

## Projet COMPLEX Arbres Cartésiens

### EXERCICE 1

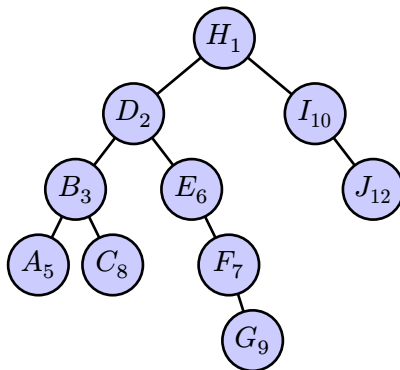


Fig. 1. – Arbre cartésien  $\mathcal{A}$

*a.* Il n'existe qu'une unique solution pour cette liste. En effet, toutes les priorités sont différentes, il n'y a donc pas de *choix* disponible au moment de la construction. Si toutes les priorités n'étaient pas différentes, alors on n'aurait pas un ordre totale. On pourrait ajouter un critère de sélection, comme la position dans la liste (la clé du nœud).

*b.* On observe que l'arbre produit à partir des clés suivantes est le même que celui obtenu dans la question 1. On observe que les clés étaient triées selon l'ordre croissant de priorité. On émet donc l'hypothèse suivante : trier les nœuds en suivant l'ordre des priorités, puis construire l'arbre binaire en suivant l'ordre des nœuds résulte en la création d'un arbre cartésien général.

Un arbre cartésien suit deux propriétés, d'une il est un arbre binaire de recherche, c'est à dire que les nœuds sont organisés de manière à ce que, pour tout nœud, les clés de son sous-arbre de gauche soient inférieures à sa clé, et celles de son sous-arbre droit soient supérieures.

Cette propriété est satisfaite par la construction de notre arbre. Il doit satisfaire la priorité que les nœuds également organisés selon la priorité, de sorte qu'un parent ait toujours une priorité inférieure à celle de ses enfants. Cette propriété est satisfaite par le fait que la liste est triée par ordre croissant de priorité. Ainsi, un nœud fils aura toujours une priorité supérieure à celle de son parent.

*c.* Voir `Node.cpp` et `Node.h` pour l'implémentation du nœud.

*d.* Voir `CartesianTree.cpp` et `CartesianTree.h` pour l'implémentation de l'arbre cartésien.

*e.* Voir la fonction `exercice_1` pour la construction « manuelle » de l'arbre cartésien de la figure 1.

### EXERCICE 2

*a.* Voir la fonction `CartesianTree::find` pour l'implémentation de la recherche d'un nœud et `exercice_2` pour un exemple de recherche minimal.

*b.* Dans le cas d'une recherche fructueuse, soit  $k$  la profondeur du nœud. On aura une comparaison par échec (tant que l'on n'est pas encore au nœud) et une comparaison pour valider que la clé est bien la bonne. On aura donc bien  $k$  comparaisons, c'est à dire  $k$  nœuds parcourus. Dans le cas d'une recherche infructueuse, on note  $k_p$  et  $k_s$  la profondeur de son prédécesseur et successeur, au sens des clés, respectivement. Un nœud absent serait toujours contenu entre son prédécesseur et son successeur. De la structure d'un arbre cartésien, s'il n'y a aucun nœud entre un prédécesseur et son successeur, c'est à dire que le nœud recherché n'est pas présent, alors on

a deux scénarios possibles. Soit le prédécesseur est le père du successeur, soit le successeur est le père du prédécesseur. Pour trouver une feuille vide, il faut donc aller jusqu'à  $\max\{k_p, k_s\}$ .

### EXERCICE 3

a. On reprend l'arbre construit en question 1.a (et 1.b). On cherche à ajouter un nouveau nœud  $K_4$  dans cet arbre. Le nœud sera ajouté, suivant la construction d'un arbre binaire, comme le fils droit de  $J_{12}$ . Cependant, soit  $\mathcal{P}(n)$  la priorité du nœud  $n$ ,  $\mathcal{P}(K_4) = 4$  et  $\mathcal{P}(J_{12}) = 12$  et  $J_{12}$  est père de  $K_4$ , ce qui contredit la propriété du tas.

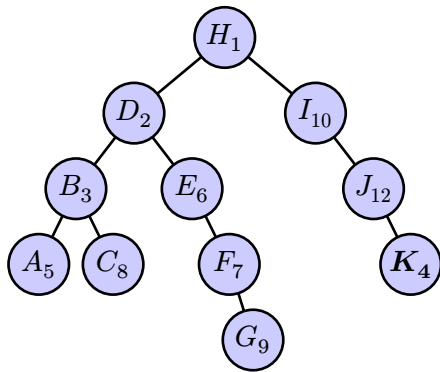


Fig. 2. – L'ajout de  $K_4$  à  $\mathcal{A}$  est faux

b. La complexité de l'ajout dans un arbre binaire est de  $O(k)$  (2.b) et la complexité d'une rotation est en temps constant. Lorsque l'on cherche le point où insérer le nouveau nœud, on fait au plus  $k$  opérations. On a donc une complexité totale pour l'insertion dans un arbre cartésien en suivant cette méthode de  $O(k)$ .

c. Voir la fonction `CartesianTree::insert` pour l'implémentation de l'insertion d'un nouveau nœud dans un arbre cartésien.

d. Voir la fonction `exercice_3` pour la construction de l'arbre de la figure 1 avec différents ordres pour les nœuds.

### EXERCICE 4

a. On propose l'algorithme pour supprimer des nœuds dans un arbre cartésien. On commence

par faire des rotations entre le nœud que l'on veut faire et son fils de plus petite priorité. On fait des itérations de ces rotations jusqu'à ce que le nœud soit une feuille. On supprime alors le nœud.

Si le nœud est déjà à une feuille, alors si on le supprime, on ne perturbe pas ses fils (il n'en a pas), on ne perturbe pas l'ordre des clés, donc on conserve bien la propriété d'arbre binaire et on ne perturbe pas l'ordre des priorités, donc on conserve bien la propriété de tas. On peut donc supprimer les feuilles dans un arbre cartésien.

On note  $p(x)$  la priorité associée à un nœud. Dans le cas où le nœud n'est initialement pas une feuille, on peut établir que puisqu'on inverse un nœud avec un de ces nœuds fils, on arrivera bien à partir d'une certaine profondeur à un nœud qui n'a plus de fils. D'où l'algorithme amène bien le nœud à une feuille. Lorsque l'on fait l'inversion entre un nœud et son fils de priorité minimale, on conserve bien un arbre binaire pour tous les nœuds sauf pour le nœud à supprimer. On va noter  $z_i$  avec deux fils gauche et droit  $z_j$  et  $z_k$ . On va s'intéresser à l'inversion de  $z_i$  et  $z_j$  sans perte de généralité. Comme l'arbre est initialement cartésien en tous points sauf en  $z_i$ , on a bien  $z_j < z_k$ . Donc lorsque l'on place  $z_j$  comme ancêtre de  $z_k$ , on est bien  $z_k$  la clé de l'arbre droit de  $z_j$  et cette clé est bien supérieure ( $z_k > z_j$ ). Donc on conserve bien la propriété d'arbre binaire. De plus, comme on sélectionne le  $\min\{p(z_j), p(z_k)\}$  par construction, donc ici,  $p(z_j) < p(z_k)$ .  $z_j$  devient alors parent de  $z_k$  et a bien une priorité inférieure. On conserve donc bien la priorité de tas.

On a bien établi que si on a un arbre cartésien en tout sommet sauf  $z$ , alors en faisant l'inversion, on obtient bien un nouvel arbre cartésien en tout sommet sauf  $z$ . Il ne reste plus qu'à établir qu'on peut supprimer sans poser de soucis  $z$  lorsque c'est une feuille. Cela est dû au fait que l'arbre est cartésien en tout sommet sauf  $z$  et qu'on a pas de fils gauche ou fils droit qui pourrait perturber la propriété d'arbre en retirant le nombre d'arêtes (tout arbre a  $n - 1$  arêtes).

Suite à la suppression du sommet qui rendait l'arbre non cartésien, l'arbre redevient alors un arbre cartésien. Ceci explique ce pourquoi ce procédé de suppression fonctionne.

*b.* On a au plus  $h - k$  inversions à faire. Chaque inversion est en temps constant et la suppression finale d'une feuille est aussi en temps constant. D'où la complexité de l'opération de suppression est en temps  $O(k)$ .

*c.* Voir la fonction `CartesianTree::delete` pour la suppression d'un nœud dans l'arbre.

*d.* Voir la fonction `exercice_4` pour un le test de suppression de nœuds dans l'arbre.

## EXERCICE 5

*a.* L'implémentation de la mesure de performance a été réalisée dans les fichiers `include/performance.h` et `src/performance.cpp`.

Les métriques choisies pour évaluer la performance des arbres cartésiens sont de deux types : des métriques structurelles et des métriques de temps d'exécution. Pour les métriques structurelles, on a choisi la hauteur de l'arbre, la profondeur moyenne d'un nœud, l'équilibre de l'arbre au sens de la hauteur de ses fils et l'équilibre de l'arbre au sens de la taille de ses fils. Les métriques de temps d'exécution concernent les opérations usuelles implémentées dans le reste de ce projet. C'est à dire la vitesse moyenne d'insertion, de recherche fructueuse, de recherche infructueuse et de suppression.

La méthodologie de test pour mesurer et comparer est la suivante. Tout d'abord, nous avons testé les métriques de temps d'exécution sur trois structures de données différentes : l'arbre cartésien, mais aussi un arbre binaire de recherche et un tas binaire. Ces deux structures ont été implémentées dans `include/BinarySearchTree.hpp` et `include/BinaryHeap.hpp` et ont été choisies parce que les arbres cartésiens combinent les propriétés de ces deux arbres. Pour les deux arbres, on a testé les métriques structurelles (ces métriques n'ont pas de sens pour les tas binaires).

On a aussi voulu tester plusieurs variations d'insertions et de collisions. Pour le tas, on a fait que tester pour chaque taille d'instance. Pour les arbres, on a généré des données aléatoires initiales, mais on a créé les arbres avec trois stratégies distinctes, soit au hasard, soit par ordre croissant au sens des clés, soit par ordre décroissant au sens des clés. Finalement, pour l'arbre cartésien, on a aussi mesuré l'impact des collisions sur la performance, on a donc introduit  $\alpha \in [0, 1]$  qui correspond à la réduction du domaine de choix pour les clés ( $\alpha = 0.1$  veut dire qu'au lieu d'avoir  $n$  clés distinctes, on aura  $\frac{n}{100}$  clés distinctes).

On a évalué à différentes tailles d'instance pour les différentes structures de données, et on a effectué chaque test en faisant 50 tirages aléatoires pour la création des données de la structure et en notant sa médiane.

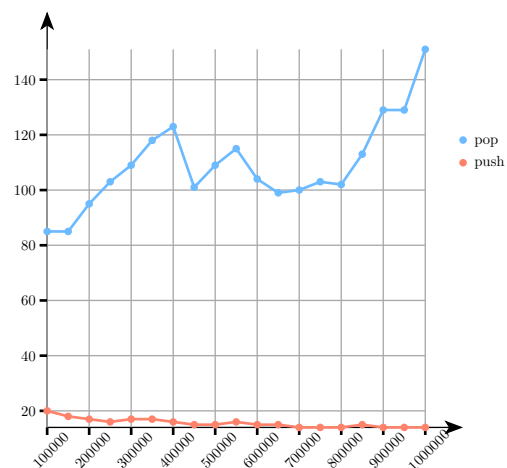


Fig. 3. – Tas binaire

Pour le tas binaire, on observe une remarquable vitesse de push et de pop. Ce qui fait sens, en revanche, dans une telle structure la vitesse de recherche est en  $O(n)$ , en comparaison à nos arbres binaires qui auront une complexité en  $O(\log n)$ . On peut quand même noter que la vitesse d'un tas ne croît pas aussi bien que celle d'un arbre pour la recherche, mais que les opérations fondamentales sont si rapides qu'il est peu probable qu'un arbre soit plus efficace dans beaucoup de cas.

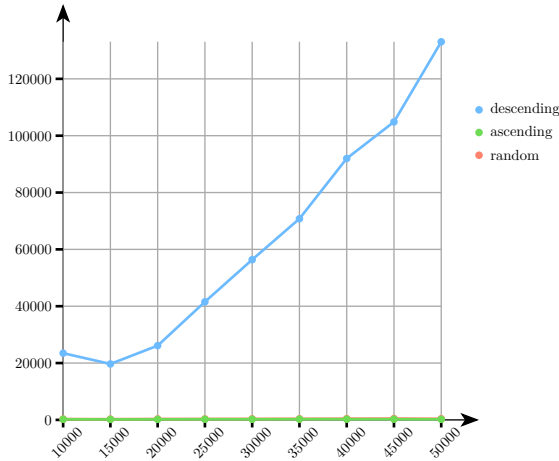


Fig. 4. – Insertion - ABR

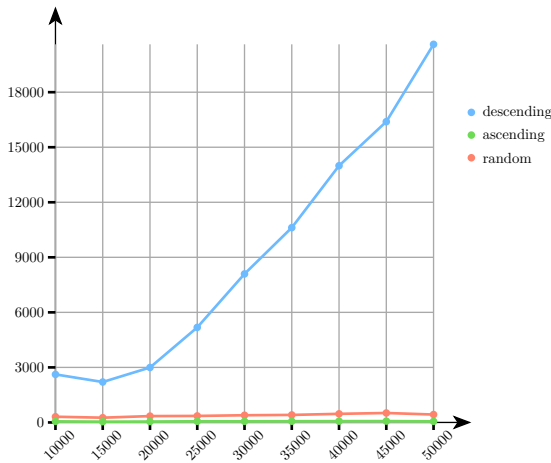


Fig. 5. – Supression - ABR

On observe la performance des trois opérations fondamentales sur un arbre binaire est parmi tous nos graphes de loin la pire. Ceci est du à un cas particulier : si on trie les nœuds à ajouter par ordre de clé décroissant, alors non seulement l'arbre est déséquilibré (ce qui est aussi le cas sur un ordre croissant), mais on se retrouve dans le pire cas pour l'insertion et la suppression. On peut voir sur ces graphes exactement à quel point ce déséquilibre a un impact sur les performances réelles de nos graphes.

On note aussi que nous avons choisi de ne pas représenter la vitesse de recherche dans toutes nos structures parce que nos mesures étaient toujours 0. Sur cette taille d'arbre, le fait de ne pas avoir à alouer de mémoire et de faire au pire un simple parcours suffit à rendre la vitesse de cette opération complètement négligeable.

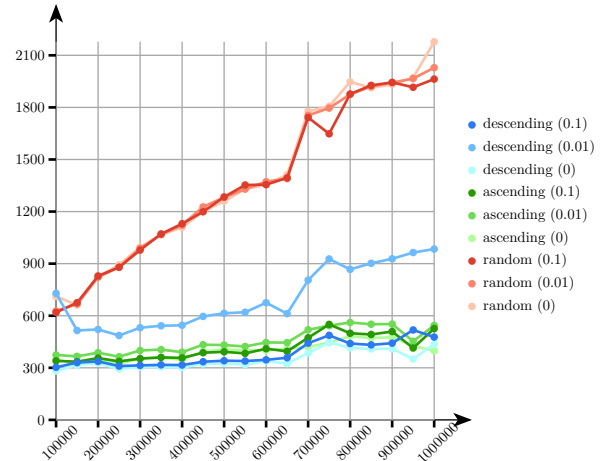


Fig. 6. – Insertion - Arbre cartésien

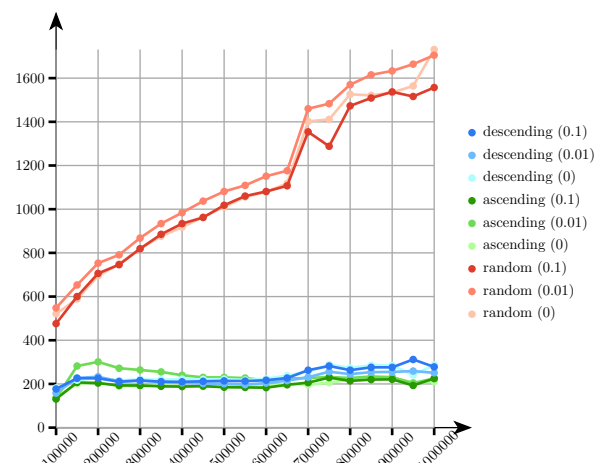


Fig. 7. – Supression - Arbre cartésien

Sur la performance de nos arbres cartésiens, on observe plusieurs résultats clés. Le plus clair étant la dégradation de performance dans le cas de priorité aléatoire. La dégradation en elle-même n'est pas étonnante, car toutes les opérations sont en  $O(k)$  avec  $k$  la profondeur moyenne d'un nœud, qui comme on le démontre dans l'exercice 6 est en  $O(\log n)$ . Cela veut dire que la performance des opérations devrait être en  $O(\log n)$ . On observe bien une courbe logarithmique, lorsque l'on suit une stratégie d'insertion aléatoire. Cela étant dit, il est donc plus surprenant d'observer que les stratégies d'insertion croissante et décroissante résultent en un gain de performance. On peut supposer que ce gain de performance est du à une simplification dans la construction de l'arbre, qui résulte en moins de rotations à faire, et que la courbe est aussi

exponentielle, même si cela est moins évident à voir.

On peut aussi voir sur ces graphes l'impact du paramètre  $\alpha$ . On constate qu'il induit une dégradation, qui est en particulier plus notable lorsque l'on suit une stratégie décroissante. Cette observation pratique sera justifiée dans le reste de la réflexion.

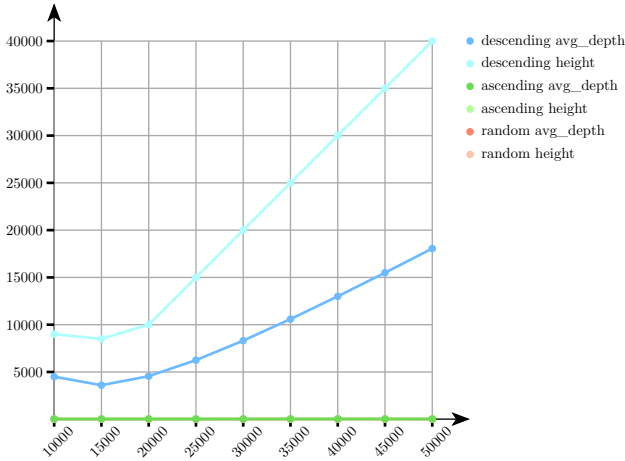


Fig. 8. – Hauteur & profondeur moyenne ABR

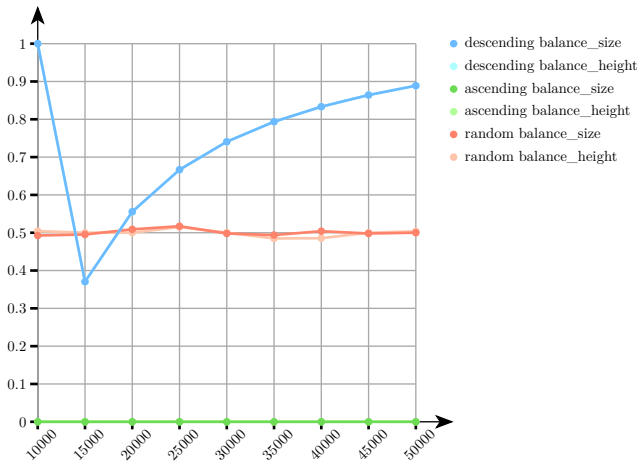


Fig. 9. – Facteurs d'équilibre ABR

On va voir dans l'exercice 6 que toutes les complexités dépendent de la profondeur moyenne. On peut sur ces graphes observer empiriquement son évolution. Pour les arbres binaires de recherches, on a peu de surprises compte tenu de nos précédentes observations. On a bien une explosion linéaire dans le cas de la stratégie décroissante et une profondeur moyenne qui vaut

la moitié de la hauteur. Cela fait du sens car on a essentiellement une liste chaînée en pratique. Sur le niveau de balance, on remarque qu'en pratique qu'en dehors du cas où on suit la stratégie aléatoire, on est loin du ratio 0.5 que l'on recherche avec ce type d'arbre.

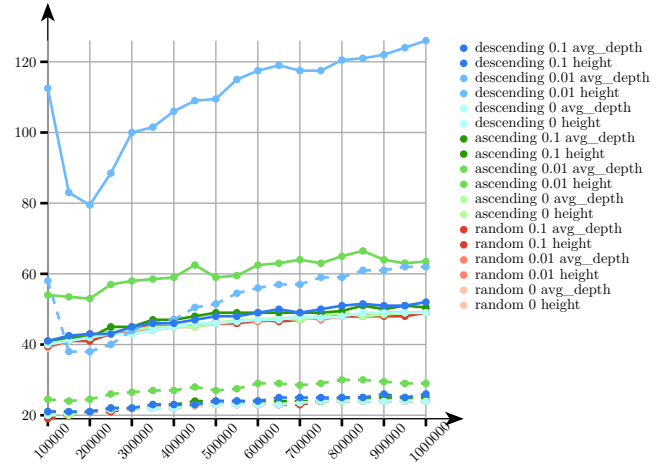


Fig. 10. – Hauteur et profondeur moyenne - Arbre cartésien

Sur les arbres cartésiens, on a plus de données. Commençons par la hauteur et la profondeur moyenne. On peut observer, comme on va le calculer dans l'exercice 6, qu'on suit bien une courbe de type  $O(\log n)$ . Cela est un point déterminant puisque cela permet d'avoir une complexité  $O(\log n)$  sur le reste de nos opérations fondamentales. Finalement, on pourrait observer que le paramètre  $\alpha$  joue sur la profondeur moyenne de l'arbre, avec les valeurs élevées menant à des arbres plus profonds, mais ce n'est pas un résultat que nous avons pu observer empiriquement de façon claire.

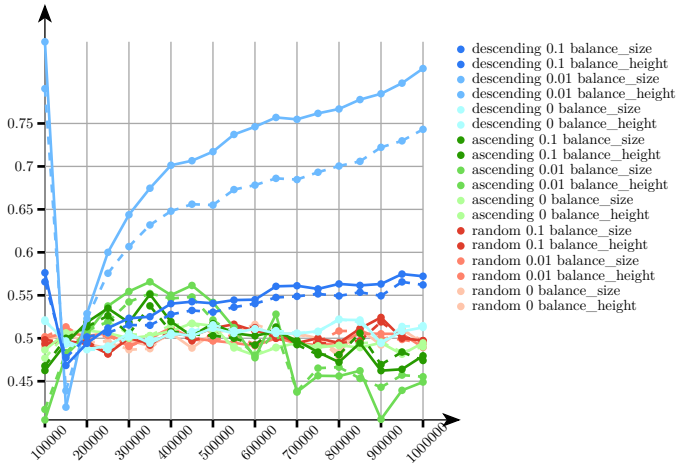


Fig. 11. – Facteurs d'équilibre - Arbre cartésien

Sur le second graphe, on observe le niveau d'équilibre de l'arbre et on peut constater que nos valeurs sont bien plus proches du 0.5 attendu. Cela confirme aussi notre réflexion sur la vitesse d'exécution de construction et destruction de l'arbre. Le choix des stratégies ne change pas la profondeur de l'arbre mesurée, mais seulement la vitesse d'insertion. On peut donc en conclure que notre hypothèse que l'ordre mène à moins de rotations est probablement correcte. Cela est tout de même notable pour une application en pratique de nos arbres cartésiens : si on a déjà tous les nœuds de l'arbre, trier ce vecteur initialement est en pratique bien plus rapide et ne résulte en aucune dégradation des propriétés de notre arbre, ce qui est une propriété intéressante. Une fois l'arbre construit, on peut continuer à insérer, chercher et supprimer des nœuds avec une complexité  $O(\log n)$ .

On remarque que notre structure d'arbre binaire de recherche peut être extrêmement déséquilibrée. En effet, dans la stratégie d'insertion par ordre croissant et par ordre décroissant, on note des arbres qui ne sont pas du tout équilibrés. Ceci est particulièrement problématique car cela rend des arbres binaires de recherche semblables à des listes chaînées, qui sont connues pour n'être en pratique pas très performantes. Ceci explique la performance extrêmement mauvaise de nos arbres binaires de recherche lorsque l'on suit ces stratégies.

Cela laisse la question de ce pourquoi les arbres cartésiens ne souffrent pas de ce problème. Le fait d'avoir une priorité aléatoire force des rotations. En effet, lorsqu'il y a une rotation, la rotation force un rééquilibrage sur le sous-arbre où la rotation a lieu. Comme les priorités sont aléatoires, alors le point de rééquilibrage est complètement choisi au hasard. Sur des grandes instances de l'arbre, on observe que ce processus atteint une moyenne et permet un rééquilibrage en moyenne de l'arbre. Finalement, comme les priorités sont choisies au hasard, alors on ne peut pas trouver de stratégies qui pourraient déséquilibrer ces arbres.

On a introduit le paramètre  $\alpha$  pour pouvoir observer précisément l'impact des collisions sur la performances dans ce type de stratégies de construction. On observe que lorsque  $\alpha$  monte, cela mène à plus de collisions et empiriquement, que les performances se dégradent. En effet, on se ramène petit à petit à un simple arbre binaire de recherche et on s'attend à la même dégradation des performances. On peut même envisager un cas théorique, où  $\alpha = 1$ , c'est à dire que toutes les priorités sont identiques. Dans ce cas, au moment de l'insertion, la propriété de tas est toujours respectée, donc on n'a jamais de rotation, donc on a exactement un arbre binaire de recherche, avec les mêmes problèmes.

## EXERCICE 6

a. Soit  $x_k$  un nœud de profondeur  $p_k$ . La profondeur de  $x_k$  est égale au nombre de nœuds ancêtres de  $x_k$ . Soit les nœuds  $x_i$  présents dans l'arbre. On a alors quatre cas de figure. Si  $x_i$  est un ancêtre de  $x_k$ , alors  $X_{ik} = 1$ . Si  $x_i = x_k$ , alors  $x_i$  n'est pas un ancêtre de  $x_k$ , donc  $X_{ik} = 0$ . Si  $x_i$  est un successeur de  $x_k$ , alors  $X_{ik} = 0$ , donc  $X_{ik} = 0$ . Finalement, si  $x_i$  et  $x_k$  ont un ancêtre commun  $x_j$ , alors  $x_i$  n'est pas un ancêtre de  $x_k$ , donc  $X_{ik} = 0$ . La variable aléatoire  $X_{ik}$  a la même fonction qu'une fonction indicatrice. Cela étant dit si calcule  $\sum_{i=1}^n X_{ik}$ , alors les seuls valeurs  $X_{ik}$  qui auront une valeur différente de 0 sont ces ancêtres, d'où  $p_k = \sum_{i=1}^n X_{ik}$ .



On obtient donc, par linéarité de l'espérance :

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

b. ( $\Rightarrow$ ) On sait que  $X_{ik} = 1$ , c'est à dire, par définition, que  $x_i$  est un ancêtre propre de  $x_k$ . On cherche à prouver que  $x_i$  a alors la plus petite priorité dans  $X(i, k)$ . Supposons qu'il ne l'est pas, alors il existe  $x_j$  avec, sans perte de généralité,  $i < j < k$  de plus petite priorité  $p(x_j)$  (on note  $p(s)$  la priorité du noeud  $s$ ).

Par la propriété du tas dans les arbres cartésiens, comme  $x_i$  est un ancêtre de  $x_k$ , on sait que  $p(x_i) < p(x_k)$ . Comme  $x_j$  est le noeud de plus petite priorité, alors il est la racine de l'arbre induit par la construction de l'arbre cartésien. De plus, on a  $i < j$ , donc la clé de  $x_i < x_j$ , donc par propriété de l'arbre binaire,  $x_i$  est dans le fils gauche de  $x_j$ . Par un raisonnement symétrique,  $x_k$  est dans le fils droit de  $x_j$ . On obtient donc que  $x_i$  est un ancêtre de  $x_k$  et que  $x_i$  et  $x_k$  sont dans deux sous-arbres différents, ce qui est une *contradiction*.

On a donc bien que  $x_i$  est le noeud qui a la plus petite priorité dans  $X(i, k)$ .

( $\Leftarrow$ ) On sait que  $x_i$  est le noeud qui a la plus petite priorité dans  $X(i, k)$ . Alors par construction de l'arbre cartésien, par priorité du tas, il en sera la racine. Il sera donc l'ancêtre de tous les noeuds du sous-arbre, en particulier de  $x_k$ . On aura donc bien, par définition,  $X_{ik} = 1$ .

c. Soit  $x_k$  un noeud de l'arbre. Soit  $p_k$  sa profondeur.

$$\begin{aligned} & \sum_{i=1}^n E(X_{ik}) \\ &= \sum_{i=1}^n (1 \times \mathbb{P}(X_{ik} = 1) + 0 \times \mathbb{P}(X_{ik} = 0)) \\ &= \sum_{i=1}^n \mathbb{P}(X_{ik} = 1) \end{aligned}$$

Comme prouvé à la question 2,  $X_{ik} = 1$  ssi  $x_i$  est le noeud de plus petite priorité dans  $X(i, k)$ . Si la

distribution des priorités est i.i.d, la probabilité que  $x_i$  ait la plus petite priorité est de :

$$\mathbb{P}(X_{ik} = 1) = \frac{1}{|X(i, k)|} = \frac{1}{|i - k| + 1}$$

Donc :

$$\begin{aligned} \sum_{i=1}^n \mathbb{P}(X_{ik} = 1) &= \sum_{i=1}^n \frac{1}{|i - k| + 1} \\ &= \sum_{i=1}^{k-1} \frac{1}{|i - k| + 1} + \frac{1}{|k - k| + 1} + \sum_{i=k+1}^n \frac{1}{|i - k| + 1} \\ &= \sum_{i=1}^{k-1} \frac{1}{k - i + 1} + \sum_{i=k+1}^n \frac{1}{i - k + 1} + 1 \\ &= \sum_{i=1}^{k-1} \frac{1}{k - i + 1} + \sum_{j=1}^{n-k} \frac{1}{j + 1} + 1 \quad (j = i - k) \\ &= \sum_{j=1}^{k-1} \frac{1}{j + 1} + \sum_{j=1}^{n-k} \frac{1}{j + 1} + 1 \quad (j = k - i) \end{aligned}$$

On remarque que :

$$\begin{aligned} \sum_{j=1}^{k-1} \frac{1}{j + 1} &\leq \sum_{j=1}^n \frac{1}{j + 1} \\ \sum_{j=1}^{n-k} \frac{1}{j + 1} &\leq \sum_{j=1}^n \frac{1}{j + 1} \end{aligned}$$

D'où :

$$\begin{aligned} & \sum_{i=1}^n \mathbb{P}(X_{ik} = 1) \\ &= \sum_{j=1}^{k-1} \frac{1}{j + 1} + \sum_{j=1}^{n-k} \frac{1}{j + 1} + 1 \\ &\leq \sum_{j=1}^n \frac{1}{j + 1} + \sum_{j=1}^n \frac{1}{j + 1} + 1 \\ &= 2 \sum_{j=1}^n \frac{1}{j + 1} + 1 \end{aligned}$$

De plus :

$$\begin{aligned}
& \sum_{j=1}^n \frac{1}{j+1} \\
&= \sum_{i=2}^{n+1} \frac{1}{i} \quad (i = j+1) \\
&\leq \sum_{i=2}^{n+1} \frac{1}{i} + \frac{1}{1} - \frac{1}{n+1} \quad \left(1 - \frac{1}{n+1} \geq 0\right) \\
&= \sum_{i=1}^n \frac{1}{i} = H_n
\end{aligned}$$

Donc :

$$\begin{aligned}
& \sum_{i=1}^n \mathbb{P}(X_{ik} = 1) \\
&\leq 2 \sum_{j=1}^n \frac{1}{j+1} + 1 \leq 2H_n + 1 \\
&\sim 2 \log n + 1 \in O(\log n)
\end{aligned}$$

De fait, la profondeur moyenne d'un nœud  $x_k$  dans l'arbre est dans l'ordre de grandeur de  $\log n$ , avec  $n$  le nombre de nœuds dans l'arbre.