TD 2

File de priorité et tas

Correction

Une file de priorité est un type abstrait permettant de gérer des taches de priorités différentes.

Les différentes opérations sur les files sont :

- Création de la file vide.
- Tester si la file de priorité est vide.
- Ajouter une nouvelle tache.
- Extraire de la file la tache de poids maximum.

La dernière tache étant la plus important, sa complexité doit être minimum.

1 Implémentation naïve à l'aide d'une liste triée

On considère différentes taches dont les priorités seront représentées par des entiers.
On utilisera le type suivant :

```
type f_p = ( int * string ) list ref ;;
```

La file devra donc toujours être triée dans l'ordre décroissante des priorités.
Exemple :

```
let (file_1 : f_p ) = ref [ ( 15 , "tache_a" ) ; ( 7 , "tache_b" ) ; ( 3 , "
tache_c" ) ];;
```

Nous supposerons que les poids sont distincts.
Pour les fonctions suivantes, nous donnerons une estimation de la complexité.

1. Écrire la fonction `cree_file` qui revoie une file vide.

   ```
   cree_file : unit -> f_p
   ```

   Correction

Listing 1 – La fonction `cree_file`

```
let cree_file () = (ref [] : f_p );;
```

2. Écrire la fonction `file_vide` qui teste si la file de priorité est vide.

   ```
   file_vide : f_p -> bool
   ```

   Correction

Listing 2 – La fonction `file_vide`

```
let file_vide (f : f_p) = !f=[];;
```

3. Écrire la fonction `extraction` qui renvoie la tache de poids maximum, et retire cette tache de la file.

   `extraction : f_p -> string`

Correction

Listing 3 – La fonction `extraction`

```
let extraction f = if file_vide f then failwith "␣file␣vide␣"
          else begin
          let ( poids,tache) = List.hd (!f )in
          f := List.tl (!f) ; tache
          end;;
```

4. Écrire la fonction `ajout` qui prend en argument une file de priorité et une tache, et insère la tache dans la files.

   `ajout : f_p -> int * string -> unit`

Correction

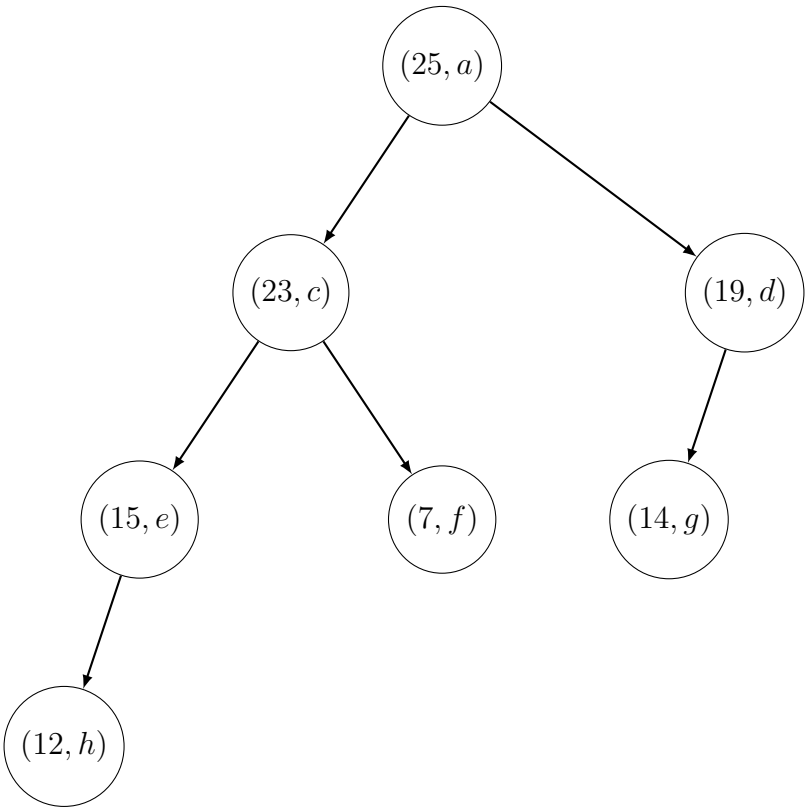
Listing 4 – La fonction `ajout`

```
let rec ajout (f: f_p) (poids,tache) = match !f with
      | [] -> f := [ (poids,tache) ]
      | (p,t)::reste when p < poids -> f := (poids,tache)::(!f)
      | (p,t)::reste  -> begin let f_2 = ref reste in
             ajout f_2 (poids,tache) ;
             f := (p,t)::(!f_2)
               end ;;
```

2 Implémentation à l'aide d'un arbre

L'idée est d'utiliser un arbre binaire dont la l'étiquette de la racine est supérieur à l'étiquette de ses fils, et il en est de même pour les fils.

Exemple :



On utilisera le type suivant :

```
type fp_arbre = V | N of fp_arbre *(int*string) * fp_arbre;;
```

Exemple :

```
let ex_1  = N(N(N(N(V,(12,"h"),V)
,(15,"e"),V),
(23,"c"), N(V,(7,"f"),V)),
(25,"a"),
N(N(V,(14,"g"),V),(19,"d"),V));;
```

1. Écrire la fonction `cree_file` qui revoie une file vide.

   `cree_file : unit -> fp_arbre`

   Correction

Listing 5 – La fonction `cree_file`

```
let cree_file () =V;;
```

2. Écrire la fonction `file_vide` qui teste si la file de priorité est vide.

   `file_vide : fp_arbre -> bool`

   Correction

Listing 6 – La fonction `file_vide`

```
let file_vide f = f = V;;
```

3. Écrire la fonction `tache_prioritaire` qui renvoie la tache de poids maximum.

   `tache_prioritaire : fp_arbre -> string`

   Correction

Listing 7 – La fonction `tache_prioritaire`

```
let tache_prioritaire f = match f with
  | V -> failwith "␣file␣vide␣"
  | N(ag,(e,t),ad) -> t;;
```

4. Écrire la fonction `fusion` qui prend deux files de priorité, et renvoie une file de priorité.

   `fusion : fp_arbre -> fp_arbre -> fp_arbre`

   Correction

Listing 8 – La fonction `fusion`

```
let rec fusion f1 f2 = match (f1,f2) with
  | ( V , _ ) -> f2
  | ( _ , V ) -> f1
  | ( N(ag1,e1,ad1),  N(ag2,e2,ad2))-> if e1 > e2 then N
  ( fusion ag1 ad1 , e1 , f2) else
           N( f1 , e2 , fusion ag2 ad2 );;
```

5. En déduire la fonction `supprime` qui supprime la racine d'une file non vide, et renvoie une file de priorité.

   `supprime : fp_arbre -> fp_arbre`

   Correction

Listing 9 – La fonction `supprime`

```
let supprime f = if file_vide f then failwith "␣erreur" else
  let N(ag,(e,t),ad) = f in fusion ag ad;;
```

6. Écrire la fonction `ajout` qui prend en argument une file de priorité et une tache, et insère la tache dans la files.

   `ajout : fp_arbre -> int * string -> fp_arbre`

   Correction

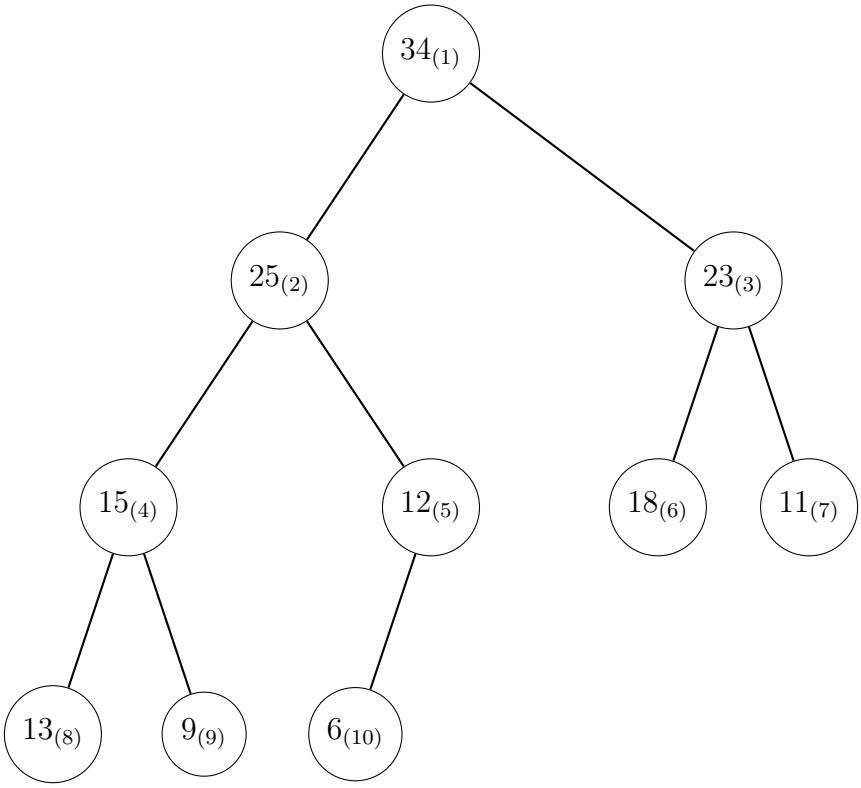
Listing 10 – La fonction `ajout`

```
let ajout f ( poids , tache ) = fusion f (N(V, ( poids , tache ), V));;
```

3 Structure de tas

L'implémentation précédente ne fait guère gagner en complexité car il n'y a pas réellement de maîtrise du squelette de l'arbre.
Un arbre binaire parfait est un arbre binaire dont tous les niveaux sont complets sauf le niveau le plus profond qui peut être incomplet auquel cas ses nœuds sont alignés à gauche de l'arbre.
Un tas est un arbre binaire parfait dont l'étiquette de la racine est supérieure aux étiquettes de ses fils, qui eux même sont des tas.

Exemple :



L'idée est de numéroter les nœuds dans l'ordre militaire, et de les implanter dans un tableau dont la première valeur est le nombre de nœuds, et les éléments d'indice i sont les nœuds en position i. Le tableau doit être d'une taille suffisante pour permettre d'ajouter des éléments dans la file.
L'exemple devient alors :

```
type f_p_tab = int array ;;
let ex_1 : f_p_tab = [| 10 ; 34; 25; 23 ; 15 ; 12 ; 18 ; 11 ;
13 ; 9 ; 6 ; 0 ;0 ; 0 ;0 ; 0 ; 0 ; 0 ; 0 ; 0 |];;
```

1. Écrire la fonction `cree_file` qui revoie une file vide. Initialement, le tableau sera de taille 20.

   ```
   cree_file : unit -> f_p_tab
   ```

   Correction

Listing 11 – La fonction `cree_file`

```
let cree_file () = (Array.make 20 0 : f_p_tab);;
```

2. Écrire la fonction `file_vide` qui teste si la file de priorité est vide.

`file_vide : f_p_tab -> bool`

Correction

Listing 12 – La fonction `file_vide`

```
let file_vide (f :  f_p_tab) = f.(0) = 0;;
```

3. On souhaite ajouter un nouvelle élément dans la file. L'idée est de le positionner à la première place vide (soit première feuille dans la structure de tas), et de le faire "remonter" jusqu'à sa place, en échangeant les étiquettes :

   (a) Écrirel la fonction `echange`, pour échanger deux valeurs d'un tableau dont on donne les indices.

   `echange : 'a array -> int -> int`

   Correction

Listing 13 – La fonction `echange`

```
let echange f i j =
  let temp = f.(i) in
  f.(i) <- f.(j) ;
  f.(j) <- temp ;;
```

   (b) Écrire la fonction `remonte`, qui prend en argument une file et un indice, et qui "remonte" la valeur à la bonne position dans la file de priorité.

   `remonte : 'a array -> int -> unit`

   Correction

Listing 14 – La fonction `remonte`

```
let rec remonte f indice =
  if indice > 1 then
    begin
      let pere = indice / 2 in
      if f.(pere) < f.(indice) then
        begin
          echange f indice pere ;
          remonte f pere
        end ;
    end;;
```

   (c) Nous supposerons ici que les tableaux sont suffisamment grands pour accueillir un nouvel élément.
   Écrire la fonction `ajout` qui ajoute dans la file un nouvel élément. Il sera initialement positionné en queue de file, et remontera à sa position.

   `ajout : f_p_tab -> int -> unit`

   Correction

Listing 15 – La fonction ajout

```
let ajout (f :f_p_tab)  valeur =
  let taille = f.(0) in
  let p = taille +1 in
  f.(p) <- valeur ;
  f.(0) <- p ;
  remonte f p ;;
```

4. Pour supprimer l'élément en tête de la file, l'idée est de l'échanger avec la dernière valeur utile du tableau, puis de "descendre" cette valeur à sa position finale. Cette opération se nomme percolation.

   (a) Écrire la fonction `percole` qui descend l'élément de tête à sa bonne place pour retrouver une structure de file de priorité.

   ```
   percole : f_p_tab -> int -> unit
   ```

   Correction

Listing 16 – La fonction percole

```
let rec percole (f :f_p_tab)  indice =
  let taille = f.(0) in
  if (indice*2 +1 <= taille) then (* deux fils *)
  begin
    if f.(indice) < f.(indice*2) || f.(indice) < f.(indice*2+1) then
    (* il doit donc y avoir echange *)
    if ( f.(indice*2) < f.(indice*2 +1) ) then (* on compare les fils
 *)
          begin
            echange f indice (indice*2 +1) ;
            percole f (indice*2 +1) ;
          end
      else
          begin
            echange f indice (indice*2) ;
            percole f (indice*2) ;
          end
  end ;
  if (indice*2 = taille) then (* un fils *)
  begin
    if f.(indice) < f.(indice*2)  then
    (* il doit donc y avoir echange *)
          begin
            echange f indice (indice*2) ;
            percole f (indice*2) ;
          end
  end ;;
```

   (b) En déduire la fonction `supprime` qui retire la tête de la file en conservant une structure de file de priorité.

`supprime : f_p_tab -> unit`

Correction

Listing 17 – La fonction `supprime`

```
let supprime (f :f_p_tab) =
  let taille = f.(0) in
  echange f 1 (taille) ;
  f.(0) <- taille -1 ;
  percole f 1;;
```

4 Tri par tas

Le but de cette partie est d'utiliser les fonctions précédentes pour trier un tableau. Les tableaux auront en premier élément leur taille.

1. Écrire la `fil_of_tab` qui prend en argument un tableau, et renvoie une file de priorité. L'idée est de remonter dans lecture du tableau, et d'utiliser la fonction `percole` pour placer chaque élément dans la file. La remontée se fera à partir du centre du tableau, le reste représentant les feuilles.

   `fil_of_tab : f_p_tab -> f_p_tab`

   Correction

Listing 18 – La fonction `fil_of_tab`

```
let fil_of_tab tab =
  (* l'objectif est de renvoyer une file a partir d'un tableau  *)
   let n = tab.(0) in
   for i = (n+1)/2 downto 1 do
    percole tab i
   done; tab;;
```

2. En déduire la fonction `tri_tas` qui prend un tableau et utilise la fonction `supprime` pour trier le tableau.

   `tri_tas : f_p_tab -> f_p_tab`

   Correction

Listing 19 – La fonction `tri_tas`

```
let tri_tas tab =
  let fil = fil_of_tab tab in
  while fil.(0)>0 do
    supprime fil
  done;
  fil;;
```

3. Estimer la complexité de la fonction `tri_tas` .

   Correction

   La complexité de `tri_tas` est en $O(n \log n)$:

- La fonction `percole` a une complexité en $O(\log n)$, du fait de la structure d'arbre.
- la fonction `fil_of_tab` prend donc un temps $O(n\log(n))$.
- La fonction *supprime* a aussi une complexité en $O(\log n)$, du fait de la fonction `percole` . Celle-ci est appelée n fois.