Exercice 1. Questions de cours

Question 1 –

```
let echange tas i j =
  let tmp = tas.elem.(i) in
  tas.elem.(i) <- tas.elem.(j);
  tas.elem.(j) <- tmp;;
```

Question 2 –

```
let make m e = {n = 0; elem = Array.make m e};;
```

Question 3 –

```
let perco_haut (fp: 'a file_prio) i0 =
  let i = ref i0 and
      i_pere = ref ((i0-1)/2) in
  while !i > 0 && snd fp.elem.(!i) > snd fp.elem.(!i_pere) do
    echange fp !i !i_pere;
    i := !i_pere;
    i_pere := (!i-1)/2;
  done;;
```

Question 4 –

```
(* On note i_fg et i_fd les indices du fils gauche et du fils droit de i.
 * La fonction max_avec_fils renvoie l'élémént de {i, i_fg, i_fd} dont
 * la priorité est maximale. *)
let max_avec_fils (fp: 'a file_prio) i =
  let i_max = ref i in
  let i_fg = 2*i + 1 in
  if i_fg < fp.n then begin
      if snd fp.elem.(i_fg) > snd fp.elem.(!i_max) then i_max := i_fg;
      let i_fd = 2*i + 2 in
      if i_fd < fp.n && snd fp.elem.(i_fd) > snd fp.elem.(!i_max) then
        i_max := i_fd
    end;
  !i_max;;
```

```
let perco_bas (fp: 'a file_prio) i0 =
  let i = ref i0 and
      j = ref (max_avec_fils fp i0) in
  while !i <> !j do
    echange fp !i !j;
    i := !j;
    j := max_avec_fils fp !i;
  done;;
```

Question 5 –

```
let ajout (fp: 'a file_prio) v p =
  if fp.n = Array.length fp.elem then failwith "ajout: file de priorité pleine";
  fp.elem.(fp.n) <- (v,p);
  fp.n <- fp.n + 1;
  perco_haut fp (fp.n-1);;
```

Question 6 –

```
let extrait_max (fp: 'a file_prio) =
  if fp.n = 0 then failwith "extrait_max: File de priorié vide";
  let res = fp.elem.(0) in
  fp.n <- fp.n - 1;
  fp.elem.(0) <- fp.elem.(fp.n);
  perco_bas fp 0;
  res;;
```

Question 7 – On utilise une file de priorité dont les valeurs sont quelconques et les priorités sont les entiers à trier. On choisit par exemple une file de priorité de type `(unit file_prio)` dont toutes les valeurs sont `()`.

```
let fp_of_array tab =
  let n = Array.length tab in
  let (fp: unit file_prio) = make n ((),0) in
  for i = 0 to n - 1 do
    ajout fp () tab.(i);
  done;
  fp;;
```

```
let sorted_array_of_fp (fp: unit file_prio) =
  let tab = Array.make fp.n 0 in
  for i = fp.n - 1 downto 0 do
    let _, p = extrait_max fp in
    tab.(i) <- p
  done;
  tab;;
```

```
let tri_par_tas tab =
  let fp = fp_of_array tab in
  sorted_array_of_fp fp;;
```

Question 8 –

- La fonction `echange` s'exécute en temps constant.
- La fonction `create` s'exécute en temps $\Theta(m)$ où m est la taille du tableau.
- Les fonctions `ajout` et `extrait_max` s'exécutent en temps $\mathcal{O}(h) = \mathcal{O}(\log n)$ où h est la hauteur de l'arbre et n est le nombre de noeuds.
- Le tri par tas s'exécute en temps $\mathcal{O}(n \log n)$ où n est la taille du tableau à trier.

Question 9 – Pour implémenter une file LIFO, on peut prendre une file de priorité et y ajouter un compteur initialisé à 0. Lorsqu'on ajoute un élément à la file de priorité, cet élément a pour priorité la valeur du compteur puis on incrémente le compteur.

Question 10 – Pour implémenter une file FIFO, on procède de la même façon, sauf que la priorité est l'opposé de la valeur du compteur.

Exercice 2. Construction d'un tas en temps linéaire

Question 1 – On construit un tas vide, puis on y ajoute les éléments du tableau les uns après les autres. Puisque l'insertion s'exécute en temps $C_m = \mathcal{O}(\log m)$ où m est le nombre de noeuds dans l'arbre, la complexité totale est en :

$$\sum_{m=0}^{n-1} C_m = \mathcal{O}(n \log n).$$

Question 2 – Il suffit d'appliquer l'opération de percolation vers le bas sur la racine de l'arbre.

Question 3 – Soit f le nombre de feuilles dans l'arbre et m le nombre de noeuds internes alors $F = [\![m, n-1]\!]$. On a 2 cas :

- Le dernier noeud interne dans le parcours en largeur a deux fils. Alors l'arbre est binaire entier donc $f = m + 1$ et $n = 2m + 1$. Donc

$$F = [\![m, n-1]\!] = [\![\lfloor n/2 \rfloor, n-1]\!]$$

- Le dernier noeud interne dans le parcours en largeur a un fils. Alors $f = m$ et $n = 2m$. Donc :

$$F = [\![m, n-1]\!] = [\![\lfloor n/2 \rfloor, n-1]\!]$$

Ainsi $F = [\![\lfloor n/2 \rfloor, n-1]\!]$.

Question 4 –

```
let tas_of_array tab =
  let n = Array.length tab in
  let elem = Array.map (fun p -> ((), p)) tab in
  let fp = {n = n; elem = elem} in
  for i = n/2 - 1 downto 0 do
    perco_bas fp i;
  done;
  {n=n; elem = Array.map snd fp.elem};;
```

Question 5 – Non, par exemple si on part du tableau `[|1; 2; 3|]`, on obtient l'arbre `[|3; 1; 2|]` avec le premier algorithme et l'arbre `[|3; 2; 1|]` avec le deuxième algorithme.

Question 6 – L'algorithme est clairement en $\Omega(n)$. Soit h la hauteur de l'arbre et k un entier avec $0 \leqslant k \leqslant h-1$. La hauteur d'un sous-arbre dont la racine est à profondeur k est au plus $h-k$. De plus, le nombre de sous-arbres dont la racine est à profondeur k est 2^k. Soit $C_\ell = \mathcal{O}(\ell)$ le coût de l'algorithme de la question 2 pour un sous-arbre de hauteur ℓ. Le coût total de la construction est majoré par :

$$\sum_{k=0}^{h-1} C_{h-k} 2^k = \sum_{\ell=1}^{h} C_\ell 2^{h-\ell} = 2^h \sum_{\ell=0}^{h-1} \frac{C_\ell}{2^\ell}$$

Comme l'arbre est presque-complet, $h = \left\lfloor \log_2(n) \right\rfloor$. Ainsi :

$$\log_2(n) - 1 < h \leqslant \log_2(n) \qquad \text{donc} \qquad n/2 < 2^h \leqslant n \qquad \text{donc} \qquad 2^h = \mathcal{O}(n)$$

De plus, $\displaystyle\sum_{\ell=0}^{h-1} \frac{\ell}{2^\ell} \leqslant \sum_{\ell=0}^{+\infty} \frac{\ell}{2^\ell} = \mathcal{O}(1)$ (série convergente), donc $\displaystyle\sum_{\ell=0}^{h-1} \frac{C_\ell}{2^\ell} = \mathcal{O}(1)$.

Finalement, le coût total est en $\mathcal{O}(n)$.

Exercice 3. Quelques questions de programmation

Question 1 –

```
let est_tas tas =
  let i = ref 1 in
  while !i < tas.n &&  tas.elem.(!i) <= tas.elem.((!i-1)/2) do
    incr i;
  done;
  tas.n = 0 || !i = tas.n;;
```

Question 2 –

```
let arbre_of_tas_max tas =
  let rec aux i =
    if i >= tas.n then
      Vide
    else
      N(tas.elem.(i), aux (2*i+1), aux (2*i+2))
  in aux 0;;
```

Question 3 –

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

```
let tas_max_of_arbre arbre = match arbre with
  | Vide -> {n = 0; elem = [||]};
  | N(e, _, _) ->
     let n = nb_noeuds arbre in
     let elem = Array.make n e in
     let rec aux i = function
       | Vide -> ()
       | N(e, g, d) -> elem.(i) <- e;
                       aux (2*i+1) g;
                       aux (2*i+2) d;
     in
     aux 0 arbre;
     {n = n; elem = elem};;
```

Question 4 –

```
let creer_fp liste_array =
  let k = Array.length liste_array in
  let (fp: int file_prio) = make k (0,0) in
  for i = 0 to k - 1 do
    match liste_array.(i) with
    | [] -> ()
    | e::q -> ajout fp i (-e)
  done;
  fp;;
```

```
(* Attention: cette fonction modifie l'argument en entree. A la fin, il
 * ne contient plus que des listes vides.
 * Si on veut eviter ce comportement, il faut copier le tableau au
 * debut de la fonction. *)
let fusion_listes liste_array =
  let fp = creer_fp liste_array in
  let rec aux () =
    if fp.n = 0 then [] else begin
        let i,_ = extrait_max fp in
        match liste_array.(i) with
        | [] -> failwith "Ne devrait pas arriver"
        | e::[] -> e::aux()
        | e1::e2::q -> ajout fp i (-e2);
                       liste_array.(i) <- e2::q;
                       e1 :: aux()
      end
  in
  aux();;
```

Exercice 4. Tas-max d-aires dynamiques

Dans cette solution on utilise des fonctions récursives, mais on aurait pu utiliser des fonctions itératives comme dans l'exercice 1. On écrit les mêmes fonctions que dans l'exercice 1, mais pour les tas d-aires.

Définition du type :

```
type 'a tas_max_dAire = {
    d: int;
    mutable n: int;
    mutable elem: 'a array;
  };;
```

Création d'un tas vide :

```
(* On fait en sorte que la taille du tableau elem soit non nulle *)
let create_dAire n e d =
  let n = max n 1 in
  {d = d; n = 0; elem = Array.make n e};;
```

Ajout d'un élément :

```
let echange_dAire (tas: 'a tas_max_dAire) i j =
  let tmp = tas.elem.(i) in
  tas.elem.(i) <- tas.elem.(j);
  tas.elem.(j) <- tmp;;
```

```
let rec perco_haut_dAire (tas: 'a tas_max_dAire) i =
  let i_pere = (i-1)/tas.d in
  if i > 0 && tas.elem.(i) > tas.elem.(i_pere) then begin
      echange_dAire tas i i_pere;
      perco_haut_dAire tas i_pere
    end;;
```

```
let doubler_taille tas =
  let old_size = Array.length tas.elem in
  let new_elem = Array.init
                   (2*old_size)
                   (fun i -> if i < old_size then
                               tas.elem.(i)
                             else
                               tas.elem.(0))
  in
  tas.elem <- new_elem;;
```

```
(* On suppose que la taille du tableau elem est non nulle *)
let rec ajout_dAire (tas: 'a tas_max_dAire) e =
  if tas.n = Array.length tas.elem then doubler_taille tas;
  tas.elem.(tas.n) <- e;
  tas.n <- tas.n + 1;
  perco_haut_dAire tas (tas.n-1);;
```

Extraction du maximum :

```
let rec perco_bas_dAire (tas: 'a tas_max_dAire) i =
  let indice_max = ref i in
  let i_fils = ref (tas.d*i + 1) in
  while !i_fils < tas.n && !i_fils <= tas.d * (i+1) do
    if tas.elem.(!i_fils) > tas.elem.(!indice_max) then indice_max := !i_fils;
    incr i_fils;
  done;
  if !indice_max <> i then begin
      echange_dAire tas i !indice_max;
      perco_bas_dAire tas !indice_max;
    end;;
```

```
let extrait_max_dAire (tas: 'a tas_max_dAire) =
  if tas.n = 0 then failwith "Tas vide";
  let res = tas.elem.(0) in
  tas.n <- tas.n - 1;
  tas.elem.(0) <- tas.elem.(tas.n);
  perco_bas_dAire tas 0;
  res;;
```