TP9b - Tries ou arbres préfixes

Arbres

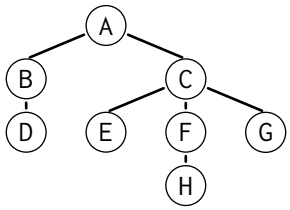
Les arbres binaires ont une structure très rigide, avec exactement deux sous-arbres à chaque nœud. Il arrive que l'on ait besoin de plus de souplesse, c'est-à-dire de structures arborescentes où chaque nœud possède un nombre variable de sous-arbres. Dans ce cas, on utilise des *arbres*.

Définition 1 – Arbres

Un *arbre* est un ensemble de $n \geqslant 1$ nœuds structurés de la manière suivante :

- un nœud particulier r est appelé la *racine* de l'arbre ;
- les $n-1$ nœuds restants sont partitionnés en $k \geqslant 0$ sous-ensembles disjoints qui forment autant d'arbres, appelés *sous-arbres* de r ;
- la racine r est liée à la racine de chacun des k sous-arbres.

Voici un exemple d'arbre contenant 8 nœuds, ici étiquetés avec les lettres `A`, … , `H`, et dont la racine est `A`.

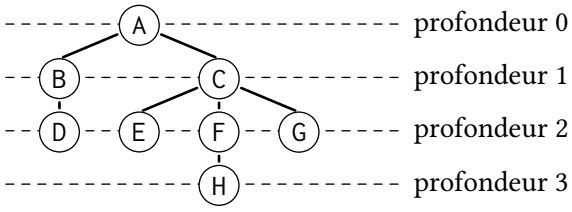


Ici, la racine possède deux sous-arbres, contenant respectivement les nœuds {`B`, `D`} et {`C`, `E`, `F`, `G`, `H`}. Le premier sous-arbre a pour racine `B`, et ainsi de suite. Les sous-arbres d'un nœud forment une séquence finie et ordonnée d'arbres que l'on appelle une *forêt*. Une forêt peut être vide. De façon équivalente, on peut donc définir un arbre comme la donnée d'un nœud, la racine, et d'une forêt, ses sous-arbres. Un arbre réduit à un unique nœud est appelé une *feuille*. Dans l'exemple ci-dessus, les nœuds `D`, `E`, `H` et `G` constituent les quatre feuilles de cet arbre.

Définition 2 – Hauteur

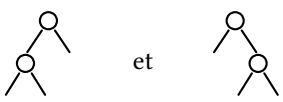
La *hauteur* d'un arbre est définie comme la plus grande distance entre la racine et un nœud de l'arbre. En particulier, un arbre réduit à un seul nœud a pour hauteur 0.

De manière équivalente, on peut définir la *profondeur* de chaque nœud, en considérant que la racine est à la profondeur 0, les racines de ses sous-arbres à la profondeur 1, etc., puis définir la hauteur comme la profondeur maximale d'un nœud. Ainsi, l'arbre ci-dessous a pour hauteur 3, cette distance étant atteinte entre la racine `A` à la profondeur 0 et le nœud `H` à la profondeur 3.



Commentaire

Aussi surprenant que cela puisse paraître, un arbre binaire *n'est pas un arbre*. En premier lieu, un arbre binaire peut être vide, c'est-à-dire ne contenir aucun nœud, là où un arbre contient toujours au moins un nœud. Par ailleurs, les arbres binaires font la distinction entre le sous-arbre gauche et le sous-arbre droit. Ainsi, les deux arbres binaires suivants sont distincts. On parle d'arbres *positionnels* pour les arbres binaires.



À la différence, il n'y a qu'un seul arbre contenant deux nœuds :

On note également que le dessin d'un arbre binaire inclut de « petites pattes », illustrant la présence de sous-arbres vides, et qu'il n'y a pas lieu de dessiner de telles petites pattes dans le cas d'un arbre.

Représentation en machine

Langage OCaml

Le programme OCaml suivant définit un type `'a tree` pour des arbres polymorphes dont les nœuds portent des étiquettes de type `'a`. On utilise ici le type prédéfini des listes d'OCaml (type `list`) pour représenter la forêt des sous-arbres.

```
type 'a tree =
  | N of 'a * 'a tree list

let rec size (N (_, tl)) =
  1 + size_forest tl
and size_forest tl = match tl with
  | []      -> 0
  | t :: tl -> size t + size_forest tl
```

Le programme contient également la définition d'une fonction `size` qui calcule le nombre de nœuds d'un arbre. Elle est définie mutuellement récursivement avec une fonction `size_forest` qui calcule le nombre de nœuds d'une forêt.

Langage C

Le programme C suivant contient la définition d'un type `tree` pour des arbres étiquetés par des entiers. Un arbre est un pointeur vers une structure `Tree` contenant trois champs : une étiquette, ici de type `int`, dans un champ `value` ; un pointeur vers le premier sous-arbre dans un champ `children` ; et un pointeur vers l'arbre suivant dans la forêt dans un champ `next`. Lorsque le nœud n'a pas de sous-arbre, le champ `children` vaut `NULL`. De même, lorsque le nœud est le dernier d'une forêt, le champ `next` vaut `NULL`.

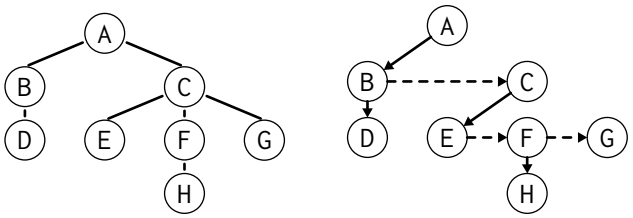
```
typedef struct Tree {
  int value;
  struct Tree *children; // premier sous-arbre
  struct Tree *next;     // suivant dans la forêt
} tree;

tree *tree_create(int v) {
  tree *t = malloc(sizeof(struct Tree));
  t->value = v;
  t->children = NULL;
  t->next = NULL;
  return t;
}

void tree_add_first_child(tree *t, tree *c) {
  assert(c->next == NULL);
  c->next = t->children;
  t->children = c;
}

int tree_size(tree *t) {
  int s = 1;
  for (tree *c = t->children; c != NULL; c = c->next) {
    s += tree_size(c);
  }
  return s;
}
```

L'arbre dessiné ci-dessous à gauche est représenté en mémoire à l'aide de 8 structures de type `Tree` où les pointeurs `children` et `next` sont positionnés selon le dessin de droite. Le champ `children` est représenté par une flèche pleine et le champ `next` par une flèche en pointillés.



Les pointeurs `NULL` ne sont pas représentés. Le programme contient également une fonction `tree_create` pour créer un nouveau nœud (avec des champs `children` et `next` initialisés à `NULL`) et une fonction `tree_add_first_child` pour ajouter un nouvel arbre en tête des sous-arbres d'un nœud donné. Ainsi, on construit l'arbre ci-dessus en ajoutant d'abord le nœud `C` comme second sous-arbre du nœud `A` puis le nœud `B` comme premier sous-arbre.

Comparaison

Les représentations en C et en OCaml semblent très différentes. Elle sont en réalité très proches. En effet, les champs `next` de la structure `Tree` forment une structure de liste simplement chaînée pour représenter la forêt et on trouve les mêmes pointeurs derrière le type `list` d'OCaml. Au final, la seule différence, minime, réside dans l'utilisation d'un seul bloc mémoire en C (la structure `Tree`) là où la représentation OCaml en utilise deux (le constructeur `N` et le constructeur `::`).

DOM

Un cas d'utilisation des arbres est la représentation des documents HTML utilisés pour les pages web. Le consortium W3C, qui standardise notamment le format HTML, définit le *Document Object Model* (DOM) qui est l'API permettant de naviguer dans un document et de le modifier. Dans cette API, une balise HTML est un nœud de l'arbre et on peut accéder au premier fils par un pointeur `firstChild` et au frère d'un nœud par un pointeur `nextSibling`.
L'API est cependant plus riche. Elle permet notamment de naviguer vers le haut avec un pointeur `parentNode` et vers la gauche avec un pointeur `previousSibling`.

Conversions

Il existe un isomorphisme naturel entre les arbres binaires et les arbres. C'est particulièrement frappant si on observe les deux types C suivants pour les arbres binaires et les arbres.

```
struct Node { int value; struct Node *left,     *right; }
struct Tree { int value; struct Tree *children, *next;  }
```

Ces deux types, absolument identiques, ne diffèrent que par les noms des structures et des champs. Plus précisément, il y a isomorphisme entre :

- un arbre binaire et une forêt, d'une part ;
- un arbre binaire non vide dont le sous-arbre droit est vide et un arbre, d'autre part.

Le programme OCaml suivant définit quatre fonctions OCaml qui réalisent ces deux isomorphismes. On note en particulier que la fonction `bintree_of_tree` n'échoue jamais mais qu'en revanche la fonction `tree_of_bintree` échoue si son argument n'est pas un arbre binaire non vide dont le sous-arbre droit est vide. En effet, si l'arbre binaire est vide on obtiendra une forêt vide et si son sous-arbre droit n'est pas vide on obtiendra une forêt contenant au moins deux arbres.

```
(* conversion d'un arbre en un arbre binaire *)
let rec btree_of_forest tl = match tl with
  | [] -> Bintree.E
  | (Tree.N (x, ch)) :: tl -> Bintree.N (btree_of_forest ch, x, btree_of_forest
     tl)

let btree_of_tree t =
  btree_of_forest [t]

(* et inversement *)
let rec forest_of_btree t = match t with
  | Bintree.E -> []
  | Bintree.N (l, x, r) -> Tree.N (x, forest_of_btree l) :: forest_of_btree r

let tree_of_btree t = match forest_of_btree t with
  | [t] -> t
  | _   -> invalid_arg "tree_of_btree"
```

De la même façon qu'on a programmé différents parcours pour les arbres binaires, on peut écrire des parcours pour les arbres. Il y a cependant une différence : le nombre de sous-arbres est maintenant arbitraire et il n'y donc plus vraiment lieu de parler de parcours infixe. En revanche, il reste pertinent de parler de parcours préfixe et postfixe, où le nœud est visité avant ou après ses sous-arbres. Dans les deux cas, on peut le proposer d'une façon très générique avec une fonction de type `('a -> unit)-> 'a tree -> unit` qui applique une fonction passée en argument à chaque nœud de l'arbre. Le programme ci-dessous contient le code OCaml de deux fonctions `preorder` et `postorder` de ce type-là. La fonction `List.iter` permet de parcourir la forêt ; on aurait pu écrire une fonction récursive spécifique.

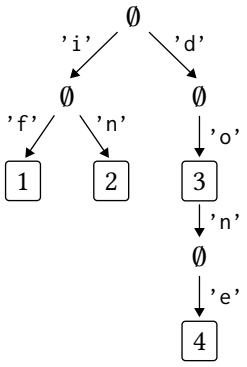
```
let rec preorder f (N (x, tl)) =
  f x;
  List.iter (preorder f) tl

let rec postorder f (N (x, tl)) =
  List.iter (postorder f) tl;
  f x
```

.....

Arbres préfixes

Cette section présente une structure d'arbre pour représenter des tableaux associatifs dont les clés sont des chaînes de caractères. Dans ces arbres, chaque branche est étiquetée par une lettre et chaque nœud contient une valeur si la séquence de lettres menant de la racine de l'arbre à ce nœud est une entrée dans le tableau associatif. Voici par exemple l'arbre représentant le tableau associatif $\{$`"if"` $\mapsto 1$, `"in"` $\mapsto 2$, `"do"` $\mapsto 3$, `"done"` $\mapsto 4\}$:



Chaque nœud correspond à un mot w décrit par le chemin depuis la racine. En particulier, la racine correspond au mot vide. Le nœud contient soit $\varnothing$ s'il n'y a pas de valeur associée à w, soit la valeur associée à w, ici encadrée. Un tel arbre est appelé un *arbre préfixe*, plus connu sous le nom de *trie* en anglais [1].
L'intérêt d'une telle structure de données est de borner le temps de recherche à la longueur de la clé la plus longue, quel que soit le nombre de clés. Plus précisément, cette propriété est garantie seulement si toutes les feuilles d'un arbre préfixe contiennent une valeur. Cette *bonne formation* des arbres préfixes sera maintenue par toutes les opérations définies ci-dessous.

Réalisation en OCaml

On se propose de réaliser une telle structure en OCaml, sous la forme d'un tableau associatif mutable dont les clés sont des chaînes de caractères et les valeurs d'un type quelconque.
Un nœud de l'arbre est représenté par le type suivant, où la valeur est stockée dans le champ mutable `value` :

```
type 'a trie = {
  mutable value: 'a option;
          branches: (char, 'a trie) Hashtbl.t;
}
```

On utilise ici la bibliothèque `Hashtbl` pour représenter le branchement vers les sous-arbres. Ainsi, dans l'exemple ci-dessus, le champ `branches` de la racine de l'arbre est une table de hachage contenant deux entrées, une associant le caractère `'i'` au sous-arbre de gauche, et une autre associant le caractère `'d'` au sous-arbre de droite. Une feuille de l'arbre a une table `branches` qui est vide.
Initialement, on crée un arbre préfixe vide comme un arbre réduit à un unique nœud où le champ `value` vaut `None` et où `branches` est une table vide :

```
  let create () =
    { value = None; branches = Hashtbl.create 8; }
```

1. Le mot *trie* vient du mot re*trie*val.

La valeur 8 est choisie ici arbitrairement. De toutes façons, les tables de hachage d'OCaml s'adaptent dynamiquement au nombre d'entrées.

Recherche d'une clé

Écrivons une fonction `get` qui renvoie la valeur associée à une clé `s`, le cas échéant, et lève l'exception `Not_found` sinon. La recherche consiste à descendre dans l'arbre en suivant les lettres de `s`. On le fait ici à l'aide d'une fonction récursive `find`, en se servant d'une variable `i` pour parcourir la chaîne `s`.

```
let get t s =
  let rec find t i =
```

Lorsqu'on parvient au bout de la chaîne, on inspecte le champ `value` pour renvoyer la valeur ou lever une exception :

```
    if i = n then
      (match t.value with None -> raise Not_found | Some v -> v)
```

Sinon, on poursuit la recherche dans le sous-arbre correspondant au `i`-ième caractère, obtenu en cherchant dans la table `branches`.

```
    else
      find (Hashtbl.find t.branches s.[i]) (i + 1)
```

Lorsque la branche n'existe pas, la fonction `Hashtbl.find` lève l'exception `Not_found`, qui ne sera pas rattrapée ici, mais c'est exactement le comportement attendu. Enfin, on lance la recherche à partir de la racine `t`, avec la valeur 0 pour `i`.

```
  in
  find t 0
```

On note que le code fonctionne correctement pour une chaîne vide, avec une inspection immédiate de la valeur située à la racine.

Ajout d'une nouvelle entrée

L'insertion d'une entrée pour la clé `s` dans un arbre préfixe consiste à descendre le long de la branche étiquetée par les lettres de `s`, de manière similaire au parcours effectué pour la recherche. C'est cependant légèrement plus subtil, car il faut éventuellement créer de nouvelles branches dans l'arbre pendant la descente.
Comme pour la recherche, on procède à la descente avec une fonction récursive locale, ici appelée `add`.

```
let put t s v =
  let rec add t i =
```

Lorsqu'on parvient au terme du mot `s`, on y écrit la valeur `v`, possiblement en écrasant une valeur précédente.

```
    if i = String.length s then
      t.value <- Some v
    else
```

Pour poursuivre la descente dans l'arbre, on commence par déterminer le sous-arbre `b` correspondant au caractère `s.[i]`, en le créant si nécessaire.

```
      let b =
        try
          Hashtbl.find t.branches s.[i]
        with Not_found ->
          let b = create () in
          Hashtbl.add t.branches s.[i] b;
          b
```

On peut alors poursuivre la descente sur le caractère suivant.

```
      in
      add b (i+1)
```

Enfin, l'insertion consiste à lancer `add` avec la valeur 0 pour `i`.

```
  in
  add t 0
```

Comme pour la recherche, l'insertion fonctionne correctement sur un mot vide. Il est instructif de prendre le temps de dérouler le code ci-dessus en ajoutant un nouvelle entrée à l'arbre pris en exemple plus haut, et idéalement pour une clé qui a un préfixe commun avec une autre entrée, par exemple `"dots"`.
Le code complet est donné dans le programme OCaml **??**. L'ajout d'une opération `remove` à cette structure de données est laissé en exercice.

```
type 'a trie = {
  mutable value: 'a option;
       branches: (char, 'a trie) Hashtbl.t;
}

let create () =
  { value = None; branches = Hashtbl.create 8; }

let get t s =
  let rec find t i =
    if i = String.length s then
      (match t.value with None -> raise Not_found | Some v -> v)
    else
      find (Hashtbl.find t.branches s.[i]) (i + 1)
  in
  find t 0

let put t s v =
  let rec add t i =
    if i = String.length s then
      t.value <- Some v
    else
      let b =
        try
          Hashtbl.find t.branches s.[i]
        with Not_found ->
          let b = create () in
          Hashtbl.add t.branches s.[i] b;
          b
      in
      add b (i+1)
  in
  add t 0
```

Complexité

Les deux fonctions `get` et `put` procède selon un parcours de la clé, caractère par caractère. Pour chaque caractère, elles effectuent des opérations de temps constant et uniquement des appels aux fonctions `Hashtbl.find` et `Hashtbl.add`, que l'on peut considérer comme s'exécutant en temps constant amorti. Dès lors, l'ajout et la recherche d'une clé s'exécutent en un temps proportionnel à la longueur de cette clé. En particulier, cela ne dépend pas du nombre d'entrées dans l'arbre préfixe.
Si les clés sont des mots de longueur bornée, alors on peut donc considérer que les opérations sur l'arbre préfixe se font en temps constant (amorti), comme avec une table de hachage. Il est alors légitime de comparer ces deux solutions lorsque l'on veut réaliser un tableau associatif où les clés sont des chaînes. D'un côté, la table de hachage reste plus efficace, en temps comme en espace. Une expérience rapide montre que stocker tous les mots du dictionnaire français dans une table de hachage prend 3 fois moins de temps et 7 fois moins de place que de les stocker dans un arbre préfixe. Cela s'explique notamment par toutes les petites tables de hachage stockées dans chaque nœud de l'arbre préfixe. D'un autre côté, l'arbre préfixe permet des opérations que la table de hachage ne permet pas. Ainsi, on peut trouver facilement toutes les clés qui ont un préfixe donné. Il suffit en effet de descendre dans l'arbre selon ce préfixe, puis de parcourir tout le sous-arbre sur lequel on est parvenu. L'arbre préfixe permet également de trouver le plus grand préfixe d'une chaîne donnée qui est une clé dans l'arbre. Il suffit en effet de descendre dans l'arbre tant que cela est possible, en maintenant le plus grand préfixe qui est une clé. Cette opération permet de compresser du texte avec l'algorithme de Lempel–Ziv–Welch..

Arbre de Patricia

Un arbre préfixe peut contenir des branches linéaires de nœuds qui ne contiennent pas d'entrée et qui n'ont qu'un seul branchement vers un autre nœud. C'est le cas sur l'exemple donné en introduction avec le branchement `'d'` puis `'o'` ou encore le branchement `'n'` puis `'e'`. Dans ce cas, on peut regrouper les nœuds en un seul et indiquer plusieurs caractères dans le branchement (ici `"do"` et `"ne"` respectivement). On parle alors d'*arbre de Patricia*.