# Les arbres binaires de recherche équilibrés (sources principaux)

Stéphane Glondu

## Table des matières



## 1 Implémentation naïve

```
1  (*****************************************************************
2   * ARBRES NAIFS
3   * par Stéphane GLONDU (Avril 2004)
4   *****************************************************************)
5
6  (** Définition des types ***************************************)
7
8  type clef = int
9
10 type 'a t =
11   | Vide
12   | Noeud of ('a t * clef * 'a * 'a t)
13
14 (** Fonctions pour modules externes ***********************************)
15
16 let vide = Vide
17 and gauche = function
18   | Noeud (g,_,_,_) -> g
19   | _ -> raise Not_found
20 and droite = function
21   | Noeud (_,_,_,d) -> d
22   | _ -> raise Not_found
23 and clef = function
24   | Noeud (_,x,_,_) -> x
25   | _ -> raise Not_found
26 and dcolor c _ = c.(0)
27
28 (** Vérification et recherche ************************************)
```

```
29
30 let verifier arbre =
31   let rec aux inf sup h = function
32     | Vide -> (0, h)
33     | Noeud (g,x,_,d) ->
34         if inf < x && x < sup then begin
35           let (ng, hg) = aux inf x h g in
36           let (nd, hd) = aux x sup h d in
37           (1+ng+nd, 1+(max hg hd))
38         end else invalid_arg "arbre incorrect"
39   in aux min_int max_int (-1) arbre
40
41 let rec rechercher x = function
42   | Vide -> raise Not_found
43   | Noeud (g,r,_,_) when x < r -> rechercher x g
44   | Noeud (_,r,_,d) when r < x -> rechercher x d
45   | Noeud (_,_,v,_) -> v
46
47 (** Insertion ***********************************************************)
48
49 let rec inserer nx nv = function
50   | Vide -> Noeud (Vide, nx, nv, Vide)
51   | Noeud (g,x,v,d) when nx < x -> Noeud (inserer nx nv g, x, v, d)
52   | Noeud (g,x,v,d) when x < nx -> Noeud (g, x, v, inserer nx nv d)
53   | Noeud (g,x,v,d) -> Noeud (g, x, nv, d)
54
55 (** Suppression *********************************************************)
56
57 let rec minimum = function
58   | Vide -> raise Not_found
59   | Noeud (Vide, x, d, _) -> (x, d)
60   | Noeud (g,_,_,_) -> minimum g
61
62 let rec supprimer_minimum = function
63   | Vide -> invalid_arg "supprimer_minimum"
64   | Noeud (Vide, _, _, d) -> d
65   | Noeud (g,x,v,d) -> Noeud (supprimer_minimum g, x, v, d)
66
67 let fusionner = function
68   | (Vide, a) | (a, Vide) -> a
69   | (g, d) ->
70       let (x,v) = minimum d in
71       Noeud (g, x, v, supprimer_minimum d)
72
73 let rec supprimer nx = function
74   | Vide -> Vide
75   | Noeud (g,x,v,d) when nx < x -> Noeud (supprimer nx g, x, v, d)
76   | Noeud (g,x,v,d) when x < nx -> Noeud (g, x, v, supprimer nx d)
77   | Noeud (g,_,_,d) -> fusionner (g, d)
78
79 (*********************************************************************)
```

## 2 Arbres AVL

```
1 (*********************************************************************
```

```
2   * ARBRES AVL
3   * par Stéphane GLONDU (Avril 2004)
4   ************************************************************)
5
6   (** Définition des types ***************************************)
7
8   type clef = int
9   let max_diff = try int_of_string (Sys.getenv "AVL_MAX_DIFF") with _ -> 1
10
11  type 'a t =
12    | Vide
13    | Noeud of ('a t * int * clef * 'a * 'a t)
14
15  let hauteur = function
16    | Vide -> -1
17    | Noeud (_,h,_,_,_) -> h
18
19  (** Fonctions pour modules externes ***********************************)
20
21  let vide = Vide
22  and gauche = function
23    | Noeud (g,_,_,_,_) -> g
24    | _ -> raise Not_found
25  and droite = function
26    | Noeud (_,_,_,_,d) -> d
27    | _ -> raise Not_found
28  and clef = function
29    | Noeud (_,_,x,_,_) -> x
30    | _ -> raise Not_found
31  and dcolor c _ = c.(0)
32
33  (** Vérification et recherche ***********************************)
34
35  let verifier arbre =
36    let rec aux inf sup = function
37      | Vide -> (0, -1)
38      | Noeud (g,h,x,_,d) ->
39          if inf < x && x < sup && begin
40            let hg = hauteur g and hd = hauteur d in
41            if hg > hd then hg - hd <= max_diff && h = hg+1
42            else hd-hg <= max_diff && h = hd+1
43          end then begin
44            let ng = fst (aux inf x g) in
45            let nd = fst (aux x sup d) in
46            (1+ng+nd, h)
47          end else invalid_arg "arbre␣AVL␣incorrect"
48    in aux min_int max_int arbre
49
50  let rec rechercher x = function
51    | Vide -> raise Not_found
52    | Noeud (g,_,r,_,_) when x < r -> rechercher x g
53    | Noeud (_,_,r,_,d) when r < x -> rechercher x d
54    | Noeud (_,_,_,v,_) -> v
55
56  (** Rééquilibrage ***********************************************)
57
58  let construire g x v d =
59    let h = 1 + max (hauteur g) (hauteur d) in
```

```
60    Noeud (g,h,x,v,d)
61
62  let equilibrer g x v d =
63    let hg = hauteur g and hd = hauteur d in
64    let error () = failwith "equilibrer" in
65    if hg > hd + max_diff then begin
66      match g with
67      | Vide -> error ()
68      | Noeud (gg,_,gx,gv,gd) ->
69          if hauteur gg >= hauteur gd then
70            (* Cas 1 *)
71            construire gg gx gv (construire gd x v d)
72          else begin
73            (* Cas 2 *)
74            match gd with
75            | Vide -> error ()
76            | Noeud (gdg,_,gdx,gdv,gdd) ->
77                construire (construire gg gx gv gdg) gdx gdv (construire gdd x v d)
78          end
79    end else if hd > hg + max_diff then begin
80      match d with
81      | Vide -> error ()
82      | Noeud (dg,_,dx,dv,dd) ->
83          if hauteur dd >= hauteur dg then
84            (* Cas 1' *)
85            construire (construire g x v dg) dx dv dd
86          else begin
87            (* Cas 2' *)
88            match dg with
89            | Vide -> error ()
90            | Noeud (dgg,_,dgx,dgv,dgd) ->
91                construire (construire g x v dgg) dgx dgv (construire dgd dx dv dd)
92          end
93    end else construire g x v d
94
95  (** Insertion ***********************************************************)
96
97  let rec inserer nx nv = function
98    | Vide -> Noeud (Vide, 0, nx, nv, Vide)
99    | Noeud (g,_,x,v,d) when nx < x -> equilibrer (inserer nx nv g) x v d
100   | Noeud (g,_,x,v,d) when x < nx -> equilibrer g x v (inserer nx nv d)
101   | Noeud (g,h,_,_,d) -> Noeud (g,h,nx,nv,d)
102
103 (** Suppression *********************************************************)
104
105 let rec minimum = function
106   | Vide -> raise Not_found
107   | Noeud (Vide, _, x, d, _) -> (x, d)
108   | Noeud (g,_,_,_,_) -> minimum g
109
110 let rec supprimer_minimum = function
111   | Vide -> failwith "supprimer_minimum"
112   | Noeud (Vide, _, _, _, d) -> d
113   | Noeud (g,_,x,v,d) -> equilibrer (supprimer_minimum g) x v d
114
115 let fusionner g d =
116   match (g, d) with
117   | (Vide, a) | (a, Vide) -> a
```

```
118     | _ ->
119         let (x,v) = minimum d in
120         equilibrer g x v (supprimer_minimum d)
121
122   let rec supprimer nx = function
123     | Vide -> Vide
124     | Noeud (g,_,x,v,d) when nx < x -> equilibrer (supprimer nx g) x v d
125     | Noeud (g,_,x,v,d) when x < nx -> equilibrer g x v (supprimer nx d)
126     | Noeud (g,_,_,_,d) -> fusionner g d
127
128   (*****************************************************************)
```

## 3 Extension de syntaxe pour les arbres rouge-noir

```
1   (** Extension de syntaxe pour les arbres rouge-noir *)
2
3   open Pcaml ;;
4
5   EXTEND
6     GLOBAL: expr patt;
7     rnterm_exp:
8       [ [ "("; g = rnterm_exp; r = LIDENT; d = rnterm_exp; ")" ->
9             <:expr< Noeud $g$ $lid:r^"c"$ $lid:r^"x"$ $lid:r^"v"$ $d$ >>
10        | a = LIDENT -> <:expr< $lid:a$ >>
11        | "{"; e = expr; "}" -> e
12        | "("; g = rnterm_exp; r = LIDENT; "["; c = expr; "]";
13            d = rnterm_exp; ")" ->
14              <:expr< Noeud $g$ $c$ $lid:r^"x"$ $lid:r^"v"$ $d$ >>
15        ] ] ;
16    rnterm_pat:
17      [ [ "("; g = rnterm_pat; r = LIDENT; d = rnterm_pat; ")" ->
18            <:patt< Noeud $g$ $lid:r^"c"$ $lid:r^"x"$ $lid:r^"v"$ $d$ >>
19        | "("; g = rnterm_pat; r = "_"; d = rnterm_pat; ")" ->
20            <:patt< Noeud $g$ _ _ _ $d$ >>
21        | a = LIDENT -> <:patt< $lid:a$ >>
22        | "{"; p = patt; "}" -> p
23        | "_" -> <:patt< _ >>
24        | "("; g = rnterm_pat; r = LIDENT; "["; c = patt; "]";
25            d = rnterm_pat; ")" ->
26              <:patt< Noeud $g$ $c$ $lid:r^"x"$ $lid:r^"v"$ $d$ >>
27        | "("; g = rnterm_pat; r = "_"; "["; c = patt; "]";
28            d = rnterm_pat; ")" ->
29              <:patt< Noeud $g$ $c$ _ _ $d$ >>
30        ] ] ;
31    expr: LEVEL "simple"
32      [ [ "@"; rn = rnterm_exp -> rn ] ];
33    patt: LEVEL "simple"
34      [ [ "@"; rn = rnterm_pat -> rn ] ];
35    END;;
```

# 4 Arbres rouge-noir

```
1  (*****************************************************************
2   * ARBRES ROUGE-NOIR
3   * par Stéphane GLONDU (Avril 2004)
4   *****************************************************************)
5
6  (** Définition des types ********************************************)
7
8
9  type clef = int
10 type couleur = Rouge | Noir | NoirNoir
11
12 type 'a t =
13   | Vide | VideN
14   | Noeud of ('a t * couleur * clef * 'a * 'a t)
15
16 let couleur = function
17   | Vide          -> Noir
18   | VideN         -> NoirNoir
19   | @(_ _[c] _) -> c
20
21 (** Fonctions pour modules externes ***********************************)
22
23 let vide = Vide
24 and gauche = function @(g _ _) -> g  | _ -> raise Not_found
25 and droite = function @(_ _ d) -> d  | _ -> raise Not_found
26 and clef    = function @(_ r _) -> rx | _ -> raise Not_found
27 and dcolor c = function
28   | @(_ _[Rouge] _) -> c.(1)
29   | @(_ _[Noir ] _) -> c.(2)
30   | _               -> raise Not_found
31
32 (** Vérification et recherche *****************************************)
33
34 let verifier arbre =
35   let error () = invalid_arg "arbre rouge-noir incorrect" in
36   if couleur arbre <> Noir then error ();
37   let hn_def = ref (-1) in
38   let rec aux inf sup h hn = function
39     | VideN | @(_ _ [NoirNoir] _) -> error ()
40     | Vide ->
41         if !hn_def = -1 then (hn_def := hn; 0, h)
42         else (if !hn_def = hn then 0, h else error ())
43     | @(g r d) ->
44         if inf < rx && rx < sup then begin
45           let hn' = hn + (if rc = Noir then 1 else 0) in
46           let (ng, hg) = aux inf rx h hn' g in
47           let (nd, hd) = aux rx sup h hn' d in
48           (1+ng+nd, 1+(max hg hd))
49         end else error ()
50   in aux min_int max_int (-1) 0 arbre
51
52 let rec rechercher x = function
53   | Vide                -> raise Not_found
54   | VideN               -> invalid_arg "arbre rouge-noir incorrect"
55   | @(g r _) when x < rx -> rechercher x g
56   | @(_ r d) when rx < x -> rechercher x d
```

```
57    | @(_ r _)              -> rv
58
59  (** Insertion ***************************************************)
60
61  let cor_ins_g = function
62    | @((a b[Rouge] c) d (e f[Rouge] g)) when couleur a = Rouge || couleur c = Rouge ->
63      @((a b[Noir ] c) d[Rouge] (e f[Noir ] g))
64    | @((a b[Rouge] (c d[Rouge] e)) f g) when couleur g = Noir ->
65      @((a b[Rouge] c) d[Noir ] (e f[Rouge] g))
66    | @(((a b[Rouge] c) d[Rouge] e) f g) when couleur g = Noir ->
67      @((a b[Rouge] c) d[Noir ] (e f[Rouge] g))
68    | a -> a
69
70  let cor_ins_d = function
71    | @((a b[Rouge] c) d (e f[Rouge] g)) when couleur e = Rouge || couleur g = Rouge ->
72      @((a b[Noir ] c) d[Rouge] (e f[Noir ] g))
73    | @(a b ((c d[Rouge] e) f[Rouge] g)) when couleur a = Noir ->
74      @((a b[Rouge] c) d[Noir ] (e f[Rouge] g))
75    | @(a b (c d[Rouge] (e f[Rouge] g))) when couleur a = Noir ->
76      @((a b[Rouge] c) d[Noir ] (e f[Rouge] g))
77    | a -> a
78
79  let inserer nx nv a =
80    let rec aux = function
81      | VideN                 -> failwith "inserer"
82      | Vide                  -> @({Vide} n[Rouge] {Vide})
83      | @(g r d) when nx < rx -> cor_ins_g @({aux g} r d)
84      | @(g r d) when rx < nx -> cor_ins_d @(g r {aux d})
85      | @(g _[c] d)           -> @(g n[c] d)
86    in match aux a with
87      | @(g r d) -> @(g r[Noir] d)
88      | _        -> failwith "inserer"
89
90  (** Suppression ***************************************************)
91
92  let eclaircir = function
93    | @(g r[NoirNoir] d) -> @(g r[Noir] d)
94    | VideN              -> Vide
95    | _                  -> invalid_arg "eclaircir"
96
97  let cor_sup_g =
98    let aux = function
99      | @(a b (c d[Noir] e)) when couleur c = Noir && couleur e = Noir ->
100       @(a b[if bc = Rouge then Noir else NoirNoir] (c d[Rouge] e))
101     | @(a b ((c d[Rouge] e) f[Noir ] g))       when couleur g = Noir ->
102       @((a b[Noir] c) d[bc] (e f[Noir] g))
103     | @(a b (c d[Noir ] (e f[Rouge] g)))                              ->
104       @((a b[Noir] c) d[bc] (e f[Noir] g))
105     | _ -> failwith "cor_sup_g"
106   in function
107     | @(a b (c d[Rouge] e)) when couleur a = NoirNoir ->
108       @({aux @({eclaircir a} b[Rouge] c)} d[Noir] e)
109     | @(a b c)              when couleur a = NoirNoir -> aux @({eclaircir a} b c)
110     | a -> a
111
112 let cor_sup_d =
113   let aux = function
114     | @((a b[Noir] c) d e) when couleur a = Noir && couleur c = Noir ->
```

```
115       @((a b[Rouge] c) d[if dc = Rouge then Noir else NoirNoir] e)
116     | @((a b[Noir ] (c d[Rouge] e)) f g)       when couleur a = Noir ->
117       @((a b[Noir] c) d[fc] (e f[Noir] g))
118     | @(((a b[Rouge] c) d[Noir ] e) f g)                              ->
119       @((a b[Noir] c) d[fc] (e f[Noir] g))
120     | _ -> failwith "cor_sup_d"
121   in function
122     | @((a b[Rouge] c) d e) when couleur e = NoirNoir ->
123       @(a b[Noir] {aux @(c d[Rouge] {eclaircir e})})
124     | @(a b c)              when couleur c = NoirNoir -> aux @(a b {eclaircir c})
125     | a -> a
126
127 let assombrir = function
128   | Vide            -> VideN
129   | @(a b[Rouge] c) -> @(a b[Noir     ] c)
130   | @(a b[Noir ] c) -> @(a b[NoirNoir] c)
131   | _               -> invalid_arg "assombrir"
132
133 let rec minimum = function
134   | @({Vide} r _) -> (rx, rv)
135   | @(g _ _)      -> minimum g
136   | _             -> failwith "minimum"
137
138 let rec supprimer_minimum = function
139   | @({Vide} _[Rouge] d) -> d
140   | @({Vide} _[Noir ] d) -> assombrir d
141   | @(a b c)             -> cor_sup_g @({supprimer_minimum a} b c)
142   | _                    -> failwith "supprimer_minimum"
143
144 let fusionner rc = function
145   | (Vide, a)  | (a, Vide) -> if rc = Noir then assombrir a else a
146   | (g, d) ->
147       let (rx, rv) = minimum d in
148       cor_sup_d @(g r {supprimer_minimum d})
149
150 let supprimer nx a =
151   let rec aux = function
152     | VideN                  -> invalid_arg "arbre␣rouge-noir␣incorrect"
153     | Vide                   -> Vide
154     | @(g r d) when nx < rx -> cor_sup_g @({aux g} r d)
155     | @(g r d) when rx < nx -> cor_sup_d @(g r {aux d})
156     | @(g _[c] d)            -> fusionner c (g, d)
157   in match aux a with
158     | VideN | Vide -> Vide
159     | @(g r d)     -> @(g r[Noir] d)
160
161 (*****************************************************************)
```

## Références


[1] *The Objective Caml system (release 3.07).* INRIA, 2003. Les documents qui suivent sont disponibles sur le site web caml.inria.fr.

[2] Xavier Leroy. *Documentation and user's manual.*

[3] Daniel de Rauglaudre. *Camlp4 - Tutorial and Reference Manual.*