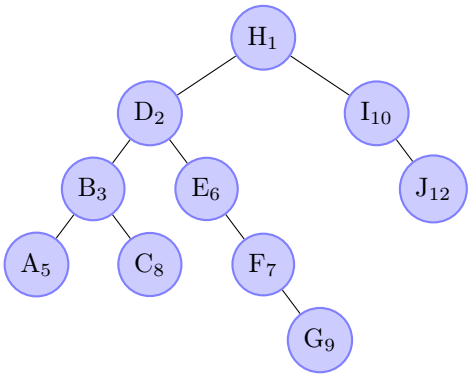
COMPLEX rapport

Paul Cibier

November 2024

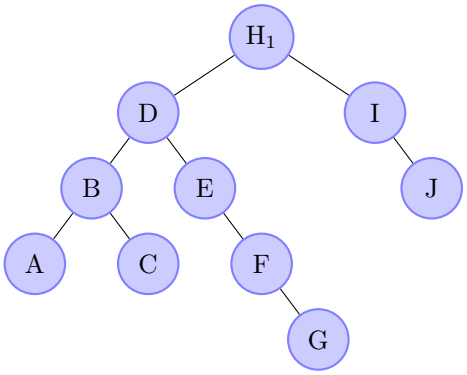
Exercice 1 Arbres cartésiens - Première propriétés

1.a] Graph question



Non il n'existe qu'une seul solution car toutes les priorités de l'arbre cartésien sont différentes. Pour un arbre cartésien dont toutes les priorités sont différentes, il n'existe toujours qu'une solution car l'arbre doit respecter la priorité d'un arbre binaire et d'un tas.

1.b] Arbre binaire de recherche



Les deux arbres on la même structure.
Pour un arbre cartésien dont toutes les priorités sont différentes, on peut construire un arbre binaire qui aura la même structure, la même solution en insérant les nœuds de l'arbre cartésien dans l'ordre croissant de priorités dans l'arbre binaire.

1.c]

Voir struct Node dans le fichier tree.rs

1.d]

Voir is empty(), get left child() et get right child() dans le fichier tree.rs

1.e]

Voir manually construte 1a() dans le fichier main.rs

Exercice 2 Recherche dans un arbre cartésien

2.a]

Voir la fonction bin search() dans le fichier tree.rs

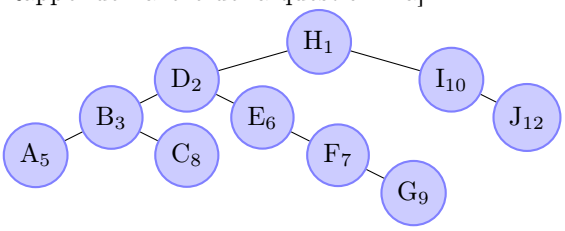
2.b] Complexité de recherche en fonction de la profondeur

Dans le cas d'une recherche fructueuse :
La complexité en fonction de la profondeur p est de p
Dans le cas d'une recherche non fructueuse :
Si la profondeur de son successeur est supérieur à celle de son successeur alors la complexité sera p la profondeur p de son successeur sinon la complexité sera p la profondeur p de son prédécesseur.

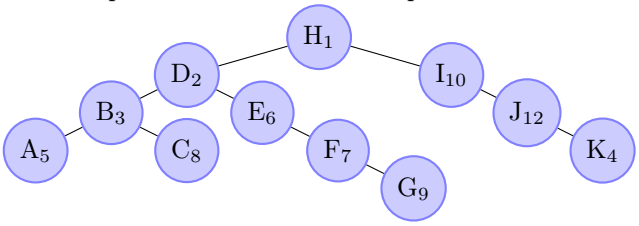
Exercice 3 Insertion dans un arbre Cartésien

3.a]

Montrons que l'insertion dans un arbre cartésien suivant méthode d'insertion dans un arbre binaire de recherche ne vérifie plus la propriété du tas.
Rappel de l'arbre de la question 1.a]



J'essaie d'insérer le nœud K avec une priorité de 4 en suivant le méthode d'insertion d'un arbre binaire de recherche.
Étapes :
1. Entre dans le nœud H : K > H donc visite du fils droit.
2. Entre dans le nœud I : K > I donc visite du fils droit.
3. Entre dans le nœud J : K > J donc visite du fils droit, pas de fils droit.
Donc on place le nœud K en tant que fils droit du nœud J.



On peut observer sur le nouvelle arbre que l'arbre ne respect pas les propriétés du tas car le nœud K avec une priorités de 4 a une priorité inférieur à celle de son parents le nœud J avec priorité 12.

3.b]

Dans le meilleur des cas $\mathcal{O}(1)$ car le nœud insérer aura une priorité inférieur à celle du nœud parent.
Dans le pire des cas il faut remonter jusqu'à la racine donc $\mathcal{O}(p)$ opération pour y arriver.
En moyenne cela donne $\mathcal{O}(p/2)$ comme complexité en général (les priorités suivent une loi uniforme).

3.c]

Voir insert() dans le fichier tree.rs J'ai considérer dans mon implémentation que chaque clé était unique donc quand on insert un nœud qui est déjà présent, l'algorithme retourne sans rien faire. On peut imaginer d'autre scénario où si la

priorité est différente de celle du nœud déjà présent alors la priorité du nœud est mise à jour avec la nouvelle.

3.d]

J'ai implémenter un test en Rust ou j'insère les différents nœuds dans l'ordre demandé et j'effectue un test à la fin pour savoir si la séquence des nœuds rencontré lors d'un bfs sur l'arbre résultant des insertions était la même que la séquence de nœuds du bfs de l'arbre de la première question. Si les deux séquences de nœuds sont égal alors l'arbre est le même. En plus du parcours BFS, je test également le parcours DFS de l'arbre. Vous pouvez vérifier et tester les tests avec la commande : Cargo test

Exercice 4 Suppression d'un nœuds dans un arbre cartésiens

4.a] Explication rotation vers le bas

Le problème si l'on supprime un nœud à la place qu'il est dans l'arbre, on se retrouve avec ses deux enfants sur les bras. Lorsque l'on fait des rotations vers le bas du nœud avec son fils ayant la plus petit priorité jusqu'à ce que le nœud se retrouve sans aucun fils et donc qu'il peut être supprimer. Cela permet de respecter la propriété du Tas et de l'arbre binaire de recherche comme dans l'insertion.

4.b] Complexité

Pour supprimer un nœud dans un arbre cartésien il faut d'abord le trouver. Donc la complicité sera d'au moins celle de la recherche en fonction de la profondeur donc p.
Dans le pire cas, l'arbre est un binaire qui ressemble à un peigne de sorte chaque nœud à deux fils mais que le fils la priorité la plus élevé n'est pas de fils et que celui ayant la priorité la plus faible ait toujours un fils ou aucun qui l'on a plus de nœud. Dans ce cas l'algorithme devra faire des rotations jusqu'à descendre à la profondeur la plus baisse qui est égale à n / 2. Dans le meilleur cas, le nœud que l'on cherche à supprimer est à la racine et il n'as pas de fils ou 1 seul. dans ce cas l'arbre est vide ou le seul fils du nœud à supprimer devient la racine de l'arbre. Dans le cas moyen ou on obtiendras une complexité qui sera entre celle de la complexité dans le pire des cas et dans le meilleur cas donc entre O(1) et O(n/2) donc O(n/4).

4.c]

Voir remove() dans tree.rs

4.d] Application de l'algorithme.

Les arbres résultant sont disponible dans le test de suppression. Cargo test pour visualiser le résultat des tests.

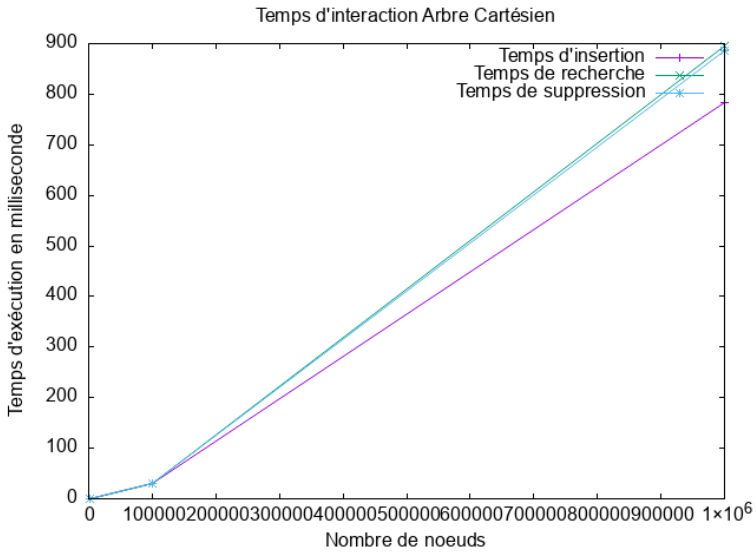
Exercice 5 Propriétés aléatoires - Aspect expérimental

5.a] Métrique utiles

— Les métriques pertinentes pour évaluer l'équilibre d'un arbre est de mesurer la différence entre le nœud vide le plus profond dans l'arbre et celui le moins profond de l'arbre ou de manière plus précise de mesurer la variance des profondeurs des nœuds de l'arbre, on peut également utiliser le coefficient de variation pour mesurer la dispersion des profondeur. Pour évaluer l'efficacité d'un arbre cartésien par rapport à d'autre structure de données sera de mesurer le temps que prennent les différents type d'interaction(INSERTION, SUPPRESSION, RECHERCHE) à s'effectuer pour N nœuds.

— Pour l'analyse du déséquilibre.
Stat depth | mean : 10.636, variance : 10.262, coef variation : 30.118
1000 nodes in 0 msec
Stat depth | mean : 20.404, variance : 20.186, coef variation : 22.020
100000 nodes in 39 msec
Stat depth | mean : 24.359, variance : 26.015, coef variation : 20.939
1000000 nodes in 976 msec
Stat depth | mean : 29.127, variance : 30.010, coef variation : 18.808
10000000 nodes in 17203 msec

Pour l'analyse de l'efficacité :



— Un exemple de séquence qui pourrait déséquilibre un arbre binaire de recherche serait une séquence de nœuds où les nœuds serait insérer par ordre croissant de clé. Avec sa structure aléatoire, un arbre cartésien se défend car le fait que les priorités soit tiré aléatoirement pour chaque nœuds, ce ne sera pas toujours le cas d'avoir le nœuds avec la clé la plus faible avec la priorité la plus faible et donc se retrouver en haut de l'arbre.

— Je ne pense pas qu'il est un réel impact, le nœud sera placé en dessous du nœud avec la même priorité que lui mais si il y en a trop l'arbre risque de se déséquilibrer.

Exercice 6 Priorités aléatoires Aspect théorique

Soit p_k : $v.a$ représentant la profondeur de x_k
On rappel que les priorités sont tirée aléatoirement sur une loi uniforme.

6.a] Montrons que $\mathbb{E}(p_k) = \sum_{i=1}^{n} \mathbb{E}(X_{ik})$

p_k : $v.a$ qui représente la profondeur du nœud x_k
X_{ij} : v.a qui vaut 1 si x_i est un ancêtre de x_j et 0 sinon.

$$\sum_{i=1}^{n} X_{ik}$$

Nous donne le nombre de sommets qui sont des ancêtres propres de x_k.
Donc nous donne la profondeur de x_k car on connaît le nombre d'ancêtre propre de ce nœud.

Donc en moyenne la profondeur de x_k sera égale à :

$$\mathbb{E}(\sum_{i=1}^{n} X_{ik})$$

Par linéarisation de l'espérance, on obtient :

$$\sum_{i=1}^{n} \mathbb{E}(X_{ik})$$

Donc la moyenne de la profondeur de x_k est de $\sum_{n=1}^{n} \mathbb{E}(X_{ik})$

6.b] Montrons $X_{ik} = 1$ ssi x_i plus petite prio de $X(i,k)$

$X(i,k)$ est l'ensemble des nœuds $\{x_i, x_{i+1}, \ldots, x_k\}$ ou $\{x_k, x_{k+1}, \ldots, x_i\}$ suivant selon i < k ou i > k.
Si x_i n'est pas le nœud avec la plus petite priorité de l'ensemble $X(i,k)$ alors il existe un nœud situé entre x_i et x_k en terme de clé qui aura la priorité la plus petite de l'ensemble $X(i,k)$ et donc pour respecter l'ordre du tas se retrouvera être l'ancêtre propre de chacun des nœuds de l'ensemble $X(i,k)$.

Dans le cas où le nœud qui la plus petite priorité n'est pas x_k :
Pour respecter la propriété d'un arbre binaire tous les nœuds inférieur à lui même seront dans le sous arbre accessible depuis sont fils gauche et tous les nœuds supérieur à lui seront dans le sous arbre de sont fils gauche.
Donc selon que i < k ou i > k.
x_i sera dans le sous arbre gauche et x_k dans le sous arbre droit.
ou
x_k sera dans le sous arbre gauche et x_i dans le sous arbre droit.
Si le nœud avec la priorité la plus petite est x_k dans l'ensemble $X(i,k)$ alors x_i ne sera pas un ancêtre propre de x_k car ce sera x_k qui sera l'ancêtre propre de x_i.

6.c] Montrons profondeur moyenne d'un nœud est $\mathcal{O}(\log n)$

p_k : $v.a$ représentant la profondeur d'un nœud x_k
X_{ik} : $v.a$ 1 si x_i est un ancêtre propre de x_k, 0 sinon.

Montrons que dans un arbre cartésien aléatoire, la profondeur moyenne d'un nœud est $\mathcal{O}(\log n)$.
On rappel que pour chaque nœud, sa priorité est tiré selon une loi uniforme.

On a montré dans la Question 6.1 que pour un nœud k avec la k-ième clé la plus petite de l'arbre Cartésien sa profondeur moyenne était égale à $\sum_{i=1}^{n} \mathbb{E}(X_{ik})$. Cette profondeur moyenne ne dépend pas de la clé k dont elle

est se généralise à l'ensemble des nœuds de l'arbre Cartésien.
Donc pour tout nœud k de l'arbre Cartésien sa profondeur sera égale à

$$\mathbb{E}(p_k) = \sum_{i=1}^{n} \mathbb{E}(X_{ik})$$

$$\mathbb{E}(p_k) = \sum_{i=1}^{n} P(\{x_i \text{ soit ancêtre propre de } x_k\})$$

On a montré précédemment que x_i était un ancêtre propre de x_k uniquement si x_i avait la plus petite priorité parmi l'ensemble $X(i,k)$ Donc 1 chance sur $|i-k|$.

$$\mathbb{E}(p_k) = \sum_{i=1}^{n} \frac{1}{|i-k|+1}$$

Rappel sur les nombres Harmonique :

$$H_n = \sum_{j=1}^{n} \frac{1}{j} \approx \ln(n) + \gamma$$

Donc

$$\begin{aligned} & \leq 2 \cdot H_n \\ \mathbb{E}(p_k) = \sum_{i=1}^{n} \frac{1}{|i-k|+1} & = 2(\ln n + \gamma + O(1) \\ & = O(\log n) \end{aligned}$$

Donc

$$\mathbb{E}(p_k) \sim \mathcal{O}(\log n)$$

Exercice 7 Supplément

7.a] Rust

J'ai choisie d'implémenter mon projet dans le langages Rust car c'est un langages considérer "memory safe" et aussi rapide et avec le même niveau de contrôle sur la mémoire que le C.

7.b] Implémentation Générique

Mon implémentation de l'arbre cartésien est implémenté avec des types générique K pour représenter le type de la clé et le type P pour représenter le type de la priorité. Cela veut dire que mon arbre cartésien peut être utilisé avec n'importe quelle type primitif (n'importe quel type integer ou flottant.) mais également utilisé des types composé comme par exemple un Arbre Cartésien donc je peux avoir des Arbres Cartésiens qui accepte des clés qui sont des arbres cartésiens.

7.c] Allocateur

J'ai également changé l'allocateur mémoire de programme et utilisé mimalloc au lieu de l'allocateur de mémoire par défaut du système ce qui me donne une réduction de 5%-15% du temps d'insertion, suppression et recherche pour 1M de nœuds. Les gains varie d'une OS à l'autre.