TP 5 : Implémentation par tableau d'un tas

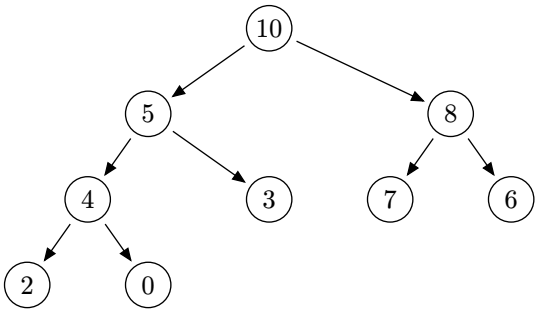
Dans ce TP, nous allons étudier une représentation efficace d'un tas, reposant sur un tableau. Nous l'utiliserons ensuite pour implémenter l'algorithme de Dijkstra puis un algorithme de tri très efficace : le tri par tas.

I. Représentation d'un tas

On représente un tas avec le type suivant :

```
type 'a tas = {
  donnees : 'a array;
  mutable taille : int
}
```

Ce type permet de représenter un tas dont les éléménts sont de type `'a`. Ces éléments sont stockés dans le tableau **`donnees`** dont les **`taille`** premiers éléments forment un arbre binaire presque complet. Par exemple, le tas représenté ci-contre sous forme d'arbre est représenté par le tableau :



10	5	8	4	3	7	6	2	0

On note par ailleurs que le champ **`taille`** est mutable.

Si on considère un élément du tas stocké à l'indice i, alors :

- son fils gauche est stocké à l'indice $2i+1$ (sous réserve qu'il existe, c'est-à-dire que $2i+1 <$ **`taille`**) ;
- son fils droit est stocké à l'indice $2i+2$ (de même, sous réserve que $2i+2 <$ **`taille`**) ;
- son parent est stocké à l'indice $\left\lfloor \frac{i-1}{2} \right\rfloor$ (sous réserve que i ne soit pas la racine, c'est-à-dire $i \neq 0$).

> **Question 1**. Écrire trois fonctions **`gauche`**, **`droite`** et **`parent`** de signature `int -> int` qui permettent respectivement de trouver l'indice du fils gauche, du fils droit et du parent d'un élément d'indice i donné.

> **Question 2**. Écrire une fonction **`tas_creer : int -> 'a -> 'a tas`** telle que **`tas_creer capacite elem`** créé un tas de capacité maximale donnée, mais initialement de taille 1, dont l'unique élément est **`elem`**. En pratique, on remplira le tableau **`donnees`** avec cet élément **`elem`**.

II. Percolations

Pour implémenter les opérations d'insertion et d'extraction dans un tas, on doit d'abord implémenter les opérations de percolation permettant de maintenir la structure de tas. On fournit la fonction suivante, permettant d'échanger deux éléments d'indice i et j dans le tas.

```
(* Échange les éléments d'indice i et j dans le tas *)
let tas_echanger (tas : 'a tas) (i : int) (j : int) =
  assert (0 <= i && i < tas.taille);
  assert (0 <= j && j < tas.taille);
  let tmp = tas.donnees.(i) in
  tas.donnees.(i) <- tas.donnees.(j);
  tas.donnees.(j) <- tmp
```

> **Question 3**. Écrire une fonction **`tas_percole_haut : 'a tas -> int -> unit`** telle que **`percole_haut tas i`** effectue une percolation vers le haut de l'élément d'indice **`i`** dans le **`tas`**. On notera que la fonction renvoie **`unit`**, le tas est donc modifié en place par la fonction.

> **Question 4**. De même, écrire une fonction **`tas_percole_bas : 'a tas -> int -> unit`** telle que **`percole_haut tas i`** effectue une percolation vers le bas de l'élément d'indice **`i`** dans le **`tas`**.

III. Insertion et extraction

> **Question 5**. Écrire une fonction **`tas_inserer : 'a tas -> 'a -> unit`** qui implémente l'insertion d'un élément dans un tas. La fonction vérifie d'abord qu'on a la place pour accueillir cet élément dans le tas ; si ce n'est pas le cas, on provoque une erreur avec **`failwith`**.

Question 6. Écrire une fonction `tas_extraire : 'a tas → 'a` qui implémente l'extraction de l'élément minimal du tas. La fonction vérifie d'abord que le tas n'est pas vide ; si ce n'est pas le cas, on provoque une erreur avec `failwith`.

IV. Algorithme de Dijkstra

Question 7. Compléter le code ci-dessous en utilisant la structure de tas min définie précédemment comme file de priorité pour implémenter l'algorithme de Dijkstra. On représente les graphes par une matrice d'adjacence pondérée, de type `float array array`. Si `graphe` est une telle matrice, `graphe.(u).(v)` représente le poids de l'arc $u \to v$ si ce nombre est positif ou nul ; si ce nombre est négatif, l'arc n'existe pas. De plus, `s : int` est l'indice du sommet dont on veut partir et `t : int` est le numéro du sommet auquel on veut arriver. On rappelle que l'algorithme calcule la longueur d'un plus court chemin entre `s` et `t`.

```
let dijkstra (graphe : float array array) (s : int) (t : int) =
  (* Initialisation des structures dont on a besoin *)
  let n = Array.length graphe in
  let dist = ... in (* Indice : le mot clé infinity existe en OCaml et est plus grand que tout float *)
  let vu = Array.make n false in
  let a_voir = tas_creer ... in (* a_voir est la file de priorite, initialement elle contient (0, s) *)
  dist.(s) <- ... ;

  (* Boucle principale *)
  while ... do
    let (dv, v) = ... in (* Extraire le prochain sommet courant *)
      if not vu.(v) then begin
        vu.(v) <- ... ;
        for w = 0 to n-1 do
          if v <> w && graphe.(v).(w) >= 0. then (* si l'arc v → w existe *)
            ...
        done
      end
  done;

  (* Conclusion *)
  if ... then failwith "On ne peut pas atteindre la cible"
  else ...
```

Pour tester votre algorithme, on fournit les graphes suivants, en précisant les valeurs attendues.

```
let graphe1 = [|
  [|-1.;85.;217.;-1.;173.;-1.;-1.;-1.;-1.;-1.|]; [|85.;-1.;-1.;-1.;-1.;80.;-1.;-1.;-1.;-1.|];
  [|217.;-1.;-1.;-1.;-1.;-1.;186.;103.;-1.;-1.|]; [|-1.;-1.;-1.;-1.;-1.;-1.;-1.;183.;-1.;-1.|];
  [|173.;-1.;-1.;-1.;-1.;-1.;-1.;-1.;-1.;502.|]; [|-1.;80.;-1.;-1.;-1.;-1.;-1.;-1.;250.;-1.|];
  [|-1.;-1.;186.;-1.;-1.;-1.;-1.;-1.;-1.;-1.|]; [|-1.;-1.;103.;183.;-1.;-1.;-1.;-1.;-1.;167.|];
  [|-1.;-1.;-1.;-1.;-1.;250.;-1.;-1.;-1.;84.|]; [|-1.;-1.;-1.;-1.;502.;-1.;-1.;167.;84.;-1.|]
|]
let d1 = dijkstra graphe1 0 9 (* doit renvoyer 487. *)

let graphe2 = [| [|-1.;1.;1.;-1.;-1.;-1.|]; [|1.;-1.;3.;-1.;-1.;10.|]; [|1.;3.;-1.;2.;4.;-1.|];
                 [|-1.;-1.;2.;-1.;-1.;4.|]; [|-1.;-1.;4.;-1.;-1.;3.|]; [|-1.;10.;-1.;4.;3.;-1.|] |]
let d2 = dijkstra graphe2 0 5 (* doit renvoyer 7. *)

let graphe3 = [| [|-1.;1.;3.;5.;7.|]; [|1.;-1.;1.;3.;5.|]; [|3.;1.;-1.;1.;3.|];
                 [|5.;3.;1.;-1.;1.|]; [|7.;5.;3.;1.;-1.|] |]
let d3 = dijkstra graphe3 0 4 (* doit renvoyer 4. *)
```

V. Tri par tas

Nous allons maintenant implémenter l'algorithme du tri par tas.

Question 8. Écrire une fonction `tas_construire : 'a array → 'a tas` qui construit un tas à partir des éléments du tableau passé en paramètre. Pour cela on crée un tas dont le champ `donnees` est initialement le tableau passé en paramètre. Ensuite, la structure de tas est établie en percolant vers le bas chaque élément du tableau, en partant de la fin.

Question 9. Écrire une fonction `tri_par_tas : 'a array → unit`. Cette fonction doit modifier le tableau passé en paramètre.