Université de Montréal

TP3 : Arbres Binaires de Recherche

Par

Catherine Laprise - C4411

Farzin Faridfar - 20031884

Baccalauréat en informatique

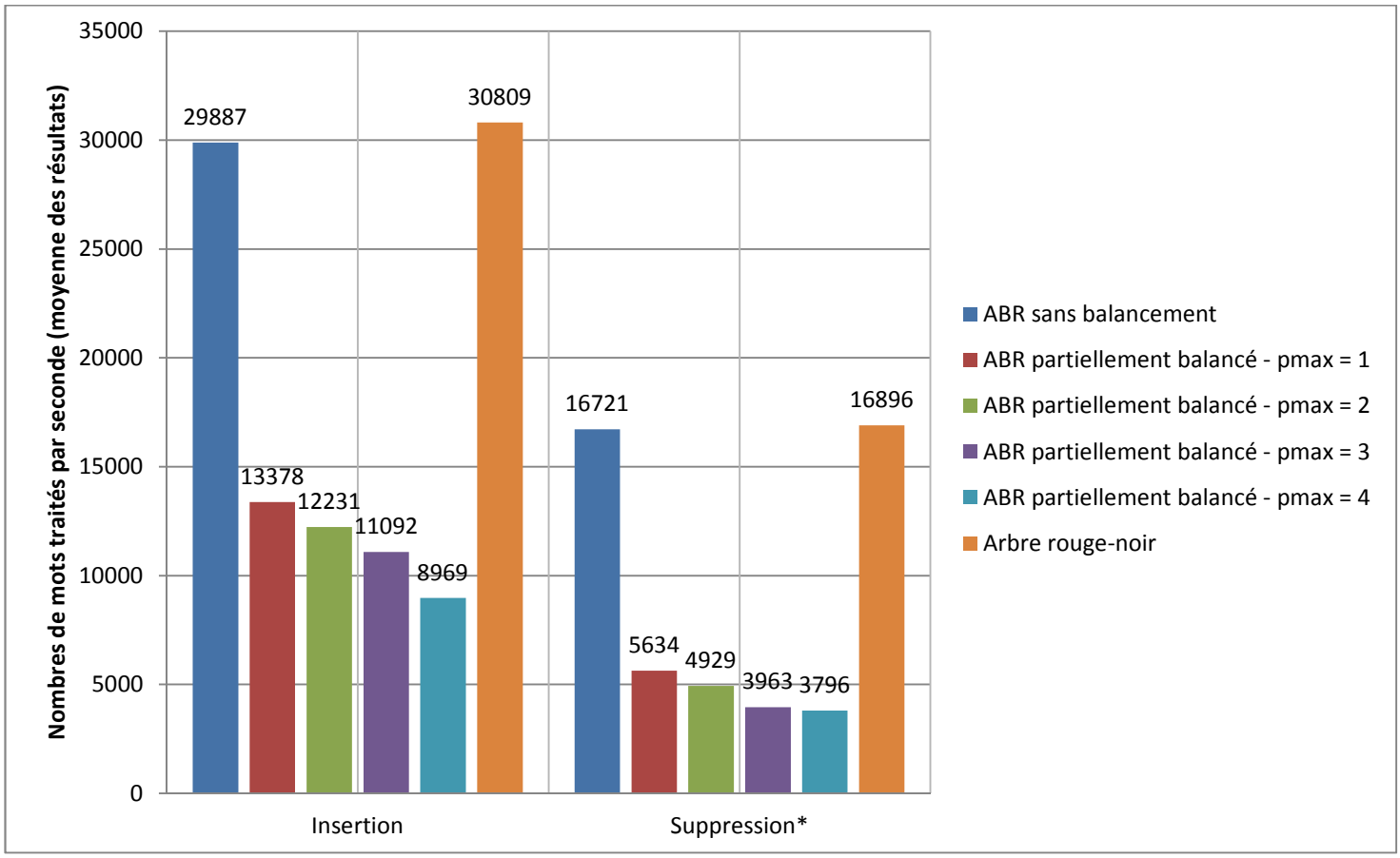
Faculté des Arts et des Sciences

Travail remis le 12 avril 2016

Dans le cadre du cours IFT2015

Structures de données

1. Graphique des résultats (moyennes de mots traités par seconde)



* Afin d'obtenir une mesure de comparaison uniforme pour les opérations de suppression, nous avons estimé le nombre d'opérations réalisées par seconde en divisant le nombre de mots traités par le temps pris pour réaliser les opérations. Le tableau ci-dessous illustre le total de mots traités, ainsi que le temps réel pris.

Résultats détaillés (nombre total de mots traités)

	Essai	ABR (aucun balancement)	ABR balancement partiel pmax = 1	ABR balancement partiel pmax = 2	ABR balancement partiel pmax = 3	ABR balancement partiel pmax = 4	Arbre rouge-noir
Insertion	Essai 1	29308	13229	12008	11238	9063	30818
	Essai 2	30335	13589	12361	10915	9079	31109
	Essai 3	30492	13530	12521	11301	8910	30986
Suppression*	Essai 1	2931 (0,18 sec)	1323 (0,24 sec)	1201 (0,23 sec)	1124 (0,28 sec)	907 (0,25 sec)	3082 (0,19 sec)
	Essai 2	3034 (0,18 sec)	1359 (0,24 sec)	1237 (0,25 sec)	1092 (0,29 sec)	908 (0,24 sec)	3111 (0,18 sec)
	Essai 3	3050 (0,18 sec)	1353 (0,23 sec)	1253 (0,28)	1131 (0,27 sec)	891 (0,23 sec)	3099 (0,18 sec)

2. Analyse théorique

Complexité des opérations

n : le nombre de nœuds
h : la hauteur de l'arbre

	Insertion	Suppression
ABR total (Aucun balancement)	$\mathcal{O}(n)$ dans le pire cas $\mathcal{O}(\log_2 n)$ attendu	$\mathcal{O}(n)$ dans le pire cas $\mathcal{O}(\log_2 n)$ attendu
ABR total (Balancement partiel)	$\mathcal{O}(\log_2 n)$ si $h \leq pmax$ Sinon : $\mathcal{O}\left(\frac{n}{2^{pmax}}\right) \in \mathcal{O}(n)$ dans le pire cas $\mathcal{O}(\log_2 n)$ attendu	$\mathcal{O}(\log_2 n)$ si $h \leq pmax$ Sinon : $\mathcal{O}\left(\frac{n}{pmax^2}\right) \in \mathcal{O}(n)$ dans le pire cas $\mathcal{O}(\log_2 n)$ attendu
Arbre Rouge-Noir (Balancement total)	$\mathcal{O}(\log_2 n)$ dans le pire cas	$\mathcal{O}(\log_2 n)$ dans le pire cas

3. Analyse numérique

Premièrement, en ce qui concerne l'ABR et l'arbre rouge-noir, on peut constater que comme prévu, l'arbre rouge-noir est plus efficace que l'ABR grâce à son balancement total qui garantit les insertions et suppressions en $\mathcal{O}(\log n)$. Toutefois, nos résultats démontrent que cette amélioration n'est pas si prononcée, et que l'arbre non balancé atteint un niveau de performance pratiquement identique. Cela peut s'expliquer par le fait que malgré que l'ABR offre une performance de $\mathcal{O}(n)$ en pire cas, ce cas dégénéré ne se produit que lorsque les données sont insérées dans l'arbre dans un ordre croissant ou décroissant. Étant donné que l'ordre des mots dans l'échantillon proposé est aléatoire, on se rapproche alors d'une complexité en $\mathcal{O}(\log n)$, ce qui résulte en un arbre quasi-balancé et dont la performance est semblable à celle d'un arbre rouge-noir.

Si nous observons les résultats calculés pour l'arbre partiellement balancé, on constate que le nombre d'insertions et suppressions faits par l'ABR sans balancement et l'arbre rouge noir est au moins deux fois plus de celui de l'ABR avec balancement partiel (jusqu'à cinq fois plus dans certains cas). Cette différence est remarquée peu importe la valeur de *pmax* (c'est-à-dire la profondeur maximum à laquelle les nœuds de l'arbre doivent être balancés). Effectivement, le nombre d'insertions et de suppressions par seconde décroît légèrement lorsque la valeur de *pmax* augmente – donc plus on effectue de rebalancements dans l'arbre, plus l'efficacité diminue. Nous supposons que ces résultats sont dûs au nombre d'opérations supplémentaires nécessaire pour maintenir les valeurs des sous-arbres à chaque nœud ainsi qu'aux opérations de rebalancement, qui s'effectuent également en $\mathcal{O}(\log n)$. Bien que la complexité croisse par un facteur constant, cette différence est assez significative pour être remarquée lors de l'analyse. Nous discutons le détail des opérations de balancement à la section suivante.

4. Discussion et conclusion

Bien que l'ABR ait été modifié afin d'inclure un rebalancement partiel à chaque 100 opérations, ce rebalancement semble occasionner une perte d'efficacité plutôt qu'une amélioration. Comme mentionné plus haut, dans le cas où l'échantillon de texte permet l'insertion de mots dans un ordre aléatoire, l'ABR offre déjà une complexité en $\mathcal{O}$ (log n) pour les insertions et suppressions. Toutefois, en ajoutant les opérations de rebalancement, bien que la complexité attendue ne change pas, le nombre d'opérations nécessaires à chaque étape augmente.

Dans un premier temps, lors d'une insertion ou suppression, il faut mettre à jour pour chaque nœud affecté la taille des sous-arbres gauches et droit. Une insertion ou suppression affecte tous les nœuds entre le nœud affecté et la racine – l'opération de mise à jour des tailles se fait donc en $\mathcal{O}$ (log n).

Lors du rebalancement, il faut tout d'abord localiser le nœud qui doit devenir la nouvelle racine. Ceci ne peut pas se faire en temps constant puisqu'il n'existe aucune méthode d'accès direct au nœud d'index [N/2]. Avec les méthodes utilisées dans le code proposé, la localisation se fait en $\mathcal{O}$ (log n) puisqu'il faut parcourir l'arbre à partir de la racine pour trouver ce nœud, qui en pire cas peut se trouver à une feuille de profondeur maximale. Une fois la nouvelle racine localisée, il faut effectuer la rotation de ce nœud jusqu'à la racine. Bien que les opérations de rotation se font en temps constant, il faut effectuer $\mathcal{O}$ (log n) rotations pour arriver jusqu'à la racine. Finalement, il faut remettre à jour une nouvelle fois toutes les valeurs de sous-arbres affectés, ce qui se fait aussi en $\mathcal{O}$ (log n) comme mentionné précédemment.

Plus la valeur de *pmax* augmente, plus les opérations doivent être effectuées fréquemment. En effet, si on considère le pire cas où tous les nœuds de profondeur ≤ *pmax* doivent être rebalancés, on doit effectuer un rebalancement sur $\sum_{i=0}^{pmax} 2^i = 2^{pmax+1} - 1$ noeuds à chaque rebalancement complet de l'arbre.

Au total, le nombre de rebalancements à effectuer est de $\frac{n}{100}\ (2^{pmax+1} - 1)\ \in \mathcal{O}(n)$, ce qui résulte en un temps amorti en $\mathcal{O}$ (log n) sur l'ensemble des opérations.

On ajoute donc une complexité de $\mathcal{O}$ (2 log n) à chaque opération effectuée.

À notre avis, ces opérations supplémentaires n'en valent pas le coût puisque, supposant un ordre d'insertion aléatoire, l'ABR offre déjà de bonnes performances s'approchant de l'arbre rouge-noir. De plus, le rebalancement partiel ne permet pas d'éviter le cas dégénéré où les éléments seraient insérés en ordre croissant ou décroissant. Bien que les *pmax* premiers niveaux soient assurément balancés, les opérations de recherche dans le pire cas s'effectueraient quand même en $\mathcal{O}\left(\frac{n}{pmax+1}\right) \in \mathcal{O}(n)$, une amélioration négligeable lorsque *pmax* est petit et n très grand.