1 Arbres binaires en C

Exercice 1 (Fonctions élémentaires) :

1.
```c
node *node_new(int key){
  node *n = malloc(sizeof(node));
  n->left = NULL;
  n->right = NULL;
  n->key = key;
  return n;
}
```

2.
```c
bst *bst_new(void){
  bst *t = malloc(sizeof(bst));
  t->root = NULL;
  t->size = 0;
}
return t;
```

3.
```c
void node_free(node *n){
  if (n != NULL) {
    node_free(n->left);
    node_free(n->right);
    free(n);
  }
}
```

4.
```c
void bst_free(bst *t){
  node_free(t->root);
  free(t);
}
```

Exercice 2 (Création et libération) :

5.
```c
bool node_member(node *n, int key){
  if (n == NULL) return false;
  if (key < n->key) return node_member(n->left, key);
  if (key > n->key) return node_member(n->right, key);
  return true;
}
```

6. (a)
```c
bool bst_member(bst *t, int key){
  return node_member(t->root, key);
}
```

(b) La complexité est en $O(h)$, où h est la hauteur de l'arbre.

7.
```c
node *node_insert(node *n, int key){
  if (n == NULL) return node_new(key);
  if (key < n->key) n->left = node_insert(n->left, key);
  if (key > n->key) n->right = node_insert(n->right, key);
  return n;
}
```

8. (a)
```c
void bst_insert(bst *t, int key){
  if (bst_member(t, key)) return;
  t->root = node_insert(t->root, key);
  t->size++;
}
```

(b) La complexité est en $O(h)$, où h est la hauteur de l'arbre.

Exercice 3 (Fonctions élémentaires) :

9.
```c
void node_to_array(node *n, int arr[], int *start_pointer){
  if (n == NULL) return;
  node_to_array(n->left, arr, start_pointer);
  arr[*start_pointer] = n->key;
  *start_pointer += 1;
  node_to_array(n->right, arr, start_pointer);
}
```

10.
```c
int *bst_to_array(bst *t){
  int *arr = malloc(t->size * sizeof(int));
  int start = 0;
  node_to_array(t->root, arr, &start);
  return arr;
}
```

11. Pour avoir un arbre équilibré à partir du tableau trié, il suffit de choisir le milieu du tableau pour la racine.

```c
node *node_balanced(int arr[], int start, int end){
  if (start >= end) return NULL;
  int mid = (start + end) / 2;
  node *n = node_new(arr[mid]);
  n->left = node_balanced(arr, start, mid);
  n->right = node_balanced(arr, mid + 1, end);
  return n;
}
```

12. (a)
```c
void bst_rebalance(bst *t){
  int *elements = bst_to_array(t);
  node_free(t->root);
  t->root = node_balanced(elements, 0, t->size);
  free(elements);
}
```

(b) L'appel `bst_to_array` effectue un simple parcours de l'arbre, avec un travail constant à chaque nœud. On fait un autre parcours pour `node_free`, et ensuite l'appel à `node_balanced` se fait également en temps proportionnel au nombre d'éléments (au total, il y a un appel récursif par nœud de l'arbre construit, et chacun de ces appels se fait en temps constant).
On a donc une complexité totale en $O(n)$, où n est la taille de l'arbre.

2 Arbres parfaits et ensemble de nombres

Exercice 4 :

13.
```
let rec hauteur_parfait t = match t with
  | E(_) -> 0
  | N(g,d) ->
    let x = hauteur_parfait g in
    let y = hauteur_parfait d in
    if x != y then raise Not_perfect
    else x+1
;;
```

14.
```
let est_parfait t =
  try
    let _ = hauteur_parfait t in true
  with Not_perfect -> false
;;
```

15.
```
let rec mem x t = match x, t with
  | 0, E(b) -> b
  | x, N(g,d) when x mod 2 = 0 -> mem (x/2) g
  | x, N(g,d) -> mem (x/2) d
  | _, _ -> false
;;
```

16.
```
let rec increase_height t = match t with
  | E(b) -> N(E(b), E(false))
  | N(g, d) -> N(increase_height g, increase_height d)
;;
```

17.
```
let rec insert_exn t x = match x, t with
  | 0, E(b) -> E(true)
  | _, E(_) -> raise No_fit
  | x, N(g,d) when x mod 2 = 0 -> N(insert_exn g (x/2), d)
  | x, N(g,d) ->  N(g, insert_exn d (x/2))
;;
```

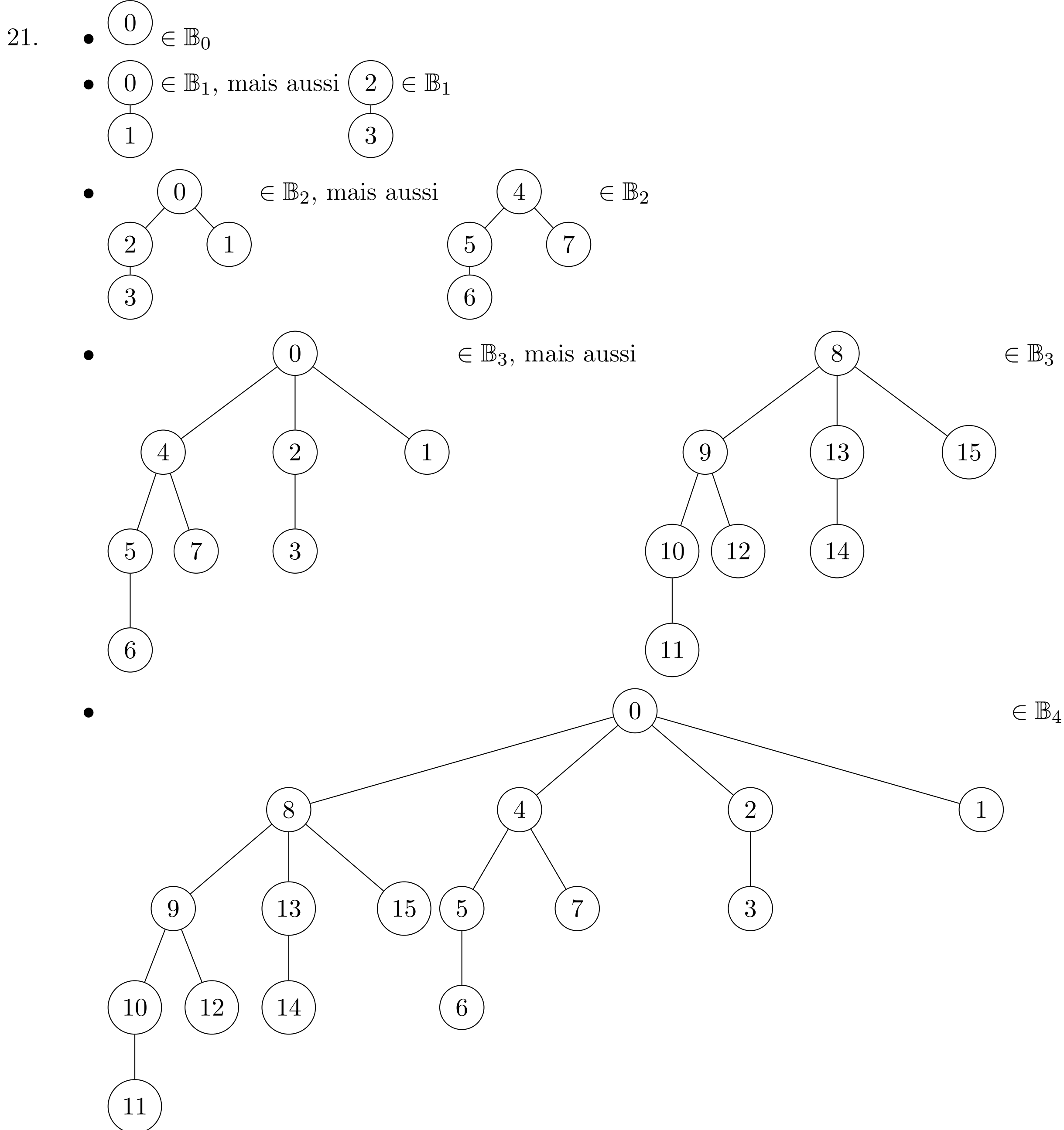
18.
```
let rec insert x t =
  try
    insert_exn t x
  with No_fit -> insert x (increase_height t)
;;
```

19.
```
let fold f t acc =
  let rec aux t v pow2 acc = match t with
    | E(true) -> f v acc
    | E(false) -> acc
    | N(g,d) ->
      let acc1 = aux g (v) (2 * pow2) acc in
      let acc2 = aux d (v + pow2) (2 * pow2) acc1 in
      acc2
  in aux t 0 1 acc
;;
```

20.
```ocaml
let union t1 t2 = fold insert t1 t2 ;;
```

3 Arbres binomiaux

Exercice 5 (Définitions, vocabulaires, propriétés) :

21. 

22. Montrons que $\forall k \in \mathbb{N}, \forall b \in \mathbb{B}_k, \mathtt{s}(b) = 2^k$.
    - $\forall b \in \mathbb{B}_0, \mathtt{s}(b) = 1 = 2^0$.
    - Supposons que $\forall b \in \mathbb{B}_k, \mathtt{s}(b) = 2^k$, soit $b \in \mathbb{B}_{k+1}$ alors il existe $b_1 = \mathtt{GN}(s_1, l_1) \in \mathbb{B}_k$ et $b_2 \in \mathbb{B}_k$ tels que $b = \mathtt{GN}(s_1, b_2 :: l_1)$.
      On a alors : $\mathtt{s}(b) = 1 + \mathtt{sl}(b_2 :: l_1) = 1 + \mathtt{s}(b_2) + \mathtt{sl}(l_1) = \mathtt{s}(\mathtt{GN}(s_1, l_1)) + \mathtt{s}(b_2) = \mathtt{s}(b_1) + \mathtt{s}(b_2) = 2^{k+1}$.
23. Montrons que $\forall k \in \mathbb{N}, \forall b \in \mathbb{B}_k, \mathtt{h}(b) = k$.
    - $\forall b \in \mathbb{B}_0, \mathtt{h}(b) = 0$.
    - Supposons que $\forall b \in \mathbb{B}_k, \mathtt{h}(b) = k$, soit $b \in \mathbb{B}_{k+1}$ alors il existe $b_1 = \mathtt{GN}(s_1, l_1) \in \mathbb{B}_k$ et $b_2 \in \mathbb{B}_k$ tels que $b = \mathtt{GN}(s_1, b_2 :: l_1)$.

On a alors : $\texttt{h}(b) = 1 + \texttt{hl}(b_2 :: l_1) = 1 + \max(\texttt{h}(b_2), \texttt{hl}(l_1)) = \max(1 + \texttt{h}(b_2), 1 + \texttt{hl}(l_1)) = \max(1 + \texttt{h}(b_2), \texttt{h}(b_1)) = k + 1$.

24. Montrons que $\forall k \in \mathbb{N}, \forall b \in \mathbb{B}_k, f(b) = \lceil 2^{k-1} \rceil$.
    - $\forall b \in \mathbb{B}_0, f(b) = 1 = \lceil 2^{-1} \rceil$.
    - Supposons que $\forall b \in \mathbb{B}_k, f(b) = \lceil 2^{k-1} \rceil$, soit $b \in \mathbb{B}_{k+1}$ alors il existe $b_1 = \texttt{GN}(s_1, l_1) \in \mathbb{B}_k$ et $b_2 \in \mathbb{B}_k$ tels que $b = \texttt{GN}(s_1, b_2 :: l_1)$.
      On a alors : $f(b) = f(b_2) + f(\texttt{GN}(s_1, l_1)) = 2^k$.
25. Par induction sur $\mathbb{N}$:
    - $\mathbb{M}_0 = \mathbb{B}_0$.
    - Supposons que $\forall i \in [\![0, k]\!], \mathbb{M}_i = \mathbb{B}_i$, alors :
      - Soit $x \in \mathbb{M}_{k+1}$,
        $x = \texttt{GN}(s, t_k :: t_{k-1} :: \ldots :: t_0)$ avec $\forall i \in [\![0, k]\!], t_i \in \mathbb{M}_i$, on a donc que $gnodest_{k-1} :: \ldots :: t_0 \in \mathbb{M}_k = \mathbb{B}_k$ et $t_k \in \mathbb{M}_\neg = \mathbb{B}_k$ et donc $\texttt{GN}(s, t_k :: t_{k-1} :: \ldots :: t_0) \in \mathbb{B}_{k+1}$.
      - Soit $x \in \mathbb{B}_{k+1}$, $x = \texttt{GN}(s, b :: l)$ avec $\texttt{GN}(s, l) \in \mathbb{B}_k = \mathbb{M}_k$ donc $l = t_{k-1} :: t_{k-2} :: \ldots :: t_0$ avec $\forall i \in [\![0, k-1]\!], t_i \in \mathbb{M}_i$, finalement $\texttt{GN}(s, b :: t_{k-1} :: t_{k-2} :: \ldots :: t_0) \in \mathbb{M}_{k+1}$.

    On a donc $\mathbb{M}_{k+1} = \mathbb{B}_{k+1}$.
26. Si $t \in \mathbb{B}_k$ alors $t \in \mathbb{M}_k$, donc la racine de t a exactement k fils (cf. définition de $\mathbb{M}_k$).

Exercice 6 (Fabrication d'arbres binomiaux) :

27.
```
let rec est_binomial g k = match g with
  | GN(x, []) -> k = 0
  | GN(x, y::q) -> (est_binomial y (k-1)) && (est_binomial (GN(x, q)) (k-1))
;;
```

28.
```
let rec map f b = match b with
  | GN(x, l) -> GN(f x, mapl f l)
and mapl f l = match l with
  | x :: xs -> (map f x) :: (mapl f xs)
  | [] -> []
;;
```

Ou alors, en utilisant `List.map` :

```
let rec map f b = match b with
  | GN(x, l) -> GN(f x, List.map (map f) l)
;;
```

29.
```
let rec pow2 n = match n with
  | 0 -> 1
  | _ -> 2 * pow2 (n-1)
;;

let rec generate k =
  if k = 0 then GN(0, [])
  else
    let x = generate (k-1) in
    let x' = map (fun i -> i + pow2 (k-1)) x in
    match x with
    | GN(r, l) -> GN(r, x'::l)
;;
```

Exercice 7 (Tas binomiaux) :

30. Une **file de priorité** est une structure abstraite, supportant les opérations suivantes :
    - **création** d'une file de priorité vide ;
    - test d'**égalité** au vide ;
    - **retrait** de l'élément de plus grande priorité d'une file non vide ;
    - **ajout** d'un élément avec une priorité donnée ;

31. $\mathsf{inserer}(f, e) = \mathsf{union}(f, \mathsf{singleton}(e))$.

32.
```
let rec est_tournoi g =
  let rec aux borneinf m =
    let GN(x, l) = m in
    x >= borneinf && List.for_all (aux x) l
  in
  let GN(x, l) = g in
  List.for_all (aux x) l
;;
```

33.
```
let rec est_tas_bin t = match t with
  | [] -> true
  | (o,g)::[] -> est_binomial g o && est_tournoi g
  | (o,g)::(o',g')::q -> o < o'
                         && est_binomial g o && est_tournoi g
                         && est_tas_bin ((o',g')::q)
;;
```

34.
```
let empty = [] ;;
```

35. (a)
```
let minimum t =
  let rec aux t m = match t with
    | [] -> m
    | GN(x,l)::q when x < m -> aux q x
    | _::q -> aux q m
  in match t with
  | [] -> failwith "tas vide"
  | GN(x,l)::q -> aux q x
;;
```

(b) La complexité est en $O(n)$, où n est le nombre d'arbres se trouvant dans le tas binomial.

36. (a)
```
let union_binom_tournoi x y =
  let GN(a, la) = x in
  let GN(b, lb) = y in
  if a < b then GN(a, y::la)
  else GN(b, x::lb)
;;
```

(b) La complexité est en $O(1)$.