Module EA4 – Éléments d'Algorithmique II

Outils pour l'analyse des algorithmes

Dominique Poulalhon
dominique.poulalhon@irif.fr

Université Paris Diderot
L2 Informatique & Math-Info
Année universitaire 2019-2020

Arbres Binaires de Recherche

IV. Opérations de modification

Point d'étape

Nous avons vu en Partie III comment effectuer les opérations de recherche (d'un élément, du minimum ou du maximum, de l'élément suivant) dans un ABR. Nous avons vu que ces opérations, sans être aussi efficaces que dans un tableau trié, sont tout de même « raisonnablement efficaces » si l'arbre est « relativement équilibré » : leur complexité est proportionnelle à la ***hauteur*** de l'arbre, et non à sa taille n, et cette hauteur est seulement en $\Theta(\log n)$ ***dans les bons cas*** (mais elle peut également être en $\Theta(n)$ ***dans les mauvais cas***, qui ressemblent fort à des listes chaînées)

En Partie I, nous avons motivé l'introduction des ABR par la nécessité de trouver une structure de données pour représenter efficacement les ensembles *dynamiques* : pour les ensembles *statiques*, les tableaux triés sont parfaits ; mais ils deviennent terriblement inefficaces dès que des opérations d'insertion ou de suppression doivent être effectuées, puisqu'elles en un coût en $\Theta(n)$ (en moyenne).

La question naturelle est donc :

> *Peut-on modifier un ABR en temps « raisonnable », par exemple* $\Theta(h)$ *comme les opérations de recherche ?*

Insertion dans un ABR

Plusieurs stratégies peuvent être envisagées pour ajouter un élément `x` à un ABR ; les deux principales étant :

1. l'insertion à la racine, à la manière d'une insertion en tête de liste chaînée
2. l'insertion aux feuilles, à la manière d'une insertion en queue de liste chaînée

Insertion à la racine : c'est peut-être le plus naturel, puisque la racine est le « point d'entrée » de l'arbre, comme la tête d'une liste chaînée [a] ; il s'agirait de créer une nouvelle racine contenant `x`, puis de reconstruire un sous-arbre gauche contenant tous les éléments plus petits que `x`, et un sous-arbre droit contenant tous les éléments plus grands que `x`.

Cela soulève deux problèmes :

- comment partitionner les éléments d'un ABR selon un « pivot » `x` ?
- comment reconstruire deux ABR avec, respectivement, les petits et les grands éléments ?

Non seulement ce n'est pas franchement évident (essayez à la main sur de petits exemples, vous verrez vite), mais en plus, il faudrait faire cela *efficacement* – c'est-à-dire sans parcourir tout l'arbre.

Moralité : essayons autrement !

---

a. et que l'insertion en tête de liste est plus simple que l'insertion en queue… n'est-ce-pas ? Si ce n'est pas clair, revoir les cours de L1 !

Insertion dans un ABR

Plusieurs stratégies peuvent être envisagées pour ajouter un élément `x` à un ABR ; les deux principales étant :

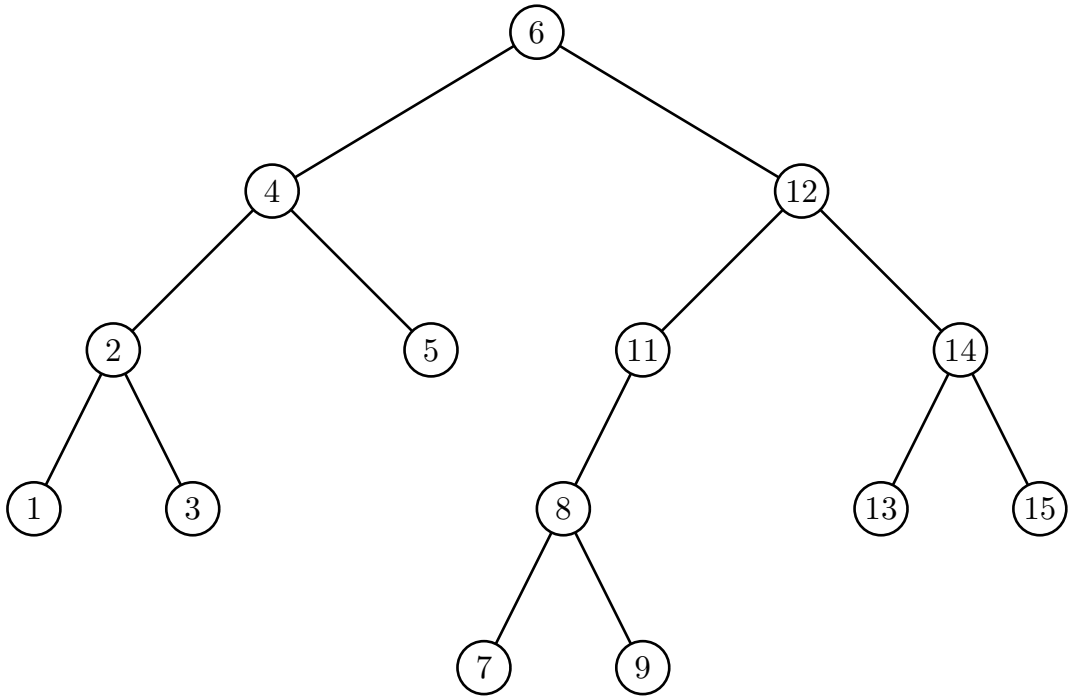
1. l'insertion à la racine, à la manière d'une insertion en tête de liste chaînée
2. l'insertion aux feuilles, à la manière d'une insertion en queue de liste chaînée

Insertion aux feuilles : il s'agit cette fois de créer une feuille à un emplacement libre, c'est-à-dire de l'attacher à un nœud non complet (n'ayant qu'un ou zéro fils) ; une différence majeure aves les listes chaînées réside évidemment dans la non-unicité de ces emplacements libres... une liste chaînée n'a que deux extrémités, un arbre binaire en a plein. C'est à la fois une difficulté, puisqu'il faut trouver la bonne feuille, et un atout, puisque cela donne de la souplesse à la structure : insérer une feuille revient, en terme de liste, à insérer un élément à n'importe quelle position, et cela ***sans effectuer de réorganisation majeure de la structure***, ce qu'on ne sait justement pas faire dans un tableau trié.

Une bonne raison supplémentaire pour réaliser l'insertion aux feuilles est, pour mémoire, que nous souhaitons éviter les doublons. Donc toute insertion doit être précédée d'une recherche. Si cette recherche réussit, `x` ne doit pas être inséré ; et si elle échoue... nous avons trouvé l'***unique emplacement*** où `x` aurait pu se trouver, c'est-à-dire l'emplacement où il faut créer une nouvelle feuille pour l'accueillir.

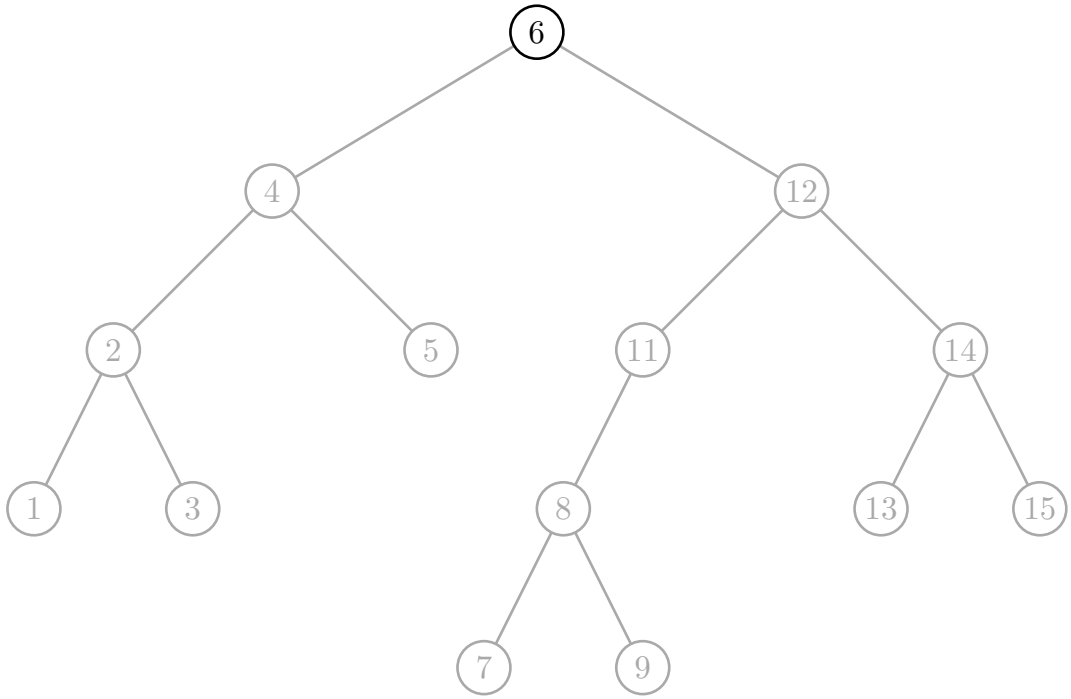
Insertion dans un ABR

Exemple – insertion de 10 :



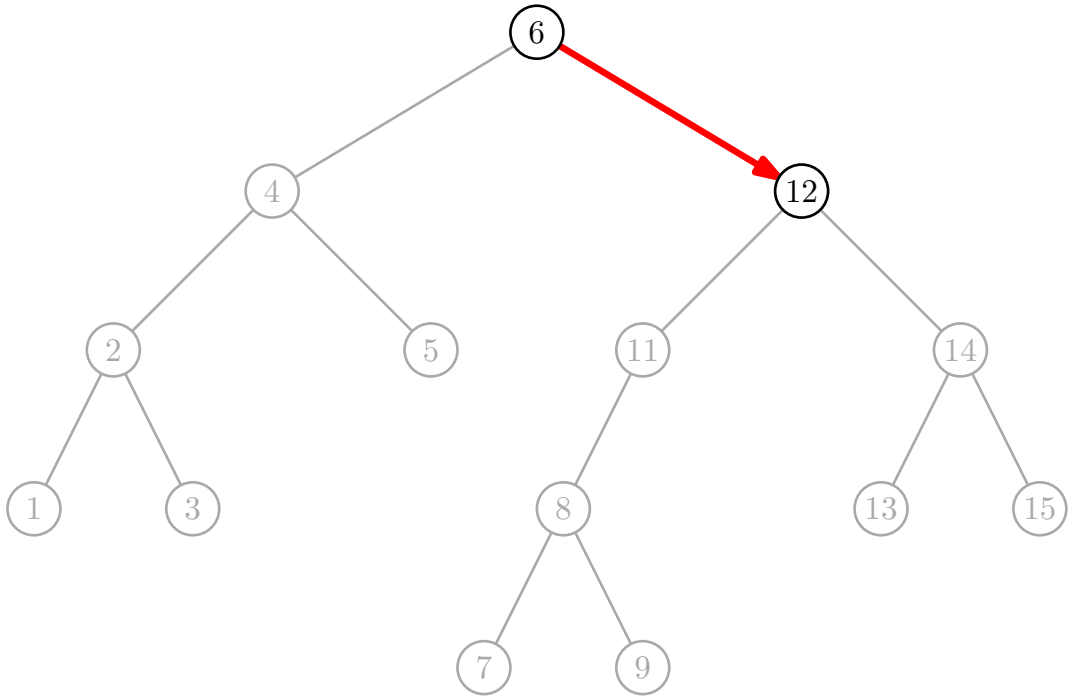
Insertion dans un ABR

Exemple – insertion de 10 :



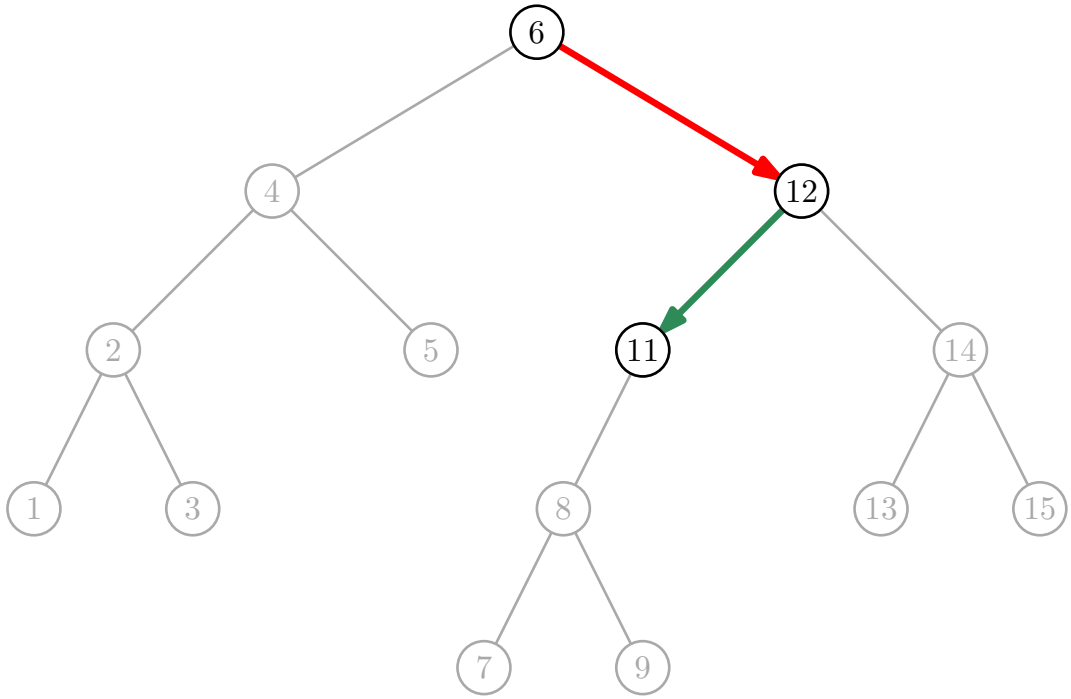
Insertion dans un ABR

Exemple – insertion de 10 :



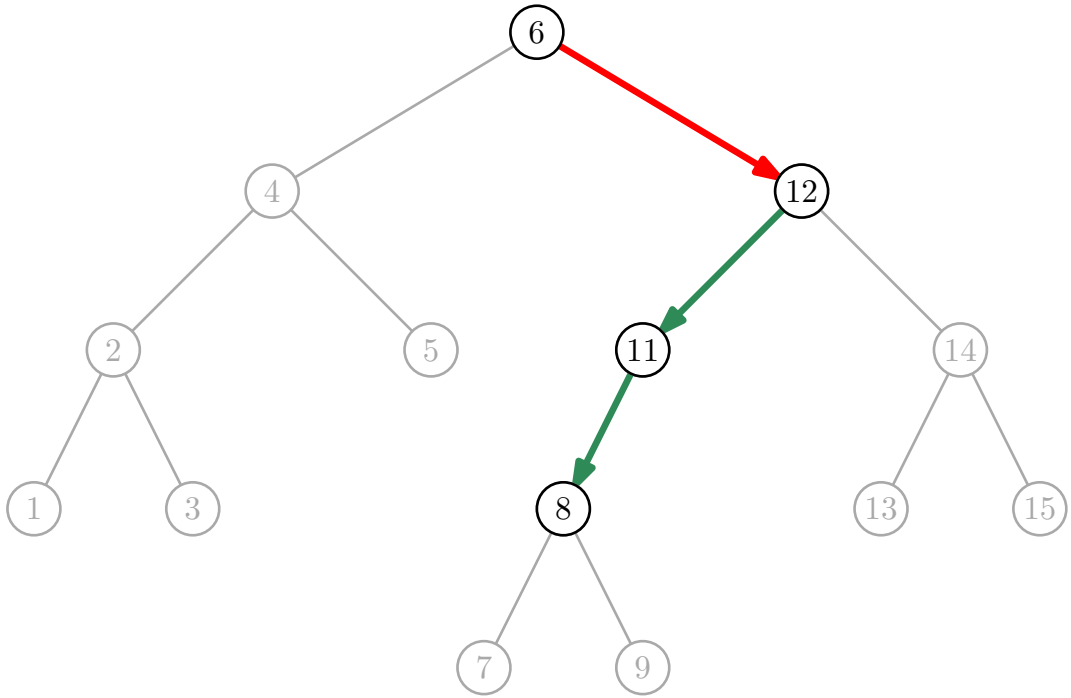
Insertion dans un ABR

Exemple – insertion de 10 :



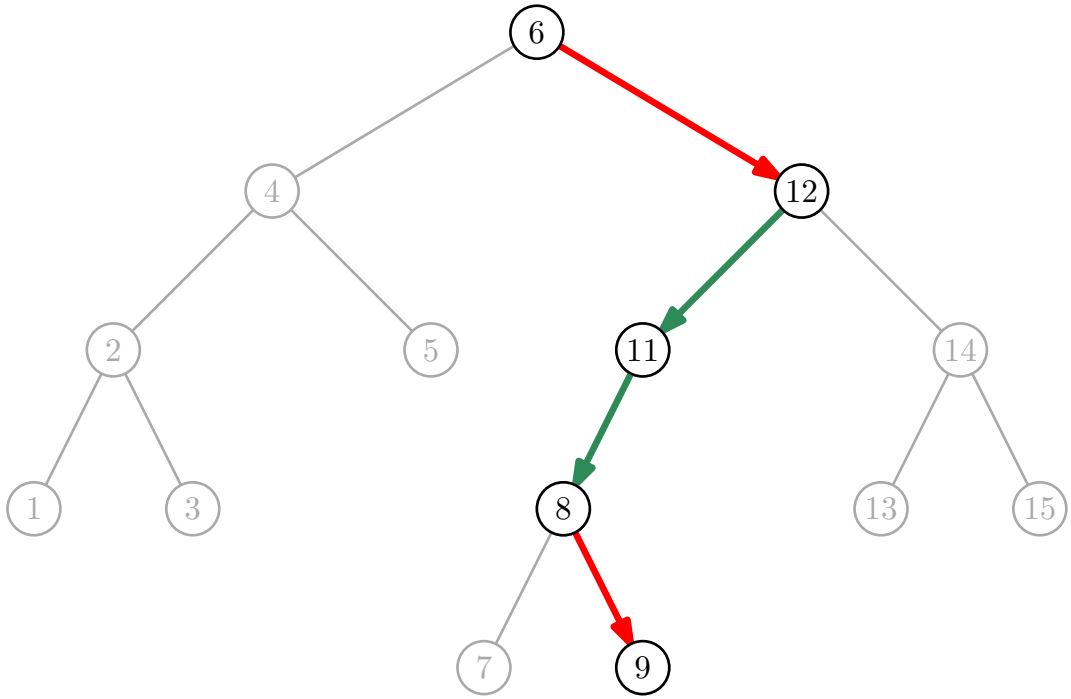
Insertion dans un ABR

Exemple – insertion de 10 :



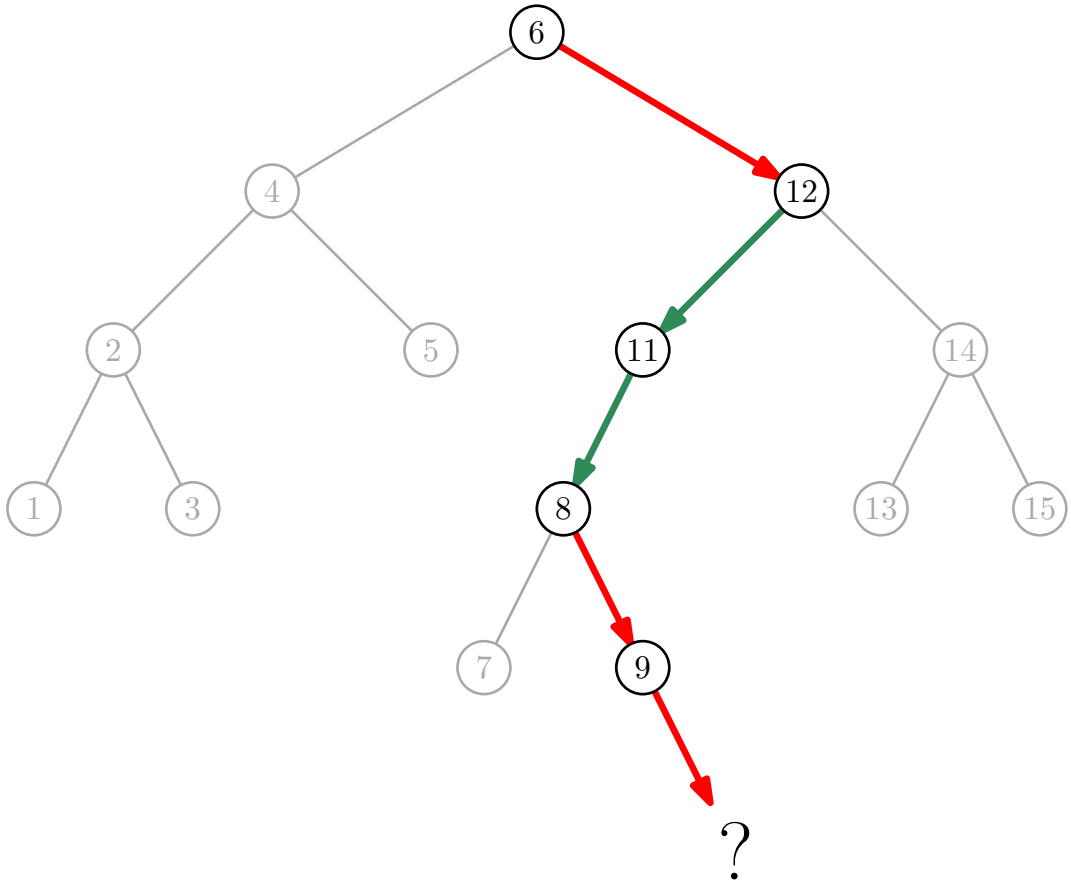
Insertion dans un ABR

Exemple – insertion de 10 :



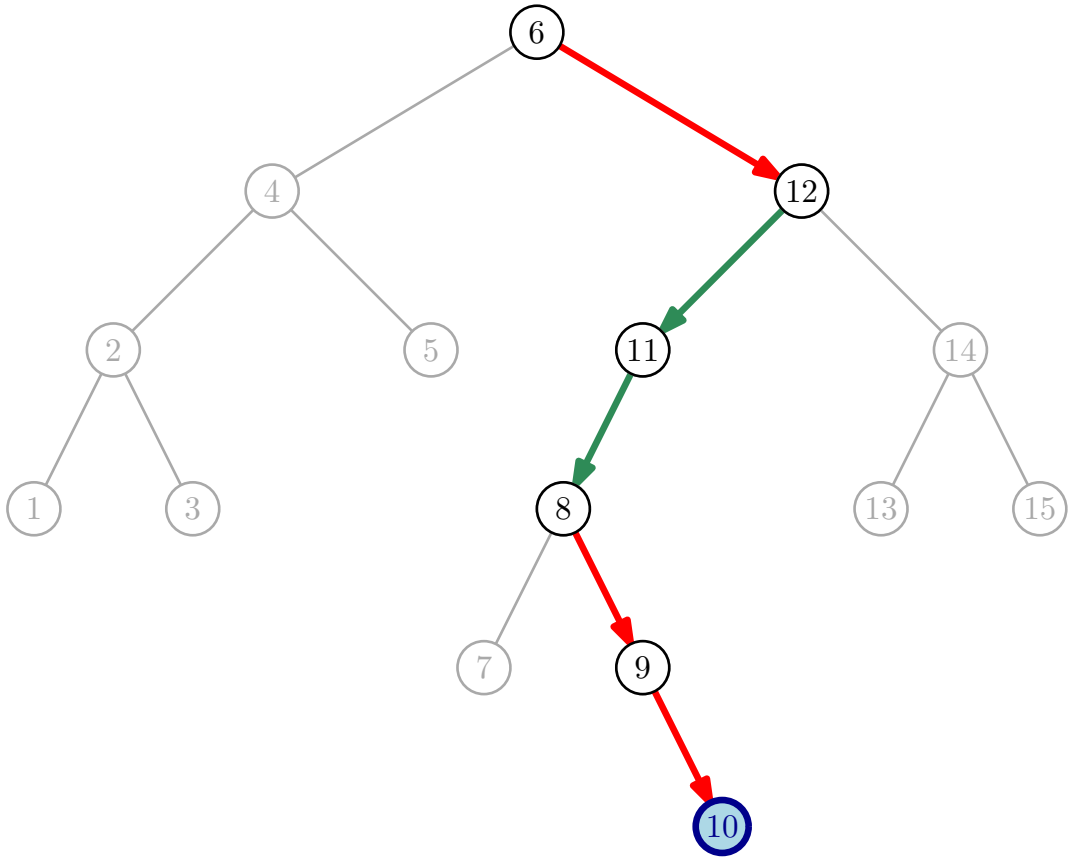
Insertion dans un ABR

Exemple – insertion de 10 :



Insertion dans un ABR

Exemple – insertion de 10 :



Puisque l'algorithme d'insertion n'est finalement qu'une petite modification de l'algorithme de recherche, sa complexité est exactement la même :

Théorème

L'insertion d'un nouvel élément dans un ABR de hauteur h *peut se faire en temps* $\Theta(h)$ *au pire.*

Une petite remarque concernant la création de la nouvelle feuille :

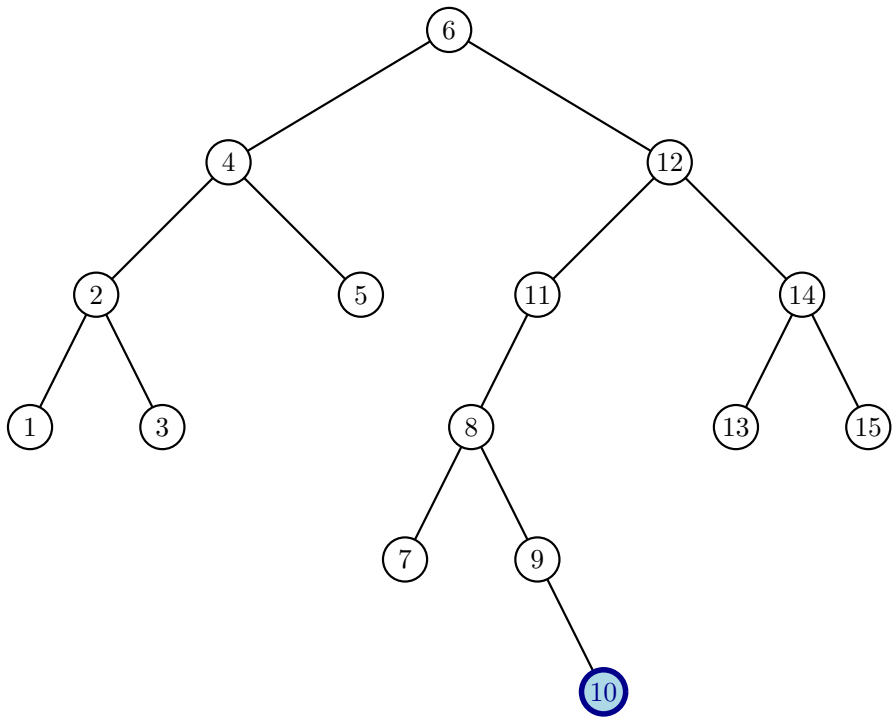
- si l'ABR est représenté uniquement par ses « vrais » sommets, ceux qui contiennent un élément, il faut faire la recherche en ayant constamment un pas de retard, sinon au moment où on arrive « dans » l'absence de feuille, on n'a pas de moyen d'accrocher la nouvelle feuille à son père.
- si l'ABR est complété, c'est beaucoup plus simple : en arrivant dans la feuille sans contenu, il suffit de la remplir, et de lui créer deux enfants vides.

Suppression dans un ABR

La suppression d'un élément `x` est relativement plus compliquée, notamment parce qu'il est utopique d'espérer éviter un peu de réorganisation lorsque `x` est contenu dans un nœud vraiment binaire.

Il y a tout de même des cas très simples :

1. si le nœud à supprimer n'a pas d'enfant : on l'enlève

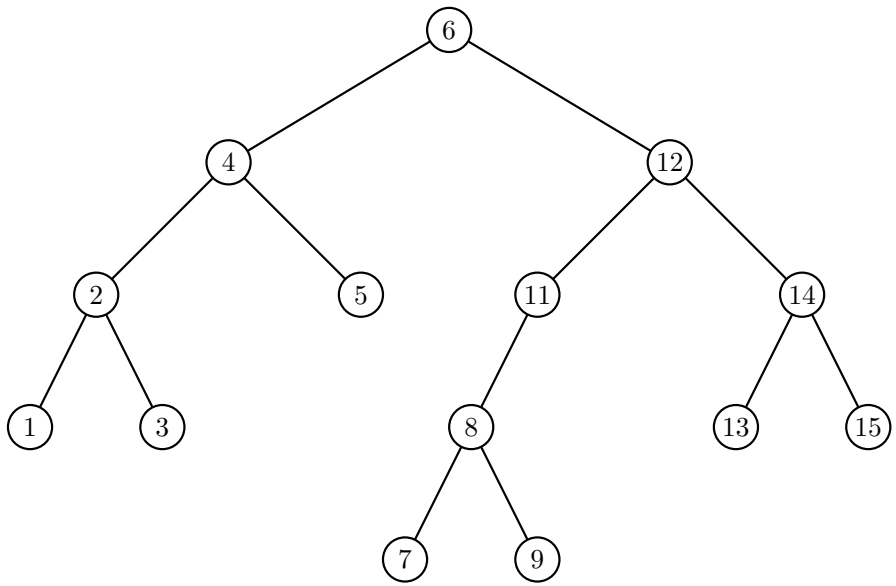


Suppression dans un ABR

La suppression d'un élément `x` est relativement plus compliquée, notamment parce qu'il est utopique d'espérer éviter un peu de réorganisation lorsque `x` est contenu dans un nœud vraiment binaire.

Il y a tout de même des cas très simples :

1. si le nœud à supprimer n'a pas d'enfant : on l'enlève

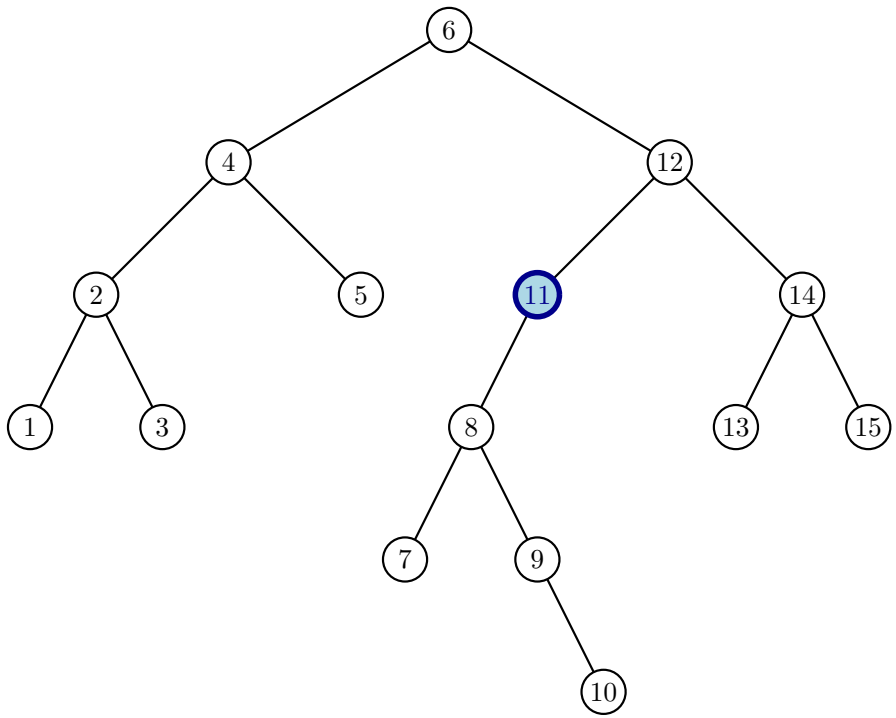


Suppression dans un ABR

La suppression d'un élément `x` est relativement plus compliquée, notamment parce qu'il est utopique d'espérer éviter un peu de réorganisation lorsque `x` est contenu dans un nœud vraiment binaire.

Il y a tout de même des cas très simples :

1. si le nœud à supprimer n'a pas d'enfant : on l'enlève
2. si le nœud à supprimer n'a qu'un enfant : on « remonte » son unique enfant

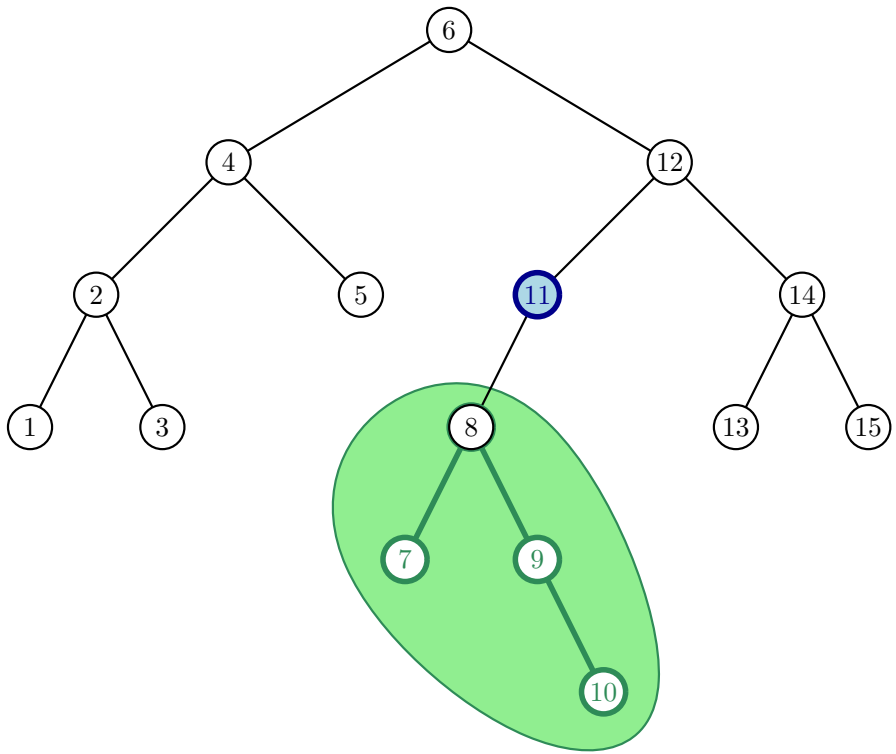


Suppression dans un ABR

La suppression d'un élément `x` est relativement plus compliquée, notamment parce qu'il est utopique d'espérer éviter un peu de réorganisation lorsque `x` est contenu dans un nœud vraiment binaire.

Il y a tout de même des cas très simples :

1. si le nœud à supprimer n'a pas d'enfant : on l'enlève
2. si le nœud à supprimer n'a qu'un enfant : on « remonte » son unique enfant

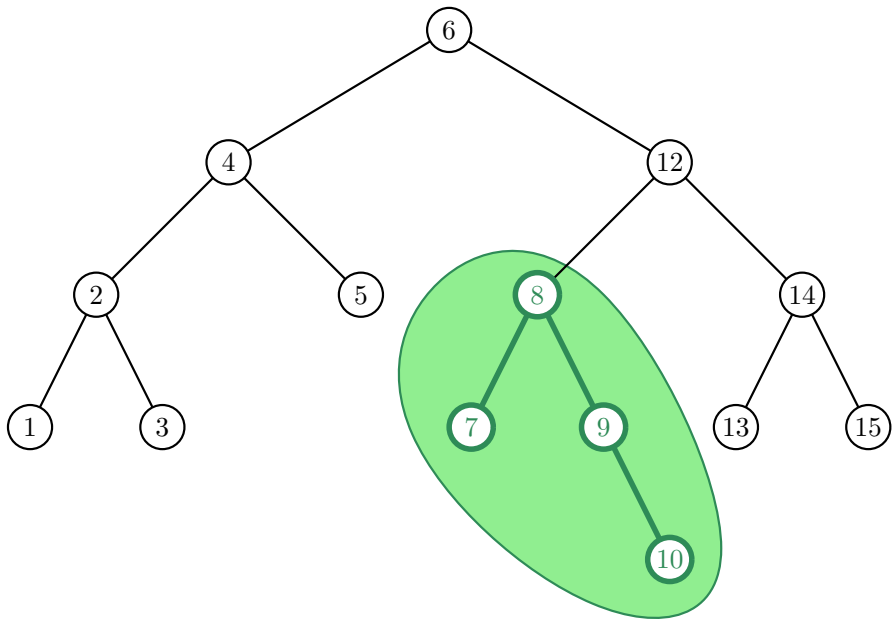


Suppression dans un ABR

La suppression d'un élément `x` est relativement plus compliquée, notamment parce qu'il est utopique d'espérer éviter un peu de réorganisation lorsque `x` est contenu dans un nœud vraiment binaire.

Il y a tout de même des cas très simples :

1. si le nœud à supprimer n'a pas d'enfant : on l'enlève
2. si le nœud à supprimer n'a qu'un enfant : on « remonte » son unique enfant

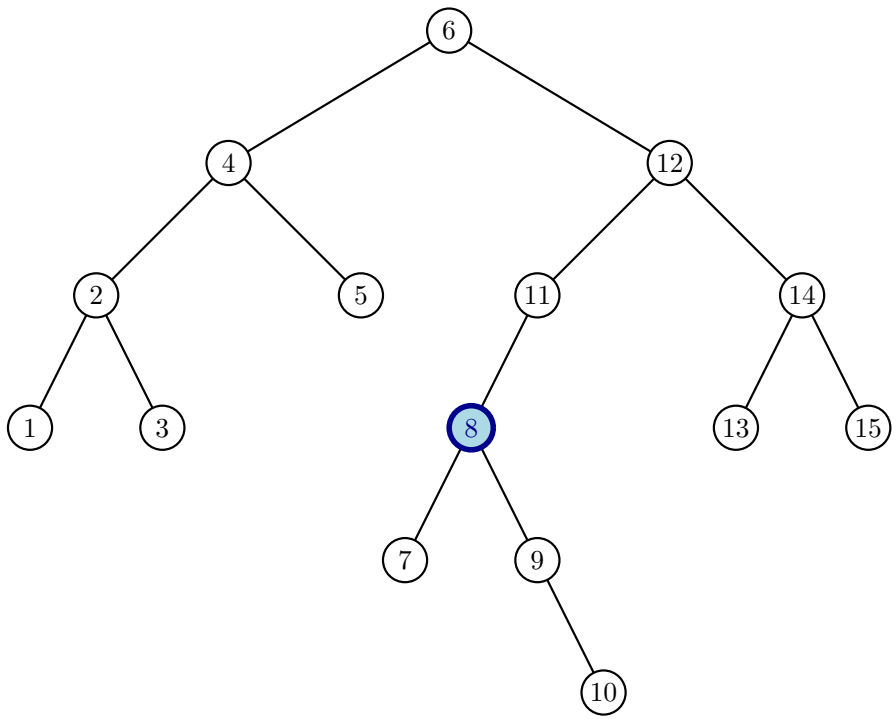


Suppression dans un ABR

Sinon, c'est-à-dire dans le cas générique où le nœud contenant `x` a deux enfants, le nœud ne peut pas être supprimé. Pour espérer ne faire que des modifications locales, il faut trouver un autre élément qui puisse prendre sa place. Pour cela, il y a exactement deux candidats : le prédécesseur et le successeur de `x`. En effet, ce sont les seuls éléments qui sont :

- plus petits que tous les (autres) éléments plus grands que `x`
- plus grands que tous les (autres) éléments plus petits que `x`

Les deux cas sont symétriques, regardons comment remplacer `x` par son successeur.

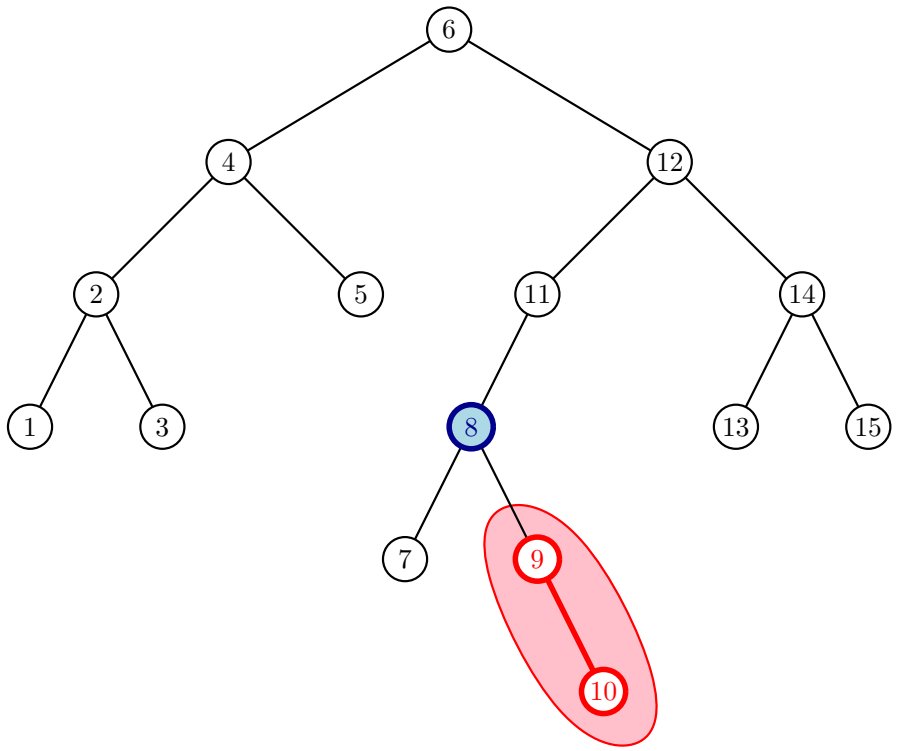


Suppression dans un ABR

Sinon, c'est-à-dire dans le cas générique où le nœud contenant `x` a deux enfants, le nœud ne peut pas être supprimé. Pour espérer ne faire que des modifications locales, il faut trouver un autre élément qui puisse prendre sa place. Pour cela, il y a exactement deux candidats : le prédécesseur et le successeur de `x`. En effet, ce sont les seuls éléments qui sont :

- plus petits que tous les (autres) éléments plus grands que `x`
- plus grands que tous les (autres) éléments plus petits que `x`

Les deux cas sont symétriques, regardons comment remplacer `x` par son successeur.

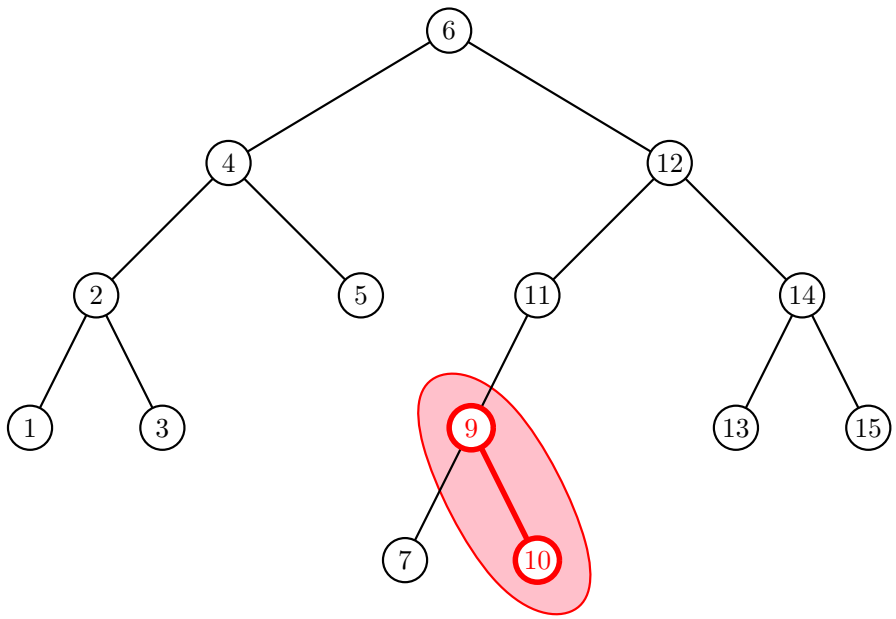


Suppression dans un ABR

Sinon, c'est-à-dire dans le cas générique où le nœud contenant `x` a deux enfants, le nœud ne peut pas être supprimé. Pour espérer ne faire que des modifications locales, il faut trouver un autre élément qui puisse prendre sa place. Pour cela, il y a exactement deux candidats : le prédécesseur et le successeur de `x`. En effet, ce sont les seuls éléments qui sont :

- plus petits que tous les (autres) éléments plus grands que `x`
- plus grands que tous les (autres) éléments plus petits que `x`

Les deux cas sont symétriques, regardons comment remplacer `x` par son successeur.

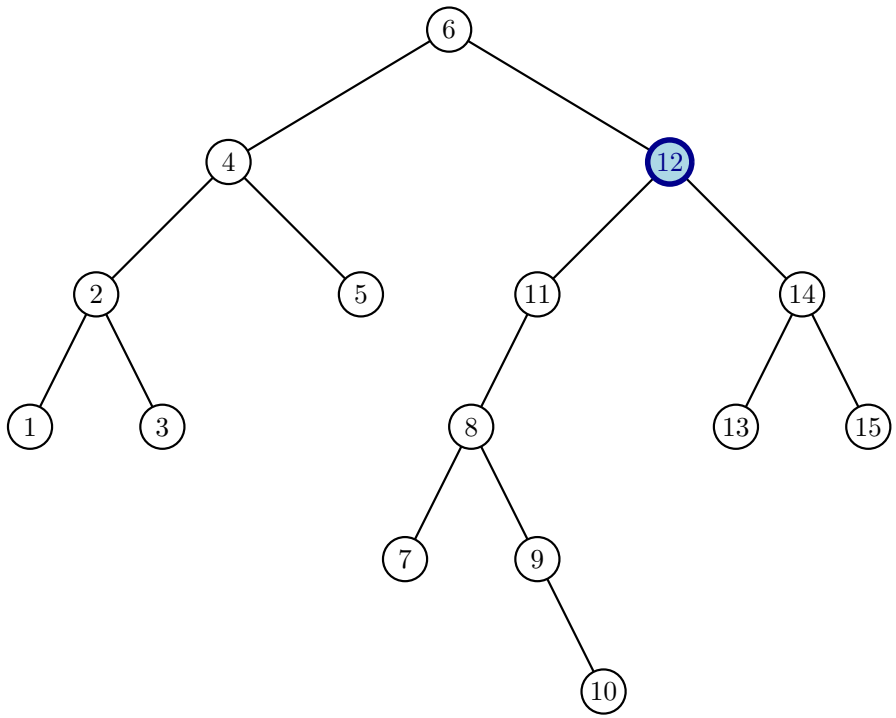


Suppression dans un ABR

Sinon, c'est-à-dire dans le cas générique où le nœud contenant `x` a deux enfants, le nœud ne peut pas être supprimé. Pour espérer ne faire que des modifications locales, il faut trouver un autre élément qui puisse prendre sa place. Pour cela, il y a exactement deux candidats : le prédécesseur et le successeur de `x`. En effet, ce sont les seuls éléments qui sont :

- plus petits que tous les (autres) éléments plus grands que `x`
- plus grands que tous les (autres) éléments plus petits que `x`

Les deux cas sont symétriques, regardons comment remplacer `x` par son successeur.

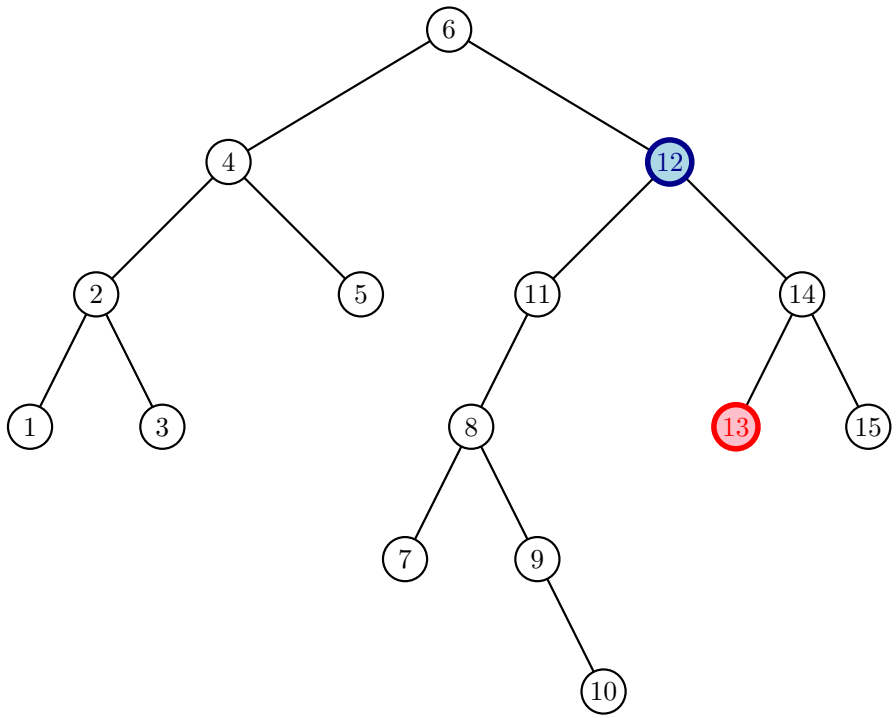


Suppression dans un ABR

Sinon, c'est-à-dire dans le cas générique où le nœud contenant `x` a deux enfants, le nœud ne peut pas être supprimé. Pour espérer ne faire que des modifications locales, il faut trouver un autre élément qui puisse prendre sa place. Pour cela, il y a exactement deux candidats : le prédécesseur et le successeur de `x`. En effet, ce sont les seuls éléments qui sont :

- plus petits que tous les (autres) éléments plus grands que `x`
- plus grands que tous les (autres) éléments plus petits que `x`

Les deux cas sont symétriques, regardons comment remplacer `x` par son successeur.

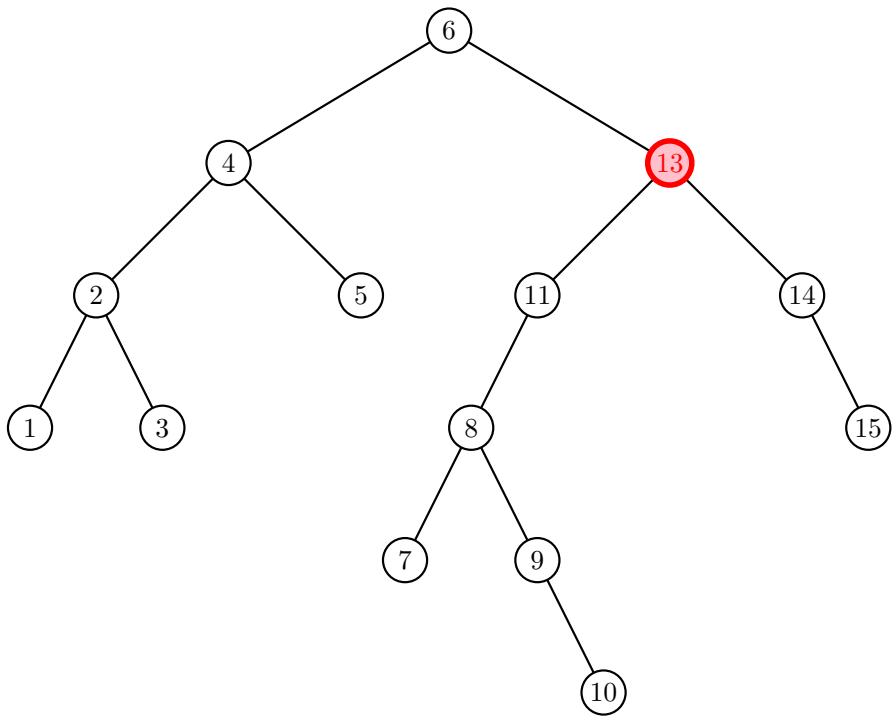


Suppression dans un ABR

Sinon, c'est-à-dire dans le cas générique où le nœud contenant `x` a deux enfants, le nœud ne peut pas être supprimé. Pour espérer ne faire que des modifications locales, il faut trouver un autre élément qui puisse prendre sa place. Pour cela, il y a exactement deux candidats : le prédécesseur et le successeur de `x`. En effet, ce sont les seuls éléments qui sont :

- plus petits que tous les (autres) éléments plus grands que `x`
- plus grands que tous les (autres) éléments plus petits que `x`

Les deux cas sont symétriques, regardons comment remplacer `x` par son successeur.

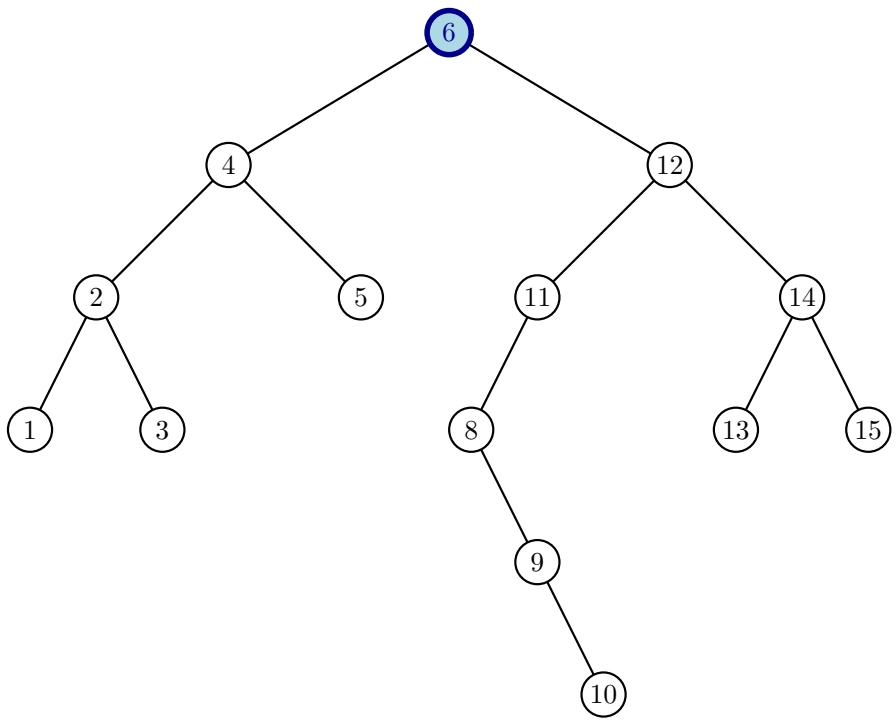


Suppression dans un ABR

Sinon, c'est-à-dire dans le cas générique où le nœud contenant `x` a deux enfants, le nœud ne peut pas être supprimé. Pour espérer ne faire que des modifications locales, il faut trouver un autre élément qui puisse prendre sa place. Pour cela, il y a exactement deux candidats : le prédécesseur et le successeur de `x`. En effet, ce sont les seuls éléments qui sont :

- plus petits que tous les (autres) éléments plus grands que `x`
- plus grands que tous les (autres) éléments plus petits que `x`

Les deux cas sont symétriques, regardons comment remplacer `x` par son successeur.

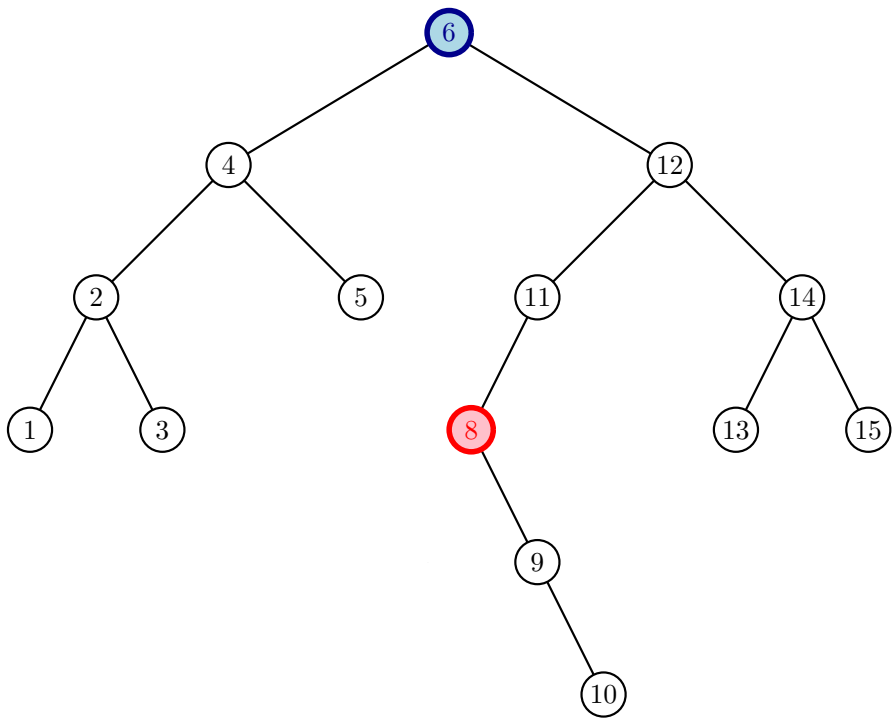


Suppression dans un ABR

Sinon, c'est-à-dire dans le cas générique où le nœud contenant `x` a deux enfants, le nœud ne peut pas être supprimé. Pour espérer ne faire que des modifications locales, il faut trouver un autre élément qui puisse prendre sa place. Pour cela, il y a exactement deux candidats : le prédécesseur et le successeur de `x`. En effet, ce sont les seuls éléments qui sont :

- plus petits que tous les (autres) éléments plus grands que `x`
- plus grands que tous les (autres) éléments plus petits que `x`

Les deux cas sont symétriques, regardons comment remplacer `x` par son successeur.

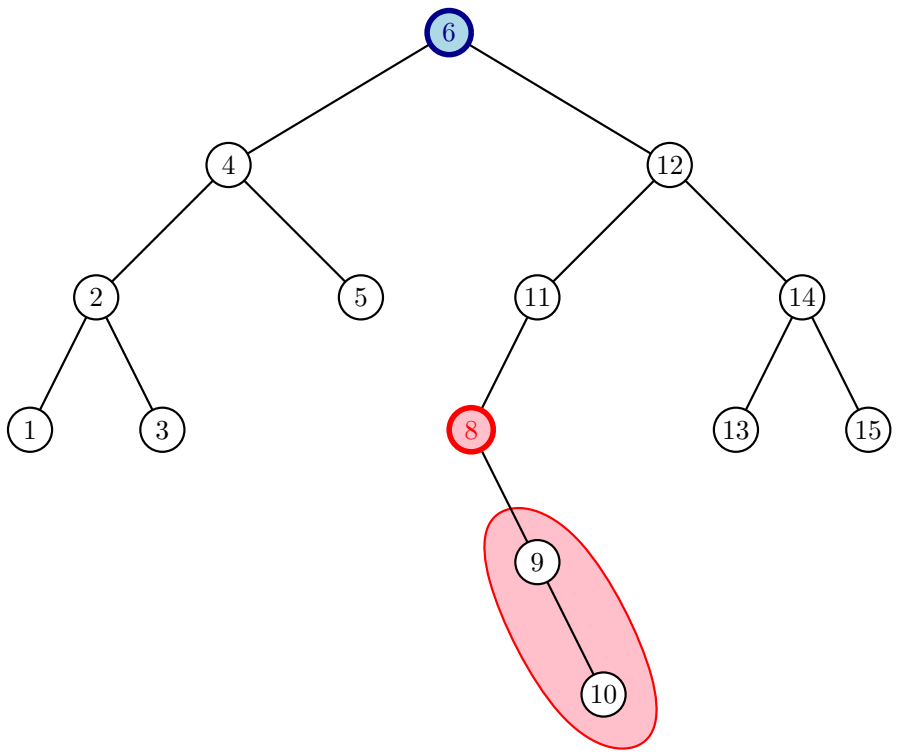


Suppression dans un ABR

Sinon, c'est-à-dire dans le cas générique où le nœud contenant `x` a deux enfants, le nœud ne peut pas être supprimé. Pour espérer ne faire que des modifications locales, il faut trouver un autre élément qui puisse prendre sa place. Pour cela, il y a exactement deux candidats : le prédécesseur et le successeur de `x`. En effet, ce sont les seuls éléments qui sont :

- plus petits que tous les (autres) éléments plus grands que `x`
- plus grands que tous les (autres) éléments plus petits que `x`

Les deux cas sont symétriques, regardons comment remplacer `x` par son successeur.

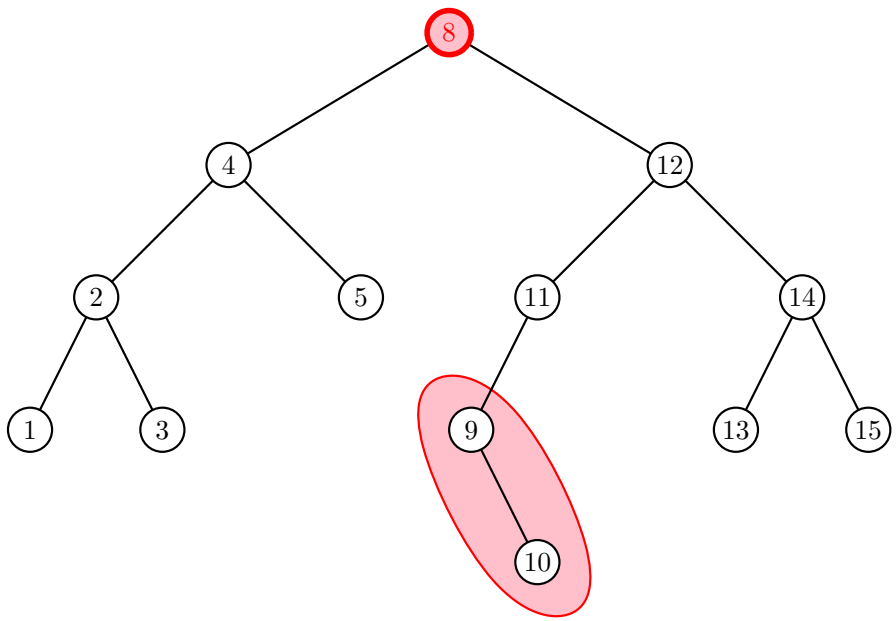


Suppression dans un ABR

Sinon, c'est-à-dire dans le cas générique où le nœud contenant `x` a deux enfants, le nœud ne peut pas être supprimé. Pour espérer ne faire que des modifications locales, il faut trouver un autre élément qui puisse prendre sa place. Pour cela, il y a exactement deux candidats : le prédécesseur et le successeur de `x`. En effet, ce sont les seuls éléments qui sont :

- plus petits que tous les (autres) éléments plus grands que `x`
- plus grands que tous les (autres) éléments plus petits que `x`

Les deux cas sont symétriques, regardons comment remplacer `x` par son successeur.



Sinon, c'est-à-dire dans le cas générique où le nœud contenant `x` a deux enfants, le nœud ne peut pas être supprimé. Pour espérer ne faire que des modifications locales, il faut trouver un autre élément qui puisse prendre sa place. Pour cela, il y a exactement deux candidats : le prédécesseur et le successeur de `x`. En effet, ce sont les seuls éléments qui sont :

- plus petits que tous les (autres) éléments plus grands que `x`
- plus grands que tous les (autres) éléments plus petits que `x`

Les deux cas sont symétriques, regardons comment remplacer `x` par son successeur.

On remarque en fait la propriété suivante :

Lemme

Le successeur d'un nœud à deux enfants n'a lui-même pas de fils gauche.

(forcément, puisque dans ce cas, il s'agit du minimum du sous-arbre droit...)

Donc en résumé :

1. cas d'une feuille : suppression simple
2. cas d'un nœud à un seul fils : l'autre fils remonte d'un niveau
3. cas où le successeur est le fils droit : le fils droit remonte d'un niveau et adopte son frère
4. autres cas : le noeud est remplacé par son successeur, dont l'unique fils (droit) remonte d'un niveau

Remarques :

- le point 3 n'est qu'un cas particulier du point 4
- la même manipulation peut être faite avec le prédécesseur plutôt que le successeur

Et donc, comme la recherche du successeur (ou du prédécesseur) est de complexité $\Theta(h)$ au pire (voir Partie III) :

Théorème

la suppression d'un nœud d'un ABR de hauteur h *se fait en temps* $\Theta(h)$ *au pire.*

Les différentes opérations vues en Parties III et IV de ce cours ont donc les complexités suivantes (au pire pour un arbre de taille n et de hauteur h) :

Théorème

la liste triée des éléments d'un ABR de taille n peut être obtenue en temps $\Theta(n)$ par un parcours en profondeur infixe.

Théorème

la recherche, l'ajout et la suppression d'un élément dans un ABR de hauteur h se font en temps $\Theta(h)$ au pire.

Corollaire

la construction d'un ABR de taille n par insertion successive de ses éléments a un coût $O(nh)$, si h est la hauteur de l'arbre obtenu.

La clé de l'efficacité de ces opérations réside donc dans la ***hauteur*** de l'ABR considéré en fonction de sa taille. En Partie V de ce cours, on démontrera :

Théorème

la hauteur moyenne d'un ABR construit par l'insertion des entiers $1, \ldots, n$ dans un ordre aléatoire choisi uniformément est en $\Theta(\log n)$.