TP 14 : Structure de tas

On a vu en cours la structure de tas-min. On va maintenant implémenter la structure très similaire de tas-max en C. On commence par implémenter la structure d'ensemble ordonné par tas (Exercice 1) puis on utilisera cette structure pour implémenter l'algorithme de tri par tas (Exercice 2). On étendra la structure de tas pour permettre de représenter une file de priorité à valeur entière (Exercice 3). **J'ai ajouté une section sur l'implémentation de tas fonctionnels par les arbres de Braun, en OCaml, en dernière partie.**

I Structure de tas et de file de priorité en C

Exercice 1 – Définition du type et implémentation de base

On utilisera le type suivant :

```c
typedef struct {
  int taille; // nombre d'éléments dans le tas
  int capacite; // taille du tableau tab
  int *tab;
} tas;
```

▶ **Question 1** Écrire une fonction `tas* creer_tas(int capacite)` qui crée un tas vide de capacité `capacite`.

▶ **Question 2** Écrire une fonction `void free_tas(tas* t)` qui libère la mémoire allouée pour le tas `t`.

▶ **Question 3** Implémenter les opérations de base sur les tas-max :
- `bool est_vide(tas* t)` : renvoie `true` si le tas est vide, `false` sinon.
- `int pere(int i)` : renvoie l'indice du père de l'élément d'indice `i`.
- `int gauche(int i)` : renvoie l'indice du fils gauche de l'élément d'indice `i`.
- `int droite(int i)` : renvoie l'indice du fils droit de l'élément d'indice `i`.
- `void echanger(tas* t, int i, int j)` : échange les éléments d'indices `i` et `j` dans le tas `t`.

▶ **Question 4** Implémenter la fonction `void percoler_bas(tas* t, int i)` qui percole l'élément d'indice `i` vers le bas pour rétablir la propriété de tas-max.

▶ **Question 5** De même, implémenter la fonction `void percoler_haut(tas* t, int i)`.

▶ **Question 6** En déduire les fonctions `void inserer(tas* t, int x)` et `int extraire_max(tas* t)`.

Si l'on connaît l'indice d'un élément dans le tas que l'on veut modifier, on peut le faire en temps logarithmique en utilisant les fonctions `percoler_bas` et `percoler_haut`.

▶ **Question 7** Écrire une fonction `void modifier(tas* t, int i, int x)` qui modifie l'élément d'indice `i` dans le tas `t` pour qu'il vaille `x`.

Exercice 2 – Tri par tas

Une utilisation immédiate des tas est le tri par tas, dont l'idée est très simple.
- On prend en entrée un tableau `t` de taille `n`.
- On crée un tas `tas` de capacité `n` et on insère successivement les éléments de `t` dans le tas.
- On extrait successivement les éléments du tas pour les remettre dans le tableau `t`.

— On obtient alors un tableau trié.

▶ **Question 1** Écrire une fonction `void tri_tas(int *t, int n)` qui trie le tableau `t` de taille `n` en utilisant un tas.

Comme la création d'un tas est en $\mathcal{O}(n)$, chaque insertion et suppression en $\mathcal{O}(\log n)$, le tri par tas est en $\mathcal{O}(n \log n)$. La complexité spatiale est cependant mauvaise, puisque la complexité spatiale est en $\mathcal{O}(n)$.
Pour contrer ce problème, on peut utiliser le tableau `t` lui-même comme le tableau d'un tas. Puisque les opérations de percolation opèrent par échange, on peut les utiliser directement sur le tableau `t`. On remarque également que lorsque l'on supprime un élément du tas, on peut le placer à l'indice `taille` du tas.

▶ **Question 2** Modifier la fonction `tri_tas` pour qu'elle trie le tableau `t` directement.

★ **Question 3** Montrer la correction de votre algorithme.

Exercice 3 – File de priorité

Dans l'Exercice 1, on a implémenté un tas-max permettant de représenter un ensemble d'élément où savoir quelle est son maximum et le retirer est une opération efficace. Pour écrire une vraie file de priorité, il faut que l'on puisse représenter des *valeurs* ayant une *priorité*, et que l'on puisse ajouter des valeurs à la file de priorité, en retirer, et modifier la priorité d'une valeur. Pour cela, on va utiliser trois tableaux. On se limitera ici à des valeurs entières comprises entre 0 et `capacite` − 1, et on supposera que les valeurs insérées dans le tas sont distinctes. Comme dans les tas implémentés précédemment, une partie de ces tableaux pourra contenir une valeur quelconque dans certaines cases.

— `priorites` : tableau de taille `capacite` tel que `priorites[i]` est la priorité de la valeur `valeurs[i]` (`priorites[i]` est quelconque pour `i >= taille`);
— `valeurs` : tableau de taille `capacite` tel que `valeurs[i]` est la valeur dont la priorité est stockée à l'indice `i` (`valeurs[i]` est quelconque pour `i >= taille`);
— `position` : tableau de taille `capacite` tel que `position[v]` est l'indice tel que `valeurs[position[v]] = v` (`position[v]` est quelconque si `v` n'est pas une valeur dans la file).

C
```c
typedef struct {
  int taille; // nombre d'éléments dans la file
  int capacite; // taille des tableaux tab, valeurs et priorites
  int *priorites;
  int *valeurs;
  int *position;
} file_priorite;
```

▶ **Question 1** Écrire une fonction `file_priorite* creer_file_priorite(int capacite)` qui crée une file de priorité vide de capacité `capacite`.

▶ **Question 2** Écrire une fonction `void free_file_priorite(file_priorite* f)` qui libère la mémoire allouée pour la file de priorité `f`.

Cette fois-ci, quand on effectue des opérations sur la file de priorité, on doit mettre à jour les trois tableaux `priorites`, `valeurs` et `position` en même temps. Pour cela, on utilisera la fonction auxiliaire suivante.

▶ **Question 3** Écrire une fonction `void echange_fp(file_priorite* f, int i, int j)` qui échange les éléments d'indices `i` et `j` (et donc les valeurs `valeurs[i]` et `valeurs[j]`) dans la file de priorité `f`.

▶ **Question 4** En déduire la fonction `void percoler_bas_fp(file_priorite* f, int i)` qui percole l'élément d'indice `i` vers le bas pour rétablir la propriété de file de priorité. De même, écrire la fonction `void percoler_haut_fp(file_priorite* f, int i)`.

▶ **Question 5** En déduire les fonctions `void inserer_fp(file_priorite* f, int v, int p)` et `int extraire_max_fp(file_priorite* f)`.

▶ **Question 6** Écrire une fonction `void modifier_priorite(file_priorite* f, int v, int p)` qui modifie la priorité de la valeur `v` dans la file de priorité `f` pour qu'elle vaille `p`.

II Définition des tas par arbre de Braun

Dans cette section, on note $|A|$ la taille de l'arbre de Braun A.

Exercice 4 – Définition et propriétés

Définition 1 – Arbre de Braun

L'ensemble des arbres de Braun est défini par induction (dans les arbres binaires) :
- l'arbre vide est un arbre de Braun ;
- pour g, d deux arbres de Braun tels que $|d| \leq |g| \leq |d| + 1$, $N(g, d)$ est un arbre de Braun.

▶ **Question 1** Montrer que pour tout $n \in \mathbb{N}$, il existe un unique arbre de Braun de taille n.

▶ **Question 2** Déterminer les arbres de Braun de taille $n \in [\![0, 6]\!]$.

▶ **Question 3** Montrer que pour A un arbre de Braun non vide, $h(A) = \lfloor \log_2 |A| \rfloor$.

On définit le type usuel suivant pour les arbres de Braun (on les étiquette par des entiers) :

OCaml
```
type braun =
  | V
  | N of int * braun * braun
```

Dans la suite, on notera toujours h la hauteur de l'arbre de Braun, et n sa taille.

▶ **Question 4** Écrire une fonction `hauteur : braun -> int` qui renvoie la hauteur d'un arbre de Braun avec une complexité en $\mathcal{O}(h)$.

Exercice 5 – Calcul de la taille d'un arbre de Braun

Grâce aux contraintes sur la taille des enfants de chaque noeud d'un arbre de Braun, on peut calculer sa taille de manière efficace.

▶ **Question 1** Soit A un arbre de Braun et $m \in \mathbb{N}$ tel que $2m \leq |A| \leq 2m + 1$. Quelles sont les tailles possibles pour les sous-arbres gauche et droit de A ?

▶ **Question 2** Même question si $2m + 1 \leq |A| \leq 2m + 2$.

▶ **Question 3** En déduire une fonction `diff : braun -> int -> int` ayant la spécification suivante :

Entrée : Un arbre de Braun A et un entier n ;
Précondition : $n \leq |A| \leq n + 1$;
Sortie : $|A| - n$ (qui sera donc dans $\{0, 1\}$).

Cette fonction aura une complexité en $\mathcal{O}(h)$.

▶ **Question 4** En déduire une fonction `taille : braun -> int` qui renvoie la taille d'un arbre de Braun.

▶ **Question 5** Déterminer sa complexité (qui n'est pas $\mathcal{O}(n)$).

Exercice 6 – Tas fonctionnel par arbre de Braun

Les arbres de Braun permettent de définir une structure de tas fonctionnel, c'est-à-dire que l'on implémente le type abstrait suivant :

Op	Spécification	Type de l'opération
C	Crée un tas vide	`'p tas`
A	Vérifie que la file en entrée est vide	`'p tas -> bool`
T	Ajoute un élément p dans le tas	`'p tas -> 'p -> 'p tas`
T	Renvoie le couple (élément min, tas sans cet élément).	`'p tas -> 'p * 'p tas`

Pour cela, on va construire des arbres de Braun respectant la propriété de tas-min (chaque nœud a une étiquette inférieure ou égale à celles de ses enfants). On les appelle *tas de Braun*.

▶ **Question 1** Implémenter les deux premières opérations.

▶ **Question 2** Implémenter la fonction `get_min : braun -> int` qui renvoie le minimum d'un arbre de Braun, `max_int` s'il est vide (ça sera utile dans la suite).

▶ **Question 3** Écrire la fonction `insert : braun -> int -> braun` qui insère un élément dans un arbre de Braun. *On s'assurera que l'arbre résultant est bien un arbre de Braun vérifiant la propriété de tas-min.*

Pour écrire la fonction `extract_min : braun -> int * braun`, implémentant la dernière opération, on va avoir besoin d'une fonction `merge : braun -> braun -> braun` dont la spécification est :

Entrée : A, B deux tas de Braun.
Précondition : $|B| \leq |A| \leq |B| + 1$;
Sortie : Renvoie un tas de Braun contenant les étiquettes de A et B.

▶ **Question 4** En supposant la fonction `merge : braun -> braun -> braun` implémentée, écrire la fonction `extract_min : braun -> int * braun` implémentant la dernière opération.

Plus qu'à écrire la fonction `merge` ! Commençons par les cas simples.

▶ **Question 5** Que doit valoir `merge g V`? Que doit valoir `merge g d` quand `get_min g <= get_min d`?

Le seul cas restant est donc celui où `get_min g > get_min d`. La racine de l'arbre que l'on veut obtenir est alors la racine de `d`. Si on essaye de le retirer de `d` pour obtenir le fils droit de notre arbre final, on peut se retrouver avec un tas-min ne vérifiant pas la propriété d'arbre de Braun.

▶ **Question 6** Écrire une fonction `extract_element : braun -> int * braun` tel que `extract_element a` renvoie le couple `(x, b)` où `x` est une étiquette quelconque de `a` et `b` est un arbre de Braun contenant les autres étiquettes de `a`.

▶ **Question 7** Écrire une fonction `replace_min : braun -> int -> braun` tel que `replace_min a x` renvoie un tas de Braun contenant les étiquettes de `a` sauf sa racine et contenant `x` en plus.

▶ **Question 8** En déduire la fonction `merge`.

▶ **Question 9** Déterminer la complexité de `extract_min`.