14 Arbres *splay*

14.1 Déploiement (*splaying*).

W(en)

Idée principale : on fait des 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. 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 :

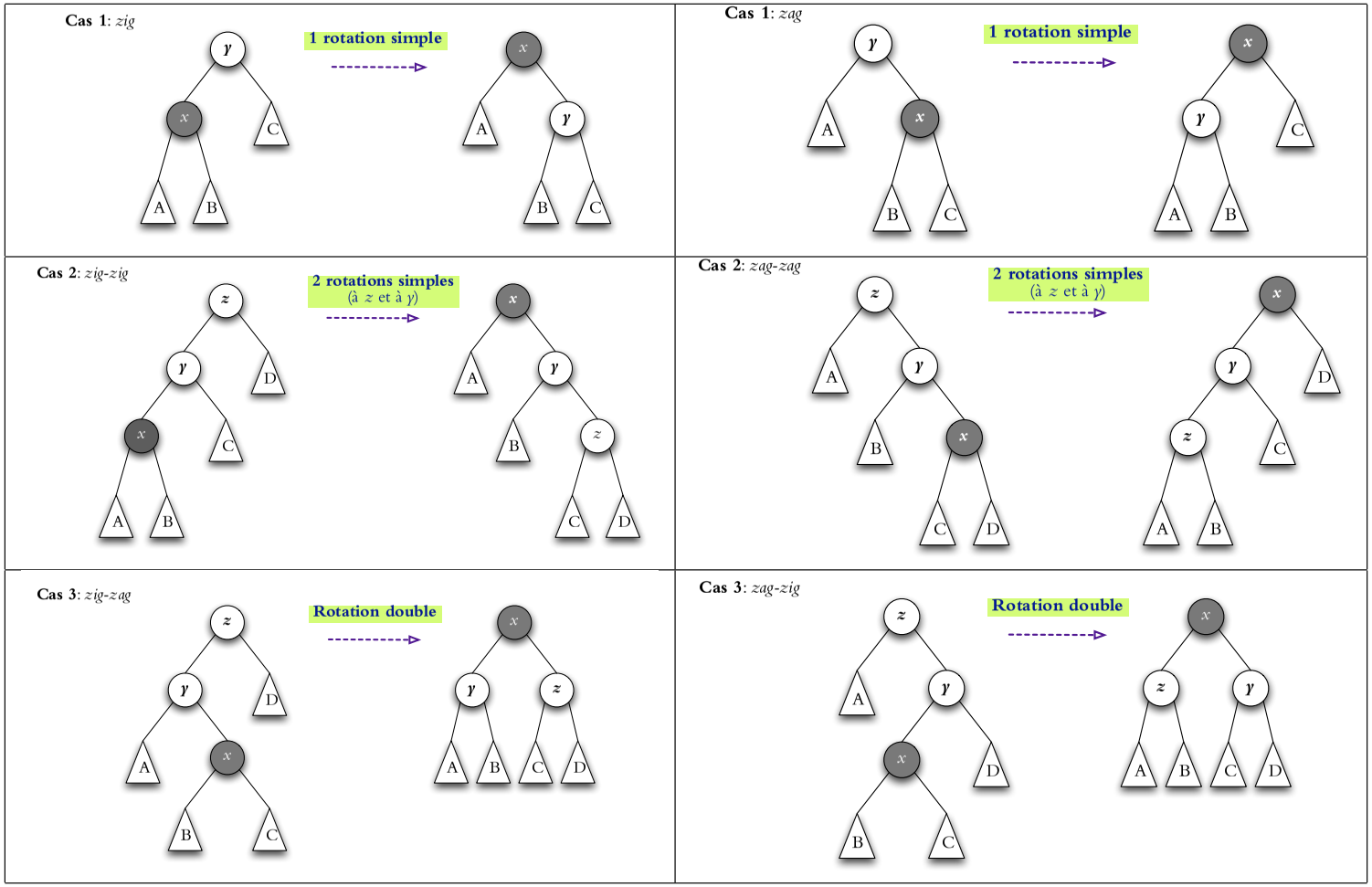
- ⋆ insert : x est le nouveau nœud
- ⋆ search : x est le nœud où on arrive à la fin de la recherche
- ⋆ delete : x est le parent du nœud *effectivement* supprimé. Attention : c'est le parent ancien du successeur (ou prédecesseur) si on doit supprimer un nœud avec deux enfants internes (logique : échange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

Théorème 14.1. *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