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>
```