Programmation – TD3 : Ensembles

{ Jeremie.Detrey, Emmanuel.Jeandel } @ens-lyon.fr

13/14 octobre 2004

Introduction

Nous nous intéressons ici à trouver une structure de données représentant les ensembles (finis). Il y a maintes opérations définissables sur les ensembles (intersection, union, appartenance, ajout d'un élément, ...) et la meilleure structure dépend bien entendu des opérations autorisées.

On cherche dans ce TD à offrir la possibilité d'effectuer les opérations d'ajout, de suppression, et le test d'appartenance.

Un module fournissant une telle structure aura la signature suivante :

```
module type SET = sig
  type t
  type set
  val empty : set
  val length : set -> int
  val mem : t -> set -> bool
  val add : t -> set -> set
  val remove : t -> set -> set
  val to_list : set -> t list
end;;
```

Expliquons la signature :

- `t` est le type des objets contenus dans l'ensemble ;
- `set` est le type ensemble ;
- `empty` est l'ensemble vide ;
- `length s` est le cardinal de l'ensemble `s` ;
- `mem x s` renvoie `true` si $x \in s$ et `false` sinon ;
- `add x s` renvoie l'ensemble $s \cup \{x\}$;
- `remove x s` renvoie l'ensemble $s \setminus \{x\}$;
- `to_list s` renvoie la liste des éléments de `s`.

Il s'agit bien d'ensembles et non pas de multiensembles ; ainsi, après le code suivant :

```
(* A est un module de signature SET pour lequel A.t = int *)
let l = A.remove 3 (A.add 3 (A.add 3 A.empty));;
```

l'ensemble `l` est bien vide.

1 Structures simples

1.1 Un exemple

Pour illustrer les propos précédents, voici une structure de données qui représente les ensembles d'entiers sous la forme suivante : un ensemble est un couple $(\texttt{f}, \texttt{l})$ où `f` est une fonction qui teste pour chaque entier son appartenance à l'ensemble, et où `l` est la longueur de l'ensemble.

```
module IntSetByFun : (SET with type t = int) =
struct
  type t = int
  type set = (int -> bool) * int

  let empty = ((fun x -> false), 0)

  let length (f,l) = l

  let mem x (f,l) = (f x)

  let add  y (f,l) =
    if (f y) then (f,l)
    else ((fun x -> if x = y then true else (f x)) , l+1)

  let remove y (f,l) =
    if (f y) then ((fun x -> if x = y then false else (f x)) , l-1)
    else (f,l)

  let to_list (f,l) =
    (* n : nombre d'éléments à trouver            *)
    (* i : élément à tester pour son appartenance *)
    let rec to_list_aux n i =
       if n = 0 then
         []
       else if (f i) then
         i::(to_list_aux (n-1) (i+1))
       else
         (to_list_aux n (i+1))
    in to_list_aux l 0
end;;
```

Une lecture avisée de ce qui précède permet de repérer rapidement en quoi cette structure est inefficace. Tant qu'on se contente d'ajouter ou de supprimer des éléments, elle remplit parfaitement son rôle. En revanche, elle est totalement inadéquate si on veut obtenir la liste de tous les éléments d'un ensemble. En effet, qu'aurait-on fait si on avait voulu faire un ensemble de chaînes de caractères, plutôt qu'un ensemble d'entiers ? Il aurait fallu, pour la fonction `to_list`, énumérer *toutes* les chaînes de caractères. La haute inefficacité d'un tel procédé est assez claire.

1.2 Listes

On représente maintenant un ensemble (par exemple d'entiers) par une liste de ces éléments.

1. Écrire un module nommé `IntSetByList` de signature `SET` utilisant cette représentation par listes :

```
module IntSetByList : (SET with type t = int) =
struct
  type t = int
  type set = t list
  (* . . . *)
end;;
```

Supposons maintenant que vous vouliez faire des ensembles de chaînes de caractères plutôt que des ensembles d'entiers. Vous devriez a priori retaper tout ce que vous avez écrit auparavant en remplaçant `int` par `string`. Un foncteur serait sans doute plus élégant.

Pour cela, il va suffire, pour chaque type d'ensemble qu'on veut avoir, de disposer d'un module représentant ce type. Ce module aura pour signature :

```
module type ELEMENT =
sig
  type t
end;;
```

2. Transformez votre code en un foncteur `SetByList` qui prend un module `A` de signature `ELEMENT` et renvoie un module représentant des ensembles d'éléments de type `A.t` :

```
module SetByList (A : ELEMENT) : (SET with type t = A.t)  =
struct
  type t = A.t
  (* . . . *)
end;;
```

Ainsi, si vous voulez avoir un ensemble d'entiers, il vous suffit d'écrire :

```
module Integer : (ELEMENT with type t = int) =
struct
  type t = int
end;;

module IntSetByList = SetByList(Integer);;
```

1.3 Vers une nouvelle structure de données

La méthode par listes que vous avez écrit auparavant marche parfaitement, mais possède néanmoins un inconvénient. Le test d'appartenance (et le test de suppression) nécessite de parcourir dans le pire cas toute la liste, de sorte que la complexité du test d'appartenance est en $\mathcal{O}(n)$ où n est le nombre d'éléments de l'ensemble. Il est assez difficile de faire mieux lorsqu'on ne dispose pas d'information supplémentaire sur le type des éléments.

Cependant, en pratique, les éléments qu'on veut mettre dans l'ensemble sont souvent d'un type particulier, ils sont *ordonnés*. C'est le cas des entiers, des chaînes de caractères, c'est en fait vrai dans Caml pour à peu près tous les types hormis les fonctions.

La suite du TD supposera donc que le type des éléments est donné par un module de signature :

```
module type ORDERED =
sig
  type t
  val compare : t -> t -> int
    (* (compare x y) < 0  si x < y *)
    (* (compare x y) = 0  si x = y *)
    (* (compare x y) > 0  si x > y *)
end;;
```

Par exemple, les entiers seront donnés par le module suivant :

```
module Integer: ORDERED =
struct
  type t = int
  let compare x y = (x - y)
end;;
```

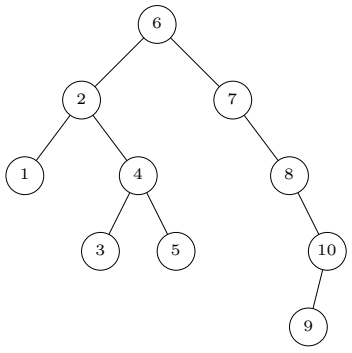
2 Arbres binaires de recherche

2.1 Définition

Maintenant que les éléments de nos ensembles sont ordonnés, on peut utiliser des structures de données plus complexes pour représenter ces ensembles. Adoptons alors pour commencer une structure d'arbre binaire de recherche : il s'agit d'un arbre binaire tel que :

- chaque nœud contient un élément x ;
- tous les éléments y du sous-arbre gauche d'un nœud contenant x vérifient $y < x$;
- tous les éléments y du sous-arbre droit d'un nœud contenant x vérifient $y > x$.

L'ensemble d'entiers $\{1, 2, \ldots, 10\}$ peut ainsi être représenté par plusieurs arbres de recherche, dont le suivant (les feuilles de l'arbre ne sont pas représentées) :

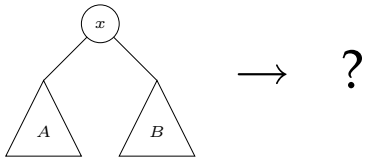


Remarque : si vous avez jeté un œil au DM (hum...), vous devez déjà avoir les idées claires à propos des arbres binaires de recherche.

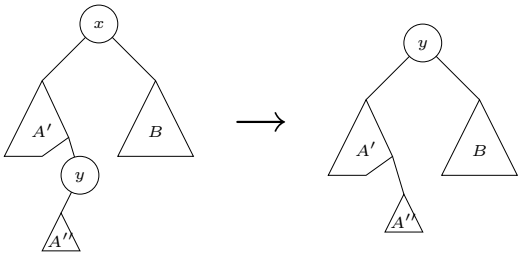
2.2 Insertions et suppressions

L'insertion dans un arbre binaire de recherche est très simple : il suffit de parcourir l'arbre jusqu'à trouver la feuille où l'élément à insérer doit aller, puis de remplacer cette feuille par un nouveau nœud.

Par contre, la suppression est plus complexe. En effet, si le nœud qui contient la valeur à supprimer x se trouve au milieu de l'arbre (c'est-à-dire que ses fils gauche et droit ne sont pas des feuilles), comment recoller ses fils correctement pour que le résultat soit toujours un arbre binaire de recherche ?



La solution est de trouver le nœud correspondant à y le prédécesseur direct de x selon la relation d'ordre : il s'agit en fait du nœud le plus à droite du sous-arbre gauche de x. On peut alors vérifier que tous les éléments z (hormis y) du sous-arbre gauche de x vérifient $z < y$ et tous ceux du sous-arbre droit vérifient $z > y$. On peut aussi vérifier que le nœud contenant y n'a qu'une feuille comme sous-arbre droit : supprimer ce nœud est alors facile, car cela revient à le remplacer par son sous-arbre gauche. On peut donc sans risque mettre y dans le nœud qui contenait x, puis supprimer le nœud qui contenait y.



Remarque : cette méthode marche aussi en prenant pour y le successeur direct de x.

2.3 À l'attaque

1. Écrivez un foncteur `SetByTree` qui prend un module `A` de signature `ORDERED` et renvoie un module représentant des ensembles d'éléments de type `A.t` en utilisant des arbres binaires de recherche :

```
module SetByTree (A : ORDERED) : (SET with type t = A.t)  =
struct
  type t = A.t
  type set = Node of (t * set * set) | Leaf
  (* . . . *)
end;;
```

Remarque : faites attention à ce que vos fonctions ne reconstruisent pas l'arbre tout entier lorsqu'elles peuvent l'éviter.

3 Arbres AVL (Adelson-Velskii et Landis)

⚠ Les arbres AVL sont au programme du cours d'algorithmique, mais ne seront abordés qu'au cours de ce TD. Des questions sur ces arbres peuvent donc tomber au partiel et/ou à l'examen d'algorithmique. N'oubliez pas de les réviser !

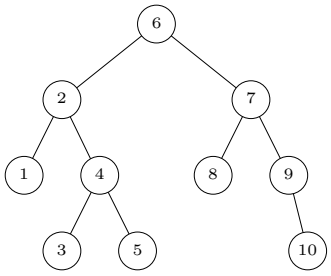
3.1 Définition

Utiliser une structure d'arbre binaire de recherche ne suffit pas pour obtenir une complexité acceptable pour l'algorithme de recherche ; après tout, une liste triée peut apparaître comme un cas particulier d'arbre binaire de recherche pour lequel le fils droit est toujours réduit à une feuille (on appelle cette structure un *peigne*). Dans le pire cas, on peut donc en être réduit à une complexité de $\mathcal{O}(n)$ pour chercher l'appartenance d'un élément à un ensemble de taille n.

Pour éviter ceci, on cherche à rendre l'arbre le plus symétrique possible. L'idéal est d'avoir un arbre où toutes les feuilles sont à la même profondeur, ce qui permet de faire une recherche en $\mathcal{O}(\log n)$, ce qui est optimal. Cependant, maintenir une telle structure lorsque l'on fait des insertions et des suppressions est délicat, et une nette dégradation des performances s'ensuit.

Au lieu d'imposer cette condition très restrictive, la méthode des arbres AVL utilise une condition plus large : un arbre de recherche est un AVL si en tout nœud, l'arbre est presque équilibré au sens où la profondeur du fils gauche ne diffère de celle du fils droit d'au plus 1.

Voici un exemple d'AVL (toujours sans représenter les feuilles) :



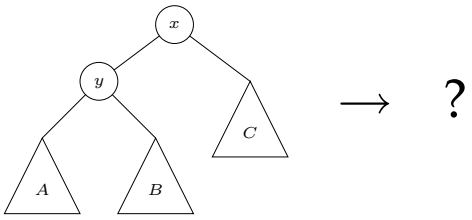
On peut montrer que la profondeur d'un AVL de n nœuds est au plus de $\sqrt{2}\log n$, de sorte qu'on est assuré, si on maintient cette contrainte, que l'algorithme de recherche sera en $\mathcal{O}(\log n)$.

Il reste maintenant à expliquer comment faire insertion et suppression.

3.2 Rotations

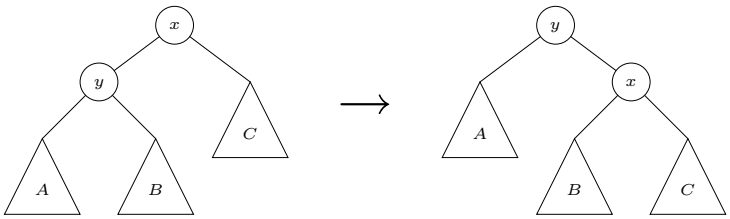
Le principe de l'insertion et de la suppression est basé sur la notion de rotations. On se pose le problème de la façon suivante : étant donné un nœud d'un arbre binaire de recherche dont la différence de profondeur entre les fils vaut 2, comment réorganiser l'arbre de façon à le transformer en AVL ?

Sans perte de généralité, supposons que c'est le fils gauche qui est trop gros :



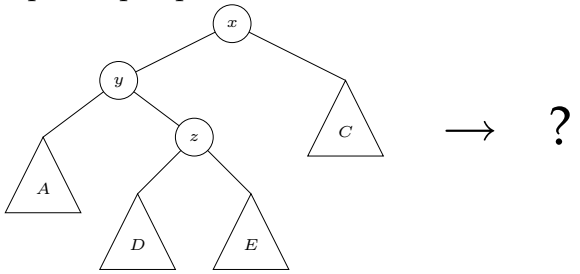
C est de profondeur n et le sous-arbre issu de y de profondeur $n+2$. Donc l'un des deux sous-arbres A et B est de profondeur $n+1$. Plusieurs cas se présentent suivant les profondeurs de A et B.

– Si A est de profondeur $n+1$, on effectue une *rotation droite* :

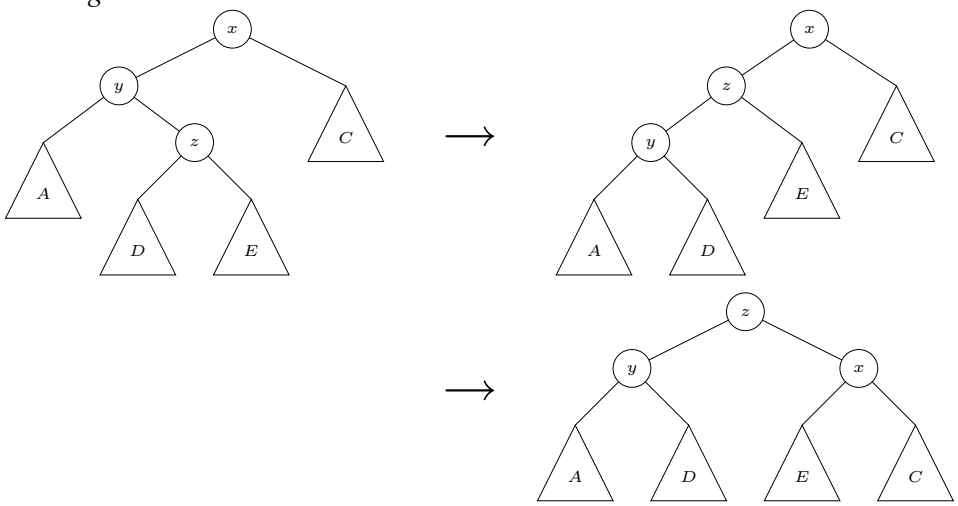


On vérifie bien que le résultat est un AVL.

– Sinon A est de profondeur n et B de profondeur $n+1$. La transformation précédente ne marche pas. Il faut découper un peu plus l'arbre :



Les sous-arbres D et E sont donc de profondeur $n-1$ ou n. Il suffit alors de faire une *double rotation gauche-droite* :



3.3 Insertions et suppressions

Une fois connu le principe des rotations, l'algorithme d'insertion s'écrit facilement par récurrence : pour insérer x dans un AVL, on insère récursivement dans la branche droite ou gauche suivant la valeur de x. Si l'insertion a provoqué un déséquilibre (donc une différence de 2), il suffit d'appliquer le principe de rotation pour rééquilibrer l'AVL.

1. Illustrer l'explication sur l'exemple suivant : insérer 8, 9, 11, 13, 2, 32, 18, 52, 19 dans un AVL initialement vide.

La suppression est similaire, avec cependant une légère variante. Comme pour les arbres binaires de recherche classique, on commence par s'arranger pour pouvoir supprimer en feuille, en changeant x avec son prédécesseur ou successeur immédiat.

La suppression d'un nœud en feuille se fait alors là encore par récurrence. Il suffit à chaque étape de supprimer le nœud dans la branche gauche ou la branche droite. Si jamais on obtient une différence de 2 entre les branches, on opère une rotation pour rétablir l'équilibre.

2. Sur l'exemple précédent, supprimer 52, 9, puis 13.

Pour votre culture, vous pouvez démontrer qu'il y a au plus une rotation lors d'une insertion dans un AVL. En revanche, la suppression peut en nécessiter plusieurs.

3.4 La vraie vie

On représente un AVL par la structure suivante :

```
type tree = Node of (A.t * avl * avl) | Leaf
and  avl = tree * int
```

Ainsi, à chaque nœud ou feuille `x` est adjoint un entier `y` correspondant à la profondeur du sous-arbre de racine `x`. Cette profondeur est donc nulle pour une feuille.

Avant d'écrire le foncteur proprement dit, on va d'abord faire quelques fonctions basiques :

3. Écrivez la fonction `l_rot : avl -> avl` qui prend un AVL et lui applique une rotation gauche. Faites attention à bien gérer les cas où la structure de l'AVL ne permet pas cette transformation.

4. Même question pour les fonctions `r_rot`, `rl_rot` et `lr_rot`.

5. Écrivez une fonction `balance : avl -> avl` qui prend un AVL possiblement déséquilibré (c'est-à-dire dont la différence de profondeur des sous-arbres fils vaut 2) et lui applique la méthode décrite plus haut pour le rééquilibrer.

Prêts pour le grand saut ?

6. Écrivez un foncteur `SetByAVL` qui prend un module `A` de signature `ORDERED` et renvoie un module représentant des ensembles d'éléments de type `A.t` en utilisant des AVL.

3.5 Un peu du bonus

7. Enrichissez votre foncteur d'une fonction `pop : set -> t` qui renvoie le plus petit élément de l'ensemble.

8. Écrivez une fonction qui trie une liste en insérant tous ses éléments (que l'on considèrera distincts) dans un ensemble, puis qui fait des `pop` successifs pour récupérer les éléments dans l'ordre croissant. Cette méthode est parfaitement stupide et inefficace (bien qu'en $\mathcal{O}(n \log n)$), mais vous permettra de tester vos fonctions d'ajout et de suppression d'éléments.

4 *Bonus* : Arbres rouge-noir

4.1 Définition

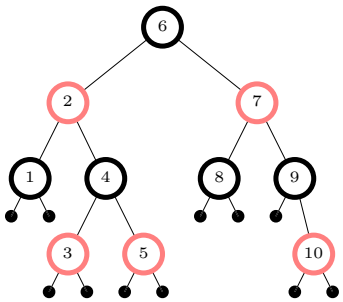
Les arbres rouge-noir[1] sont, tout comme les arbres AVL, des arbres binaires de recherche presque équilibrés. Les opérations d'insertion, suppression et recherche d'un élément se font donc en $\mathcal{O}(\log n)$.

Tous les nœuds d'un arbre rouge-noir ont une couleur, qui peut-être soit rouge soit noire (ou bien turquoise et caca d'oie si vous préférez être aux normes de la charte graphique de l'ENS). Un arbre rouge-noir doit alors vérifier les quatre propriétés suivantes :

1. sa racine est noire ;
2. ses feuilles sont noires ;
3. les deux fils d'un nœud rouge sont noirs ;
4. les chemins de chaque feuille vers la racine ont tous le même nombre de nœuds noirs.

Ces propriétés permettent de déduire que le chemin le plus long de la racine vers une feuille d'un arbre rouge-noir ne peut être plus de deux fois plus long que le chemin le plus court. Cela garantit un équilibre suffisant de l'arbre pour assurer son efficacité.

Voici un exemple d'arbre rouge-noir (pour ces arbres, on représente les feuilles, pour plus de clarté) :



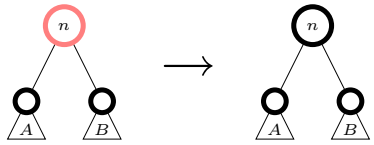
[1] Rien à voir avec l'anarcho-syndicalisme, Stendhal ou Jeanne Mas.

4.2 Insertion (on se motive)

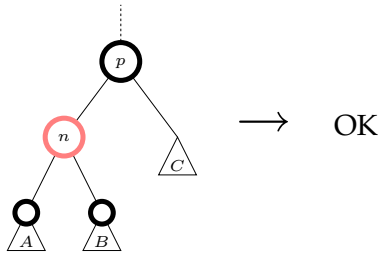
L'insertion d'une valeur n (comme *Node*) dans un arbre rouge-noir se fait dans un premier temps comme l'insertion dans un arbre binaire de recherche : on crée un nouveau nœud contenant n à la place d'une feuille de l'arbre, et on colore ce nœud en rouge.

Il va falloir ensuite considérer plusieurs cas (comme la fonction va être récursive, on se place ici dans le cas général ou l'on vient de colorer un nœud n en rouge, ses deux fils étant noirs) :

1. Si le nœud n est la racine de l'arbre, il suffit de colorer ce nœud en noir :

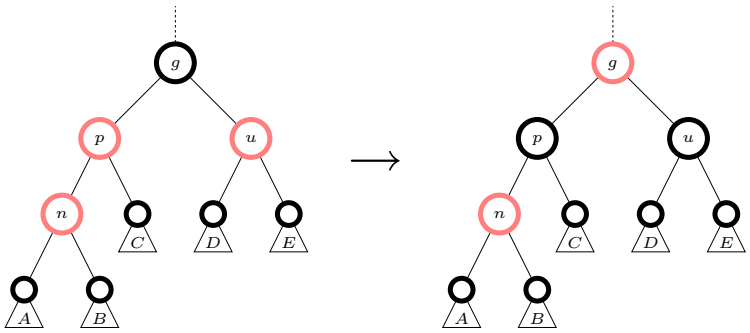


2. Sinon, si le père p (comme *Parent*) de n est noir, c'est bon : l'arbre est toujours un arbre rouge-noir ;

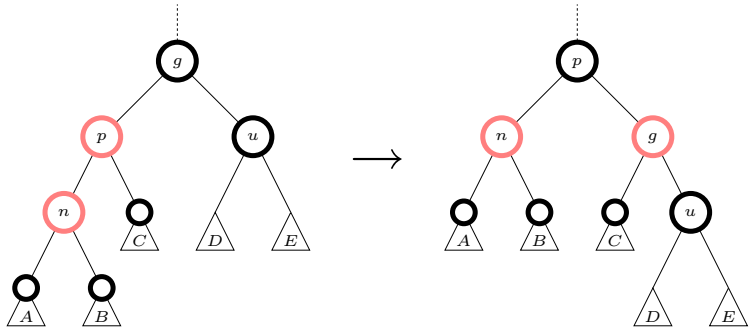


Par contre, si p est rouge, p ne peut être racine de l'arbre. Il a donc un père noir g (comme *Grand-parent*), ainsi qu'un frère u (comme *Uncle*), ce dernier pouvant être une feuille. On considère par la suite que p est le fils gauche de g, et u son fils droit (dans le cas contraire, la méthode à appliquer se déduit de celle-ci par symétrie). On a alors :

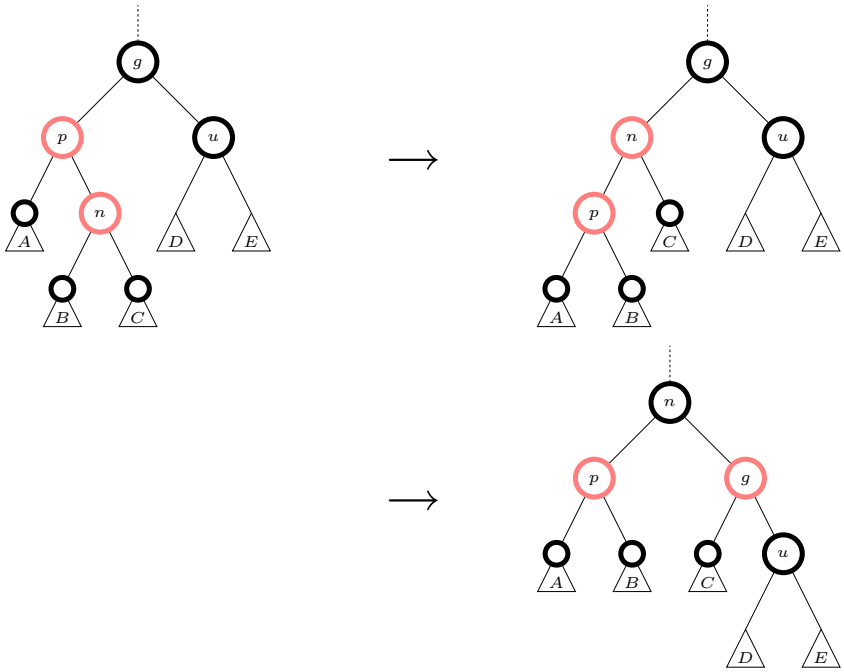
3. Si p et u sont rouges, on peut les colorer en noir, puis, si g n'est pas la racine, le colorier en rouge pour maintenir la propriété 4. Cependant, il se peut que le père de g soit aussi rouge. Il faut alors appliquer récursivement cette méthode à g :



4. Si p est rouge mais u est noir, et si n est le fils gauche de p, alors on effecture une rotation droite en g pour rééquilibrer l'arbre, tout en colorant g en rouge et p en noir :



5. Enfin, si p est rouge, u est noir et n est le fils droit de p, alors on peut effectuer une double rotation gauche-droite, puis colorer g en rouge et n en noir (en fait la première rotation gauche nous ramène au cas précédent) :

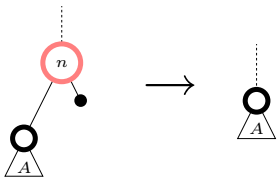


4.3 Suppression (pour les gens qui n'en veulent)

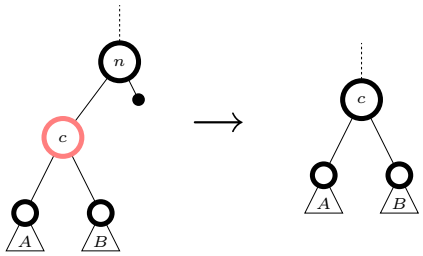
Pour supprimer un élément x dont les deux fils ne sont pas une feuille, on procède comme pour les arbres binaires de recherche, et on intervertit le nœud x avec le nœud de son prédécesseur (ou de son successeur) immédiat y.

On se ramène donc ainsi au cas suivant (et où c'est bien le boxon) : on veut supprimer un nœud n donc au moins un des fils est une feuille. Partant de là, on a plein de cas :

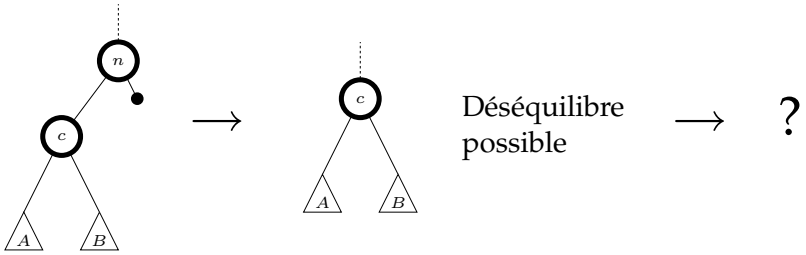
1. Si n est rouge, pas de problème : on le remplace par celui de ses fils qui n'est pas une feuille (ou par une feuille si ses deux fils sont des feuilles) :



2. Si n est noir, mais que celui de ses fils qui n'est pas une feuille (que l'on note c comme *Child*) est rouge, alors on peut remplacer n par c et colorer ce dernier en noir :

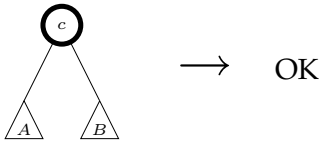


Si n et c sont noirs, on peut remplacer n par c. Malheureusement, ceci invalide la propriété 4 car il manque un nœud noir sur les chemins des feuilles à la racine passant par c.



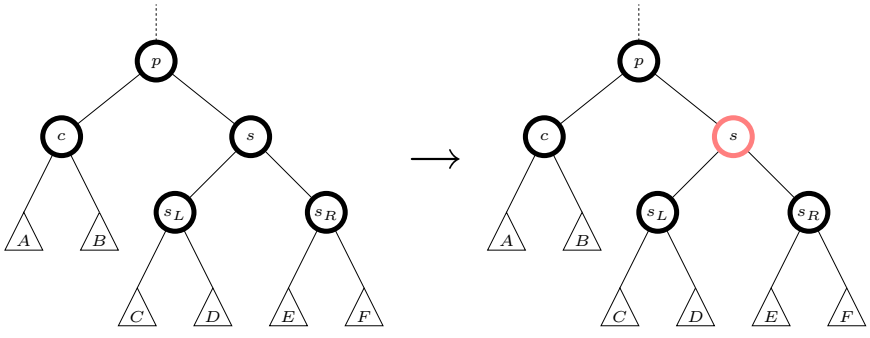
Tentons alors de rétablir l'équilibre par la procédure suivante :

1. Si n était la racine, alors en supprimant n on n'a pas invalidé la propriété 4, car un nœud noir a été enlevé de tous les chemins feuille-racine :

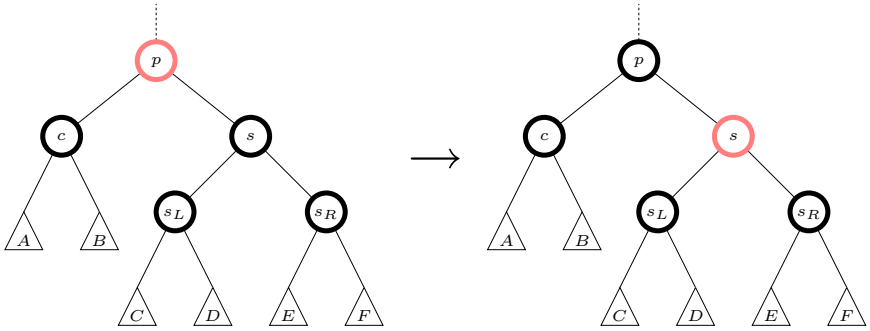


Par contre, si n n'était pas racine, on peut considérer son père p, ainsi que son frère s (comme *Sibling*), qui sont désormais le père et le frère de c. Comme l'arbre était correct avant de supprimer n, et que n était noir, la propriété 4 nous indique que s doit forcément avoir deux fils (qui peuvent être des feuilles), que l'on note s_L et s_R pour les fils gauche et droit respectivement. On ne considère ici que les cas où c est le fils gauche de p, les autres se déduisant par symétrie. Cela nous donne alors :

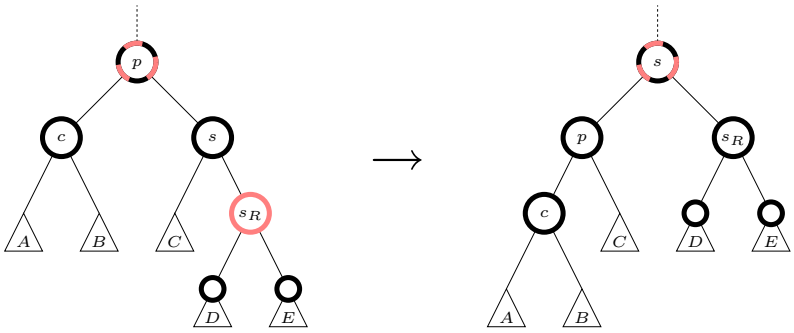
2. Si p, s, s_L et s_R sont noirs, alors colorer s en rouge va supprimer un nœud noir dans tous les chemins feuille-racine passant par s, et donc équilibrer le sous-arbre enraciné en p. Par contre, l'arbre total reste déséquilibré, car nous n'avons corrigé que les chemins feuille-racine passant par p. Il faut alors réitérer la méthode récursivement pour rééquilibrer l'arbre en considérant cette fois-ci le nœud p :



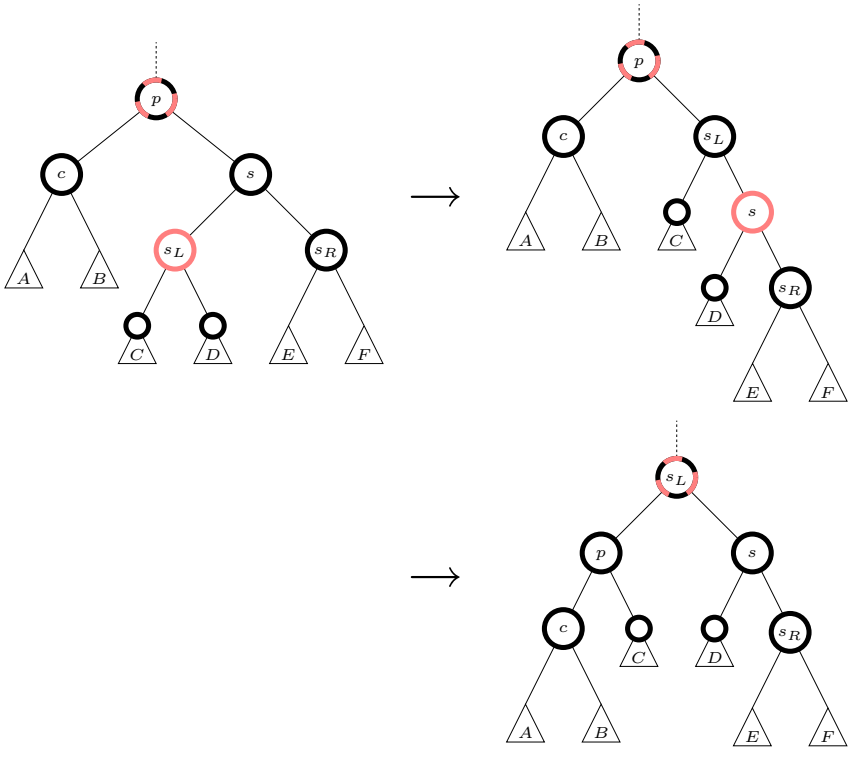
3. Si p est rouge, mais s, s_L et s_R sont noirs, alors il suffit de colorer p en noir et s en rouge :



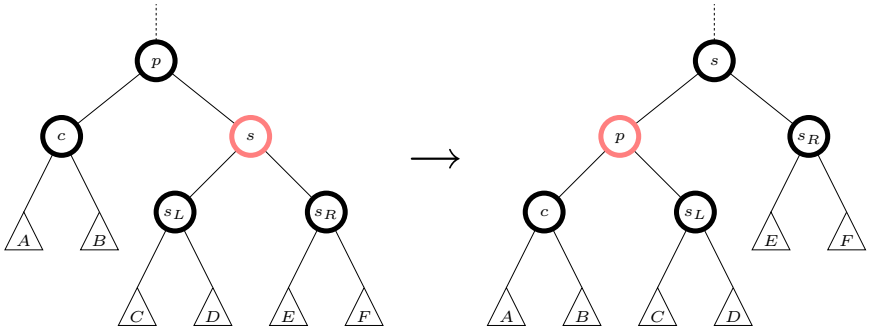
4. Si s est noir, mais s_R est rouge (quelles que soient les couleurs de p et s_L), alors on effectue une rotation gauche, et on colore s de la couleur de p, puis p et s_R en noir :



5. Si s et s_R sont noirs, mais s_L est rouge (quelle que soit la couleur de p), on effectue une double rotation droite gauche, et on colore s_L de la couleur de p, puis p et s en noir (en fait, après la première rotation droite, si on colore s en rouge et s_L en noir, on se ramène au cas précédent) :



6. Enfin, si s est rouge, p, s_L et s_R sont noirs. On peut alors effectuer une rotation gauche, puis colorer p en rouge et s en noir. L'équilibre n'est pas rétabli, mais désormais le parent de c est rouge et son frère est noir. Cela nous ramène à un des trois cas précédents :



7. Ouf, c'est fini !

Remarque : tout le baratin sur ces arbres rouge-noir est honteusement repompé de Wikipedia (`http://en.wikipedia.org/wiki/Red-black_tree`). Allez y faire un tour, c'est vraiment très intéressant.

4.4 Allons-y gaiement

1. Retroussez vos manches, et écrivez un foncteur `SetByRBTree` qui prend un module `A` de signature `ORDERED` et renvoie un module représentant des ensembles d'éléments de type `A.t` en utilisant des arbres rouge-noir.

2. Allez au foyer boire un coup, vous l'avez plus que mérité !