1) La fonction récursive s'écrit facilement :

```
let rec taille = function
  | [] -> 0
  | _ :: q -> 1 + taille q
;;
```

2) On utilise un accumulateur qui commencera la liste vide `[]` et contiendra la fin toutes les valeurs de la liste `l` insérées dans l'ordre inverse :

```
let miroir l =
  let rec parcours accu = function
    | [] -> accu
    | t::q -> parcours (t::accu) q
  in parcours [] l
;;
```

3) Pour les fonctions suivantes, on fait comme le suggère l'énoncé et on force le type des arguments :

```
let new_file () : 'a file = [], [];;
```

4) Une file est vide si et seulement si les deux listes `arrive` et `repart` sont vides :

```
let est_vide (f : 'a file) = (f = ([], []));;
```

5) On peut utiliser `fst` et `snd` pour récupérer `arrive` et `repart` :

```
let enfile x (f : 'a file) : 'a file = (x::(fst f), snd f);;
```

On peut aussi utiliser un `match ... with` :

```
let enfile x (f : 'a file) : 'a file =
  match f with
  | arrive, repart -> (x :: arrive), repart
```

6) Le sujet ne précisait rien, mais on peut utiliser `failwith "File vide"` pour renvoyer une erreur si les deux listes `repart` et `arrive` sont vides :

```
let defile (f : 'a file) : 'a * 'a file =
  match f with
  | a, t::q -> t, (a, q)  (* s'il y a une valeur dans repart *)
  | a, [] -> match miroir a with
             | [] -> failwith "File vide"
             | t::q -> t, ([], q)
;;
```

8) On ajoute les tailles des deux listes :

```
let longueur (f : 'a file) = match f with
    a, r -> (taille a) + (taille r)
;;
```

9) On se rappelle que `^` fait la concaténation de deux chaines de caractères :

```
let elements (f : int file) =
  let rec ecrire = function
    | [] -> ""
    | a::[] -> string_of_int a
    | t::q -> string_of_int t ^ ">>>" ^ (ecrire q)
  in ecrire (miroir (fst f) @ snd f)
;;
```

10) `let arbre_ex = N(N(N(V, 2, V), 15, N(V, 22, V)), 28, N(V, 41, V));;`

On peut aussi écrire une fonction `feuille` qui simplifie l'écriture de `N(V, x, V)` :

```
let feuille x = N(V, x, V);;
let arbre_ex = N(N(feuille 2, 15, feuille 22), 28, feuille 41);;
```

11) Les deux fonctions ont la même forme, la `hauteur` utilise le `max` des hauteurs des fils gauche et droite, et la `taille` utilise la somme :

```
let rec hauteur = function
  | V -> -1
  | N (g, _, d) -> 1 + max (hauteur g) (hauteur d)
;;

let rec taille = function
  | V -> 0
  | N (g, _, d) -> 1 + (taille g) + (taille d)
;;
```

12) C'est comme `somme` mais avec `e +` si on trouve un nœud étiquetté par `e` au lieu de `1 +` :

```
let rec somme = function
  | V -> 0
  | N (g, e, d) -> e + (somme g) + (somme d)
;;
```

13) On ne peut pas faire autrement que d'utiliser une concaténation de listes (`@`) :

```
let rec liste_etiquettes = function
  | V -> []
  | N (g, e, d) -> e :: (liste_etiquettes g) @ (liste_etiquettes d)
;;
```

14) On utilise un `match ... with` avec un `when e = f` :

```
let rec descendants e = function
  | V -> []
  | N (_, f, _) as a when e = f -> liste_etiquettes a
  | N (g, _, d) -> (descendants e g) @ (descendants e d)
;;
```

15) C'est plus compliqué. Une façon de l'écrire, avec une fonction récursive, est la suivante :

```
let hamming n =
  let rec aux h2 h3 h5 n =
    if n=0 then []
    else
```

```
      let e2, hp2 = defile h2 in
      let e3, hp3 = defile h3 in
      let e5, hp5 = defile h5 in
      let e = min (min e2 e3) e5 in
      e::(aux (enfile (2*e) (if e=e2 then hp2 else h2))
              (enfile (3*e) (if e=e3 then hp3 else h3))
              (enfile (5*e) (if e=e5 then hp5 else h5))
              (n-1))
  in aux (enfile 1 (new_file())) (enfile 1 (new_file()))
         (enfile 1 (new_file())) n
;;
```

16) La suite est plus compliquée.

```
type dict = char arbre;;

let rec estDansDict mot (db : dict) = match (mot, db) with
  | (_, V) -> false
  | ([], N (gauche, lettre, droit)) -> lettre = '$'
  | (l::fin_mot, N (gauche, lettre, droit)) ->
      if l = lettre then estDansDictBin fin_mot gauche
      else estDansDictBin mot droit
;;

let contenuDict (db : dict) =
  let rec parcours prefixe = function
    | V -> []
    | N (gauche, lettre, droit) ->
        if lettre = '$'
        then (miroir prefixe)::(parcours prefixe droit)
        else (parcours (lettre::prefixe) gauche) @ (parcours prefixe droit)
  in parcours [] db
;;

let rec ajoutDict mot (db : dict) : dict =
  match mot, db with
  | ([], V) -> N (V, '$', V)
  | ([], N (gauche, lettre, droit)) ->
     if lettre = '$' then db else N (V, n, db)
  | (car::fin_mot, V) -> N (ajoutDict fin_mot V, car, V)
  | (car::fin_mot, N (gauche, lettre, droit)) ->
     if lettre = car
     then N (ajoutDict fin_mot gauche, lettre, droit)
     else N (gauche, lettre, ajoutDict mot droit)
;;
```