Cours de Programmation
MIM1, 2000-2001, Devoir 2
Arbres binaires : ABR et AVL, CODES CAML

Laure DANTHONY, Sylvain FOUR

pour le 10 novembre 2000

1 Boîte à outils files : FIFO.ml

```
(* ---- Définition des files et constructeurs ---- *)

type 'a cell = {mutable father:'a ptr ; mutable son:'a ptr ; mutable item: 'a}
    and 'a ptr = Nil | Box of 'a cell
    and 'a queue = {mutable first:'a ptr ; mutable last: 'a ptr};;

let make_empty_queue ()=
  {first=Nil ; last=Nil};;
(*make_empty_queue : unit -> 'a queue = <fun>*)

let is_empty_queue queue = ((queue.first = Nil) & (queue.last = Nil));;
(*is_empty_queue : 'a queue -> bool = <fun>*)


(* ---- Destructeur ---- *)

let pointed_value=function
    Box(x) -> x
    | Nil    -> failwith "pointed_value : no Pointed value" ;;
(*pointed_value : 'a ptr -> 'a cell = <fun>*)


(* ---- Traducteur file -> liste ---- *)

let queue2list q=
    let rec aux=function
        Nil -> []
        | (Box x)->(x.item)::(aux x.son)
    in aux q.first;;
(*queue2list : 'a queue -> 'a list = <fun>*)


(* ---- Fonctions de base : push et pop ---- *)

let push it queue =
    if (is_empty_queue queue)
    then
        begin
            let new_box  = Box{father = Nil ; son = queue.first ; item = it} in
```

```
            queue.first <- new_box;
            queue.last  <- new_box
        end
    else
        begin
            let new_box  = Box {father = Nil ; son = queue.first ; item = it} in
                (pointed_value (queue.first)).father <- new_box;
            queue.first <- new_box
        end;;
(*push : 'a -> 'a queue -> unit = <fun>*)

let pop queue =
    if (is_empty_queue queue)
    then
        failwith "pop : empty queue"
    else
        let pointed = pointed_value queue.last in
        if queue.first=queue.last
        then
            begin
                queue.first<-Nil;
                queue.last <-Nil;
                pointed.item
            end
        else let previous=pointed_value pointed.father in
        begin
            previous.son <- Nil;
            queue.last<-pointed.father;
            pointed.item
        end;;
(*pop : 'a queue -> 'a = <fun>*)
```

2 Boîte à outils graphiques : `graphic.ml`

```
(*on a préalablement besoin des définitions des binary_tree*)
```

2.1 Première fonction d'affichage graphique

```
(* ---- Création d'une pile interne et de ses outils ---- *)

let (is_empty_queue, push, pop, flush) =
  let queue = ref(make_empty_queue ())
  in let is_empty_queue_temp () = ((!queue).first = Nil) && ((!queue).last = Nil)
  in let push_temp (it : binary_tree)  =
    if (is_empty_queue_temp())
    then
      begin
        let new_box  = Box{father = Nil ; son = (!queue).first ; item = it} in
        (!queue).first <- new_box;
        (!queue).last  <- new_box
      end
```

```
    else
      begin
        let new_box  = Box {father = Nil ; son = (!queue).first ; item = it} in
        (pointed_value ((!queue).first)).father <- new_box;
        (!queue).first <- new_box
      end
  and pop_temp () =
    if (is_empty_queue_temp ())
    then
      failwith "pop : empty queue"
    else
      let pointed = pointed_value (!queue).last in
      if (!queue).first=(!queue).last
      then
        begin
          (!queue).first<-Nil;
          (!queue).last <-Nil;
          pointed.item
        end
      else let previous=pointed_value pointed.father in
      begin
        previous.son <- Nil;
        (!queue).last<-pointed.father;
        pointed.item
      end
  and flush_temp () = (queue := make_empty_queue ())
  in (is_empty_queue_temp, push_temp, pop_temp, flush_temp);;


(* ---- Fonction puissance de deux ---- *)

let rec deux_puis n =
  match n with
    0 -> 1
  | n -> 2 * deux_puis (n-1) ;;


(* ---- Arbre fantôme ---- *)

let phantom_b = make_b  (-1) (0 : color) (empty_b()) (empty_b ());;


(* ---- Outils sur les listes ---- *)

let car l =
  match l with
    [] -> failwith "car"
  |x :: m -> x ;;

let cdr l =
  match l with
    [] -> failwith "cdr"
  |x :: m -> m ;;
```

```
(* ---- Création de la liste relative au parcours en largeur infixe
renversée ---- *)

let make_liste tr =
  if (is_empty_b tr)
  then []
  else
    begin
      flush () ;
      push tr ;
      let l = ref([])
      and aux = ref (empty_b())
      and h = (height_b tr) + 1
      in
      for i=1 to (deux_puis(h)-1) do
        aux := pop () ;
        l := (get_key_b !aux, get_colour_b !aux) :: (!l) ;
        if not(is_empty_b (get_left_son_b !aux))
        then (push (get_left_son_b !aux))
        else (push phantom_b) ;
        if not(is_empty_b (get_right_son_b !aux))
        then (push (get_right_son_b !aux))
        else (push phantom_b) ;
      done ;
      !l
    end;;


(* ---- Création de la liste relative au parcours en profondeur infixe ---- *)

let rec infix_list tr =
  if (is_empty_b tr)
  then []
  else (infix_list (get_left_son_b tr))@[get_key_b tr]@(infix_list (get_right_son_b tr)) ;;


(* ---- Type coord ---- *)

type coord = {mutable x : int ; mutable y : int ; mutable exist :
bool} ;;


(* ---- Impression des cercles ---- *)


let racine_2_sur_2 = sqrt 2. /. 2. ;;

let make_circle x1 y1 c r n =
  set_color c ;
  fill_circle x1 y1 (int_of_float r) ;
  set_color black ;
```

```
  draw_circle x1 y1 (int_of_float r) ;
  let n_size = text_size (string_of_int n)
  in moveto (x1 - (fst n_size)/2) (y1 - (snd n_size)/2) ;
  draw_string (string_of_int n) ;;


(* ---- Fonction d'affichage graphique version 1 ---- *)

exception Too_large ;;

let print_tree1 tr =
  clear_graph ();
  let l = make_liste tr
  in let ptr_l = ref (l)
  and h = height_b tr + 1
  in let j = deux_puis(h-1) - 1
  and aux = ref (0,0)
  and max_number_of_digits = List.fold_left max 0 (List.map fst (List.map text_size (List.map
 string_of_int (infix_list tr))))
  in let tab = Array.make (deux_puis(h)-1) (ref {x = 0 ; y = 0 ; exist = false})
  and r_int = int_of_float( sqrt ( float_of_int(max_number_of_digits*max_number_of_digits)
/. 4. +. 81.)) + 2
  in let r_float = float_of_int r_int
  in let d = min 4 (max 0
  (int_of_float((float_of_int(size_x())-.(2.*.float_of_int(j+1)+.1.)
*.r_float)/.(3.*.float_of_int(j+1)-.1.))))
  in let d2 = 2*d ;
  in
  if (h = 1)
  then
    begin
      aux := car(!ptr_l) ;
      tab.(j) <- ref {x = 0 ; y = 0 ; exist = (fst (!aux)<>(-1))} ;
      !(tab.(j)).x <- 2*r_int ;
      !(tab.(j)).y <- 2*r_int ;
      if fst (!aux)<>(-1)
      then
        begin
          make_circle !(tab.(j)).x !(tab.(j)).y (snd !aux) r_float (fst !aux) ;
        end ;
    end
  else
    begin
      for i = deux_puis (h-2) downto 1 do
        aux := car(!ptr_l) ;
        tab.(2*i+j-1) <- ref {x = 0 ; y = 0 ; exist = (fst (!aux)<>(-1))} ;
        !(tab.(2*i+j-1)).x <- i*4*r_int+i*d+(i-1)*d2;
        !(tab.(2*i+j-1) ).y <- 2*r_int ;
        if (!(tab.(2*i+j-1)).x > size_x()) or (!(tab.(2*i+j-1)).y > size_y())
        then () (*raise Too_large *) ;
        if fst (!aux)<>(-1)
        then
          begin
```

```
        make_circle !(tab.(2*i+j-1)).x !(tab.(2*i+j-1)).y (snd !aux) r_float (fst !aux) ;
      end ;
    ptr_l := cdr (!ptr_l) ;
    aux := car(!ptr_l) ;
    tab.(2*i+j-2) <- ref {x = 0 ; y = 0 ; exist = (fst (!aux)<>(-1))} ;
    !(tab.(2*i+j-2)).x <- (2*i-1)*2*r_int+(i-1)*d2+(i-1)*d ;
    !(tab.(2*i+j-2)).y <- 2*r_int ;
    if fst (!aux)<>(-1)
    then
      begin
        make_circle !(tab.(2*i+j-2)).x !(tab.(2*i+j-2)).y (snd !aux) r_float (fst !aux) ;
      end ;
    ptr_l := cdr (!ptr_l) ;
  done ;
  for i = j downto 1 do
    aux := car (!ptr_l) ;
    tab.(i-1) <- ref {x = 0 ; y = 0 ; exist = (fst (!aux)<>(-1))} ;
    !(tab.(i-1)).y <- !(tab.(2*i-1)).y + 6*r_int ;
    !(tab.(i-1)).x <- (!(tab.(2*i-1)).x + !(tab.(2*i)).x) / 2 ;
    if fst (!aux)<>(-1)
    then
      begin
        make_circle !(tab.(i-1)).x !(tab.(i-1)).y (snd !aux) r_float (fst !aux) ;
        if !(tab.(2*i-1)).exist
        then
          begin
            moveto !(tab.(i-1)).x (!(tab.(i-1)).y-r_int) ;
            lineto !(tab.(2*i-1)).x (!(tab.(2*i-1)).y+r_int) ;
          end ;
        if !(tab.(2*i)).exist
        then
          begin
            moveto !(tab.(i-1)).x (!(tab.(i-1)).y-r_int) ;
            lineto !(tab.(2*i)).x (!(tab.(2*i)).y+r_int) ;
          end ;
      end ;
    ptr_l := cdr (!ptr_l) ;
  done
end;;
```

2.2 Deuxième fonction d'affichage graphique

```
(* ---- Type arbre de coordonnées ---- *)

type coordinates_tree = Empty_c | Node_c of node_c
and node_c = {left_son_c: coordinates_tree ;
              mutable x: int ;
              key_c : int ;
              colour_c : color ;
              mutable y: int ;
              right_son_c: coordinates_tree} ;;
```

```
(* ---- Constructeurs et destructeurs de base ---- *)

let empty_c() = Empty_c ;;

let is_empty_c t = (t = empty_c()) ;;

let make_c left x_c n_c c_c y_c right = Node_c{left_son_c=left ; x =
x_c ; colour_c = c_c ; key_c = n_c ; y = y_c ; right_son_c = right};;

let get_x tr_c =
  match tr_c with
  | Empty_c -> failwith "get_x"
  | Node_c(n) -> n.x ;;

let get_key_c tr_c =
  match tr_c with
  | Empty_c -> failwith "get_key_c"
  | Node_c(n) -> n.key_c ;;

let get_colour_c tr_c =
  match tr_c with
  | Empty_c -> failwith "get_colour_c"
  | Node_c(n) -> n.colour_c ;;

let get_y tr_c =
  match tr_c with
  | Empty_c -> failwith "get_y"
  | Node_c(n) -> n.y ;;

let get_left_son_c tr_c=
  match tr_c with
  | Empty_c -> failwith "get_left_son_c"
  | Node_c(n) -> n.left_son_c ;;

let get_right_son_c tr_c =
  match tr_c with
  | Empty_c -> failwith "get_right_son_c"
  | Node_c(n) -> n.right_son_c ;;


(* ---- Largeurs gauche et droite ---- *)

let rec left_length tr_c =
  if is_empty_c tr_c
  then 0
  else
    begin
      if (is_empty_c (get_left_son_c tr_c)) && (is_empty_c (get_right_son_c tr_c))
      then 0
      else
        begin
          if is_empty_c (get_left_son_c tr_c)
          then max 0 (left_length (get_right_son_c tr_c) - (get_x (get_right_son_c tr_c))
```

```
+ (get_x tr_c))
          else
            begin
              if is_empty_c (get_right_son_c tr_c)
              then (left_length (get_left_son_c tr_c) - (get_x (get_left_son_c tr_c))
+ (get_x tr_c))
              else max 0 (max (left_length (get_left_son_c tr_c) - (get_x (get_left_son_c tr_c))
+ (get_x tr_c))
           (left_length (get_right_son_c tr_c) - (get_x (get_right_son_c tr_c)) + (get_x tr_c)))
            end
        end
    end ;;


let rec right_length tr_c =
  if is_empty_c tr_c
  then 0
  else
    begin
      if (is_empty_c (get_left_son_c tr_c)) && (is_empty_c (get_right_son_c tr_c))
      then 0
      else
        begin
          if is_empty_c (get_left_son_c tr_c)
          then max 0 (right_length (get_right_son_c tr_c) + (get_x (get_right_son_c tr_c))
- (get_x tr_c))
          else
            begin
              if is_empty_c (get_right_son_c tr_c)
              then max 0 (right_length (get_left_son_c tr_c) + (get_x (get_left_son_c tr_c))
- (get_x tr_c))
              else max 0 (max (right_length (get_left_son_c tr_c) + (get_x (get_left_son_c tr_c))
- (get_x tr_c))
           (right_length (get_right_son_c tr_c) + (get_x (get_right_son_c tr_c)) - (get_x tr_c)))
            end
        end
    end ;;


(* ---- Modification des coordonnées d'un arbre coordonnées ---- *)

let change_x tr_c new_x =
  match tr_c with
    Empty_c -> failwith "change_x"
  |Node_c(n) -> (n.x <- new_x) ;;

let change_y tr_c new_y =
  match tr_c with
    Empty_c -> failwith "change_y"
  |Node_c(n) -> (n.y <- new_y) ;;

let rec push_right tr_c n =
  if (is_empty_c tr_c)
```

```
  then ()
  else
    begin
      change_x tr_c ((get_x tr_c)+n) ;
      push_right (get_left_son_c tr_c) n ;
      push_right (get_right_son_c tr_c) n ;
    end ;;

let rec push_up tr_c n =
  if (is_empty_c tr_c)
  then ()
  else
    begin
      change_y tr_c ((get_y tr_c)+n) ;
      push_up (get_left_son_c tr_c) n ;
      push_up (get_right_son_c tr_c) n ;
    end ;;


(* ---- Passage d'un arbre binaire à un arbre coordonnées ---- *)

let rec make_tr_c tr r y=
  if is_empty_b tr
  then empty_c ()
  else
    begin
      if (is_empty_b (get_left_son_b tr) && is_empty_b (get_right_son_b tr))
      then make_c (empty_c()) (2*r) (get_key_b tr) (get_colour_b tr) (2*r) (empty_c())
      else
        begin
          if is_empty_b (get_left_son_b tr)
          then
            begin
              let tr_c_right = make_tr_c (get_right_son_b tr) r y
              in let x_c = get_x tr_c_right
              in if x_c < (3 * r)
              then push_right tr_c_right (3*r - x_c) ;
              make_c (empty_c ()) ((get_x tr_c_right) - r) (get_key_b tr) (get_colour_b tr)
((get_y tr_c_right) + y) tr_c_right ;
            end
          else
            begin
              if is_empty_b (get_right_son_b tr)
              then
                begin
                  let tr_c_left = make_tr_c (get_left_son_b tr) r y
                  in make_c tr_c_left ((get_x tr_c_left) + r) (get_key_b tr)
(get_colour_b tr) ((get_y tr_c_left) + y) (empty_c ()) ;
                end
              else
                begin
                  let tr_c_right = make_tr_c (get_right_son_b tr) r y
                  and tr_c_left = make_tr_c (get_left_son_b tr) r y
```

```
                  in let l_right = ref (right_length tr_c_left)
                  and h_right = height_b (get_right_son_b tr) + 1
                  and h_left = height_b (get_left_son_b tr) + 1
                  in if h_right > h_left
                  then push_up tr_c_left ((h_right - h_left)*y)
                  else
                    begin
                      if h_right < h_left
                      then push_up tr_c_right ((h_left - h_right)*y)
                    end ;
                  let d = ref 0 in
                  if ((is_empty_c (get_left_son_c tr_c_left)) &&
(is_empty_c (get_right_son_c tr_c_left))) || ((is_empty_c (get_left_son_c tr_c_right))
 && (is_empty_c (get_right_son_c tr_c_right)))
                  then
                    begin
                      push_right tr_c_right ((get_x tr_c_left) - (get_x tr_c_right) + 2 * r) ;
                      if ((get_x tr_c_right) - (left_length tr_c_right) - r) < r
                      then (d := 2 * r -(get_x tr_c_right) + (left_length tr_c_right))
                    end
                  else push_right tr_c_right ((get_x  tr_c_left) + !l_right) ; (* +r-r *)
                  let res = make_c tr_c_left (((get_x tr_c_left)+(get_x tr_c_right))/2)
(get_key_b tr) (get_colour_b tr) ((get_y tr_c_left)+y) tr_c_right in
                  if (!d <>0)
                  then push_right res (!d) ;
                  res
                end
            end
        end
    end ;;


(* ---- Fonction d'affichage graphique version 2 ---- *)

let print_tree2 tr =
  clear_graph ();
  let max_number_of_digits = List.fold_left max 0 (List.map fst (List.map text_size
(List.map string_of_int (infix_list tr))))
  in let r_int = int_of_float( sqrt (float_of_int(max_number_of_digits * max_number_of_digits)
/. 4. +. 81.)) + 2
  in let r_float = float_of_int r_int
  in let rec draw2 tr_c r =
    if is_empty_c tr_c
    then ()
    else
      begin
        make_circle (get_x tr_c) (get_y tr_c) (get_colour_c tr_c) (float_of_int r) (get_key_c tr_c) ;
        if not(is_empty_c (get_left_son_c tr_c))
        then
          begin
            moveto (get_x tr_c) ((get_y tr_c)-r)  ;
            lineto (get_x (get_left_son_c tr_c)) ((get_y (get_left_son_c tr_c))+r)   ;
          end ;
```

```
        if not(is_empty_c (get_right_son_c tr_c))
        then
          begin
            moveto (get_x tr_c) ((get_y tr_c)-r)  ;
            lineto (get_x (get_right_son_c tr_c)) ((get_y (get_right_son_c tr_c))+r) ;
          end ;
        draw2 (get_left_son_c tr_c) r ;
        draw2 (get_right_son_c tr_c) r ;
      end
  in
  if (height_b tr = 0)
  then  draw2 (make_tr_c tr r_int  (6*r_int) ) r_int
  else    draw2 (make_tr_c tr r_int (min (6*r_int) (int_of_float(((float_of_int(size_y()-15)
-.(2.*.r_float))/.(float_of_int(height_b tr))))))) r_int ;;
```

2.3 Fusion des deux versions

```
let draw_binary_tree tr =
  try
    print_tree1 tr ;
  with
    Too_large -> print_tree2 tr ;;
```

3 Les arbres binary_tree.ml

3.1 Définition des binary_tree

```
# use "FIFO.ml";; (*il faut la toolbox files pour le parcours en largeur*)
open Graphics;;


(* ---- Définition ---- *)

type binary_tree = Empty_b | Node_b of node_b
        and node_b = {mutable key_b : int ; mutable colour_b : color ;
         mutable left_son_b : binary_tree  ; mutable right_son_b : binary_tree};;


(* ---- Constructeurs et destructeurs de base ---- *)

let empty_b() = Empty_b;;
(*empty_b : unit -> binary_tree = <fun>*)

let is_empty_b tree = match tree with
        Empty_b -> true
        | _ -> false ;;
(*is_empty_b : binary_tree -> bool = <fun>*)

let make_b k c l r =
        Node_b{key_b = k ; colour_b = c ; left_son_b = l ; right_son_b = r};;
(*int -> color -> binary_tree -> binary_tree -> binary_tree = <fun>*)
```

```
let get_node_b tree = match tree with
        Empty_b -> failwith "get_node_b : Empty binary_tree"
        | Node_b n -> n;;
(*get_node_b : binary_tree -> node_b = <fun>*)

let get_key_b tree = match tree with
        Empty_b -> failwith "get_key_b : Empty binary_tree"
        | Node_b{key_b = k} -> k;;
(*get_key_b : binary_tree -> int = <fun>*)

let get_colour_b tree = match tree with
        Empty_b -> failwith "get_colour_b : Empty binary_tree"
        | Node_b{colour_b = c} -> c;;
(*get_colour_b : binary_tree -> color = <fun>*)

let get_left_son_b tree = match tree with
        Empty_b -> failwith "get_left : Empty binary_tree"
        | Node_b{left_son_b = l} -> l;;
(*get_left_son_b : binary_tree -> binary_tree = <fun>*)

let get_right_son_b tree = match tree with
        Empty_b -> failwith "get_right : Empty binary_tree"
        | Node_b{right_son_b = r} -> r;;
(*get_right_son_b : binary_tree -> binary_tree = <fun>*)

let rec height_b t =
if (is_empty_b t) then (-1)
else 1 + (max (height_b (get_left_son_b t)) (height_b (get_right_son_b t)));;
(*height_b : binary_tree -> int = <fun>*)
```

3.2 Parcours graphiques

3.2.1 En profondeur préfixe

```
let deep_first_graphic tree =
   let rec deep_first_graphic_bis tr =
        if is_empty_b tr then ()
        else
                begin
                        colorize (get_node_b tr) red;
                        draw_binary_tree tree;
                        for i=1 to 1000000 do () done;
                        deep_first_graphic_bis (get_left_son_b tr) ;
                        deep_first_graphic_bis (get_right_son_b tr)
                end
        in deep_first_graphic_bis tree;;
(*deep_first_graphic : binary_tree -> unit = <fun>*)
```

3.2.2 En largeur préfixe

```
let bread_first_graphic tree =
```

```
        let aux = make_empty_queue() in
                begin
                        push tree aux;
                        while not(is_empty_queue aux) do
                                let res = pop aux in
                                        begin
                                                colorize (get_node_b res) red;
                                                draw_binary_tree tree;
                                                for i=1 to 1000000 do () done;
                                                if not(is_empty_b (get_left_son_b res))
                                                then push (get_left_son_b res) aux;
                                                if not(is_empty_b (get_right_son_b res))
                                                then push (get_right_son_b res) aux;
                                        end
                        done
                end;;
(*bread_first_graphic : binary_tree -> unit = <fun>*)
```

3.3 Recherche d'une clef dans un ABR

3.3.1 La fonction de recherche

```
(*coloration des noeuds visités en c1=color*)
(*coloration  de l'éventuel noeud trouvé en c2 = color*)

exception Found of node;;

let colorize nd c =
    nd.colour <- c;;
(*colorize : node -> color -> unit = <fun>*)

let rec search_bis k t c1 =
    if is_empty_b t then ()
    else
        begin
          let nd = get_node_b t in
           colorize nd c1;
           if (nd.key = k) then raise (Found nd)
           else
               if (nd.key > k) then search_bis k (get_left_b t) c1
               else search_bis k (get_right_b t) c1
         end;;

let search k t c1 c2 =
    (try
        (search_bis k t c1)
    with
        Found(nd) -> colorize nd c2);;

(*search_bis : int -> binary_tree -> color -> unit = <fun>*)
(*search : int -> binary_tree -> color -> color -> unit = <fun>*)
```

3.3.2 La fonction de recherche avec compteur

```
(*compteur*)

let make_counter () =
    let c = ref (0) in
        ((function () -> (c := 0)) ,
        (function () -> (c := !c + 1)) ,
        (function () -> (!c) ));;

(*recherche avec compteur*)

let rec search_bis k t c1 tick=
   if is_empty_b t then ()
   else
       begin
           let nd = get_node_b t in
           colorize nd c1;
           tick();
           if (nd.key = k) then raise (Found nd)
           else
               if (nd.key > k) then search_bis k (get_left_b t) c1 tick
               else  search_bis k (get_right_b t) c1 tick
        end;;

let search k t c1 c2 =
    let (init,tick,get)=make_counter() in
    begin
        init();
        (try
                (search_bis k t c1 tick)
        with
                Found(nd) -> colorize nd c2);
        get()
    end;;

(*search_bis : int -> binary_tree -> color -> (unit -> 'a) -> unit = <fun>*)
(*search : int -> binary_tree -> color -> color -> int = <fun>*)
```

3.3.3 La recherche graphique

```
let graphic_search_aux k t c1 =
        let rec graphic_search_aux2 k tr c1 =
        if is_empty_b tr then ()
        else
                begin
                        let nd = get_node_b tr in
                        colorize nd c1;
                        draw_binary_tree t;
                        for i=1 to 1000000 do () done;
                        if (nd.key_b = k) then raise (Found nd)
                        else
```

```
                                if (nd.key_b > k)
                                then graphic_search_aux2  k (get_left_son_b tr) c1
                                else graphic_search_aux2 k (get_right_son_b tr) c1
                end
        in graphic_search_aux2 k t c1;;

let graphic_search k t c1 c2 =
        (try
                (graphic_search_aux k t c1)
        with
                Found(nd) ->
                        begin
                                colorize nd c2;
                                draw_binary_tree t
                        end);;
(*search : int -> binary_tree -> color -> color -> unit = <fun>*)

let search_n_init k t = search k t cyan cyan ;;
(*search_n_init : int -> binary_tree -> unit = <fun>*)
```

3.4 Création d'exemples

```
let real_life_binary_tree ()=
        make_b 11 cyan
        (make_b  4 cyan
         (make_b 2 cyan
          (make_b  1 cyan (empty_b ()) (empty_b ()))
          (make_b  3 cyan (empty_b ()) (empty_b ()))
         )
         (make_b  6 cyan
          (make_b 5 cyan (empty_b ()) (empty_b ()))
          (make_b  8 cyan
           (make_b  7 cyan (empty_b ()) (empty_b ()))
           (make_b  9 cyan (empty_b ()) (empty_b ()))
          )
         )
        )
        (make_b 20 cyan
         (make_b  15 cyan
          (make_b  12 cyan (empty_b ()) (empty_b ()))
          (make_b 17 cyan (empty_b ()) (empty_b ()))
         )
         (make_b  21 cyan (empty_b ()) (empty_b ()))
        );;
(*real_life_binary_tree : unit -> binary_tree = <fun>*)

(*coloration d'un arbre entier en même temps*)

let rec tag_b t c =
    if not (is_empty_b t) then let nd = get_node_b t in
        begin
```

```
            nd.colour_b <- c;
            tag_b nd.left_son_b c;
            tag_b nd.right_son_b c
        end;;
(*tag_b : binary_tree -> color -> unit = <fun>*)

(*création d'un ABR par insertion successives aux feuilles*)

let rec brt_insertion k t =
    if (is_empty_b t) then
        make_b k cyan (empty_b()) (empty_b())
    else
        if (k = (get_key_b t)) then t
        else if (k < (get_key_b t)) then
                let result = brt_insertion k (get_left_son_b t) in
                begin
                  (get_node_b t).left_son_b <- result;
                  t
                end
         else let result = brt_insertion k (get_right_son_b t) in
                 begin
                   (get_node_b t).right_son_b <- result;
                   t
                 end;;
 (*brt_insertion : int -> binary_tree -> binary_tree = <fun>*)

 let rec succ_insertion_b = function
     [] -> Empty_b
   | a::r -> brt_insertion a (succ_insertion_b r);;
 (*succ_insertion_b : int list -> binary_tree = <fun>*)

 let rec random_list n m =
    if (n = 0) then []
    else (Random.int m)::(random_list (n-1) m);;
 (*random_list : int -> int -> int list = <fun>*)

 let big_brt n =
 let random_tree = succ_insertion_b (random_list (n-1) (10*n)) in
     begin
         tag_b random_tree cyan;
         random_tree
     end;;
 (*big_brt : int -> binary_tree = <fun>*)
```

4 Les arbres Adelson, Velskii, Landis : AVL.ml

4.1 Définition des AVL

```
(*# use "FIFO.ml";;*)

open Graphics;;
```

```
(* ---- Définition du type avl ---- *)

type avl = Empty | Node of node
        and node = {mutable delta : int ; mutable key : int ; mutable colour : color ;
         mutable left_son : avl ; mutable right_son : avl};;


(* ---- Constructeurs et destructeurs ---- *)

let empty_avl() = Empty;;
(*empty_avl : unit -> avl = <fun>*)

let is_empty tree = match tree with
        Empty -> true
        | _ -> false ;;
(*is_empty : avl -> bool = <fun>*)

let make_avl d k c l r =
        Node{delta =d ; key = k ; colour = c ; left_son = l ; right_son = r};;
(*make_avl : int -> int -> color -> avl -> avl -> avl = <fun>*)

let get_node tree = match tree with
        Empty -> failwith "get_node : Empty avl"
        | Node n -> n;;
(*get_node : avl -> node = <fun>*)

let get_key tree = match tree with
        Empty -> failwith "get_key : Empty avl"
        | Node{key = k} -> k;;
(*get_key : avl -> int = <fun>*)

let get_colour tree = match tree with
        Empty -> failwith "get_colour : Empty avl"
        | Node{colour = c} -> c;;
(*get_colour : avl -> color = <fun>*)

let get_left tree = match tree with
        Empty -> failwith "get_left : Empty avl"
        | Node{left_son = l} -> l;;
(*get_left : avl -> avl = <fun>*)

let get_right tree = match tree with
        Empty -> failwith "get_right : Empty avl"
        | Node{right_son = r} -> r;;
(*get_right : avl -> avl = <fun>*)

let get_delta tree = match tree with
        Empty -> failwith "get_delta : Empty avl"
        | Node{delta = d} -> d;;
(*get_delta : avl -> int = <fun>*)

let rec height tree =
```

```
        if is_empty tree then (-1)
        else 1+ (max (height (get_left tree)) (height (get_right tree)));;
(*height : avl -> int = <fun>*)
```

4.2 Rotations

```
let avl_right_rotation q =
        if (is_empty q) then failwith "avl_right_rotation : Empty avl"
        else
                let p = get_left q in
                let np = get_node p and nq = get_node q in
                begin
                        if (is_empty p) then
                            failwith "avl_right_rotation : Empty left son";
                        nq.left_son <- np.right_son;
                        np.right_son <- q;
                        p
                end;;
(*avl_right_rotation : avl -> avl = <fun>*)

let avl_left_rotation p =
        if (is_empty p) then failwith "avl_left_rotation : Empty avl"
        else
                let q = get_right p in
                let np = get_node p and nq = get_node q in
                begin
                        if (is_empty q) then
                            failwith "avl_left_rotation : Empty right son";
                        np.right_son <- nq.left_son;
                        nq.left_son <- p;
                        q
                end;;
(*avl_left_rotation : avl -> avl = <fun>*)

let avl_left_right_rotation q =
        if (is_empty q) then failwith "avl_left_right_rotation : Empty avl"
        else
                let r = get_left q in
                let nr = get_node r and nq = get_node q in
                if (is_empty r) then
                    failwith "avl_left_right_rotation : Empty left son"
                else let s=get_right r in let ns=get_node s in
                begin
                        if (is_empty s) then
                            failwith "avl_left_right_rotation : Empty left-right son";
                        nr.right_son<-ns.left_son;
                        nq.left_son <- ns.right_son;
                        ns.left_son <- r;
                        ns.right_son <- q;
                        s
                end;;
(*avl_left_right_rotation : avl -> avl = <fun>*)
```

```
let avl_right_left_rotation q =
        if (is_empty q) then failwith "avl_right_left_rotation : Empty avl"
        else
                let r = get_right q in
                let nr = get_node r and nq = get_node q in
                if (is_empty r) then
                    failwith "avl_right_left_rotation : Empty right son"
                else let s=get_left r in let ns=get_node s in
                begin
                        if (is_empty s) then
                            failwith "avl_right_left_rotation : Empty right-left son";
                        nq.right_son<-ns.left_son;
                        nr.left_son <- ns.right_son;
                        ns.left_son <- q;
                        ns.right_son <- r;
                        s
                end;;
(*avl_right_left_rotation : avl -> avl = <fun>*)
```

4.3 Adjonction d'une clef dans un AVL

```
let rec tag t c =
    if not(is_empty t) then  let n=get_node t in
    begin
        n.colour<-c;
        tag n.left_son c;
        tag n.right_son c
    end;;
(*tag : avl  -> color -> unit = <fun>*)

let avl_insertion k t=
        let rec insert2 ke tr=
                begin
                  if (is_empty tr) then
                         ( make_avl 0 ke blue (empty_avl ()) (empty_avl ()) , true)
                    else let key_tr=get_key tr in
                    if (k=key_tr) then (tr, false)
                       else if (k<key_tr) then let (t1,b1)=insert2 ke (get_left tr) in
                         if b1 then (* la hauteur gauche a augmenté*)
                            begin
                              if ((get_delta tr)<=0) then
                                begin
                                  (get_node tr).delta<-(get_delta tr)+1;
                                  (get_node tr).left_son<-t1;
                                  (tr,((get_delta tr)=1))
                          (*si le nv déséquilibre est 1, la hauteur a augmenté*)
                                end
                              else
                                begin
                                if ((get_delta t1)=1) then
                                  begin (* faire une rotation droite sur tr*)
                                    let result=avl_right_rotation tr in
```

```
                  begin
                           tag result red;
                    (get_node result).delta<-0;
                    (get_node (get_node result).right_son).delta<-0;
                    (result,false)
                  end
              end
            else
              begin(* faire une rotation gauche-droite sur tr*)
                let result=avl_left_right_rotation tr in
                  begin
                    tag result red;
                    if ((get_delta result)=1) then (*cas a*)
                       begin
                         (get_node (get_left result)).delta <- 0;
                            (*il était à -1*)
                         (get_node (get_right result)).delta <- (-1)
                            (*il était à 1*)
                       end
                       else if ((get_delta result)= (-1)) then
                       begin
                         (get_node (get_left result)).delta <- 1;
                             (*il était à -1*)
                         (get_node (get_right result)).delta <- 0
                             (*il était à 1*)
                       end
                       else  (* 3eme cas oublié ds l'énoncé *)
                       begin
                         (get_node (get_left result)).delta <- 0;
                         (get_node (get_right result)).delta <- 0
                       end;
                    (get_node result).delta<-0;
                    (result,false)
                  end
              end
            end
        end
     else (*la hauteur gauche n'a pas varié*)
        begin
          (get_node tr).left_son<- t1;
          (tr,false)
        end
   else let (t1,b1)=insert2 ke (get_right tr) in
     if b1 then (* la hauteur droite a augmenté*)
        begin
          if ((get_delta tr)>=0) then
            begin
              (get_node tr).delta<-(get_delta tr)-1;
              (get_node tr).right_son<-t1;
              (tr,((get_delta tr)= -1))
      (*si le nv déséquilibre est -1, la hauteur a augmenté*)
            end
          else
```

```
                                begin
                                if ((get_delta t1)= (-1)) then
                                  begin (* faire une rotation gauche sur tr*)
                                    let result=avl_left_rotation tr in
                                      begin
                                        tag result red;
                                        (get_node result).delta<-0;
                                        (get_node (get_node result).left_son).delta<-0;
                                        (result,false)
                                      end
                                  end
                                else
                                  begin(* faire une rotation droite-gauche sur tr*)
                                    let result=avl_right_left_rotation tr in
                                      begin
                                        tag result red;
                                        if ((get_delta result)=1) then
                                           begin
                                             (get_node (get_left result)).delta <- 0;
                                                   (*il etait à -1*)
                                             (get_node (get_right result)).delta <- (-1)
                                                   (*il etait à 1*)
                                           end
                                           else if ((get_delta result)=(-1)) then
                                           begin
                                             (get_node (get_left result)).delta <- 1;
                                                    (*il etait à -1*)
                                             (get_node (get_right result)).delta <- 0
                                                    (*il etait à 1*)
                                           end
                                           else  (* 3eme cas oublié ds l'énoncé *)
                                           begin
                                             (get_node (get_left result)).delta <- 0;
                                             (get_node (get_right result)).delta <- 0
                                           end;
                                        (get_node result).delta<-0;
                                        (result,false)
                                      end
                                  end
                                end
                            end
                         else
                            begin
                              (get_node tr).right_son<- t1;
                              (tr,false)
                            end
                end
        in fst (insert2 k t);;
(* avl_insertion : int -> avl -> avl = <fun> *)
```

4.4 Impression graphique d'un AVL

```
let ee = int_of_float;; (* astuce !*)
(*ee : float -> int = <fun>*)

(*open_graph "";;*)

let draw_int x y i=
      moveto x y;
      draw_string (string_of_int i);;
(*draw_int : int -> int -> int -> unit = <fun>*)

let rec draw_avl x y hx hy = function
  | Empty -> ()
  | Node(n)->
   let xg = x-. hx and yg = y -. hy
   and xd = x +. hx and yd = y -. hy in
      moveto (ee x) (ee y);
      set_color  n.colour;
      let aux=ee ((log ( float_of_int n.key))/. (log 10.))
      in fill_circle (2+(3*aux)+(ee x)) (3+(ee y)) (6+(3*aux)) ;
      set_color (if n.colour=black then white else black);
      draw_int (ee x) (ee y) n.key;
      if not(is_empty n.left_son) then
       begin
        moveto ((ee x)-2) ((ee y)-2);
        lineto ((ee xg)+2) ((ee yg)+2);
        draw_avl  xg yg (hx*. 0.5) hy n.left_son
       end;
      if not(is_empty n.right_son) then
        begin
          moveto ((ee x)+2) ((ee y)-2);
          lineto (ee xd) (ee yd);
          draw_avl  xd yd (hx*. 0.5) hy n.right_son
        end ;;
(* draw_avl : float -> float -> float -> float -> avl -> unit = <fun> *)

let draw_tree_avl t =
   clear_graph ();
   let y_max =float_of_int (size_y ())
   and x_max=float_of_int (size_x ()) in
   let hy = (y_max /. (1.+.(float_of_int (height t))))
   and hx = 0.2*.x_max in
   draw_avl (x_max/.2.) (y_max *. 0.9) hx hy t;;
(* draw_tree_avl : avl -> unit = <fun> *)

(* fonction graphique de debugage qui imprime les fonctions de déséquilibre *)

let rec draw_avl_debug x y hx hy = function
  | Empty -> ()
  | Node(n)->
   let xg = x-. hx and yg = y -. hy
   and xd = x +. hx and yd = y -. hy in
```

```
      moveto (ee x) (ee y);
      set_color  n.colour;
      let aux=ee ((log ( float_of_int n.key))/. (log 10.))
      in fill_circle (2+(3*aux)+(ee x)) (3+(ee y)) (10+(3*aux)) ;
      set_color (if n.colour=black then white else black);
      draw_int (ee x) (ee y) n.key;
      draw_int (ee x) ((ee y)-10) n.delta;
      if not(is_empty n.left_son) then
       begin
        moveto ((ee x)-2) ((ee y)-2);
        lineto ((ee xg)+2) ((ee yg)+2);
        draw_avl_debug  xg yg (hx*. 0.5) hy n.left_son
       end;
      if not(is_empty n.right_son) then
        begin
          moveto ((ee x)+2) ((ee y)-2);
          lineto (ee xd) (ee yd);
          draw_avl_debug  xd yd (hx*. 0.5) hy n.right_son
        end ;;
(* draw_avl_debug : float -> float -> float -> float -> avl -> unit = <fun> *)


let draw_debug t =
   clear_graph ();
   let y_max =float_of_int (size_y ())
   and x_max=float_of_int (size_x ()) in
   let hy = (y_max /. (1.+.(float_of_int (height t))))
   and hx = 0.2*.x_max in
   draw_avl_debug (x_max/.2.) (y_max *. 0.9) hx hy t;;
(* draw_debug : avl -> unit = <fun> *)
```

4.5 Création d'exemples

```
let rec succ_insertion=function
        [] -> Empty
        | a::r -> avl_insertion a (succ_insertion r);;
(*succ_insertion : int list -> avl = <fun>*)


let rec random_list n m=if n=0 then [] else (Random.int m)::(random_list (n-1) m);;
(*random_list : int -> int -> int list = <fun>*)

let big_test n=let tree=succ_insertion (random_list n (10*n)) in
        begin
          tag tree magenta;
          draw_tree_avl tree
        end;;
(*big_test : int -> unit = <fun>*)


let lovely_example()  =
  let t=[|12;3;2;5;4;7;9;11;14;10|] in
```

```
  let aux = ref (empty_avl()) in
    for i = 0 to 9
    do
      begin
        aux:=avl_insertion (t.(i)) !aux;
        draw_tree_avl (!aux);
        for i=1 to 10000000 do () done;
        tag (!aux) cyan;
      end
    done;
    tag (!aux) cyan;
    draw_tree_avl (!aux);;
(*lovely_example : unit -> unit = <fun>*)
```

5 Fichier de tests test_b.ml

```
(* --------------------------------- Tests binary_tree ----------------------------- *)

# use "binary_tree.ml";;
# use "FIFO.ml";;

(*ocamlmktop -custom -o myocaml unix.cma graphics.cma -cclib -lgraphics -cclib -lX11 *)

open Graphics;;


(*illustration des parcours graphiques*)

let t = real_life_binary_tree();;
open_graph"";;
draw_binary_tree t;;
deep_first_graphic t;;
tag_b t cyan;;
draw_binary_tree t;;
bread_first_graphic t;;
tag_b t cyan;;
close_graph();;

(*recherche dans un ABR*)

let ex2 = big_brt 10;;
open_graph"";;
draw_binary_tree ex2;;
tag_b ex2 cyan;graphic_search 84 ex2 green red;;
close_graph();;

(*hauteur moyenne d'une clef dans un ABR*)

stat_brt 1000 100;;

(*profondeur moyenne d'une clef dans un ABR*)
```

```
make_n_search_brt 10 10000;;
```

6 Fichier de tests tests_avl.ml

```
(* ----------------------------------- Tests AVL ------------------------------------- *)

# use "AVL.ml";;
# use "FIFO.ml";;

(*ocamlmktop -custom -o myocaml unix.cma graphics.cma -cclib -lgraphics -cclib -lX11 *)

open Graphics;;

(*insertion d'une clef dans un AVL*)

open_graph"";;
lovely_example();;
close_graph();;

(*lovely_example, étape par étape*)

open_graph "";;
let aux=empty_avl();;
let t1=avl_insertion 12 aux;; (*pas de rotation*)
draw_tree_avl t1;;
tag t1 cyan;;
let t2 = avl_insertion 3 t1;; (*pas de rotation*)
draw_tree_avl t2;;
tag t2 cyan;;
let t3 = avl_insertion 2 t2;; (*rotation-droite*)
draw_tree_avl t3;;
tag t3 cyan;;
draw_tree_avl t3;;
let t4 = avl_insertion 5 t3;; (*pas de rotation*)
draw_tree_avl t4;;
let t5 = avl_insertion 4 t4;; (*rotation droite*)
draw_tree_avl t5;;
tag t5 cyan;;
draw_tree_avl t5;;
let t6 = avl_insertion 7 t5;; (*rotation gauche*)
draw_tree_avl t6;;
tag t6 cyan;;
draw_tree_avl t6;;
let t7 = avl_insertion 9 t6;; (*rotation gauche-droite*)
draw_tree_avl t7;;
tag t7 cyan;;
draw_tree_avl t7;;
let t8 = avl_insertion 11 t7;; (*pas de rotation*)
draw_tree_avl t8;;
let t9 = avl_insertion 14 t8;; (*pas de rotation*)
```

```
draw_tree_avl t8;;
let t10 = avl_insertion 10 t9;; (*rotation droite-gauche*)
draw_tree_avl t9;;
tag t10 cyan;;
draw_tree_avl t10;;
close_graph();;
```