```
(* définition d'un arbre hétérogène à l'aide de listes *)
type ('f,'n) arbre=
                  | Feuille of 'f
                  | Noeud of 'n * (('f,'n) foret)
and  ('f,'n) foret=
                    Arbres of ('f,'n) arbre list;;

(* définition d'un arbre hétérogène à l'aide de tableaux *)
#type ('f,'n) arbre =
                    Feuille of 'f
                  | Noeud of 'n * (('f,'n) foret)
and  ('f,'n) foret =
                    Arbres of ('f,'n) arbre vect;;
```

Test d'un arbre de recherche :

```
#let grandInt = 1000000;; (* très grand *)
grandInt : int = 1000000

#let rec testminmax = function
              | Vide -> (true, grandInt, -grandInt)
              | Noeud (gauche , n , droite) ->
                    match (testminmax gauche,testminmax droite) with
                      ((testg,ming,maxg),(testd,mind,maxd)) ->
                          (testg && testd && (maxg <= n) && (n < mind),
                           min (min ming mind) n,
                           max (max maxg maxd) n);;

testminmax : int arbre_bin -> bool * int * int = <fun>

#let teste_arbre_rech a =
    match testminmax a with (test,_,_) -> test;;
teste_arbre_rech : int arbre_bin -> bool = <fun>
```

Recherche dans un arbre binaire de recherche :

```
#let rec figure_dans_arbre x a =
      match a with
        | Vide -> false
        | Noeud (_ , n , _) when x = n -> true
        | Noeud (gauche , n , _) when x < n -> figure_dans_arbre x gauche
        | Noeud (_ , n , droite) (* x > n *) -> figure_dans_arbre x droite;;
figure_dans_arbre : 'a -> 'a arbre_bin -> bool = <fun>
```

Insertion en une feuille

```
#let rec insere_dans_arbre x a =
              match a with
                 | Vide -> Noeud (Vide , x , Vide)
                 | Noeud (gauche , n , droite) when x <= n  ->
                           Noeud( insere_dans_arbre x gauche, n, droite)
                 | Noeud (gauche , n , droite)  (* x > n *) ->
                           Noeud( gauche, n, insere_dans_arbre x droite);;
insere_dans_arbre : 'a -> 'a arbre_bin -> 'a arbre_bin = <fun>
```

Transformation d'une liste en un arbre binaire de recherche :

```
#let rec liste_en_arbre = function
                    | [] -> Vide
                    | h::r -> insere_dans_arbre h (liste_en_arbre r);;
liste_a_arbre : 'a list -> 'a arbre_bin = <fun>
```

suivi d'un parcours infixe de l'arbre : analogue au tri rapide

```
#let arbre_en_liste_triee =
     let rec aux = function
         | Vide -> []
      | Noeud(gauche,n,droite) -> (aux gauche) @ (n::(aux droite))
     in aux;;
```

Insertion à la racine : on partitionne l'arbre en les éléments plus petits que x et les éléments plus grands que x, puis on met x à la racine :

```
let insere x a =
    let rec partitionne = function
         | Vide -> Vide,Vide
         | Noeud(gauche,n,droite) when n < x ->
                let (g1,d1) = partitionne droite in
                     (Noeud(gauche,n,g1) , d1)
         | Noeud(gauche,n,droite) ->
                let (g1,d1) = partitionne gauche in
                     (g1 , Noeud(d1,n,droite))
    in let (gauche,droite) = partitionne a in
            Noeud (gauche,x,droite);;
```

Suppression d'un élément, schéma général

```
let rec supprime_dans_arbre x a=
      match a with
         | Vide -> raise Not_found
         | Noeud (gauche , n , droite) when x = n -> supprime_racine a
         | Noeud (gauche , n , droite) when x < n ->
               Noeud( supprime_dans_arbre x gauche, n, droite)
         | Noeud (gauche , n , droite) (* x > n *) ->
              Noeud( gauche, n, supprime_dans_arbre x droite)
where supprime_racine = ..... ;;
```

Recherche et suppression de l'élément le plus à droite (le plus grand)

```
#let rec suppr_a_droite = function
      | Vide -> raise Not_found
      | Noeud (gauche, n, Vide) -> (gauche , n)
      | Noeud (gauche, n, droite) ->
            let (nouveau_droite , plus_a_droite) = suppr_a_droite droite in
                  Noeud (gauche, n, nouveau_droite) , plus_a_droite;;
suppr_a_droite : 'a arbre_bin -> 'a arbre_bin * 'a = <fun>
```

Suppression de la racine

```
#let supprime_racine = function
      | Vide -> raise Not_found
      | Noeud (gauche, _ , Vide) -> gauche
      | Noeud (Vide, _ , droite) -> droite
      | Noeud (gauche, r , droite) ->
           let (nouveau_gauche , nouvelle_racine) = suppr_a_droite gauche in
           Noeud(nouveau_gauche, nouvelle_racine , droite);;
supprime_racine : 'a arbre_bin -> 'a arbre_bin = <fun>
```

```
(* définition d'un arbre hétérogène à l'aide d'enregistrements *)
#type  ('f,'n) arbre   =
                  | Feuille of 'f t_feuille
                  | Noeud of ('f,'n) t_noeud
and    'f t_feuille   = { mutable etiq_feuille :'f}
and   ('f,'n) t_noeud = { mutable etiq_noeud : 'n ;
                          mutable branches : ('f,'n) arbre vect};;

(* définition d'un arbre binaire hétérogène *)
#type ('f,'n) arbre_bin =
                    Feuille of 'f
                  | Noeud of (('f,'n) arbre) * 'n * (('f,'n) arbre);;
```

```
 (* nombre de noeuds, de feuilles, hauteur d'un arbre hétérogène (listes) *)
#type ('f,'n) arbre=
                  | Feuille of 'f
                  | Noeud of 'n * (('f,'n) arbre list);;
#let rec nb_noeuds_arbre = function
                |  Feuille _ -> 0
                |  Noeud ( _ , branches) -> 1 + (nb_noeuds_foret branches)
and nb_noeuds_foret = function
                |  [] -> 0
                |  t :: q -> (nb_noeuds_arbre t) + (nb_noeuds_foret  q);;
#let rec nb_feuilles_arbre = function
                   Feuille _ -> 1
                |  Noeud ( _ , branches) -> (nb_feuilles_foret branches)
and nb_feuilles_foret = function
                   [] -> 0
                |  t :: q -> (nb_feuilles_arbre t) + (nb_feuilles_foret  q);;
#let rec hauteur_arbre = function
                |  Feuille _ -> 0
                |  Noeud ( _ , branches) -> 1 + (hauteur_foret branches)
and hauteur_foret = function
                |  [] -> 0
                |  t :: q -> max (hauteur_arbre t) (hauteur_foret  q);;
```

```
(* nombre de noeuds, de feuilles, hauteur d'un arbre hétérogène (tableaux) *)
#type ('f,'n) arbre=
                  | Feuille of 'f
                  | Noeud of 'n * (('f,'n) arbre vect);;

#let rec nb_noeuds_arbre = function
                  |  Feuille _ -> 0
                  |  Noeud ( _ , branches) -> 1 + (nb_noeuds_foret branches)
and nb_noeuds_foret f =
        let nb = ref 0 in (* on initialise à 0 le nombre de noeuds de la forêt *)
        for i = 0 to (vect_length f) -1 do
            (* on ajoute le nombre de noeuds *)
             nb := !nb + nb_noeuds_arbre (f.(i))
        done;
        !nb ;;

#let rec nb_feuilles_arbre = function
                     Feuille _ -> 1
                  |  Noeud ( _ , branches) -> (nb_feuilles_foret branches)
and nb_feuilles_foret f =
          let nb = ref 0 in
               (* on initialise à 0 le nombre de feuilles de la forêt *)
```

```
          for i = 0 to (vect_length f) -1 do
               (* on ajoute le nombre de feuilles *)
                nb := !nb + nb_feuilles_arbre (f.(i))
          done;
          !nb ;;

#let rec hauteur_arbre = function
                  |  Feuille _ -> 0
                  |  Noeud ( _ , branches) -> 1 + (hauteur_foret branches)
and hauteur_foret f =
        let nb = ref 0 in
           (* on initialise à 0 le nombre de feuilles de la forêt *)
        for i = 0 to (vect_length f) -1 do
                (* on ajoute le nombre de feuilles *)
                nb := max !nb  (hauteur_arbre f.(i))
        done;
        !nb ;;
```

```
(* nombre de noeuds, de feuilles, hauteur d'un arbre binaire hétérogène *)
#type ('f,'n) arbre_bin =
                  | Feuille of 'f
                  | Noeud of (('f,'n) arbre_bin * 'n * ('f,'n) arbre_bin);;

#let rec nb_noeuds_arbre = function
                  |  Feuille _ -> 0
                  |  Noeud ( gauche, _ , droite) ->
                       1 + (nb_noeuds_arbre gauche)+(nb_noeuds_arbre droite) ;;

#let rec nb_feuilles_arbre = function
                  |  Feuille _ -> 1
                  | Noeud ( gauche, _ , droite) ->
                      (nb_feuilles_arbre gauche) + (nb_feuilles_arbre droite) ;;

#let rec hauteur_arbre = function
                  |  Feuille _ -> 0
                  |  Noeud ( gauche, _ , droite) ->
                        1 + max (hauteur_arbre gauche) (hauteur_arbre droite);;
```

```
(* parcours préfixé, postfixé ou infixé d'un arbre hétérogène (listes) *)
#type ('f,'n) arbre=
                  | Feuille of 'f
                  | Noeud of 'n * (('f,'n) arbre list);;

#let parcours_prefixe explore_feuille explore_noeud =
        let rec prefixe_arbre = function
                  | Feuille f -> explore_feuille f
                  | Noeud (n,foret) -> explore_noeud n; prefixe_foret foret
        and prefixe_foret = function
                  | [] -> ()
                  | t::q -> prefixe_arbre t; prefixe_foret q
        in prefixe_arbre;;

#let parcours_postfixe explore_feuille explore_noeud =
        let rec postfixe_arbre = function
                  | Feuille f -> explore_feuille f
                  | Noeud (n,foret) -> postfixe_foret foret; explore_noeud n
        and postfixe_foret = function
                  | [] -> ()
                  | t::q -> postfixe_arbre t; postfixe_foret q
        in postfixe_arbre;;
```

```
#let parcours_infixe explore_feuille explore_noeud =
        let rec infixe_arbre = function
                  | Feuille f -> explore_feuille f
                  | Noeud (n,foret) -> infixe_foret n foret
        and infixe_foret n = function
                  | [] -> failwith "arbre incorrect"
                  | [branche] -> infixe_arbre branche
                  | t::q -> infixe_arbre t; explore_noeud n; infixe_foret n q
        in infixe_arbre;;

##let a = Noeud (‘a‘,[Noeud (‘b‘,[Noeud (‘d‘,[Feuille 2;Feuille 4]);Feuille 1]);
                      Feuille 5; Noeud (‘c‘,[Feuille 3])])
and explore_feuille = print_int and explore_noeud = print_char in
parcours_prefixe explore_feuille explore_noeud a;;

abd2415c3- : unit = ()

##let a = Noeud (‘a‘,[Noeud (‘b‘,[Noeud (‘d‘,[Feuille 2;Feuille 4]);Feuille 1]);
                     Feuille 5; Noeud (‘c‘,[Feuille 3])])
and explore_feuille = print_int and explore_noeud = print_char in
parcours_postfixe explore_feuille explore_noeud a;;

24d1b53ca- : unit = ()
```

Parcours générique

```
#let parcours explore_feuille pre_explore_noeud in_explore_noeud post_explore_noeud=
        let rec parcours_arbre = function
                  | Feuille f -> explore_feuille f
                  | Noeud (n,foret) -> pre_explore_noeud n; parcours_foret n foret
        and parcours_foret n = function
                  | [] -> failwith "arbre incorrect"
                  | [branche] -> parcours_arbre branche; post_explore_noeud n
                  | t::q -> parcours_arbre t; in_explore_noeud n; parcours_foret n q
        in parcours_arbre;;
```

Un exemple intéressant qui imprime la syntaxe concrète d'un arbre

```
#let a = Noeud (‘a‘,[Noeud (‘b‘,[Noeud (‘d‘,[Feuille 2;Feuille 4]);Feuille 1]);
                     Feuille 5; Noeud (‘c‘,[Feuille 3])])
and explore_feuille = print_int
and pre_explore_noeud n = print_char n; print_char ‘(‘
and in_explore_noeud n = print_char ‘,‘
and post_explore_noeud n = print_char ‘)‘ in
parcours explore_feuille pre_explore_noeud in_explore_noeud post_explore_noeud a;;

a(b(d(2,4),1),5,c(3))- : unit = ()
```

Un exemple amusant qui imprime la syntaxe Caml de l'arbre Caml (!!)

```
#let a = Noeud (‘a‘,[Noeud (‘b‘,[Noeud (‘d‘,[Feuille 2;Feuille 4]);Feuille 1]);
                     Feuille 5; Noeud (‘c‘,[Feuille 3])])
and explore_feuille f = print_string "Feuille ";print_int f
and pre_explore_noeud n = print_string "Noeud (‘";print_char n; print_string "‘,["
and in_explore_noeud n = print_char ‘;‘
and post_explore_noeud n = print_string "])" in
parcours explore_feuille pre_explore_noeud in_explore_noeud post_explore_noeud a;;

Noeud (‘a‘,[Noeud (‘b‘,[Noeud (‘d‘,[Feuille 2;Feuille 4]);Feuille 1]);Feuille 5;
Noeud (‘c‘,[Feuille 3])]) - : unit = ()
```

Parcours d'arbres binaires

```
#type ('f,'n) arbre_bin =
                  | Feuille of 'f
                  | Noeud of (('f,'n) arbre_bin * 'n * ('f,'n) arbre_bin);;
let parcours_prefixe explore_feuille explore_noeud =
          let rec prefixe = function
                  |  Feuille f -> explore_feuille f
                  |  Noeud ( gauche, n , droite) ->
                        explore_noeud n; prefixe gauche; prefixe droite
          in prefixe;;
let parcours_postfixe explore_feuille explore_noeud =
          let rec postfixe = function
                  |  Feuille f -> explore_feuille f
                  |  Noeud ( gauche, n , droite) ->
                        postfixe gauche; postfixe droite; explore_noeud n
          in postfixe;;
let parcours_infixe explore_feuille explore_noeud =
          let rec infixe = function
                  |  Feuille f -> explore_feuille f
                  |  Noeud ( gauche, n , droite) ->
                        infixe gauche; explore_noeud n; infixe droite
          in infixe;;
```

Théorème 0.1 *1. Arbre général : # aretes = # noeuds+# feuilles -1*

2. Arbre binaire :

(a) # *feuilles* = # *noeuds +1*

(b) # *noeuds de profondeur* $i \leq 2^i$

(c) # *feuilles* $\leq 2^{hauteur}$

(d) $hauteur \geq \log_2(\#\ feuilles\ )$

(e) $hauteur \geq \log_2(\#\ noeuds\ ) - 1$

(f) $hauteur \leq \#\ noeuds = \#\ feuilles\ -1$

Soit A un arbre binaire pris au hasard possédant n noeuds. Nous allons estimer la longueur moyenne $L(n)$ d'un chemin allant de la racine à un noeud x pris au hasard. Soit a_0 la racine de l'arbre, A_1 et A_2 les deux branches issues de a_0. Supposons que A_1 possède i noeuds ; en conséquence A_2 possède $n-i-1$ noeuds. Il y a i chances que x soit un noeud de A_1 avec une longueur moyenne de chemin égale à $L(i)+1$, $n-i-1$ chances que x soit un noeud de A_2 avec une longueur moyenne de chemin égale à $L(n-i-1)+1$ et enfin 1 chance que x soit égale à a_0 avec une longueur de chemin égale à 0. On en déduit que la longueur moyenne du chemin dans le cas où A_1 possède i noeuds est égale à $\frac{1}{n}\Big(i\big(L(i)+1\big)+(n-i-1)\big(L(n-i-1)+1\big)\Big)$. Maintenant, pour un arbre pris au hasard, toutes les valeurs de i de 0 à $n-1$ sont équiprobables et donc la longueur moyenne du chemin d'accès est égale à

$$\begin{aligned}
L(n) &= \frac{1}{n}\sum_{i=0}^{n-1}\frac{1}{n}\Big(i\big(L(i)+1\big)+(n-i-1)\big(L(n-i-1)+1\big)\Big)\\
&= \frac{1}{n^2}\left(\sum_{i=0}^{n-1} iL(i)+\sum_{i=0}^{n-1}(n-i-1)L(n-i-1)+n(n-1)\right)\\
&= \frac{2}{n^2}\sum_{i=0}^{n-1} iL(i)+\frac{n-1}{n} \leq \frac{2}{n^2}\sum_{i=1}^{n-1} iL(i)+1
\end{aligned}$$

puisque $\displaystyle\sum_{i=1}^{n-1} i = \sum_{i=1}^{n-1}(n-i-1) = \frac{n(n-1)}{2}$ et

$$\sum_{i=0}^{n-1} iL(i) = \sum_{i=0}^{n-1}(n-i-1)L(n-i-1)$$

(changer i en $n-1-i$). Nous allons montrer par récurrence sur n que $L(n) \leq 4\log n$. C'est vrai pour $n=1$ et pour $n=2$. Supposons donc l'inégalité vérifiée pour $i = 1, \ldots, n-1$. On a alors

$$L(n) \leq \frac{8}{n^2}\sum_{i=1}^{n-1} i\log(i) + 1$$

Comme la fonction $x \mapsto x\log x$ est croissante sur $[1, +\infty[$, on a

$$\forall i \in \mathbf{N},\ i\log i \leq \int_i^{i+1} t\log t\ dt$$

d'où

$$L(n) \leq \frac{8}{n^2}\int_1^n t\log t\ dt + 1 = \frac{8}{n^2}\left(\frac{n^2}{2}\log n - \frac{n^2}{4} + \frac{1}{4}\right) + 1 \leq 4\log n$$

dès que $n \geq 3$, ce qui achève la récurrence.