Skip lists et arbres binaires de recherche probabilistes

P. Corbineau[1]

1: Laboratoire de Recherche en Informatique
UMR 8623 (CNRS – Université Paris Sud)
PCRI – INRIA Futurs
91405 Orsay CEDEX, France
pierre.corbineau@lri.fr

Résumé

En 1989, William Pugh publia un article intitulé *Skip lists : a probabilistic alternative to binary trees* [9] dans lequel il analysait en détail les performances d'une structure de données mutable appelée *skip list*. Cette structure de données permet de représenter un ensemble fini ordonné à l'aide d'une liste chaînée à niveaux dans laquelle chaque élément est situé à un niveau aléatoire. Dans cet article nous décrivons comment une tentative d'implantation en Objective Caml [5] d'une version fonctionnelle de ces listes à niveaux a permis d'identifier une correspondance forte entre cette structure et une structure d'arbres binaires de recherche probabilistes dont nous décrirons l'implantation et les propriétés.

1. Introduction

L'implantation d'algorithmes efficaces pour la représentation d'ensembles finis est un problème ancien auquel beaucoup de solutions ont été proposées. La plupart d'entre elles repose sur le maintien d'une structure ordonnée permettant de rechercher un élément en temps logarithmique. On peut toutefois citer quelques exceptions : les tables de hachage, qui reposent sur l'association à chaque élément d'une clef rapide à calculer, et des structures plus spécialisées basées sur les arbres de préfixes (*tries*) pour représenter des ensembles de mots sur un alphabet fini, dont les arbres de Patricia ou même les diagrammes de décision binaire (BDD) vus comme des ensembles de mots binaires.

Parmi les structures basées sur des ensembles ordonnés, les variantes autour de la notion d'arbre de recherche, et notamment d'arbres binaires de recherche, sont très nombreuses. Le principal problème, outre l'utilisation massive de la récursivité — peu populaire chez les programmeurs impératifs —, est que le maintien du coût logarithmique de la recherche d'éléments passe par une discipline stricte des opérations d'ajout et de retrait d'éléments afin de préserver un équilibre de l'arbre binaire, et par conséquent une hauteur de l'arbre logarithmique par rapport au nombre de ses nœuds.

Les méthodes les plus populaires pour arriver à ce résultat sont les arbres AVL[1] (utilisés dans la bibliothèque standard d'Objective Caml) et les arbres rouges et noirs [2] (utilisés dans la bibliothèque standard de SML-NJ). On peut aussi citer les *treaps* [11], qui sont des structures hybrides, à la fois arbres binaires de recherche (*trees*) et files de priorité (*heaps*) dont les nœuds contiennent, en plus des éléments de l'ensemble, une priorité aléatoire.

La structure de *skip list*, ou liste à niveaux, décrite par Pugh en 1989 [9], diffère notablement de la notion d'arbre de recherche : elle est présentée comme une structure linéaire contenant plusieurs niveaux où chaque niveau est une liste ordonnée d'éléments qui sont aussi présents dans les listes de niveau inférieur. Il nous a paru pertinent de tenter d'implanter cette structure de façon persistante dans un langage fonctionnel car elle paraissait optimisée pour une implantation mutable dans un langage impératif. Cette démarche nous a permis d'identifier les *skip lists* à une forme probabiliste d'arbres binaires de recherche, ce que nous expliquerons dans la partie 2. Nous décrirons plus en détail l'implantation d'une bibliothèque d'opérations ensemblistes sur ces arbres probabilistes (partie 3). Ensuite nous analyserons les performances obtenues ainsi que la complexité des algorithmes implantés (partie 4). Enfin nous tenterons de donner un aperçu des travaux existant dans ce domaine et de conclure sur la pertinence de l'utilisation de telles structures.

2. Des *skip lists* aux arbres binaires de recherche probabilistes

2.1. Description des *skip lists*

L'idée principale de la structure de *skip list* est d'associer un niveau entier à chaque élément stocké dans la liste, et de construire les listes croissantes d'éléments de niveau supérieur ou égal à un niveau donné : dans l'exemple de la figure 1, les éléments 3, 5, 13 et 34 sont de niveau 0 ; 21 est de niveau 1 ; 2 est de niveau 2 et 8 de niveau 3. Chaque élément est donc stocké avec un tableau contenant les pointeurs vers la suite des niveaux auxquels il appartient.

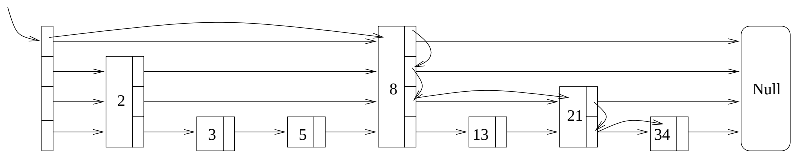

FIG. 1 – Recherche d'un élément dans une *skip list*.

La recherche d'un élément consiste à parcourir les niveaux de la structure en commençant par le plus élevé. Lorsque l'on trouve dans un niveau un élément strictement supérieur à celui que l'on recherche ou le pointeur *Null*, on descend d'un niveau à partir de l'élément précédent sur ce niveau puis on continue la recherche sur ce niveau inférieur jusqu'à ce que l'on trouve l'élément recherché ou que l'on échoue au niveau le plus bas.

L'efficacité de cette structure vient de ce que les éléments de niveau élevé inférieurs à l'élément recherché évitent de faire la comparaison avec des éléments de plus bas niveau qui leurs sont inférieurs. Ce type de recherche est donc similaire à la recherche dichotomique dans un arbre binaire de recherche ou un tableau trié, c'est juste le choix de l'élément pivot qui diffère.

Mais alors, comment choisir le niveau des éléments à insérer pour minimiser le coût (en sauts ou en comparaisons) de la recherche d'un élément ? Dans le cas dégénéré où tous les éléments ont le même niveau, la *skip list* est équivalente à une liste chaînée triée, par conséquent la recherche a une complexité linéaire, de même que l'insertion et la suppression. W. Pugh remarque que si les éléments dont le rang (dans l'ordre de la liste) est multiple de 2^i sont de niveau i (*cf* figure 2), alors la recherche devient logarithmique. Cependant, il est difficile d'écrire des fonctions d'addition et de suppression préservant la structure puisque tous les éléments supérieurs à l'élément ajouté ou supprimé devraient changer de niveau, ce qui nous donnerait une complexité au moins linéaire pour construire la nouvelle structure.

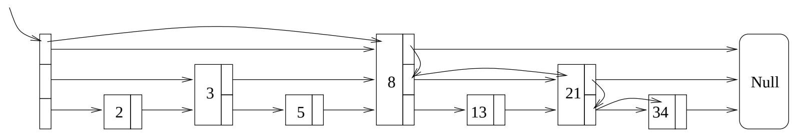

FIG. 2 – Une *skip list* en configuration optimale.

Pour obtenir un coût logarithmique à la fois pour l'addition, la suppression et la recherche, Pugh propose donc de garder le niveau des éléments conservés lors des opérations d'ajout et de suppression, et d'attribuer à chaque nouvel élément un niveau aléatoire, ce qui en fait une structure probabiliste. Ce niveau aléatoire est choisi par un tirage indépendant des opérations précédentes et de la liste dans laquelle l'élément est inséré.

La distribution utilisée pour la variable aléatoire L associée au niveau du nouvel élément est une loi géométrique de paramètre $(1-p)$, c'est-à-dire telle que $\mathrm{Prob}\{L=k\}=(1-p)p^k$, où $p \in [0,1]$ est un paramètre qui va déterminer le rapport moyen entre le nombre d'éléments de niveau $n+1$ et le nombre d'éléments de niveau n. Le choix d'un nombre p trop proche de 1 (fig. 3 en haut) va provoquer la génération de niveaux élevés et imposer un nombre excessif de sauts verticaux (descente d'un niveau) dans la recherche. Toutefois, si p est trop proche de 0, bien que le nombre de sauts verticaux diminue car il y a moins de niveaux, le nombre de sauts horizontaux (à effectuer au sein d'un niveau) va augmenter (fig. 3 en bas).

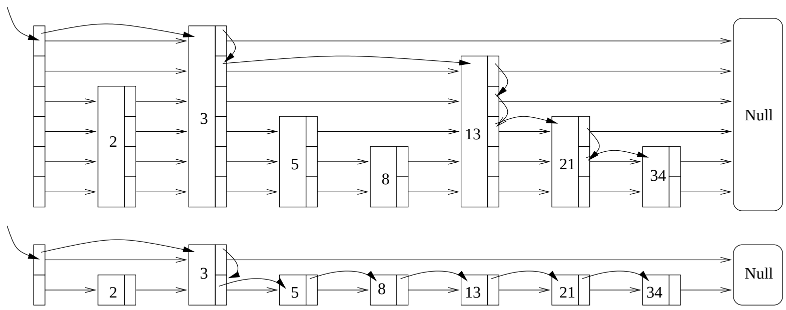

FIG. 3 – *Skip-lists* construites avec p élevé (en haut) ou p faible (en bas).

Les opérations d'ajout et de suppression consistant à rechercher le point d'insertion du nouvel élément ou l'élément à supprimer, et à rétablir les pointeurs modifiés, elles ont un coût en sauts équivalent à la somme du nombre de sauts horizontaux de la recherche et du nombre de niveaux de la liste, et donc équivalent au coût de la recherche d'un élément absent de la structure.

2.2. Adaptation en vue d'une implantation fonctionnelle

Pour réaliser une implantation fonctionnelle sans avoir à reconstruire tous les nœuds à gauche d'une modification, nous avons choisi de supprimer les pointeurs horizontaux menant d'un nœud à un nœud de niveau strictement supérieur. En effet, ces pointeurs ne sont jamais utilisés pour les recherches d'éléments et ne sont pas indispensables pour la concaténation de listes lors de la suppression d'un élément.

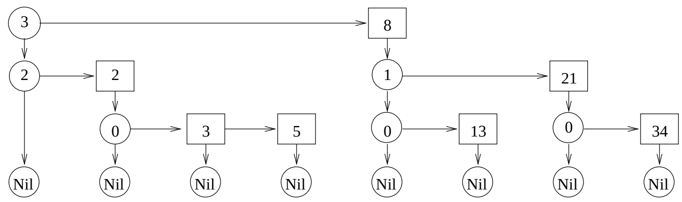

FIG. 4 – Une *skip list* fonctionnelle.

On peut donc représenter une *skip list* d'éléments de type `elt` par le type Caml suivant :

```
type t =
  | Level of int * level * t
  | Nil
and level =
  | Node of elt * level * t
  | Last of elt * t
```

Pour illustrer cette représentation, nous avons dessiné dans la figure 4 la représentation par le type `t` de la *skip list* de la figure 1. Les nœuds ronds sont de type `t` et les nœuds carrés de type `level`. Ce choix d'implantation n'est toutefois pas satisfaisant car les nœuds construits avec le constructeur `Level` (ronds marqués d'un nombre sur la figure 4) ne contiennent pas d'information essentielle et lorsque l'on est sur un nœud lors de la recherche d'éléments, il faut consulter l'élément contenu dans le nœud situé à sa droite pour décider si l'on descend vers un niveau inférieur ou si l'on poursuit la recherche au même niveau (*cf* figure 5).

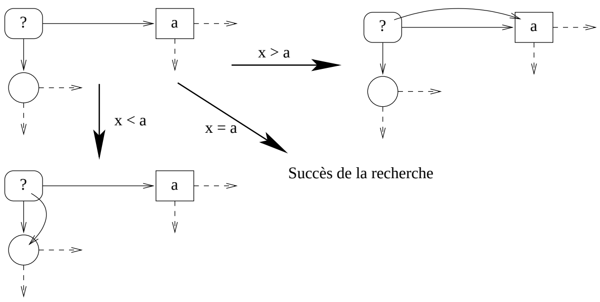

FIG. 5 – Étape élémentaire de recherche de x dans une *skip list* fonctionnelle.

Pour à la fois diminuer le nombre de sauts et éviter de consulter le nœud de droite pour rien, on peut ajouter le marquage de niveau avec chaque élément et déplacer les éléments dans le nœud à gauche de leur position actuelle, là où la décision les concernant est prise. On obtient la structure de la figure 6.

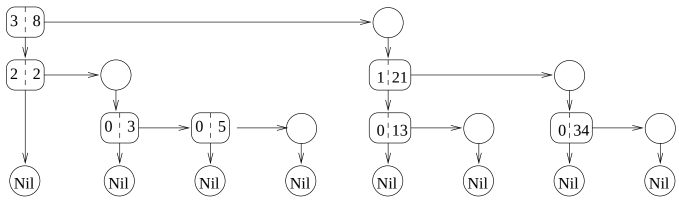

FIG. 6 – Première transformation : on décale les éléments vers la gauche.

Ensuite on va supprimer les nœuds de fin de niveau qui ne servent à rien et on obtient une structure d'arbre homogène (il n'y a plus qu'un type de nœud). Ces arbres ont la propriété d'être des arbres binaires de recherche : l'élément situé à la racine est un majorant des éléments du sous-arbre gauche et un minorant des éléments du sous-arbre droit, et ces deux sous-arbres sont eux aussi des arbres-binaires de recherche (figure 7).

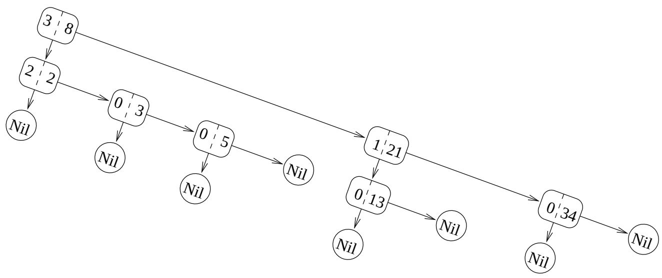

FIG. 7 – Seconde transformation : on supprime les nœuds désormais vides.

Ces arbres sont également des files de priorité par rapport aux niveaux des éléments : tout nœud a un niveau supérieur à celui de ses fils. Ces arbres ressemblent à des *treaps* car on retrouve l'idée d'associer à chaque élément une valeur aléatoire et que chaque valeur soit choisie indépendamment des autres. Cependant, dans le cas des *treaps*, les priorités des éléments sont supposées être toutes distinctes, et suivre une loi uniforme sur un intervalle borné ($[0;1]$ par exemple) ; alors que nos niveaux suivent une loi discrète (la loi géométrique de paramètre $1-p$) et ne sont donc pas bornés, et de plus il n'ont pas besoin d'être tous distincts.

Pour obtenir un isomorphisme entre *skip lists* et arbres probabilistes, il faut ajouter aux arbres la contrainte suivante : le fils gauche d'un nœud a toujours un niveau strictement inférieur à celui de ce nœud (pour les files de priorité, on suppose simplement qu'il est inférieur ou égal). Sous cette contrainte, on a la propriété suivante : soit Φ la bijection entre *skip lists* et arbres probabilistes, les opérations `mem`, `insert` et `remove` de recherche, d'ajout à un niveau fixé et de retrait d'un élément vérifient les propriétés suivantes :

$$\begin{array}{rcl} \mathtt{mem}(x, A) & = & \mathtt{mem}(x, \Phi(A)) \\ \Phi(\mathtt{insert}(l, x, A)) & = & \mathtt{insert}(l, x, \Phi(A)) \\ \Phi(\mathtt{remove}(x, A)) & = & \mathtt{remove}(x, \Phi(A)) \end{array}$$

Dans la suite de cet article nous proposerons une implantation en Objective Caml de la structure obtenue, que nous appellerons arbres binaires de recherche probabilistes ou simplement arbres probabilistes.

3. Arbres binaires de recherche probabilistes

3.1. Représentation des arbres

Pour représenter les arbres probabilistes, nous supposons que nous disposons d'un type `elt` des éléments et d'une fonction de comparaison `compare : elt -> elt -> int`. Cette fonction renvoie 0 si les éléments comparés x et y sont égaux, sinon le signe du nombre renvoyé indique si $x < y$ (résultat strictement négatif) ou $x > y$ (résultat strictement positif). Nous définissons le type Caml suivant pour représenter nos arbres :

```
type t =
  | Empty
  | Node of int * t * elt * t
```

Dans la suite de l'article nous omettrons les feuilles des arbres (constructeur `Empty`) dans les figures. La recherche d'un élément dans un arbre probabiliste s'effectue comme dans tout arbre binaire de recherche : on compare l'élément recherché x avec l'élément a de la racine de l'arbre et si $x = a$, on a trouvé x ; sinon, si $x < a$ on recherche x dans le sous-arbre gauche et si $x > a$ on recherche x dans le sous-arbre droit. On peut implanter la recherche avec la fonction `mem` suivante, en utilisant la récursivité et l'évaluation paresseuse des booléens.

```
let rec mem x = function
  | Empty -> false
  | Node( _, left, a, right) ->
      let cmp = compare x a in
        cmp = 0 || mem x (if cmp < 0 then left else right)
```

Pour donner aux nouveaux éléments un niveau aléatoire qui suive une loi géométrique de paramètre $1-p$, on peut simuler une séquence d'épreuves de Bernoulli à l'aide du générateur pseudo-aléatoire de la bibliothèque standard d'Objective Caml. La solution la plus simple pour ce faire est d'utiliser la fonction récursive suivante :

```
let rec gen () =
  if Random.float 1.0 >= p then 0 else 1 + gen ()
```

Cette fonction termine presque sûrement pourvu que `p` soit strictement inférieur à 1. Toutefois, comme nous le verrons dans la partie 4, il est possible de limiter le niveau des éléments à $\log_{1/p}(N)$ (où N est une constante bornant le nombre d'éléments contenus dans les ensembles manipulés) cette limite n'a alors que peu d'influence sur les performances obtenues.

Pour l'implantation nous avons choisi le paramètre $p = 1/2$. On peut, dans ce cas, utiliser comme niveau le rang du premier bit non nul d'une suite de bits aléatoires. En pratique, au lieu d'utiliser le rang i comme niveau, on peut utiliser 2^i car l'exponentielle est une fonction strictement croissante, et on peut alors imposer $i \leq \log_2(\text{max\_int})$ et donc $2^i \leq \text{max\_int}$ car on manipulera rarement des ensembles de cardinal supérieur à **max_int**[1]. On obtient ainsi une fonction non récursive en utilisant l'identité $2^{\lfloor \log_2(x) \rfloor} = x \,\&\, (-x)$.

```
let gen () =
  let tmp = Random.bits () in
    if tmp = 0 then max_int else tmp land (-tmp)
```

3.2. Addition d'un élément

Pour ajouter un élément x à la racine d'un arbre s, on va diviser l'arbre en deux sous-arbres représentant respectivement $\{a \in s \mid a < x\}$ et $\{a \in s \mid a > x\}$. La fonction `split` réalise cette division ; elle retourne également un booléen indiquant si $x \in s$. On remarquera que `split` préserve le niveau des éléments de s.

```
let rec split x = function
  | Empty -> (Empty,false,Empty)
  | Node (lvl,left,a,right) ->
    let cmp = M.compare x a in
      if cmp = 0 then
        left,true,right
      else if cmp < 0 then
        let l,p,r = split x left in
          l,p,Node(lvl,r,a,right)
      else (* cmp > 0 *)
        let l,p,r = split x right in
          Node(lvl,left,a,l),p,r
```

On peut alors écrire une fonction `insert` qui, étant donné un niveau l, un élément x et un ensemble s, insère l'élément x en dessous de tout élément de niveau strictement supérieur à l, de façon à obtenir un nouvel arbre binaire de recherche. La position d'un élément par rapport à ceux de même niveau peut être calquée sur celles des *skip lists*— un élément est toujours en dessous des éléments de même niveau qui lui sont strictement inférieurs — ou alors on peut le placer à la racine des éléments de niveau l dans ce sous-arbre. C'est ce dernier choix que nous avons fait.

On remarquera que si l'élément x est déjà présent dans l'arbre, l'arbre est inchangé et x conserve son niveau, de même les autres éléments conservent leur niveau quoi qu'il arrive. L'addition d'un élément consiste simplement à l'insérer à un niveau choisi aléatoirement en utilisant `split` pour diviser l'arbre situé au point d'insertion, alors que l'addition dans les arbres équilibrés rouges et noirs ou AVL se fait toujours au niveau des feuilles avant de rééquilibrer l'arbre par des techniques de rotation.

```
let rec insert lvlx x s =
  match set with
    | Empty -> Node(lvlx,Empty,x,Empty)
    | Node(lvla,left,a,right) ->
          if lvlx >= lvla then
            let l,p,r = split x s in
              if p then set else Node(lvlx,l,x,r)
          else (* lvlx < lvla *)
            let cmp = M.compare x a in
              if cmp = 0 then s
              else if cmp < 0 then Node(lvla,insert lvlx x left,a,right)
              else (* cmp > 0 *)   Node(lvla,left,a,insert lvlx x right)
```

[1] `max_int` est le plus grand entier représentable dans le type `int` ($2^{30} - 1$ sur une machine 32 bits)

```
let add x s = insert (gen ()) x s
```

Le fait que les éléments de s conservent leur niveau, et en particulier x s'il est déjà dans s permet de conserver le caractère aléatoire et indépendant des niveaux des éléments de s après addition de x. En particulier il faut éviter d'introduire un biais en utilisant systématiquement le plus élevé des deux niveaux de x (l'ancien et le nouveau), ce qui peut se produire si on oublie de tester la présence de x après l'appel à `split`. Ce problème se produira aussi dans le cas de l'union de deux ensembles comme nous le verrons plus bas.

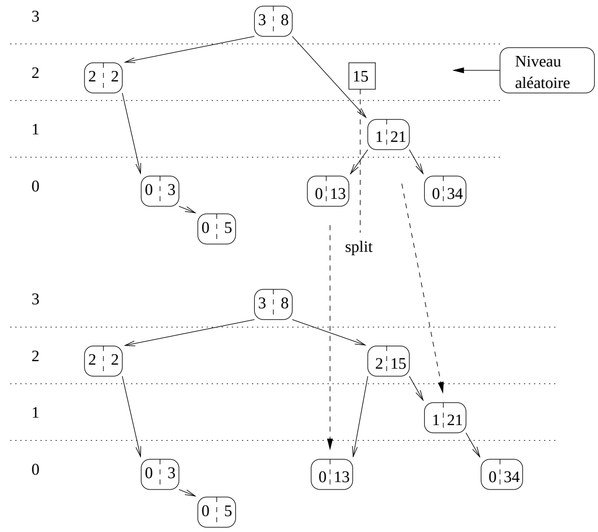

FIG. 8 – Addition d'un élément dans un arbre.

3.3. Suppression d'un élément

Pour supprimer un élément situé à la racine d'un arbre probabiliste, il suffit de former un arbre contenant les éléments du sous-arbre gauche et ceux du sous-arbre droit, avec les mêmes niveaux. C'est ce que fait la fonction `merge`, qui suppose que les éléments de son premier argument sont tous inférieurs aux éléments du second argument.

```
let rec merge lset rset =
  match lset,rset with
    | Empty,  set  -> set
    |  set , Empty -> set
    | Node (llvl,lleft,lx,lright), Node (rlvl,rleft,rx,rright) ->
      if llvl < rlvl then
          Node (rlvl,merge lset rleft,rx,rright)
        else
          Node (llvl,lleft,lx,merge lright rset)
```

Dans le cas où les têtes des deux arbres sont de même niveau, un choix arbitraire s'impose : si on veut respecter le comportement des ***skip lists***, il faut mettre en racine celle de l'arbre de gauche, sinon on peut soit tirer cela au sort, soit alterner en fonction d'un argument booléen supplémentaire. De toute façon notre analyse de complexité ne repose que sur la distribution de probabilité des niveaux et les invariants structurels.

Pour supprimer un élément x qui n'est pas en tête, il suffit d'utiliser `merge` pour fusionner les sous-arbres de celui qui a x pour racine, et de reconstruire l'arbre à l'identique au-dessus de ce nœud. Voici la fonction `remove` qui réalise cette opération :

```
let rec remove x = function
  | Empty -> Empty
  | Node (lvla,left,a,right) ->
      let cmp = M.compare x a in
        if cmp = 0 then
          merge left right
        else if cmp < 0 then
          Node (lvla,remove x left,a,right)
        else
          Node (lvla,left,a,remove x right)
```

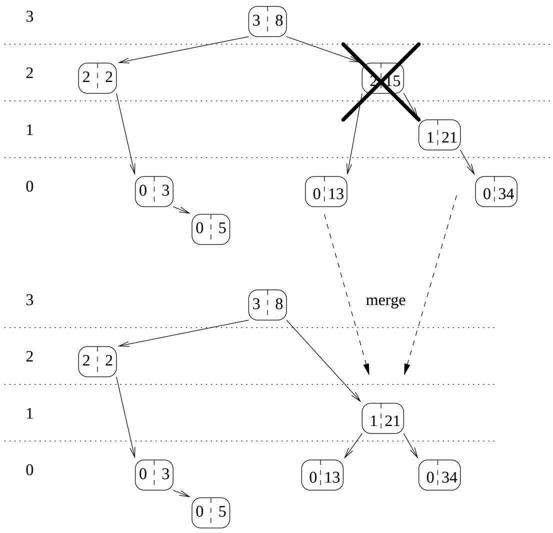

FIG. 9 – Suppression d'un élément dans un arbre.

3.4. Autres opérations ensemblistes

Pour compléter notre bibliothèque de fonctions sur les ensembles finis ordonnés, on souhaite ajouter des opérations binaires telles que union, intersection ou différence. Pour l'union, la difficulté est de bien conserver la distribution de probabilité des niveaux des éléments. La solution que nous avons adoptée est de donner aux éléments de l'union de s_1 et s_2 une nouvelle priorité s'ils sont à la fois dans s_1 et s_2 et leur priorité d'origine sinon. Pour l'intersection et la différence, on choisit simplement la priorité des éléments dans s_1.

Il est difficile d'adapter l'algorithme d'union fourni dans [10] qui procède de la gauche vers la droite de manière destructive. Pour calculer l'union de deux ensembles, on procède donc récursivement du haut vers le bas : on examine les racines x_1 et x_2 des deux ensembles fusionnés s_1 et s_2, et on compare leurs niveaux l_1 et l_2. Si $l_1 \geq l_2$, on utilisera la racine de s_1 comme racine de l'union. On utilise alors `split` pour couper en deux s_2 par rapport à x_1. On applique alors récursivement l'union aux deux paires de sous-arbres gauches et droits et deux cas se présentent : si $x_1 \in s_2$, alors on ajoute l'élément x_1 (avec une nouvelle priorité) à l'un des deux sous-arbres et on fusionne ces arbres avec `merge` ; en revanche si $a_1 \notin s_2$, on se contente de reconstruire un arbre de racine a_1 avec les sous-arbres obtenus. Si $l_1 < l_2$, on procède de façon symétrique.

```
let rec union s1 s2 =
  match s1,s2 with
    | Empty, set -> set
    | set, Empty -> set
```

```
  | Node (lvl1,left1,x1,right1), Node (lvl2,left2,x2,right2) ->
      if lvl1 >= lvl2 then
        let l2,p2,r2 = split x1 s2 in
          if p2 then merge (union left1 l2) (add x1 (union right1 r2))
          else Node(lvl1,union left1 l2,x1,union right1 r2)
      else (* lvl1 < lvl2 *)
        let l1,p1,r1 = split x2 s1 in
          if p1 then (merge (union l1 left2) (add x2 (union r1 right2)))
          else Node(lvl2,union l1 left2,x2,union r1 right2)
```

L'intersection et la différence procèdent de façon plus asymétrique : on divise s_2 suivant la racine x_1 de s_1 et on calcule récursivement l'intersection des deux sous-arbres obtenus avec les deux sous-arbres de s_1. Si $x_1 \in s_2$, alors on l'ajoute au sommet des deux arbres obtenus, sinon on utilise `merge` pour fusionner les arbres obtenus.

```
let rec inter s1 s2 =
  match s1,s2 with
  | Empty ,   _   -> Empty
  |   _   , Empty -> Empty
  | Node (lvl1,left1,x1,right1), s2 ->
        let l2,p2,r2 = split x1 s2 in
          if p2 then
            Node(lvl1,inter left1 l2,x1,inter right1 r2)
          else
            merge (inter left1 l2) (inter right1 r2)
```

Pour calculer la différence entre deux ensembles il suffit d'échanger les cas où $a_1 \in s_2$ et $a_1 \notin s_2$. Nous avons complété notre bibliothèque de façon à obtenir la même signature que la bibliothèque `Set` d'Objective Caml. Notre implantation est disponible à l'adresse suivante : `http://www.lri.fr/~corbinea/ftp/programs/randomtree.ml`.

4. Analyse des performances

4.1. Analyse probabiliste

Pour analyser les performances des arbres probabilistes, nous allons avoir besoin de quelques notions simples de probabilités. Les paramètres que nous allons manipuler (niveaux des éléments, longueurs des chemins) seront des variables aléatoires discrètes.

Définition 1 ($=_{\text{prob}}$, $\leq_{\text{prob}}$) *Soient X et Y deux variables aléatoires à valeurs dans $\mathbb{N}$, on écrit $X =_{\text{prob}} Y$ (respectivement $X \leq_{\text{prob}} Y$) si et seulement si $\forall k \in \mathbb{N}, \text{Prob}\{X \geq k\} = \text{Prob}\{Y \geq k\}$ (respectivement si et seulement si $\forall k \in \mathbb{N}, \text{Prob}\{X \geq k\} \leq \text{Prob}\{Y \geq k\}$).*

Définition 2 (loi géométrique, loi binomiale, loi binomiale négative) *Soit X une variable aléatoire à valeurs dans $\mathbb{N}$:*

- *On dit que X suit la **loi géométrique** de paramètre p, notée $\mathcal{G}(p)$, si et seulement si $\forall k \in \mathbb{N}, \text{Prob}\{X = k\} = p(1-p)^k$. Cette distribution représente le nombre d'échecs avant le premier succès dans une suite d'expériences de Bernoulli indépendantes avec une probabilité p de succès.*
- *On dit que X suit la **loi binomiale** de paramètres n et p, notée $\mathcal{B}(n,p)$, si et seulement si $\forall k \in \mathbb{N}, \text{Prob}\{X = k\} = C_n^k p^k (1-p)^{n-k}$. Cette distribution représente le nombre de succès dans une suite de n expériences de Bernoulli indépendantes avec une probabilité p de succès.*
- *On dit que X suit la **loi binomiale négative** de paramètres n et p, notée $\mathcal{J}(n,p)$, si et seulement si $\forall k \in \mathbb{N}, \text{Prob}\{X = k\} = C_{k+n-1}^k p^n (1-p)^{k-n}$. Cette distribution représente le nombre d'échecs précédant le n-ième succès dans une suite de n expériences de Bernoulli indépendantes avec une probabilité p de succès. En particulier, on a $\mathcal{G}(p) =_{\text{prob}} \mathcal{J}(1,p)$.*

4.2. Étude de la complexité de la recherche d'un élément

Pour analyser le coût (en comparaisons ou en sauts de pointeurs) de la recherche d'un élément on va suivre le chemin de la recherche à l'envers, on va donc considérer la longueur du chemin conduisant d'un nœud de niveau à son plus petit ancêtre de niveau au moins k, dans un arbre supposé infini. Cette quantité sera une variable aléatoire notée L_k.

Lemme 1 $L_k =_{\text{prob}} \mathcal{J}(k,p)$

Démonstration : Sachant qu'un nœud est de niveau l, le niveau l' de son père suit la loi $l+\mathcal{G}(1-p)$. Grâce à l'indépendance des niveaux de tous les nœuds, on peut décomposer le chemin comme suit : on commence avec 0 comme niveau courant, et la longueur du chemin est alors 0. À chaque étape de notre raisonnement, on fait une expérience de Bernoulli de probabilité $1-p$.

- Si elle réussit, le nœud suivant dans notre chemin ascendant est au niveau courant, on incrémente la longueur du chemin.
- Si elle échoue, le prochain nœud est à un niveau supérieur, on incrémente le niveau courant.

Quand le niveau courant devient égal à k, on arrête et on décompte le nombre de nœuds rencontrés qui est L_k. Si l'on a rencontré aucun nœud, c'est que le nœud de départ était déjà de niveau supérieur ou égal à k. Pour aller jusqu'au niveau k, il a fallu k échecs de notre expérience de Bernoulli, et la longueur du chemin est le nombre de succès avant ces k échecs, ce qui, en inversant succès et échecs et p et $1-p$, permet de dire que $L_k =_{\text{prob}} \mathcal{J}(k,p)$. □

Avec ce résultat, on peut donner une borne probabiliste de la profondeur d'un élément d'un arbre probabiliste de n éléments. En effet, on peut diviser le chemin d'un élément à la racine en deux parties indépendantes : celle située en dessous du niveau $\lceil \log_{1/p}(n) \rceil$ et celle située au-dessus.

D'abord, la longueur d'un chemin d'un noeud donné jusqu'au au niveau k dans un arbre fini est probabilistiquement inférieure à L_k, car en particulier elle est bornée par le nombre d'éléments de l'arbre. Ensuite, la longueur du chemin situé au-dessus du niveau $\lceil \log_{1/p}(n) \rceil$ est bornée par le nombre total de nœuds ayant un niveau supérieur ou égal à $\log_{1/p}(n)$ dans l'arbre considéré. Soit L la variable aléatoire associée au niveau d'un élément de l'arbre, on a $\text{Prob}\{L >= k\} = p^k$ donc $\text{Prob}\{L >= \log_{1/p}(n)\} = \text{Prob}\{L >= \lceil \log_{1/p}(n) \rceil\} \leq 1/n$. Le nombre de nœuds de niveau supérieur ou égal à $\log_{1/p}(n)$ est alors borné probabilistiquement par la loi binomiale $\mathcal{B}(n, 1/n)$, donc la hauteur H d'un élément dans un arbre de n éléments vérifie $H_n \leq_{\text{prob}} \mathcal{J}(\lceil \log_{1/p}(n) \rceil, p) + \mathcal{B}(n, 1/n)$. On en déduit que l'espérance de H_n vérifie

$$\begin{array}{rcl} E(H_n) & \leq & E\left(\mathcal{J}(\lceil \log_{1/p}(n) \rceil, p) + \mathcal{B}(n, \frac{1}{n})\right) \\ & \leq & E\left(\mathcal{J}(\lceil \log_{1/p}(n) \rceil, p)\right) + E\left(\mathcal{B}(n, \frac{1}{n})\right) \\ & \leq & \frac{1-p}{p} \lceil \log_{1/p}(n) \rceil + 1 \end{array}$$

On en conclut que la recherche d'un élément dans un arbre probabiliste est en $O(\log(n))$ (borne probabiliste) par rapport au nombre d'éléments de l'arbre. Les opérations d'ajout et de retrait, dont la complexité est aussi proportionnelle à la hauteur des éléments, sont donc également de complexité logarithmique.

La borne de $\log_{1/p}(N)$ proposée pour les niveaux d'un arbre d'au plus N éléments est justifiée par le fait que pour un arbre de n éléments l'espérance du nombre d'éléments dont le niveau dépasse la borne est $E(\mathcal{B}(n, \frac{1}{N})) = \frac{n}{N}$: cette borne n'a donc que peu d'influence sur l'arbre obtenu tant que $n \leq N$.

4.3. Étude expérimentale

Pour obtenir des résultats expérimentaux, nous avons passé notre bibliothèque d'arbres probabilistes au banc d'essai avec les arbres AVL d'Objective Caml et des arbres rouges et noirs implantés par J.-C. Filliâtre. On en trouvera les résultats dans le tableau 4.3. Ces essais sont peu concluants car on remarque que dans presque tous les cas nos arbres sont moins efficaces d'environ 20 à 30% que les arbres AVL ou rouges et noirs, que ce soit en temps de calcul, en nombre de comparaisons on en nombre d'allocations.

Toutefois, l'insertion séquentielle d'éléments est plus rapide pour nos arbres, sans doute grâce au faible nombre d'allocations de nouveaux nœuds. En effet l'insertion séquentielle dans les arbres

	arbres AVL			arbres rouges et noirs			arbres probabilistes		
n	t (s)	nb. comp.	nb. alloc.	t (s)	nb. comp.	nb. alloc.	t (s)	nb. comp.	nb. alloc.
addition séquentielle de n éléments									
1000	0.00	9966	11955	0.01	11925	14904	0.00	5606	5924
10000	0.03	133602	153587	0.03	168244	198221	0.01	72827	76165
100000	0.26	1668911	1868893	0.33	2183976	2483948	0.19	763010	796296
1000000	3.11	19951404	21951383	3.85	26836444	29836411	2.01	9176624	9510198
addition aléatoire de n éléments (peu de répétitions)									
1000	0.01	8075	8530	0.00	8264	9421	0.01	9519	9241
10000	0.04	115787	120397	0.04	116432	127350	0.05	146492	143692
100000	0.86	1501377	1547818	0.76	1509165	1618887	1.04	1911178	1883387
1000000	14.73	18374823	18836853	12.47	18540087	19633526	18.04	24714554	24436348
addition aléatoire de n éléments (beaucoup de répétitions)									
1000	0.00	2857	1879	0.00	2955	1981	0.00	3370	2381
10000	0.01	60045	50271	0.01	60570	50897	0.01	74509	64630
100000	0.22	936964	839262	0.21	922062	825323	0.23	1152329	1053475
1000000	4.21	12772284	11795391	3.99	12814697	11847839	4.87	16567439	15578801
Recherche aléatoire dans ensemble séquentiel de taille n									
1000000	1.65	12362936	0	1.64	12469391	0	2.23	17630558	0
Recherche aléatoire dans ensemble aléatoire dense de taille n									
1000000	2.15	12488612	0	2.13	12516497	0	2.66	16000684	0
Recherche aléatoire dans ensemble aléatoire peu dense de taille n									
1000000	2.42	13781678	0	2.31	13841077	0	2.93	17518116	0
suppression séquentielle dans ensemble séquentiel									
100	0.02	102800	95000	0.02	120100	141400	0.01	138142	128142
1000	0.11	898500	823200	0.20	1200100	1400600	0.16	1394371	1294371
10000	1.04	7973400	7222300	1.80	11070500	13067100	1.44	12134907	11134907
suppression aléatoire dans ensemble aléatoire dense									
100	0.00	50681	50147	0.01	50415	58334	0.01	62663	61744
1000	0.15	841268	840537	0.17	839386	922740	0.17	1084990	1082894
10000	4.45	11807453	11805300	5.13	11794427	12632675	5.45	15414805	15407452
suppression aléatoire dans ensemble aléatoire peu dense									
100	0.01	69380	69366	0.01	68964	69077	0.01	83544	83531
1000	0.23	1036996	1037013	0.26	1033610	1035508	0.27	1289641	1289622
10000	6.65	13772008	13771948	6.63	13830271	13849314	7.90	17592835	17592319
10000 réunions d'ensembles de taille n									
100	0.32	1696106	1783797	0.23*	853255*	861513*	0.40	1855601	2472833
1000	4.38	17605079	18753138	4.94*	8610623*	8617211*	6.53	19566439	26147141
5000	45.71	88530302	94435267	86.40*	43099035*	43107713*	60.10	98474917	131746365
10000 intersections d'ensembles de taille n									
100	0.31	1709814	1889373	0.18*	853255*	400663*	0.27	1439383	1577424
1000	4.10	17783584	19983200	3.13*	8610623*	4015389*	3.53	15103565	16730685
5000	36.23	89383770	100657142	47.00*	43099035*	20079687*	33.30	76136985	84453495
10000 différences d'ensembles de taille n									
100	0.34	1660487	1622067	0.20*	853255*	230425*	0.26	1444019	1335504
1000	3.65	17291540	17404693	2.69*	8610623*	2300911*	3.02	15104281	14329345
5000	29.09	86941946	87853743	39.71*	43099035*	11514013*	25.00	76220337	72655077

TAB. 1 – Comparaison expérimentale des arbres probabilistes avec d'autres arbres.

*Les résultats des opérations binaires sur les arbres rouges et noirs sont peu significatifs étant donné que ces opérations sont implantées par traduction en listes chaînées et opérations sur ces listes équilibrés produit toujours un déséquilibre du même côté que l'on doit compenser par un coûteux processus de rééquilibrage.

5. Conclusion et travaux liés

Pour mieux comprendre ce que sont les *skip lists*, on pourra lire [9] et [10] qui décrivent en détail les algorithmes utilisés. Des recherches plus poussées sur l'analyse probabiliste des *skip lists*, d'un point de vue mathématique, on été conduites par Kirschenhofer et Prodinger dans [3]. L'article [7] de Xavier Messeguer propose un isomorphisme entre *skip lists* et une variante de B-arbres, ainsi que des méthodes pour l'accès concurrent à ces arbres. En ce qui concerne les arbres probabilistes, on trouvera dans [11] la description des *treaps* et de leurs propriétés. Dans [6], on trouvera une autre forme d'arbres probabilistes où chaque élément d'un arbre a une probabilité égale d'être à la racine de cet arbre.

Nous avons proposé une implantation originale d'arbres probabilistes persistants qui évite l'inconvénient majeur des *treaps* : nos arbres peuvent avoir des nœuds ayant le même niveau tandis que les nœuds des *treaps* ont des niveaux tous distincts et uniformément répartis (ce modèle n'est pas incrémental). En pratique, cela conduit à devoir choisir les niveaux dans un ensemble fini. Quand on tire au hasard une priorité déjà existante, il faut alors reconstruire l'arbre avec de nouvelles priorités, opération complexe que l'on souhaite voir amortir par le nombre d'opérations nécéssaires pour reproduire ce genre de configuration. Or, comme le fait remarquer Chris Okasaki dans [8], ce qui dans le cas mutable serait un coût amorti par le nombre d'opérations peut devenir critique pour des structures de données persistantes car on peut répéter de façon arbitraire des opérations plus coûteuses qu'à l'ordinaire sur la même structure.

L'adaptation des *skip lists* pour les rendre persistantes conduit à une structure radicalement différente bien qu'isomorphe aux *skip lists*. Ceci est dû aux contraintes fortes liées à la persistance : si on modifie un nœud dans une structure, tous les nœuds qui pointent vers celui-ci doivent être reconstruits ; cela oblige à minimiser les dépendances dans la structure, ce que l'on fait en utilisant une structure d'arbre. La structure d'arbre obtenue s'implante de façon très naturelle, c'est peut-être une des implantations les plus simples des arbres binaires de recherche, hormis l'implantation naïve qui est inefficace. Bien que les résultats expérimentaux soient moins bons que ceux des arbres explicitement équilibrés (AVL ou rouges et noirs), ils ne sont pas catastrophiques si l'on tient compte de la simplicité du code mis en œuvre. De plus, l'approche probabiliste permet d'éviter les cas dégénérés où les autres algorithmes sont systématiquement moins performants (par exemple l'insertion séquentielle) : à moins que l'utilisateur puisse prévoir les niveaux que l'on va choisir, il lui sera impossible de trouver une séquence d'opérations mettant en défaut à coup sûr nos algorithmes. De plus, la plus grande efficacité des opérations d'intersection et de différence ensembliste peuvent faire des arbres probabilistes un choix possible pour des applications où ces opérations sont fréquentes.

Références

[1] G. M. Adel'son-Vel'skii and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3 :1259–1262, 1962.

[2] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *19th IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[3] P. Kirschenhofer and H. Prodinger. The path length of random skip lists. *Acta informatica*, 31(9) :775–792, 1994.

[4] D. E. Knuth. *The Art of Computer Programming : Sorting and Searching*, volume 3. Addison-Wesley, Reading, MA, 1973.

[5] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.08, Documentation and user's manual*, 2004. `http://caml.inria.fr/ocaml/htmlman/`.

[6] C. Martinez and S. Roura. Randomized binary search trees. Technical Report LSI97-8-R, Universitat Politecnica de Catalunya, 1997.

[7] X. Messeguer. Skip trees, an alternative data structure to skip lists in a concurrent approach. *Informatique Théorique et Applications*, 31(3) :251–269, 1997.

[8] C. Okasaki. *Purely functionnal data structures.* Cambridge University Press, 1999.

[9] W. Pugh. Skip lists : A probabilistic alternative to balanced trees. In *Workshop on Algorithms and Data Structures*, pages 437–449, 1989.

[10] W. Pugh. A skip list cookbook. Technical Report CS-TR-2286.1, Institute for Advanced Computer Studies, University of Maryland, 1990.

[11] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5) :464–497, 1996.