Vincent Simonet (http://cristal.inria.fr/~simonet/)

Arbres rouge et noir

Les arbres binaires de recherche permettent de représenter des ensembles d'éléments pour lesquels on dispose d'une relation d'ordre. Les opérations de base (telles que l'ajout, la recherche ou la suppression d'un élément) peuvent être effectuées en temps linéaire par rapport à la hauteur de l'arbre. Ces opérations sont donc efficaces si les arbres considérés sont bien « équilibrés » : leur temps d'exécution est alors logarithmique par rapport au nombre d'éléments de l'arbre. Cependant, si la hauteur de l'arbre est grande, leur performance peut ne pas être meilleure que pour une liste chaînée (i.e. linéaire en le nombre d'éléments). Les arbres rouge et noir sont une amélioration des arbres binaires de recherche qui assure que les arbres manipulés sont toujours équilibrés.

1 Des arbres binaires de recherche aux arbres rouge et noir

Un *arbre rouge et noir* est un arbre binaire de recherche donc chaque nœud contient une information supplémentaire, sa couleur, qui peut valoir soit *rouge* soit *noir*. En contrôlant la manière dont les nœuds sont colorés sur n'importe quel chemin allant de la racine à une feuille, les arbres rouge et noir garantissent qu'aucun des ces chemins n'est plus de deux fois plus long que n'importe quel autre, ce qui rend l'arbre approximativement équilibré. Nous représenterons en Caml les arbres rouge et noir grâce aux types suivants :

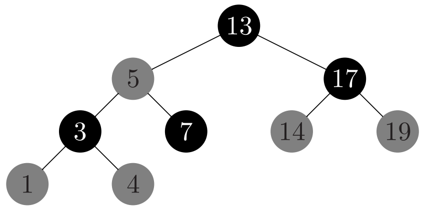
```
type color =
    Red
  | Black
;;

type 'a tree =
    Empty
  | N of color * 'a tree * 'a * 'a tree
;;
```

Un objet de type `'a tree` représente un arbre binaire dont les nœuds sont colorés en rouge ou en noir et portent des étiquettes de type `'a`. On supposera que l'on dispose d'une relation d'ordre sur ces éléments. Un tel arbre est dit *de recherche* si et seulement si pour tout nœud étiqueté par l'élément x, tous les éléments présents dans le sous-arbre gauche sont inférieurs à x et tous ceux du sous-arbre droit sont supérieurs à x. Un arbre binaire de recherche est un arbre rouge et noir s'il satisfait les deux propriétés suivantes :

(1) Si un nœud est rouge alors ses fils sont noirs.

(2) Chaque chemin simple reliant un nœud à une feuille contient le même nombre de nœuds noirs.

Voici un exemple d'arbre rouge et noir :



(les nœuds rouges sont représentés grisés).

▶ **Question 1** *Écrivez une fonction* `mem` *qui cherche si un élément est présent dans un arbre rouge et noir.*

```
val mem: 'a -> 'a tree -> bool
```

Nous allons maintenant écrire quelques fonctions permettant de vérifier qu'un objet de type `'a tree` vérifie bien les propriétés d'un arbre rouge et noir.

▶ **Question 2** *Écrivez une fonction* `check1` *qui vérifie qu'un objet de type* `'a tree` *vérifie la propriété (1), i.e. que tout fils d'un nœud rouge est noir.*

```
val check1: 'a tree -> bool
```

La *hauteur noire* $\mathrm{hn}(t)$ d'un arbre rouge et noir t est le nombre de nœuds noirs présents dans un chemin quelconque descendant de sa racine à n'importe quelle feuille.

▶ **Question 3** *Écrivez une fonction* black_height *qui calcule la hauteur noire d'un objet de type* 'a tree*. Dans le cas où cette hauteur n'est pas définie (parce que l'arbre de vérifie pas la propriété (2)), vous lèverez une exception.*

```
val black_height: 'a tree -> int
```

▶ **Question 4** *Déduisez-en une fonction* check2 *qui vérifie qu'un objet de type* 'a tree *vérifie bien la propriété (2), i.e. que tout chemin simple reliant un nœud à une feuille descendante contient le même nombre de nœuds noirs.*

```
val check2: 'a tree -> bool
```

2 Insertion d'un élément

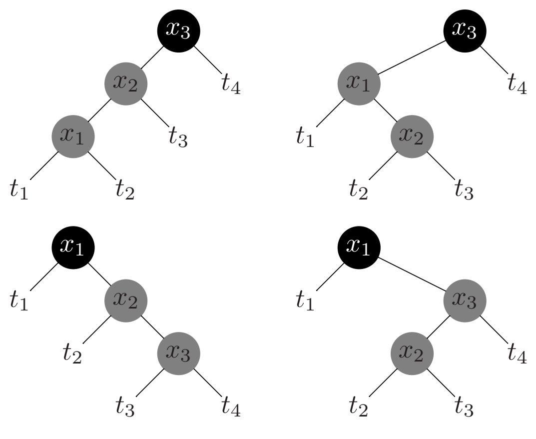
Le principe de l'insertion d'un élément x dans un arbre rouge et noir est le suivant. On effectue tout d'abord une insertion en procédant comme pour un arbre binaire de recherche simple. Un nœud est donc créé à la place d'une feuille, à une position choisie de manière à respecter l'ordre entre éléments. On colorie ce nouveau nœud en *rouge*, ce qui permet de préserver la propriété (2). Ainsi, après cette première étape, seule la propriété (1) peut être violée : il se peut en effet que le père du nœud introduit soit lui aussi rouge ! L'idée est alors de remonter la structure de l'arbre en réarrangeant astucieusement les nœuds et leurs couleurs de manière à remonter ce conflit et à le faire disparaître — tout en maintenant la propriété (2).

▶ **Question 5** *Écrivez une première fonction* simple_insert *qui insère un élément dans un arbre en vous limitant à la première étape décrite ci-dessus, c'est à dire sans vous préoccuper du respect de la propriété (1).*

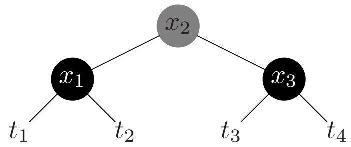
```
val simple_insert: 'a -> 'a tree -> 'a tree
```

2.1 Conflits rouge

Un *conflit rouge* est un arbre binaire de recherche d'une des quatre formes suivantes :



(où les sous-arbres t_1, t_2, t_3 et t_4 sont des arbres rouge et noir de même hauteur noire). Il est possible de « résoudre » n'importe lequel de ces conflits en réécrivant l'arbre sous la forme suivante :



▶ **Question 6** *Vérifiez que si t_1, t_2, t_3 et t_4 sont bien des arbres rouge et noir alors l'arbre obtenu est lui aussi rouge et noir.*

▶ **Question 7** *Écrivez une fonction* conflict *qui prend pour argument un objet de type* 'a tree*. S'il s'agit d'un conflit, votre fonction réarrangera l'arbre comme indiqué ci-dessus. Dans tous les autres cas, elle rendra l'arbre inchangé.*

```
val conflict: 'a tree -> 'a tree
```

2.2 Insertion avec résolution des conflits

Si la racine d'un arbre rouge et noir est rouge, l'arbre obtenu en colorant cette racine en noir est également un arbre rouge et noir. On peut donc toujours se ramener au cas où l'arbre considéré a une racine noire. On supposera ainsi que les arbres manipulés ont une racine noire.

▶ **Question 8** *Écrivez une fonction* black_root *qui colore en noir la racine d'un arbre.*

```
val black_root: 'a tree -> 'a tree
```

▶ **Question 9** *Adaptez votre fonction* simple_insert *de manière à résoudre les conflits lors de la remontée à l'aide de la fonction* conflict*. Vous pourrez supposer que l'arbre initial a une racine noire et veillerez à ce que l'arbre obtenu ait également une racine noire.*

```
val insert: 'a -> 'a tree -> 'a tree
```

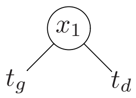
Le coût de l'insertion d'un élément dans un arbre rouge et noir est le même que pour un arbre binaire de recherche : elle est proportionelle à la hauteur de l'abre. En effet, les manipulations effectuées pour résoudre les éventuels conflits sont *locales* et limitées au chemin menant de la racine de l'arbre au nœud ajouté. Elle n'affecte donc la complexité théorique de la fonction que d'un facteur constant.

3 Suppression d'un élément

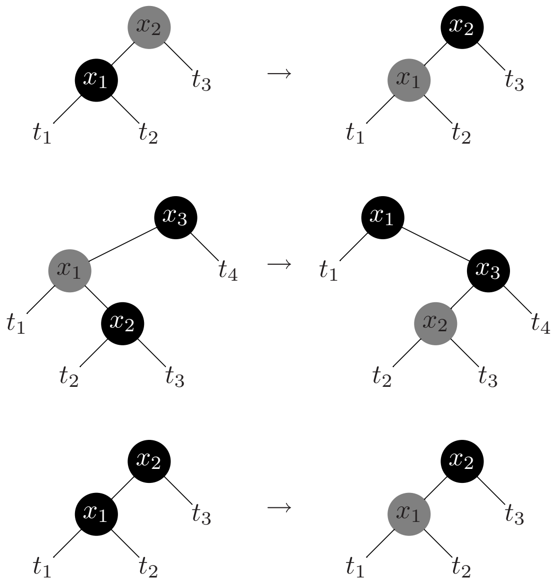
Pour la suppression d'un élément d'un arbre rouge et noir, la même démarche que pour l'insertion peut être adoptée : on commence par effectuer une suppression comme dans un arbre binaire de recherche, puis on essaye de rétablir la structure d'arbre rouge et noir. La suppression dans un arbre binaire de recherche se ramène toujours à la suppression d'une feuille : si l'élément à supprimer est porté par un nœud interne, on permute celui-ci avec le plus petit élément du sous-arbre droit ou le plus grand du sous-arbre gauche. La suppression d'une feuille préserve naturellement la propriété (1). Cependant, elle peut contrarier la propriété (2) en introduisant un *déséquilibre* dans l'arbre si le nœud supprimé est noir.

3.1 Déséquilibres

Un *déséquilibre droit* est un arbre binaire de recherche vérifiant (1) de la forme suivante :



où t_g et t_d sont des arbres rouge et noir tels que $\mathrm{hn}(t_g) = 1 + \mathrm{hn}(t_d)$.
On peut en fait distinguer trois cas de déséquilibres droits suivant la couleur des nœuds. Chacun d'entre-eux peut être corrigé en réarrangeant localement la structure de l'arbre :



Dans chacun des cas, l'arbre obtenu suite au réarrangement peut comporter un *conflit rouge* qu'il convient de résoudre comme dans la partie 2.1. De plus, dans les deux premiers cas, la hauteur noire de l'arbre obtenu correspond à celle du sous-arbre gauche, on dit que le déséquilibre est *absorbé*. Dans le troisième cas, elle correspond à celle du sous-arbre droit : le déséquilibre est *propagé*.

▶ **Question 10** *Écrivez une fonction* unbalanced_right *qui corrige un déséquilibre droit. Votre fonction rendra une paire formée de l'arbre obtenu et d'un booléen indiquant si le déséquilibre est propagé.*

```
unbalanced_right: 'a tree -> 'a tree * bool
```

On définit de manière symétrique la notion de *déséquilibre gauche.*

▶ **Question 11** *Écrivez une fonction* unbalanced_left *qui corrige un déséquilibre gauche. Votre fonction rendra une paire formée de l'arbre obtenu et d'un booléen indiquant si le déséquilibre est propagé.*

```
unbalanced_left: 'a tree -> 'a tree * bool
```

3.2 Suppression

▶ **Question 12** *Écrivez deux fonction* min_tree *et* max_tree *calculant respectivement le plus petit et le plus grand élément d'un arbre rouge et noir.*

```
val min_tree: 'a tree -> 'a
val max_tree: 'a tree -> 'a
```

▶ **Question 13** *Écrivez une fonction* remove *qui supprime un élément d'un arbre rouge et noir.*

```
val remove: 'a -> 'a tree -> 'a tree
```

Vous pourrez pour cela écrire une fonction auxilliaire remove_aux prenant les mêmes arguments mais retournant un couple. La première composante de ce couple sera le nouvel arbre obtenu, la deuxième composante de ce couple sera un booléen indiquant si la suppression a réduit la hauteur noire de l'arbre d'une unité ou non, i.e. si un déséquilibre a été introduit. Vous résoudrez les déséquilibres lors de la « *remonté* », à l'aide des fonctions unbalanced_left et unbalanced_right.

▶ **MP & MP* – Option Informatique**
Année 2002–2003, Quatrième TP Caml
Vincent Simonet (http://cristal.inria.fr/~simonet/)

Arbres rouge et noir

Un corrigé

▶ **Question 1** La recherche dans un arbre rouge et noir s'effectue comme dans un arbre binaire de recherche, les couleurs n'intervenant pas.

```
let rec mem x = function
    Empty -> false
  | N (_, t1, y, t2) ->
      x = y || mem x (if x < y then t1 else t2)
;;
```

▶ **Question 2** On recherche la présence de deux nœuds rouges successifs à la racine grâce à un *pattern-matching*. Si le motif n'est pas trouvé, on poursuit récursivement la recherche dans les deux sous-arbres.

```
let rec check1 = function
    Empty -> true
  | N (Red, N (Red, _, _, _), _, _)
  | N (Red, _, _, N (Red, _, _, _)) -> false
  | N (_, t1, _, t2) -> check1 t1 && check1 t2
;;
```

▶ **Question 3** A chaque noeud, on calcule la hauteur noire du sous-arbre gauche. On vérifie ensuite qu'elle est égale à celle du sous-arbre droit. Enfin on calcule la hauteur noire de l'arbre entier en fonction de la couleur de la racine.

```
exception Black_height;;

let rec black_height = function
    Empty -> 0
  | N (c, t1, _, t2) ->
      let h = black_height t1 in
      if h <> (black_height t2) then raise Black_height;
      if c = Black then 1 + h else h
;;
```

▶ **Question 4** On se contente de faire appel à la fonction `black_height` et de rattraper une éventuelle exception.

```
let check2 t =
  try
    let _ = black_height t in
    true
  with
    Black_height -> false
;;
```

▶ **Question 5**

```
let rec simple_insert x = function
      Empty -> N (Red, Empty, x, Empty)
    | N (c, t1, y, t2) ->
        if x < y then
          N (c, simple_insert x t1, y, t2)
        else
          N (c, t1, y, simple_insert x t2)
  ;;
```

▶ **Question 7** On détecte les quatres cas de conflits à l'aide d'un *pattern-matching* sur la structure de l'arbre.

```
let conflict = function
    N(Black, N(Red, N(Red, t1, x1, t2), x2, t3), x3, t4) ->
      N(Red, N(Black, t1, x1, t2), x2, N(Black, t3, x3, t4))
  | N(Black, N(Red, t1, x1, N(Red, t2, x2, t3)), x3, t4) ->
      N(Red, N(Black, t1, x1, t2), x2, N(Black, t3, x3, t4))
  | N(Black, t1, x1, N(Red, N(Red, t2, x2, t3), x3, t4)) ->
      N(Red, N(Black, t1, x1, t2), x2, N(Black, t3, x3, t4))
  | N(Black, t1, x1, N(Red, t2, x2, N(Red, t3, x3, t4))) ->
      N(Red, N(Black, t1, x1, t2), x2, N(Black, t3, x3, t4))
  | t -> t
;;
```

▶ **Question 8**

```
let black_root = function
    Empty -> Empty
  | N (_, t1, y, t2) -> N(Black, t1, y, t2)
;;
```

▶ **Question 9** La fonction `insert_aux` effectue l'insertion comme `simple_insert`, mais traite les conflits lors de la remonté à l'aide de la fonction `conflict`. La fonction `insert` effectue un appel à `insert_aux` et s'assure que la racine de l'arbre obtenu est bien noire.

```
let rec insert_aux x = function
    Empty -> N (Red, Empty, x, Empty)
  | N (c, t1, y, t2) ->
      if x < y then
        conflict (N (c, insert_aux x t1, y, t2))
      else
        conflict (N (c, t1, y, insert_aux x t2))
;;

let insert x t =
  black_root (insert_aux x t)
;;
```

▶ **Question 10** On détecte les trois cas de déséquilibre à l'aide d'un *pattern-matching* sur la structure de l'arbre. Dans les trois cas, on applique conflict à l'arbre obtenu de manière à résoudre un éventuel conflit rouge. résultant du réarrangement.

```
let unbalanced_right = function
    N (Red, N (Black, t1, x1, t2), x2, t3) ->
      conflict (N (Black, N (Red, t1, x1, t2), x2, t3)),
      false
  | N (Black, N (Red, t1, x1, N (Black, t2, x2, t3)), x3, t4) ->
      conflict (N (Black, t1, x1, N (Black, N (Red, t2, x2, t3), x3, t4))),
      false
  | N(Black, N (Black, t1, x1, t2), x2, t3) ->
      conflict (N (Black, N (Red, t1, x1, t2), x2, t3)),
      true
  | t ->
      t, false
;;
```

▶ **Question 11**

```
let unbalanced_left = function
    N (Red, t1, x1, N (Black, t2, x2, t3)) ->
      conflict (N (Black, t1, x1, N (Red, t2, x2, t3))),
      false
  | N (Black, t1, x1, N (Red, N (Black, t2, x2, t3), x3, t4)) ->
      conflict (N (Black, N (Black, t1, x1, N (Red, t2, x2, t3)), x3, t4)),
      false
  | N (Black, t1, x1, N (Black, t2, x2, t3)) ->
      conflict (N (Black, t1, x1, N (Red, t2, x2, t3))),
      true
  | t ->
      t, false
;;
```

▶ **Question 12**

```
let rec min_tree = function
    Empty -> invalid_arg "min_tree"
  | N (_, Empty, x, _) -> x
  | N (_, t, _, _) -> min_tree t
;;

let rec max_tree = function
    Empty -> invalid_arg "max_tree"
  | N (_, _, x, Empty) -> x
  | N (_, _, _, t) -> max_tree t
;;
```

▶ **Question 13** Nous écrivons tout d'abord une fonction auxilliaire remove_aux qui effectue la suppression. Elle retourne une paire formée de l'arbre obtenu et d'un booléen indiquant si la suppression a réduit la hauteur noire de l'arbre d'une unité ou non.

```
let rec remove_aux x = function
    Empty -> Empty, false
  | N (Red, Empty, y, Empty) when x = y -> Empty, false
  | N (Black, Empty, y, Empty) when x = y -> Empty, true
  | N (c, Empty, y, t2) when x = y ->
      let y' = min_tree t2 in
      let t2', o = remove_aux y' t2 in
      let t' = N (c, Empty, y', t2') in
      if o then unbalanced_right t' else (conflict t', false)
  | N (c, t1, y, t2) when x = y ->
      let y' = max_tree t1 in
      let t1', o = remove_aux y' t1 in
      let t' = N (c, t1', y', t2) in
      if o then unbalanced_left t' else (conflict t', false)
  | N (c, t1, y, t2) when x < y ->
      let t1', o = remove_aux x t1 in
      let t' = N (c, t1', y, t2) in
      if o then unbalanced_left t' else (conflict t', false)
  | N (c, t1, y, t2) (* when x > y *) ->
      let t2', o = remove_aux x t2 in
      let t' = N (c, t1, y, t2') in
      if o then unbalanced_right t' else (conflict t', false)
;;
```

On en déduit la fonction de suppression.

```
let remove x t =
  let t', _ = remove_aux x t in
  black_root t'
;;
```