IFT2015 hiver 2010 — Notes de cours

Miklós Csűrös

17 mars 2010

14 Équilibre d'un ABR

14.1 Hauteur

Toutes les opérations prennent $O(h)$ dans un arbre de hauteur h.

→ **Meilleur cas** Arbre binaire complet : $2^{h+1}-1$ nœuds dans un arbre de hauteur h, donc hauteur $h = \lceil \lg(n+1) \rceil - 1$ pour n nœuds est possible. Donc $h = \Omega(\log n)$

→ **Pire cas** Insertions consécutives de $1, 2, 3, 4, \ldots, n$ mènent à un arbre avec $h = n-1$. Donc $h = \Theta(n)$ au pire cas.

⇒ Est-ce qu'il est possible d'assurer que $h = O(\log n)$ en général ?

- ⋆ Réponse 1 [randomisation] : la hauteur est de $O(\log n)$ *en moyenne* (permutations aléatoires de $\{1, 2, \ldots, n\}$)
- ⋆ Réponse 2 [amortisation] : exécution des opérations en temps *amorti* $O(\log n)$ pour des arbres *splay*
- ⋆ Réponse 3 [optimisation] : la hauteur est de $O(\log n)$ *en pire cas* pour beaucoup de genres d'arbres de recherche équilibrés : arbre AVL, arbre rouge-noir, arbre 2-3-4. Insértion/délétion plus compliquées, mais toujours $O(\log n)$.

14.2 Performance moyenne

Théorème 14.1. *La hauteur d'un arbre de recherche construit en insérant les valeurs* $1, 2, \ldots, n$ *selon une permutation aléatoire est* $\alpha \lg n$ *en moyenne où* $\alpha \approx 2.99$.

La preuve du thèorème 14.1 est compliquée pour les buts de ce cours. On peut analyser le cas moyen en regardant la profondeur moyenne (ou niveau moyen) plutôt. Le coût de chaque opération dépend de la profondeur du nœud accédé dans l'arbre. On va démontrer que la profondeur moyenne est $O(\log n)$. Donc le temps moyen d'une recherche fructueuse est en $O(\log n)$. La preuve exploîte la correspondance à une exécution du tri rapide : le pivot du sous-tableau correspond à la racine du sous-arbre.

Définition 14.1. *Soit* x *un nœud [non-null] d'un ABR, et soit* T_x *le sous-arbre enraciné à* x. *Pour tout nœud* $y \in T_x$, *la distance* $d(x, y)$ *est définie comme la longueur du chemin de* x *à* y. *On définit* $d(x) = \sum_{y \in T_x} d(x, y)$ *comme la somme des profondeurs dans le sous-arbre.*

Avec cette définition, $d(\mathsf{racine}, y)$ est la profondeur (ou niveau) du nœud y et $\frac{d(\mathsf{racine})}{n}$ est la moyenne des profondeurs dans l'arbre.

Théorème 14.2. *Soit* $D(n) = \mathbb{E}d(\mathsf{racine})$ *l'espérance de la somme des profondeurs dans un arbre aléatoire avec* n *nœuds comme en Théorème 14.1. Alors,*

$$\frac{D(n)}{n} = O(\log n). \tag{14.1}$$

Démonstration. Si x n'a que des enfants null (feuille), alors $d(x) = 0$ et donc $D(1) = 0$. On définit aussi $D(0) = 0$ (correspondant à un arbre vide).

Si x est un nœud interne avec enfants u, v, alors

$$D(n) = \mathbb{E}d(\mathsf{racine}) = \underbrace{\big(1 + \mathbb{E}d(u)\big)}_{\text{profondeurs à la gauche}} + \underbrace{\big(1 + \mathbb{E}d(v)\big)}_{\text{profondeurs à la droite}} = \big(|T_x| - 1\big) + \mathbb{E}d(u) + \mathbb{E}d(v),$$

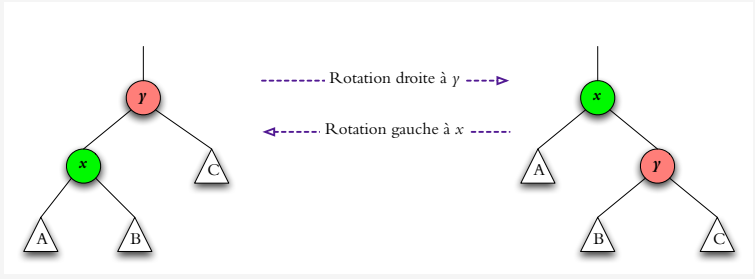
où $|T_x|$ dénote le nombre de nœuds dans le sous-arbre de x (incluant x). La racine peut être le 1^{er}, $2^{\text{ème}}$, $\ldots$, $n^{\text{ème}}$ élément avec la même probabilité $1/n$. Si la racine est le $(i+1)$-ème élément, alors $|T_u| = i$ et $|T_v| = n - 1 - |T_u| = n - 1 - i$. Donc,

$$\begin{aligned} D(n) &= n - 1 + \sum_{i=0}^{n-1} \frac{D(i) + D(n-1-i)}{n} \\ &= n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} D(i). \end{aligned}$$

C'est la même récurrence que pour le temps du calcul du tri rapide dans le cas moyen (Lemme 10.2). ■

14.3 Rotations

Arbres équilibrés : on maintient une condition qui assure que les sous-arbres ne sont trop différents à aucun nœud. Si l'on veut maintenir une condition d'équilibre, il faudra travailler un peu plus à chaque (ou quelques) opérations — mais on veut toujours maintenir $O(\log n)$ par opération.



La technique principale dans l'établissement de l'équilibre est la **rotation**. Les rotations (gauche ou droite) — préservent la propriété des arbres de recherche et prennent seulement $O(1)$.

14.4 Arbres *splay*

On utilise souvent des variables auxiliares pour maintenir l'équilibre de l'arbre p.e., arbre rouge et noir : au moins un bit (couleur). Arbre *splay* : aucune variable — mais temps $O(\log n)$ seulement comme coût amorti.

Idée principale : rotations sans tests spécifiques pour l'équilibre.Quand on accède à nœud x, on performe des rotations sur le chemin de la racine à x pour monter x à la racine.

Déploiement (*splaying*). Pour un nœud x : itérations successives avec rotations pour «lever» x jusqu'à ce que parent(x) devienne null (et donc x devient la racine de l'arbre).

```
    SPLAY(x) // déploiement du nœud x
S1  tandis que x n'est pas la racine faire
S2      si parent(x) = racine alors zig ou zag
S3      sinon
S4          si x et parent(x) au même coté (gauche-gauche ou droit-droit) alors zig-zig ou zag-zag
S5          sinon zig-zag ou zag-zig
```

Choix de x pour déploiement :

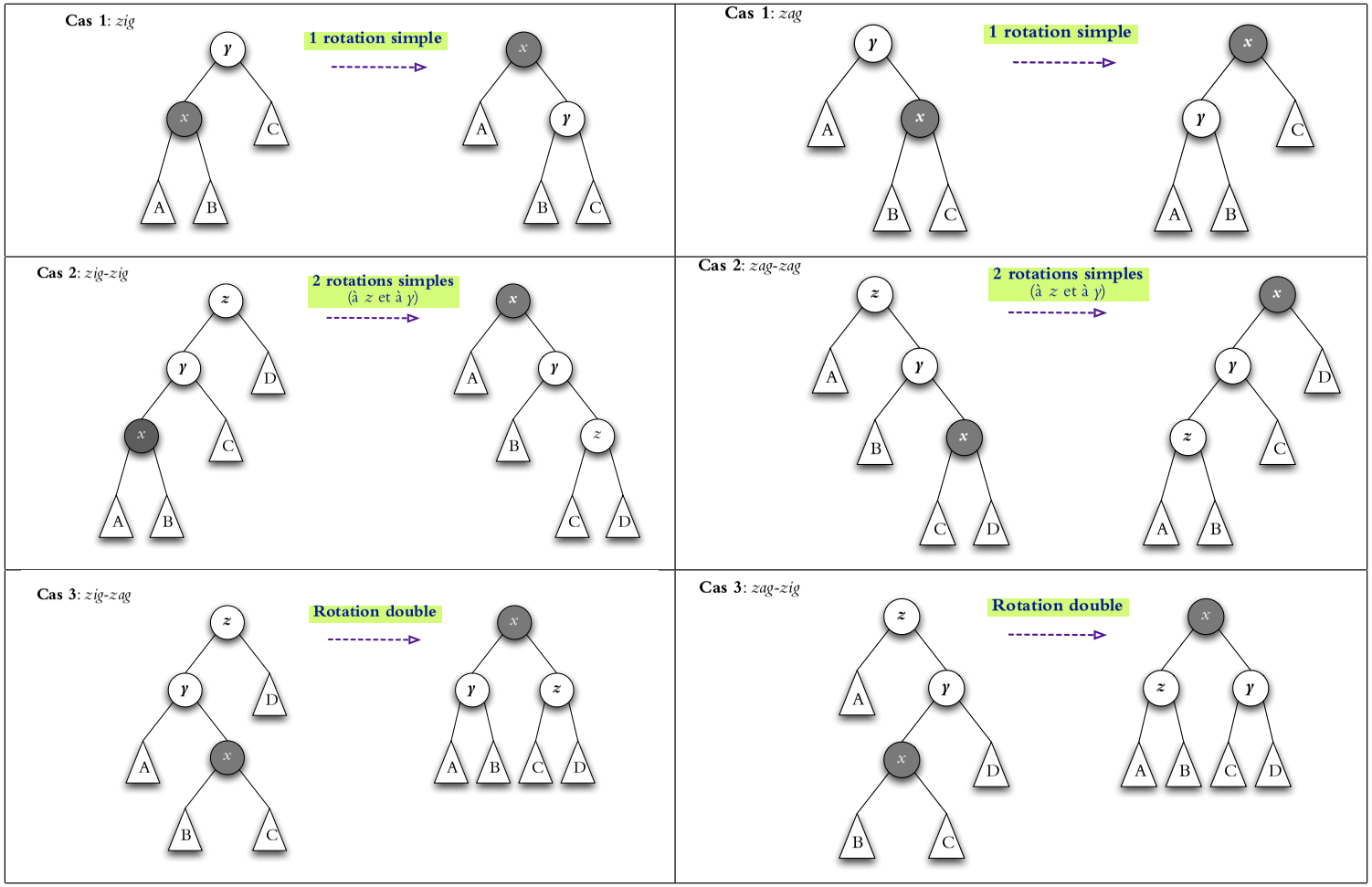
- insert : x est le nouveau nœud
- search : x est le nœud où on arrive à la fin de la recherche
- delete : x est le parent du nœud *effectivement* supprimé. Attention : c'est le parent ancien du successeur (ou prédecesseur) si on doit supprimer un nœud à deux enfants (logique : échange de nœuds, suivi par la suppression du nœud sans enfant).

Performance. Il s'agit d'une structure d'auto-ajustement (*self-adjusting*), comme dans le cas de compression de chemin en Union-Find. De nouveau, on a un temps de calcul efficace dans le sense amorti.

Théorème 14.3. *Le temps pour exécuter une série de m opérations avec* search, insert *et* delete *en commençant avec l'arbre vide est de $O(m \log n)$ où n est le nombre d'opérations d'*insert *dans la série.*

→ il peut arriver que l'exécution est très rapide au début et tout d'un coup une opération prend très long — ceci peut être problématique dans le contexte d'opération **online**.

→ cela ne fait aucune différence pour le temps de calcul d'un **algorithme** qui utilise la structure


Cas 1: zig
1 rotation simple
Cas 1: zag
1 rotation simple
Cas 2: zig-zig
2 rotations simples
(à z et à y)
Cas 2: zag-zag
2 rotations simples
(à z et à y)
Cas 3: zig-zag
Rotation double
Cas 3: zag-zig
Rotation double