Principes de Programmation
Première Phase du Projet de Programmation
À rendre pour le 10 février minuit

5 février 2019

Objectif : Premier pas en Haskell, manipulation des arbres.

Il s'agit, dans cette première phase, de pouvoir manipuler des arbres binaires de recherche par intervalles (ABRI) .

Un arbre binaire de recherche par intervalles est un arbre binaire dont chaque noeud est étiqueté par un couple (n_1, n_2) d'entiers relatifs, appelés bornes des clés de sorte à ce que tout entier $n_1 \leq n \leq n_2$ est appelé clé ; et avec la propriété suivante : si un noeud est étiqueté par (n_1, n_2), alors son fils gauche ne contient que des clés strictement plus petites que $n_1 - 1$ et son fils droit des clés strictement plus grandes que $n_2 + 1$.

Des exemples sont dispo sur les slides passés en cours.

Vous devez reprendre le canevas préparé et le remplir avec :

1. un type pour ces arbre binaires de recherche par intervalles,
2. une creer qui cré un arbre vide,
3. une fonction de recherche,
4. une fonction d'insertion d'une clé,
5. une fonction qui imprime le contenu d'un arbre comme une suite de segments de la forme `[1..3][5..12][42..230]`,
6. une fonction de taille : elle rend le nombre de clés dans l'arbre, c'est à dire la somme des longueurs des intervalles),
7. une fonction de taille réelle : elle rend le nombre de noeuds (ou d'intervalles) dans l'arbre,
8. ($\star$) une fonction de suppression,
9. ($\star$) une fonction d'insertion d'une liste : on doit insérer un par un tous les éléments de la liste,
10. ($\star$) une fonction d'insertion d'un intervalle : on dispose de deux entier (début et fin de l'intervalle) et on doit insérer tous les éléments dans cet intervalle, de préférence sans le faire un par un mais en fusionnant avec les intervalles de l'arbre (attention a bien comprimer l'arbre résultant).

Les 7 premiers points sont obligatoires pour rendre le projet valide (autrement il ne passera pas le teste automatique). Les trois dernières fonctions ne sont pas obligatoires, mais rapportent des points.

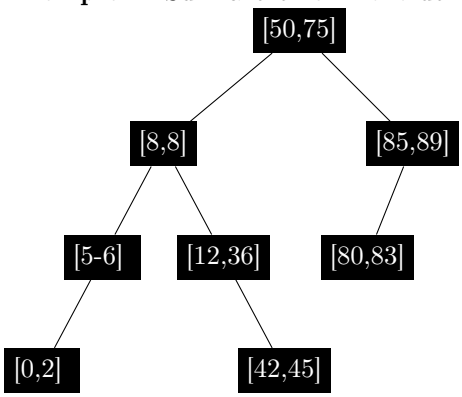
Pour les courageux qui veulent s'avancer, il y a en plus un défi :

`Maybe` est une structure de donnée paramétrée (comme liste par exemple), un élément de type `Maybe Int` peut être soit une donnée vide appelée `Nothing` soit un entier encapsulé par le constructeur `Just`, comme `Just 3`. Formellement, il y a le code suivant dans la bibliothèque standard

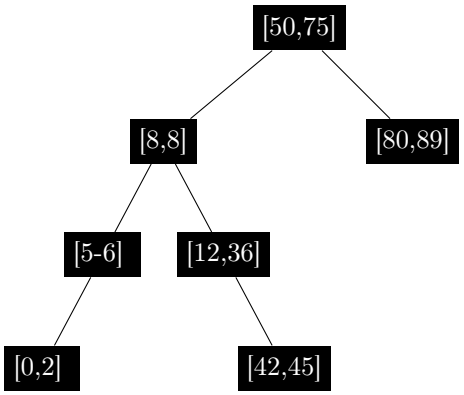
```
data Just a = Nothing | Just a
```

Le but du défi est de comprendre cette structure et de réécrire une suppression `supprimerM` qui retourne son résultat sous forme d'un maybe : on retourne `Nothing` si on n'a rien trouvé, et le nouvel arbre encapsulé si on a effectivement supprimé l'élément.

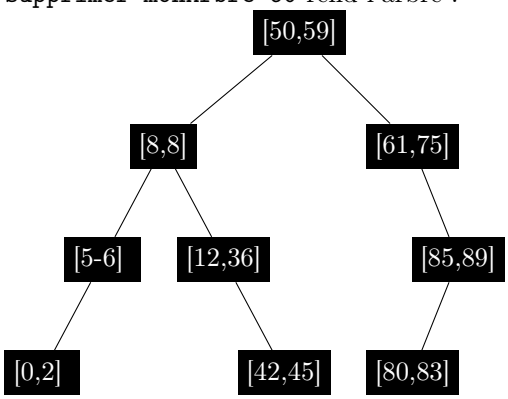
Exemple : Sur l'arbre `monArbre` definit comme :



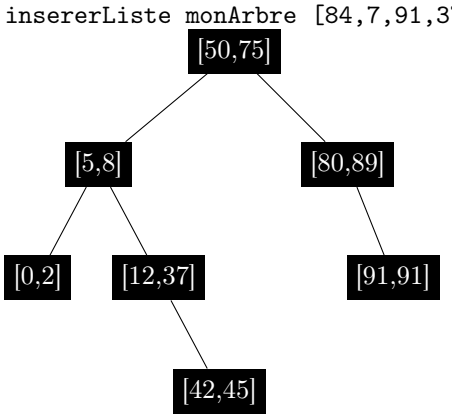
On a les résultats :
`recherche monArbre 25` rend `True`,
`recherche monArbre 48` rend `False`,
`imprimer monArbre` rend `[0,2][5,6][8,8][12,36][42,45][50,75][85,89]`,
`taille monArbre` rend `65`,
`tailleReelle monArbre` rend `7`
`inserer monArbre 84` rend l'arbre :



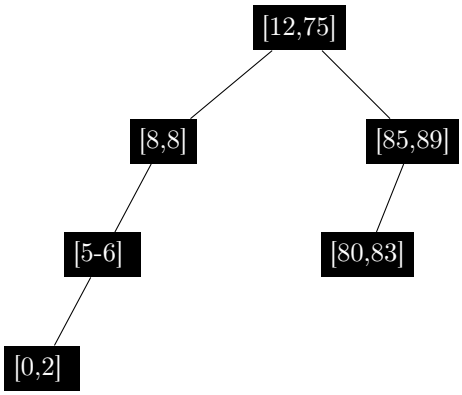
`supprimer monArbre 60` rend l'arbre :



`insererListe monArbre [84,7,91,37]` rend l'arbre :



`insererListe monArbre 25 55` rend l'arbre :


[12,75]
[8,8]
[85,89]
[5-6]
[80,83]
[0,2]