TP 2 : Arbres d'intervalles

Les arbres d'intervalles sont des structures de données qui permettent de gérer une famille d'intervalles. Plus précisément, cette structure de données permet de trouver efficacement tous les intervalles qui chevauchent un intervalle ou un point donné. Elle est souvent utilisée pour des requêtes de fenêtrage (quand plusieurs fenêtres numériques sont ouvertes simultanément, quelle portion de chacune d'elles afficher à l'écran ?) ou encore pour déterminer les éléments visibles à l'intérieur d'une scène en trois dimensions. Durant ce TP, nous ne prendrons en compte que des segments $[a, b]$ représentés par un couple d'entiers (a, b), ce qui nous amène à définir le type :

```
type intervalle = int * int ;;
```

On dit que deux intervalles I et I' se chevauchent lorsque $I \cap I' \neq \emptyset$. Il est facile de constater qu'un couple d'intervalles (I, I') ne peut vérifier qu'une et une seule des trois propriétés suivantes :

(i) I est à gauche de I' ;

(ii) I et I' se chevauchent ;

(iii) I est à droite de I'

Question 1. Écrire une fonction `chevauche` de type `intervalle -> intervalle -> bool` indiquant si deux intervalles donnés se chevauchent. Rappel : pour forcer le type `intervalle` en entrée d'une fonction `f`, la définir comme `let f (i:intervalle)=...` ou encore `let f ((a,b):intervalle)=....` On rappelle aussi que `fst` et `snd` sont deux fonctions donnant les première et seconde composantes d'un couple.

```
#   chevauche i1 i2, chevauche i1 i3, chevauche i2 i3 ;;
- : bool * bool * bool = (true, true, false)
```

Un *arbre d'intervalles* est un arbre binaire de recherche dont les données sont des intervalles et les clés les extrémités de ceux-ci (ordonnés suivant l'ordre lexicographique, en particulier l'énumération infixe des nœuds de l'arbre ne gardant que les bornes gauches est croissante). En plus des intervalles eux-mêmes, chaque nœud x contient une valeur qui représente la valeur maximale parmi les extrémités d'intervalles stockés dans le sous-arbre enraciné en x, voir figure 1.

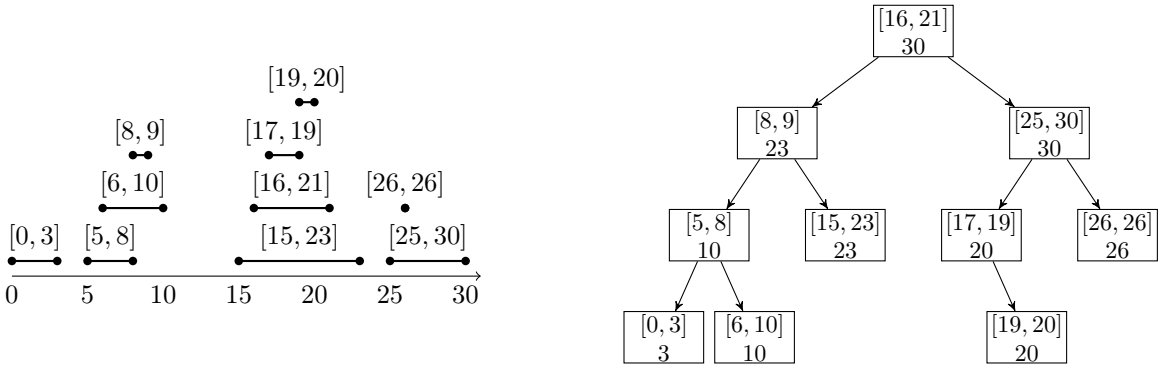

FIGURE 1: Des intervalles et un arbre binaire associé

Pour représenter les arbres, on utilise le type :

```
type arbre = V | N of intervalle * int * arbre * arbre ;;
```

Question 2. Écrire une fonction `maxi: arbre -> int` qui prend en argument un arbre d'intervalles supposé non vide et renvoie la valeur maximale parmi les extrémités d'intervalles stockés dans cet arbre. On impose une complexité $O(1)$!

```
# maxi exemple_1 ;;
- : int = 30
# maxi V ;;
Exception: Failure "vide".
```

Vérification. Dans l'annexe vous sont données des fonctions permettant la vérification des propriétés d'arbres d'intervalle, que vous n'utiliserez que pour vous assurer que vos fonctions sont correctes.

Question 3. Écrire une fonction `recheche: arbre -> intervalle -> intervalle` qui prend en argument un arbre d'intervalles et un intervalle donné I, et renvoie un intervalle J chevauchant I s'il en existe un. S'il n'en existe pas, on lèvera l'exception `Not_found` avec `raise Not_found`. On impose une complexité linéaire en la hauteur de l'arbre. *Remarque : réfléchissez avant d'écrire votre fonction !*

```
# recherche exemple_1 (21,22) ;;
- : intervalle = (16, 21)
# recherche exemple_1 (22,23) ;;
- : intervalle = (15, 23)
# recherche exemple_1 (24,24) ;;
Exception: Not_found.
# recherche exemple_1 (0,11) ;;
- : intervalle = (8, 9)
```

Question 4. Écrire une fonction `liste_tous: arbre -> intervalle -> intervalle list` renvoyant la liste de tous les intervalles chevauchant un intervalle donné. On impose une complexité linéaire en le produit hauteur de l'arbre $\times$ taille du résultat.

```
# liste_tous exemple_1 (0,11) ;;
- : intervalle list = [(8, 9); (5, 8); (0, 3); (5, 8)]
```

Question 5. Écrire une fonction `insere: arbre -> intervalle -> arbre` réalisant l'insertion d'un intervalle dans un arbre d'intervalles. On procédera comme dans le cours, et on renverra l'arbre à l'identique si l'intervalle est déja présent.

```
# enum_infixe (insere exemple_1 (1,7)) ;;
- : intervalle list =
[(0, 3); (1, 7); (5, 8); (6, 10); (8, 9); (15, 23); (16, 21); (17, 19); (19, 20); (25, 30); (26, 26)]
# verif (insere exemple_1 (1,7)) ;;
- : bool = true
```

On veut maintenant gérer la suppression d'un intervalle.

Question 6.

1. Écrire une fonction `cons i g d` prenant en entrée un intervalle i et deux arbres d'intervalles g et d, tels que i est strictement supérieur aux éléments de g (pour l'ordre lexicographique) et strictement inférieur à ceux de d, et renvoie l'arbre d'intervalles de racine i et de sous-arbres gauche et droit g et d.
2. Écrire une fonction `fusion g d` prenant en entrée deux arbres d'intervalles tels que les nœuds du premier soient strictement inférieurs au nœuds du second, et renvoyant un arbre d'intervalles contenant les nœuds de g et d. On procédera comme dans le cours.
3. En déduire une fonction `suppr t i` supprimant de l'arbre `t` l'intervalle `i`. On renverra l'arbre à l'identique si `i` n'est pas présent.

```
# enum_infixe (suppr exemple_1 (6,10)) ;;
- : intervalle list =
[(0, 3); (5, 8); (8, 9); (15, 23); (16, 21); (17, 19); (19, 20); (25, 30); (26, 26)]
# verif (suppr exemple_1 (6,10)) ;;
- : bool = true
```

Remarque : il est possible d'équilibrer les arbres d'intervalles, de façon à garantir l'efficacité des opérations.