Programmation : TP 16, tri et maximier

Un arbre binaire partiellement ordonné est un arbre binaire qui vérifie la propriété suivante : tout nœud est inférieur (pour un ordre fixé) à ses fils. Exemple figure **??**.

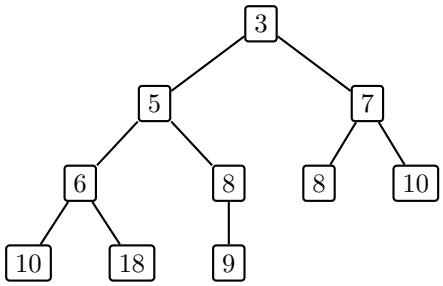

Fig. 1 – Un arbre partiellement ordonné

Insertion

L'insertion d'un nouvel élément dans un arbre partiellement ordonné se fait de la manière suivante :

- On compare l'élément x à ajouter à la racine r. Si l'élément x est plus petit que la racine, x doit être la racine du nouveau maximier : on insère r (récursion) dans l'un de ses fils. Sinon, on insère x dans l'un des fils de r.
- Le choix du fils pour l'insertion récursive est délicat : si on choisit toujours le même (e.g. le droit), l'arbre produit sera déséquilibré. L'astuce consiste à faire tout de même de cette façon mais à inverser dans le même temps le fils droit et le fils gauche.

Exemple : on insère 4 dans l'arbre de la figure **??**. Puisque 4 est plus grand que la racine (3), on l'insère dans le sous-arbre droit qu'on échange avec le sous-arbre gauche (figure **??**). Et ainsi de suite (figures **??** et **??**).

Quelle est la complexité de l'opération d'insertion (en fonction de la taille de l'arbre) ?

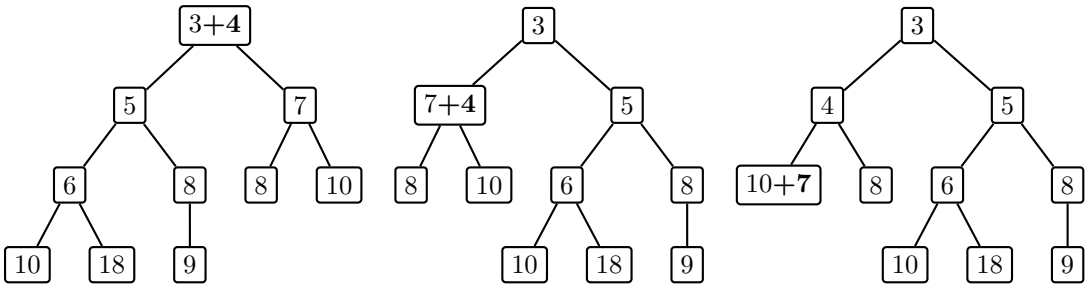

Fig. 2 – Insertion dans un arbre partiellement ordonné

Suppression du plus petit élément

Le plus petit élément d'un arbre partiellement ordonné se trouve à sa racine. Pour le supprimer :

- on fait « remonter » le plus petit des deux fils à la place de la racine ;
- on répète l'opération récursivement.

Dans l'exemple, on obtient les arbres des figures **??** (suppression de 3, remontée de 5) et **??** (remontée de 6), et **??** (remontée de 10).

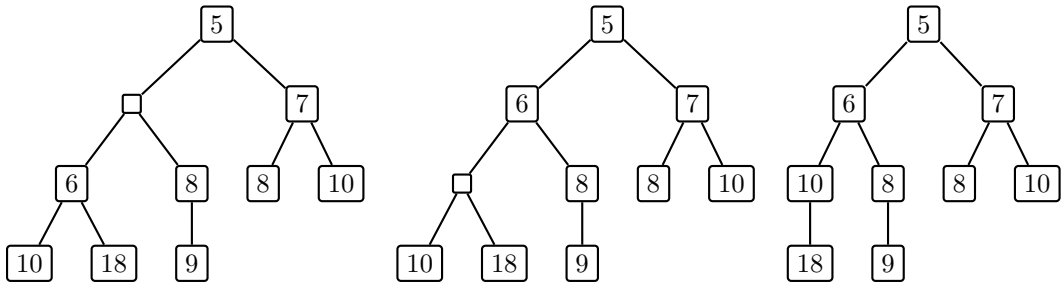

Fig. 3 – Suppression du plus petit élément

Quelle est la complexité de l'opération de suppression ?

Codage en Caml

On défini le type `'a maximier` des arbre maximiers par :

```
#type 'a maximier = Vide
#                 | Noeud of 'a*('a maximier)*('a maximier);;
type 'a maximier = Vide | Noeud of 'a * 'a maximier * 'a maximier
```

Remarque : On représente le cas d'un maximier vide (ne contenant aucun élément). Ainsi une feuille est représentée par un nœud dont les deux fils sont des arbres vides.

1. Écrire la fonction

```
#insertion;;
 - : ('a -> 'a -> bool) -> 'a -> 'a maximier -> 'a maximier = <fun>
```

   qui ajoute un élément à un maximier.

2. Écrire la fonction

```
#suppression;;
 - : ('a -> 'a -> bool) -> 'a maximier -> 'a * 'a maximier = <fun>
```

   qui extrait le plus petit élément d'un maximier (le premier argument est la fonction de comparaison de deux éléments du maximier). La fonction retourne le plus petit élément **et** le nouveau maximier (ou lève une exception). Indication : commencer par écrire la fonction plus simple `ote_sommet` qui supprime le sommet d'un maximier et rend uniquement le nouvel arbre.

3. On veut trier une liste L. L'idée du tri maximier est de ranger les éléments de L dans un arbre binaire partiellement équilibré puis de les extraire un par un, dans l'ordre croissant pour construire une nouvelle liste triée.

   Écrire cette fonction `tri` en utilisant les fonctions précédentes.

```
#tri (<) [4;3;6;1;1;3;7;8];;
 - : int list = [1; 1; 3; 3; 4; 6; 7; 8]
```

   Remarque : On pourra utiliser la fonction prédéfinie `List.fold_right`.

4. Reprendre le code des question précédentes et structurer en modules :
   - module "Maximier" avec son interface "maximier.mli" où on déclarera le type `maximier` comme abstrait (i.e. sans définition, c-à-d sans `= ...`) ;
   - module "Tri".

   NB : l'abstraction du type nécessite l'ajout d'une fonction `est_vide` dans le module `Maximier`.