Travaux pratiques

option informatique

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 prendront 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. Rédiger une fonction **chevauche** qui prend en arguments deux intervalles I et J et qui retourne un booléen traduisant le chevauchement ou non de I et de J.

```
chevauche : intervalle -> intervalle -> bool
```

Un *arbre d'intervalles* est un arbre binaire de recherche dont les données sont des intervalles et les clés les extrémités gauches de ceux-ci. 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.

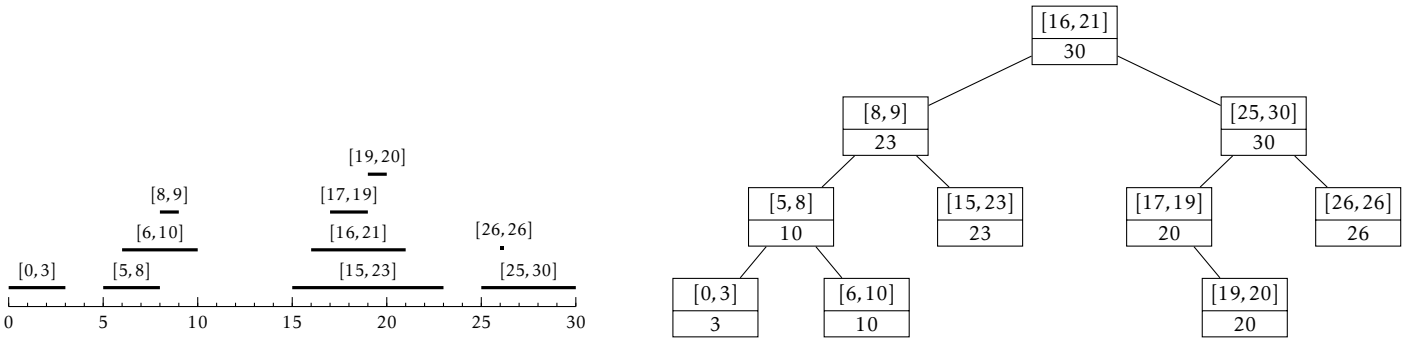

FIGURE 1 – Un ensemble de 10 intervalles, et l'arbre d'intervalles qui les représente.

Pour les représenter, on définit le type :

```
type int_tree = Nil | Noeud of intervalle * int * int_tree * int_tree ;;
```

Question 2. Rédiger une fonction **maxi** 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.

```
maxi : int_tree -> int
```

Question 3. Rédiger une fonction **recherche** qui prend en arguments un intervalle I et un arbre d'intervalles A, et qui retourne un intervalle de A qui chevauche I s'il en existe ou déclenche l'exception **Not_found** dans le cas contraire.

```
recherche : intervalle -> int_tree -> intervalle
```

On impose un temps d'exécution en $O(h(A))$, où $h(A)$ désigne la hauteur de l'arbre A.

Question 4. Rédiger une fonction **cons** qui prend en arguments un intervalle I et deux arbres d'intervalles A_1 et A_2, et qui retourne l'arbre d'intervalles dont la racine est étiquetée par I et les fils gauche et droit égaux respectivement à A_1 et A_2.

```
cons : intervalle -> int_tree -> int_tree -> int_tree
```

Question 5. Rédiger une fonction **insere** qui prend en arguments un intervalle I et un arbre d'intervalles A, et qui retourne un nouvel arbre A′ dans lequel l'intervalle I aura été inséré à la racine.

```
insere : intervalle -> int_tree -> int_tree
```

On impose un temps d'exécution en $O(h(A))$, où $h(A)$ désigne la hauteur de l'arbre A.

Remarque. On ne se préoccupera pas de savoir si I est déjà présent dans l'arbre A.

Question 6. Rédiger une fonction **supprime** qui prend en arguments un intervalle I et un arbre d'intervalles A, et qui retourne un nouvel arbre A′ dans lequel l'intervalle I aura été supprimé (s'il se trouve dans A).

```
supprime : intervalle -> int_tree -> int_tree
```

On impose un temps d'exécution en $O(h(A))$, où $h(A)$ désigne la hauteur de l'arbre A.