Cours 08 - Les arbres binaires

MPSI - Prytanée National Militaire

Pascal Delahaye

17 juin 2016

La structure de donnée `arbre binaire` est une structure permettant :

1. d'une part de stocker des données organisées sous la forme d'un arbre,
2. d'autre part d'organiser des données afin d'optimiser le temps d'accès.

`Exemple 1.`

1. Arbre généalogique ou phylogénétique
2. Arbre de décision
3. Organisation des fichiers par un système d'exploitation

Cette structure est à la fois de type "somme" et de type "produit" et se définit de façon récursive.

Rappels sur les types "somme étiquetés" :

La structure de définition d'un type somme étiqueté est :

```
type nom_type =
    | toto              # si on souhaite que la valeur toto soit de ce type
    | nom1 of type1
    | nom2 of type2
```

On définit alors les variables de ce nouveau type de la façon suivante :

```
let var = toto;;
let var1 = nom1(valeur de type1);;
let var2 = nom2(valeur de type2);;
```

1 Définition 1

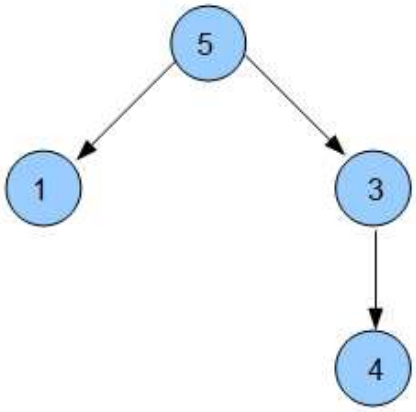
Dans cette première définition, les cellules contenant l'information sont soit vide, soit d'un type donné :

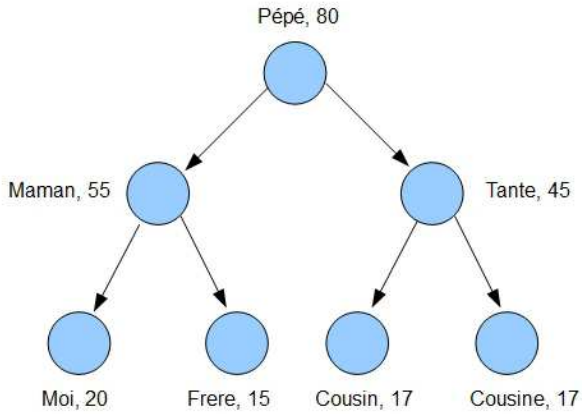
1. Initiatisation : un arbre vide
2. Définition récursive : arbre = (arbre, element, arbre)

```
type 'a arbre =
   |Vide                                    # Souvent noté "nil"
   |Noeud of ('a arbre) * 'a * ('a arbre);;
```

Remarque 1. les mots "arbre", "Vide" et "Noeud" peuvent être remplacés par n'importe quels autres mots comme "arbbin", "Nil" et "Sommet".

Les arbres sont par convention représentés à l'envers.





1.
```
let arbre1 = Noeud(Vide , 1 , Vide) ,
             5,
             Noeud(Noeud(Vide, 4 , Vide),
                   3,
                   Vide);;
```

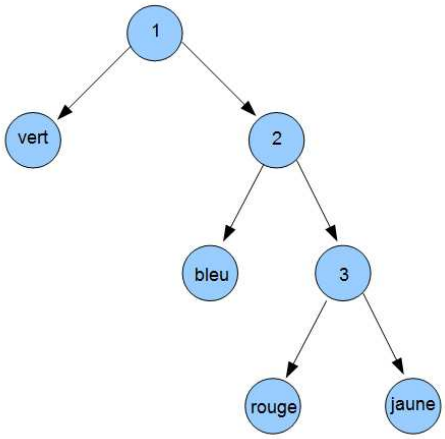
2.
```
let arbre2 = Noeud(Noeud(Vide, ("moi", 20), Vide) ,
                   ("maman", 55) ,
                    Noeud(Vide, ("frere", 15), Vide)) ,
                ("pépé",80),
                 Noeud(Noeud(Vide, ("cousin", 17), Vide),
                   ("tante",45) ,
                    Noeud(Vide, ("cousine", 25) , Vide ));;
```

Remarque 2. Pour ne pas se perdre dans la définition d'un arbre, on veillera à respecter la présentation ci-dessus.

2 Définition 2

Il est aussi possible de définir une structure d'arbre où les noeuds sont d'un type donné et les feuilles (noeuds situés aux extrémités de l'arbre) d'un autre type :

```
type ('n, 'f) arbre =
    |nil
    |Feuille of 'f
    |Noeud of (('n, 'f) arbre) * 'n * (('n, 'f) arbre);;
```



Cet arbre est de type `int * string`

```
let arbre3 = Noeud( Feuille "vert" ,
                    1,
                    Noeud( Feuille "bleu",
                           2 ,
                           Noeud( Feuille "rouge",
                                  3,
                                  Feuille "jaune" )));;
```

Remarque 3. Plus simplement, on pourra définir le type `arbre` de la façon suivante :

```
type ('n, 'f) arbre =
    |F of 'f
    |N of (('n, 'f) arbre) * 'n * (('n, 'f) arbre);;
```

Remarque 4. En Python, il est possible de définir un arbre grâce à la structure de liste. En effet, les composantes des listes pouvant être de types distincts, l'arbre précédent peut être codé sous forme de listes emboitées par :

```
arbre = [["vert"],1,[["bleu"],2,[["rouge"],3,["jaune"]]]]
```

3 Vocabulaire

1. Chaque élément de l'arbre est appelé un *sommet*.

2. Chaque sommet renvoyant sur d'autres données (intersection) est appelé un *noeud*.
   On parle aussi parfois de *noeuds internes*.
   Chaque noeud contient une information d'un type (déterminé lors de la définition de l'arbre) et deux liens (adresses) vers les deux sous-arbres qu'il relie.

3. Le sommet de l'arbre est un noeud particulier appelé la *racine* de l'arbre

4. Les sommets terminaux (extrémités) de l'arbre sont appelés les *feuilles* de l'arbre.
   On parle aussi parfois de *noeuds externes*.
   Chaque feuille contient une information d'un type (déterminé lors de la définition de l'arbre) et qui peut différer du type des noeuds.

5. Un noeud donné possède deux *noeuds fils* : un fils gauche et un fils droit (qui éventuellement peuvent être des feuilles). Selon la définition, les feuilles quant à elles, renvoient soit sur des noeuds vides (avec la définition 1) soit sur rien (avec la définition 2).

6. A l'exception de la racine, tout noeud possède un unique *père* : le noeud dont il est le fils.

7. Les noeuds et feuilles situés sous un noeud donné sont appelés les *descendants* du noeud.
   On peut définir cette notion de façon récursive.
   La relation "*est un descendant de*" définit un ordre partiel sur les noeuds et feuilles de l'arbre.
   La racine de l'arbre est le plus petit élément pour cette relation d'ordre.

8. Les noeuds situés au dessus d'un noeud ou d'une feuille donné sont appelés les *ancêtres* du noeud.
   On peut définir cette notion de façon récursive.

9. Les liens entre un noeud et ses fils sont appelés les *branches* de l'arbre.

10. Si l'on enlève un noeud à un arbre binaire donné ainsi que toutes les branches qui en partent ou qui y mènent, on obtient :
    — soit un nouvel arbre
    — soit 2 ou 3 arbres que l'on appelle alors une *forêt*.

11. On appelle l'*arité* d'un noeud, le nombre de branche qui partent de ce noeud.
    Dans le cas des arbres binaires, cette arité est égale à 2.

4 Opérations sur la structure d'arbre

Opérations du type "arbre" :

1. **Trois constructeurs :**
   1. *creer-arbre-vide* de type `unit -> arbre`
      qui crée un arbre vide
   2. *feuille* de type `element -> arbre`
      qui construit un arbre limité à une feuille contenant `element`
   3. *noeud* de type `element -> arbre -> arbre -> arbre`
      qui construit un arbre de sommet `element` et dont les deux fils (droit et gauche) sont les deux arbres en arguments

2. **Un prédicat :**
   1. *est-vide* de type `arbre -> bool`
      qui teste si un arbre est vide

3. **Quatre fonctions de sélection :**
   1. *fils_gauche* de type `arbre -> arbre`
      qui renvoie l'arbre fils de gauche
   2. *fils_droit* de type `arbre -> arbre`
      qui renvoie l'arbre fils de droite
   3. *etiquette_interne* de type `arbre -> element`
      qui renvoie la donnée stockée à la racine de l'arbre lorsqu'il est non vide et non réduit à une feuille.
   4. *etiquette_externe* de type `arbre -> element`
      qui renvoie la donnée stockée à la racine de l'arbre lorsque celui-ci est réduit à une feuille.

5 Quelques algorithmes de base

La structure d'"arbre" étant définie de façon récursive (comme d'ailleurs la structure de "liste" en CAML), elle est particulièrement adaptée à la programmation récursive.

Exercice : 1

Nombre de feuilles et nombre de noeuds :

1. Calcul du nombre de feuilles d'un arbre :

```
let rec nbrf = function
               | F(_) -> 1
               | N(g,_,d) -> nbrf g + nbrf d;;
```

2. Calcul du nombre de noeuds d'un arbre :

```
let rec nbrn = function
               | F(_) -> 0
               | N(g,_,d) -> 1 + nbrn g + nbrn d;;
```

3. Relation entre le nombre de noeuds et le nombre de feuilles :
   Pour un arbre t, on note :
   — $n(t)$ le nombre de noeuds de l'arbre t
   — $f(t)$ le nombre de feuilles de l'arbre t

   Montrer par induction fonctionnelle (sorte de récurrence sur les noeuds de l'arbre en prenant une feuille pour initialisation) que dans le cas d'un arbre binaire, on a la relation :

$$f(t) = n(t) + 1$$

Exercice : 2

Hauteur d'un arbre

1. Il s'agit de la profondeur maximale d'une feuille de l'arbre où la profondeur d'une feuille est définie comme le nombre de branches qui séparent la feuille de sa racine.

```
let rec hauteur = function
               | F(_) -> 0
               | N(g,_,d) -> 1 + max (hauteur g, hauteur d);;
```

2. En reprenant les notations précédentes, et en notant $h(t)$ la hauteur d'un arbre t, prouver par induction fonctionnelle que :

$$h(t) + 1 \leq f(t) \leq 2^{h(t)}$$

3. Montrer, à l'aide des deux arbres suivants (que vous représenterez par un dessin) que les bornes peuvent être atteintes.

Arbre complet et Arbre peigne droit

```
let rec arbre_complet = function
          |1 -> F("feuille")
          |n -> let t = arbre_complet (n-1) in N(t,n, t);;
```

Vérifier que $c(n) = O(2^n)$.

```
let rec peigne_droit = function
          |1 -> F("feuille")
          |n -> let t = peigne_droit (n-1) in N(F("feuille"), n, t);;
```

Vérifier que $c(n) = O(n)$.

Remarque 5. **La complexité minimale d'un algorithme de tri par comparaison 2 à 2 est un** $O(n\ln(n))$
Les différentes étapes de la mise en oeuvre d'un algorithme de tri par comparaison 2 à 2 des éléments peuvent se représenter par un arbre binaire où chaque branche partant d'un noeud correspond aux deux éventualités lors de la comparaison de 2 éléments. L'arbre droit correspondant à la suite de l'algorithme dans l'un des cas et l'arbre gauche la suite dans l'autre cas. Chaque feuille de l'arbre représente alors un permutation effectuée sur la configuration initiale. Comme il y a en tout $n!$ permutations possibles des n éléments, l'arbre doit contenir $n!$ feuilles et a donc au minimum une hauteur de l'ordre d'un $O(\ln(n!))$. Dans le pire des cas, la complexité de l'algorithme en terme de nombre de comparaisons sera alors un $O(\ln(n!))$ ce qui donne un $O(n\ln(n))$ d'après la formule de Stirling.

Exercice : 3

Parcours d'arbres binaires

Il est important de savoir parcourir tous les noeuds et feuilles d'un arbre afin par exemple, de :
- Modifier les contenus (étiquettes) de chaque noeud
- Rechercher un noeud de contenu (étiquette) particulier
- Compter les noeuds de contenu donné... etc...

Il existe deux façons distinctes de parcourir un arbre : le parcours en profondeur et le parcours en largeur.

1. **Parcours en profondeur :**
   On visite entièrement le sous-arbre de gauche avant de visiter celui de droite. Cela donne l'algorithme suivant :

```
let rec parcours_prof = function
                        | F(_) -> action à effectuer
                        | N(g, a , d) -> action à effectuer sur "a"
                                         parcours_prof g;
                                         parcours_prof d;;
```

   Représenter le parcours effectué par cet algorithme sur un arbre de votre choix.

Parcours d'un arbre en profondeur

   Si l'on considère les noeuds comme des îles et les branches comme des bandes de terres, on constate que le parcours effectué par cet algorithme correspond au parcours effectué par un navigateur qui circulerait autour de l'arbre dans le sens trigonométrique en partant de la racine.

2. **Parcours en largeur :**
   On commence par la racine, puis on visite chacun de ses fils (qui sont en profondeur 1) en commençant par la gauche. On visite les noeuds et les feuilles situées en profondeur 2... etc...

Parcours d'un arbre en largeur

La programmation de cette méthode est plus compliquée que la précédente et sera vue en deuxième année...