Corrigé

option informatique

Représentation d'ensembles par arbres radix

Question 1.

```
let rec cherche n = function
  | Nil                                -> false
  | 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 ;;
```

Question 2.

```
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 3.

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

ou avec une fonctionnelle :

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

Question 4.

```
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) ;;
```

Question 5.

```
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 6.

```
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) ;;
```

Question 7.

```
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 ;;
```

Le paramètre p désigne la puissance de 2 correspondant à la profondeur du nœud exploré (si le nœud est à la profondeur k alors $p = 2^k$) et le paramètre n à l'entier associé au nœud.

On pourrait éviter le recours à la concaténation en ajoutant un accumulateur pour « transporter » la liste des éléments déjà trouvés dans l'ensemble, ce qui donnerait la version suivante :

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

Question 8.

```
let rec elague = function
  | Nil                   -> Nil
  | Noeud (true, fg, fd)  -> Noeud (true, elague fg, elague fd)
  | Noeud (false, fg, fd) -> let fge = elague fg and fde = elague fd in
                             if fge = Nil && fde = Nil then Nil else Noeud(false, fge, fde) ;;
```