Projet LSINF1121 - Algorithmique et structures de données

-

Rapport intermédiaire Mission 4

Groupe 3.2

Boris Dehem (5586-12-00)

Sundeep Dhillon (6401-11-00)

Alexandre Hauet (5336-08-00)

Jonathan Powell (6133-12-00)

Mathieu Rosar (4718-12-00)

Tanguy Vaessen (0810-14-00)

Année académique 2014-2015

Questions et réponses

Question 1

Les clés mémorisées dans les nœuds d'un arbre binaire de recherche peuvent-elles être autre chose que des nombres ? Si oui, donnez un exemple, sinon justifiez pourquoi.
Comment faire pour énumérer en ordre croissant toutes les clés mémorisées dans un arbre binaire de recherche ? Quelle est la complexité temporelle de cette opération ?
Si une même clé est mémorisée deux fois dans un arbre binaire de recherche, les nœuds correspondants sont-ils en relation père-fils ? Parfois, toujours, jamais ? Justifiez votre réponse. (Boris)

Les clés d'un arbre binaire de recherche pourraient être de n'importe quel type d'objet, tant que celui ci peut être comparé à d'autres objets du même type, c'est à dire que pour deux objets différents on puisse définir lequel des deux est supérieur à l'autre. En d'autres mots tant qu'une classe implémente l'interface Comparable, les objets de celle ci peuvent être utilisés comme clés dans un arbre binaire de recherche. Par exemple, on pourrait utiliser des Strings, et classer les éléments de l'arbre alphabétiquement.
On peut définir une fonction récursive qui liste toutes les clés mémorisées dans un arbre binaire de recherche en ordre croissant. Cette fonction discerne deux cas : si elle est appelée sur une feuille, elle ne fait rien, sinon, elle appelle la même fonction sur le sous arbre de gauche, puis elle imprime la clé de l'arbre courant, puis elle s'appelle sur le sous arbre de droite. La complexité temporelle de cette opération sera $O(n)$ n étant le nombre de d'éléments dans l'arbre, car elle effectuera n fois un nombre constant d'opérations.
Une clé identique ne peut jamais être mémorisée deux fois dans un même arbre de recherche binaire, car lorsqu'on veut ajouter un nœud dont la clé existe déjà dans l'arbre, la valeur de ce nœud sera modifiée sans ajouter de nœud. Si c'était quand même possible, ces deux nœuds devraient alors être en relation père fils, car sinon, il y aurait un nœud dont la clé se situe entre les deux nœuds de même clé, ce qui est impossible car ils sont identiques.

Question 2

Un arbre binaire de recherche présente-t-il un avantage par rapport à une table de hachage pour réaliser un *Map* ? Pourquoi ? Et par rapport à une *Skip List* ?
La forme d'un arbre binaire de recherche dépend-elle de l'ordre d'insertion des clés ? La forme d'un arbre binaire de recherche dépend-elle de l'ordre de suppression des clés ? Quelle est la complexité temporelle de l'insertion de n clés identiques dans un arbre binaire de recherche initialement vide ? Quelle est la propriété particulière des arbres binaires de recherche qui justifie l'intérêt d'autres structures de données comme les Arbres AVL ou les Arbres (2,4) ? (**Sundeep**)

Si on utilise un arbre binaire de recherche, l'avantage principal est lié à l'efficacité des méthodes qui ne seront plus en complexité temporelle $O(n)$ mais bien en $O(log(n))$ dans le meilleur des cas et ce, surtout si l'arbre est équilibré. Au contraire, dans le pire des cas, la complexité temporelle sera en $O(n)$ vu que l'arbre n'est pas du tout équilibré et donc, la hauteur est de n.

Face à une *Skip List*, cet avantage n'en est plus un vu que la complexité temporelle dans ces deux cas est identique.

La forme d'un arbre binaire de recherche dépend effectivement de l'ordre dans lequel on ajoutera ou supprimera des clés. Si l'on ajoute les clés dans l'ordre croissant (ou décroissant), l'arbre ne sera dans plus du tout équilibré et aura une hauteur de n. La meilleure chose à faire serait d'ajouter les différentes clés en prenant les clés centrales.
Lors de la suppressions de clés, les vides sont compensés du fait qu'on va aller chercher la clé la plus à gauche du fils de droite pour remplir le vide créé. Ces suppressions peuvent donc bel et bien influencer la forme de l'arbre.

La complexité temporelle de l'insertion de n clés identiques dans un arbre binaire de recherche initialement vide sera de $O(n^2)$. A chaque ajout n, on devra parcourir l'entièreté de l'arbre. Le fait de devoir donc multiplier n par n explique bien le pourquoi du n^2.

La particularité des arbres binaire est que la complexité des méthode de bases varie entre O(n) et O(log n). On cherchera donc a maximiser l'apparition des situation dans lesquels la complexité se limitent à O(log n).

Pour ce qui concerne l'intérêt d'autres structures de données comme les Arbres AVL, il s'agit tout simplement d'essayer d'avoir une complexité temporelle de base qui varie entre $O(n)$ et $O(log(n))$ pour les principales opérations. Pour ce faire, nous allons maximiser l'obtention de situations telles que la complexité se limite à $O(log(n))$ et la seule correction à effectuer consiste à ajouter une règle à la définition de l'arbre binaire de recherche pour maintenir une hauteur en logarithme pour l'arbre. La règle en question est la **propriété d'équilibre de l'hauteur**, celle-ci caractérise la structure d'un arbre

binaire de recherche T en termes des hauteurs de ses noeuds internes. Autrement dit, pour chaque noeud interne v de T, les hauteurs des enfants de v diffèrents de 1 au maximum. Tout arbre binaire de recherche T qui satisfait cette dernière propriété est appelé un **Arbre AVL** [1]. Une conséquence directe de cette propriété est qu'un sous-arbre d'un arbre AVL est d'office un arbre AVL. Une autre propriété est que la hauteur d'un arbre AVL pour stocker n entrées est $O(log(n))$.

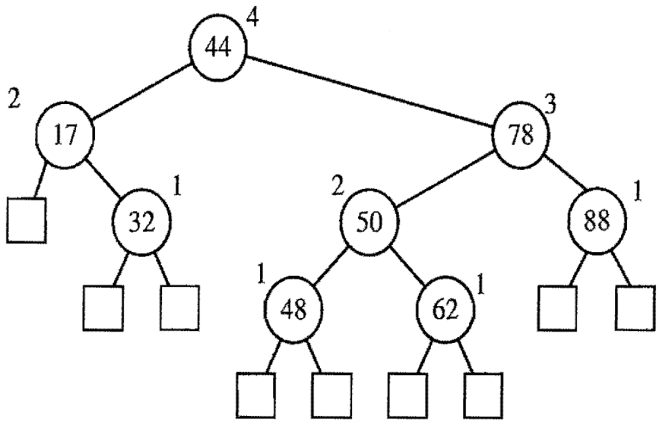


FIGURE 1 – Exemple d'arbre AVL. Les clés des entrées sont mises à l'intérieur des noeuds et les hauteurs des noeuds sont indiquées à côté de ces derniers.

1. Ce nom provient de ses inventeurs, à savoir : Adel'son-Vel'skii et Landis

Question 3

Proposez un algorithme pour réaliser la méthode *firstEntry()* (voir DSAJ-5 page 403 et DSAJ-6 page 396) dans un arbre binaire de recherche. Proposez un algorithme pour réaliser la méthode *higherEntry(k)* (voir DSAJ-5 page 403 et DSAJ-6 page 396) où k est une clé présente dans un arbre binaire de recherche.

Quelles sont les complexités temporelles de vos algorithmes en fonction de h (la hauteur de l'arbre) et en fonction de n (le nombre de noeuds de l'arbre) ? (Tanguy) [2]

Algorithme pour *firstEntry()*

```
public Entry<K,V> firstEntry(){
    Entry<K,V> p = root;
        if (p != null)
            while (p.left != null)
                p = p.left;
        return p;
}
```

Algorithme pour *higherEntry(k)*

```
public Entry<K,V> getHigherEntry(K key) {
    Entry<K,V> p = root;
    while (p != null) {
        int cmp = compare(key, p.key);
        if (cmp < 0) {
            if (p.left != null)
                p = p.left;
            else
                return p;
        } else {
            if (p.right != null) {
                p = p.right;
            } else {
                Entry<K,V> parent = p.parent;
                Entry<K,V> ch = p;
                while (parent != null && ch == parent.right) {
                    ch = parent;
                    parent = parent.parent;
                }
                return parent;
            }
        }
    }
    return null;
}
```

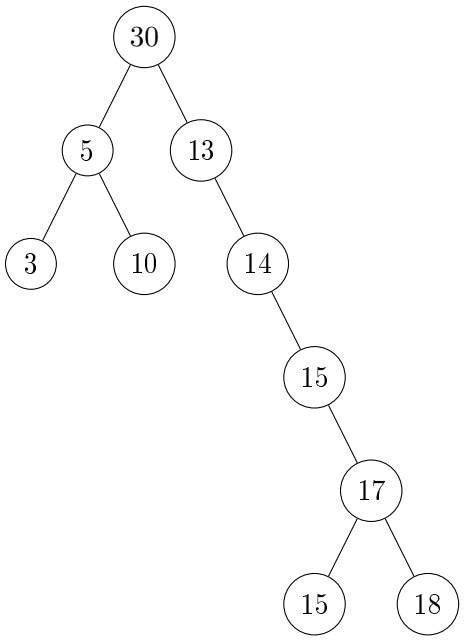
2. Sources : `http://docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html` et "TreeMap.java" issu du SDK 7

Pour les deux méthodes, la complexité temporelle est en $O(h)$ par rapport à la hauteur car il faut descendre tout en bas de l'arbre et en $O(\log(n))$ en fonction du nombre de noeuds car on ne tient compte que du sous-arbre de gauche à chaque itération.

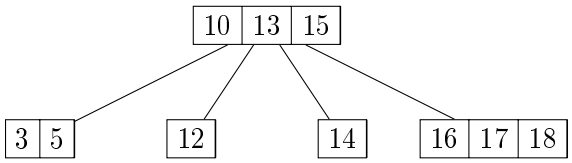
Question 4

Partant d'un arbre binaire de recherche initialement vide, comment se présente l'arbre après y avoir inséré les clés 12, 5, 10, 3, 13, 14, 15, 17, 18, 15 ? Pour les mêmes données comment se présenterait l'arbre finalement obtenu s'il s'agissait d'un arbre (2,4) ? Pour les mêmes données comment se présenterait l'arbre finalement obtenu s'il s'agissait d'un B-arbre d'ordre 4 ou d'un Splay Tree ? Cet exemple illustre-t-il les avantages ou inconvénients de ces différentes structures de données ? Pourquoi ? (Alexandre)

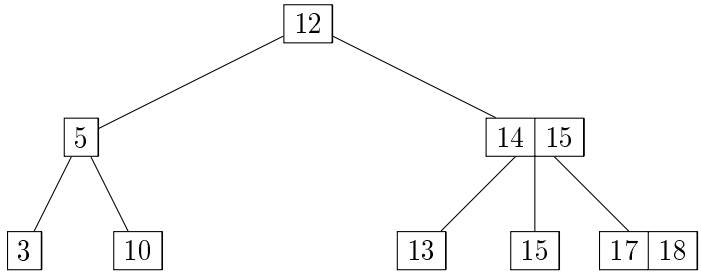
Arbre binaire de recherche



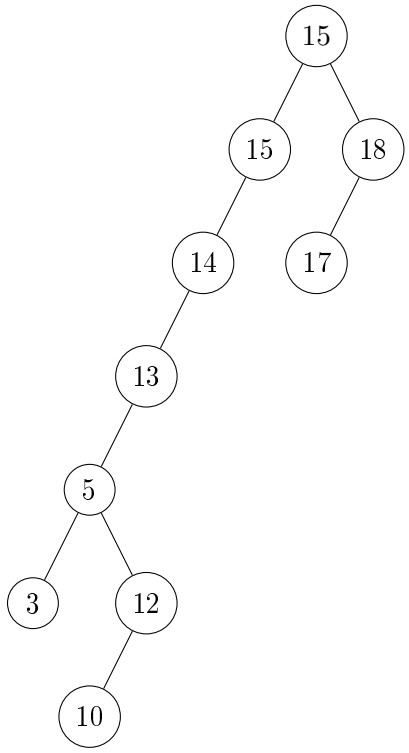
Arbre (2,4)



B-arbre d'ordre 4



Splay tree



Conclusion

Ces différents schémas permettent de mettre en évidence les différences qu'ils existent entre eux comme la profondeur de l'arbre, le nombre maximum d'élément fils d'un nœud...

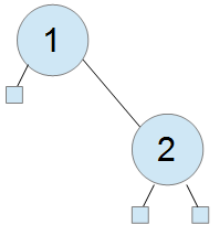
Question 5

L'ordre d'insertion des clés dans un AVL a t-il une influence sur la forme finale de l'arbre ? Jamais, parfois, toujours ? Justifiez vos réponses. Dessinez un arbre AVL de hauteur 5 ayant un nombre minimal de nœuds. Que peut-on dire de la relation entre hauteur h et nombre de nœuds n dans un arbre AVL ? *(Jonathan)*

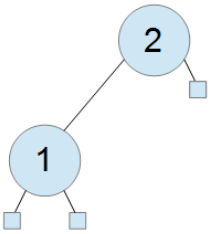
Oui, l'ordre d'insertion des clés dans un AVL à parfois de l'influence sur la forme finale de l'arbre.

Prenons le cas simple d'un arbre avec seulement deux clés 1 et 2. Suivant l'ordre dans lequel on insère les clés, la forme final de l'arbre sera différent.

Soit 1 puis 2

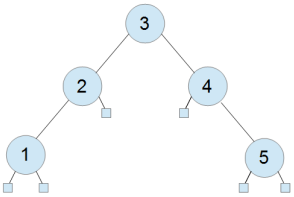


Soit 2 puis 1

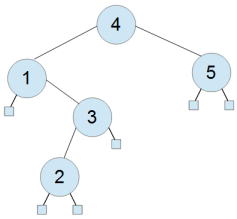


Ou bien encore pour un arbre avec 5 clés

Ordre : 3-2-1-4-5

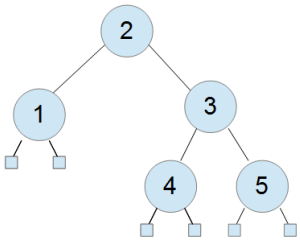


Ordre : 4-1-5-3-2

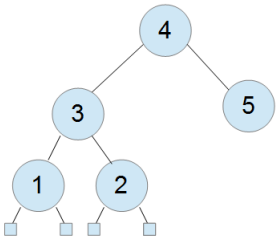


Cependant ces deux arbres ne sont pas des arbre AVL, il faut prendre en compte la propriété d'équilibre. On obtient donc les arbres ci dessous.

Ordre : 3-2-1-4-5 AVL



Ordre : 4-1-5-3-2 AVL



On remarque bien que l'ordre des clés a vraiment de l'importance pour la forme finale de l'arbre. Cependant, parfois ce n'est pas le cas. Prenons un exemple avec un AVL qui possède 3 clés 1, 2 et 3. dans l'ordre : 1-2-3 ou dans l'ordre : 3-2-1 on obtient le même arbre grâce à la propriété d'équilibre. Celui-ci est représenté ci dessous.

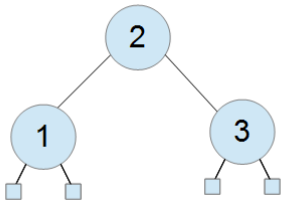


FIGURE 2 – Arbre AVL

C'est deux cas différents simple mais efficace, nous permis de prouver que dans la plupart des cas l'autre des clés à de l'influence sur la forme finale de l'arbre.

Voici un arbre AVL de hauteur 5 ayant un nombre minimal de nœuds De manière générale, on peut borner la hauteur d'un arbre binaire équilibré en fonction du nombre n d'entrées par la relation suivante :

$$log(n) \leq h \leq 2 \cdot log(n) + 2$$

que l'on peut aussi exprimer comme suivant : $2^{(h/2-1)} \leq n \leq 2^h - 1$.

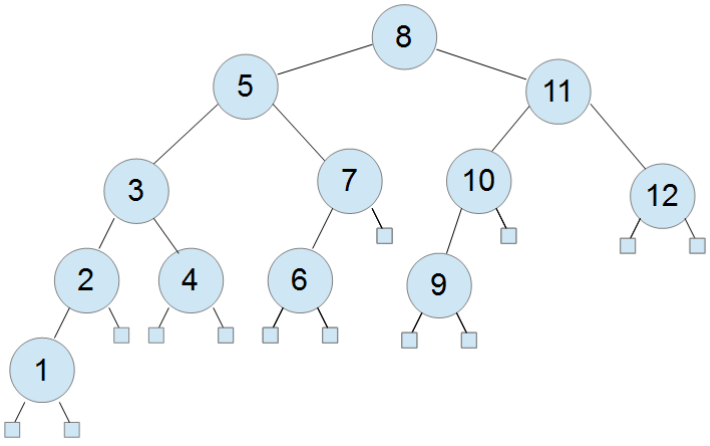


FIGURE 3 – Arbre AVL de hauteur 5 avec un nombre minimal de nœuds

Question 6

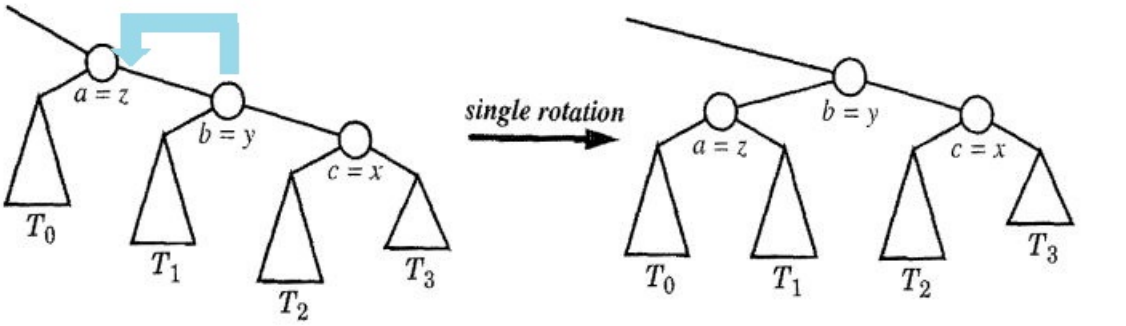
Nous considérons T et U deux arbres (2,4) mémorisant respectivement n et m clés tels que toutes les clés dans T sont strictement inférieures à tous les clés dans U. Proposez un algorithme pour fusionner T et U dans un seul arbre (2,4). Ce nouvel arbre (2,4) doit donc contenir toutes les clés de T et toutes les clés de U. La complexité temporelle de votre algorithme doit être en $O(logn + logm)$

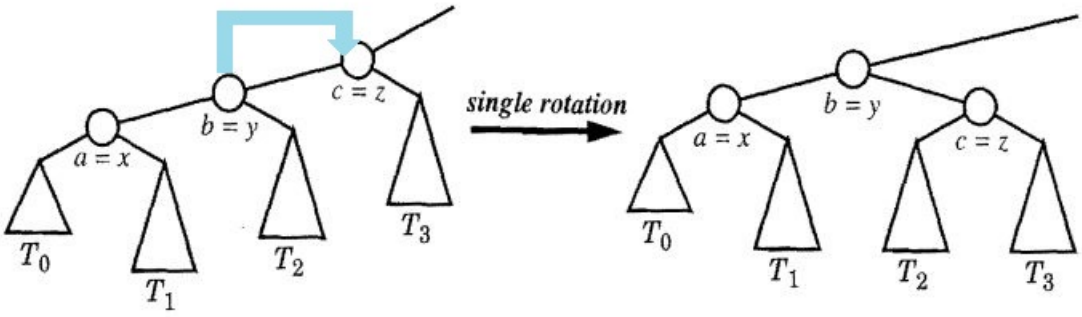
- Appliquez votre algorithme sur les deux arbres illustrés ci-dessous. Illustrez graphiquement la construction de l'arbre résultat pour chaque étape principale de votre algorithme. Note : votre algorithme doit pouvoir s'appliquer à n'importe quelle paire d'arbres (2,4) satisfaisant les conditions reprises dans l'énoncé de la question. On vous demande simplement d'illustrer le fonctionnement de votre algorithme général sur un cas particulier.
- Justifier pourquoi la complexité temporelle de votre algorithme est bien en $O(logn+logm)$, où n et m correspondent aux nombres de clés respectivement dans T et U. Opération d'ajout peut causer un overflow/une mauvaise taille dans l'arbre 1 comme dans l'arbre 2, alors put qui s'effectue en log(n) où n est le nombre d'entrées de l'arbre peut se propager vers le haut de l'arbre complet (arbre 1 précédent = n entrées) mais aussi vers le bas de l'arbre complet (arbre 2 précédent = m entrées) et ainsi s'effectuer en O(log(n)+log(m)) opérations.

Question 7

Faites une liste exhaustive des différents cas qui peuvent se présenter lorsque l'on ajoute un élément dans un arbre AVL et indiquez dans chaque cas la technique de ré-équilibrage utilisée (vue comme une ou plusieurs rotations). Complétez pour ce faire les cas décrits en annexe de ce document. (Tanguy) [3]

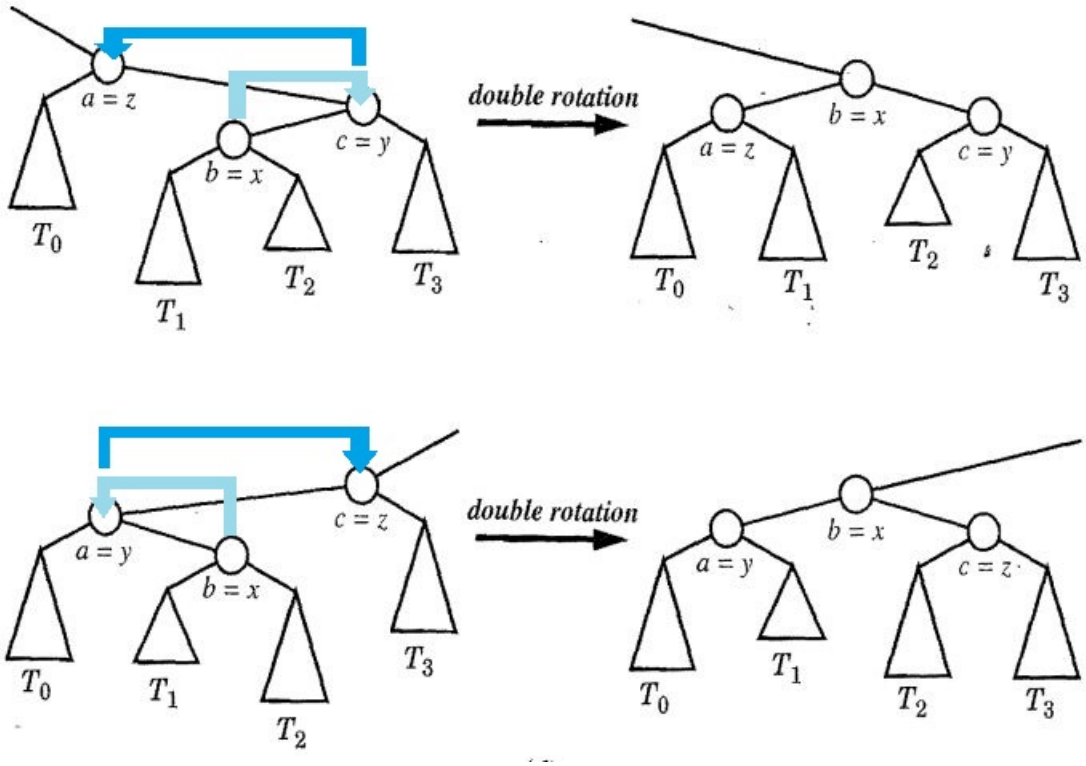
Cas 1 : Rééquilibrage par simple rotation





Cas 2 : Rééquilibrage par double rotation

3. Les figures sont issues de DSAJ-5 p448



Question 8

Supposons que nous disposions d'un arbre binaire de recherche mémorisant des clés entières entre 1 et 1000 et que nous y cherchions la clé 363. Parmi les séquences suivantes, quelles sont celles qui ne peuvent pas correspondre à la séquence des clés examinées ?

- 2, 252, 401, 398, 330, 344, 397, 363
- 924, 220, 911, 244, 898, 258, 362, 363
- 925, 202, 911, 240, 912, 245, 363
- 2, 399, 387, 219, 266, 382, 381, 278, 363
- 935, 278, 347, 621, 29

(Jonathan)

Toutes ces séquences sont possible sauf deux d'entre elle. La troisième séquences ainsi que la dernière des séquences.

En effet pour la séquence : 925, 202, 911, 240, 912, 245, 363 Le résultat 912 ne peut pas se trouver dans la séquence des clés. En effet après 911 nous nous trouvons dans son sous arbre gauche. Il est donc impossible de trouver la clés 912 dans ce sous arbre de gauche.

Pour la séquence 935, 278, 347, 621, 299, 392, 358, 363 Lors de la recherche après avoir passé la clés 347 nous nous trouvons dans son sous arbre de gauche. Étant donné que la

clés suivante est 612 nous nous trouvons forcément à une valeur supérieure à 347, mais inférieure à 621. Il est donc impossible de rencontrer la valeur 299.