Tas et files de priorité (modifiables)

Résumé des aventures précédentes : la structure de données *file de priorité* a pour opérations structurelles

1. `creer_fp_vide` : créer une file de priorité vide ;
2. `est_fp_vide` : teste si une file de priorité est vide ;
3. `inserer` : ajoute un élément (avec sa priorité) à la file de priorité ;
4. `retire_prioritaire` : retire un élément avec la plus grande priorité et le retourne.

On souhaite implémenter cette structure de données. On va voir dans ce chapitre qu'on peut utiliser des tas, mais :

- les tas ont un intêrêt en soi, indépendamment de l'implémentation des files de priorité ;
- les files de priorité peuvent être implémentés autrement.

I Tas max

I.1 Définition

On suppose avoir fixé un ensemble d'étiquettes $\mathcal{E}$ et une application cle $: \mathcal{E} \to \mathcal{K}$ où $\mathcal{K}$ est un ensemble totalement ordonné. Autrement dit, à toute étiquette on peut associer sa **clé** et l'ensemble des clés est totalement ordonné.

Exemples 1 :

1. Si le type des étiquettes est totalement ordonné, on peut simplement prendre pour clés les étiquettes elles-mêmes.
2. Le type des étiquettes peut être de la forme `'a*'b` et les clés les éléments de type `'b` (les élément de type `'a` étant les *valeurs* des étiquettes).
3. Le type des étiquettes peut-être un type enregistrement dont l'un des champs correspond aux clés.

Oui, jusqu'ici j'ai copié-collé le cours sur les ABR. Attention, maintenant, ça change.

Définition 1 : Arbre quasi-complet

Un arbre de hauteur h est **quasi-complet** si pour $0 \leqslant k < h$ il y a exactement 2^k nœuds de profondeur k, et pour $k = h$ tout les nœuds de profondeur h sont "entassés à gauche".

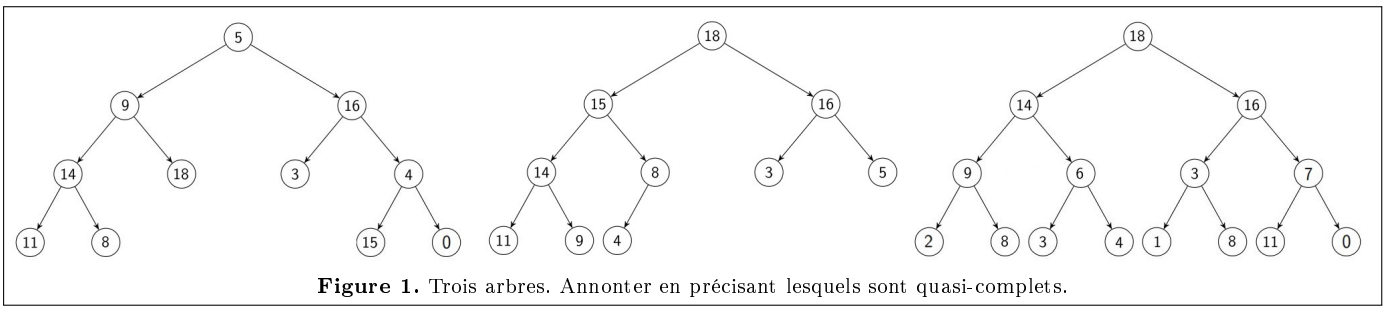

Figure 1. Trois arbres. Annonter en précisant lesquels sont quasi-complets.

Remarque 1 : L'intérêt des arbres quasi-complets est que, pour une taille d'arbre donnée, ce sont des arbres de hauteur minimale ayant cette taille. En particulier la hauteur d'un arbre quasi-complet à n nœuds est .....

Remarque 2 : Pour tester si un arbre est quasi-complet, il suffit de :

Définition 2 : Tas max

Étant donné un ensemble d'étiquettes $\mathcal{E}$ muni de clés, on appelle tas max (dans la suite et pour faire simple : tas) un arbre a étiquetté par $\mathcal{E}$ qui vérifie les deux propriétés suivantes :

- a est quasi-complet ;
- chaque nœud de a a une étiquette supérieure aux étiquettes de ses fils.

Préciser sur la *figure 1* quels arbres sont des tas, s'il en existe.

Exercice bonus : écrire une fonction qui détermine si un arbre est un tas.

Remarque 3 : Un *tas min* est un tax max pour l'ordre dual.
Donc c'est pareil et on va dire que si on a compris les tas max on a compris les tas min.

Les tas ont deux intérêts : dans un tas de taille `t`, on peut rajouter un élément en préservant la structure de tas, ou retirer la racine du tas en préservant la structure de tas, tout cela en temps $O\big(\lg(t)\big)$, comme on va le voir ci-dessous.

I.2 Implémentation modifiable

Rappelons ici l'idée vue dans le cours sur les arbres. On peut implémenter un élément de type `'a arbre` par un tableau `t` d'éléments de type `'a option`, de longueur suffisamment grande (au moins n avec n la taille de l'arbre) : l'étiquette de la racine (si elle n'est pas vide) est stockée en `t.(0)` ; et, pour chaque nœud dont l'étiquette est stockée en `t.(i)`, l'étiquette de son fils gauche (s'il est non vide) sera stocké en `t.(2i+1)` et celle de son fils droit (s'il est non vide) en `t.(2i+2)` ; les sous-arbres `Vide` (ou plus généralement les cases inutiles du tableau) étant représentés par `None`.

Exemple 2 : Prenons l'exemple des trois arbres de la *figure* 1, on obtient :

Remarque 4 :
Cela correspond à écrire dans le tableau les étiquettes des nœuds de l'arbre dans l'ordre du .....

Implémentation modifiable des tas : on implémente les tas par le type suivant :

```
type 'e tas = { mutable taille : int ; contenu : 'e option array; };;
```

Où `'e` est le type des étiquettes (voir l'introduction).

Remarques 5 :

1. Bien sûr, un terme de ce type ne représente effectivement un tas que si la taille est inférieure à la longueur du tableau (et strictement inférieure à l'indice du premier `None` rencontré dans le tableau).
2. L'intérêt à avoir une longueur strictement supérieure à la taille est de pouvoir rajouter des nœuds à notre tas. Mais la longueur du tableau reste un majorant du nombre de nœuds maximal de notre tas et tenter d'en rajouter davantage provoquera une erreur. Une solution serait, comme pour les tableaux redimensionnables, de recopier le contenu du tas dans un tableau deux fois plus grand lorsqu'une telle erreur se produit ; on se contentera ici (comme on l'avait fait avec l'implémentation modifiable des files par un tableau circulaire) de prendre une longueur de tableau suffisamment grande pour rendre très improbable une telle erreur.
3. Le type `'e option` ne s'impose pas puisqu'on sait précisément quels sont les cases du tableau qui contiennent une information pertinente (les cases 0 à `t`-1 où `t` est la taille), on pourrait mettre n'importe quoi dans les autres (sauf que justement si `'e` est quelconque on ne connaît pas d'élément remarquable de `'e`...).
4. Même pour une implémentation immuable, il est préférable de définir les tas comme un couple (taille, arbre) car la taille permet de déterminer la position du dernier nœud non vide dans l'arbre, c'est-à-dire le dernier élément du tas. On va voir dans la suite qu'il est important de connaître sa localisation (mais son calcul est linéaire dans le cas des arbres immuables).

Avec cette implémentation modifiable, il est très facile de connaître les indices du fils gauche, du fils droit et du père d'un nœud dont on connaît l'indice !

```
let fg i =          ;;

let fd i =          ;;

let pere i =        ;;
```

Dans la suite, on aura également besoin d'une fonction `permuter : int → int → 'a array → unit` qui prend comme arguments deux entiers `i` et `j` et un tableau `c`, ne retourne rien mais a pour effet de bord de permuter les éléments `c.(i)` et `c.(j)` de `c`.

I.3 Ajout d'un élément dans un tas modifiable

On travaille volontairement sur des `'a array` car c'est plus général (pour obtenir un `'e option array` il suffit de se placer dans le cas `'a = 'e option`). Notons le `c` (pour *contenu*) dans la suite.

Pour insérer un nouveau nœud dans un tableau `c` codant un tas de taille `t`, on commence par l'insérer au premier emplacement disponible, c'est-à-dire en position `c.(t)`, puis on le permute avec son père tant qu'il est plus grand que ce dernier.

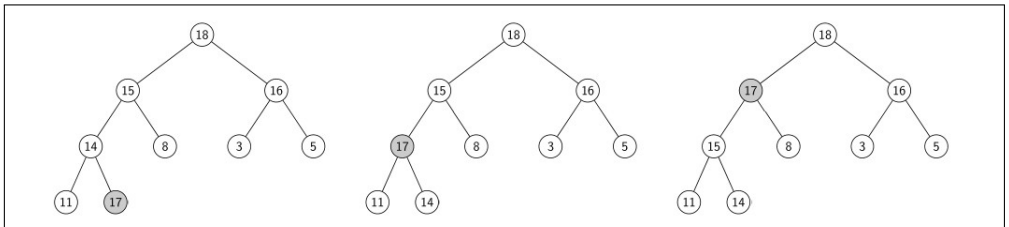

Figure 3. Insertion de l'élément 17 dans un tas codé par `8,[|18;15;16;14;8;3;5;11;...|]`.

Écrire une fonction `insertion : 'a → int → 'a array → unit` permettant d'insérer un élément dans un tas. Si `c`, un tableau de longueur $>$`t`, code le contenu d'un tas à `t` nœuds, `insertion x t c` ne retourne rien mais a pour effet de bord de modifier `c` pour qu'il code un tas à `t`+1 nœuds ayant les mêmes nœuds qu'auparavant, plus le nœud x, à sa place.

Complexité : la complexité de `insertion x t c` est .

I.4 Retrait de l'élément maximal d'un tas modifiable

On souhaite maintenant retirer l'élément maximal d'un tableau `c` codant un tas à `t` nœuds. L'élément maximal est `c.(0)`, il est donc facile à obtenir. Mais comment préserver ensuite la structure de tas ? L'idée est la suivante : on permute `c.(0)` avec `c.(t−1)` puis on fait descendre l'élément permuté à sa place, en le permutant avec le plus grand de ses deux fils tant qu'il est plus petit que ce dernier.

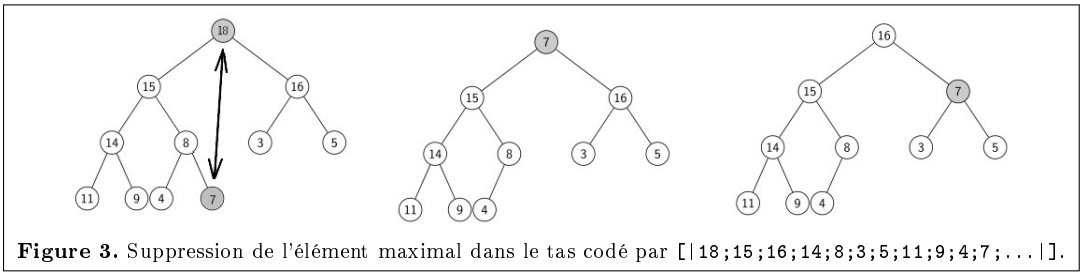

Figure 3. Suppression de l'élément maximal dans le tas codé par `[|18;15;16;14;8;3;5;11;9;4;7;...|]`.

Écrire une fonction `sortir_max : int → 'a array → 'a` permettant de sortir l'élément maximal d'un tas. Si `c`, un tableau de longueur $\geqslant 1$, code un tas à `t`$\geqslant 1$ nœuds, `sortir_max t c` retourne l'étiquette de la racine du tas et a pour effet de bord de modifier `c` pour qu'il code un tas à `t`-1 nœuds ayant les mêmes nœuds qu'auparavant,à leur place, sauf la racine.

Complexité : la complexité de `sortir_max t c` est .

II Implémentation des files de priorité par des tas

II.1 Pourquoi utiliser des tas ?

- Une première idée est de les implémenter par des listes.
  Problème : dans tous les cas, le retrait de l'élément de priorité maximale est en $O(n)$.
- Pour résoudre le problème on peut les implémenter par des listes **triées**.
  Problème : dans le pire des cas, l'ajout d'un nouvel élément est dans le pire des cas en $O(n)$.
- La solution retenue à l'aide des tas permet de réaliser ces deux opérations en temps $O(\lg(n))$. C'est la fête.

Dans la suite, on se limite à l'implémentation modifiable des files de priorité en utilisant l'implémentation modifiable des tas.

On peut par exemple utiliser le type `'a*'b` pour les étiquettes où `'b` est le type des éléments et `'a` le type des priorités :

```
type ('a,'b) file_de_priorite = ('a*'b) tas;;
```

II.2 File de priorité vide

Écrire les fonctions `creer_fp_vide` et `est_fp_vide`, où `est_fp_vide` est de type `'a file_de_priorite → bool` et où on pourra considérer que `creer_fp_vide` est de type `int → 'a file_de_priorite` où l'entier donné en argument donne la longueur maximale de la file de priorité. Ces fonctions doivent être de coût constant.

II.3 Insertion dans une file de priorité.

On utilisant `insertion`, écrire une fonction `inserer : 'a → 'b → 'a file_de_priorite → 'a file_de_priorite`. Cette fonction doit être logarithmique en la taille de la file de priorité.

II.4 Insertion dans une file de priorité.

On utilisant la fonction `sortir_max`, écrire une fonction `retire_prioritaire : 'a file_de_priorite → 'a`. Cette fonction doit être logarithmique en la taille de la file de priorité.

III Applications

III.1 Tri par file de priorité

On rappelle le principe du tri par insertion sur les listes : on prend un à un les éléments de la liste à trier et on les insère à leur place dans une liste initialement vide.

C'est lent car les insertions à leur place dans une liste sont au pire cas de coût linéaire, ce qui conduit à une complexité quadratique. Maintenant si au lieu d'insérer les éléments à leur place dans une liste initialement vide, on les insère dans une `('a,'a) file_de_priorite` initialement vide (les priorités étant les valeurs de l'élément elles-mêmes), chaque insertion est de coût logarithmique, donc la file de priorité est remplie au pire cas en $O\big(n\lg(n)\big)$. Si on la vide ensuite dans une liste initialement vide, on obtient une liste triée contenant les mêmes éléments que la liste initiale, et chaque insertion est logarithmique, ce qui permet finalement d'obtenir un algorithme de tri en $O\big(n\lg(n)\big)$.

Écrire une telle fonction `tri_par_file_de_priorite : 'a list → 'a list`.

III.2 Tri par tas

On peut faire mieux si on souhaite trier un tableau plutôt qu'une liste : plutôt qu'utiliser une file de priorité auxilliaire, on peut utiliser le tableau à trier lui-même comme un tas, en utilisant les fonctions de la partie I.
Ceci permet de trier le tableau sans utiliser de tableau auxiliaire : on dit qu'on a un tri **en place**.

Écrire une telle fonction `tri_par_tas : 'a array → unit`.