Arbres équilibrés et géométrie algorithmique

Ce sujet traite de l'implémentation d'ensembles finis par des arbres équilibrés dits AVL et de leur application à un problème de géométrie algorithmique classique, la localisation d'un point dans le plan.

Ce sujet est conçu pour être traité linéairement. Les parties I et II introduisent les arbres AVL et leur construction et la partie III les utilise pour la localisation d'un point dans le plan. Pour traiter une question, on peut admettre le résultat de toute question précédente.

Langage OCaml. Ce sujet utilise notamment les tableaux d'OCaml. On peut créer un tableau avec la fonction `Array.make`. L'appel de `Array.make n x` crée un tableau de taille `n` dont toutes les cases contiennent la valeur `x`. Les cases d'un tableau sont numérotées à partir de 0. La fonction `Array.length` renvoie la taille d'un tableau. Pour un tableau `tab`, on accède à l'élément d'indice `i` avec `tab.(i)` et on le modifie avec `tab.(i) <- v`.

Dans le code OCaml fourni dans le sujet, on utilise la construction **`assert`** `false` pour signaler un point de code inatteignable, c'est-à-dire un cas de figure qui n'est pas censé se produire si les préconditions de la fonction sont respectées.

Pour les questions de programmation, il n'est pas demandé de justifier la correction du code donné en réponse.

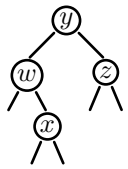
Complexité. Par *complexité en temps* d'un algorithme A on entend le nombre d'opérations élémentaires (comparaison, addition, soustraction, multiplication, division, affectation, test, etc.) nécessaires à l'exécution de A dans le cas le pire. Par *complexité en espace* d'un algorithme A on entend l'espace mémoire minimal nécessaire à l'exécution de A dans le cas le pire. Lorsque la complexité dépend d'un ou plusieurs paramètres $\kappa_0, \cdots, \kappa_{r-1}$, on dit que A a une complexité en $O(f(\kappa_0, \cdots, \kappa_{r-1}))$ s'il existe une constante $C > 0$ telle que, pour toutes les valeurs de $\kappa_0, \cdots, \kappa_{r-1}$ suffisamment grandes (c'est-à-dire plus grandes qu'un certain seuil), pour toute instance du problème de paramètres $\kappa_0, \cdots, \kappa_{r-1}$, la complexité est au plus $C \cdot f(\kappa_0, \cdots, \kappa_{r-1})$. De même, on dit que A a une complexité en $\Theta(f(\kappa_0, \cdots, \kappa_{r-1}))$ s'il existe deux constantes $C_1 > 0$ et $C_2 > 0$ telles que, pour toutes les valeurs de $\kappa_0, \cdots, \kappa_{r-1}$ suffisamment grandes, la complexité de A est au moins $C_1 \cdot f(\kappa_0, \cdots, \kappa_{r-1})$ et au plus $C_2 \cdot f(\kappa_0, \cdots, \kappa_{r-1})$.

Dans tout ce sujet, on note « log » le logarithme à base 2. On utilisera exclusivement ce logarithme.

Partie I. Préliminaires

On cherche à construire une structure de données pour représenter des sous-ensembles finis d'un ensemble U donné. L'ensemble U est supposé muni d'un ordre strict total noté $<$, c'est-à-dire que, pour tous $x, y \in U$, on a $x < y$ ou $x = y$ ou $y < x$. On va utiliser des arbres binaires de recherche équilibrés pour représenter des sous-ensembles de U.

Arbres binaires. Un *arbre binaire* est défini récursivement de la manière suivante : soit comme l'arbre vide, noté $\langle\rangle$, qui ne contient aucun nœud ; soit comme un *nœud*, noté $\langle \ell, x, r \rangle$ avec un sous-arbre gauche ℓ, un élément $x \in U$ à la racine et un sous-arbre droit r. On dessine un arbre tel que $\langle\langle\langle\rangle, w, \langle\langle\rangle, x, \langle\rangle\rangle\rangle, y, \langle\langle\rangle, z, \langle\rangle\rangle\rangle$ de la manière suivante



(*A*)

avec la racine (ici y) en haut, reliée aux sous-arbres gauche et droit dessinés en-dessous. Les sous-arbres vides ne sont pas dessinés, mais les liaisons vers les sous-arbres vides le sont.

On note $n(t)$ le nombre de nœuds et $h(t)$ la hauteur d'un arbre binaire t, définis par

$$n(\langle\rangle) = h(\langle\rangle) = 0, \quad n(\langle \ell, x, r\rangle) = 1 + n(\ell) + n(r) \quad \text{et} \quad h(\langle \ell, x, r\rangle) = 1 + \max(h(\ell), h(r))$$

On notera ici que, pour des raisons purement techniques, la hauteur de l'arbre vide est fixée à 0 contrairement au programme qui la fixe à -1. Ainsi, pour l'arbre A pris en exemple ci-dessus, on a $n(A) = 4$ et $h(A) = 3$.

Le *parcours infixe* d'un arbre binaire est une séquence d'éléments de U définie récursivement de la manière suivante :

— le parcours infixe de l'arbre vide $\langle\rangle$ est la séquence vide ;
— le parcours infixe d'un arbre $\langle \ell, x, r\rangle$ est la concaténation, dans cet ordre, du parcours infixe de l'arbre ℓ, de l'élément x et du parcours infixe de l'arbre r.

Ainsi, le parcours infixe de l'arbre précédent est la séquence w, x, y, z.

Un arbre binaire t est un *arbre binaire de recherche* si la séquence de ses éléments donnée par un parcours infixe est triée par ordre strictement croissant. En particulier, un élément n'apparaît qu'au plus une fois dans cette séquence. Tous les arbres binaires manipulés dans ce sujet seront toujours des arbres binaires de recherche.

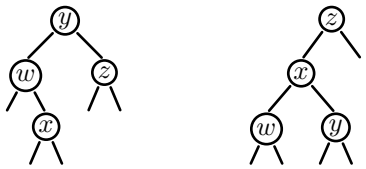
Question 1. Combien y a-t-il d'arbres binaires de recherche contenant exactement les trois éléments x, y et z, avec $x < y < z$? Les dessiner.

Question 2. Montrer qu'un arbre t est un arbre binaire de recherche si et seulement si t est l'arbre vide ou si t est un arbre $\langle \ell, x, r\rangle$ avec ℓ et r des arbres binaires de recherche où tous les éléments de ℓ sont strictement inférieurs à x et tous les éléments de r sont strictement supérieurs à x.

Arbres AVL. Pour équilibrer nos arbres binaires de recherche, on va imposer une condition supplémentaire : pour tout nœud $\langle \ell, x, r\rangle$, on a

$$|h(\ell) - h(r)| \leq 1 \tag{1}$$

De tels arbres sont dits AVL (du nom de leurs auteurs, Adelson–Velsky et Landis). Ainsi, dans les deux arbres ci-dessous,



celui de gauche est un AVL mais celui de droite n'en est pas un (en supposant $w < x < y < z$ par ailleurs).

Question 3. Combien y a-t-il d'arbres AVL contenant exactement les trois éléments x, y et z, avec $x < y < z$? Les dessiner. Même question avec les quatre éléments $w < x < y < z$. Justifier les réponses.

Question 4. Montrer que pour tout AVL t, on a $h(t) \leq 2\log(n(t) + 1)$. Indication : on pourra considérer N_h, le nombre minimal de nœuds d'un AVL de hauteur h, et minorer cette quantité en fonction de h.

Mise en œuvre en OCaml. On suppose que les éléments de U sont représentés par un type OCaml `elt`, qui n'est pas précisé pour l'instant. Pour deux éléments x et y donnés, on peut tester si $x = y$ avec `eq` x y et tester si $x < y$ avec `lt` x y.

```
val eq: elt -> elt -> bool
val lt: elt -> elt -> bool
```

Un AVL est représenté par une valeur du type OCaml suivant :

```
type tree = E | N of int * tree * elt * tree
```

Le constructeur `E` représente l'arbre binaire vide $\langle\rangle$. Une valeur `N`(z, ℓ, x, r) représente l'arbre binaire non vide $\langle \ell, x, r \rangle$, avec l'*invariant* que la valeur z du premier champ est toujours égale à la hauteur de l'arbre, *i.e.* $z = h(\langle \ell, x, r \rangle)$. On se donne une fonction `height` pour obtenir la hauteur d'un arbre :

```
let height t = match t with E -> 0 | N(h, _, _, _) -> h
```

On note que cette fonction s'évalue en temps constant. Pour maintenir l'invariant, on se donne une fonction `node` pour construire un nouveau nœud :

```
let node l x r = N(1 + max (height l) (height r), l, x, r)
```

On note que cette fonction s'évalue également en temps constant. Dans tout ce qui suit, on utilisera toujours la fonction `node` pour construire un nœud et jamais le constructeur `N` directement. On utilisera le constructeur `N` uniquement dans les motifs de filtrage.

Question 5. Écrire une fonction `mem: elt -> tree -> bool` qui prend en arguments un élément x et un arbre AVL t et qui détermine si x apparaît dans t. La complexité doit être en $O(\log n(t))$ et on demande de la justifier.

Partie II. Construire des arbres AVL

Maintenir la propriété d'arbre AVL pendant la construction des arbres binaires de recherche demande un certain effort. On propose ici de concentrer cet effort dans une unique fonction, `join`, dont le code est donné figure 1. La fonction `join` reçoit en paramètres deux arbres AVL `l` et `r` et un élément `x`. Elle suppose que les éléments de `l` (resp. `r`) sont strictement plus petits (resp. plus grands) que `x`. Elle renvoie un arbre AVL contenant l'union des éléments de `l`, de `r` et du singleton {`x`}.

Si les hauteurs de `l` et `r` satisfont déjà à la propriété d'AVL, alors il suffit d'appeler `node` (ligne 4). Sinon, on appelle `join_right` (resp. `join_left`) selon que le déséquilibre vient de `l` (ligne 2) ou de `r` (ligne 3). La fonction `join_right` procède récursivement, en descendant le long de la branche droite de l'arbre `l` (lignes 8–11) jusqu'à trouver un sous-arbre de hauteur

acceptable (lignes 4–7). Si besoin, des rotations sont effectuées localement avec les deux fonctions `rotate_left` et `rotate_right` (en haut de la figure 1). Ces deux fonctions implémentent la notion usuelle de rotation dans les arbres binaires de recherche, illustrée juste en-dessous de leur code. La fonction `join_left` est symétrique et son code est omis. Dans les raisonnements sur la fonction `join`, on pourra se contenter de considérer seulement le cas où `join_right` est appelée, en considérant l'autre cas symétrique.

On prendra le temps de bien lire et de bien comprendre le code de la fonction `join`. L'objectif des questions suivantes est de se persuader que `join` fonctionne correctement, ce qui est plus subtile qu'il n'y paraît.

Question 6. Montrer que tout appel à `join`, avec deux arguments `l` et `r` qui sont des AVL, termine et ne peut pas « planter », au sens où la ligne 3 de `rotate_left`, la ligne 3 de `rotate_right` et la ligne 12 de `join_right` ne sont jamais atteintes.

Question 7. Montrer que, lorsque la fonction `join_right` est appelée avec un AVL `l` de hauteur h et un AVL `r` de hauteur $\leq h-2$, alors elle renvoie

— soit un arbre de hauteur h ;
— soit un arbre de hauteur $h+1$ avec un sous-arbre gauche de hauteur $h-1$ et un sous-arbre droit de hauteur h.

Indication : pour chaque cas qu'on choisit de distinguer dans la preuve, on pourra se contenter d'un schéma donnant la forme de l'arbre renvoyé annoté avec les hauteurs pertinentes.

Question 8. Montrer que la fonction `join`, appliquée à deux AVL `l` et `r`, renvoie bien un arbre AVL dont la hauteur est au plus $1 + \max(h(\texttt{l}), h(\texttt{r}))$.

Question 9. Montrer que la fonction `join` a une complexité en $O(|h(\texttt{l}) - h(\texttt{r})|)$.

Insertion d'un élément. Pour insérer un nouvel élément dans un arbre AVL, on se donne la fonction `insert` (voir figure 2). Elle utilise une fonction auxiliaire, `split`, dont le code est également donné.

Question 10. Expliquer ce que fait la fonction `split` appliquée à un AVL `t` et un élément `y`.

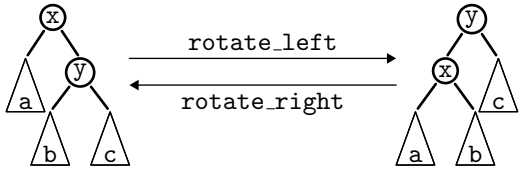
Question 11. Illustrer les étapes de l'insertion de l'élément c dans un AVL contenant les éléments a, b et d, avec $a < b < c < d$.

Question 12. Montrer que la fonction `split`, lorsqu'elle est appliquée à un AVL `t` et un élément `y`, renvoie deux arbres AVL dont la hauteur n'excède pas celle de `t`.

Question 13. Montrer que la fonction `insert`, lorsqu'elle est appliquée à un élément `x` et un arbre AVL `t`, renvoie bien un arbre AVL et a une complexité en $O(\log n(\texttt{t}))$.

```
let rotate_left t = match t with
  | N(_, a, x, N(_, b, y, c)) -> node (node a x b) y c
  | _ -> assert false

let rotate_right t = match t with
  | N(_, N(_, a, x, b), y, c) -> node a x (node b y c)
  | _ -> assert false
```




```
let rec join_right (l: tree) (x: elt) (r: tree) : tree =
  match l with
  | N(_, ll, lx, lr) ->
      if height lr <= height r + 1 then
        let t = node lr x r in
        if height t <= height ll + 1 then node ll lx t
        else rotate_left (node ll lx (rotate_right t))
      else
        let t = join_right lr x r in
        let t' = node ll lx t in
        if height t <= height ll + 1 then t' else rotate_left t'
  | E -> assert false

let rec join_left (l: tree) (x: elt) (r: tree) : tree =
  ...

let join (l: tree) (x: elt) (r: tree) : tree =
  if height l > height r + 1 then join_right l x r
  else if height r > height l + 1 then join_left l x r
  else node l x r
```

FIGURE 1 – Fonction `join` sur les AVL.

```
let rec split (t: tree) (y: elt) : tree * bool * tree =
  match t with
  | E -> E, false, E
  | N(_, l, x, r) ->
     if eq y x then l, true, r
     else if lt y x then let ll, b, lr = split l y in ll, b, join lr x r
     else let rl, b, rr = split r y in join l x rl, b, rr

let insert (x: elt) (t: tree) : tree =
  let l, _, r = split t x in
  join l x r
```

FIGURE 2 – Fonctions `split` et `insert` sur les AVL.

Question 14. Soit n un entier positif ou nul. On construit un tableau a de n arbres AVL de la manière suivante :

```
let a = Array.make n E
let () = for i = 1 to n - 1 do a.(i) <- insert (i-1) a.(i-1) done
```

Calculer le nombre total $f(\mathtt{n})$ d'entiers (possiblement répétés) stockés dans l'ensemble des arbres du tableau a. Montrer que la complexité en espace de la construction de a est en $O(g(\mathtt{n}))$ pour une fonction g telle que $g(n) = o(f(n))$. Expliquer pourquoi il n'y a pas de contradiction.

Supprimer un élément. On souhaite maintenant écrire une fonction `remove` pour supprimer un élément dans un arbre AVL, qui ait une complexité logarithmique comme la fonction `insert`.

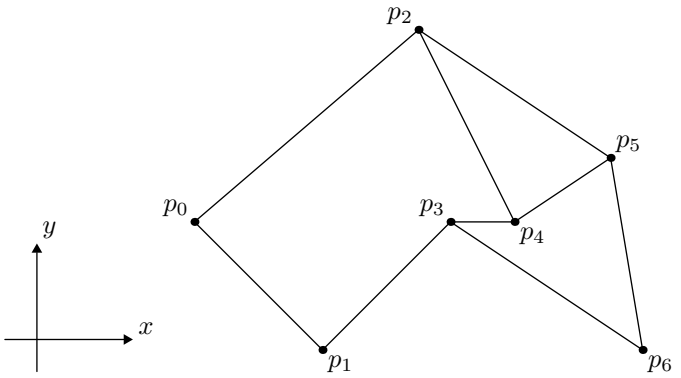
Question 15. Écrire une fonction `split_last: tree -> tree * elt` qui prend en argument un arbre AVL t, supposé non vide, et qui renvoie une paire (u, x) où x est le plus grand élément de t et u un arbre AVL contenant les éléments de t autres que x. La complexité doit être $O(h(t))$. On ne demande pas de la justifier.

Question 16. Écrire une fonction `join2: tree -> tree -> tree` qui prend en arguments deux arbres AVL ℓ et r, en supposant les éléments de ℓ strictement plus petits que ceux de r, et qui renvoie un arbre AVL t contenant l'union des éléments de ℓ et de r. La complexité doit être $O(h(\ell) + h(r))$. On ne demande pas de la justifier.

Question 17. Écrire une fonction `remove: elt -> tree -> tree` qui prend en arguments un élément x et un AVL t, et qui renvoie un arbre AVL contenant les éléments de t privés de x. (Si x n'apparaît pas dans t, le résultat contient les mêmes éléments que t.) La complexité doit être $O(\log n(t))$. On ne demande pas de la justifier.

Partie III. Application : localisation d'un point dans le plan

On considère une subdivision du plan définie par un ensemble fini de polygones. Ces polygones peuvent partager des sommets et des arêtes. Les arêtes ne se croisent pas. Voici un exemple avec 7 points et trois polygones :



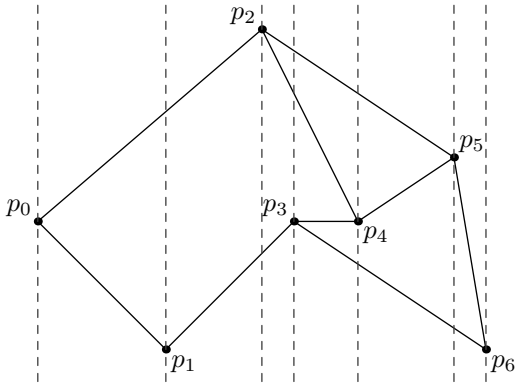
Le plan est ici divisé en quatre zones : l'intérieur de chaque polygone et la zone extérieure. Notre objectif est de construire une structure de données pour représenter cette subdivision, afin de répondre ensuite efficacement à la question « étant donné un point, dans quelle zone se trouve-t-il ? ». On s'intéresse notamment à la complexité de cette structure (temps et espace pour sa construction) et à la complexité de son utilisation ensuite. Ces complexités seront calculées en fonction du nombre total de sommets, noté N dans la suite. On admet que le nombre d'arêtes est en $O(N)$.

Notations. Pour un point p, on note $x(p)$ son abscisse et $y(p)$ son ordonnée. Les N sommets sont notés $p_0, p_1, \ldots, p_{N-1}$ et ordonnés de la gauche vers la droite. On suppose qu'il n'y a pas deux sommets sur la même abscisse, c'est-à-dire

$$x(p_0) < x(p_1) < \cdots < x(p_{N-1})$$

Une arrête e est un couple (p_i, p_j) avec $0 \leq i < j < N$. On note $start(e)$ son extrémité gauche, c'est-à-dire le point p_i, et $end(e)$ son extrémité droite, c'est-à-dire le point p_j. On note $xmin(e)$ l'abscisse de p_i et $xmax(e)$ l'abscisse de p_j. L'arrête e s'étend donc de l'abscisse $xmin(e)$ à l'abscisse $xmax(e)$.

Approche. Notre approche consiste à diviser le plan en $N + 1$ tranches verticales, délimitées par les abscisses des N points. Sur notre exemple, on a donc 8 tranches :



La tranche 0 correspond aux abscisses $]-\infty, x(p_0)]$, la tranche 1 aux abscisses $]x(p_0), x(p_1)]$, la tranche 2 aux abscisses $]x(p_1), x(p_2)]$, etc., et la tranche N aux abscisses $]x(p_{N-1}), +\infty[$. Chaque tranche est traversée par un certain nombre d'arêtes. On note que, dans chaque tranche, les segments d'arêtes qui traversent cette tranche peuvent être ordonnés verticalement (car, on le rappelle, les arêtes ne se croisent pas). Ainsi, dans la tranche 3 ci-dessus, on a, de bas en haut, l'arête (p_1, p_3), l'arête (p_2, p_4) puis enfin l'arête (p_2, p_5).

L'idée est alors de trier les tranches selon les abscisses, puis, dans chaque tranche, de trier les arêtes la traversant selon les ordonnées. Ainsi, étant un point du plan à localiser, il suffira d'une recherche dichotomique pour déterminer la tranche contenant le point puis d'une seconde recherche dichotomique pour déterminer entre quelles arêtes se trouve le point (et, par conséquent, dans quel polygone se trouve le point, le cas échéant).

Question 18. On note T_i le nombre d'arêtes qui traversent la tranche i. (Dans l'exemple ci-dessus, on a $T_0 = 0$, $T_1 = T_2 = 2$, $T_3 = 3$, etc.) Montrer que la quantité $\sum T_i$ peut être en $\Theta(N^2)$.

Mise en œuvre en OCaml. On utilise des nombres flottants pour les coordonnées des points. On introduit les deux types suivants pour les points et les arêtes, respectivement.

```
type point = float * float
type edge = point * point (* avec x1 < x2 *)
```

On se donne deux fonctions pour récupérer l'abscisse gauche et l'abscisse droite d'une arête :

```
let xmin ((x, _), _) = x
let xmax (_, (x, _)) = x
```

Pour représenter chaque tranche, on va utiliser un arbre AVL dont les éléments sont des arêtes, c'est-à-dire le type `tree` introduit plus haut dans le sujet avec

```
type elt = edge
```

Une tranche est alors représentée par le type suivant,

```
type slab = { xleft: float; xright: float; tree: tree }
```

où `xleft` est l'abscisse du bord gauche de la tranche, `xright` l'abscisse du bord droit et `tree` l'arbre AVL contenant les arêtes qui traversent cette tranche, ordonnées de bas en haut.

Question 19. Écrire une fonction `lt: edge -> edge -> bool` qui prend en arguments deux arêtes e_1 et e_2, supposées traversant une même tranche, et qui détermine si e_1 est strictement en dessous de e_2.

Construction des tranches. On représente l'ensemble des tranches par un tableau `slabs: slab array` de taille $N + 1$. Il est trié selon les abscisses, c'est-à-dire

$$\begin{aligned} -\infty = \texttt{slabs.(0).xleft} &< \texttt{slabs.(0).xright} = x(p_0) = \texttt{slabs.(1).xleft} \\ &< \texttt{slabs.(1).xright} = x(p_1) = \texttt{slabs.(2).xleft} \\ &< \dots \\ &< \texttt{slabs.(}N\texttt{).xright} = +\infty \end{aligned}$$

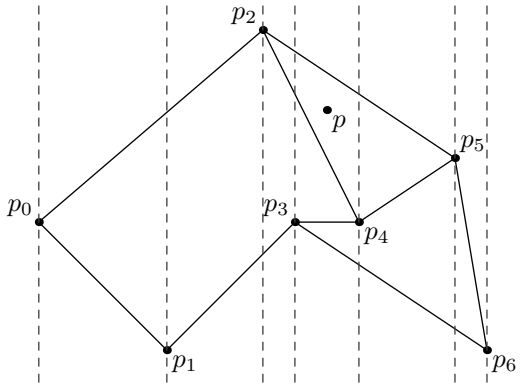
(Le type `float` inclut des valeurs `neg_infinity` pour $-\infty$ et `infinity` pour $+\infty$.) On construit les arbres AVL de chaque tranche de la gauche vers la droite, avec l'algorithme suivant :

— L'arbre de la tranche 0 est vide.
— Pour construire l'arbre de la tranche i, pour $1 \leq i \leq N$,
  — on retire les arêtes qui se terminent au point p_{i-1} ;
  — on ajoute les arêtes qui démarrent au point p_{i-1}.

En particulier, l'arbre de la dernière tranche est vide.

Question 20. Justifier que l'on peut construire, à partir de la liste des arêtes, le tableau `slabs` en temps et en espace $O(N \log N)$. On ne demande pas d'écrire le code, mais on demande d'être précis quant aux structures de données et algorithmes utilisés et on demande d'être convaincant quant à la complexité. On ne demande pas de vérifier que tous les sommets ont des abscisses différentes.

Localisation d'un point. Enfin, on peut utiliser le tableau `slabs` pour localiser un point p donné. On fait l'hypothèse que ce point n'est sur aucune arête, et est donc en particulier différent des N points p_i. Si on prend l'exemple suivant



alors le point p est d'abord localisé dans la tranche 4 puis, dans cette tranche, localisé entre les arêtes (p_2, p_4) et (p_2, p_5).

Question 21. Écrire une fonction `find_slab: slab array -> float -> slab` qui prend en arguments les tranches et l'abscisse du point p recherché et renvoie la tranche dans laquelle se trouve le point p. La complexité doit être $O(\log N)$. On ne demande pas de la justifier.

Question 22. Écrire une fonction `find: slab array -> point -> answer` qui prend en arguments les tranches et un point p et renvoie la zone dans laquelle le point p se trouve, sous la forme d'une valeur du type

```
type answer = Outside | Between of edge * edge
```

où `Outside` signifie que le point p est à l'extérieur de tout polygone et `Between`(e_1, e_2) signifie que le point p se trouve à l'intérieur d'un polygone, entre les arêtes e_1 et e_2. La complexité doit être $O(\log N)$ et on demande de la justifier.

* *
*