Les algorithmes seront écrits en code CAML et seront accompagnés de commentaires et d'explications qui permettront de les comprendre aisément.

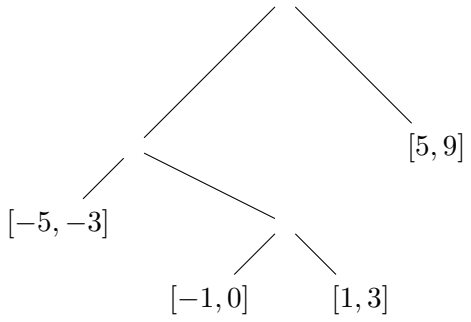
Problème 1 - Arbres d'intervalles

Soit A une partie de $\mathbb{R}$. On dit que A est de type RI quand A est la réunion finie de segments disjoints (on obtient l'ensemble vide quand A est une réunion de 0 segment). On supposera toujours que les intervalles sont rangés dans l'ordre croissant des bornes (on dira que l'intervalle $[a, b]$ est inférieur à l'intervalle $[c, d]$ quand $a \leqslant b < c \leqslant d$).

Par exemple, $\varnothing$, $[-1, 5]$, $[1, 5] \cup [8, 12]$, $[0, 1] \cup [2, 3] \cup [4, 5]$ sont des ensembles de type RI.

On représente un ensemble A de type RI par un arbre binaire qui est
- soit vide, quand $A = \varnothing$;
- soit réduit à une feuille, quand A est un segment non vide ;
- soit un nœud sans étiquette, dont les deux fils représentent chacun un morceau de A.

Par exemple, l'arbre suivant représente l'ensemble $A = [-5, -3] \cup [-1, 0] \cup [1, 2] \cup [5, 9]$:



On impose cependant quelques contraintes sur de tels arbres : tout intervalle situé dans le fils gauche d'un nœud doit être inférieur à tout intervalle situé dans le fils droit. Un tel arbre sera dit « bien formé ». On peut remarquer qu'il n'y a pas unicité de l'arbre représentant un ensemble de type RI.

Pour représenter de tels arbres en CAML, on crée le type suivant (pour simplifier le code, on supposera que les bornes sont entières désormais) :

```
type arbre = Vide | Feuille of int * int | Noeud of arbre * arbre;;
```

L'arbre précédent est donc noté

```
let a = Noeud(
  Noeud(
    Feuille(-5,-3),
    Noeud(
      Feuille(-1,0),
      Feuille(1,3)
    )
  ),
  Feuille(5,9)
);;
```

Question 1) Donnez une fonction mini a, de type `arbre -> int`, qui calcule le minimum de l'ensemble A représenté par l'arbre bien formé a. Dans le cas d'un arbre vide, on interrompt le calcul par un appel à failwith .

Faites de même avec le maximum.

Quelle est la complexité de vos fonctions en fonction de h, hauteur de l'arbre ?

Question 2) Écrivez une fonction liste a, de type `arbre -> int list`, qui calcule la liste des bornes des segments de l'ensemble A dans l'ordre gauche-droite.

Par exemple, avec l'arbre a ci-dessus, liste a = [−5; −3; −1; 0; 1; 2; 5; 9].

Question 3) Á quelle condition sur liste a l'arbre a est-il bien formé ? Écrivez une fonction verifie a de type `arbre -> bool` qui détermine si l'arbre a est bien formé.

Question 4) Écrivez une fonction appartient v a, de type `int -> arbre -> (bool * int * int)`, qui calcule un triplet (b, x, y) tel que

— si v appartient à l'ensemble A, $b = true$ et x, y sont les bornes du segment $[x, y]$ de A qui contient v ;
— sinon $b = false$ et x, y sont ce que vous voulez.

Question 5) Écrivez une fonction decoupe v a, de type `int -> arbre -> (arbre * arbre * arbre)`, qui calcule le triplet a1, a2, a3 tel que

— a1 est un arbre bien formé représentant l'union des segments de A égale à $A \cap]-\infty, v[$;
— a2 est un arbre bien formé représentant le segment de A contenant v ;
— a3 est un arbre bien formé représentant l'union des segments de A égale à $A \cap]v, +\infty[$.

Question 6) La fusion d'un certain nombre de segments est le plus petit segment qui les contient.

Écrivez une fonction ajoute (u,v) a, de type `int * int -> arbre -> arbre`, qui calcule un arbre représentant la partie B de type RI, obtenue à partie de A en conservant les segments disjoints de $[u, v]$ et en fusionnant en un seul segment ceux qui ne sont pas disjoints de $[u, v]$.

Avec A la partie définie en exemple ci-dessus et $[u, v] = [2, 4]$, alors $B = [-5, -3] \cup [-1, 0] \cup [1, 4] \cup [5, 9]$.

Avec A la partie définie en exemple ci-dessus et $[u, v] = [2, 6]$, alors $B = [-5, -3] \cup [-1, 0] \cup [1, 9]$.

Avec A la partie définie en exemple ci-dessus et $[u, v] = [10, 11]$, alors $B = [-5, -3] \cup [-1, 0] \cup [1, 3] \cup [5, 9] \cup [10, 11]$.

Question 7) Écrivez une fonction reunit a b de type `arbre -> arbre -> arbre` qui calcule un arbre bien formé représentant la réunion des ensembles A et B, eux-mêmes représentés par les arbres bien formés a et b.

Problème 1

Question 1)

```
let rec mini a =
  match a with
  | Vide -> failwith "arbre vide"
  | Feuille(x, y) -> x
  | Noeud(Vide, fd)-> mini fd
  | Noeud(fg, fd) -> mini fg;;
```

Les cas de base sont évidents. Dans le cas d'un nœud, on étudie d'abord si le fils gauche est vide : si oui, le minimum est dans le fils droit, sinon comme l'arbre est bien formé, il est forcément dans le fils gauche.

```
let rec maxi a =
  match a with
  | Vide -> failwith "arbre vide"
  | Feuille(x, y) -> y
  | Noeud(fg, Vide)-> maxi fg
  | Noeud(fg, fd) -> maxi fd;;
```

Comme à chaque appel récursif, on n'explore qu'un seul côté de l'arbre, on suit donc un chemin qui mène de la racine à la feuille la plus à gauche (ou à droite) de l'arbre, donc le nombre d'appels récursifs est inférieur à la hauteur de l'arbre. De plus, les opérations effectuées sont élémentaires, donc la complexité est en $O(h)$.

Plus formellement, la complexité vérifie une relation de récurrence de la forme $C(h) = C(h-1) + O(1)$, donc $C(h) = O(h)$.

Question 2) C'est en fait un parcours en profondeur de l'arbre (inutile de distinguer préfixe, infixe ou postfixe, puisque qu'il n'y a pas de valeur à traiter dans les nœuds.

```
let rec liste a =
  match a with
  | Vide -> []
  | Feuille(x, y) -> [x; y]
  | Noeud(fg, fd) -> liste fg @ liste fd;;
```

Question 3) L'arbre est bien formé si et seulement si il est vide, réduit à une feuille ou alors si ses fils sont eux-mêmes bien formés et le maximum du fils gauche est inférieur ou égal au minimum du fils droit.

```
let rec verifie a =
  match a with
  | Vide | Feuille(_,_) -> true
  | Noeud(fg, fd) -> verifie fg && verifie fd && maxi fg < mini fd;;
```

Question 4) Sans difficultés particulières.

```
let rec appartient v a =
  match a with
  | Vide -> (false, 0, 0)
  | Feuille(x, y) -> (x <= v && v <= y, x, y)
  | Noeud(fg, fd) -> let (b, x, y) = appartient v fg in
    if b then (b, x, y)
    else appartient v fd;;
```

Question 5) Les cas de base sont clairs. Le cas récursif mérite quelques explications : on calcule d'abord les minimum et maximum de l'arbre, si v est en dehors de cet intervalle, alors c'est facile (cas 1 et 2) ; sinon, on regarde si v est dans l'intervalle englobant du fils gauche (auquel cas, on découpe le fils gauche), entre le fils gauche et le droit, ou s'il est dans l'intervalle englobant du fils droit (on découpe alors le fils droit).

```
let englobant a =
  (mini a, maxi a);;

let rec decoupe v a =
 match a with
 | Vide -> (Vide,Vide,Vide)
 | Feuille(x,y) -> if v < x then (Vide,Vide,a)
   else if v <= y then (Vide, a, Vide)
   else (a,Vide,Vide)
 | Noeud(fg,fd) ->
   let (x,t) = englobant a in
   if v < x then (Vide,Vide,a) (* 1 *)
   else if t < v then (a,Vide,Vide) (* 2 *)
   else let (_,y) = englobant fg and (z,_) = englobant fd in
     if v <= y then let (g1,g2,g3) = decoupe v fg in (g1,g2,Noeud(g3,fd))
     else if v < z then (fg,Vide,fd)
     else let (d1,d2,d3) = decoupe v fd in (Noeud(fg,d1),d2,d3);;
```

Question 6) On découpe l'arbre selon les valeurs de u et v. Les cas 1 et 2 indiquent que le segment est inférieur ou supérieur à tous ceux de l'arbre, on l'ajoute donc du bon côté. Sinon on regarde si u et v sont dans l'arbre : si u n'y est pas et v y est, alors on fusionne le segment $[u, v]$ avec celui contenant v et on le place au bon endroit dans l'arbre (au milieu) ; de même dans les autres cas.

```
let ajoute (u,v) a =
  let (u1,u2,u3) = decoupe u a and (v1,v2,v3) = decoupe v a in
  if u2 = Vide && u3 = Vide then Noeud(a, Feuille(u,v)) (* 1 *)
  else if v1 = Vide && v2 = Vide then Noeud(Feuille(u,v), a) (* 2 *)
  else match (u2, v2) with
          | Vide, Feuille(_,y) -> Noeud (Noeud(u1, Feuille(u,y)), v3)
          | Feuille(x,_), Vide -> Noeud (Noeud(u1, Feuille(x,v)), v3)
          | Feuille(x,_),Feuille(_,y) -> Noeud (Noeud(u1, Feuille(x,y)), v3)
          | _, _ -> Noeud (Noeud(u1, Feuille(u,v)), v3);;
```

Question 7)

```
let rec reunit a b =
  match a with
  | Vide -> b
  | Feuille(x, y) -> ajoute (x,y) b
  | Noeud(fg, fd) ->
    let b' = reunit fg b in
    reunit fd b';;
```

Simple utilisation récursive de la fonction précédente : on ajoute les segments de gauche puis ceux de droite (on pourrait faire le contraire).