;
```

□

Question 22. Recherche de branche

Écrire une fonction `chercher_branche (s:string)(enfants:branch list): ((string * string * string)* (string * patricia))option` qui, étant donné un mot `s` et une liste de branches (donc de tuples (mot,arbre)), recherche une branche dont l'étiquette possède un préfixe commun non vide avec `s`.Si une telle branche `(x,t)` existe, alors elle est unique (par invariant des arbres de patricia) et la fonction renvoie `Some((p,rx,rs),(x,t))` où `p ^ rx = x && p ^ rs = s && p <> ""` est vrai. Si aucune branche ne possède de préfixe commun non vide avec `s`, la fonction renvoie `None`.

Solution. Voici un code possible.

```
let rec chercher_branche s enfants =
  match enfants with
  | [] -> None
  | (x,t)::q ->
      let (p, rx, rm) = split_common x s in
      if p = "" then chercher_branche s q
      else Some ((p, rx, rm), (x, t))
;;
```

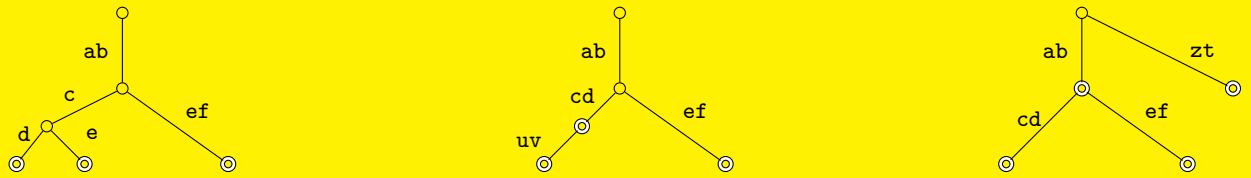
□

Pour un arbre de Patricia, l'insertion ne consiste plus simplement à « descendre lettre par lettre ». À chaque étape, on considère le mot restant `m` à insérer et une branche `(x,t)` issue du nœud courant.

On compare alors `x` et le mot restant. Il se présente quatre cas :

1. `x` est un préfixe strict de `m` (c.a.d. qu'existe `s` tel que `x^s = m`) : on descend dans le sous-arbre `t` en poursuivant l'insertion avec le suffixe restant `s` ;
2. `x` est exactement égal à `m` : on rend le nœud `t` acceptant (voir : ajout de **ab** figure 4) ;
3. le mot restant est un préfixe strict de `x` (c.a.d. qu'existe `s` tel que `m^s=x`) : on coupe l'arête en deux en créant un nouveau nœud acceptant (voir : ajout de **abce** figure 4) ;
4. `x` et le mot restant possèdent un préfixe commun `p` non vide sans que l'un soit préfixe de l'autre (c.a.d. qu'existent `s1,s2` tels que `p^s1 = x` et `p^s2 = m`) : on crée un nouveau nœud d'embranchement correspondant à ce préfixe commun (voir : ajout de **abcduv** figure 4) ;.

Si `x` et le mot restant n'ont pas de préfixe commun non vide, on passe à la branche suivante. Si aucune branche ne convient, on ajoute une nouvelle branche correspondant au mot restant (voir : ajout de **zt** figure 4).



(a) Après insertion de **abce** **(b)** Après insertion de **abcduv** **(c)** Après insertion de **ab** puis **zt**

FIGURE 4 – Arbre de Patricia de la figure 3 après ajouts de mots

Question 23.

Écrire la fonction `insert_pat (pat:patricia)(s:string): patricia` qui insère le mot `s` dans l'arbre `pat`. Pour chaque nœud, on cherche si une des branches (x, t) vérifie que x partage un préfixe non vide avec le mot courant.

Ces différents cas peuvent être déterminés à l'aide de la fonction `split_common`.

```
# let t = {accepting = false;
 children =
  [("ab",
    {accepting = false;
     children =
      [("de", {accepting = true; children = []});
       ("cd", {accepting = true; children = []})]});
   ("x", {accepting = true; children = []})]}
```

```
in insert_pat t "abdz";;
- : patricia =
{accepting = false;
 children =
  [("ab",
    {accepting = false;
     children =
      [("d",
        {accepting = false;
         children =
          [("z", {accepting = true; children = []});
           ("e", {accepting = true; children = []})]});
       ("cd", {accepting = true; children = []})]});
   ("x", {accepting = true; children = []})]}
```

Solution. Code

```
let rec insert_pat pat s =
  if s = "" then { pat with accepting = true }
  else
    let set_epsilon = { accepting = true; children = [] } in
    match chercher_branche s pat.children with
    | None ->
        { pat with children = (s, set_epsilon) :: pat.children }

    | Some ((p, rx, rm), (x, t)) ->
        let sans_x = List.filter (fun (y,_) -> y <> x) pat.children in
        match rx, rm with
        | "", "" -> (* x = s *)
            { pat with
              children = (x, { t with accepting = true }) :: sans_x }

        | _, "" -> (* x = s ^ rx *)
            let fils = { accepting = true; children = [(rx, t)] } in
            { pat with children = (p, fils) :: sans_x }

        | "", _ -> (* s = x ^ rm *)
            { pat with
              children = (p, insert_pat t rm) :: sans_x }

        | _, _ -> (* x = p ^ rx et s = p ^ rm *)
            let fils = { accepting = false; children = [(rx, t)] } in
            { pat with
              children = (p, insert_pat fils rm) :: sans_x }
;;
```

□

Question 24.

Écrire une fonction `build_pat : string list -> patricia` qui forme un arbre de Patricia à partir d'une liste de mots.

```
# build_pat ["abcd"; "abde"];;
- : patricia =
{accepting = false;
 children =
  [("ab",
    {accepting = false;
     children =
      [("de", {accepting = true; children = []});
       ("cd", {accepting = true; children = []})]})]}
```

Solution. Code

```
let build_pat = List.fold_left (insert_pat) vide_pat;;
```

□

2.3.3 Suppression

La suppression d'un mot présent dans un arbre de Patricia se fait en deux temps :
— Suppression logique (changement du caractère acceptant d'un nœud) ;
— normalisation de l'arbre.

Question 25. Arbre vide

Écrire une fonction `empty_pat (pat:patricia): bool` qui teste si l'arbre de Patricia `pat` ne reconnaît aucun mot.

Solution. Voici un code possible.

```
let empty_pat pat =
  (not pat.accepting) && pat.children = []
;;
```

□

Question 26. Normalisation d'une branche

Écrire une fonction `normalize_branch (b:branch): branch list` qui prend en argument une branche `(x,t)` et renvoie une liste vide ou singleton :

— si `t` est vide, la branche est supprimée ;

— si `t` n'est pas acceptant et possède un unique fils `(y,u)`, on fusionne les deux arêtes et on renvoie la branche `(x ^ y,u)` ;

— dans les autres cas, la branche est inchangée.

```
# normalize_branch ("ab",vide_pat);;
  - : (string * patricia) list = []
# normalize_branch ("ab",{accepting = false; children =
    [("cd", {accepting = true; children = []})]});;
- : (string * patricia) list = [("abcd", {accepting = true; children = []})]
# normalize_branch ("ab",{accepting = true; children =
    [("cd", {accepting = true; children = []})]});;
- : (string * patricia) list =
[("ab",
  {accepting = true; children = [("cd", {accepting = true; children = []})]})]
```

Solution. Voici un code possible.

```
let normalize_branch (x,t) =
  if not t.accepting then
    match t.children with
    | [] -> []
    | [(y,u)] -> [(x ^ y, u)]
    | _ -> [(x,t)]
  else
    [(x,t)]
;;
```

□

Question 27. Branche préfixe

Écrire une fonction `find_prefix_branch (s:string)(l:branch list): branch option` qui renvoie `Some(x,t)` s'il existe dans `l` une branche `(x,t)` telle que `x` soit un préfixe du mot non vide `s`, et `None` sinon. On peut utiliser `trouver_pat` et une assertion pour faire tenir le code en une ligne.

Solution. Voici un code possible.

```
let rec find_prefix_branch s l =
  assert (s <> "");
  match l with
  | [] -> None
  | (x,t)::q ->
      if prefixe x s 0 then Some (x,t)
      else find_prefix_branch s q
      ;;

let find_prefix_branch s l =  assert (s <> ""); trouver_pat s l 0;;
```

□

La suppression dans un arbre de Patricia consiste à suivre le chemin correspondant au mot à supprimer, puis à rétablir la propriété de compression de l'arbre.

On procède de la manière suivante.

— Si le mot à supprimer est vide, on rend le nœud courant non acceptant.

— Sinon, on recherche parmi les branches `(x,t)` du nœud courant une branche telle que `x` soit un préfixe du mot restant.
  — S'il n'en existe pas, le mot n'est pas présent dans l'arbre, lequel reste inchangé.
  — Sinon, on poursuit récursivement la suppression dans le sous-arbre `t` avec le suffixe restant du mot.
— Après la suppression récursive, il peut apparaître des nœuds inutiles. On simplifie alors l'arbre :
  — si un sous-arbre n'est plus acceptant et ne possède aucun fils, on supprime la branche correspondante ;
  — si un sous-arbre n'est pas acceptant et possède un unique fils, on fusionne les deux arêtes en concaténant leurs étiquettes.

On obtient ainsi un arbre de Patricia représentant l'ensemble initial privé du mot considéré.

Question 28. Suppression

Écrire la fonction `remove_pat (pat:patricia)(s:string): patricia` qui supprime le mot `s` de l'arbre de Patricia `pat` lorsqu'il est présent.

On veille, après suppression, à maintenir la propriété de compression des arbres de Patricia.

Solution. Voici un code possible.

```
let rec remove_pat pat s =
  if s = "" then { pat with accepting = false }
  else(
    match find_prefix_branch s pat.children with
    | None -> pat (*s pas trouvé*)
    | Some (x,t) -> (*exists p tq x ^ p = s*)
       let p = String.sub s (String.length x) (String.length s - String.length x) in
       let t' = remove_pat t p in
       let others = List.remove_assoc x pat.children in
       { pat with children = (normalize_branch (x,t')) @ others}
  );;
```

□

3 Appendice

3.1 Enregistrement et mot clé `with`

En OCaml, lorsqu'on manipule des enregistrements, il est fréquent de vouloir construire un nouvel enregistrement très proche d'un enregistrement existant, en ne modifiant qu'un petit nombre de champs.

Pour cela, on peut utiliser la construction `{ e with champ = valeur }`.

Si `e` est un enregistrement, alors cette expression désigne un nouvel enregistrement dont tous les champs sont identiques à ceux de `e`, à l'exception du champ `champ`, remplacé par `valeur`.

Par exemple, si `t` est de type `trie`, l'expression `{ t with terminal = true }` construit un nouveau trie ayant les mêmes fils que `t`, mais dont le champ `terminal` vaut `true`.

Cette écriture est particulièrement utile dans un cadre fonctionnel, car elle permet de modifier conceptuellement une structure sans détruire la valeur initiale.

3.2 String

On rappelle comment récupérer un caractère et la longueur d'un objet de type `string` :

```
# "toto".[0], String.length "toto";;(*2 opérations en O(1)*)
- : char * int = ('t', 4)
```

Si `s1` et Cs2 sont deux chaînes de caractères, `s1^s2` est leur concaténation et elle s'obtient en $O(|s_1|+|s_2|)$. La comparaison `s1=s2` s'effectue en $O(\min(|s_1|,|s_2|))$.

La fonction `String.sub s i n` permet d'extraire la sous-chaîne de longueur n de la chaîne s démarrant à l'indice i.

```
let s = "algorithmique";;

let t1 = String.sub s 0 5;;    (* "algor" *)
let t2 = String.sub s 5 4;;    (* "ithm" *)
let t3 = String.sub s 9 4;;    (* "ique" *)
```

Une exception `Invalid_argument` est soulevée si $i<0$, $n<0$ ou $n>|s|$. La complexité en temps et en mémoire est un $O(n)$.