Chapitre 10
Arbres binaires de recherche

On a étudié au chapitre précédent le problème de la recherche dans une collection d'éléments ordonnés entre eux : on a montré que lorsque n éléments sont représentés par une liste contiguë triée, on peut toujours effectuer une recherche en $\Theta(\log n)$ comparaisons. Mais la représentation contiguë est mal adaptée lorsque la collection évolue : une adjonction ou une suppression peuvent nécessiter $\Theta(n)$ opérations.

Lorsque la collection des éléments n'est pas figée, il paraît donc nécessaire d'utiliser une représentation chaînée, bien que cela prenne de la place supplémentaire en mémoire. Si l'on utilise une liste chaînée (cf. chapitre 5), l'adjonction d'un élément peut être effectuée en un nombre constant d'opérations, mais c'est la recherche et la suppression d'un élément qui demandent trop de temps ($\Theta(n)$ comparaisons au pire).

Quand on veut que les trois opérations – recherche, adjonction, suppression – soient effectuées avec la même efficacité, les représentations les mieux adaptées sont les structures arborescentes. Les méthodes de recherche arborescentes reposent toutes sur le même principe : la comparaison avec le (ou les) élément(s) d'un nœud de l'arbre permet d'orienter la suite de la recherche dans l'arbre. La structure fondamentale est celle d'*arbre binaire de recherche.*

1. Définition et recherche

On peut représenter une collection ordonnée de n éléments par un arbre étiqueté ayant n nœuds (chapitre 7). Chaque nœud contient un élément et la répartition des éléments dans l'arbre permet de guider la recherche en faisant des comparaisons.

1.1. Arbre binaire de recherche

Définition : Un ***arbre binaire de recherche*** est un arbre binaire étiqueté tel que pour tout nœud ν de l'arbre :

- les éléments de tous les nœuds du sous-arbre gauche de ν sont inférieurs ou égaux à l'élément contenu dans ν,
- et les éléments de tous les nœuds du sous-arbre droit de ν sont supérieurs à l'élément contenu dans ν.

Il résulte immédiatement de cette définition que le *parcours symétrique* d'un arbre binaire de recherche produit la suite des éléments *triée en ordre croissant.*

Sur la figure 1.a on a dessiné un arbre binaire de recherche qui représente l'ensemble $E = \{a, d, e, g, i, l, q, t\}$ muni de l'ordre alphabétique. Notons qu'il y a plusieurs représentations possibles d'un même ensemble par un arbre binaire de recherche : l'arbre de la figure 1.b représente aussi E.

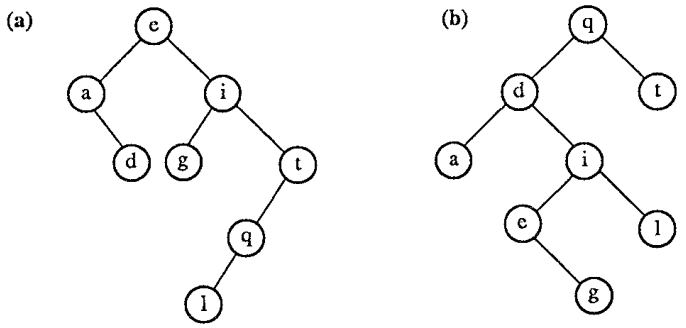

Figure 1. Arbres Binaires de Recherche.

1.2. Recherche d'un élément

Pour rechercher (une occurrence) d'un élément dans un arbre binaire de recherche, on compare cet élément au contenu de la racine :

- s'il y a égalité, on a trouvé l'élément et la recherche est terminée (*recherche positive*),
- si l'élément est plus petit (resp. plus grand) que celui de la racine, on poursuit la recherche dans le sous-arbre gauche (resp. droit); si ce sous-arbre est vide, il y a échec : l'élément n'appartient pas à l'arbre binaire de recherche initial *(recherche négative)*.

Rappelons que si $B =< o,\ G,\ D >$ est un arbre étiqueté dont la racine contient l'élément r, on note (cf. chapitre 7) $B =< r,\ G,\ D >$.

La spécification de l'opération de recherche d'un élément (que l'on suppose de sorte Elément) dans un arbre binaire de recherche s'écrit :

rechercher : Elément $\times$ Arbre $\rightarrow$ Booléen

Si G et D sont des variables de sorte Arbre, x et r des variables de sorte Elément, on a les axiomes suivants :

axiomes

rechercher(x, *arbre-vide*) = faux
$x = r \Rightarrow$ *rechercher*(x, $< r$, G, $D >$) = vrai
$x < r \Rightarrow$ *rechercher*(x, $< r$, G, $D >$) = *rechercher*(x, G)
$x > r \Rightarrow$ *rechercher*(x, $< r$, G, $D >$) = *rechercher*(x, D)

Une version récursive de l'algorithme de recherche est donnée dans la fonction *chercher*, qui utilise le type Pascal ARBRE.

```
type ARBRE  = ↑nœud;
       nœud = record
                val : Elément;
                g, d : ARBRE
              end;

function chercher(X : Elément; A : ARBRE) : boolean;
{cette fonction récursive recherche si un élément X appartient à un arbre binaire de recherche A; le résultat est true si X appartient à A, et false sinon}
begin
  if A = nil then chercher := false
  else if A↑.val = X then chercher := true
    else if X < A↑.val then chercher := chercher (X, A↑.g)
      else chercher := chercher(X, A↑.d)
end chercher;
```

Cet algorithme a une forme semblable à celle de l'algorithme de recherche dichotomique; la récursion est terminale, mais la fonction renvoie un résultat. La transformation en une version itérative est laissée en exercice.

2. Adjonction d'un élément

L'adjonction d'un élément dans une collection quelle qu'en soit la représentation, se fait toujours selon le principe suivant : d'abord déterminer la place où l'on doit faire l'adjonction, puis réaliser l'adjonction à cette place. Dans un arbre binaire de recherche, l'opération d'adjonction se décompose en une étape de recherche, qui renvoie des informations sur la place où doit être inséré le nouvel élément, suivie de l'adjonction proprement dite.

2.1. Adjonction aux feuilles

Pour ajouter un élément dans un arbre binaire de recherche, on le compare au contenu de la racine pour déterminer s'il faut l'ajouter dans le sous-arbre gauche

ou le sous-arbre droit, et l'on rappelle la procédure récursivement. Le dernier appel récursif se fait sur un arbre vide et l'on crée alors à cette place le nœud contenant l'élément à ajouter. Le nouveau nœud devient donc une feuille de l'arbre binaire de recherche. Décrivons la spécification de l'opération d'adjonction aux feuilles d'un arbre binaire de recherche :

ajouter-feuille : Elément × Arbre → Arbre

Si G et D sont des variables de sorte Arbre, x et r des variables de sorte Elément, on a les axiomes suivants :

axiomes

ajouter-feuille(x, *arbre-vide*) = < x, *arbre-vide*, *arbre-vide* >

$x \leq r \Rightarrow$ *ajouter-feuille*(x, < r, G, D >) =< r, *ajouter-feuille*(x, G), D >

$x > r \Rightarrow$ *ajouter-feuille*(x, < r, G, D >) =< r, G, *ajouter-feuille*(x, D) >

2.1.1. Procédures d'adjonction

Les procédures *ajouterfeuille* et *ajouterfeuilleiter* sont une version récursive et une version itérative de l'adjonction aux feuilles dans un arbre binaire de recherche. Dans le programme récursif, c'est le passage par référence du paramètre A qui permet d'affecter à un de ses champs un pointeur vers le nœud créé. Dans une version itérative de l'algorithme, l'affectation de ce lien doit être explicite. La procédure *ajouterfeuilleiter* résoud le problème en testant avant chaque progression dans l'arbre si le nœud suivant sera une feuille. (Une autre façon de faire, laissée en exercice, consiste à conserver à chaque étape de la progression dans l'arbre des informations concernant le nœud précédent.)

```
procedure ajouterfeuille (X : Elément; var A : ARBRE);
{procédure récursive d'adjonction d'un élément X aux feuilles d'un arbre
binaire de recherche A}
begin
   if A = nil then begin new(A); A↑.val := X; A↑.g := nil; A↑.d :=nil end
               else if X ≤ A↑.val then ajouterfeuille(X, A↑.g)
                                  else ajouterfeuille(X, A↑.d)
end ajouterfeuille;


procedure ajouterfeuilleiter(X : Elément; var A : ARBRE);
{procédure itérative d'adjonction d'un élément X aux feuilles d'un arbre
binaire de recherche A}
var B, Aux : ARBRE;
    poursuite : boolean;
begin
    new(B); B↑.val := X; B↑.g := nil; B↑.d := nil;
    {création du nœud contenant l'élément à ajouter}
```

```
if A = nil then A := B
else begin
    Aux := A; poursuite := true;
    while poursuite do
        if X ≤ Aux↑.val then
        {on teste avant chaque progression dans l'arbre si le nœud
        suivant sera une feuille}
          if Aux↑.g = nil then begin
              Aux↑.g := B;
              poursuite := false
          end
          else Aux := Aux↑.g {on descend à gauche}
        else if Aux↑.d = nil then begin
              Aux↑.d := B;
              poursuite := false
          end
          else Aux := Aux↑.d {on descend à droite}
end
end ajouterfeuilleiter;
```

2.1.2. *Construction d'un arbre binaire de recherche par adjonctions successives aux feuilles*

On peut construire un arbre binaire de recherche par adjonctions successives à partir de la donnée de la suite de ses éléments : par exemple, l'adjonction successive des éléments e, i, a, t, d, g, q, l conduit aux arbres binaires de recherche dessinés sur

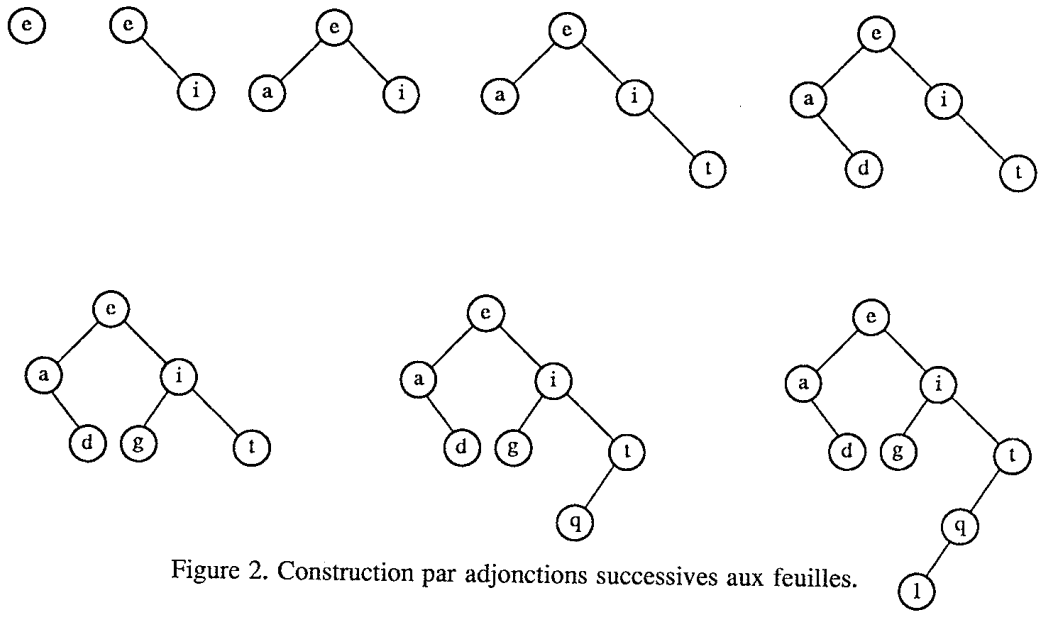

Figure 2. Construction par adjonctions successives aux feuilles.

la figure 2, et l'arbre résultant de toutes les adjonctions est celui de la figure 1.a. D'autres façons d'ordonner la suite des éléments en entrée peuvent aussi conduire au même arbre binaire de recherche résultat, par exemple, l'adjonction successive des éléments dans l'ordre e, a, d, i, g, t, q, l conduit au même arbre. On étudie en exercice le nombre de façons d'ordonner la suite des éléments en entrée qui produisent le même arbre binaire de recherche.

2.2. Adjonction à la racine

On peut ajouter un élément, non seulement aux feuilles, mais à n'importe quel niveau d'un arbre binaire de recherche, et en particulier à la racine (ceci est à rapprocher des méthodes de recherche séquentielle autoadaptative : l'adjonction à la racine peut présenter des avantages, par exemple dans le cas où c'est juste après l'adjonction d'un élément que l'on effectue le plus de recherches le concernant).

Pour ajouter un élément X à la racine d'un arbre binaire de recherche, il faut d'abord couper l'arbre binaire de recherche initial en deux arbres binaires de recherche G et D contenant respectivement tous les éléments inférieurs ou égaux à X, et tous les éléments supérieurs à X (figure 3), puis former l'arbre dont la racine contient X, et qui a pour sous-arbre gauche (resp. droit) l'arbre binaire de recherche G (resp. D). On obtient bien un arbre binaire de recherche.

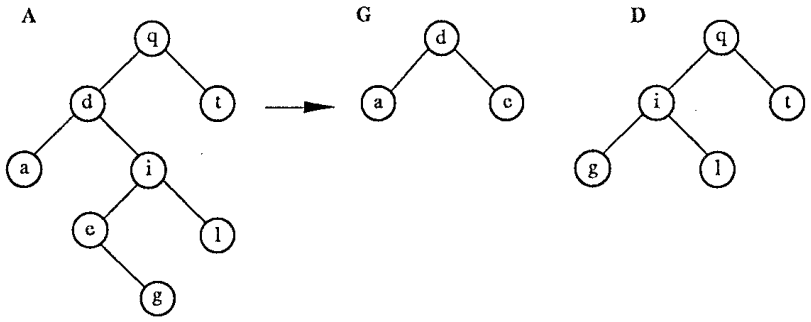

Figure 3. Coupure selon l'élément f.

C'est évidemment l'étape de *coupure* qui est la plus délicate; il est important de remarquer qu'il n'est pas nécessaire de parcourir tous les nœuds de l'arbre initial A pour former G et D. Ce sont les nœuds situés sur le chemin C suivi lors de la recherche de X qui déterminent la coupure : en effet, si un nœud de C contient un élément plus petit (resp. plus grand) que X, il vient se placer sur le bord droit de G (resp. bord gauche de D), et par la propriété d'arbre binaire de recherche, il entraîne avec lui dans G (resp. D) tout son sous-arbre gauche (resp. droit).

La spécification de l'opération d'adjonction à la racine dans un arbre binaire de recherche se décrit comme suit :

ajouter-rac : Elément × Arbre → Arbre,

Si G et D sont des variables de sorte Arbre, x et r des variables de sorte Elément, on a les axiomes suivants :

axiomes

ajouter-rac$(x,$ *arbre-vide*$) = <x,$ *arbre-vide*, *arbre-vide*$>$
racine(*ajouter-rac*$(x, <r, G, D>)) = x$
$x < r \Rightarrow$ *g*(*ajouter-rac*$(x, <r, G, D>)) =$ *g*(*ajouter-rac*$(x, G))$
d(*ajouter-rac*$(x, <r, G, D>)) = <r,$ *d*(*ajouter-rac*$(x, G)), D>$
$x \geq r \Rightarrow$ *g*(*ajouter-rac*$(x, <r, G, D>)) = <r, G,$ *g*(*ajouter-rac*$(x, D))>$
d(*ajouter-rac*$(x, <r, G, D>)) =$ *d*(*ajouter-rac*$(x, D))$

2.2.1. Procédure de coupure

La procédure *couper* réalise la coupure de l'arbre binaire de recherche A selon l'élément X ; le résultat est un couple (G, D) d'arbres binaires de recherche : G contient tous les éléments de A plus petits que X (au sens large) et D contient tous les élément de A plus grands que X. Le passage par référence des paramètres G et D permet, comme dans la procédure *ajouterfeuille*, de «raccrocher» les sous-arbres. De plus il faut passer le paramètre A par référence, car l'arbre d'origine est modifié du fait de la manipulation des pointeurs.

```
procedure couper(X : Elément; var A, G, D : ARBRE);
{procédure récursive de coupure de l'arbre binaire de recherche A, selon
l'élément X, en deux arbres binaires de recherche G et D ; G contient
tous les éléments de A inférieurs ou égaux à X et D tous les élément de A
strictement supérieurs à X}
begin
    if A = nil then begin G := nil; D := nil end
            else if X < A↑.val then begin
                                    D := A; couper(X, A↑.g, G, D↑.g)
                                    end
                                    else begin
                                    G := A; couper(X, A↑.d, G↑.d, D)
                                    end
end couper;
```

Notons que dans le cas où X est dans l'arbre A, le chemin C de coupure selon l'élément X dans la procédure *couper* est le chemin de recherche de X suivi du bord gauche du sous-arbre droit du (premier) nœud contenant X. On peut améliorer cette procédure en arrêtant la récursion dès que l'on rencontre l'élément de coupure (cf. exercices).

D'autre part, la procédure *couper* étant récursive terminale, elle peut être dérécursifiée sans pile. L'écriture de la version itérative de la coupure est renvoyée en exercice (attention au passage par référence des paramètres G et D).

2.2.2. *Justification de l'algorithme*

Il est intéressant de prouver la correction de l'algorithme de coupure, en particulier dans le cas où l'élément X est déjà dans l'arbre A.

Tout d'abord l'algorithme termine toujours : à chaque appel récursif, on descend d'un niveau dans l'arbre, et l'on arrive donc au bout d'un nombre fini d'appels (puisque la hauteur de l'arbre est finie) sur un arbre vide, où l'on s'arrête.

Ensuite raisonnons par récurrence sur la hauteur des arbres, pour montrer que l'algorithme construit un arbre binaire de recherche G qui contient tous les éléments de A inférieurs ou égaux à X et un arbre binaire de recherche D qui contient tous les éléments de A strictement supérieurs à X :

- si A est vide, alors G et D sont vides, et la propriété est vraie,
- sinon, notons r le contenu de la racine de A.

Supposons que $X < r$.

L'algorithme de coupure sur $g(A)$ construit deux arbres binaires de recherche G_0 et D_0. Montrons que si la propriété est vraie pour G_0 et D_0, elle est aussi vraie pour les arbres $G = G_0$ et $D =< r, D_0, d(A) >$ construits à cette étape.

Par hypothèse de récurrence, G_0 (resp. D_0) contient tous les éléments de $g(A)$ inférieurs ou égaux à X (resp. strictement supérieurs à X).

Puisque $G = G_0$ et que $X < r$, G est un arbre binaire de recherche qui contient tous les éléments de A inférieurs ou égaux à X.

Montrons maintenant que $D =< r, D_0, d(A) >$ est bien un arbre binaire de recherche dont tous les éléments sont strictement supérieurs à X.

Le sous-arbre gauche D_0 de D provient de $g(A)$; ses éléments sont donc inférieurs ou égaux à r. Le sous-arbre droit de D est $d(A)$, donc ses éléments sont strictement supérieurs à r. Ainsi, D est un arbre binaire de recherche. De plus il contient tous les éléments de A strictement supérieurs à X, à savoir r, les éléments de $d(A)$, et tous les éléments de $g(A)$ strictement supérieurs à X.

Le cas où $X \geq r$ se traite de façon analogue.

2.2.3. *Procédure d'adjonction à la racine*

La procédure *ajouterrac*, d'adjonction d'un élément X à la racine d'un arbre binaire de recherche A, utilise la procédure de coupure de l'arbre binaire de recherche selon cet élément.

```
procedure ajouterrac(X : Elément; var A : ARBRE);
{cet algorithme réalise l'adjonction de l'élément X à la racine de l'arbre binaire de recherche A ; il utilise la procédure de coupure d'un arbre binaire de recherche}
var R : ARBRE;
begin
    new(R); R↑.val := X;
    couper(X, A, R↑.g, R↑.d);
    A := R
end ajouterrac;
```

3. Suppression d'un élément

Pour supprimer un élément d'une collection, il faut tout d'abord déterminer sa place, puis effectuer la suppression proprement dite, qui s'accompagne éventuellement d'une réorganisation des éléments.

Dans un arbre binaire de recherche, la suppression de l'élément commence par sa recherche, et la suite dépend de la place de l'élément à supprimer dans l'arbre.

• Si c'est *un nœud sans fils*, la suppression est immédiate (c'est le cas, par exemple, pour le nœud contenant g, noté simplement g, dans l'arbre binaire de recherche de la figure 1.a).

• Si c'est un *nœud qui a un seul fils*, il suffit de le remplacer par ce fils, et l'arbre résultant reste un arbre binaire de recherche (par exemple, la suppression de t dans l'arbre de la figure 1.a donne l'arbre binaire de recherche de la figure 4.a).

• Si c'est un *nœud qui a deux fils*, il y a deux solutions pour conserver la structure d'arbre binaire de recherche (éléments en ordre croissant dans la lecture symétrique), tout en modifiant le moins possible l'arbre binaire initial :

- soit remplacer le nœud contenant l'élément à supprimer par celui qui lui est *immédiatement inférieur* (qui contient le plus grand élément de son sous-arbre gauche) : le remplacement de e par d dans l'arbre de la figure 4.a donne l'arbre de la figure 4.b,
- soit remplacer le nœud contenant l'élément à supprimer par celui qui lui est *immédiatement supérieur* (qui contient le plus petit élément de son sous-arbre-droit) : le remplacement de i par l dans l'arbre de la figure 4.a donne l'arbre de la figure 4.c.

Ces deux solutions sont équivalentes lorsque tous les éléments de l'arbre binaire de recherche sont distincts; (le lecteur est invité à étudier le cas où l'arbre binaire de recherche contient des éléments égaux). On a choisi ici de décrire l'algorithme de suppression en utilisant la première solution.

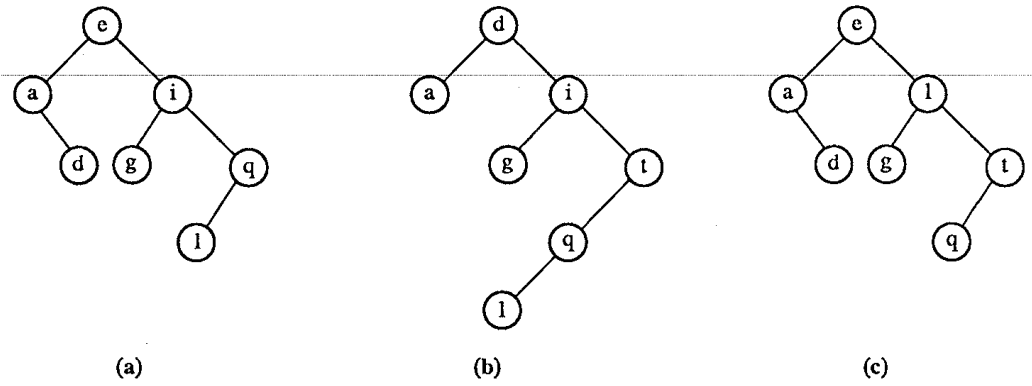

Figure 4. Suppressions dans un arbre binaire de recherche.

3.1. Spécification de la suppression

L'opération de suppression d'un élément dans un arbre binaire de recherche utilise l'opération *max*, qui renvoie l'élément maximal d'un arbre binaire de recherche, et l'opération *dmax* qui renvoie l'arbre privé de cet élément.

supprimer : Elément × Arbre → Arbre
max : Arbre → Elément
dmax : Arbre → Arbre

Si G et D sont des variables de sorte Arbre, x et r des variables de sorte Elément, on a les préconditions et axiomes suivants :

préconditions

max(G) **est-défini-ssi** $G \neq$ *arbre-vide*
dmax(G) **est-défini-ssi** $G \neq$ *arbre-vide*

axiomes

supprimer(x , *arbre-vide*) = *arbre-vide*
$x = r \Rightarrow$ *supprimer*(x, $< r, G,$ *arbre-vide*$>$) = G
$x = r$ & $D \neq$ *arbre-vide*
$\Rightarrow$ *supprimer*(x, $< r,$ *arbre-vide*, $D >$) = D
$x = r$ & $G \neq$ *arbre-vide* & $D \neq$ *arbre-vide*
$\Rightarrow$ *supprimer*($x, < r, G, D >$) = <*max*(G), *dmax*(G), $D >$
$x < r \Rightarrow$ *supprimer*($x, < r, G, D >$) = $< r,$ *supprimer*(x, G), $D >$
$x > r \Rightarrow$ *supprimer*($x, < r, G, D >$) = $< r, G,$ *supprimer*(x, D) $>$

max (< *r*, *G*, *arbre-vide*>) = *r*
dmax (< *r*, *G*, *arbre-vide*>) = *G*
D ≠ *arbre-vide* ⇒ *max*(< *r*, *G*, *D* >) = *max*(*D*)
D ≠ *arbre-vide* ⇒ *dmax*(< *r*, *G*, *D* >) =< *r*, *G*, *dmax*(*D*) >

3.2. Algorithme de suppression

La procédure *supprimerabr* utilise la procédure *supmax* de suppression du maximum dans un arbre binaire de recherche.

```
procedure supmax(var MAX : Elément; var A : ARBRE);
{cet algorithme supprime le nœud contenant l'élément maximal dans un
arbre binaire de recherche A non vide ; il donne pour résultats l'élément
maximal, et l'arbre binaire de recherche A privé du nœud contenant cet
élément}
begin
   if A↑.d =nil then begin MAX := A↑.val; A := A↑.g end
               else supmax(MAX, A↑.d)
end supmax;

procedure supprimerabr(X : Elément; var A : ARBRE);
{cet algorithme supprime un nœud contenant X dans l'arbre binaire de
recherche A ; il donne pour résultat A si X n'est pas dans A, et sinon
A privé de X ; il utilise la procédure de suppression du maximum dans
un arbre binaire de recherche}
var MAX : Elément;
begin
   if A<>nil then
      if X < A↑.val then supprimerabr(X, A↑.g)
      else if X > A↑.val then supprimerabr(X, A↑.d)
         else {A↑.val = X}
            if A↑.g =nil then A := A↑.d
            else if A↑.d =nil then A := A↑.g
               else {A↑.d <> nil et A↑.g <> nil}
                  begin
                  supmax(MAX, A↑.g);
                  A↑.val := MAX
                  end
end supprimerabr;
```

4. Analyse du nombre de comparaisons

On analyse maintenant les différents algorithmes présentés dans le paragraphe précédent, dans le cas où *tous les éléments sont distincts.* On montre que leur

complexité en temps est logarithmique en moyenne, alors qu'elle est linéaire dans le pire des cas.

Les algorithmes de recherche, adjonction et suppression dans un arbre binaire de recherche opèrent par comparaisons : la comparaison entre deux éléments est l'opération fondamentale pour mesurer la complexité en temps de ces algorithmes. Or, le nombre de comparaisons faites par chacun des algorithmes peut être lu directement sur l'arbre binaire de recherche.

Pour l'algorithme de recherche :

- dans le cas d'une recherche positive terminée sur le nœud ν, le nombre de comparaisons est $2\,prof(\nu)+1$, où $prof(\nu)$ est la profondeur du nœud ν dans l'arbre (cf. chapitre 7 pour la définition de la profondeur d'un nœud dans un arbre) ;
- dans le cas d'une recherche négative terminée après le nœud ν, le nombre de comparaisons est $2(prof(\nu)+1)$.

Dans l'algorithme d'adjonction aux feuilles, il faut diviser par deux le nombre de comparaisons pour une recherche négative (il y a un test par nœud au lieu de deux).

Pour une adjonction à la racine, l'algorithme compare l'élément à insérer uniquement aux contenus des nœuds situés sur son chemin de recherche dans l'arbre ; le nombre de comparaisons effectuées est donc exactement le même que pour l'algorithme d'adjonction aux feuilles.

Pour la suppression, le nombre de comparaisons entre éléments est le même que pour la recherche de l'élément à supprimer dans l'arbre.

Remarquons que si l'on veut aussi tenir compte du temps de recherche de l'élément qui remplace l'élément supprimé, il faut ajouter la complexité de la procédure *supmax* ; dans cette procédure, le *test d'arbre vide* est l'opération fondamentale. (Dans le cas de la recherche et de l'adjonction, le nombre de tests d'arbre vide n'est pas significatif, car il est égal au nombre de comparaisons d'éléments.)

L'analyse des algorithmes de recherche, adjonction et suppression, se ramène donc à l'étude de la profondeur des nœuds dans les arbres binaires de recherche.

4.1. Cas extrêmes d'arbres binaires de recherche

On a vu au chapitre 7 que la profondeur et la profondeur moyenne d'un arbre binaire de taille n sont des quantités toujours comprises entre n et $\log_2 n$.

Dans un arbre binaire de recherche *dégénéré*, la recherche d'un élément nécessite en moyenne et dans le cas le pire, un nombre de comparaisons *linéaire* par rapport à la taille de l'arbre : par exemple, si l'on construit un arbre binaire de recherche par adjonctions successives, en insérant les éléments par ordre croissant, on obtient

un arbre dégénéré comme celui de la figure 5.a dans lequel la recherche du $n^{ième}$ élément inséré nécessite $2n - 1$ comparaisons, et la recherche d'un élément quelconque de l'arbre nécessite en moyenne un nombre de comparaisons égal à

$$\frac{1}{n} \sum_{1 \leq i \leq n} (2i - 1) = n$$

On est dans la même situation si l'on insère les éléments dans un «ordre» mal choisi, par exemple dans l'ordre f, a, e, b, d, c, comme à la figure 5.b.

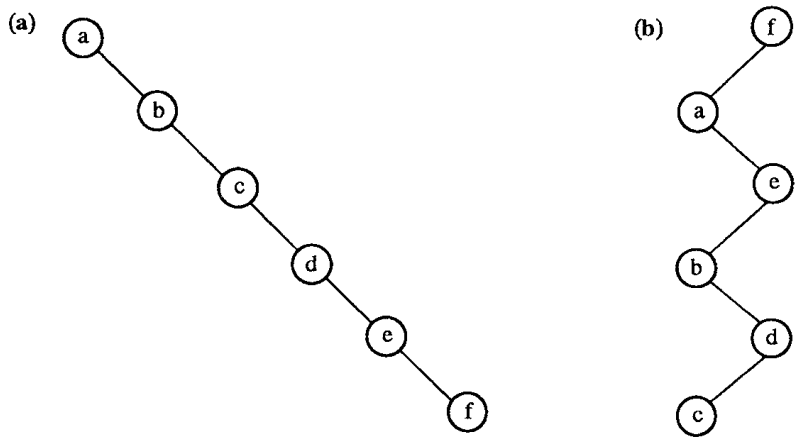

Figure 5. Arbres binaires de recherche dégénérés.

En revanche, si l'arbre binaire de recherche est «bien équilibré» (toutes les feuilles sont situées sur deux niveaux seulement), la recherche d'un élément est, en moyenne et dans le pire des cas, logarithmique par rapport à la taille de l'arbre (cf. chapitre 9, analyse de la procédure *dicho*).

4.2. Cas moyen d'arbres binaires de recherche

On montre ici que le nombre de comparaisons pour la recherche, l'adjonction ou la suppression d'un élément dans un arbre binaire de recherche de taille n est en moyenne de l'ordre de $\log n$. Il faut pour cela calculer la profondeur moyenne d'un nœud dans un arbre binaire de recherche quelconque. On s'intéresse donc à une double moyenne : moyenne sur tous les nœuds d'un arbre, qui est un arbre moyen parmi tous les arbres binaires de recherche.

4.2.1. Notion de moyenne

Les arbres binaires de recherche sont-ils en moyenne plutôt dégénérés ou plutôt équilibrés ? Il faut ici s'arrêter sur la notion de *moyenne* sur l'ensemble des arbres binaires de recherche. On considère que les arbres binaires de recherche sont obtenus uniquement par adjonctions successives aux feuilles de n éléments distincts, et

on fait l'hypothèse que les $n!$ ordres d'adjonction possibles de ces éléments sont équiprobables.

A chacun de ces ordres correspond un arbre binaire de recherche. Des ordres différents peuvent donner des arbres binaires de recherche identiques. Par exemple, les adjonctions successives de 2, 3, 1 ou de 2, 1, 3 donnent l'unique arbre de la figure 6.

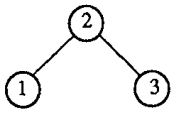

Figure 6

On a donc un ensemble avec répétitions de $n!$ arbres binaires de recherche qui sont équiprobables. On appelle ABR_n cet ensemble. Par exemple, pour $n = 3$, on a les arbres de la figure 7.

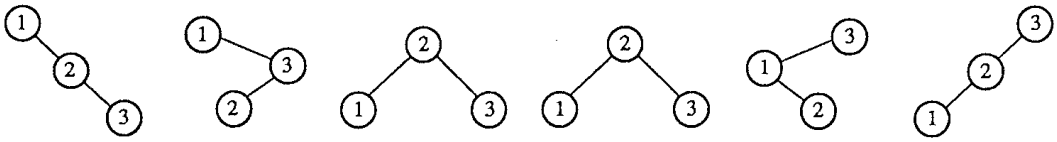

Figure 7

L'analyse qui suit montre qu'un arbre binaire de recherche est en général bien équilibré : la moyenne des profondeurs moyennes, interne et externe, sur les arbres de ABR_n est en $\Theta(\log n)$.

4.2.2. Complétion locale d'un arbre binaire de recherche

Revenons à la notion de construction d'un arbre binaire de recherche par adjonctions successives aux feuilles. Dans un arbre binaire T contenant n éléments, il y a $n+1$ possibilités d'adjonction aux feuilles : on peut matérialiser ces $n+1$ possibilités en complétant l'arbre T par des feuilles carrées , de manière à ce que tout nœud de T ait deux fils. La figure 8 montre un arbre binaire de recherche T et son complété T_c (cf. chapitre 7 pour la définition de la complétion locale d'un arbre binaire, et les propriétés associées). L'arbre T_c a n nœuds internes et $n+1$ feuilles carrées.

Si T est un arbre binaire de recherche, chaque feuille carrée du complété T_c (à l'exception de la première et de la dernière) est située, en lecture symétrique, entre deux nœuds internes n_i et n_j, et représente l'endroit où se termine la recherche négative d'un élément compris entre les contenus de n_i et de n_j. La première (resp. dernière) feuille marque l'endroit où se termine la recherche négative de tout élément inférieur (resp. supérieur) à tous les éléments de l'arbre T.

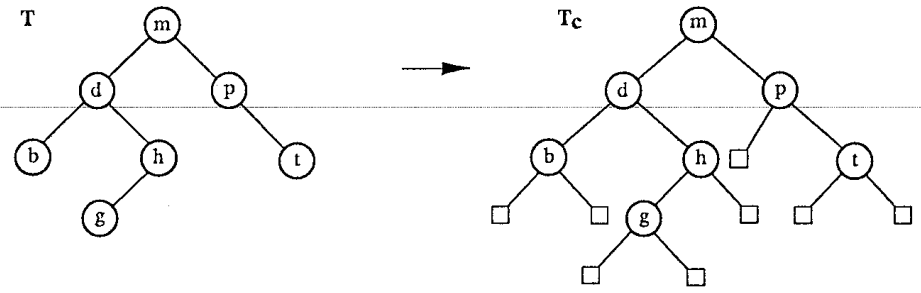

Figure 8. Complétion locale d'un arbre binaire de recherche.

4.2.3. Profondeur moyenne de recherche dans les arbres de ABR_n

Par définition, les arbres binaires de recherche contenus dans ABR_n sont tous obtenus par adjonction aux feuilles d'un élément dans un arbre binaire de recherche de $n-1$ éléments, lui-même obtenu uniquement par des adjonctions aux feuilles. Tous les ordres d'adjonction étant équiprobables, ce dernier élément peut, avec une même probabilité $1/n$, être le plus petit, le $p^{ième}$ plus petit ($p = 2, \ldots, n-1$), ou le plus grand. L'adjonction peut donc se faire en chacune des n places possibles avec une probabilité $1/n$.

Soit T un arbre de ABR_n et T_c son arbre complété. L'arbre T_c a $n+1$ feuilles. On définit la profondeur moyenne d'adjonction (ou de recherche négative) comme la profondeur moyenne externe de T_c : $PE(T_c) = \frac{1}{n+1} \cdot LCE(T_c)$.

On considère à présent PE_n, la profondeur moyenne d'adjonction (ou de recherche négative) dans un arbre de ABR_n, les $n!$ arbres binaires de recherche étant équiprobables :

$$PE_n = \frac{1}{n!} \sum_{T \in ABR_n} PE(T_c) = \frac{1}{n!} \sum_{T \in ABR_n} \frac{1}{n+1} \cdot LCE(T_c).$$

L'étude de $\sum_{T \in ABR_n} LCE(T_c)$ va permettre d'établir une relation de récurrence sur PE_n.

Soit T' un arbre binaire de recherche de $(n-1)$ éléments, et soit f une feuille de son complété T'_c sur laquelle est faite une adjonction. Soit T le résultat de l'adjonction (figure 9) :

$$\begin{aligned} LCE(T_c) &= LCE(T'_c) - prof(f) + 2(prof(f) + 1) \\ &= LCE(T'_c) + prof(f) + 2 \end{aligned}$$

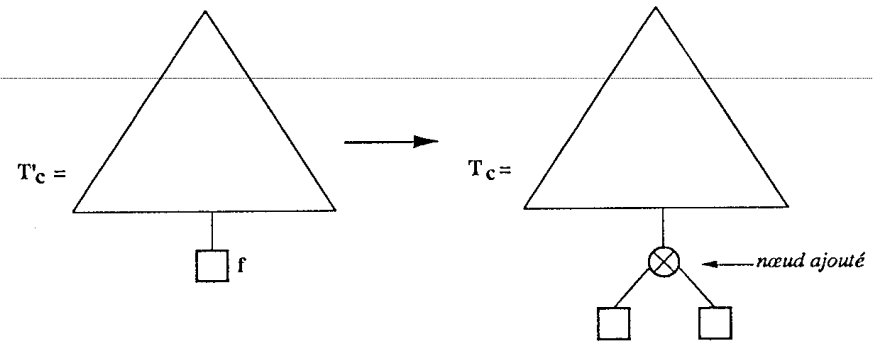

Figure 9

Par définition, chaque arbre de ABR_n provient d'un arbre de ABR_{n-1} dans lequel on a fait une adjonction aux feuilles, d'où

$$\sum_{T \in ABR_n} LCE(T_c) = \sum_{T' \in ABR_{n-1}} \left(\sum_{f \text{ feuille de } T'_c} (LCE(T'_c) + prof(f) + 2) \right)$$

or

$$\sum_{T' \in ABR_{n-1}} \left(\sum_{f \text{ feuille de } T'_c} LCE(T'_c) \right) = n \cdot \sum_{T' \in ABR_{n-1}} LCE(T'_c)$$

et

$$\sum_{T' \in ABR_{n-1}} \left(\sum_{f \text{ feuille de } T'_c} prof(f) \right) = \sum_{T' \in ABR_{n-1}} LCE(T'_c)$$

d'où

$$\sum_{T \in ABR_n} LCE(T_c) = 2n \cdot (n-1)! + (n+1) \sum_{T' \in ABR_{n-1}} LCE(T'_c)$$

En utilisant la définition de PE_n, $PE_n = \frac{1}{n!} \sum_{T \in ABR_n} \frac{1}{n+1} \cdot LCE(T_c)$, on obtient :

$$PE_n = \left(\frac{1}{n!} \sum_{T' \in ABR_{n-1}} LCE(T'_c) \right) + \frac{2}{n+1}, \text{ c'est-à-dire } PE_n = PE_{n-1} + \frac{2}{n+1}.$$

Cette relation de récurrence, avec la condition initiale $PE_1 = 1$ (pour l'arbre binaire de recherche contenant un seul élément), admet pour solution :

$$PE_n = 2H_{n+1} - 2,$$

où $H_{n+1} = \sum_{1 \leq i \leq n+1} \frac{1}{i}$ est le $(n+1)^{ième}$ nombre harmonique.

La profondeur moyenne PI_n d'un nœud interne dans un arbre de ABR_n, c'est-à-dire la *profondeur moyenne d'une recherche positive*, se calcule de manière semblable :

$$PI_n = \frac{1}{n!} \sum_{T \in ABR_n} \frac{1}{n} \cdot LCI(T_c)$$

or, puisque T_c est localement complet, $LCI(T_c) = LCE(T_c) - 2n$, d'où

$$PI_n = \left(1 + \frac{1}{n}\right) PE_n - 2 = 2\left(1 + \frac{1}{n}\right) H_{n+1} - \frac{2}{n} - 4 = 2\left(1 + \frac{1}{n}\right) H_n - 4$$

La proposition suivante résume les résultats obtenus.

Proposition 1 : Dans un arbre binaire de recherche obtenu par adjonctions successives de n éléments distincts, les $n!$ permutations possibles des éléments étant équiprobables, les nœuds externes et internes sont en moyenne à une profondeur de l'ordre de $\log n$. L'expression des profondeurs moyennes externes et internes est plus précisément :

$$PE_n = 2H_{n+1} - 2 \quad \text{et} \quad PI_n = 2\left(1 + \frac{1}{n}\right) H_n - 4\,.$$

4.3. Conclusion

On a donc montré que la complexité des algorithmes de recherche, adjonction et suppression dans un arbre binaire de recherche est en moyenne logarithmique, alors qu'elle est linéaire dans le cas le pire où l'arbre est dégénéré.

Notons $\mathrm{Max}_{\text{rech+}}(n)$ (resp. $\mathrm{Max}_{\text{rech-}}$, $\mathrm{Max}_{\text{adj}}$, $\mathrm{Max}_{\text{supp}}$) le nombre maximum de comparaisons pour une recherche positive (resp. pour une recherche négative, pour une adjonction, pour une suppression) dans un arbre binaire de recherche contenant n éléments.

La hauteur de l'arbre dégénéré étant $n - 1$, ces quatre quantités sont en $\Theta(n)$:

$\mathrm{Max}_{\text{rech+}}(n) = \mathrm{Max}_{\text{supp}}(n) = 2n - 1$
$\mathrm{Max}_{\text{rech-}}(n) = 2n$
$\mathrm{Max}_{\text{adj}}(n) = n.$

Le facteur multiplicatif deux pour la recherche et la suppression vient du fait qu'il y a dans ces deux algorithmes deux tests par nœud, et il n'y a qu'un test par nœud dans les algorithmes d'adjonction.

Notons aussi $\mathrm{Moy}_{\text{rech+}}(n)$ (resp. $\mathrm{Moy}_{\text{rech-}}$, $\mathrm{Moy}_{\text{adj}}$, $\mathrm{Moy}_{\text{supp}}$) le nombre moyen de comparaisons pour une recherche positive (resp. pour une recherche négative,

pour une adjonction, pour une suppression) dans un arbre binaire de recherche quelconque, *i.e.* obtenu par adjonctions successives de n éléments distincts, les $n!$ ordres d'adjonction étant équiprobables.

D'après la proposition 1 :

$$\text{Moy}_{\text{rech+}}(n) = \text{Moy}_{\text{supp}}(n) = 2PI_n + 1$$
$$\text{Moy}_{\text{rech-}}(n) = 2PE_n$$
$$\text{Moy}_{\text{adj}}(n) = PE_n.$$

Ces quatre quantités sont donc en $\Theta(\log n)$.

Les arbres binaires de recherche permettent de rechercher, ajouter et supprimer un élément d'une collection avec une complexité qui est en moyenne logarithmique. Cependant, la complexité au pire est linéaire. Le chapitre suivant présente des structures arborescentes pour lesquelles la complexité de ces opérations reste logarithmique dans le cas le pire.