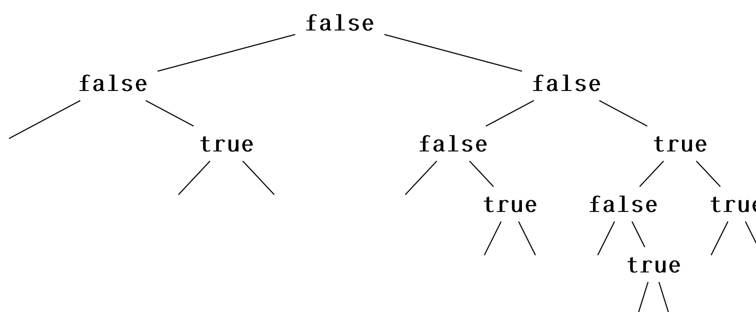
Corrigé du contrôle : représentation d'ensembles par arbres radix

Question 1. L'ensemble $\{2, 3, 5, 7, 11\}$ est représenté par l'arbre suivant :



Question 2. L'arbre présenté dans l'énoncé ne peut être un arbre radix car il possède un nœud étiqueté par `true` et qui est le *fils gauche* d'un autre nœud ; or hormis 0, aucun entier n'a une écriture binaire qui débute par un 0.
Pour être un arbre radix, il est nécessaire qu'aucun fils gauche ne soit étiqueté par le booléen `true`.

Question 3. Notons que si $[b_p, b_{p-1}, \dots, b_1, b_0]_2$ désigne la décomposition en base 2 d'un entier n, alors la quotient et le reste de la division euclidienne de n par 2 valent respectivement $[b_p, b_{p-1}, \dots, b_1]_2$ et b_0. Pour tester l'appartenance d'un entier à un ensemble, on définit donc la fonction :

```
let rec cherche n = function
  | Nil                             -> false    (* cas de l'ensemble vide *)
  | Noeud (b, fg, fd) when n = 0     -> b
  | Noeud (b, fg, fd) when n mod 2 = 0 -> cherche (n / 2) fg
  | Noeud (b, fg, fd)               -> cherche (n / 2) fd ;;
```

(Quand n est un entier pair, il faut le chercher dans le sous-arbre gauche ; quand il est impair dans le sous-arbre droit.)

Question 4. Pour ajouter un entier à un ensemble, on définit la fonction :

```
let rec ajoute n = function
  | Nil when n = 0                  -> Noeud (true, Nil, Nil)
  | Nil when n mod 2 = 0            -> Noeud (false, ajoute (n / 2) Nil, Nil)
  | Nil                             -> Noeud (false, Nil, ajoute (n / 2) Nil)
  | Noeud (b, fg, fd) when n = 0     -> Noeud (true, fg, fd)
  | Noeud (b, fg, fd) when n mod 2 = 0 -> Noeud (b, ajoute (n / 2) fg, fd)
  | Noeud (b, fg, fd)               -> Noeud (b, fg, ajoute (n / 2) fd) ;;
```

Question 5. Pour construire un ensemble, il suffit maintenant d'ajouter un à un les éléments à l'ensemble vide :

```
let rec construit = function
  | []   -> Nil
  | t::q -> ajoute t (construit q) ;;
```

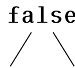
ou de manière plus concise :

```
let construit = it_list (fun ens n -> ajoute n ens) Nil ;;
```

Question 6. Pour supprimer un élément, on définit la fonction :

```
let rec supprime n = function
  | Nil                             -> Nil
  | Noeud (b, fg, fd) when n = 0     -> Noeud (false, fg, fd)
  | Noeud (b, fg, fd) when n mod 2 = 0 -> Noeud (b, supprime (n / 2) fg, fd)
  | Noeud (b, fg, fd)               -> Noeud (b, fg, supprime (n / 2) fd) ;;
```

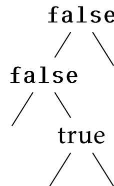
Le problème est qu'en supprimant un élément, on risque de ne plus maintenir l'invariant exigé en faisant apparaître un noeud de la forme



là où l'élément a été supprimé. Ajouter le motif suivant :

```
Noeud (true, Nil, Nil) -> Nil
```

ne résout pas le problème, car la disparition d'un élément peut entraîner la nécessité d'élaguer toute une branche de l'arbre. Par exemple, supprimer l'élément 2 de l'ensemble {2} doit transformer l'arbre suivant en l'arbre vide :



J'ai donc choisi d'ajouter une fonction `elague` de type *arbre -> arbre* dont le but est d'élaguer toute branche morte. Pour supprimer un élément en préservant l'invariant exigé, il faudra donc composer les fonctions `elague` et `supprime`.

```
let rec elague = function
  | Nil                    -> Nil
  | Noeud (false, Nil, Nil) -> Nil
  | Noeud (false, fg, fd) when (elague fg = Nil && elague fd = Nil) -> Nil
  | Noeud (b, fg, fd)       -> Noeud (b, elague fg, elague fd) ;;
```

Question 7. On réunit deux ensembles de la façon suivante :

```
let rec union = fun
  | Nil ens                                     -> ens
  | ens Nil                                     -> ens
  | (Noeud (b1, fg1, fd1)) (Noeud (b2, fg2, fd2)) ->
                  Noeud (b1 || b2, union fg1 fg2, union fd1 fd2) ;;
```

Question 8. L'intersection se traite de la même manière que la réunion :

```
let rec intersection = fun
  | Nil ens                                     -> Nil
  | ens Nil                                     -> Nil
  | (Noeud (b1, fg1, fd1)) (Noeud (b2, fg2, fd2)) ->
                  Noeud (b1 && b2, intersection fg1 fg2, intersection fd1 fd2) ;;
```

mais là encore se pose le problème de l'apparition d'éventuelles branches mortes. Pour respecter l'invariant, il faudra donc faire suivre cette fonction de la fonction `elague`.

Question 9. On obtient la liste des éléments d'un ensemble de la façon suivante :

```
let elements =
  let rec aux p n = function
    | Nil                  -> []
    | Noeud (false, fg, fd) -> (aux (2 * p) n fg) @ (aux (2 * p) (n + p) fd)
    | Noeud (true, fg, fd)  -> n::(aux (2 * p) n fg) @ (aux (2 * p) (n + p) fd)
  in aux 1 0 ;;
```

Cette fonction utilise deux accumulateurs p et n. Lorsqu'elle examine un nœud de profondeur h, p contient la valeur 2^h et n l'entier éventuellement stocké au niveau de ce nœud.

Question 10. L'invariant imposé par l'énoncé assure l'unicité de la représentation d'un ensemble par un arbre radix. On peut donc tester l'égalité de deux ensembles en se contentant de tester l'égalité de valeur des arbres associés :

```
let egal ens1 ens2 = (elague ens1) = (elague ens2) ;;
```