TP 7 : Implémentation des tas en OCaml

Notre première application de la programmation impérative en OCaml, faisant intervenir notamment des tableaux et des boucles, sera l'implémentation des tas. Les tas sont des structures de données très utilisées en informatique, notamment pour implémenter des files de priorité. On va donc commencer par les définir et les implémenter en OCaml.

Exercice 1 – Implémentation simple des tas-min

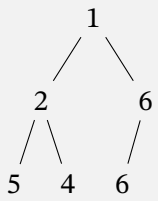
On va implémenter directement les tas-min. On va utiliser un tableau pour stocker les éléments du tas, et on va les organiser pour que le minimum soit en première position. Dans cette implémentation, on utilise le type suivant :

```OCaml
type 'a tas_simple =
  { mutable taille : int;
    tab : 'a array }
```

Dans ce type, `taille` est la taille du tas, et `tab` est le tableau contenant les éléments du tas. Attention, `tab` est un tableau de taille fixe N : on utilise les `taille` premières cases de `tab` pour stocker les éléments du tas[a], avec donc $0 \leq$ `taille` $\leq N$.

Exemple 1

Représentons le tas-min suivant à 6 éléments dans un tas de capacité 9 :




```OCaml
{ taille = 6;
  tab =
    [| 1;2;5;4;6;6;8;4;9 |] }
```

On remarque que les valeurs de `tab` à partir de l'indice `taille` sont quelconques.

▶ **Question 1** Écrire une fonction `creer : int -> 'a -> 'a tas_simple` qui crée un tas-min vide de taille maximale donnée. *Le second argument donne une valeur permettant d'initialiser les cases du tableau.*

▶ **Question 2** Écrire une fonction `est_vide : 'a tas_simple -> bool` qui teste si un tas est vide.

On a besoin de plusieurs fonctions techniques pour manipuler les tas simplement.

▶ **Question 3** Écrire une fonction `echange : 'a array -> int -> int -> unit` qui échange les éléments d'indices donnés dans un tableau.

▶ **Question 4** Écrire une fonction `pere : int -> int` qui renvoie l'indice du père d'un élément donné. De même, écrire les fonctions `fils_gauche : int -> int` et `fils_droit : int -> int` qui renvoient les indices des fils gauche et droit d'un élément donné.

On peut maintenant implémenter l'opération la plus simple : l'ajout d'un élément dans un tas.

▶ **Question 5** Écrire une fonction `ajouter : 'a tas_simple -> 'a -> unit` qui ajoute un élément dans un tas. On utilisera `failwith` pour signaler une erreur si le tas est plein. *On pourra implémenter une fonction* `percoler_haut : 'a tas_simple -> int -> unit` *qui percole un élément vers le haut du tas : elle pourra être implémentée récursivement ou à l'aide d'une boucle* `while`*. On n'oubliera pas les deux cas d'arrêt des appels récursifs : soit l'élément est à la racine, soit il est plus grand que son père.*

La percolation vers le bas est un peu plus délicate car il faut choisir le plus petit des deux fils pour l'échanger avec le père. Il y a donc trois cas à traiter :

— Si l'élément d'indice `i` n'a pas de fils (`fils_gauche i >= tas.taille`);

— si l'élément d'indice `i` a un seul fils (qui est donc un fils gauche, et `fils_droit i = tas.taille`);
— si l'élément d'indice `i` a deux fils.

▶ **Question 6** Écrire une fonction `argmin3 : 'a tas_simple -> int -> int` tel que l'appel `argmin3 tas i` renvoie :
— `i` si c'est une feuille du tas ;
— l'indice du plus petit élément entre `i` et son fils gauche si `i` a un seul fils (`i` en cas d'égalité) ;
— l'indice du plus petit élément entre `i`, son fils gauche et son fils droit si `i` a deux fils (`i` en cas d'égalité).

Cette fonction permet de déterminer si l'élément d'indice `i` doit être échangé avec son fils gauche, son fils droit ou aucun des deux. Notez qu'elle se charge du cas de base où `i` est une feuille.

▶ **Question 7** En déduire une fonction `retirer_min : 'a tas_simple -> 'a` qui retire et renvoie le minimum d'un tas. *On pourra implémenter une fonction* `percoler_bas : 'a tas_simple -> int -> unit` *qui percole un élément vers le bas du tas : elle pourra être implémentée récursivement ou à l'aide d'une boucle* `while`.

a. Les autres contiennent des valeurs quelconques : par exemple, elles pourraient être la trace d'anciennes valeurs du tas, ou alors des valeurs qui n'ont jamais été modifiées depuis la création du tas.

Exercice 2 – Application à l'algorithme de Dijkstra

L'algorithme de Dijkstra permet de trouver les plus courts chemins dans un graphe pondéré. On représentera un graphe pondéré par une liste d'adjacence, où chaque sommet est associé à une liste de paires (sommet, poids) représentant les arêtes sortantes de ce sommet.

OCaml
```
type graphe = (int * int) list array

let exemple_tp15_itc =
  [| [(1, 10); (3, 5)];
    [(2, 1); (3, 2)];
    [(4, 4)];
    [(1, 3); (2, 9); (4, 2)];
    [(0, 7); (2, 6)] |]
```

Dans un graphe `g : graphe`, `g.(i)` est la liste des arêtes sortantes du sommet `i` (les sommets sont numérotés de 0 à $n-1$).

▶ **Question 1** Écrire une fonction `nombre_sommets : graphe -> int` qui renvoie le nombre de sommets d'un graphe.

▶ **Question 2** Écrire une fonction `nombre_aretes : graphe -> int` qui renvoie le nombre d'arêtes d'un graphe.

Pour implémenter une file de priorité contenant des valeurs `'a` avec des priorités `'p`, on va utiliser un tas-min contenant des paires `('p, 'a)`.

▶ **Question 3** Écrire une fonction `initialiser_file : graphe -> int -> (int * int) tas_simple` tel que `initialiser_file g s` renvoie une file de priorité de capacité suffisante contenant uniquement le sommet `s` avec une priorité nulle.

▶ **Question 4** Écrire une fonction `dijkstra : graphe -> int -> int array` qui renvoie un tableau des distances minimales depuis un sommet donné. *On pourra utiliser la valeur* `max_int : int` *pour représenter* $+\infty$.

▶ **Question 5** Tester votre fonction `dijkstra` sur le graphe `exemple_tp15_itc` en partant du sommet 0.

Je vous fournis dans un fichier annexe les fonctions nécessaires à la lecture d'un graphe depuis un fichier texte au format DIMACS simplifié, vous pouvez réutiliser les fichiers des TP 14/15 d'ITC. Vous pouvez également adapter ces fonctions pour lire des fichiers plus conséquents et tester vos algorithmes sur des graphes plus grands en visitant le site `http://www.diag.uniroma1.it/challenge9/download.shtml`.

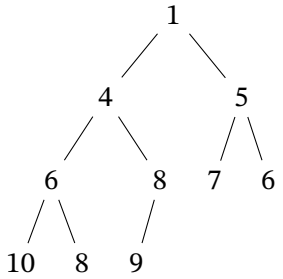
▶ **Question 6** Vérifier que vos algorithmes fonctionnent en temps raisonnable sur des graphes de taille plus importante. Comparer la vitesse d'exécution de ces fonctions avec celles en Python. *On fournit une fonction* `time : ('a -> 'b) -> 'a -> 'b` *tel que* `time f x` *renvoie le résultat de* `f x` *après avoir affiché son temps d'exécution.*

★ **Question 7** Adapter les fonctions de lecture de graphe pour qu'elles lisent les fichiers DIMACS de position des sommets. Implémenter l'algorithme A* pour trouver le plus court chemin entre deux sommets d'un graphe pondéré. Comparer le nombre de sommets visités par A* avec celui de Dijkstra sur des graphes de taille importante.

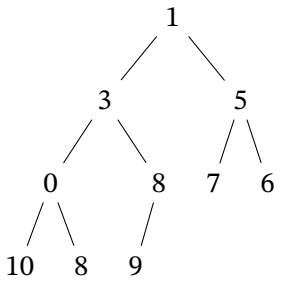
Exercice 3 – Réduction de priorité dans un tas-min

Dans la prochaine partie, on va essayer d'adapter la structure de file de priorité afin de permettre de modifier les priorités des éléments déjà présents dans la file. Ajouter cette opération permet de garantir que dans la file de priorité de Dijkstra ou A*, chaque sommet n'apparaît au plus qu'une fois dans la file de priorité.
Tentons d'abord de réduire la priorité d'un élément déjà présent dans la file de priorité, connaissant sa position initiale :

Voici le tas-min initial :



Modifions la priorité de l'élément d'indice 3 (priorité 6 à gauche) pour la réduire à 0 :



On peut alors percoler cet élément vers le haut pour rétablir la propriété de tas-min : comme pour une insertion classique, on compare l'élément avec son père et on l'échange si nécessaire, jusqu'à ce que l'élément soit supérieur ou égal à son père ou qu'il soit à la racine.

▶ **Question 1** Implémenter cette fonction `diminuer_priorite : 'a tas_simple -> int -> 'a -> unit` qui réduit la priorité d'un élément d'indice donné dans un tas-min. *On pourra réutiliser la fonction* `percoler_haut` *pour percoler l'élément vers le haut.*

On pourrait également vouloir augmenter la priorité d'un élément déjà présent dans la file de priorité, ce qui impliquerait une percolation vers le bas. On peut l'implémenter, mais ce n'est pas utile pour l'algorithme de Dijkstra.
L'algorithme de Dijkstra s'adapte à cette nouvelle opération sur les files de priorité s'exprime comme suis :

Pseudo-code

```
1 : Fonction DIJKSTRA(𝒢, s)
2 :     dist ← [∞; ∞; ... ; ∞]                ▷ tableau de taille n = |S|
3 :     dist[s] ← 0
4 :     file = FILEPRIOVIDE()
5 :     INSÉRER(file, s, 0)
6 :     Tant que ¬ESTVIDE(file) , faire
7 :         x ← RETIREMIN(file)
8 :         Pour y successeur de x , faire
9 :             d ← dist[x] + p(x, y)
10 :            Si d < dist[y] alors
11 :                Si dist[y] < ∞ alors
12 :                    DIMINUERPRIORITÉ(file, y, d)
13 :                Sinon
14 :                    INSÉRER(file, y, d)
15 :                dist[y] ← d
16 :    retourne dist
```

On remarque que notre implémentation de l'opération de réduction de priorité est nécessaire pour implémenter cet algorithme, mais notre façon de la coder pour l'instant est insuffisante : en effet, lors de l'appel DIMINUERPRIORITÉ(file, y, d), on ne connaît pas l'indice de y dans la file de priorité. Il nous faut donc modifier la structure de file de priorité pour nous permettre d'accéder et de maintenir à jour cette information.

OCaml

```
type tas_avec_valeurs = {
  mutable taille : int;
  priorites : 'a array;
  position : int array;
  cle : int array;
}
```

Dans ce type :

- `taille` désigne la taille actuelle du tas,
- La capacité de la file est fixée à l'initialisation et est la taille des tableaux `priorites`, `position`, `cle`. Selon le même principe que pour les piles implémentées par des tableaux, une partie de ces tableaux contiendra des valeurs quelconques.
- `priorites` correspond à la notion de tas vue en cours : la tranche `priorites[0:taille]` représente un arbre binaire complet à `taille` nœuds respectant la propriété de tas min.
- Pour le tableau `position`, on a deux cas :
  - si `x` est dans la file de priorité, `position.(x)` est la position de la priorité de `x` dans le tableau `priorite`,
  - sinon, `position.(x) = -1`
- pour `0 <= i < taille`, `cle.(i)` est le sommet dont la priorité est stockée dans `position.(i)`.

Pour un sommet `x`, sa priorité est donc stockée en `priorite.(position.(x))` et on a `cle.(position.(x)) = x`.

▶ **Question 2** Implémenter la fonction `create : int -> tas_complet` tel que `create n` crée une file de priorité vide de capacité maximale `n`. Implémenter également `is_empty : tas_complet -> bool`.

Commençons par le commencement : si une simple fonction `echange` habituelle permettait de permuter deux éléments dans le tableau représentant nos tas dans le cours, ici il faut faire attention à garder à jour les trois tableaux en même temps.

▶ **Question 3** Écrire la fonction `echange_complet : tas_complet -> int -> int -> unit` tel que `echange_complet f x y` échange les positions dans le tas des sommets `x` et `y`. On maintiendra tous les

invariants cités précédemment, sauf la propriété de tas min.

▶ **Question 4** Adapter les fonctions `percoler_haut` et `percoler_bas` pour qu'elles maintiennent la propriété de tas min dans notre nouvelle structure de tas.

▶ **Question 5** En déduire une implémentation des fonctions suivantes :
- `ajouter : 'a tas_complet -> int -> 'a -> unit` qui ajoute un élément dans la file de priorité ;
- `retirer_min : 'a tas_complet -> int` qui retire et renvoie le sommet de plus petite priorité ;
- `diminuer_priorite : 'a tas_complet -> int -> 'a -> unit` qui diminue la priorité d'un élément déjà présent dans la file de priorité.

▶ **Question 6** En déduire une implémentation de l'algorithme de Dijkstra utilisant cette nouvelle structure de file de priorité.

▶ **Question 7** Comparer la taille maximale des tas et le temps d'exécution de l'algorithme de Dijkstra avec les deux implémentations de file de priorité.

▶ **Question 8** Comparer leurs temps d'exécution.

▶ **Question 9** Faire de même pour l'algorithme A*.