Structures de données (LU2IN006)

Cours 9 : Des alternatives aux AVL

Nawal Benabbou

Licence Informatique - Sorbonne Université

2022-2023

Type de données abstrait : liste triée

Définition

Une liste triée est un type de données abstrait défini par une suite finie d'éléments $(x_1, \ldots, x_n)$ telle que $x_i \leq x_{i+1}$ pour tout $i \in \{1, \ldots, n-1\}$. Opérations classiques : recherche, suppression, insertion, etc.

Une première implémentation possible

Comme pour une liste, une liste triée peut être codée par une liste (simplement/doublement) chaînée (cf cours 2). Dans ce cas, nous obtenons les complexités pire-cas suivantes :

- Recherche : opération en $O(n)$ car dans le pire des cas, on doit parcourir toute la liste chaînée pour trouver l'élément.
- Suppression : opération en $O(n)$ car on doit d'abord effectuer une recherche (en $O(n)$), puis recoller la liste en mettant à jour un pointeur.
- Insertion : opération en $O(n)$ car il ne s'agit pas d'insérer en tête, il faut garantir que la liste reste triée. Il faut donc rechercher sa place, et dans le pire des cas, on parcourt toute la liste chaînée.

Une deuxième implémentation possible

Une liste triée peut-être codée par un AVL. En effet, la liste triée se retrouve en faisant un parcours infixe de l'arbre. Les opérations sont toutes en $O(log(n))$.

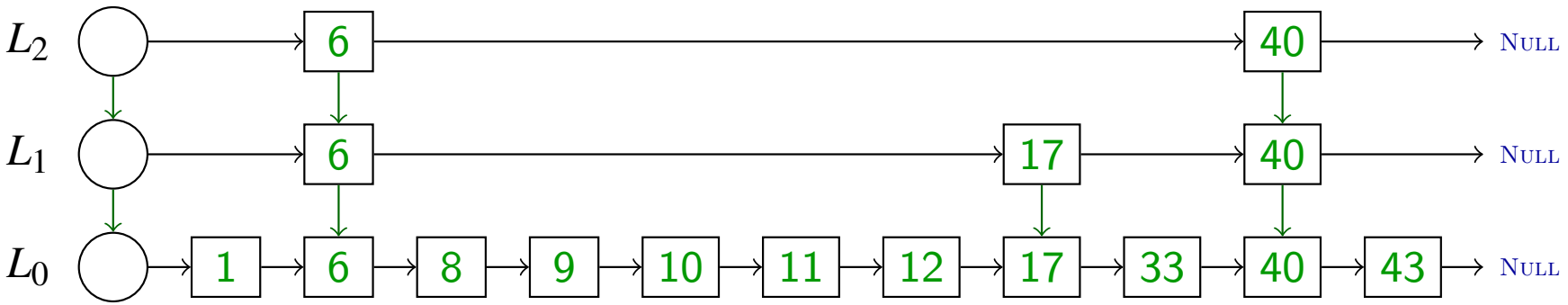
Implémentation d'une liste triée par une skip list

L'ajout/suppression d'un élément dans un AVL peut bouleverser complètement la structure suite aux rotations nécessaires à l'équilibrage. Ceci limite l'efficacité d'une programmation concurrente multi-thread, car cela nécessite de bloquer l'accès à un nombre importants de noeuds. Les skip list n'ont pas ce défaut.

Présentation

Une *skip list* (aussi appelée liste à enjambements ou liste à saut) est une structure de données probabiliste constituée de plusieurs couches $(L_0, L_1, L_2, \ldots)$, où chaque couche correspond à une liste chaînée. La couche L_0 correspond à la liste triée que l'on souhaite stocker. Les autres couches représentent des raccourcis, que l'on construit itérativement en respectant la règle suivante : un élément e de la couche L_i a une probabilité p fixée de faire partie de la couche L_{i+1}, et si c'est la cas, l'élément e de L_{i+1} pointe vers l'élément e de L_i.

Illustration :

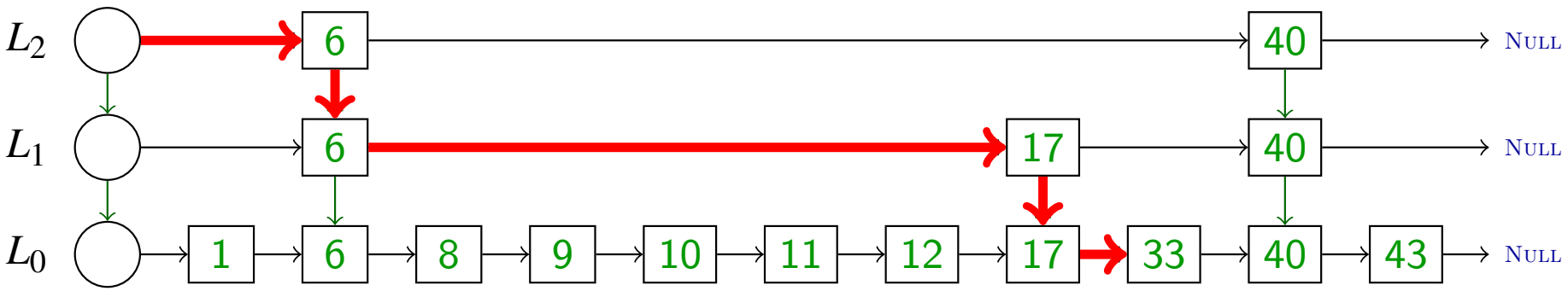


Recherche dans une skip list

Opération de recherche

Dans une skip list, on commence la recherche sur la couche de plus grand indice (la plus haute). On parcourt cette liste jusqu'à trouver un élément e inférieur ou égal à la valeur recherchée, et tel que son suivant est strictement plus grand que cette valeur (ou que son suivant n'existe pas). À ce moment là, on passe à la couche juste en dessous (en utilisant le pointeur de e), et on poursuit la recherche à partir de là (en procédant de la même façon). Quand on atteint la première couche, on effectue une recherche comme dans une liste chaînée triée classique.

Illustration (Recherche de la valeur 33) :



Implémentation en langage C

```c
typedef struct s_cell {
    int val;
    struct s_cell *next;
    struct s_cell *below;
} Cell;

typedef struct s_layer {
    int level;
    Cell *first;
    struct s_layer *above;
    struct s_layer *below;
} Layer;

typedef struct {
    float p;
    Layer* top;
    Layer* bottom;
} Skiplist;
```

Remarque :

Comme on ne connaît pas le nombre de couches a priori, on les stocke dans une liste (doublement) chaînée.

Recherche dans une skip list

```
1  Cell *skiplist_search(Skiplist* sl, int value){
2      Layer* layer = sl->top;
3      while (layer->first->val > value){
4          layer = layer->below;
5          if (layer == NULL)
6              return NULL;
7      }
8      Cell* c = layer->first;
9      while (c != NULL){
10         if (c->val == value)
11             break;
12         if (c->next == NULL || c->next->val > value){
13             c = c->below;
14         }else{
15             c = c->next;
16         }
17     }
18     if (c != NULL){
19         while (c->below != NULL){
20             c = c->below;
21         }
22         return c;
23     }else{
24         return NULL;
25     }
26 }
```

Opération de suppression et d'insertion

Suppression et insertion dans une skip list

- La suppression consiste à rechercher un élément, à le supprimer de la première couche (s'il a été trouvé), puis le supprimer des éventuelles couches supérieures qui le contiennent.
- L'insertion procède comme la recherche, jusqu'à arriver sur la première couche où on réalise une insertion dans une liste chaînée triée. Une fois l'élément ajouté à la première couche, on l'ajoute à la deuxième couche avec une probabilité p. S'il n'a pas été ajouté à cette couche, l'opération s'arrête. Sinon, on itère le processus sur la deuxième couche (créant de nouvelles couches si besoin).

Remarque : à propos de p

Quand $p = 0$, une skip list est équivalente à une liste chaînée triée (autrement dit, pas de raccourci). Plus p est proche de 1, plus il y a de couches (et donc plus de raccourcis, mais de moins en moins efficaces). La valeur de p permet de définir un compromis entre temps de calcul et espace mémoire utilisé (en pratique, on prend souvent $p = 1/2$ ou $p = 1/4$).

Complexité des opérations dans une skip list

Pour une structure probabiliste, le coût des opérations ne dépend pas que des données en entrée, mais aussi de la structure en elle-même, qui n'est pas définie de manière déterministe. Pour ces structures, la complexité pire-cas n'est pas très informative car le pire-cas peut avoir une très faible probabilité.

Complexité pire-cas en espérance

Pour une structure de données probabiliste, on peut s'intéresser à la complexité pire-cas en espérance, qui consiste à borner le coût moyen, calculé sur toutes les structures possibles. Le pire-cas en espérance correspond aux données en entrée pour lesquelles le coût moyen est le plus grand possible.

Pour les skip lists, le paramètre p induit une distribution de probabilité sur les skip lists possibles. Il s'agit d'en tenir compte lors de l'analyse de la complexité des opérations sur cette structure.

Complexité des opérations dans une skip list

Pour simplifier, on considère le cas où $p = 1/2$ et on note n le nombre d'éléments de la liste triée (c-à-d le nombre d'éléments de la première couche).

Complexité spatiale pire-cas en espérance

La complexité en espace est en $O(n)$. En effet, comme la probabilité qu'un élément fasse partie d'une couche L_i quelconque est $1/2^i$ alors la moyenne du nombre de cellules total est :

$$n(1 + 1/2 + 1/4 + 1/8 + \ldots) = 2n$$

Complexité temporelle pire-cas en espérance

Dans une skip list, la complexité des opérations d'insertion, de suppression et de recherche sont du même ordre. Leur complexité temporelle pire-cas en espérance est en $O(log(n))$. En effet on peut montrer que la hauteur moyenne de la structure est en $O(log(n))$ (en moyenne, on divise par deux le nombre d'éléments en passant d'une couche à une autre), et que le nombre moyen de comparaisons au total est aussi en $O(log(n))$.

En pratique, les skip lists sont utilisées comme alternatives aux AVL dans de nombreuses applications (systèmes distribués, files de priorités avec accès concurrents, etc).

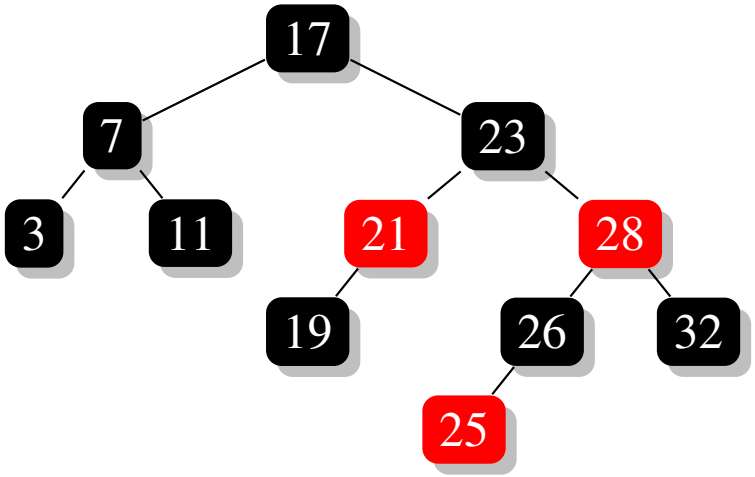
Les arbres rouge-noir

Les arbres rouge-noir forment une variante des AVL nécessitant moins de rotations en pratique.

Définition

Un arbre rouge-noir (ou bicolore) est un arbre binaire de recherche tel que :

- chaque nœud est de couleur noire ou rouge.
- la racine est toujours de couleur noire.
- les fils d'un nœud rouge sont toujours de couleur noire.
- si un nœud a moins de deux fils, on lui ajoute des fils fictifs noirs.
- le nombre de nœuds noirs sur un chemin de la racine vers une feuille est toujours le même (ce nombre est appelé *hauteur noire* et est noté h_N).



Quelques propriétés sur les arbres rouge-noir

Observation

Dans un arbre rouge-noir, le plus long chemin de la racine vers une feuille est au plus deux fois plus long que le plus petit chemin. De plus, la hauteur h est au plus égale à $2h_N$.

Preuve : comme tous les chemins possèdent exactement h_N nœuds noirs, le plus petit chemin possède au moins h_N nœuds. Par ailleurs, sur un chemin, on ne peut avoir deux nœuds rouges consécutifs. Par conséquent, le chemin le plus long possède au plus $2h_N$ nœuds (en alternant des noeuds rouges et noirs).

Proposition

La hauteur d'un arbre rouge-noir est en $O(log(n))$.

Preuve : Montrons par récurrence sur la hauteur noire qu'un arbre rouge-noir vérifie $n \geq 2^{h_N} - 1$. Si $h_N = 0$, alors on a forcément $n = 0$, et donc l'inégalité est trivialement vérifiée. Considérons un arbre rouge-noir de hauteur $h_N > 0$ et soit n son nombre de nœuds. Comme ses sous-arbres gauches et droits ont au moins une hauteur noire de $h_N - 1$, ils possèdent chacun au moins $2^{h_N-1} - 1$ nœuds par hypothèse de récurrence, et donc $n \geq 1 + 2 \times (2^{h_N-1} - 1) = 2^{h_N} - 1$. Ainsi, dans un arbre rouge-noir, on a toujours $n \geq 2^{h_N} - 1$, c-à-d $h_N \leq log_2(n+1)$. Comme on a déjà prouvé que $h \leq 2h_N$, on obtient $h \leq 2log_2(n+1)$.

Opérations dans un arbre rouge-noir et implémentation

Recherche dans un arbre rouge-noir

Dans un arbre rouge-noir, la recherche d'un élément se fait exactement comme dans une arbre binaire de recherche. La complexité pire-cas de cette opération est donc en $O(h)$ où h est la hauteur de l'arbre (c'est donc en $O(log(n))$).

En revanche, les opérations d'insertion et de suppression sont différentes, et nécessite de rajouter un champs `pere` dans la structure (en plus de la couleur).

```
1  #define NOIR 0
2  #define ROUGE 1
3
4  typedef struct arn {
5      int val;
6      int couleur;
7      struct noeud *fg;
8      struct noeud *fd;
9      struct noeud *pere;
10 } ARNtree;
```

Insertion dans un arbre rouge-noir

Principe général

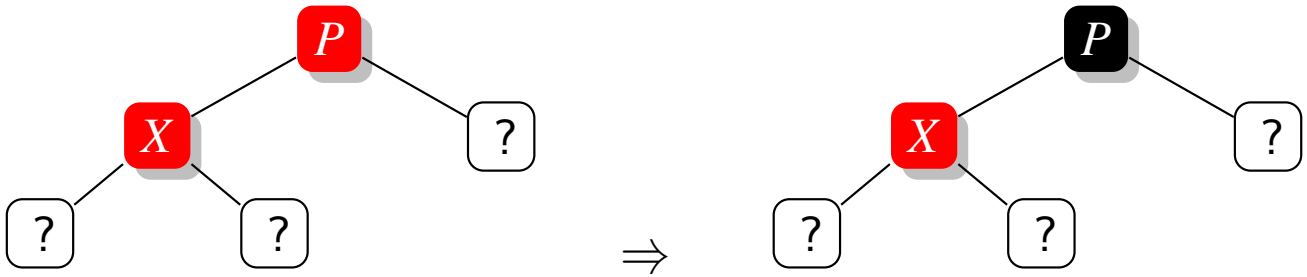
Comme un arbre rouge-noir est un arbre binaire de recherche, on commence par insérer l'élément à sa place, sans tenir compte des couleurs. Puis on colorie le nouveau noeud en rouge pour ne pas changer le nombre de noeuds noirs sur un chemin. Ensuite, de manière récursive, on vérifie que les propriétés des arbres rouge-noir sont vérifiées en remontant vers la racine. Si ce n'est pas le cas, on modifie la couleur des ancêtres et/ou on réalise des rotations.

Soit X le nouveau noeud créé de couleur rouge. Son insertion ne pose aucune difficulté, sauf quand son père est rouge (les fils d'un noeud rouge doivent être noirs). Pour restaurer les propriétés de l'arbre rouge-noir après l'insertion de X, on utilise un algorithme récursif (`restaure_abr`) qui procède comme suit :

Cas 1 : Si X est la racine de l'arbre, colorier X en noir et terminer.

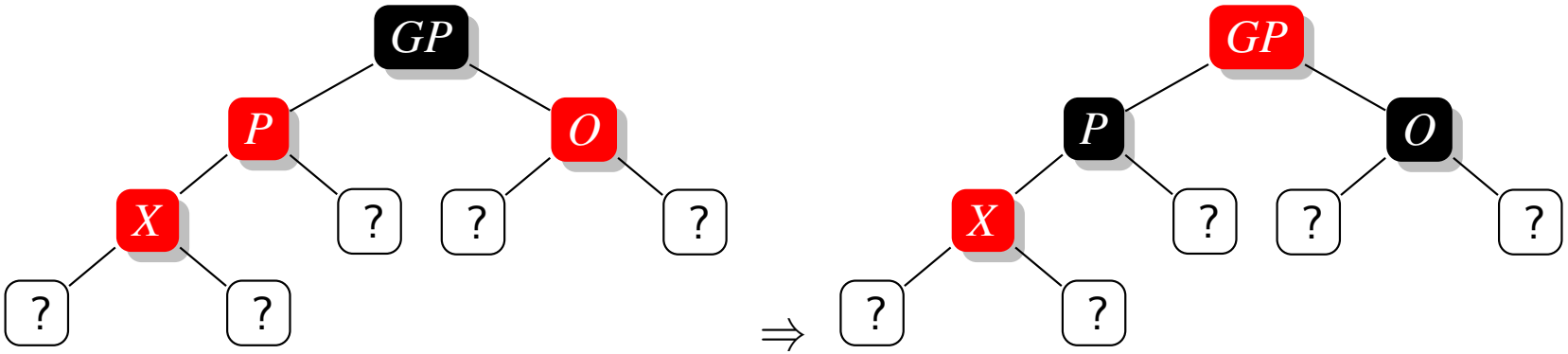
Cas 2 : Si père(X) est noir, terminer (rien à restaurer).

Cas 3 : Si père(X) est la racine et est rouge, colorier père(X) en noir, terminer.



Insertion dans un arbre rouge-noir

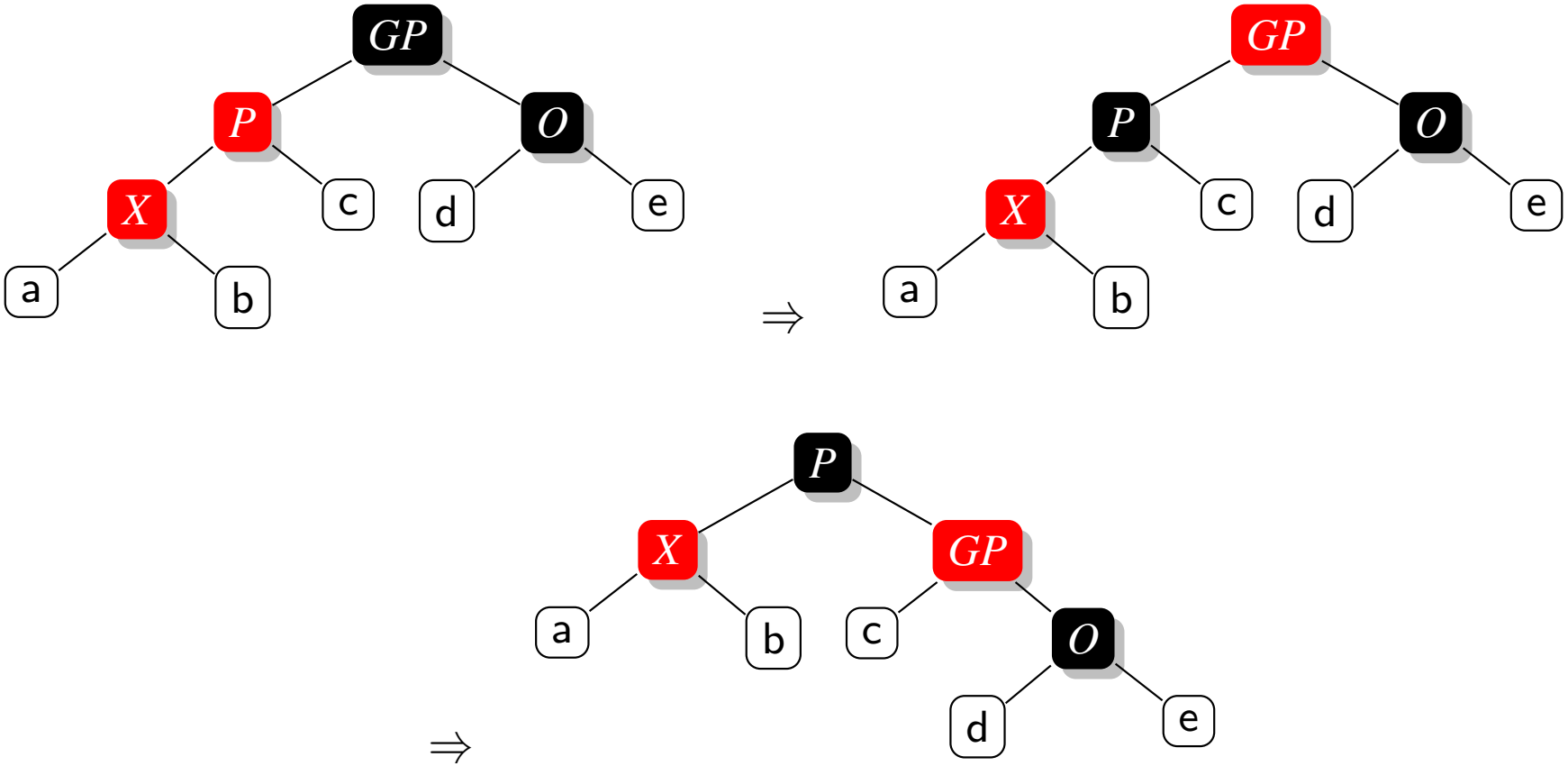
Cas 4 : Si père(X) est rouge et que X possède un oncle de couleur rouge, colorier père(X) et oncle(X) en noir, colorier grand-père(X) en rouge, puis appeler `restaurer_abr(grand-père(`X`))` et terminer.



Cas 5 : Si père(X) n'est pas une racine, père(X) est rouge et X ne possède pas d'oncle de couleur rouge, plusieurs sous-cas sont à considérer, induisant des rotations différentes.

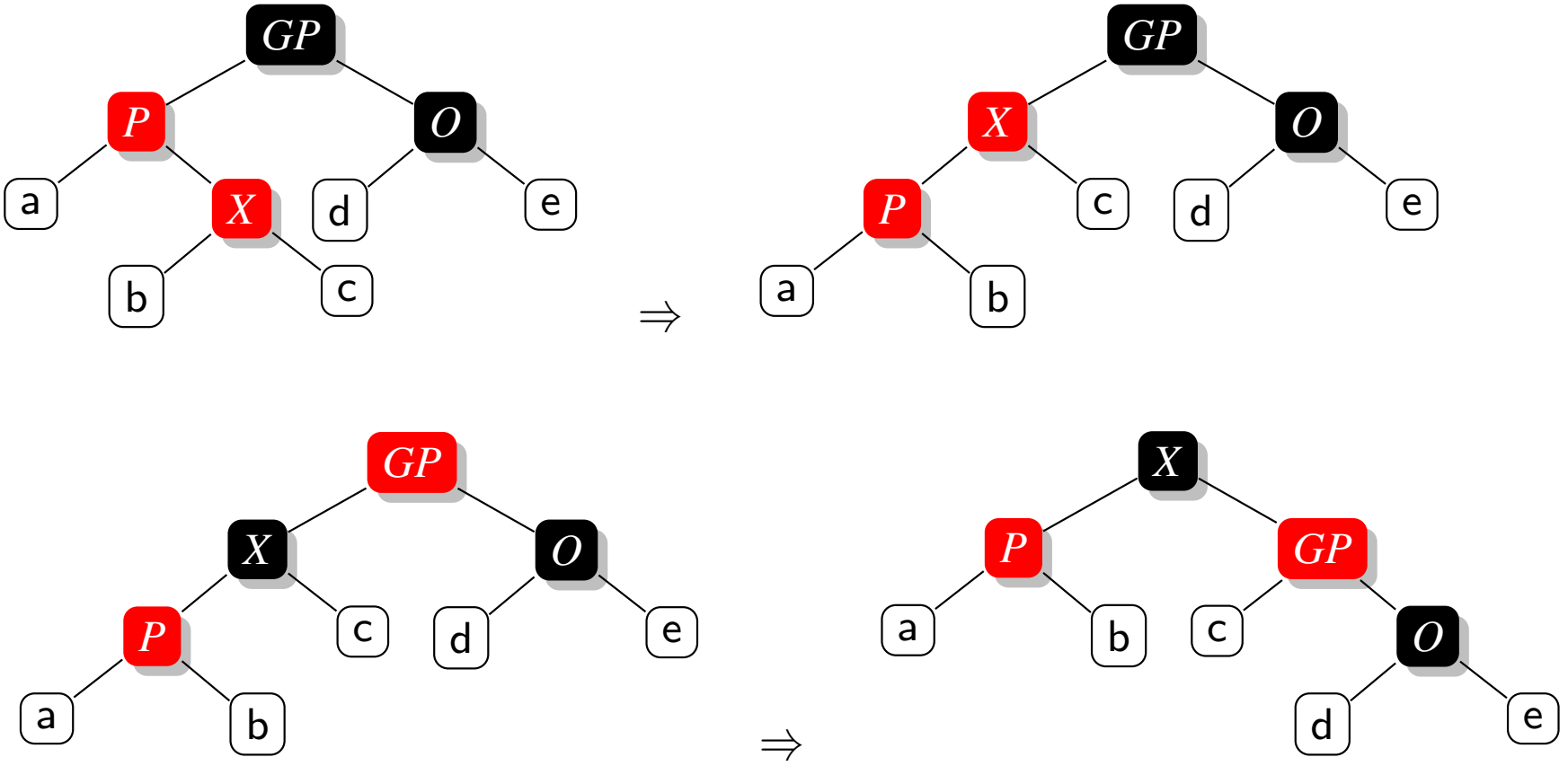
Insertion dans un arbre rouge-noir

Cas 5a : Si X est un fils gauche et que père(X) est aussi un fils gauche, colorier père(X) en noir, colorier grand-père(X) en rouge, faire une rotation droite de grand-père(X), et terminer.



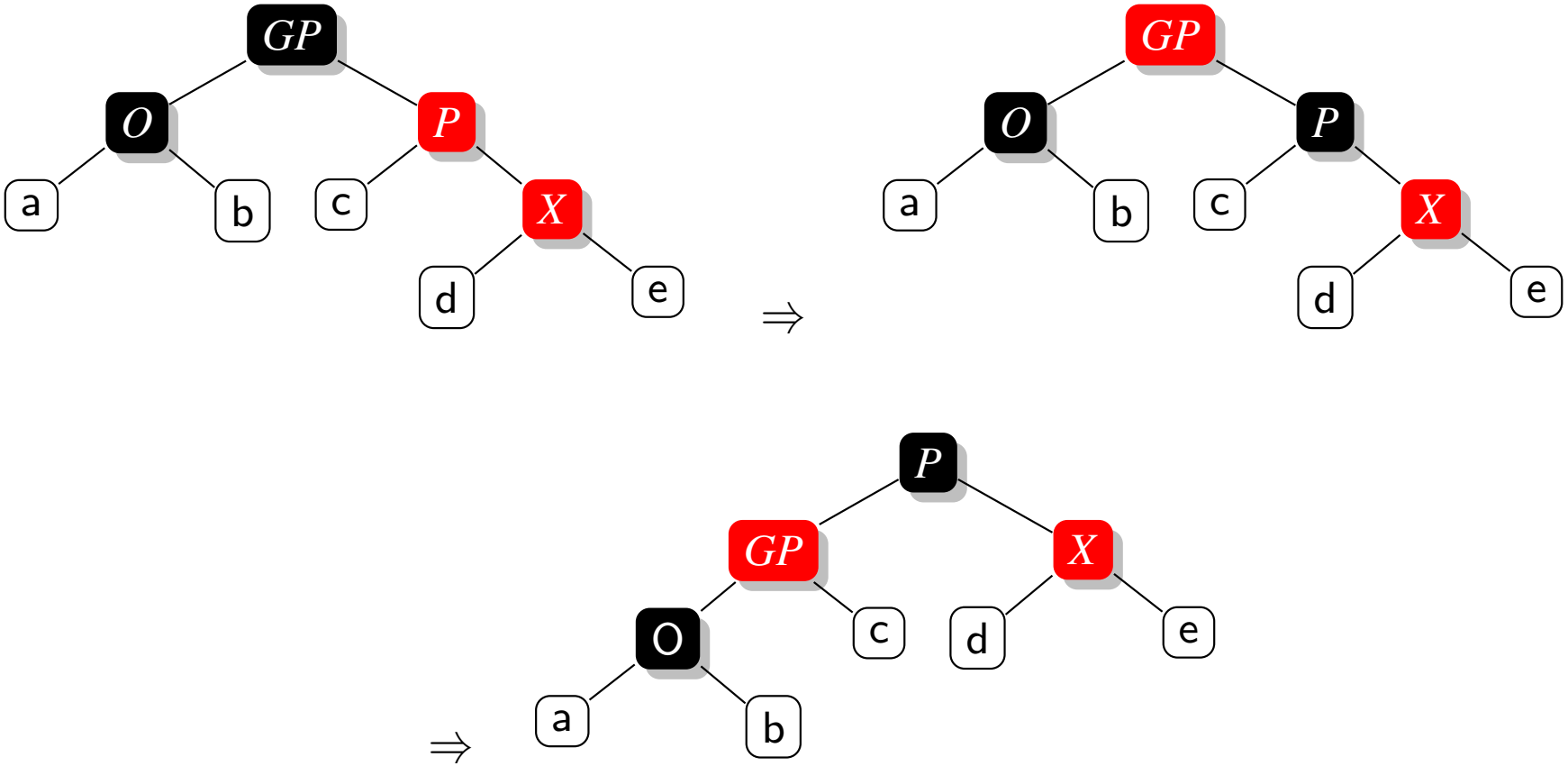
Insertion dans un arbre rouge-noir

Cas 5b : Si X est un fils droit et que père(X) est un fils gauche, faire une rotation gauche de père(X) pour se ramener au cas précédent (avec inversion de X et son père).



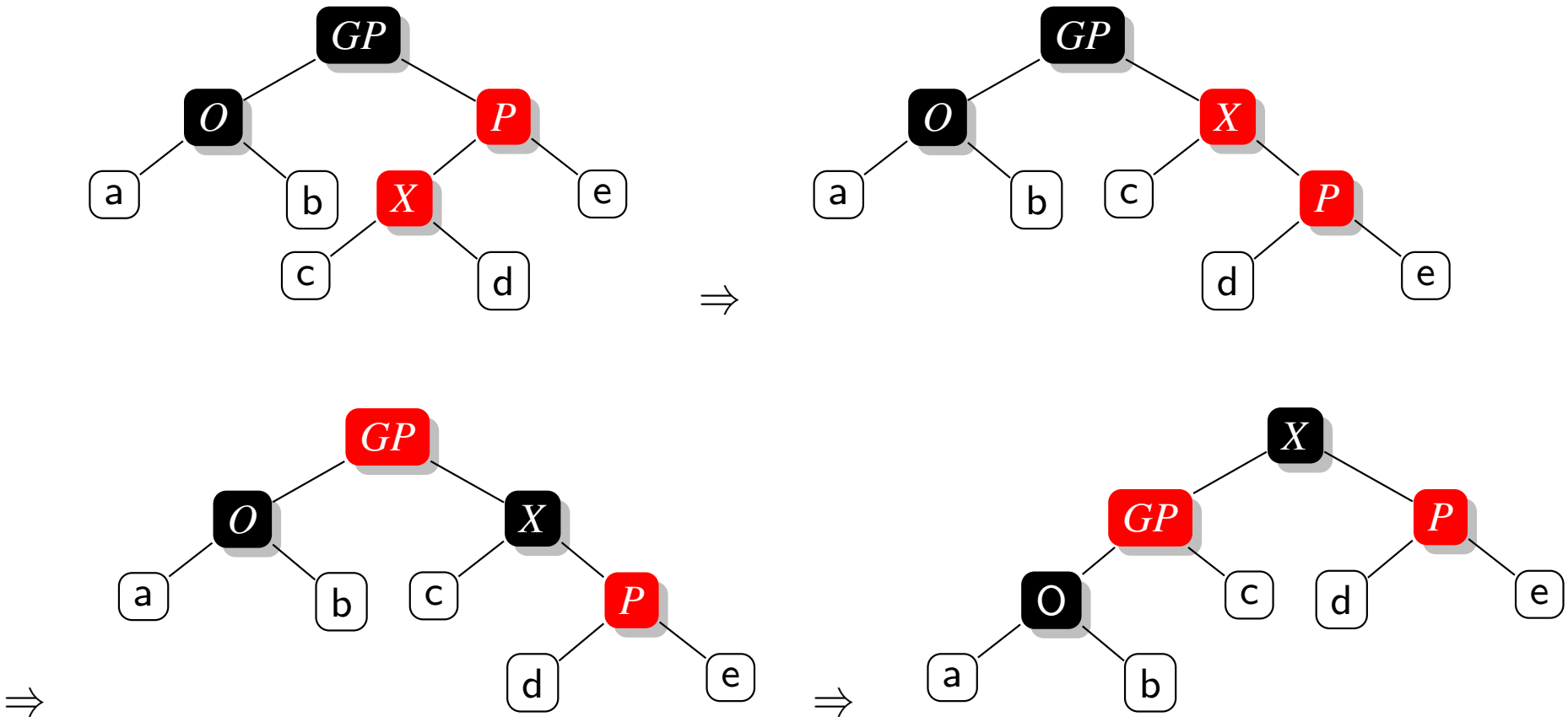
Insertion dans un arbre rouge-noir

Cas 5c : Si Si X est un fils droit et que père(X) est aussi un fils droit, colorier père(X) en noir, colorier grand-père(X) en rouge, faire une rotation gauche de grand-père(X), et terminer.



Insertion dans un arbre rouge-noir

Cas 5d : Si X est un fils gauche et que père(X) est un fils droit, faire une rotation droite de père(X) pour se ramener au cas précédent (avec inversion de X et son père).



Insertion dans un arbre rouge-noir

Remarques :

- On réalise un appel récursif uniquement dans le cas 4. Dans ce cas, l'appel récursif se fait sur un ancêtre (le grand-père). Par conséquent, dans le pire des cas, on devra remonter jusqu'à la racine de l'arbre avant de terminer.
- On effectue des rotations uniquement dans le cas 5. Par conséquent, on doit faire au plus deux rotations par insertion.
- Pour chaque cas, on doit changer la couleur d'au plus deux noeuds. Le nombre total de changements de couleur est donc au plus deux fois la hauteur de l'arbre.

Complexité pire-cas

L'insertion dans un arbre-rouge noir est donc en $O(log(n))$. De manière similaire, on peut montrer que l'opération de suppression est aussi en $O(log(n))$.

Les arbres rouge-noir sont utilisés dans de nombreuses applications, comme par exemple l'ordonnanceur du noyau Linux (Completely Fair Scheduler).