20 Divertissement

REMARQUE. Les sujets de cette note sont hors-examen.

20.1 Déploiement (*splaying*) [Tarjan]

W(en)

Idée principale : on fait des rotations sans tests spécifiques pour l'équilibre de l'arbre binaire de recherche. Quand on accède à nœud x, on performe des rotations sur le chemin de la racine à x pour monter x à la racine. La position de x par rapport à son parent (gauche au droit), et celle du parent par rapport au grand-parent déterminent le genre de rotations à appliquer (v. illustration sur Page 2).

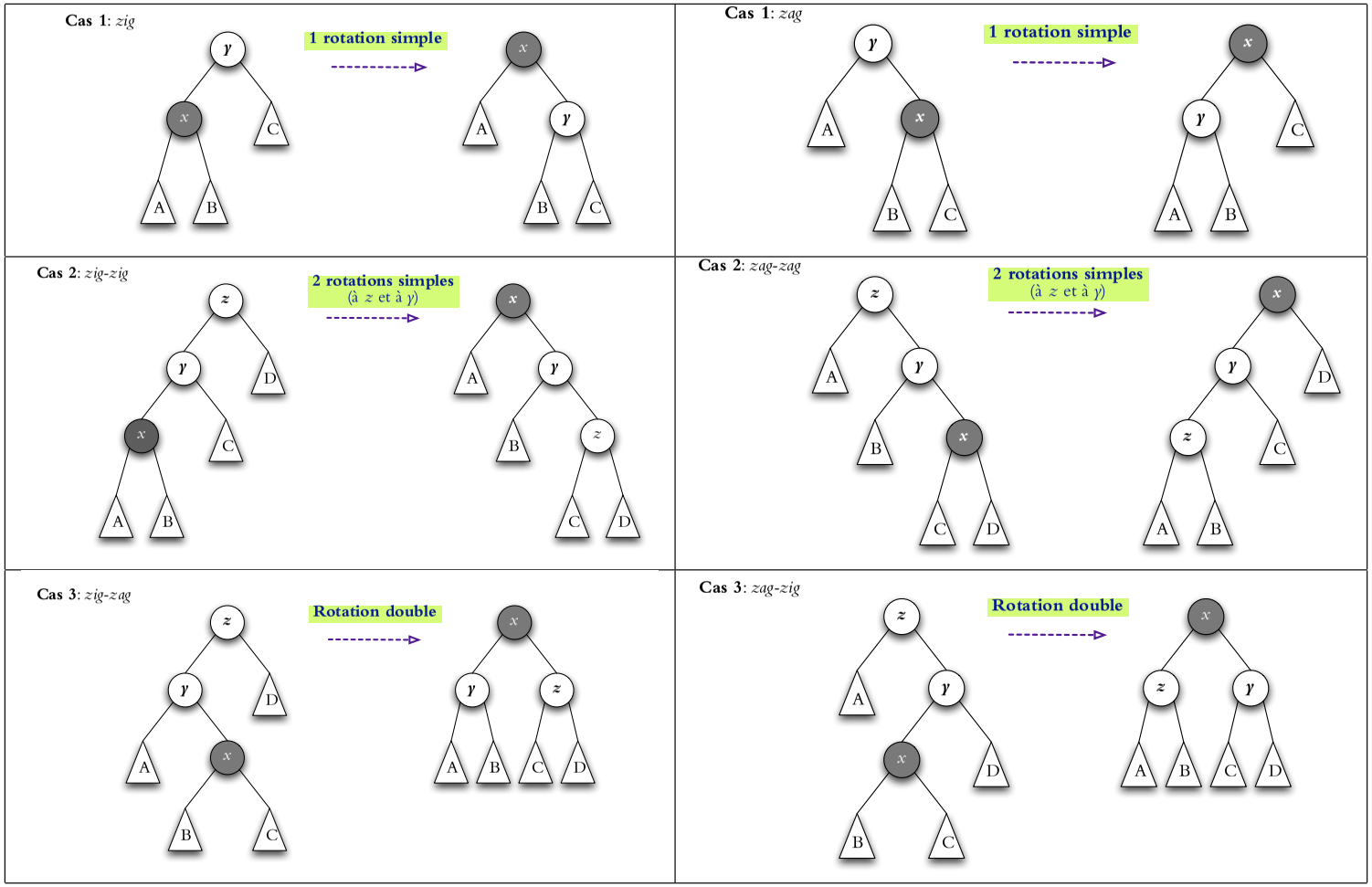
```
SPLAY(x) // déploiement du nœud x
S1  while x.parent ≠ null do // tant que x n'est pas la racine
S2      if x.parent = root then zig ou zag selon orientation
S3      else
S4          if x et x.parent au même coté (gauche-gauche ou droit-droit) then zig-zig ou zag-zag
S5          else zig-zag ou zag-zig
```

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 avec deux enfants internes (logique : échange de nœuds, suivi par la suppression du nœud sans enfant).

Théorème 20.1. *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 s'agit d'une structure d'auto-ajustement (*self-adjusting*). On a un temps de calcul efficace dans le sense amorti.
→ 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'opérations 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

20.2 Compteur $\log\log$ [Morris]

W(en)

On a besoin de $\lfloor \lg(n+1) \rfloor$ bits si on veut compter de 0 jusqu'à n. Est-ce qu'on peut compter avec $o(\log n)$ bits ? Oui, si on veut juste approximer le compte ! Il suffit de stocker $\lg n$ dans une variable X si on veut juste savoir la magnitude de n. Idéalement, on aura $X = 0$ pour $n = 1$, $X = 1$ pour $n = 2..3$, $X = 2$ pour $n = 4..7$ etc. Donc il faut attendre l'arrivée de 2^X éléments avant d'incrémenter X. La solution est d'incrémenter X avec probabilité $1/2^X$; on estime $n = f(X) = 2^X - 1$ à la fin.

```
1 COMPTEUR-INCREMENT()
2 faire X ← X + 1 avec probabilité 2^{-X}
```

Théorème 20.2. *L'estimateur $f(X)$ est non-biasé : après n appels à* COMPTEUR-INCREMENT*, on a que l'espérance $\mathbb{E}f(X) = n$.*

$\Rightarrow$ On peut compter jusqu'à N avec une variable X représentée sur $\lg\lg N$ bits.

20.3 Hachage pour compter [Flajolet]

Problème : comment compter le nombre d'éléments différents sur une grande liste ? Exemples : nombre de pages Web, nombre d'addresses IP différents dans un ensemble de paquets envoyés (diagnostique d'attaques sur l'internet).
Formellement : on a une grande liste $x_1, \ldots, x_n$ et on voudrait savoir le nombre d'éléments différents (**cardinalité**). On sait que $x_i \in U$ ou U est un ensemble fini (p.e., addresses IP). Solution possible : tableau booléen de taille $|U|$ — prend trop de mémoire
Idée de base : on utilise une bonne fonction de hachage h. La valeur $h(x_i)$ est un nombre binaire sur t bits. Chaque bit de $h(x_i)$ est 0 ou 1 avec des probabilités 50–50%. Nombre de bits 0 au début : au moins k avec probabilité 0.5^k. Soit $\rho(h(x))$ le nombre de 0s au début de $h(x)$. Si $x_i = x_j$, alors $h(x_i) = h(x_j)$. Supposons qu'on a m éléments différents parmi les n. Si on regarde $\rho(h(x_i))$, on trouvera en moyenne $m/2^k$ éléments différents avec $\rho(h(x_i)) \geq k$. $\Rightarrow$ le maximum $\max_{i=1,\ldots n} \rho(h(x_i))$ devrait être de l'ordre $\lg m$.

```
1 CARDINALITÉ(x_1, x_2 ...)
2 initialiser ρ_max ← 0
3 pour x_1, x_2, ..., faire ρ_max ← max{ρ_max, ρ(h(x_i))}
4 estimer m ← 2^{ρ_max} / φ.
```

La constante de la ligne 4 $\phi = e^{-\gamma}\sqrt{2} \approx 0.78$ vient de l'analyse précise de la distribution de $\rho_{\max}$. Si $\rho_{\max} \leq r$, on peut le stocker sur $\lg(r+1)$ bits $\Rightarrow$ avec $O(\log\log M)$ bits on peut estimer une cardinalité jusqu'à M.
Pour en lire davantage, consultez Durand et Flajolet "Loglog counting of large cardinalitites" [ESA 2003] : `http://algo.inria.fr/flajolet/Publications/DuFl03-LNCS.pdf`.