Projet Complex
Arbres Cartésiens

Exercice 1

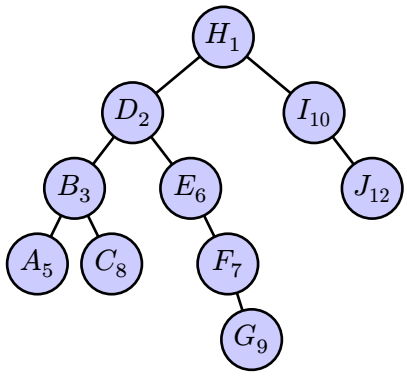

Fig. 1. – Arbre cartésien $\mathcal{A}$

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 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 triés selon l'ordre croissant de priorité. On emet 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 supérieure. 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 un 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 1 comparaison par échec (tant que l'on est pas encore au nœud) et 1 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 prédecesseur et successeur respectivement. On devra alors 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.

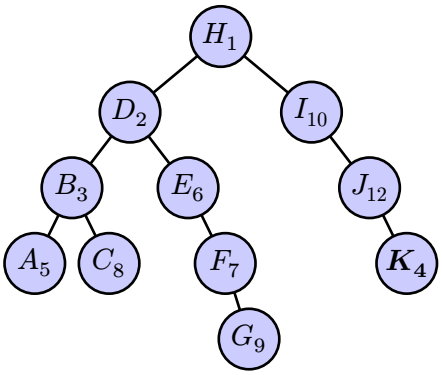

Fig. 2. – L'ajout de K_4 à $\mathcal{A}$ est faux

b. La complexité de l'ajout dans un arbre binaire est de $O(k)$ (2.b) et la complexité de la rotation est en temps constant. 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 un 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'a 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 pertube pas ces 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.

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 noeuds sauf pour le nœud supprimé. On va noter z_i avec deux fils gauche et droit z_j et z_k. On va s'interesser 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 piorité 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 somme sauf z. Il ne reste plus qu'a établir qu'on peut supprimer sans poser de soucis z lorsque c'est une feuille. Cela est du au fait que l'arbre est cartésien en tout sommet sauf z et que on a pas de fils gauche ou fils droit qui pourrait perturber la propriété d'arbre en retirant le nombre d'arête (tout arbre a $n-1$ arête).

Suite a 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 k inversions a 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 suppresion est en temps $O(k)$.

c. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

d. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat.

Exercice 5

a.

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat. Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim aeque doleamus animo, cum corpore dolemus, fieri tamen permagna accessio potest, si aliquod aeternum et infinitum impendere malum nobis opinemur. Quod idem licet transferre in voluptatem, ut postea variari voluptas distinguique possit, augeri amplificarique non possit. At etiam Athenis, ut e patre audiebam facete et urbane Stoicos irridente, statua est in quo a nobis philosophia defensa et collaudata est, cum id, quod maxime placeat, facere possimus, omnis voluptas assumenda est, omnis dolor repellendus. Temporibus autem quibusdam et.

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 présent 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 on 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. ($\Longrightarrow$) 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 généraité, $i < j < k$ de plus petite priorité $p(x_j)$ (on note $p(s)$ la priorité du noeud s).

Par la propriété du tas dans les arbres cartésien, 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 symmétrique, 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 nœud qui a la plus petite priorité dans $X(i,k)$.

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

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

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

Comme prouvé à la question 2, $X_{ik} = 1$ ssi x_i est le nœud de plus petite priorité dans $X(i,k)$. Si la distribution des priorités est i.i.d, la probabilitié 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 :

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

On remarque que :

$$\sum_{j=1}^{k-1} \frac{1}{j+1} \leq \sum_{j=1}^{n} \frac{1}{j+1}$$
$$\sum_{j=1}^{n-k} \frac{1}{j+1} \leq \sum_{j=1}^{n} \frac{1}{j+1}$$

D'où :

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

De plus :

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

D'où :

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

Du 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.