Cours 08 - Les arbres binaires

MPSI - Prytanée National Militaire

Pascal Delahaye

20 mai 2017

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 :

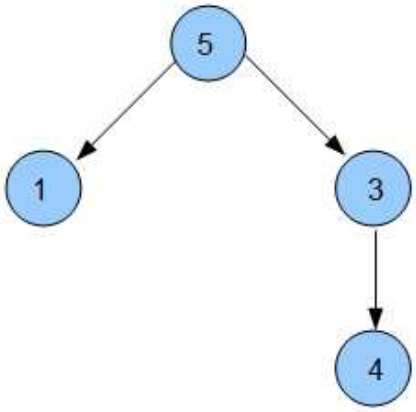
1. Initiatisation : un arbre vide
2. Définition récursive : arbre = (arbre, element, arbre)

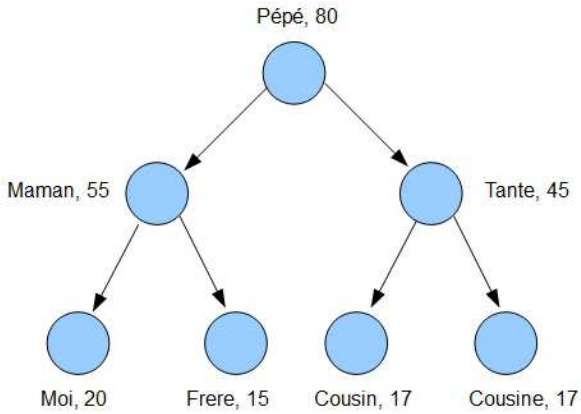
CAML

```
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);;
```

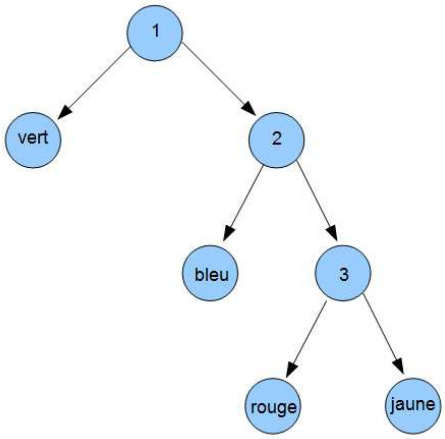
2.
```
let arbre2 = Noeud(Noeud(Vide, ("moi", 20), Vide) ,
                   ("maman", 55) ,
                    Noeud(Vide, ("frere", 15), Vide)) ,
                ("pépé",80),
                 Noeud(Noeud(Vide, ("cousin", 17), Vide),
                   ("tante",45) ,
                    Noeud(Vide,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

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



Cet arbre est de type : `(int, string) arbre`

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

Remarque 3. Plus simplement, si toutes les branches se terminent par des feuilles, on pourra définir le type `arbre` de la façon suivante :

CAML
```
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. La *hauteur* d'un noeud ou d'une feuille est le nombre de branches qui le ou la relie à la racine.

11. La *hauteur* d'un arbre est le maximum des hauteurs des feuilles.

12. La somme des hauteurs des différents noeuds et feuilles est appelée sa *longueur de cheminement*.

13. 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*.

14. 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 à gauche de la racine
   2. *fils_droit* de type `arbre -> arbre`
      qui renvoie l'arbre fils à droite de la racine
   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 :

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

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

CAML
```
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 complet**, 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.

CAML
```
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 : $f(t) = 2^{h(t)}$	**Arbre peigne droit :** $f(t) = h(t) + 1$

Construction d'un arbre complet :

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

Dessiner l'arbre obtenu pour $n = 4$.
Vérifier que $c(n) = O(2^n)$.

Construction d'un peigne droit :

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

Dessiner l'arbre obtenu pour $n = 4$.
Vérifier que $c(n) = O(n)$.

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;;
```

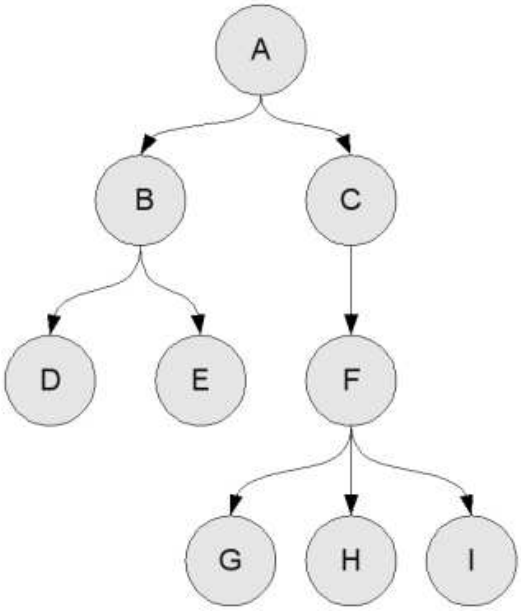
   Représenter le parcours effectué par cet algorithme sur un arbre de votre choix.

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...



La programmation de cette méthode utiliser une file F de stockage.
L'idée est alors la suivante :

Parcours d'un arbre en largeur : File

CAML

```
- On commence par lire la valeur du sommet S puis on stocke dans
  la file ses deux arbres fils N(a) et N(b), puis
- On retire de la file N(a), on récupère la valeur a, puis
  on stocke dans la file les deux fils de a : N(c) et N(d), puis
- On retire de la file N(b), on récupère la valeur b, puis
  on stocke dans la file les deux fils de b : N(e) et N(f) puis... etc...
- on poursuit ainsi l'algorithme jusqu'à ce que la file soit vide ...
```

6 TROIS exemples d'application des arbres

6.1 Complexité minimale d'un algorithme de tri par comparaison

On fixe un algorithme de tri par comparaison 2 à 2 des éléments donné.
Les différentes étapes de la mise en oeuvre de cet algorithme 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 parcours allant de la racine à une feuille de l'arbre (branche de l'arbre) représente les étapes nécessaires au tri d'un liste donnée. Sachant qu'il y a $n!$ listes possibles, alors notre arbre binaire contiendra $n!$ branches (ou feuilles).
Or, pour que ce soit possible, la hauteur h de l'arbre doit vérifier la condition $2^h \geq n!$ et donc $h \leq \frac{\ln n!}{\ln 2}$.
Or, la hauteur h de l'arbre correspond à la liste qui nécessite le plus de comparaisons pour être traitée. La complexité de notre algorithme (en terme de nombre de comparaisons) dans le pire des cas sera donc supérieure à $\frac{\ln n!}{\ln 2}$.
Or, la formule de Stirling nous permet de vérifier que $\ln n! = O(n \ln n)$.

Nous avons ainsi démontré que la complexité dans le pire des cas (en terme de nombre de comparaisons) d'un algorithme de tri est au minimum un $O(n \ln n)$.

Question : Comment peut-on procéder pour rechercher la meilleur complexité moyenne possible ?

6.2 Référencement et compression de données

Lire les deux articles suivants de Hervé Lehning, professeur au lycée Janson de Sailly.
Ces articles présentent deux applications de la structure d'arbre :

1. Une application à la compression de données.

2. Une application pour effectuer une recherche efficace dans un index.

W^xyz

Compression de fichiers

© UC SANTA CRUZ

Pour réduire la taille d'un texte dans un fichier informatique, rien de tel que de dessiner un arbre. On retrouve ainsi plus vite les lettres qui reviennent le plus souvent, et on économise de la place.

Des arbres à compresser

Hervé Lehning,
professeur de mathématiques spéciales au lycée Janson-de-Sailly.
herve.lehning@prepas.org

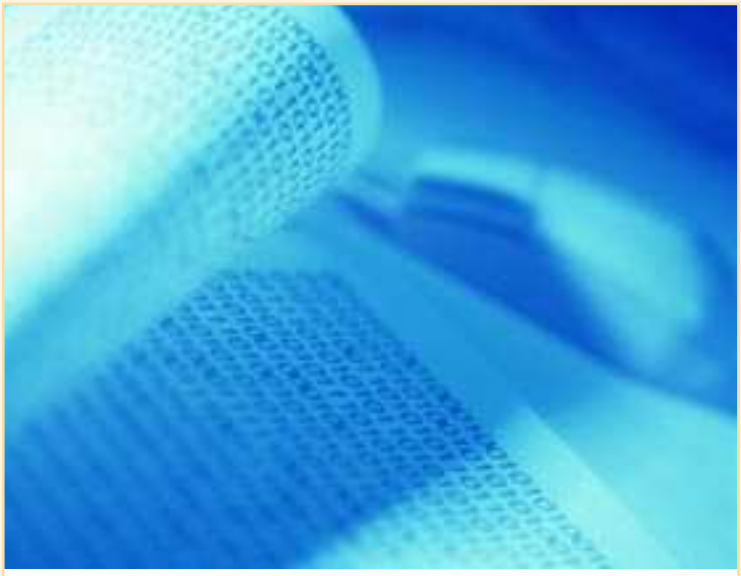
« Par ma foi ! il y a plus de quarante ans que je dis de la prose sans que j'en susse rien. » Dans cette phrase extraite du *Bourgeois gentilhomme,* monsieur Jourdain utilise, sans le savoir non plus, 10 « e », 8 « a » et seulement 3 « q » et 2 « j ». L'ordinateur ne le sait pas plus lorsqu'il lit le fichier numérique permettant d'afficher ce texte. Et pourtant, cette redondance peut s'avérer utile. Comment ?

Avant de se plonger dans l'analyse des lettres, rappelons que, quelle que soit leur nature (texte, image ou son), les fichiers informatiques sont des suites d'octets, c'est-à-dire des groupes de huit bits (0 ou 1). Compresser un fichier signifie le transformer en un autre fichier de plus petite taille. La compression se fait au moyen d'algorithmes. Elle est dite « sans perte » s'il est toujours possible de reconstituer l'original. Que l'on compresse une chanson ou un film, le travail de l'algorithme est le même : il s'agit de réduire la taille des suites d'octets. Un acte devenu banal dans l'usage grand public de l'informatique.

Huit bits = 256 valeurs

Un groupe de huit bits peut prendre $2^8 = 256$ valeurs différentes (toutes les combinaisons de 0 et de 1 comprises entre 00000000 à 11111111) et permet donc de coder 256 caractères suivant, par exemple, le code ASCII (American Standard Code for Information Interchange). Celui-ci faisait initialement appel à sept bits ; il a été étendu à huit bits pour inclure les caractères accentués (qui n'étaient pas prévus à l'origine car le code avait été conçu pour l'anglais).

Comme nous l'avons évoqué, tous les caractères ne sont pas utilisés avec la même fréquence. D'où l'idée de coder les plus fréquents avec moins de bits que les autres. Pour ses premiers Macintosh, la société Apple a ainsi imaginé de crypter les quinze caractères les plus répétés – en français, ce sont les voyelles e, a, i, etc. ainsi que les signes de ponctuation – sur quatre bits. Une seizième configuration des quatre bits (par exemple 1111) est réservée pour signifier : « il ne s'agit pas de l'un des 15 caractères les plus courants. » Le caractère en question est alors codé avec huit autres bits – donc douze en



LXXXXXXXXX XXXXXXE à transmettre est compSequis et iure molore magnim nit, quat, velit del ute tinim nostie ming ex el digna feum doleniamet aute dolore consequam ent lobor sit in xxxxxxxxven 200 s. © MARK WRAGG/IMAGESTATE/EYEDEA.FR

W^xyz

Compression de fichiers

tout. Douze bits au lieu de huit pour les caractères les moins fréquents, quatre au lieu de huit pour les plus fréquents : si ces derniers comptent pour 80 % de l'ensemble des caractères, alors d'un côté on perd 10 % (la moitié de 20 %) et de l'autre, on gagne 40 % (la moitié de 80 %). Le gain moyen est finalement de 30 %.

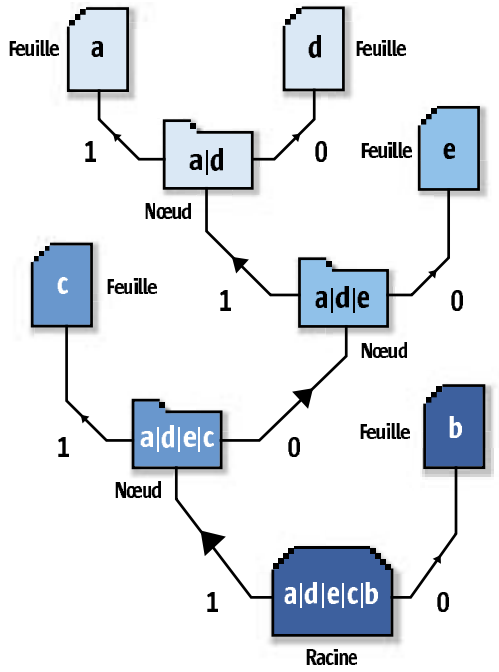
Nœuds père et fils

Peut-on améliorer ce résultat ? Oui, à condition d'accepter que les méthodes soient plus longues à appliquer. L'une des solutions les plus célèbres est due à l'informaticien américain David Huffman. En 1952, alors qu'il était étudiant au Massachusetts Institute of Technology, il eut l'idée de faire varier davantage la longueur des codes en écrivant le caractère le plus fréquent avec un bit, le suivant avec deux et ainsi de suite [1]. Il utilisa des codes dont aucun n'est le préfixe d'un autre, comme 0, 101, 111, 100 et 1101, afin de pouvoir les distinguer même lorsqu'ils sont écrits les uns à la suite des autres. Par exemple, avec le codage ci-dessus, la suite 11111010100 se découpe en 111, 1101, 0, 100.

Comment créer de tels codes ? Imaginons un arbre dont les feuilles seraient des lettres. Cet arbre est un « arbre binaire ». Il s'agit d'une structure de données composée de nœuds : chaque nœud a un père (sauf le premier, que l'on appelle la racine) et deux fils (sauf les derniers, que l'on appelle les feuilles). Le tout se présente comme un arbre généalogique [fig. 1]. En partant de la racine, on atteint chaque lettre en empruntant successivement les branches dans une direction ou dans une autre. On code le trajet par une suite de bits en notant, à chaque étape, 0 ou 1, selon que l'on emprunte la branche de gauche ou celle de droite. Enfin, on attribue à chaque lettre, le code du trajet qui y mène.

Cette façon de procéder nous assure qu'aucun code n'est le préfixe d'un autre, car un chemin vers une feuille ne peut passer par une autre feuille. La longueur d'un code correspond à la « hauteur » de la lettre dans l'arbre. Les feuilles les plus hautes représentent donc les caractères les moins fréquents.

Fig.1 L'algorithme de Huffman



LE TEXTE à transmettre est composé des lettres a, b, c, d et e. Leurs fréquences d'apparition sont respectivement égales à 5 %, 50 %, 20 %, 10 % et 15 %. L'arbre de Huffman se construit en partant du bas et en remontant. Il s'utilise ensuite dans le sens contraire. Les codes des lettres correspondent au trajet pour les atteindre depuis la racine, soit 0 pour *b*, 11 pour *c*, 100 pour *e*, 1010 pour *d* et 1011 pour *a*. © INFOGRAPHIE BRUNO BOURGEOIS

Greffe de lettres

Concrètement, comment procède-t-on pour construire un tel arbre ? Tout d'abord, il faut classer les lettres suivant leur fréquence d'apparition dans le texte. On retire de cette liste les deux qui se révèlent être les moins fréquentes, disons y et z. On greffe ces deux lettres à une racine : elles deviennent respectivement le « fils gauche » et le « fils droit » de ce nœud – la racine dans ce premier cas – et l'on obtient un petit arbre noté y|z. On additionne maintenant la fréquence de y et celle de z. La nouvelle valeur est celle de l'arbre y|z : elle peut être intégrée dans notre classement, l'arbre prenant la place des deux lettres et la racine devenant un nœud. Il ne reste plus qu'à répéter l'opération autant de fois que nécessaire pour obtenir un arbre unique. C'est l'arbre de Huffman du texte.

Limites du codage par octets

Ce codage permet d'obtenir une compression d'environ 40 % en moyenne : personne n'a réussi à faire mieux. Lorsque l'on évoque des compressions de 80 % pour du texte, c'est que l'on a abandonné le codage par octets. En effet, en s'affranchissant de cette contrainte, on peut apporter des améliorations fondées, par exemple, sur la création d'un dictionnaire des mots rencontrés : avec un tel dictionnaire, on peut remplacer un mot par son « adresse » [2]. Cette manière de procéder est surtout efficace pour les textes car, pour les autres types de fichiers, on rencontre moins de longs mots répétitifs. De plus, ces algorithmes améliorés demandent des temps d'exécution importants. Pour un codage et un décodage rapide, l'algorithme de Huffman garde l'avantage.

La démarche décrite ici est une stratégie classique d'optimisation. L'idée sous-jacente consiste à dire que l'on peut arriver à une solution optimale en effectuant un choix optimal à chaque étape. Un tel choix est qualifié de « glouton » car il évoque un goinfre se jetant toujours sur le plus gros morceau sans réfléchir à la suite. En procédant ainsi à chaque étape, l'algorithme peut conduire à une solution optimale – c'est le cas avec la méthode de Huffman –, mais ce n'est pas toujours le cas.

Cet algorithme est très différent de ceux couramment utilisés aujourd'hui pour la compression d'images ou de sons : ceux-ci, jouant sur les faiblesses des sens humains, sont destructifs. C'est le cas des ondelettes, utilisées pour le standard Jpeg2000 [3]. Cependant, on utilise l'algorithme de Huffman après une première compression avec perte pour réduire encore la taille du fichier. ■

[1] D. Huffman, *Proceedings of the I.R.E., 40,* 1098, 1952.

[2] A. Lempel et J. Ziv, *IEEE Transactions on Information Theory, 24,* 530, 1978.

[3] M. Nowak et Y. Meyer, « La surprenante ascension des ondelettes », *La Recherche,* février 2005, p. 56.

POUR EN SAVOIR PLUS

Xavier Marsault, *Compression et cryptage des données multimédia,* Hermès, 1995.

W^xyz
Bases de données

Lorsque l'on cherche une information dans une grande base de données, on recourt à un index. Comment concevoir celui-ci pour trouver le plus rapidement possible ce que l'on cherche ?

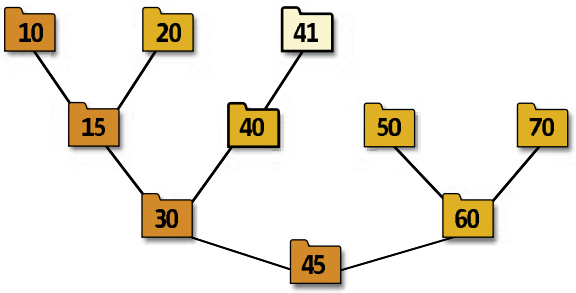
EVGENII LANDIS co-inventeur en 1962 de l'arbre de recherche équilibré. © COURTESY OF LENA LANDIS

Le classement : un arbre à la hauteur

Hervé Lehning,
professeur de mathématiques spéciales au lycée Janson-de-Sailly.
herve.lehning@prepas.org

Vous entrez dans une bibliothèque pour chercher un livre conseillé par un ami. Il vous a donné le titre et le nom de l'auteur. Vous vous renseignez. On vous envoie à deux fichiers : l'un est classé par thème, l'autre par nom d'auteur. Vous choisissez le second et, très vite, vous trouvez votre auteur. À côté, vous lisez la référence du livre que vous cherchez, un numéro de classement dans les rayonnages. Sans le savoir peut-être, vous avez consulté une base de données. Au numéro dit dans les rayonnages, vous trouvez votre livre.

Fig.1 L'AVL, arbre de recherche équilibré



L'ARBRE est composé de branches et de nœuds portant une étiquette. Les éléments classés sont ici des nombres. Si l'on cherche l'élément 40, on le compare à la racine (45) : il est inférieur. Si cet élément existe, il appartient par convention à la branche (le « fils ») gauche. On le compare ensuite à la racine du fils gauche (30). Il est supérieur donc appartient cette fois à son fils droit. On constate qu'il s'agit de la racine du fils droit. Le processus s'arrête. Si on veut maintenant ajouter l'élément 41, on opère de même.

Recherche séquentielle

Les bases de données sont nos meilleurs outils pour trouver rapidement une information ou un objet parmi un grand nombre d'éléments. De façon générale, on peut voir une base de données comme un tableau dont les lignes représentent les données (les livres de notre bibliothèque) et les colonnes les renseignements les concernant (numéro de classement dans la base, nom de l'auteur, thème, etc.). Ce type de formalisme fonctionne aussi bien pour un dictionnaire que pour un fichier de clients d'une entreprise.

Lorsque la base est grande – disons qu'elle comporte un million de fiches, il est difficile d'y trouver ce que l'on cherche si les fiches ne sont pas classées. La seule manière de conduire la recherche est alors de parcourir les fiches une par une, de la première à la dernière. Avec de la chance, la fiche que l'on cherche est la première. Mais elle peut tout aussi bien être la dernière. En moyenne, il faut consulter la moitié des fiches pour trouver la bonne. Une telle recherche est dite séquentielle. Le temps nécessaire pour la mener à bien est proportionnel au nombre d'éléments dans la base. Aussi, même avec l'aide d'un ordinateur, n'est-elle praticable que pour de petites bases de données.

Pour simplifier la consultation, on peut utiliser des index avec un classement suivant un critère tel que le nom de l'auteur. Imaginons que l'on cherche le titre d'un livre dans un million de fiches classées. On prend d'abord la fiche du milieu. La recherche s'arrête si c'est la bonne. Sinon, on poursuit. Si la fiche choisie comporte un nom « antérieur » à celui du titre que nous cherchons, alors notre fiche se trouve entre les fiches 500 001 et 1 000 000 ; sinon elle se situe entre les numéros 1 et 499 999. On poursuit ensuite de la même manière : à chaque étape, la longueur de l'intervalle de recherche est divisée par deux. Dans le pire des cas, la consultation est terminée après 20 tentatives, puisque $2^{20} = 1\,048\,576$.

W^xyz
Bases de données

De façon générale, dans un index ordonné de N fiches, la recherche se fait dans un temps proportionnel au logarithme* de base 2 de N.

Algorithme récursif

Ce type de classement est bien adapté aux bases de données qui n'évoluent pas ou peu. Mais que faire quand la composition de la bibliothèque évolue et que l'on veut ajouter, modifier ou supprimer une fiche ? Prenons le cas d'un ajout (les autres cas sont semblables). Aucune difficulté pour la base elle-même : on place la nouvelle fiche à la suite des autres. Pour les index, c'est plus difficile. Si vous voulez insérer une nouvelle fiche, il faut la placer au bon endroit. Il est facile de le trouver, il suffit de le chercher comme précédemment. Mais, pour insérer la fiche, il faut décaler une par une celles qui la suivent et corriger leur référence (on change le numéro de ligne dans notre tableau). En moyenne, le temps pour faire une insertion (ou une suppression) est donc proportionnel à la taille de la base. Trop long dès que l'on a beaucoup de données.

Pour simplifier la gestion d'une base de données, on fait appel, pour l'index, à une structure de la forme d'un arbre. Celle-ci est composée de nœuds, chacun portant une étiquette sur laquelle figure le critère de recherche. À chaque nœud, l'arbre se divise en deux. L'étiquette du « fils » gauche est antérieure à celle du nœud et celle du « fils » droit est postérieure. On appelle ce type d'arbre un arbre de recherche « équilibré » ou « arbre AVL » en hommage aux deux mathématiciens russes qui l'ont inventé en 1962, Georgii Adelson-Velskii et Evgenii Landis .

La « hauteur » d'un tel arbre, c'est-à-dire le nombre de branches de bas en haut, est de l'ordre de $\log_2 N$ s'il comporte N éléments. En effet, s'il y a 1 élément à la base, 2 au premier niveau, puis 4, 8, etc., alors, pour une hauteur X, le nombre total d'éléments est égal à $2^{X+1} - 1$, donc de l'ordre de 2^X.

Comment procède-t-on pour trouver un nom ? On part de la base de l'arbre : si le nom que l'on cherche est « antérieur » au nom étiqueté sur le nœud, alors il se trouve sur la branche de gauche ; sinon il se trouve sur la branche de droite. La recherche se poursuit ainsi en parcourant les arêtes de l'arbre [fig. 1]. Le temps nécessaire en moyenne est proportionnel à la hauteur de l'arbre, donc de l'ordre de $\log_2 N$. Pas mieux qu'avec une liste triée ; en revanche, il est beaucoup plus facile d'ajouter un élément.

Lorsque l'arbre possède déjà un premier nœud (une « racine »), c'est-à-dire s'il n'est pas vide, on ajoute l'élément à la branche de gauche s'il est « antérieur » à ce nœud, à la branche de droite sinon. Un tel algorithme est dit récursif : on peut le réappliquer autant de fois que nécessaire.

Principe de récurrence

Comment montrer que cet algorithme fonctionne toujours ? Grâce au principe de récurrence. L'algorithme s'applique sans problème pour les arbres de hauteur nulle, puisqu'ils sont vides. Imaginons maintenant que l'on a réussi à montrer qu'il fonctionne pour tous les arbres jusqu'à ceux d'une hauteur égale à H. On considère un arbre de hauteur $H + 1$. De sa racine partent deux arbres. L'algorithme fonctionne sur ces deux arbres car leur hauteur est inférieure ou égale à H. On reprend alors la recette énoncée auparavant : si l'élément à ajouter est antérieur à la racine, il convient de l'ajouter à son fils gauche, sinon à son fils droit. Nous savons le faire dans les deux cas, donc l'algorithme fonctionne pour les arbres de hauteur $H + 1$. Le principe de récurrence permet ainsi d'affirmer que l'algorithme remplit bien son office.

Le temps d'exécution est identique à celui d'une recherche : il est proportionnel à la hauteur de l'arbre, donc de l'ordre de $\log_2 N$. Reste un problème : le nouvel arbre n'est plus forcément équilibré – il peut avoir plus de nœuds d'un côté que de l'autre. Il faut donc veiller à le rééquilibrer. On le fait par « rotations » des parties déséquilibrées, c'est-à-dire en intervertissant les éléments de sorte à avoir un fils de part et d'autre d'un nœud [fig. 2].

Du point de vue théorique, les arbres AVL sont la solution idéale pour structurer les index des bases de données. Mais, dans la pratique, les bases sont stockées sur des disques magnétiques, et donc le temps d'exécution est fortement influencé par l'accès à la mémoire. C'est pourquoi les langages de gestion de bases de données comme SQL *(Structured Query Language)* ou Oracle utilisent plutôt des arbres légèrement différents, appelés arbres B, du nom de leur inventeur, Rudolf Bayer. Ceux-ci, fondés sur les mêmes principes, contiennent plusieurs étiquettes sur chaque nœud, ce qui permet de réduire leur hauteur. La recherche reste active sur ce point car les arbres utilisés doivent toujours être adaptés aux spécificités des nouveaux supports de stockage.

POUR EN SAVOIR PLUS

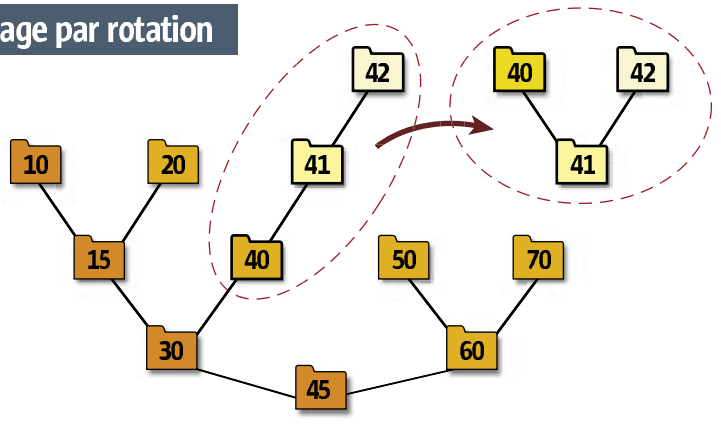
G. M. Adelson-Velskii et E. M. Landis, *Doklady Akademii Nauk SSSR, 146*, 263, 1962.

R. Bayer et E. McCreight, *Acta informatica, 1*, 173, 1972.

G. Gardarin, *Bases de données*, Eyrolles, 2003.

* Le **logarithme** de base 2 de N est le nombre X tel que $2^X = N$. On le note $\log_2 N$.

Fig.2 Rééquilibrage par rotation



L'ARBRE se déséquilibre lorsque l'on ajoute les deux éléments 41 et 42. Pour le rééquilibrer, il suffit d'opérer une rotation sur son élément médian. On obtient ainsi un élément de chaque côté. © INFOGRAPHIES BRUNO BOURGEOIS