# DS5 MP2I

Durée 2h

Calculatrices autorisées

## Consignes

— La lisibilité des code et leur concision sont des facteurs d'appréciations.

— Sauter des lignes et laisser une marge suffisante.

— Pas de code long sans explication.

— Toute fonction qui n'est pas demandée explicitement par le sujet doit être accompagnée d'une description précisant ce qu'elle est censée faire.

— Répondre aux questions dans l'ordre quitte à laisser des blancs.

## Exercices

**Exercice 1.** On donne la structure suivante qui représente les arbres binaires d'entiers.

```
typedef struct _btree_t{
  int label;
  struct _btree_t *left;
  struct _btree_t *right;
}btree_t;
```

1. Écrire la fonction

```
btree_t* inode(int x, btree_t* left, btree_t* right)
```

qui renvoie un arbre de fils gauche `left`, droit `right` et d'étiquette `x`.

2. Écrire la fonction `btree_t * miroir (btree_t *bt)` qui renvoie un pointeur sur l'arbre miroir de celui passé en paramètre.
3. Écrire la fonction `bool search(btree_t *bt, int x)` qui cherche si $x$ est une étiquette de l'arbre pointe par `bt`.
4. Écrire `int hauteur(btree_t *bt)` qui retourne la hauteur de l'arbre pointé par `bt`.
5. Écrire la fonction `somme(btree_t *bt)` qui renvoie la somme des étiquettes de l'arbre sur lequel pointe `bt`.

*Solution.* Voici

```
// créer un node interne
btree_t* inode(int x, btree_t* left, btree_t* right){
  btree_t *n = malloc(sizeof(btree_t));
  n->label = x;
  n->left = left;
  n->right = right;
  return n;
}


void _somme(btree_t *bt, int * s){
  if (bt!=NULL){
    *s=bt->label + *s;
    _somme(bt->left,s);
    _somme(bt->right,s);
  }
}

int somme(btree_t *bt){
  int x = 0;
  int * s =&x;
```

```
  _somme(bt,s);
  return x;
}

btree_t * miroir (btree_t *bt){
  if (bt != NULL){
      return inode(bt->label, miroir(bt->right),miroir(bt->left));}
  return NULL;
}

int _max(int a, int b){
  if (a>=b)
    return a;
      return b;
}

int hauteur(btree_t *bt){
  if (bt == NULL)
    return -1;
  return 1+_max(hauteur(bt->left),hauteur(bt->right));
}

```

□

**Exercice 2.** On donne le type suivant des arbres arithmétiques et un exemple d'instance :

```
type arith_tree =
    | V of int
    | Plus of arith_tree * arith_tree (*Somme*)
    | Fois of arith_tree * arith_tree (*Produit*)
    | Opp of arith_tree;;(*Opposé d'un nb*)

let t1 = Opp ( Plus (V 5, Fois (V 4, V 2) )) ;; (*un exemple*)
```

Ecrire la fonction `eval` de type `arith_tree -> int` qui évalue un arbre arithmétique.

Par exemple `eval t1` retourne $-13$.

*Solution.* Voici

```
let rec eval a= match a with
   | V v-> v
   | Opp f-> - eval f
   | Plus(g,d)-> (eval g) + (eval d)
   | Fois(g,d)-> (eval g) * (eval d);;
```

□

**Exercice 3.**

1. Soit $u_n$ le nombre de squelettes d'arbres binaires à $n$ nœuds.

   Exprimer $u_n$ en fonction de nombres $u_k$ d'indices inférieurs.

2. Soit un squelette $S$ d'arbre binaire à $n$ nœuds et $n$ nombres distincts.

   Combien y a-t-il de façons différentes de placer les $n$ nombres dans $S$ pour en faire un ABR ?

3. Soit $v_n$ le nombre de squelettes d'arbres binaires de recherches à $n$ nombres distincts (il y a $n$ étiquettes et elles sont toutes différentes).

   Evaluer $v_n$.

4. Déterminer le nombre d'arbres binaires de recherche de hauteur 2 associés aux nombres $\{1, 2, 3, 4, 5, 6\}$.

   Justifier.

*Solution.*

1. Si l'arbre est vide $u_0 = 1$, si c'est une feuille alors $u_1 = 0$.

   Pour un arbres à $n > 1$ nœuds dont le le fils gauche est de taille $k$, le droit est d etaille $n - k - 1$. Alors il y a $u_k u_{n-k-1}$ tels arbres.

   En considérant toutes les tailles possibles de fils gauche :

$$u_n = \sum_{k=0}^{n-1} u_k u_{n-k-1}$$

2. On montre qu'un squelette à $n$ nœuds ne peut être rempli que d'une façon.

   C'est vrai pour $n = 0$ et $n = 1$.

   Supposons vraie la propriété pour tout $k \leq n$, tout squelette de taille $n$ et tout ensemble de $n$ nombres distincts.

Soit un squelette à $n+1$ nœuds, de fils gauche à $k$ nœuds et $n+1$ nombres distincts. Notons les $a_1 < \cdots < a_{n+1}$.

Seul le nombre $a_{k+1}$ peut être placé à la racine : ses prédecesseurs doivent être ses decsendants à gauche et les successeurs à droite.

Par HR, il y a une seule façon de remplir le fils gauche avec $a_1, \dots, a_k$ et le droit avec $a_{k+1}, \dots, a_{n+1}$.

Ainsi le squelette ne peut être rempli que d'une façon.

3. $u_n = v_n$, car une fois le squelette choisi, il y a une seule façon de placer les $n$ nombres.
4. On vient de voir qu'il y a une seule façon de remplir un squelette de 6 nœuds avec 6 nombres pour en faire un ABR.

   Il suffit de compter les squelettes d'abr à 6 nœuds de hauteur 2.

   Si il y a un seul nœud en profondeur 1, alors en profondeur 2 il peut y a voir au plus deux nœuds. Mais cela fait $1+1+2=4$ places ; or il en faut 6. Impossible.

   Donc il y a deux nœuds à la profondeur 1. Il y a donc 4 emplacement possibles pour mettre les 3 nombres restant. Cela nous donne $\binom{4}{3} = 4$ possibiltés d'ABR de hauteur 2 pour y placer 6 nombres distincts.

□

**Exercice 4.** On considère le type

```
1 type 'a arbre = Vide | N of 'a * 'a arbre * 'a arbre;;
```

1. On range dans un tableau les étiquettes des feuilles d'un arbre par un parcours en largeur (en respectant leurs positions relatives et en commençant à l'indice 0).

   Cette opération est-elle injective ? Justifier.

2. Dans un tableau sont rangés, à partir de l'indice 0, les étiquettes d'un arbre binaire complet gauche obtenues par un parcours en largeur.

   Écrire une fonction `build` de type `build` de type `'a array -> 'a arbre` qui permet de retrouver l'arbre complet gauche à partir du tableau.

```
# build [|1;2;3;4;5|];;
- : int arbre =
N (1, N (2, N (4, Vide, Vide), N (5, Vide, Vide)), N (3, Vide, Vide))
```

*Solution.* Non injective : `N(1,2,Vide)` et `N(1,Vide,2)` donnent `[|1;2|]`.

On commence en 0 : les fils du nœudd'indice $k$ sont en $2k+1$ et $2k+2$.

```
let build t =
  let rec aux k =
    if k>= Array.length t then Vide
    else
      N(t.(k),aux (2*k+1), aux (2*k+2))
  in aux 0;;
```

□

# Peignes

**Exercice 5.** On suppose défini le type arbre de la manière suivante :

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

On dit qu'un arbre est un *peigne* si tous les nœuds internes (à l'exception éventuelle de la racine) ont au moins une feuille pour fils. On dit qu'un peigne est un peigne *strict* si sa racine a au moins une feuille pour fils, ou s'il est réduit à une feuille. On dit qu'un peigne est *rangé* si le fils droit d'un nœud interne est toujours une feuille. Un arbre réduit à une feuille est considéré comme un peigne rangé.

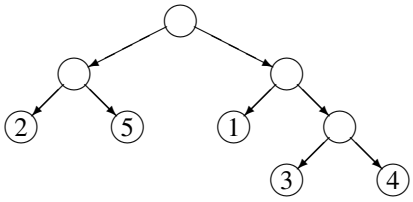
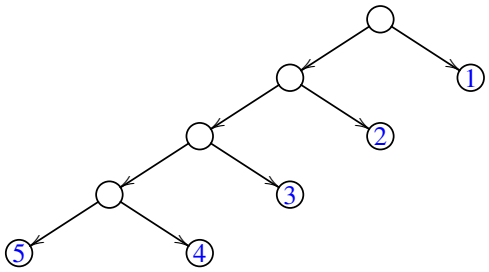



Figure 1 : un peigne à cinq feuilles

1. Représenter un peigne rangé à 5 feuilles.

   *Solution.* On a




   Et voici son codage en OCAML :

```
let r = Noeud(Noeud(Noeud(Noeud(Feuille 5,Feuille 4),
      Feuille 3),Feuille 2),Feuille 1);;
```

   □

2. Quelle est la hauteur d'un peigne rangé à $n$ feuilles ? On justifiera la réponse.

   *Solution.*

   *Remarque.* Dans vos rédactions, indiquez bien à quel endroit vous appliquez l'hypothèse d'induction ou de récurrence.

**Par récurrence** On montre que $n = h + 1$ ($n$ : nombre de feuilles, $h$ :hauteur) par récurrence sur $n$.

Avec cette définition, la hauteur $h = 0$ d'un peigne rangé à $n = 1$ feuille vérifie $n = h + 1$.

Si on a $k = h + 1$ pour tout peigne rangé à $k \leq n$ feuilles, soit un peigne rangé $A$ à $n + 1$ feuilles de hauteur $h$. Son fils droit est une feuille de hauteur 0. Son fils gauche $G$ a pour hauteur $h - 1$ et possède $n$ feuilles. Alors la hauteur de $G$ vérifie, par HR : $n = (h - 1)$. Et donc $n = h + 1$. OK

**Par Induction** Pour un arbre $A$ on note $h(A)$ sa hauteur et $f(A)$ son nombre de feuilles.

Cas de base pour l'arbre feuille : OK.

Soit $A$ un peigne rangé dont le fils gauche $G$ vérifie l'hypothèse d'induction.

Alors $h(G) = f(G) - 1$ (par HI). Or $f(A) = f(G) + 1$ et

$$h(A) = \max(h(G), 1) + 1 = h(G) + 1 = f(G) - 1 + 1 = f(G) = f(A) - 1$$

□

3. Ecrire une fonction `est_range : arbre → bool` qui renvoie `true` si l'arbre donné en argument est un peigne rangé.

   *Solution.* Voici

```
let rec est_range a =
  match a with
  | Feuille _ -> true
  | Noeud(g, Feuille _) -> est_range g
  | _ -> false;;
```

□

4. Ecrire une fonction `est_peigne_strict : arbre → bool` qui renvoie `true` si l'arbre donné en argument est un peigne strict. En déduire une fonction `est_peigne : arbre → bool` qui renvoie `true` si l'arbre donné en argument est un peigne.

   *Solution.* Voici

```
let rec est_peigne_srict a =
  match a with
  | Feuille _ -> true
  | Noeud(g, Feuille _) -> est_peigne_srict g
  | Noeud(Feuille _, d) -> est_peigne_srict d
  | _ -> false;;

let est_peigne a =
  match a with
  | Feuille _ -> true
  | Noeud(g, d) -> est_peigne_srict g && est_peigne_srict d;;
```

□

5. On souhaite ranger un peigne donné. Supposons que le fils droit $D$ de sa racine ne soit pas une feuille. Notons $A_1$ le sous-arbre gauche de la racine, $f$ l'une des feuilles du noeud $D$ et $A_2$ l'autre sous-arbre du noeud $D$. On va utiliser l'opération de *rotation* qui construit un nouveau peigne où

   - le fils droit de la racine est le sous-arbre $A_2$ ;
   - le fils gauche de la racine est un noeud de sous-arbre gauche $A_1$ et de sous-arbre droit la feuille $f$.

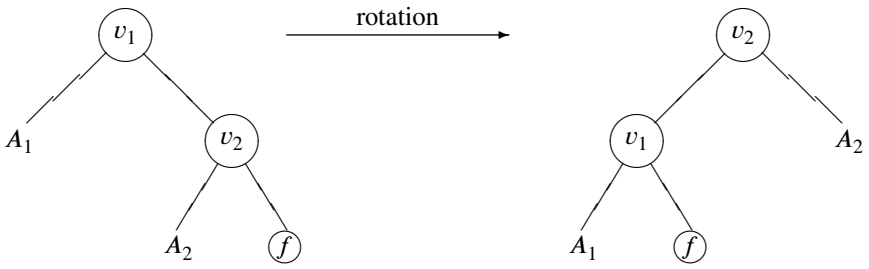
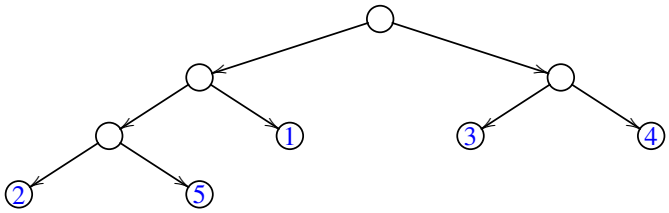



Figure 2 : une rotation

(a) Donner le résultat d'une rotation sur l'arbre de la figure 1.

*Solution.* Après une rotation :




□

(b) Ecrire une fonction `rotation : arbre → arbre` qui effectue l'opération décrite ci-dessus. La fonction renverra l'arbre initial si une rotation n'est pas possible.

*Solution.* Pas de difficulté, mais il faut faire un filtrage un peu détaillé selon que le fils droit de la racine à une feuille à droite ou à gauche :

```
let rotation a =
  match a with
  | Noeud (a1, Noeud (a2, Feuille x)) ->
     Noeud(Noeud (a1, Feuille x),a2)
  | Noeud (a1, Noeud (Feuille x, a2)) ->
     Noeud(Noeud (a1, Feuille x),a2)
  | _ -> a ;;
```

□

(c) Ecrire une fonction `rangement : arbre → arbre` qui range un peigne donné en argument, c'est à dire qu'il renvoie un peigne rangé ayant les mêmes feuilles que celui donné en argument. La fonction renverra l'arbre initial si celui-ci n'est pas un peigne.

*Solution.* On commence par une fonction auxiliaire :

```
let leaf a =
  (*indique si a est une feuille*)
  match a with
  | Feuille _ -> true
  | _ -> false;;
```

```
let rangement a =
  (*pour prévenir les boucles infinies*)
```

```
3   if not est_peigne a then a
4   else
5     begin
6       let rec _ranger a = match a with
7         | Feuille _ -> a
8         | Noeud(g,d) when leaf d -> Noeud (_ranger g, d)
9         | _ -> _ranger(rotation a)
10      in _ranger a;;
11    end;;
```

□

## Arbres d'intervalles

D'après un travail de Jean-Pierre BECIRSPAHIC.

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.

Dans ce devoir, nous ne prennons en compte que des segments $[a, b]$ représentés par un couple d'entiers $(a, b)$, ce qui nous amène à définir le type

```
1   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 :

1. $I$ est à gauche de $I'$ ;
2. $I$ et $I'$ se chevauchent ;
3. $I$ est à droite de $I'$ .

**Q1** Rédiger une fonction `chevauche` de type `intervalle -> intervalle -> bool` 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 (1,5) (3,10);;
- : bool = true
# chevauche (-11,-5) (3,10);;
- : bool = false
```

*Solution.* Pour chaque intervalle, il faut et il suffit que sa borne inférieure soit plus petite que la borne droite de l'autre.

```
let chevauche (i1 : intervalle) (i2 : intervalle) =
  let a,b = i1 and c,d = i2 in a<=d && c<=b;;
```

□

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 $x$.

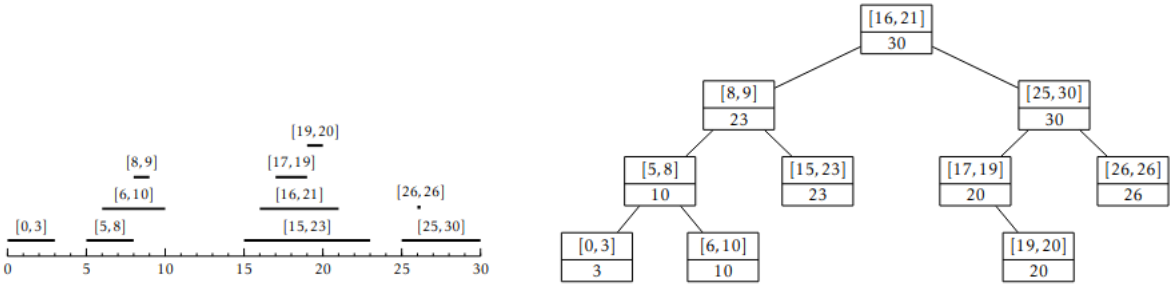



FIGURE 1 – Un ensemble de 10 intervalles, et l'arbre d'intervalles qui les représente (JP. BECIRSPAHIC).

Pour les représenter, on définit le type :

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

**Q2** Donner le code OCAML du sous-arbre gauche de l'arbre de la figure 1.

*Solution.* Voici

```
let ag = Noeud((8,9),23,Noeud((5,8),10,Noeud((0,3),3,Nil,Nil),
                                       Noeud((6,10),10,Nil,Nil)),
                    Noeud((15,23),23,Nil,Nil));;
```

□

**Q3** Rédiger une fonction `maxi` de type `int_tree -> 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.

*Solution.* Voici

```
let maxi a = match a with
  | Nil -> raise Not_found
  | Noeud(_,x,_,_)->x;;
```

□

**Q4** Rédiger une fonction `recherche` de type `intervalle -> int_tree -> bool` 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.

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

*Solution.* Recherche classique dans les ABR. On s'arrête dès qu'on a trouvé un chevauchement.

```
 let rec recherche ((x,y):intervalle) a = match a with
  | Nil -> raise Not_found
  | Noeud(i2,_,g,d) when chevauche i2 (x,y) -> i2
  | Noeud((a,b),_,g,d) -> if x<a then recherche (x,y) g
        else recherche (x,y) d;;
```

□

**Q5** Rédiger une fonction `cons` de type `intervalle -> int_tree -> int_tree -> int_tree` 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 $A_2$.

*Remarque.* Dans la suite, on utilisera la fonction `cons` lorsuqe le nouvel intervalle est plus grand que les étiquettes de $A_1$ et plus petit que celles de $A_2$.

*Solution.* Sans difficulté. Une fonction auxiliaire calcule le maximum entre la borne gauche de l'intervalle à insérer, la plus grande borne droite à gauche et à a droite.

```
let cons i a1 a2 =
  let maximum i a1 a2 = match a1,a2 with
    | Nil, Nil -> snd i
    | Nil, _ -> max (snd i) (maxi a2)
    | _, Nil -> max (snd i) (maxi a1)
    | _, _ -> max (max (snd i) (maxi a1)) (maxi a2)
  in  Noeud(i,maximum i a1 a2,a1,a2);;
```

□

**Q5** 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 $I$ est inséré à la racine.

Cette insertion à la racine impose parfois de faire basculer certains nœuds initialement d'un côté de la racine de $A$ vers l'autre côté.

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

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

Expliquer votre algorithme.

*Solution.* Selon la racine de l'arbre $A$, on doit faire passer des nœuds de la branche gauche vers la droite ou inversement.

Dans ce qui suit, on utilise la relation d'ordre de l'ABR : un intervalle est plus petit qu'un autre si et seulement si sa borne inférieure est inférieure à celle du second.

L'idée est de récupérer un tuple : On agglomère les nœuds plus petits que le nouvel intervalle pour en faire un arbre. De la même façon les nœuds plus grands forment un autre arbre.

Une fonction auxiliaire `separe` se déplace dans l'arbre binaire de recherche selon que l'intervalle à insérer est plus grand que l'étiquette du fils gauche ou non.

— Si l'intervalle est plus grand que l'étiquette du fils gauche de $A$, alors des nœud s du fils droit peuvent potentiellement se retrouver à gauche. On applique `separe` au fils droit pour obtenir deux arbres d'intervalles : le premier est constitué d'intervalle plus petits que l'intervalle à insérer, dans l'autre les intervalles sont plus grands.

Grâce à la fonction `cons` on ajoute l'arbre formé des nœuds plus petits au fils gauche.

— La procédure est symétrique si l'intervalle à insérer est plus petit que l'étiquette du fils gauche. Alors des nœuds du fils gauche sont susceptibles de passer à droite.

Appliquant la fonction `separe` à l'arbre $A$, on obtient deux arbres : le premier n'a que des étiquettes plus petites que l'intervalle à insérer, dans le second les étiquettes sont plus grandes. Il ne reste plus qu'à assembler l'intervalle et les deux arbres obtenus via la fonction `cons`.

□

**Q6** On veut supprimer un intervalle d'un arbre

**Q6a** Ecrire une fonction `supprime_min` de type `int_tree -> intervalle * int_tree` qui prend en paramètre un arbre et retourne un couple constitué de son intervalle le plus petit et de ce qu'est devenu l'arbre après suppression de cet intervalle.

```
let ad = Noeud((25,30),30,Noeud((17,19),20,Nil,
                                Noeud((19,20),20,Nil,Nil)),
               Noeud((26,26),26,Nil,Nil)) in
               supprime_min ad;;
      - : intervalle * int_tree =
((17, 19),
 Noeud ((25, 30), 30, Noeud ((19, 20), 20, Nil, Nil),
  Noeud ((26, 26), 26, Nil, Nil)))
```

**Q6b** Rédiger une fonction `supprime` de type `intervalle -> int_tree -> int_tree` qui prend en arguments un intervalle $I$ et un arbre d'intervalles $A$, et qui retourne une version de $A$ dans laquelle le sous-arbre $A'$ contenant l'intervalle $I$ aura été supprimé (s'il se trouve dans $A$).

Dans le cas où $A'$ est une feuille ou n'a qu'un fils, le procédé est évident. Dans le cas où $A'$ a deux fils, le sous-arbre mis à la place de $A'$ a pour racine le plus petit intervalle plus grand que $I$

dans $A'$.

La complexité attendue est $O(h(A))$.

*Solution.* On parcourt l'arbre à la recherche de l'intervalle $I$ à supprimer.

Si on le trouve dans un nœud à un seul fils, on remonte ce fils à la place de son père.

Si on le trouve dans un nœud $N$ à deux fils $G$, $D$, on applique `supprime_min` au fils droit. On forme un nouvel arbre dont la racine a pour étiquette le plus petit intervalle $K$ plus grand que $I$, le fils gauche est $G$ et le fils droit est ce qu'est devenu $D$ après suppression du nœud d'étiquette $K$.

```
let rec supprime i a =
  match a with
  | Nil -> Nil
  | Noeud(j,_,g,Nil) when i = j -> g
  | Noeud(j,_,Nil,d) when i = j -> d
  | Noeud(j,_,g,d) when i = j -> let k,d' = supprime_min d
    in cons k g d'
  |  Noeud(j,_,g,d) when fst i < fst j ->
    let g' = supprime i g in cons j g' d
  | Noeud(j,_,g,d) ->
    let d' = supprime i d in cons j g d';;
```

□