Faculté des Sciences et Ingénierie - Sorbonne Université
Master Informatique parcours ANDROIDE

COMPLEX - Complexité, algorithmes randomisés et approchés

Rapport de projet

Arbres Cartésien et Algorithmes associés

Réalisé par :

PINHO FERNANDES Enzo

BEN SALAH Adel

Novembre 2024

Table des matières

Introduction du projet **2**

1 Arbres cartésiens - Premières propriétés **4**
a. Construction de l'arbre cartésien 4
b. Insertion de noeuds dans un ordre prédéfini 6
c. Implémentation de la structure de données d'un noeud 7
d. Implémentation de la structure de données d'un arbre cartésien 7
e. Construction manuelle de l'arbre cartésien de la question 1.a 7

2 Recherche dans un arbre cartésien **8**
a. Implémentation de l'algorithme de recherche d'un noeud 8
b. Complexité de l'algorithme de recherche de noeud 8

3 Insertion dans un arbre cartésien **10**
a. Violation de la propriété du tas lors de l'insertion (ABR) 10
b. Complexité de l'algorithme d'insertion d'un arbre cartésien 12
c. Implantation de l'algorithme d'insertion d'un arbre cartésien 14
d. Tests d'insertion de noeuds dans un arbre cartésien 14

4 Suppression dans un arbre cartésien **15**

Introduction du projet

Les arbres cartésiens sont une structure de données qui combine les propriétés des arbres binaires de recherche et des tas. Ils ont été proposés par Jean Vuillemin en 1980. Voici un aperçu de leur structure et de leur fonctionnement :

Structure

— **Noeuds :** chaque noeud d'un arbre cartésien contient une clé (qui respecte l'ordre d'un arbre binaire de recherche) et une priorité (qui respecte l'ordre d'un tas).

— **Arbre binaire de recherche :** les noeuds sont organisés de manière à ce que, pour tout noeud, les clés de son sous-arbre gauche soient inférieures à sa clé, et celles de son sous-arbre droit soient supérieures.

— **Tas :** les noeuds sont également organisés selon la priorité, de sorte qu'un parent ait toujours une priorité inférieure à celle de ses enfants.

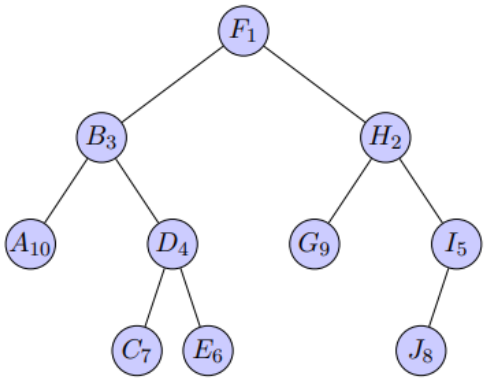


FIGURE 1 – Exemple d'arbre cartésien contenant les clés de {A,B,...,J} avec les valeurs de priorités dans {1,...,10} données en indice.

Propriétés

— **Équilibre probabiliste :** Les arbres cartésiens maintiennent une structure équilibrée de manière **probabiliste**. Cela signifie que les opérations d'insertion, de suppression et de recherche ont une complexité en temps moyenne de $O(\log n)$ même si dans le pire des cas, cela peut atteindre $O(n)$ (où n désigne le nombre de noeuds de l'arbre cartésien).

— **Insertion et suppression :** Lors de l'insertion, un nouveau noeud est placé comme un noeud feuille dans l'arbre binaire de recherche. Puis, si sa priorité est inférieure à celle de son parent, il est "élevé" à travers des rotations jusqu'à ce que l'ordre du tas soit respecté (*cf.* Exercice 3).

— **Recherche :** La recherche d'une clé suit les règles d'un arbre binaire de recherche, ce qui la rend efficace.

Les arbres cartésiens sont utilisés dans diverses applications où une structure de données dynamique et équilibrée est nécessaire, comme la gestion de fichiers, les bases de données et les algorithmes de traitement d'événements. Le but de ce projet est d'implanter et d'analyser cette structure de données et d'effectuer des tests d'efficacités.

Exercice 1

Arbres cartésiens - Premières propriétés

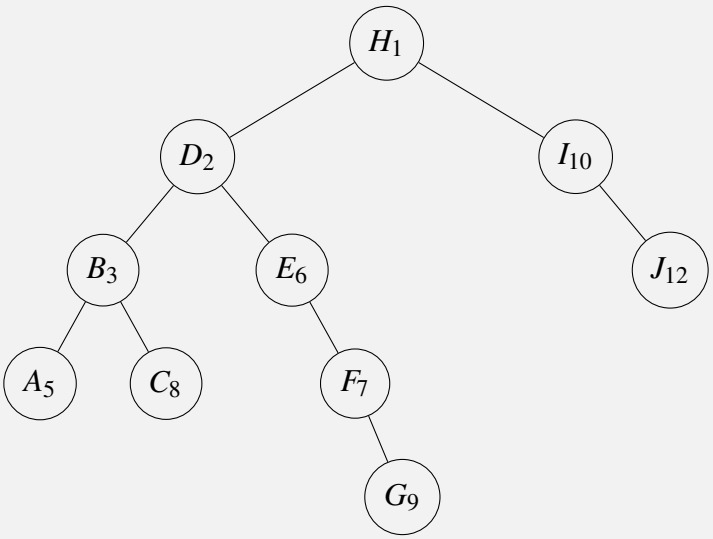
1.a] Construire un arbre cartésien dont les noeuds sont donnés par la liste suivante (la lettre représentant la clé du noeud et l'entier sa priorité) :

$$(A:5),(B:3),(C:8),(D:2),(E:6),(F:7),(G:9),(H:1),(I:10),(J:12)$$

Existe-t-il plusieurs solutions ? Qu'en est-il pour un arbre cartésien dont toutes les priorités sont différentes (ce que nous ferons dans toute la suite du projet). Justifier votre réponse.

Réponse à la 1.a - 1/2

Arbre cartésien de la liste précédente :



Il n'existe pas plusieurs solutions, et cela est vrai également pour tout arbre cartésien dont toutes les priorités sont différentes, grâce aux propriétés d'un arbre de recherche binaire et d'un tas.

En effet, dans un arbre cartésien, la **racine** doit être le noeud avec **la priorité la plus faible** parmi tous les noeuds par la propriété du tas. Par conséquent, il n'y a qu'une possibilité, le choix de la racine est unique.

Réponse à la 1.a - 2/2

Une fois cela fait, l'ensemble des autres noeuds de priorité inférieure seront divisés en deux sous-ensembles selon leur clé, d'après la propriété d'un arbre de recherche binaire :

- Le sous-arbre gauche contiendra tous les noeuds ayant des clés inférieures à celle de la racine.
- Le sous-arbre droit contiendra tous les noeuds ayant des clés supérieures à celle de la racine.

Ces deux sous-ensembles sont indépendants l'un de l'autre, et surtout uniques. Chacun de ces sous-ensembles doivent représenter eux-même un arbre cartésien.

On suivra le même raisonnement récursivement sur les sous-ensembles gauche et droit :

- On choisit le noeud du premier sous-ensemble avec la priorité la plus faible comme racine.
- On choisit le noeud du second sous-ensemble avec la priorité la plus faible comme racine.

Et on recommence jusqu'à que tous les noeuds soient placés dans l'arbre cartésien.

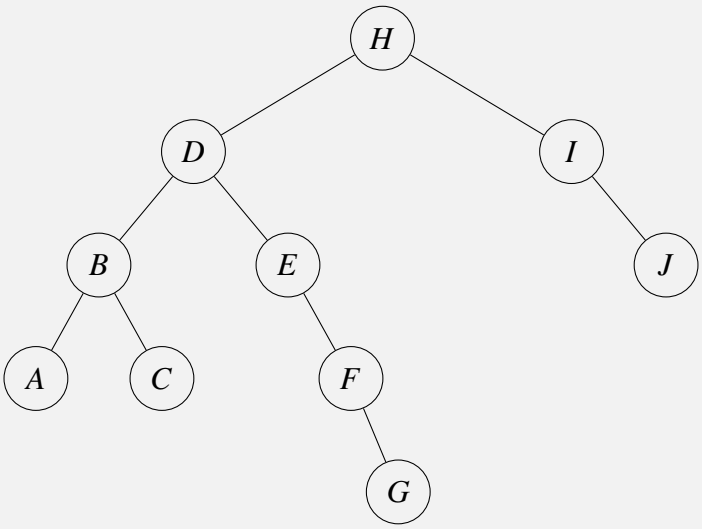
En conclusion, **tout arbre cartésien est unique** (si toutes les priorités sont différentes) puisque chaque décision de construction est unique, alias :

- Le choix de la racine, de par sa priorité la plus faible de son sous-ensemble, grâce à la propriété du tas.
- Le choix des deux sous-ensembles, de par les clés des noeuds, grâce à la propriété des arbres de recherche binaire.

1.b] Considérer l'arbre binaire de recherche construit en insérant dans l'ordre les noeuds dont les clés sont les suivantes : $H, D, B, A, E, F, C, G, I$ et J et comparer à l'arbre cartésien de la question précédente. Généraliser et démontrer ce résultat pour un arbre cartésien général (dont les priorités sont différentes).

Réponse à la 1.b - 1/1

L'arbre issu de l'insertion de ces noeuds, dans cet ordre, donne exactement le même arbre cartésien de la question précédente.



Nous avons cette corrélation entre les deux arbres parce que nous avons inséré les noeuds selon leur priorité (en ordre croissant) dans l'arbre cartésien.

On peut généraliser ce résultat en : un arbre cartésien est un ABR où nous avons inséré les noeuds dans l'ordre croissant de leur priorité (si elles sont toutes différentes).

En effet, trois propriétés sont visibles dans ce cas précis :

- La racine des deux arbres sont les mêmes, alias le noeud dont la priorité est la plus faible.
- La propriété de l'ABR est respecté dans les deux cas, étant donné la nature des arbres.
- La propriété du tas est respecté pour l'arbre cartésien bien évidemment, mais également pour cet ABR. En effet, pour ce dernier, on insère les noeuds dans l'ordre croissant de leur priorité. D'après l'algorithme d'insertion de ce dernier, il deviendra forcément une des feuilles de l'arbre à la fin de l'insertion, et il n'y a pas de rotation. Par conséquent, tout parent de cet arbre aura une priorité plus faible que leurs enfants.

Étant donné que tout arbre respectant les propriétés d'un ABR et d'un tas est unique, et que dans ce cas précis, nos deux arbres respectent ces propriétés et possèdent la même racine, alors... Ils sont égaux, dû à leur unicité.

1.c] Programmer la structure de données d'un noeud (contenant la clé et la priorité, le pointeur vers le fils gauche, le pointeur vers le fils droit) avec les constructeurs et les fonctions associés.

Réponse à la 1.c - 1/1

Nous avons décidé d'implémenter nos différentes structures de données en JAVA, pour sa simplicité quant à l'utilisateur des pointeurs ainsi que de la mémoire.

En effet, le garbage collector s'occupe de libérer la mémoire dès qu'un objet n'est plus référencé, ce qui nous sera très pratique au moment de la suppression des noeuds.

L'implémentation se trouve dans le fichier : ./src/Node.java

1.d] Programmer la structure de données d'un arbre cartésien (avec au minimum un constructeur pour l'arbre cartésien vide, une fonction pour tester si un arbre cartésien est vide, pour accéder au fils droit/gauche d'un noeud donné).

Réponse à la 1.d - 1/1

L'implémentation se trouve dans le fichier : ./src/CartesianTree.java

1.e] Avec votre implantation, construire "manuellement" l'arbre cartésien de la question 1.a.

Réponse à la 1.e - 1/1

La construction de l'arbre se trouve dans le fichier test : ./test/Exo_1_e.java

Le test "testTreeStructure" affiche l'arbre obtenu dans la console de débogage.

Exercice 2

Recherche dans un arbre cartésien

2.a] Implanter l'algorithme de recherche d'un noeud contenant une clé donnée dans un arbre cartésien. L'algorithme est identique à celui de recherche dans un arbre binaire de recherche.

Réponse à la 2.a - 1/1

Les méthodes `searchKey` et `searchKeyRec` dans le fichier ./src/CartesianTree.java correspondent à l'implantation de l'algorithme de recherche de noeud. (l.60-85)

2.b] Donner la complexité de cet algorithme en fonction de la profondeur du noeud recherché (en cas de recherche fructueuse) ou de la profondeur de son successeur ou prédécesseur du noeud recherché (en cas de recherche infructueuse).

Réponse à la 2.b - 1/2

L'algorithme de recherche d'un noeud dans un arbre cartésien est le même que celui dans un ABR.

Lors de la recherche, à chaque itération, on identifie trois cas :

- Si la clé recherchée est la clé du noeud courant, on a trouvé le noeud.
- Si la clé recherchée est inférieure à celle du noeud courant, on continue la recherche chez le fils gauche.
- Si la clé recherché est supérieure à celle du noeud courant, on continue la recherche chez le fils droit.

En toute logique, la complexité de la recherche dépendra donc du nombre de noeuds parcourus depuis la racine, jusqu'au noeud recherché si la recherche est fructueuse, sinon, jusqu'à une feuille en cas de recherche infructueuse.

Posons $d(x)$ la profondeur du noeud x étant soit le noeud recherché en cas de recherche fructueuse, sinon la feuille atteinte. Alors la complexité de la recherche est en $O(d(x))$.

Réponse à la 2.b - 2/2

Mais ce n'est pas tout, on peut distinguer deux cas :

— **Le cas moyen :** Quand l'arbre cartésien est "relativement équilibré", alors la profondeur moyenne d'un noeud est environ à $O(\log n)$, n étant le nombre de noeuds de l'arbre. Ce sera donc la complexité de la recherche dans le cas d'un arbre cartésien dont les ensembles de clés et de priorités sont bien répartis.

— **Le cas critique :** Quand l'arbre est déséquilibré, par exemple un arbre qui représente une liste, alors la profondeur serait en $O(n)$. Ce sera donc la complexité dans le pire cas, où l'ensembles de clés et de priorités sont mal répartis.

En conclusion, on aura comme complexité pour un arbre cartésien à n noeuds :

— **S'il est équilibré :** $O(\log n)$

— **S'il est déséquilibré :** $O(n)$

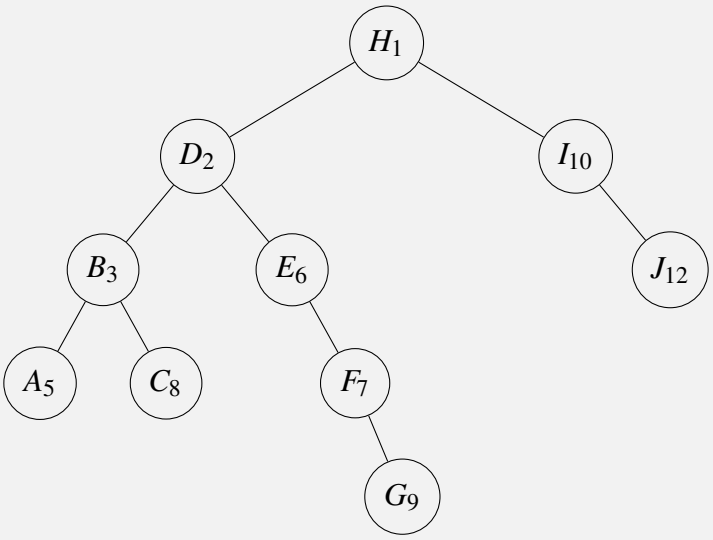
Exercice 3

Insertion dans un arbre cartésien

3.a] Montrer, sur l'exemple de la question 1.a. que l'insertion d'un noeud dans un arbre cartésien en suivant la méthode d'insertion dans un arbre binaire de recherche, peut résulter en un arbre qui ne vérifie plus la propriété de tas.

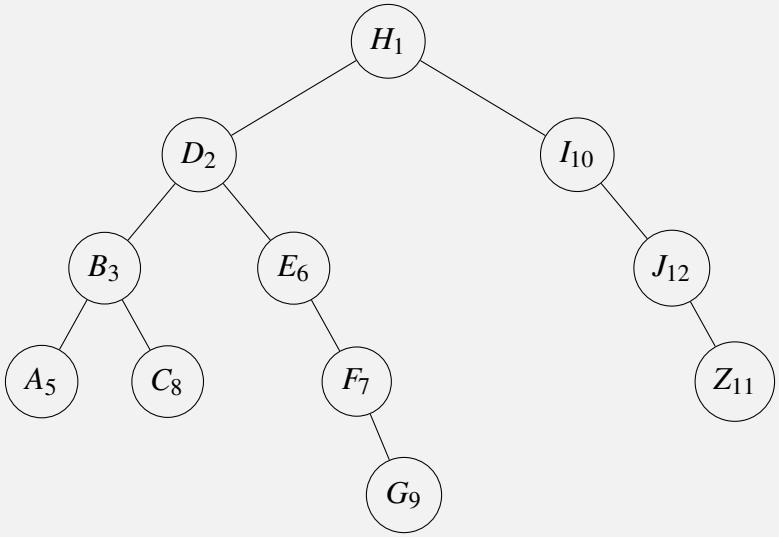
Réponse à la 3.a - 1/2

Prenons l'exemple de la question 1.a. Pour rappel, l'arbre cartésien est ce dernier :



Réponse à la 3.a - 2/2

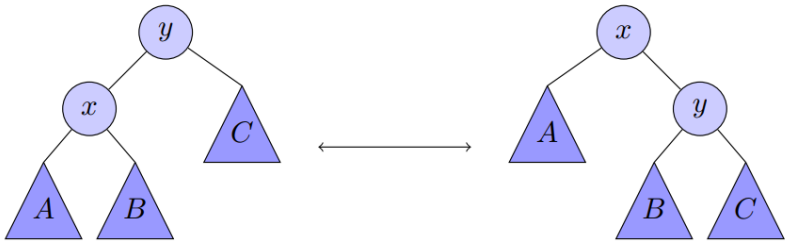
Essayons d'insérer le noeud (Z : 11) suivant l'algorithme d'insertion d'un ABR. Le résultat sera :



Nous voyons bien que cet arbre ne respecte pas les propriétés d'un arbre cartésien, étant donné qu'un parent a une priorité plus grande qu'un de ses fils, ici (J : 12) alias le parent de (Z : 11).

Par conséquent, l'insertion d'un noeud dans un arbre cartésien en suivant la méthode d'insertion dans un arbre binaire de recherche peut résultat en un arbre qui ne vérifie plus la propriété du tas.

Pour réparer cela, l'insertion dans un arbre cartésien va se faire arbre cartésien va se faire initialement comme dans un arbre binaire de recherche mais devra effectuer des *rotations* d'arbre (voir le diagrame suivant pour une rotation au noeud x) pour rétablir la propriété du tas.



Une telle rotation préserve la propriété d'arbre binaire mais permet de rétablir la propriété de tas. Désormais pour insérer un noeud z dans un arbre cartésien, il sera inséré comme dans un arbre binaire de recherche mais une fois cela effectué, tant que sa priorité est inférieure à celle de son noeud parent, effectuer une rotation au noeud z. Chaque rotation diminue la profondeur du noeud z de 1 et augmente celle de son parent de 1. Les rotations peuvent être effectuées en temps constant puisqu'il s'agit simplement de manipulation de pointeurs.

3.b] Donnez la complexité de cet algorithme en fonction de la profondeur de son prédécesseur.

Réponse à la 3.b - 1/2

L'insertion dans un arbre cartésien suit deux étapes :

- L'insertion classique dans un arbre binaire de recherche.
- La succession de rotations jusqu'à que la propriété du tas soit respectée.

Posons x le noeud à insérer.

L'insertion d'un noeud dans un ABR consiste à une suite de comparaison pour rechercher la future place de ce dernier, en respectant l'ordre des clés. Comme pour son algorithme de recherche, son insertion repose sur le nombre de comparaisons nécessaires pour trouver la position du nouveau noeud, alias $d(x)$ la profondeur du noeud inséré.

Donc pour la première étape, l'insertion classique d'un ABR a une complexité de $O(d(x))$.

Réponse à la 3.b - 2/2

La seconde étape consiste à une succession de rotations jusqu'à que la propriété du tas soit respectée. Ces rotations n'affectent en rien la propriété des ABR. A chaque rotation, la profondeur du noeud inséré diminuera de 1. On recommence cette étape jusqu'à que la propriété du tas soit respectée.

Dans le pire des cas, nous devons remonter le noeud jusqu'à la racine, on aura donc $d(x)$ rotations à faire car on le remonte d'un seul niveau par rotation. C'est le nombre maximal de rotations.

Etant donné que chaque rotation se fait en temps constant, la complexité totale des rotations est en $O(d(x))$.

La complexité totale de l'insertion dans un arbre cartésien est donc de :

$$O(d(x)) + O(d(x)) = O(d(x))$$

En conclusion, pour les mêmes raisons qu'en 2.b, on aura comme complexité pour un arbre cartésien à n noeuds :

- **S'il est équilibré :** $O(\log n)$
- **S'il est déséquilibré :** $O(n)$

3.b] Implanter cet algorithme.

Réponse à la 3.c - 1/1

Les méthodes `insert` et `insertRec` dans le fichier ./src/CartesianTree.java correspondent à l'implantation de l'algorithme d'insertion de noeud. (l.95-136)

3.d] Appliquer votre algorithme pour créer les arbres cartésiens obtenus en insérant les noeuds suivant dans l'ordre donné :

1. (A :5), (B :3), (C :8), (D :2), (E :6), (F :7), (G :6), (H :1), (I :10), (J :12)
2. (H :1), (G :9), (A :5), (B :3), (D :2), (F :7), (C :8), (J :12), (I :10), (E :6)
3. (E :6), (H :1), (B :3), (D :2), (C :8), (F :7), (G :9), (J :12), (A :5), (I :10)

et vérifier que vous obtenez bien à chaque fois le même arbre cartésien que dans la question **1.a.**

Réponse à la 3.d - 1/1

Les tests d'insertion se trouvent dans le fichier test : ./test/Exo_3_d.java

Chaque test vérifie que l'arbre obtenu est bien celui de la question **1.a** grâce à la méthode `equals` qui teste si un arbre est égal à un autre (selon les clés). De plus, chaque arbre est affiché dans la console de débogage.

Exercice 4

Suppression dans un arbre cartésien

A finir...