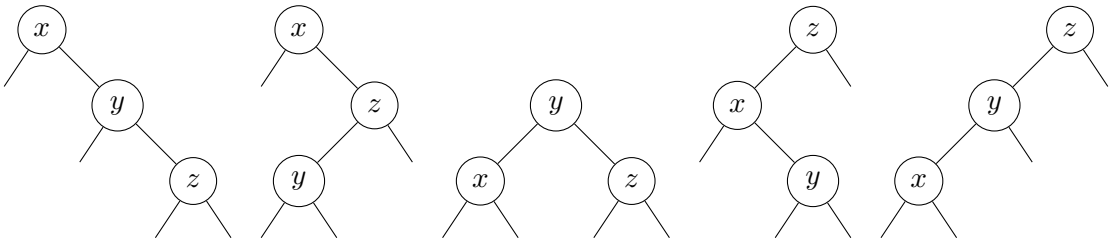
Un corrigé : X-ENS INFO A 2024
Arbres équilibrés et géométrie algorithmique

Pour toutes les remarques ou corrections, vous pouvez m'envoyer un mail à
galatee.hemery@gmail.com

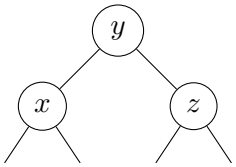
Partie I. Préliminaires

Question 1. Il en existe 5 :

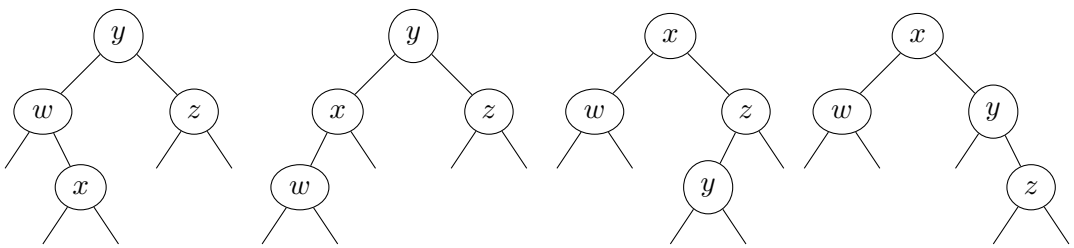


Question 2. L'arbre vide $\langle\rangle$ est bien un arbre binaire de recherche. Ainsi l'équivalence est vrai pour ce cas particulier.
Soit $t = \langle \ell, x, r \rangle$ un arbre binaire.
$\Rightarrow$: Si t est un arbre binaire de recherche, alors le parcours infixe donne une séquence triée par ordre strictement croissant par définition.
Or il s'agit de la concaténation du parcours infixe de ℓ, puis de x, puis du parcours infixe de r donc nécessairement tous les éléments de ℓ sont strictement inférieurs à x et tous les éléments de r strictement supérieur à x.
De plus, ℓ et r sont des arbres binaires de recherches car la séquence pour t est triée : un préfixe de cette séquence est la séquence pour l'arbre ℓ et un suffixe de cette séquence est la séquence pour l'arbre r. Elles sont bien triées.
Ainsi, ℓ, r vérifient bien les hypothèses de l'équivalence.
$\Leftarrow$: Réciproquement, si ℓ, r sont des arbres binaires de recherche et si tous les éléments de ℓ sont strictement inférieurs à x et tous les éléments de r strictement supérieur à x, alors la séquence obtenue par concaténation est bien triée et t est bien un arbre binaire de recherche.

Question 3. Il en existe 1 à trois éléments, les autres arbres binaires de recherche vus dans la question 1 ne vérifient pas la condition (1) :



Il en existe 4 à 4 éléments. Il est nécessaire d'avoir 1 et 2 éléments dans les deux sous-arbres fils principaux.



Question 4. Comme l'invite l'énoncé, on s'intéresse à N_h le nombre minimal de noeuds d'un AVL de hauteur h.
On note que c'est une suite croissante car on peut enlever des noeuds pour diminuer la hauteur.
On a $N_0 = 0$ (l'arbre vide est de hauteur nulle) et $N_1 = 1$ (arbre réduit à un noeud).
Soit $t = \langle \ell, x, r \rangle$ un arbre de hauteur h à N_h noeuds, alors sans perte de généralité ℓ est de hauteur $h-1$ et a N_{h-1} noeuds et r est de hauteur $h-2$ et a N_{h-2} noeuds (la hauteur ne peut être inférieure sinon la condition (1) n'est pas vérifiée).
On obtient $N_h = N_{h-1} + N_{h-2} + 1$.
En particulier on a

$$N_h \geq 2N_{h-2} + 1 \geq 2(2N_{h-4} + 1) + 1 \geq \cdots \geq 2^{\lfloor h/2 \rfloor} N_\alpha + 2^{\lfloor h/2 \rfloor - 1} + \cdots + 1 \geq 2^{h/2} - 1$$

avec $\alpha = 0$ si h pair et on minore par $2^{h/2} - 1$ exactement et $\alpha = 1$ sinon et on minore par $2^{\lfloor h/2 \rfloor + 1} - 1 \geq 2^{h/2} - 1$.
Ainsi $n(t) \geq N_{h(t)} \geq 2^{h(t)/2} - 1$, puis $n(t) + 1 \geq 2^{h(t)/2}$ puis $\log(n(t) + 1) \geq h(t)/2$. Enfin, on a bien montré que $h(t) \leq 2\log(n(t) + 1)$.

Question 5. On écrit une fonction récursive pour laquelle chaque appel récursif considère un arbre de hauteur au moins un de moins. Ainsi, on a au plus $h(t)$ appels récursif pour un appel sur un arbre t. Le reste du code : filtrage et appels à `eq` et `lt` est en temps constant.
D'après la question 4, $h(t) \leq 2\log(n(t) + 1)$, ainsi, on a bien une complexité en $O(\log(n(t)))$.

```
1  let rec mem x t = match t with
2      | E -> false
3      | N(h,l,e,r) when eq e x -> true
4      | N(h,l,e,r) when lt x e -> mem x l
5      | N(h,l,e,r) -> mem x r
```

Partie II. Construire des arbres AVL

Question 6. Trois cas se présentent. Si les deux arbres vérifient la condition (1) sur les hauteurs, on termine avec un appel à `node`, sinon on fait appels à une des deux fonctions récursives selon le cas.
Par symétrie, on ne traite que `join_right` qui est appelée lorsque `height l > height r + 1`.

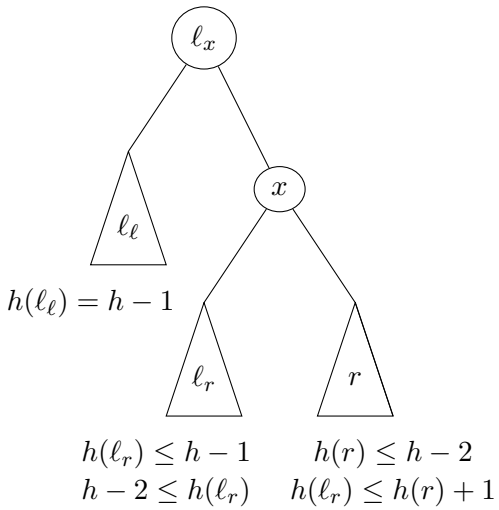
Pour montrer la terminaison, on utilise un variant pour la fonctions récursive `join_right`.
En particulier, lorsque `height l > height r + 1`, `l` ne peut pas être `E` et on ne lit pas la ligne 12 de `join_right` pour le premier appel.
On a un appel récursif lorsque `height lr > height r +1` et en particulier `l` ne peut pas être `E` et on ne lit pas la ligne 12 que `join_right` pour les appels récursifs.
De plus la hauteur de l'arbre `lr` est plus petite strictement que la hauteur de `l` : c'est notre **variant**. Ainsi, on a nécessairement `height lr <= height r +1` après un certain nombre d'appels car `r` reste le même.

Justifions maintenant que les fonctions de rotations n'échouent pas.
Dans la ligne 7 de `join_right`, on appelle `rotate_right` sur un arbre `t` qui est construit à l'aide de `node lr x r` donc il n'est pas vide et on a :
`height lr <= height r+1` et `height t > height ll + 1`
donc `height t  = height lr + 1 > height ll + 1` car la hauteur correspond à `ll` ou `lr`, donc `lr` n'est pas vide : donc la ligne 3 n'est pas lue.
Ce qui est produit est nécessairement non vide donc la ligne 3 de `rotate_left` n'est pas lue non plus.

Dans la ligne 11, on appelle `rotate_left` sur `t'` construit à partir de `node ll lx t` non vide et `height t > height ll + 1` donc `t` est non vide et la ligne 3 de `rotate_left` est non lue.
Au total, aucune des lignes avec `assert false` n'est lue lors d'un appel à `join` et on a bien la terminaison grâce au variant sur la hauteur.

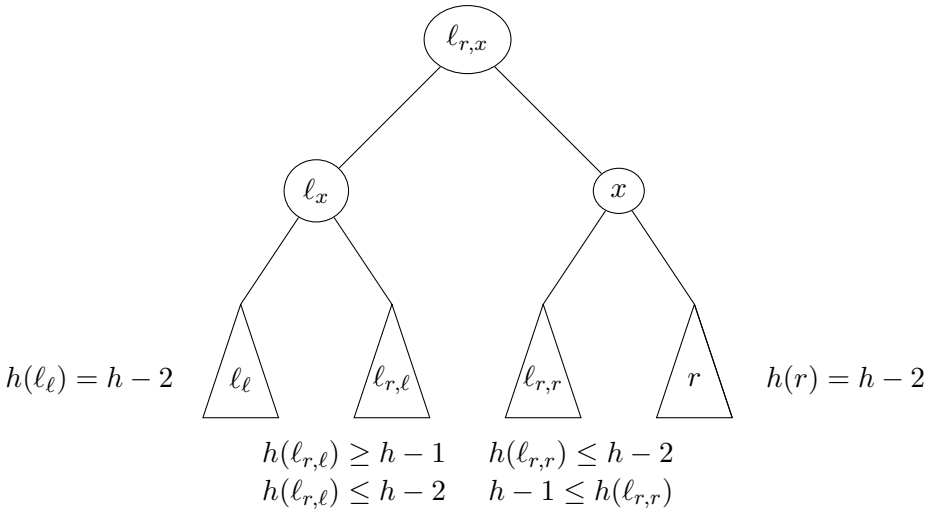
Question 7. (*on traite la partie AVL de la question suivante dans cette question).*
Soit ℓ, r les AVL d'encodage `l,r` de hauteur h et au plus $h-2$.
$\ell = \langle \ell_\ell, \ell_x, \ell_r \rangle$ encodé par `ll,lx,lr`.
On a $h(\ell_\ell)$ ou $h(\ell_r)$ qui vaut $h-1$ et l'autre $h-1$ ou $h-2$ par la propriété (1) sur les hauteurs dans un AVL.
On procède par récurrence sur $h(\ell_r)$ pour montrer que le résultats est un AVL de hauteur h ou $h+1$ avec un fils gauche de hauteur $h-1$ contenant les éléments de ℓ, r et x exactement.
Initialisation :
Si $h(\ell_r) \leq h(r) + 1 \leq h-1$, alors $h-1 \leq h(t) \leq h$ avec $t = \langle \ell_r, x, r \rangle$ et :

— si $h(\ell_\ell) = h-1$ on renvoie $\langle \ell_\ell, \ell_x, t \rangle$ de hauteur au plus $h+1$ et le sous arbre gauche ℓ_ℓ est bien de hauteur $h-1$.



L'ordre des éléments est bien respecté car ℓ_ℓ, ℓ_r, r AVL et les élément de ℓ_ℓ sont plus petits strictement que ℓ_x, ceux que ℓ_r que x et ceux que r plus grand strictement que que x et ℓ_x et $\ell_x < x$.
Les noeuds vérifie (1) dans les trois sous-arbres et $|h(\ell_r) - h(r)| \leq 1$ car $h(r) \leq h(\ell_r) \leq h(r)+1$ et $|h(\ell_\ell) - h(t)| \leq 1$ car $h(t) = h(\ell_r)$ et ℓ AVL.

— sinon, $h(\ell_\ell) = h-2$ et $h(\ell_r) = h-1$, puis $h(t) = h > h-2+1$ et on renvoie le produit des rotations.
On pose $\ell_r = \langle \ell_{r,\ell}, \ell_{r,x}, \ell_{r,r} \rangle$.
La rotation droite sur $t = \langle \langle \ell_{r,\ell}, \ell_{r,x}, \ell_{r,r} \rangle, x, r \rangle$ donne $\langle \ell_{r,\ell}, \ell_{r,x}, \langle \ell_{r,r}, x, r \rangle \rangle$.
La rotation gauche sur $\langle \ell_\ell, \ell_x, \langle \ell_{r,\ell}, \ell_{r,x}, \langle \ell_{r,r}, x, r \rangle \rangle \rangle$ donne $\langle \langle \ell_\ell, \ell_x, \ell_{r,\ell}, \rangle \ell_{r,x}, \langle \ell_{r,r}, x, r \rangle \rangle$.
Les quatre sous arbres sont de hauteurs au plus $h-2$, avec au moins une égale à $h-2$, donc l'arbre est de hauteur h.



L'ordre est respecté et on peut vérifier sur les nouveaux noeuds la condition (1).

Hérédité :
Si $h-1 \geq h(\ell_r) > h(r)+1$.

- Si $h(\ell_r) = h-2$, alors $h(r) \leq h-4$ et on peut appliquer l'hypothèse de récurrence à ℓ_r de hauteur $h-2$ et r.
  Ainsi le résultat t de l'appel récursif est un AVL de hauteur $h-1$ ou $h-2$ contenant les éléments de ℓ_r, r et x.
  Donc $t' = \langle \ell_\ell, \ell_x, t\rangle$ de hauteur h car ℓ_ℓ de hauteur $h-1$ nécessairement.
  On renvoie t' car $h(\ell_\ell)+1 = h \geq h(t) \geq h-1$. L'ordre est bien respecté et le nouveau noeud vérifie (1) donc c'est bien un AVL.
- Si $h(\ell_r) = h-1$, alors $h(r) \leq h-3$ et on peut appliquer l'hypothèse de récurrence à ℓ_r de hauteur $h-1$ et r.
  Ainsi le résultat t de l'appel récursif est un AVL de hauteur h ou $h-1$ contenant les éléments de ℓ_r, r et x.
  Donc $t' = \langle \ell_\ell, \ell_x, t\rangle$ de hauteur au plus $h+1$ et ℓ_ℓ de hauteur au plus $h-1$.
  - Si $h(t) = h-1$, alors $h(t) \leq (h-2)+1 \leq h(\ell_\ell)+1$ donc on renvoie t' qui est alors de hauteur h et le nouveau noeud vérifie (1), l'ordre est respecté, c'est bien un AVL.
  - Si $h(t) = h$, alors t a un fils gauche de hauteur $h-2$.
    - Si $h(\ell_\ell) = h-1$, alors on renvoie t' de hauteur $h+1$ dont le fils gauche ℓ_ℓ est bien de hauteur $h-1$.
      L'ordre est bien respecté et le nouveau noeud vérifie bien (1). C'est bien un AVL.
    - Si $h(\ell_\ell) = h-2$, alors $h(t) > h(\ell_\ell)+1$ (t' n'est pas un AVL en l'état), et on renvoie le résultat de la rotation de $t' = \langle \ell_\ell, \ell_x, \langle t_\ell, t_x, t_r\rangle\rangle$, qui donne $\langle\langle \ell_\ell, \ell_x, t_\ell\rangle, t_x, t_r\rangle$ de hauteur h car $h(\ell_\ell) = h(t_\ell) = h-2$ et $h(t_r) = h-1$.
      La rotation permet de conserver le bon ordre et les noeuds après rotation vérifie (1).

La disjonction de cas permet de conclure dans tous les cas.

Question 8. D'après ce que l'on fait dans la question précédente, `join_right` renvoie bien un AVL de hauteur au plus $h(\ell)+1$.
Par symétrie, on a `join_left` renvoie bien un AVL de hauteur au plus $h(r)+1$ lorsque $h(\ell) \leq h(r)-2$.
Sinon on renvoie $\langle \ell, x, r\rangle$ de hauteur $1+\max(h(\ell), h(r))$ qui est bien un AVL car les noeuds de ℓ, r vérifient (1) et les deux arbres ont une différences de hauteur d'au plus 1 donc l'arbre

d'étiquette x vérifie (1).
Au total, `join` renvoie bien un AVL de hauteur au plus $1 + \max(h(\ell), h(r))$ car l'énoncé précise que les éléments de ℓ sont strictement plus petit que x et les éléments de r plus grand que x dans l'introduction de la partie..

Question 9. Les opérations de rotation, de construction et les comparaisons de hauteurs sont en temps constant pour chaque appel dans `join,join_right,join_left`.
Pour `join_right`, chaque appel récursif se fait sur avec `ll` de hauteur au moins 1 de moins que `r` jusqu'à obtenir une hauteur plus petite que $h(\texttt{r}) + 1$. Ainsi, on a au plus $h(\texttt{l}) - h(\texttt{r})$ appels.
Par symétrie, on a pour `join_left` au plus $h(\texttt{r}) - h(\texttt{l})$ appels.
On en déduit la complexité en $O(|h(\texttt{r}) - h(\texttt{l})|)$ pour `join`.

Question 10. La fonction sépare un AVL `t` en deux AVL selon un élément `y` : elle renvoie `a1,b,a2` où `a1` est un AVL contenant tous les éléments strictement plus petits que `y` dans `t`, `a_2` un AVL contenant tous les éléments strictement plus grands que `y` dans `t` et `b` est un booléen qui vaut `true` si `x` est dans `t` et `false` sinon.
Pour cela la fonction procède récursivement en explorant le sous-arbre susceptible de contenir `y` jusqu'à obtenir un arbre vide ou bien `y` et la fonction fusionne les AVL des résultats des appels récursifs pour conserver tous les éléments de `t`.

Question 11. Un AVL contennant a, b, d est de la forme `N( N(E,a,E),b,N(E,d,E))`.
On applique `split`, on appelle récursivement sur `N(E,d,E)` puis sur E qui donne `E,false,E`.
L'appel sur `N(E,d,E)` renvoie `E, false, join E d E` soit `E, false, N(E,d,E)`.
L'appel sur `N( N(E,a,E),b,N(E,d,E))` renvoie `join N(E,a,E) b E, false, N(E,d,E)` soit `N( N(E,a,E),b,E), false, N(E,d,E)`.
Pour l'insertion, on fait appel à `join N(N(E,a,E),b,E) N(E,d,E)` et la différence de hauteur est 1 donc on a directement `N(N(N(E,a,E),b,E),c,N(E,d,E))`

Question 12. Montrons le par induction.
Cas de base :
Si `t = E`, on renvoie deux AVL `E` de même hauteur.
Cas inductif :
Si `t = N(_,l,x,r)`, il y a trois cas possibles :

- si `x = y`, on renvoie `l,r` de hauteur strictement plus petite que `t` en tant que sous-arbre ;
- si `y < x`, on réalise un appel récursif `split l y` qui renvoie par hypothèse d'induction deux AVL `ll,lr` de hauteur au plus celle de `l` et $h(\texttt{l}) \leq h(\texttt{t}) - 1$. Puis on renvoie `ll` de hauteur au plus $h(\texttt{t})-1$ et `join lr x r` de hauteur au plus $1+\max(h(\texttt{t})-1, h(\texttt{r})) \leq h(\texttt{t})$;
- si `x < y`, le cas est symétrique au précédent.

Question 13. La fonction `insert x t` renvoie un AVL contenant les éléments de `t` et `x` car `split` renvoie deux AVL contenant les éléments de `t` strictement supérieurs et inférieurs à `x` |et `join` renvoie un AVL à partir de deux AVL et d'un éléments contenant exactement tous les éléments.
La complexité de `insert` correspond à la somme des complexités de `split t x` et `join l x r`. `join l x r` se fait en $O(|h(\texttt{l}) - h(\texttt{r})|)$, soit en $O(h(\texttt{t}))$ soit en $O(\log(n(t)))$ car `l` et `r` sont de hauteur au plus $h(\texttt{t})$ par la question 12.
Il reste à discuter de la complexité de `split t x`.
Notons $T(\texttt{t}, \texttt{y})$ la complexité de `split t y` :
On a $T(\texttt{t}, \texttt{y}) = O(1)$ lorsque `y = x`, $T(\texttt{t}, \texttt{y}) = O(1) + T(\texttt{l}, \texttt{y}) + O(|h(\texttt{lr}) - h(\texttt{r})|)$ lorsque $y < x$ et $T(\texttt{t}, \texttt{y}) = O(1) + T(\texttt{r}, \texttt{y}) + O(|h(\texttt{l}) - h(\texttt{rl})|)$ lorsque $y > x$.
Notons C_1, C_2 les constantes correspondant aux grands O et traitons le second cas (le troisième est symétrique) en choisissant $C_1 \leq C_2$.
$T(\texttt{t}, \texttt{y}) \leq T(\texttt{l}, \texttt{y}) + C_1 + C_2|h(\texttt{lr}) - h(\texttt{r})|$.

On pose `tl, _, tr = split t y`.
Montrons par récurrence sur la hauteur de `t` que $T(\texttt{t},\texttt{y}) \leq C_1 + (C_1+C_2)\cdot h(\texttt{t}) + C_2\cdot(h(\texttt{tl}) + h(\texttt{tr}))$.
Si `t` est vide, alors le résultat est obtenu en temps constant, donc le résultat est vrai. Il suffit que C_1 majore le nombre d'opérations élémentaires effectuées.
Comme `t` est un AVL, on a $h(\texttt{l})$ et $h(\texttt{r})$ qui valent $h(\texttt{t})-1$ ou $h(\texttt{t})-2$:

— si $h(\texttt{r}) \geq h(\texttt{lr})$, alors $h(\texttt{t}) \geq h(\texttt{r})+1 \geq h(\texttt{tr}) \geq h(\texttt{r})$ d'après la question 7 et
$T(\texttt{t},\texttt{y}) \leq T(\texttt{l},\texttt{y}) + C_1 + C_2\cdot h(\texttt{r}) - C_2\cdot h(\texttt{lr}) \leq T(\texttt{l},\texttt{y}) + C_1 + C_2\cdot h(\texttt{tr}) - C_2\cdot h(\texttt{lr})$
Par hypothèse de récurrence, comme `ll,b,lr = split l y` :
$T(\texttt{t},\texttt{y}) \leq C_1 + (C_1+C_2)\cdot h(\texttt{l}) + C_2\cdot h(\texttt{ll}) + C_2\cdot h(\texttt{lr}) + C_1 + C_2\cdot h(\texttt{tr}) - C_2\cdot h(\texttt{lr})$
$T(\texttt{t},\texttt{y}) \leq 2C_1 - C_2 - C_1 + (C_2+C_1)\cdot h(\texttt{t}) + C_2\cdot h(\texttt{ll}) + C_2\cdot h(\texttt{tr})$
Or `ll = tl` donc $T(\texttt{t},\texttt{y}) \leq C_1 + (C_1+C_2)\cdot h(\texttt{t}) + C_2\cdot h(\texttt{tl}) + C_2\cdot h(\texttt{tr})$

— sinon, $h(\texttt{r}) < h(\texttt{lr})$. On a de plus $h(\texttt{lr}) \leq h(\texttt{l}) \leq h(\texttt{r})+1$ par la question 12 et la propriété (1) sur les AVL.
Nécessairement, on a $h(\texttt{lr}) = h(\texttt{l}) = h(\texttt{r})+1 = h(\texttt{t})-1$
$T(\texttt{t},\texttt{y}) \leq T(\texttt{l},\texttt{y}) + C_1 + C_2\cdot h(\texttt{lr}) - C_2\cdot h(\texttt{r})$
On applique l'hypothèse de récurrence :
$T(\texttt{t},\texttt{y}) \leq C_1 + (C_1+C_2)\cdot h(\texttt{l}) + C_2\cdot h(\texttt{ll}) + C_2\cdot h(\texttt{lr}) + C_1 + C_2\cdot h(\texttt{lr}) - C_2\cdot h(\texttt{r})$
$T(\texttt{t},\texttt{y}) \leq 2C_1 - C_2 - C_1 + (C_1+C_2)\cdot h(\texttt{t}) + C_2\cdot h(\texttt{ll}) + C_2\cdot h(\texttt{l}) + C_2\cdot h(\texttt{l}) - C_2\cdot h(\texttt{l}) + C_2$
$T(\texttt{t},\texttt{y}) \leq C_1 + (C_1+C_2)\cdot h(\texttt{t}) + C_2\cdot h(\texttt{ll}) + C_2\cdot h(\texttt{l})$
Or `tl = ll` et $h(\texttt{tr}) = h(\texttt{lr})$ ou $h(\texttt{lr})+1$ soit $h(\texttt{l})$ ou $h(\texttt{l})+1$.
Enfin $T(\texttt{t},\texttt{y}) \leq C_1 + (C_1+C_2)\cdot h(\texttt{t}) + C_2\cdot h(\texttt{tl}) + C_2\cdot h(\texttt{tr})$

Le cas symétrique est traité de la même façon.
Enfin, $T(\texttt{t},\texttt{y}) \leq C_1 + (C_1+C_2)\cdot h(\texttt{t}) + C_2\cdot h(\texttt{tl}) + C_2\cdot h(\texttt{tr})$ permet d'avoir $T(\texttt{t},\texttt{y}) \leq C_1 + C_3\cdot h(\texttt{t})$ avec $C_3 = C_1 + 3C_2$ car les arbres `tl,tr` sont de plus petite hauteur que `t`.
Enfin, $h(\texttt{t}) = \Theta(\log(n(t)))$ donc on obtient bien une complexité pour `split` en $O(\log(n(t)))$.
Au total, en ajoutant les complexité `split` et `join` dans `insert`, on obtient une complexité en $O(\log(n(t)))$ pour `insert`.

Question 14. Dans la construction l'AVL `a.(i)` contient les entiers de `0` à `i-1` car on les insère de proche en proche.
Ainsi, `a.(i)` contient `i` entiers.
Au total, on a $f(\texttt{n}) = \sum_{i=0}^{n-1} i = \frac{(\texttt{n}-1)\texttt{n}}{2}$.
En mémoire, la fonction `insert` réalise la construction en $O(\log(\texttt{i}))$ d'après la question 13 pour l'arbre `a.(i)` donc on a la création d'au plus $O(\log(\texttt{i}))$ noeuds supplémentaires, les autres noeuds sont déjà stockés dans la mémoire, et OCaml exploite des pointeurs en mémoire vers les mêmes noeuds déjà stockés dans la mémoire (il n'y a pas de copie).
Ainsi, la complexité temporelle de construction majore la complexité en espace qui est donc en $O(\sum_{i=1}^{n-1}\log(i))$ qui est majoré par $O(n\log(n))$.
On a $g(n) = n\log(n) = o(n^2) = o(f(n))$.

Question 15.

```
1 let rec split_last t = match t wih
2     | E -> assert false
3     | N(_, l, x, E) -> l,x
4     | N(_, l, x, r) -> let ru,rx = split_last r in join l x ru, rx
```

Question 16.

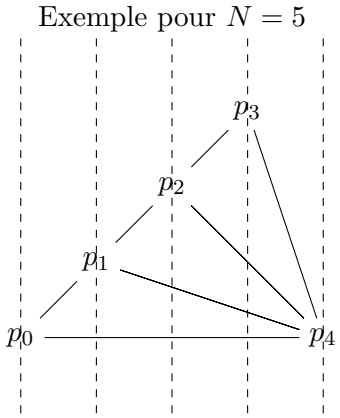
```
let rec join2 l r =
    let u,x = split_last l in
    join u x r
```

Question 17.

```
let rec remove x t =
    let l,_,r = split x t in
    join2 l r
```

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

Question 18. On suppose que l'on a $p_i = (i, i)$ pour $0 \leq i \leq N-2$ et $p_{N-1} = (N-1, 0$ et les arêtes (p_i, p_{N-1}) et (p_i, p_{i+1}) pour $0 \leq i \leq N-2$.



La tranche i correspond aux abscisses $]i-1, i]$ pour $1 \leq i \leq N-2$ et contient les arêtes (p_j, p_{N-1}) pour $0 \leq j < i$ et l'arête (p_{i-1}, p_i) donc $T_i = i + 1$.
La tranche $N-1$ contient les arêtes (p_j, p_{N-1}) pour $0 \leq j \leq N-2$ donc $T_{N-1} = N-1$.
Les tranches 0 et N ne contiennent pas d'arêtes.
Dans ce cas, $\sum T_i$ vaut $N - 1 + \sum_{i=1}^{N-2}(i+1) = \Theta(N^2)$.

Question 19. Les arêtes ne se croisent pas et on souhaite les comparer sur une tranche donc il suffit de vérifier en pour une abscisse (milieu de la bande commune) que l'ordonnée de e_1 est plus petite que l'ordonnée de e_2.

```
let max a b = if a>b then a else b
let min a b = if a<b then a else b

let lt e1 e2 =
    let ((x1,y1),(x1',y1')) = e1 in
    let ((x2,y2),(x2',y2')) = e2 in
    let x0 = ((max x1 x2)+.(min x1' x2'))/. 2. in
    let y01 = x1 +. ((y1'-.y1)/.(x1'-.x1)) *. (x0-.x1) in
    let y02 = x2 +. ((y2'-.y2)/.(x2'-.x2)) *. (x0-.x2) in
    y01 < y02
```

Question 20. On rappelle que le nombre d'arêtes est en $O(N)$.
Étape 1 : $O(N \log(N))$ temps et espace

A partir de la liste des arêtes, on récupère toutes les abscisses dans une liste par un parcours. On trie cette liste (par un tri classique efficace par exemple partition-fusion) et on supprime les doublons par un parcours de la liste triée.
Étape 2 : $O(N)$ temps et espace
On initialise un tableau de taille $N+1$ où N est le nombre d'abscisses avec les `xleft,xright` correspondants aux abscisses et des arbres vides `E`.
Étape 3 : $O(N)$ temps et espace
Création de deux tableaux `depart` et `arrivee` de taille N tel que `depart.(i)` contient une liste des arêtes démarrant du point ayant la $(i-1)$-ème abscisse la plus grande et `arrivee.(i)` la même chose avec arrivant au lieu de démarrant.
Un parcours de la liste suffit.
Étape 4 : $O(N\log(N))$ temps et espace
On applique l'algorithme décrit.
Pour $i = 1, ..., N-1$, on modifie `slabs.(i).tree` en utilisant `remove` ($O(\log(N))$) pour enlever toutes les arrêtes de `arrivee.(i-1)`, puis on ajoute les arrêtes de `depart.(i-1)` à l'aide de `insert` ($O(\log(N))$).
Chaque arête est ajoutée et retirée une unique fois donc la complexité temporelle de cette étape est en $O(N\log(N))$ et également en espace car la complexité temporelle est plus importante.
Au total, on a bien la complexité attendue.

Question 21.

```
let find_slab slabs x =
    let rec dichotomie i j =
        if i = j then slabs.(i) else begin
            let m = (i+j)/2 in
            if slabs.(m).xleft > x then dichotomie i (m-1)
            else if slabs.(m).xright < x then dichotomie (m+1) j
            else slabs.(m)
        end
    in dichotomie 0 (Array.length slabs -1)
```

Question 22.

```
let rec cherche p t =
    (* cherche l'arête au dessus et en dessous de p dans un AVL t *)
    match t with
        | E -> (p,p),(p,p)
               (* le segment p,p sert pour l'absence de segment trouvé *)
        | N(_,l,x,r) when lt (p,p) x -> (* p est sous x *)
            let e1,e2 = cherche l in
            if e2 = p,p then e1,x else e1,e2
        | N(_,l,x,r) -> (* p est au dessus de x *)
            let e1,e2 = cherche r in
            if e1 = p,p then x,e2 else e1,e2
let find slabs p =
    let x,y = p in
    let sl = find_slabs x in
    let e1,e2 = cherche p sl.tree in
    if e1 = p,p || e2 = p,p then Outside
    else Between (e1,e2)
```