Chapitre 3

Ensembles totalement ordonnés : arbres binaires de recherche

Le but de ce chapitre est d'étudier une implémentation au service du type Abstrait Ensemble mais en disposant d'un ordre total sur les éléments. Avec un ordre total, les éléments de l'ensemble peuvent être classés et le désir de dichotomie est présent. La recherche dans un tableau trié est en $O(\log n)$ grâce à la dichotomie mais il peine via l'ajout et la suppression. La liste, même triée, quant à elle peine sous tous les aspects. Oublions donc la liste ou presque...

Dans ce chapitre, nous allons évoquer une structure de données qui combine les avantages des tableaux triés et des listes tout en laissant les inconvénients au dortoir. L'avantage du tableau triés est clair et on va le garder : le tri. L'avantage des listes passent par les pointeurs : facilité d'ajouter un élément et de supprimer un élément s'il est au début. Pour cela, il faut voir en 2D ! On va aborder un pilier des structures de données : les **arbres**. Avec eux :

— On va pouvoir utiliser la dichotomie
— Bénéficier de l'aspect dynamique des listes
— Avoir des complexités en $O(\log n)$ dans le pire des cas.

Autres avantages :

— des opérations spécifiques comme *predecesseur* et *successeur*, etc.
— persistance

3.1 Idée : tri rapide

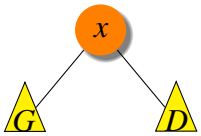
Le tri rapide est un algorithme de type 'diviser pour régner' qui permet de trier un tableau. Il fonctionne de la manière suivante :

— Diviser : On choisit un élément e du tableau (qu'on appelle pivot). On parcourt le tableau pour constituer deux tableaux : le premier, T_G qui contient tous les éléments plus petits ou égal à e et un autre, T_D qui contient les éléments qui sont strictement plus grands que e.
— Régner : on trie les deux tableaux T_G et T_D.
— Combiner : on concatène le résultat du tri de T_G à $[e]$ et au résultat du tri de T_D.

Et pourquoi ne pas représenter un ensemble trié de cette façon mais avec une structure en deux dimensions, à savoir, un arbre.

3.2 Arbres binaires

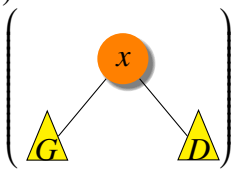
C'est une structure inductive comme la liste, mais en deux dimensions cette fois ci.

Définition 9. *Un arbre binaire est : l'arbre vide (∅) ou* 

avec G et D deux arbres binaires.

Définition 10. *La hauteur d'un arbre est définie inductivement par*

— $h(\varnothing) = -1$

— *et* $h\left(\right.$ 

$\left.\right) = 1 + \max\{h(G), h(D)\}$.

On appelle racine d'un arbre son premier nœud et feuille un nœud tel que $g = d = \varnothing$.

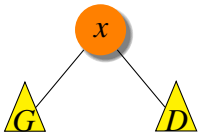
Théorème 14. *Dans un arbre binaire, on a* $1 + h \leqslant n \leqslant 2^{h+1} - 1$*, avec h la hauteur et n le nombre de nœuds.*

De même, $f \leqslant 2^h$ *avec f le nombre de feuilles.*

3.3 Arbres binaires de recherche

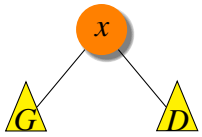
On considère que l'ensemble des éléments admet un ordre total. Un arbre binaire de recherche est un arbre binaire où les éléments sont triés de gauche à droite. On a vraiment l'image du tri rapide en tête : la racine est le pivot e et les sous-arbres correspondent respectivement aux éléments plus petits ou égaux à e et aux éléments plus grands strictement que e.

3.3.1 Définition

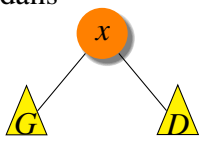
Définition 11. *Un* ***arbre binaire de recherche*** *est un arbre binaire tel que pour tout nœud* 

de cet arbre, pour tout $n \in G$, $n \leqslant x$ *et pour tout* $n \in D$, $n \geqslant x$.

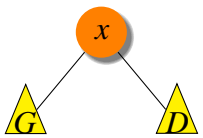
3.3.2 Opérations

Pour chercher i dans 

on cherche i dans G si $i < x$ et dans D si $i > x$.

Pour ajouter i à 

, on cherche s'il y est. Dans ce cas, on ne fait rien. Sinon, on l'ajoute à G si $i < x$ et à D sinon.

Pour supprimer i à 

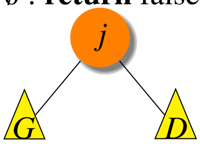
, on le supprime dans G si $i < x$, dans D si $i > x$ et si $i = x$, on renvoie $N(G', y, D)$ avec $y = \max G$ et $G' = G \setminus \{y\}$.

Ces trois opérations sont en $O(h)$ avec h la hauteur de l'arbre.

— Entrée : T un ABR, e un élément

— Sortie : true iff T contient e

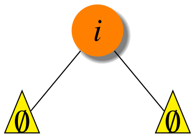
function contient(T, e)
    match T
        $\emptyset$: **return** false

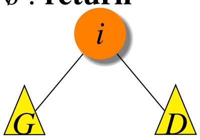


        :
            if $j = e$
                return true
            else if $e < j$
                return contient(G, e)
            else
                return contient(D, e)

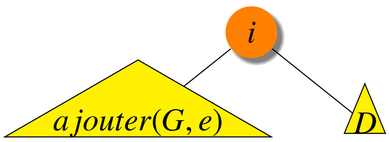
— Entrée : T un ABR, e un élément
— Sortie : un ABR qui contient les éléments que T et e

function ajouter(T, e)
    match T
        $\emptyset$: **return**

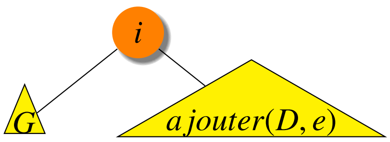




        :
        if $i = e$
            return T
        else if $e < i$
            return

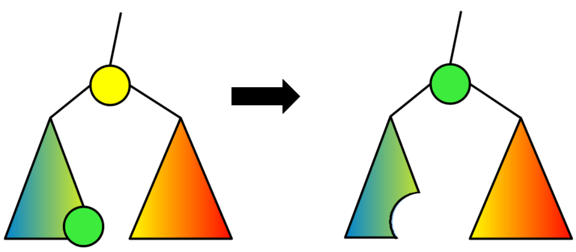


        else
            return



        endIf
    endMatch
endFunction

La suppression est moins triviale. On recherche l'élément e à supprimer dans T. Une fois trouvé, nous avons un arbre de racine e avec deux sous-arbres. Si un des sous-arbres est vide... c'est pratique ! On remplace l'arbre courant par ce sous-arbre. Sinon, on remplace la racine par le maximum du sous-arbre de gauche tout en supprimant l'élément maximal du sous-arbre gauche comme le montre l'image ci-dessous :



Pour trouver le maximum d'un arbre, on file tout droit à droite ! !

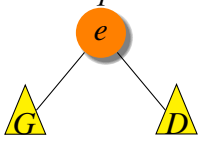
Dans le pire des cas, toutes ces opérations sont réalisées en $O(h)$ où h est la hauteur de l'arbre. Dans le pire des cas, l'arbre peut être linéaire et la hauteur est alors égale à n, le nombre

d'élément dans l'ensemble. Dans la section suivante, nous allons introduire des opérations de rééquilibrage pour éviter ce cas défavorable.

3.4 Arbres binaires de recherche équilibrés

Dans cette section, nous proposons une technique pour garder des arbres équilibrés. Cette technique est cruciale : elle permet d'avoir des hauteurs en $O(\log n)$ et donc des implémentations d'insertion, recherche et suppression efficace. On introduit l'équilibrage proposé par Adelson-Velsky-Landis. Définissons cette équilibrage.

Définition 12. *On définit inductivement les* ***arbres binaires de recherche équilibrés*** *par :*

- $\varnothing$ *est équilibré*
- 

est équilibré ssi G et D le sont et $|h(G) - h(D)| \leqslant 1$.

On appellera AVL (pour les initiales) tout arbre binaire de recherche équilibré de cette manière. Cet équilibrage fournit une borne logarithme pour la hauteur. On a donc une excellente complexité sur la classe des AVL.

Proposition 1. *Si A est un arbre binaire de recherche équilibré,* $\log_2(n+1) \leqslant h+1 \leqslant 1.44\log_2(n)$ *où n est le nombre de noeuds et h est la hauteur de l'arbre.*

Démonstration.

Première inégalité Dans tout arbre binaire on a $n \leqslant 2^{h+1} - 1$ (d'ailleurs, l'égalité est atteinte pour les arbres binaires de hauteur h qui contiennent le maximum de nœud). Ainsi on a $\log_2(n+1) \leqslant h+1$.

Deuxième inégalité À présent, montrons la seconde inégalité. Montrer $h+1 \leqslant 1.44\log_2(n)$ revient à étudier les arbres équilibrés qui contiennent le moins de nœuds.

Notations

Soit $\mathcal{M}_h$ l'ensemble des arbres binaires de recherche équilibrés de hauteur h avec le nombre minimum de nœuds. Soit $n(h)$ le nombre de nœuds d'un tel arbre. Si on montre que $h+1 \leqslant 1.44\log_2(n(h))$, la démonstration sera finie.

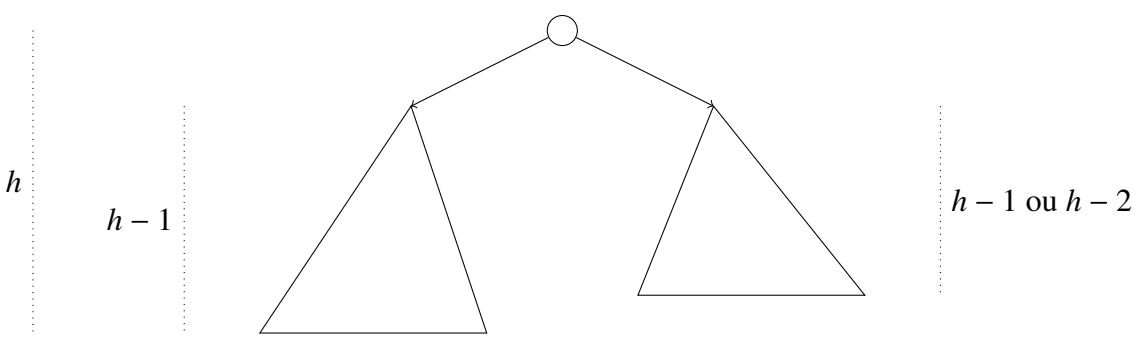
Etape 1 : trouver une relation de récurrence des $n(h)$

Voici les cas de base :

— $n(-1) = 0$ (il n'y a qu'un seul arbre de hauteur -1 : il s'agit de l'arbre vide) ;

— $n(0) = 1$ (car les arbres binaire de recherche minimaux (!) de hauteur 0 sont les arbres-racines).

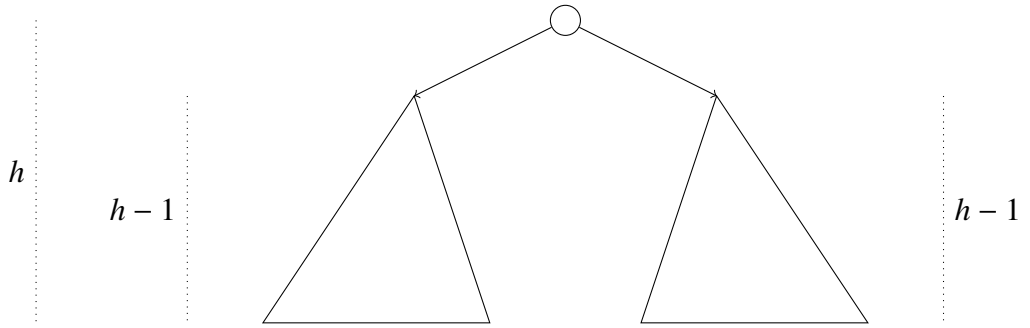
Soit $h \geq 1$ un entier. Un arbre A de $\mathcal{M}_h$ est formé d'une racine à laquelle on attache deux sous-arbres. L'un des sous-arbres (disons que c'est celui de gauche) est de hauteur $h-1$. L'autre (de droite donc) est de hauteur $\leq h-1$. Le sous-arbre de droite est de hauteur $h-2$ ou $h-1$ car A est équilibré.



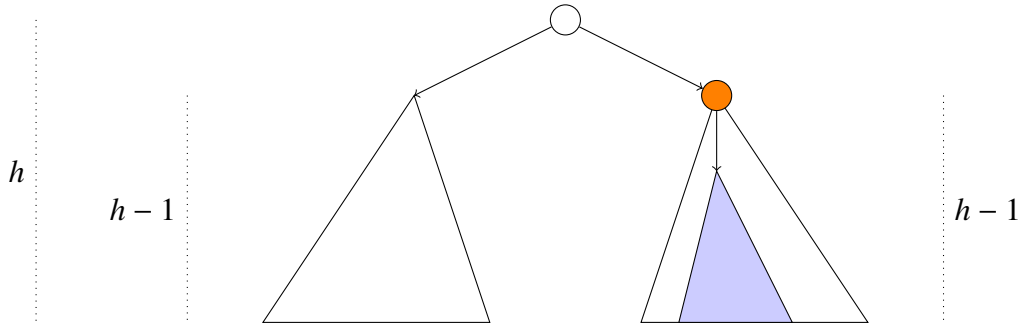
Fait : le sous-arbre droit de A est forcément de hauteur $h-2$.

Démonstration.

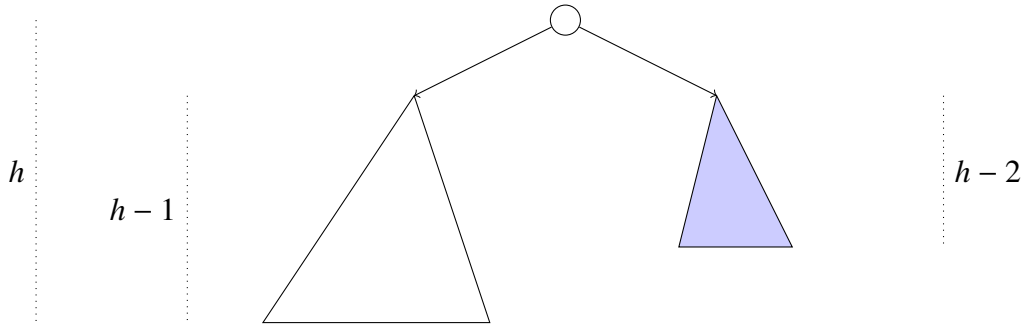
Démontrons le fait. Par l'absurde, supposons que le sous-arbre droit D de A est de hauteur $h-1$.



L'un des sous-arbres de D que l'on appelle D' est de hauteur $h-2$ (peint en bleu sur le dessin suivant).



Remplaçons alors D par D' dans A.



On obtient un arbre qui est toujours équilibré et qui a strictement moins de noeud que A. En effet, D' a strictement moins de noeuds que D (car on a au moins enlevé la racine de D (en orange)). Ce qui contredit la minimalité de A. Donc le sous-arbre droit de A est forcément de hauteur $h-2$. ■

Les sous-arbres directs de A sont respectivement des arbres de $\mathcal{M}_{h-1}$ et de $\mathcal{M}_{h-2}$ pour garantir la minimalité de A en nombre de nœuds.

Ainsi :

$$\text{si } h \geq 1, \text{ on a } n(h) = n(h-1) + n(h-2) + 1.$$

C'est une variante de la suite de Fibonacci.

Etape 2 : trouver une expression explicite de $n(h)$

En l'étudiant, on peut conclure la démonstration et obtenir l'inégalité souhaitée. TODO :

■

Malheureusement, les opérations définies dans la section précédentes détruisent l'équilibrage. A terme, l'arbre initialement équilibré pourrait ressembler à un arbre linéaire. Tout ceci est bien dommage.

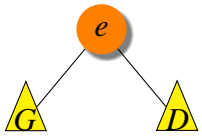
Heureusement, nous allons introduire des opérations de rééquilibrage (des rotations) et on les applique dès que la différence de hauteur de deux sous-arbres dépasse 2 en valeur absolu.

Remarque. Calculer la hauteur, même si son calcul est logarithmique, est couteux. En effet, nous devons avoir les hauteurs à chaque étape. On ne va pas calculer la hauteur à chaque fois, on la stocke dans A.h et on la met à jour au fur et à mesure. Il s'agit de la même idée que lorsque nous avons étendu la structure de données Ensemble avec un champ E.cardinal pour éviter de recalculer le cardinal de l'ensemble à chaque fois.

La fonction d'ajout ne change pas, mais on doit rééquilibrer après avoir ajouté un élément.

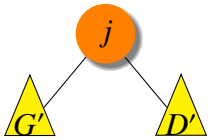
On dira qu'un ABR T a ses hauteurs à jour ssi dans tous les sous-arbres A de T, $A.h$ soit égale à la hauteur de A.

Algorithme 8 : rééquilibrage

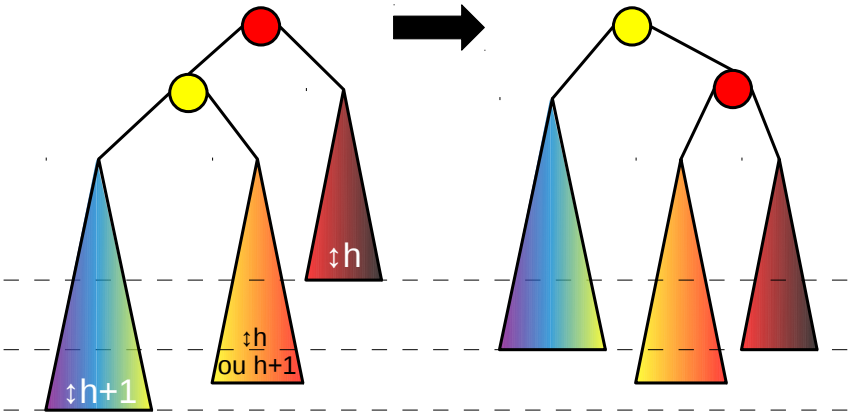
Entrées : Un ABR $A =$ 

, qui a ses hauteurs à jour et tel que G et D sont des AVL et $|G.h - D.h| \leq 2$

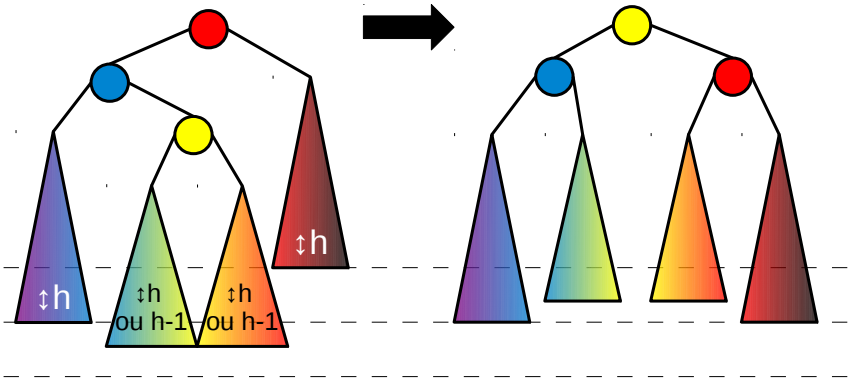
Sorties : Un ABR équilibré A' qui contient les même éléments que A et qui a ses hauteurs à jour

```
1  si |G.h − D.h| ⩽ 1 alors
2      retourner A
3  sinon
4      si G.h = 2 + D.h alors
           
           
5          G s'écrit
6          si G'.h ⩾ D'.h alors
7              retourner rotD(A)
8          sinon
9              retourner rotGD(A)
10     sinon
11         on fait de même avec rotG et rotDG
```

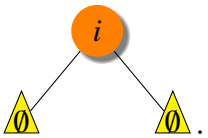
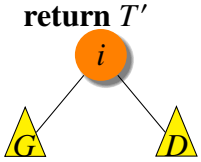
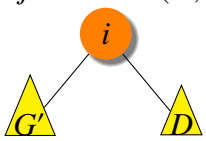
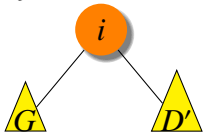
avec la rotation droite définie par :



et la rotation gauche droite définie par :



— Entrée : T un AVL qui a ses hauteurs à jour, e un élément
— Sortie : un AVL T', qui a ses hauteurs à jour et qui contient exactement e et tous les éléments de T et tel que $T.h \leq T'.h \leq T.h + 1$

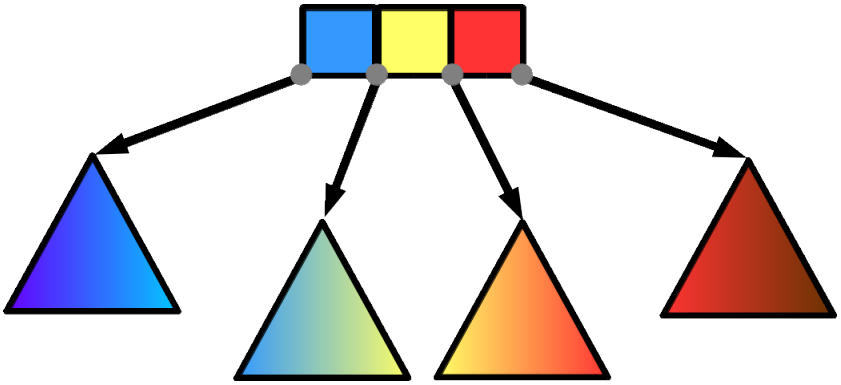
```
function ajouterAVL(T, e)
    match T
        case ∅ :
            T′ := 
            T′.h = 0 ;
            return T′
        case  :
            if i = e
                return T
            else if e < i
                G′ = ajouterAVL(G, e)
                T' = 
                T'.h = 1 + max(G'.h, D.h) ;
                return reequilibrer(T′)
            else
                D′ = ajouterAVL(D, e)
                T' = 
                T'.h = 1 + max(G.h, D'.h) ;
                return reequilibrer(T′)
            endIf
    endMatch
endFunction
```

Remarque.
Il y a au plus un rééquilibrage effectué. En effet, à chaque rééquilibrage, h est constante.

L'algorithme de suppression est le même, sauf qu'on rééquilibre le résultat. Il peut y avoir plusieurs rééquilibrages.

3.5 B-arbres

Les arbres binaires de recherche ne sont pas efficaces lorsque l'arbre est stocké sur un disque dur. En effet, il faut faire des E/S à chaque nœud. Une solution consiste à faire qu'un nœud contiennent plusieurs valeurs. Un B-arbre a la forme suivante :



3.5.1 Équilibrage via des contraintes

Le problème est maintenant l'équilibrage. La solution est d'imposer les contraintes suivantes aux arbres :

- Tout nœud a au plus $2B - 1$ clefs ;
- Tout nœud non racine a au moins $B - 1$ clefs ;
- Si l'arbre est non vide, la racine est non vide ;
- Les feuilles sont à la même profondeur h.

où $B \geq 2$ est un entier fixé.

Un nœud est dit *complet* lorsqu'il a exactement $2B - 1$ clefs.

Theorem 2. *Un B-arbre de hauteur h qui contient n clefs est tel que*

$$h \leq \log_B \left(\frac{n+1}{2} \right).$$

Démonstration.

à la profondeur ...	on a au moins ... nœuds	et au moins ... clefs
0	1	1
1	2	$2(B-1)$
2	$2B$	$2B(B-1)$
3	$2B^2$	$2B^2(B-1)$
$\vdots$	$\vdots$	$\vdots$
h	$2B^{h-1}$	

D'où

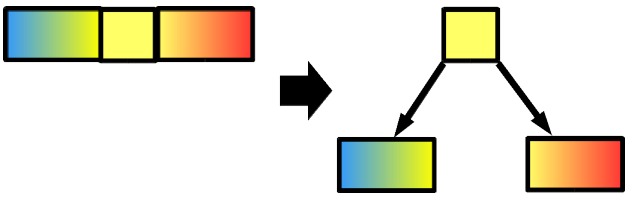
$$n \geq 1 + 2(B-1) \sum_{i=0}^{h-1} B^i = 1 + 2(B-1)\frac{B^h - 1}{B-1} = 1 + 2(B^h - 1) = 2B^h - 1.$$

Ce qui prouve le théorème. ■

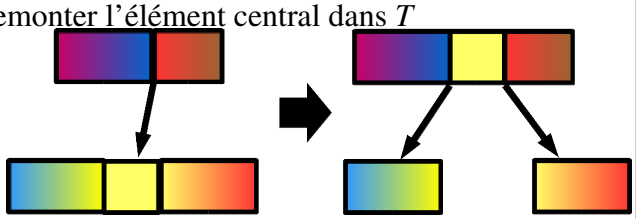
3.5.2 Insertion

La procédure commence à la racine puis on descend dans l'arbre. Si le noeud courant est complet, on l'éclate. Par exemple, au début, on éclate la racine si elle est complète. Si on noeud internet est complet, on l'éclate en utilisant le fait que son père n'est pas complet (car sinon, on l'aurait éclaté). Puis l'insertion se fait dans une feuille (non complète, sinon on l'aurait éclaté).

```
function insererDepuisRacine(T, k)
    if T complet
        S := eclaterRacine(T)
        insererDepuisNoeudIncomplet(S, k)
    else
        insererDepuisNoeudIncomplet(T, k)
    endIf
endFunction
```

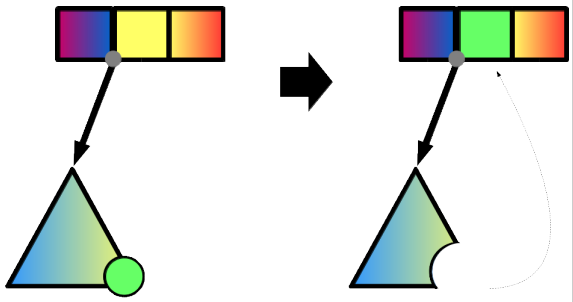


```
function insererDepuisNoeudIncomplet(T, k)
    // pre : T n'est pas complet
    if T feuille
        on insère la clef k dans T
    else
        S := le sous-arbre de T qui correspond au bon intervalle pour k
        if S complet
            On éclate S en faisant remonter l'élément central dans T
            S := le sous-arbre de T qui correspond au bon intervalle pour k
        endIf
        insererDepuisNoeudIncomplet(S, k)
    endIf
endFunction
```

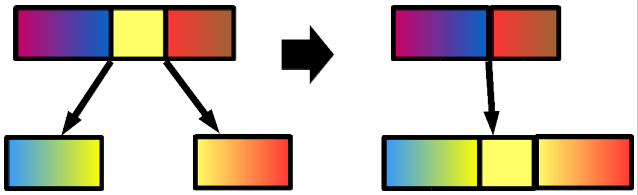


3.5.3 Suppression

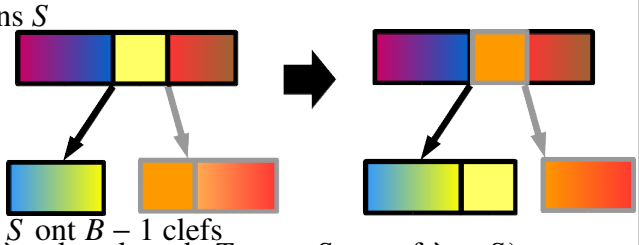
function suppression(T, k)
    //pre : T contient plus de B clefs ou alors T est la racine
    if T contient la clef k
        if T est une feuille
            supprimer k de T
        else if un des fils directs de k contient plus de B clefs
            Soit S ce fils
            suppression du min/max dans S
            on remplace k par ce min/max



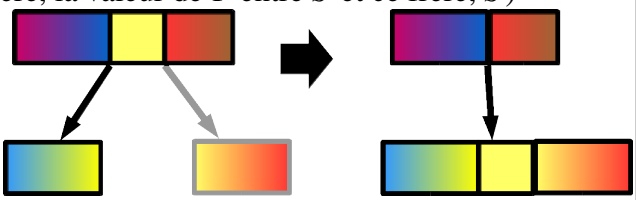
        else // les deux fils directs de k ont $B-1$ clefs
            On supprime k de T
            On crée le fils avec les clefs des deux fils directs et k que l'on connecte à T



    else //T ne contient pas la clef k
        Soit S le sous-fils de T qui correspond à l'intervalle qui contient k
        if S a $B-1$ clefs
            if un des frères de S a plus de B clefs
                On remonte un des éléments de ce frère dans T et on fait descendre une clef de T dans S



            else if tous les frères de S ont $B-1$ clefs
                S := fusion(ce frère, la valeur de T entre S et ce frère, S)



                Si T devient une racine vide, on supprime cette racine
        suppression(S, k)
endFunction

Pour supprimer le minimum dans S, on appelle une fonction suppressionMin(..) (ou suppressionMax(..)) construite sur le schéma de suppression(...) :

— tout le cas 'T contient la clef k' est remplacé par 'si T est une feuille, alors supprimer l'élément le plus à gauche' ;
— Sinon, on considère S, le sous-fils de T le plus à gauche.

Alors que dans l'insertion l'invariant au cours des appels était :

le noeud courant est incomplet.

Ici, lors des appels pour la suppression d'une clef k, l'invariant est :

le noeud courant est la racine ou alors a $\geq B$ clefs.

3.6 Arbres rouges et noirs

En fait, on peut représenter les B-arbres avec $B = 4$, aussi appelés arbres 2-3-4 avec des arbres binaires de recherche, appelé arbres rouges noirs. On simule les noeuds avec plusieurs clefs par plusieurs noeuds identifiés par des couleurs. La couleur noire représente une clef 'élue' comme représentante d'un noeud d'un B-arbre alors que la couleur rouge représente les autres clefs. Plus précisément :

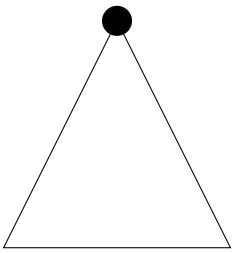
— Un noeud x d'arité 2 est représenté par un noeud noir x ;
— Un noeud $[x, y]$ d'arité 3 est représenté par un noeud noir x avec un fils rouge y (ou alors un noeud y noir avec un fils rouge x).
— Un noeud $[x, y, z]$ d'arité 4 est représenté par un noeud noir y avec deux fils rouges x et z.

Les arbres rouges et noirs sont une autre technique que les AVL pour obtenir une hauteur logarithme en le nombre de nœuds. Voici la comparaison (à prendre avec des précautions...) avec les arbres AVL :

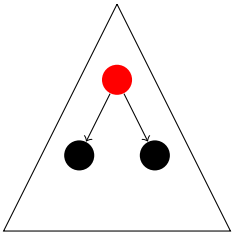
	Arbre AVL	Arbre rouge et noir
insertion	$O(\log n)$ mais assez lent	rapide
suppression	$O(\log n)$ mais assez lent	rapide
recherche	$O(\log n)$ rapide	mais assez lent

Pour cela, on introduit des couleurs : chaque nœud est peint en noir ou en rouge. Les arbres doivent en plus satisfaire les propriétés suivantes :

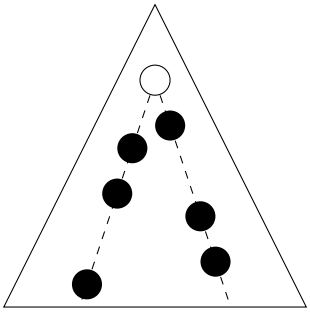
1. La racine est noire ;



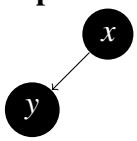
2. Si un nœud est rouge, alors ses fils sont noirs ;



3. Pour tout nœud x, toutes les branches partant de x vers un arbre vide $\emptyset$ contiennent le même nombre de nœuds noirs.



Exemple 19. *L'arbre suivant n'est pas un arbre rouge et noir :*



En effet, de x partent trois branches ! L'une passe par y et arrive dans l'arbre vide ∅ fils gauche de y. Une autre passe par y et arrive dans l'arbre vide ∅ fils droit de y. Ces deux branches contiennent 2 noeuds noirs (x et y). Et une dernière arrive dans l'arbre vide ∅ fils droit de x. Cela là ne contient qu'un seul noeud noir : x. Donc cet arbre viole la propriété 3.

Dans les algorithmes exposés ci-dessous, si on fait référence à l'arbre vide ∅, on le considère comme noir.

Theorem 3. *La hauteur d'un arbre rouge et noir est au plus* $2\log_2(n+1)$ *où n est le nombre de nœuds.*

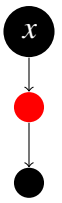
Démonstration.
Soit $hn(x)$ la hauteur noire d'un nœud. C'est le nombre de noeuds noirs entre x (exclus) et une feuille (incluse). Montrons par récurrence forte sur la hauteur h d'un arbre que :

$$\text{un nœud } x \text{ de hauteur } h \text{ contient au moins } 2^{hn(x)} - 1 \text{ nœuds.}$$

Si la hauteur est -1, l'arbre est vide et il y a 0 noeuds. Par ailleurs, la hauteur noire vaut 0 et $2^0 - 1 = 0$.

Prenons désormais un nœud x de hauteur $h \geq 0$. Chaque fil a une hauteur au plus $h-1$. La hauteur noire d'un des fils est $hn(x)$ ou $hn(x)-1$ selon que x est noir ou rouge. Par hypothèse de récurrence, il y a au moins $2^{hn(x)-1} - 1$ nœuds dans chaque fils. Le nœud x a deux fils (au pire, l'un des fils est l'arbre vide que l'on a traité, c'est le cas de base). L'arbre de racine x a donc au moins $2^{hn(x)-1} - 1 + 2^{hn(x)-1} - 1 + 1 = 2^{hn(x)} - 1$ nœuds. Cela achève la démonstration par récurrence.

Maintenant concluons. Soit h la hauteur de l'arbre rouge-noir. Comme tout nœud rouge a ses fils en noir, la hauteur noire est au moins $\frac{h}{2}$. Voici un exemple de cas limite :



On a $hn(x) = 1$ (il n'y a qu'un nœud noir en plus de x que l'on ne compte pas) et $h = 2$.

Donc dans l'arbre rouge et noir, via la propriété que l'on vient de montrer, il y au moins $2^{hn(x)} - 1 \geq 2^{\frac{h}{2}} - 1$ nœuds. Donc, $2^{\frac{h}{2}} \geq n+1$. En passant au logarithme, on a l'inégalité du théorème. ■

Maintenant, on implémente des opérations d'insertion et de suppression qui garantissent que l'arbre est toujours un arbre rouge et noir. Pour cela, on utilisera les rotations vues dans la partie précédente.

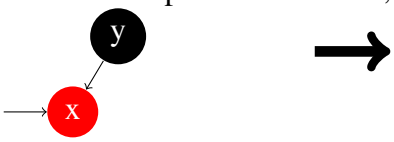
3.6.1 Insertion

L'insertion commence par une insertion pour les arbres binaires de recherche puis on peint le noeud inséré en rouge. Ensuite, le noeur inséré devient le noeud courant puis on applique les transformations de la figure suivante :

Transformations post-insertion

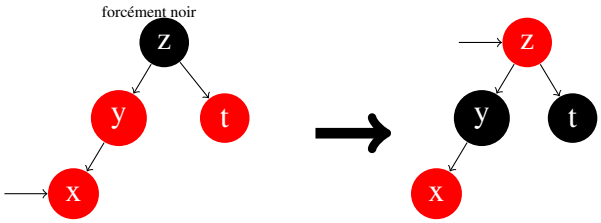
Si la racine est le noeud courant et est rouge, on la repeint.

Si le père de x est noir, tout est bon :

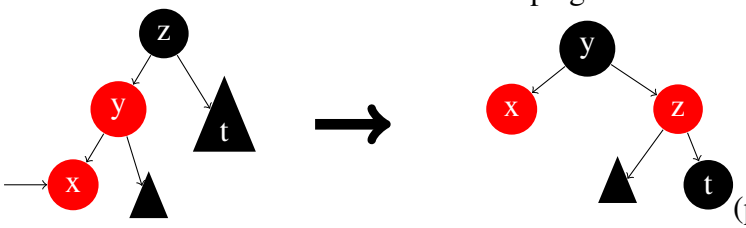


processus terminé

On considère alors le cas où le père de x est rouge. x a forcément un grand-père (sinon, son père serait noir). Voici la règle où l'oncle de x est rouge, où l'on repeint en noir le père et l'oncle rouges tous les deux :

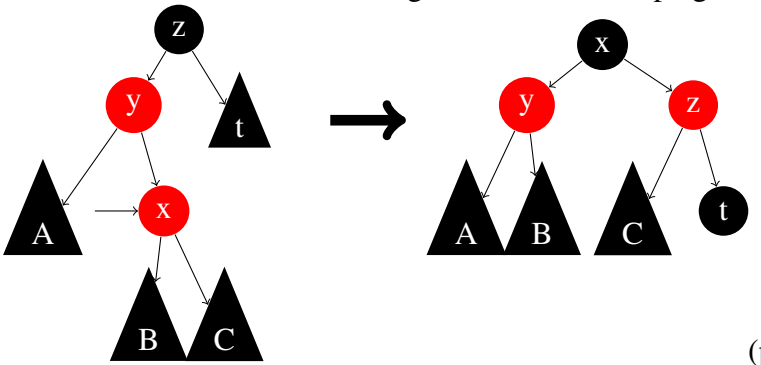


Si l'oncle est noir et que x est 'loin' de l'oncle (ici, x est à gauche et l'oncle à droite), alors on fait une rotation droite en repeignant les nœuds :



(processus terminé)

Si l'oncle est noir et que x est 'près' de son oncle (ici, x est un fils droit et l'oncle est à droite), alors on fait une rotation gauche-droite en repeignant les nœuds :



(processus terminé)

Durant le processus de rééquilibrage, soit on remonte de deux noeuds vers la racine, soit on fait une rotation (éventuellement une double-rotation) qui termine le processus. L'insertion est donc toujours en $O(\log n)$.

3.6.2 Suppression

Pour la suppression de x on reprend la suppression traditionnelle. On rappelle dans le cas générale d'une suppression où x a deux sous-arbres, il va 'croquer' le maximum m dans le sous-arbre gauche et on le place en x. Le placement de m dans le nœud x est anodin et le gros du travail réside dans la suppression du maximum. Une chose sûre, ce nœud maximum a au plus un sous-arbre. On se place alors ici dans le cas d'une suppression où le nœud au plus un sous-arbre et, s'il en a un, il est à gauche.

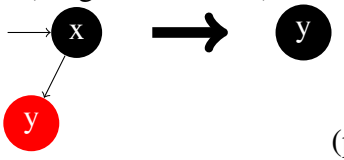
Si x n'a aucun fils et est rouge, il n'y a de danger.

rien (processus terminé)

Si x n'a aucun fils mais est noir, son père pointe désormais vers un 'nil' différent que l'on dit *gris* [1] : il symbolise un déséquilibre (son frère contient un nœud noir !).

(il va falloir gérer ce nœud gris)

Si x est noir mais a un fils (rouge forcément !), alors on le remplace par son fils peint en noir :



(processus terminé)

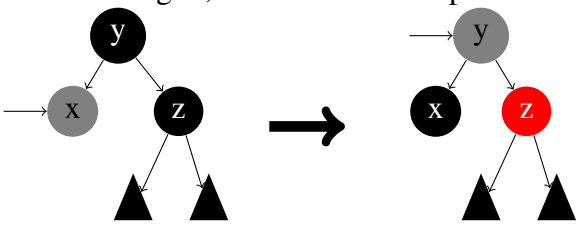
Toutes les opérations conservent le fait que c'est un arbre rouge et noir sauf l'apparition d'un nœud gris. Maintenant, il va falloir transformer l'arbre pour gérer le nœud gris. Le drapeau 'nœud gris' va remonter dans l'arbre jusqu'à la racine à l'aide des règles suivantes.

1. Beauquier dit 'dégradé'. Cormen parle p. 269 de noeud 'doublement noir'

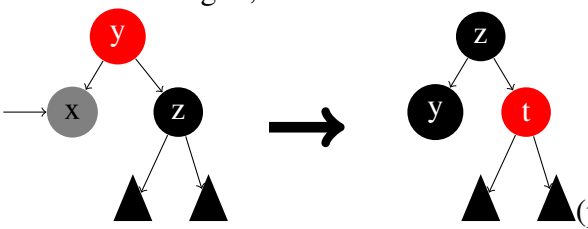
Transformations post-suppression

Si la racine est ‘gris’, on la peint en noir :

Si le frère du nœud gris, ses neveux et son père sont noirs :

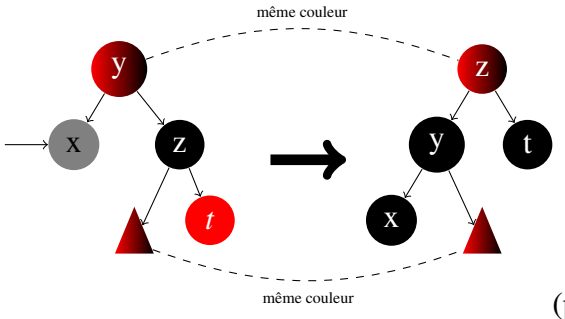


Si le frère du nœud gris, ses neveux sont noirs mais son père est rouge :



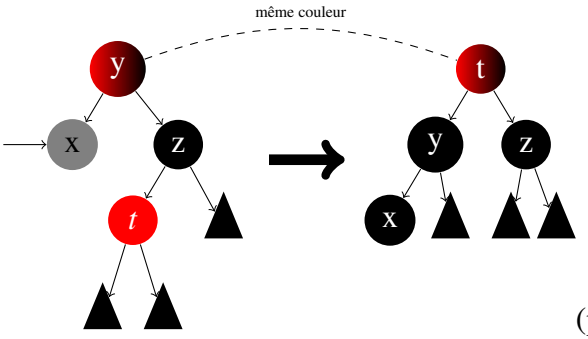
(processus terminé)

Si le frère du nœud gris est noir, et le neveu de droite est rouge (l’autre est de couleur quelconque) :



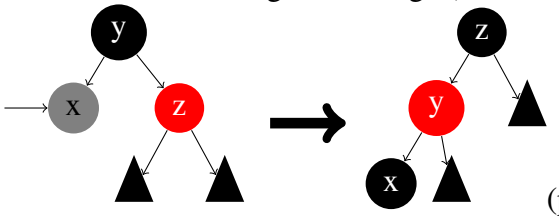
(processus terminé)

Si le frère du nœud gris est noir, et le neveu de gauche est rouge et le neveu de droite est noir :



(processus terminé)

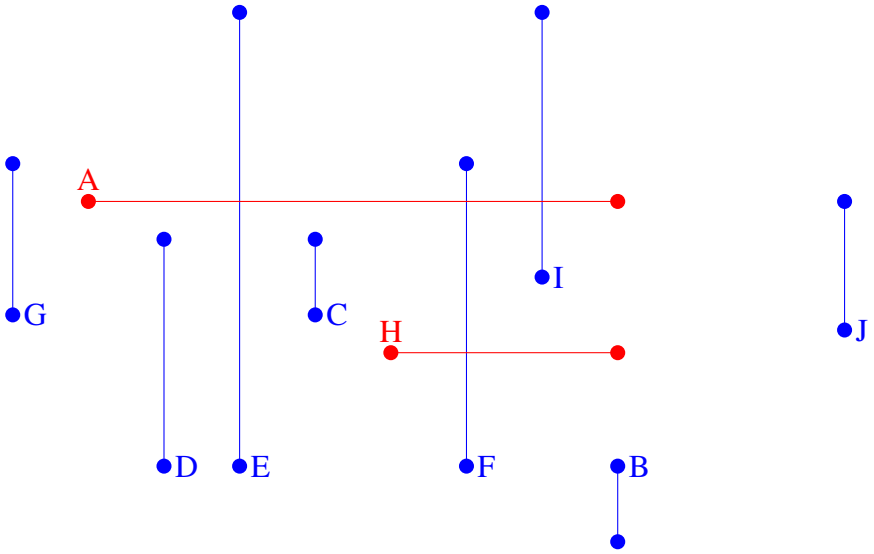
Si le frère du nœud gris est rouge (alors son père est noir) :



(processus terminé)

3.7 Application : Intersections de segments horizontaux et verticaux

[Sed84][Chapter 27] Il arrive souvent que l'on ait besoin de savoir si des objets s'intersectent. Par exemple dans les jeux vidéos, nous avons besoin de savoir si deux personnages se touchent. Dans la création d'un circuit électronique, nous avons besoin de savoir si deux fils se touchent. A priori, si l'on dispose de n objets, l'algorithme naïf consiste à tester toutes les paires possibles. C'est un algorithme en $O(n^2)$. Nous proposons une belle idée dans le cas où il s'agit de calculer l'intersection de segments horizontaux et verticaux. Prenons l'exemple suivant.



Sur cet exemple, le but est trouver que :
— A intersecte E, F et I ;
— et que H intersecte F.
Plus précisément, on pourra se donner comme premier objectif de construire une fonction

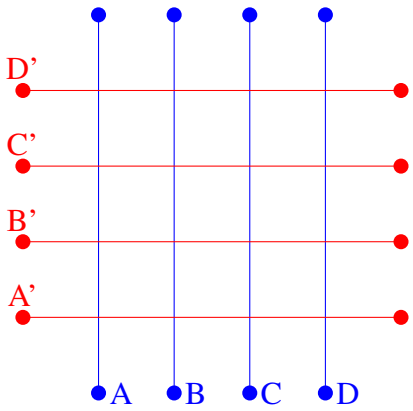
$$isPointIntersection(S)$$

qui prend en entrée une liste de segments horizontaux et verticaux S et qui retourne vraie si et seulement si deux segments s'intersectent. On veut que cette fonction soit efficace ($O(n \log n)$ par exemple ?).

Un deuxième objectif pourrait être de construire une fonction

$$getPointsIntersections(S)$$

qui prend en entrée une liste de segments horizontaux et verticaux S et qui retourne la liste des points d'intersections. Au pire des cas, la fonction $getPointsIntersections(S)$ est au moins en $\Theta(n^2)$ où n est le nombre de segments puisqu'il est possible d'avoir $\Theta(n^2)$ avec n segments et il faut au moins le temps de construire la liste des points d'intersections :



Mais souvent, le nombre d'intersections est petit. Et on voudrait un algorithme efficace dans ce cas. Ainsi, nous allons évaluer la complexité en le nombre de segments n et en le nombre de points d'intersections i ($O(n \log n + i)$ par exemple ?).

L'idée est de simuler un parcours de bas en haut. On 'active' un segment vertical lorsque l'on trouve son extrémité la plus basse. Lorsque l'on trouve l'extrémité la plus haute d'un segment vertical, il redevient inactif. Lorsque l'on trouve un segment horizontal, on regarde s'il intersecte les segments verticaux actifs. On note $\mathcal{V}_{\text{actifs}}$ l'ensemble des segments verticaux actifs.

Pour se faire, l'algorithme commence par trier les segments dans une liste triée par ordonnée croissante. Chaque segment vertical apparaît deux fois et chaque segment horizontal apparaît une fois. Dans le premier exemple cela donne :

$$B(\quad)B\quad D(\quad E(\quad F(\quad H\quad J(\quad C(\quad G(\quad)D\quad I(\quad C(\quad A\quad)G\quad)J\quad)F\quad)E\quad)I$$

où $B($ signifie que B devient actif, $)B$ signifique que B redevient inactif, etc.

Cette première phase coûte $O(n \log n)$ où n est le nombre de points.

Puis chaque extrémité la plus basse, par exemple $B($, correspond à l'ajout du segment vertical B dans $\mathcal{V}_{\text{actifs}}$:

$$\mathcal{V}_{\text{actifs}}.ajouter(B).$$

Chaque extrémité la plus haute, par exemple $)B$, correspond à la suppression du segment vertical B dans $\mathcal{V}_{\text{actifs}}$:

$$\mathcal{V}_{\text{actifs}}.supprimer(B).$$

Pour un segment vertical, par exemple H , nous allons regarder si le segment H intersecte un des segments verticaux de $\mathcal{V}_{\text{actifs}}$:

$$\mathcal{V}_{\text{actifs}}.intersecte?(H).$$

Une implémentation astucieuse consiste à implémenter l'ensemble $\mathcal{V}_{\text{actifs}}$ avec un arbre binaire de recherche équilibré où la clef d'un segment vertical B est son abscisse $B.x$. Tester l'intersection $\mathcal{V}_{\text{actifs}}.intersecte?(H)$ consiste à chercher s'il existe des éléments dans $\mathcal{V}_{\text{actifs}}$ dont la clef est comprise entre $H.x1$ et $H.x2$.

Chaque opération sur l'arbre binaire de recherche équilibré coûte $O(\log n)$. D'où une complexité de $O(n \log n)$ pour $isPointIntersection(S)$.

Si on veut afficher les points d'intersections, il faut remplacer $\mathcal{V}_{\text{actifs}}.intersecte?(H)$ par une fonction $\mathcal{V}_{\text{actifs}}.getIntersection(H)$ qui retourne tous les segments qui s'intersectent avec H. Un appel s'exécute en $O(\log n)$ + 'nombre d'éléments retournés'. En sommant, on a $O(n \log n + i)$.

Il est possible d'étendre cet algorithme si les segments sont quelconques [Sed84][Chapter 27].

L'algorithme peut s'écrire comme cela :

```
function isPointIntersections(H, V)
    T := un tableau de 2|V| + |H| cases
    for segment h ∈ H do
        ajouter h dans la prochaine case vide de T avec la clé (h.y, 1/2)
    endFor
    for segment v ∈ V do
        ajouter v[ dans la prochaine case vide de T avec la clé (s.ybas, 0)
        ajouter v] dans la prochaine case vide de T avec la clé (s.yhaut, 1)
    endFor
    Trier T selon la clé avec l'ordre ≤
    V_actifs := ∅
    for i := 1 à n do
        if T[i] est de la forme v[
            V_actifs.ajouter(v.x)
        else if T[i] est de la forme v]
            V_actifs.ajouter(v.x)
        else (T[i] est de la forme h, où h est un segment horizontal)
            if V_actifs.contientElementsEntre?(h.x1, h.x2) then
                return vrai
            endIf
    endFor
    return faux
endFunction
```

3.8 Variantes

— *Treaps* (ou *arbre-tas*). Comme l'équilibrage n'est parfois pas le meilleur choix et que l'on souhaite contrôler qui est plus en haut et en bas dans l'arbre, lorsqu'on insère un élément, on indique également une priorité. L'arbre est un arbre binaire de recherche pour les valeurs mais un tas pour les priorités. L'ajout se fait comme pour un ABR mais on fait des rotations pour corriger les priorités ne sont pas respectées. Les priorités peuvent aussi être générées aléatoirement et on a alors un arbre binaire de recherche aléatoire.

— *Splay trees*. Ce sont des ABR où, lors d'une recherche, l'élément recherché remonte à la racine. TODO :

— *Arbres d'intervalles*. Des fois, on veut stocker un ensemble d'intervalles $[d, f]$. Une requête peut-être, étant donné un intervalle I, de trouver s'il existe un intervalle $[d, f]$ de l'ensemble tel que $[d, f] \cap I \neq \emptyset$. Un arbre d'intervalles est un ABR pour les valeurs de début des intervalles d et on stocke également en plus dans chaque noeud x la borne supérieure la plus grande de tous les intervalles présents dans le sous-arbre issu de x.

— *Arbres persistents* Surtout dans les langages fonctionnels, on a besoin de souvenir de tous les ensembles au cours de l'utilisation de la structure de données. L'insertion est une fonction qui à un ABR associe un autre ABR dans lequel on a inséré un élément. Mais l'ABR avant insertion existe toujours. Par exemple, si on considère
$A := ABR.ajouter(arbrevide, 1)$
$A' := ABR.ajouter(A, 2)$
$A'' := ABR.ajouter(A', 3)$
alors A représente l'ensemble $\{1\}$, A' l'ensemble $\{1, 2\}$ et A'' l'ensemble $\{1, 2, 3\}$.
Pour cela, on recopie uniquement une branche jusqu'à l'élément inséré.

— *Prédécesseur et successeur en $O(1)$*. On construit une structure de données qui est la superposition d'un ABR et d'une liste doublement chaînée.