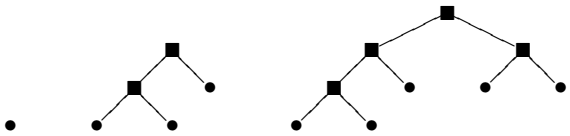
▶ **MPSI – Option Informatique**
Année 2001, Huitième TP Caml
Vincent Simonet (`http://cristal.inria.fr/~simonet/`)

Arbres binaires de recherche

1 Arbres binaires

Dans cette première partie, nous introduisons une structure de données très classique en Informatique : les arbres binaires. De manière informelle, il s'agit de structures de la forme suivante :



Il est naturel de définir la notion d'arbre binaire (étiqueté par des éléments d'un ensemble X) de manière récursive :
- Il existe un arbre binaire appelé « arbre vide » et noté $\circ$.
- Si g et d sont deux arbres binaires et x est un élément de X alors $\square(g, x, d)$ est un arbre binaire.

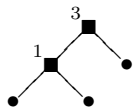
De telles arborescences permettent de repésenter des données de divers genres : structures hierarchiques, expressions arithmétiques ou, comme nous le verrons dans la deuxième partie, des ensembles ordonnés et, dans la troisième partie, des tables d'association.

Pour manipuler des arbres binaires en *Caml*, nous allons créer un type « variante » *tree* :

```
type 'a tree =
    Empty
  | Node of ('a tree) * 'a * ('a tree)
;;
```

Dans cette définition, la variable de type 'a représente le type des éléments qui étiquettent les nœuds de l'arbre. Il est à noter que cette définition est récursive et très proche de la définition formelle de la notion d'arbre que nous avons donné précédemment.

▶ **Question 1** *Définissez en Caml l'arbre suivant :*



Pour manipuler de tels arbres, on écrit en général des fonctions récursives avec un filtrage, comme pour les listes :

```
let ma_fonction = function
    Empty -> ...
  | Node (g, x, d) -> ...
;;
```

1.1 Taille et hauteur

On appelle *taille* d'un arbre le nombre de ses nœuds (sans compter les feuilles).

▶ **Question 2** *Écrivez en Caml une fonction* size *qui calcule la taille d'un arbre.*

value size : *'a tree → int*

On appelle *hauteur* d'un arbre la longueur maximale d'un chemin *direct* de la racine à un nœud quelconque de l'arbre.

▶ **Question 3** *Écrivez en Caml une fonction* height *qui calcule la hauteur d'un arbre.*

value height : *'a tree → int*

▶ **Question 4** *Pouvez-vous majorer la taille d'un arbre binaire en fonction de sa hauteur ? Inversement, peut-on majorer sa hauteur en fonction de sa taille ?*

1.2 Maximum

On suppose désormais que l'ensemble X des étiquettes est ordonné.

▶ **Question 5** *Écrivez une fonction* max_tree *qui calcule la plus grande étiquette présente dans un arbre. Si l'arbre passé en argument est vide, vous lèverez une exception.*

value max_tree : $'a\ tree \to 'a$

2 Arbres binaires de recherche

Un arbre binaire de recherche est un arbre binaire vérifiant la propriété suivante : *Pour tout nœud* $\square(g, x, d)$*, les étiquettes apparaissant dans le sous-arbre gauche* g *sont strictement inférieures à* x *et celles du sous-arbre droit* d *strictement supérieures.*
On souhaite généralement disposer de trois opérations « primitives » sur de telles structures : la recherche (tester si un élément est présent ou non), l'ajout d'un élément et la suppression. Ces arbres permettent ainsi de représenter des ensembles en effectuant les opérations ensemblistes usuelles de manière relativement efficace.

▶ **Question 6 (Recherche)** *Écrivez une fonction* mem *qui teste si une étiquette* x *apparaît dans un arbre binaire de recherche* a*. La complexité en temps de votre fonction devra être dominée par la hauteur de l'arbre.*

value mem : $'a \to 'a\ tree \to bool$

Pour ajouter un élément x à un arbre binaire de recherche, on peut procéder de la manière suivante. On parcourt l'arbre en partant de la racine vers le bas. À un nœud étiqueté y, on poursuit vers la gauche si $x < y$ et vers la droite si $x > y$.

▶ **Question 7 (Insertion)** *Programmez une fonction* add *qui ajoute un élément à un arbre binaire de recherche. (Vous pourrez supposer que l'élément à ajouter n'est pas déjà dans l'arbre.)*

value add : $'a \to 'a\ tree \to 'a\ tree$

La suppression est un opération plus délicate : si on veut supprimer une étiquette située sur un nœud intérieur d'un arbre, il faut la remplacer par une autre étiquette ! Supposons que vous souhaitiez supprimer l'étiquette y présente dans un nœud $\square(g, y, d)$. Deux cas se présentent :

1. Si le sous-arbre gauche g est vide, il suffit de remonter le sous-arbre droit d.
2. Sinon, il faut rechercher la plus grande étiquette z de g, la supprimer de g et remplacer y par z.

▶ **Question 8 (Suppression)** *Écrivez une fonction* remove_max *prenant pour argument un arbre binaire de recherche non-vide* a *et qui retourne le couple* (a', z) *où* z *est la plus grande étiquette apparaissant dans* a *et* a' *l'arbre obtenu à partir de* a *en retirant le nœud étiqueté par* z*.*
Déduisez-en alors une fonction remove*.*

value remove_max : $'a\ tree \to 'a\ tree \times 'a$

value remove : $'a \to 'a\ tree \to 'a\ tree$

3 Tables d'association

Soient K et E deux ensembles, les éléments de K étant appelés *clefs*, et ceux de E *objets*. On supposera l'ensemble K ordonné. Une *table d'association* M sur K et E est une partie de $K \times E$ telle que pour toute clef $k \in K$ il existe au plus un objet $e \in E$ tel que le couple (k, e) soit dans M.
On peut donc voir une table d'association M comme une *application partielle* de K dans E. On peut manipuler efficacement une table d'association en la représentant sous la forme d'un arbre binaire de recherche, en utilisant sur les couples l'ordre $\leq$ défini par :

$$(k, e) \leq (k', e') \Leftrightarrow k \leq k'$$

On souhaite disposer de trois opérations élémentaires sur les tables d'association :
– *find* : étant données une table M et une clef k trouver l'objet e tel que $e = M(k)$ s'il existe.
– *add* : étant donnés une table M, une clef k n'apparaissant pas dans M et un objet e, ajouter l'association (i.e. le couple) (k, e) à M.
– *remove* : étant données une table M et une clef k telles qu'il existe e tel que $(k, e) \in M$, retirer le couple (k, e) de M.

▶ **Question 9** *Adaptez les fonctions de la section 2 de manière à implémenter efficacement ces trois opérations sur les tables d'association.*

▶ **MPSI – Option Informatique**
Année 2001, Huitième TP Caml
Vincent Simonet (`http://cristal.inria.fr/~simonet/`)

Arbres binaires de recherche

Un corrigé

▶ Question 1

```
Node (Node (Empty,
            1,
            Empty),
      3,
      Empty)
;;
```

▶ Question 2

```
let rec size = function
    Empty -> 0
  | Node (g, x, d) -> 1 + size g + size d
;;
```

▶ Question 3

```
let rec height = function
    Empty -> 0
  | Node (g, x, d) -> 1 + max (height g) (height d)
;;
```

▶ Question 5

```
let rec max_tree = function
    Empty ->
      raise (Invalid_argument "max_tree")
  | Node (Empty, x, Empty) -> x
  | Node (g, x, Empty) -> max x (max_tree g)
  | Node (Empty, x, d) -> max x (max_tree d)
  | Node (g, x, d) ->
      max x (max (max_tree g) (max_tree d))
;;
```

▶ Question 6

```
let rec mem x = function
    Empty -> false
  | Node (_, y, _) when x = y -> true
  | Node (g, y, d) ->
      if x < y then mem x g else mem x d
;;
```

▶ Question 7

```
let rec add x = function
    Empty -> Node (Empty, x, Empty)
  | Node (g, y, d) ->
      if x < y then Node (add x g, y, d)
      else Node (g, y, add x d)
;;
```

▶ Question 8

```
let rec remove_max = function
    Empty ->
      raise (Invalid_argument "remove_max")
  | Node (g, x, Empty) -> (g, x)
  | Node (g, y, d) ->
      let (d', x) = remove_max d in
      (Node (g, y, d'), x)
;;

let rec remove x = function
    Empty -> Empty
  | Node (Empty, y, d) when y = x -> d
  | Node (g, y, d) when y = x ->
      let g', z = remove_max g in
      Node (g', z, d)
  | Node (g, y, d) ->
      if x < y then Node (remove x g, y, d)
      else Node (g, y, remove x d)
;;
```

▶ Question 9

```
let rec find k = function
    Empty -> raise Not_found
  | Node (g, (k', e), d) when k = k' -> e
  | Node (g, (k', _), d) ->
      if k < k' then find k g
      else find k d
;;

let rec add k e = function
    Empty -> Node (Empty, (k, e), Empty)
  | Node (g, (k', e), d) ->
      if k < k' then Node (add k e g, (k', e), d)
      else Node (g, (k', e), add k e d)
;;

let rec remove k = function
    Empty -> Empty
  | Node (Empty, (k', e), d) when k = k' -> d
  | Node (g, (k', e), d) when k = k' ->
      let g', z = remove_max g in
      Node (g', z, d)
  | Node (g, (k', e), d) ->
      if k < k' then Node (remove k g, (k', e), d)
      else Node (g, (k', e), remove k d)
;;
```