Travaux pratiques #4 - Arbres et compagnie

L'objectif (final) de ce TP est de programmer une classe de gestion d'**arbres rouge et noir**. Ces arbres sont des arbres binaires de recherche possédant des propriétés additionnelles. Ils sont notamment utilisés (eux ou une version dérivée) en programmation fonctionnelle, géométrie algorithmique ou encore dans un planificateur de tâches intégré dans les noyaux Linux actuels.

Les arbres rouge et noir **dérivant** de structures d'arbre plus générales, ils vont être implémentées en utilisant les principes fondamentaux de la Programamtion Orientée Objet.

Arbre binaire...

Un **arbre binaire** est une structure de données représentée par une racine, où chaque noeud possède *au plus* deux fils. Les noeuds possèdent également une valeur, qui sera considérée entière dans ce TP.

Implémenter une classe permettant de gérer des arbres binaires d'entiers, avec les opérations suivantes :

- insertion d'un élément (le choix d'implémentation est laissé libre)
- recherche d'un élément
- suppression d'un élément
- parcours de l'arbre

Les noeuds sont-ils des objets, ou peuvent-ils être représentés autrement ?

... de recherche...

Les arbres binaires **de recherche** sont des arbres binaires dont les clés appartiennent à un ensemble **totalement ordonné**. Plus précisément, pour un noeud donné N de clé c, tout noeud du sous-arbre gauche enraciné en N a une clé inférieure ou égale à c, alors que tout noeud du sous-arbre droit enraciné en N a une clé strictement plus grande que c.

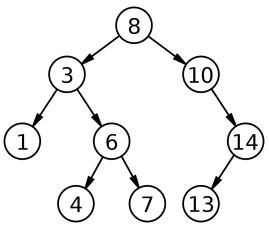


FIGURE 1 – Un exemple d'arbre binaire de recherche. © Wikipedia [1].

Implémenter une classe permettant de gérer des arbres binaires de recherche d'entiers, avec les mêmes opérations que les arbres binaires.

Quelles méthodes doivent changer, et quelles méthodes peuvent être utilisées telles quelles ?

1. Comme la totalité des images -et certaines explications- de ce sujet de TP

... rouge et noir.

Les arbres **rouge et noir** sont des arbres binaires de recherche possédant quelques propriétés supplémentaires. Leur nom vient du fait que, en plus de leur valeur entière et de la connaissance de leurs fils gauche et droit, les noeuds possèdent maintenant un attribut supplémentaire : leur **couleur**.
Les propriétés d'un arbre rouge et noir sont les suivantes :

Propriété 1. un noeud est soit rouge soit noir ;

Propriété 2. la racine est noire ;

Propriété 3. toutes les feuilles sont noires et `NULL` (voir Figure 2) ;

Propriété 4. le parent d'un noeud rouge est noir ;

Propriété 5. le chemin de chaque feuille à la racine contient le même nombre de noeuds noirs.

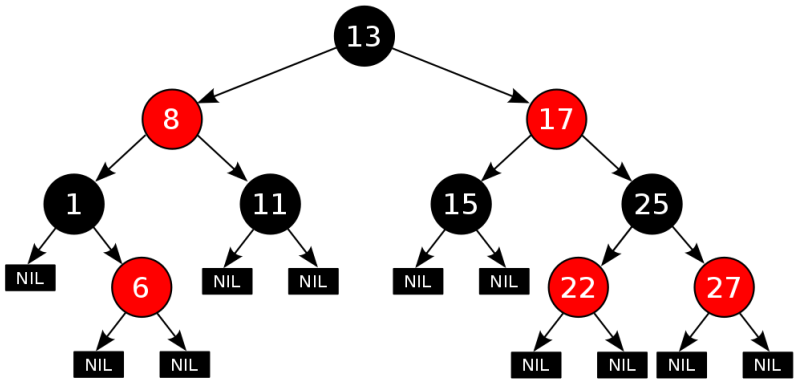


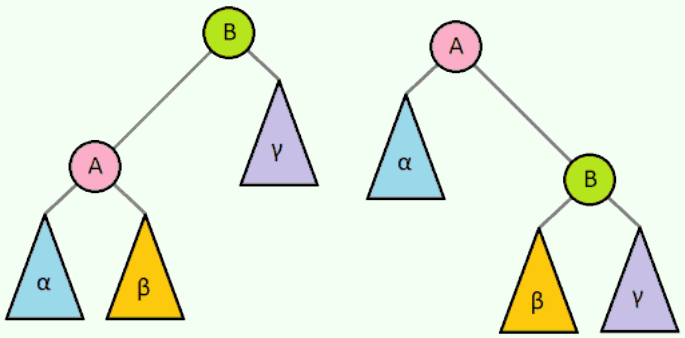
FIGURE 2 – Un exemple d'arbre binaire de recherche. © Wikipedia.

Afin de manipuler plus facilement les arbres rouge et noir, il est conseillé d'implémenter de simples fonctions de liens de *parenté* entre noeuds :

- `parent(n)` : retourne le noeud parent de `n` (s'il existe)
- `grandparent(n)` : retourne le noeud grand-parent de `n` (s'il existe)
- `oncle(n)` : retourne le noeud oncle de `n` (s'il existe)
- `frere(n)` : retourne le frère de `n` (s'il existe)

ainsi que des fonctions de **rotations**, qui permettent elles de maintenir un arbre binaire de recherche équilibré. Ces fonctions s'exécutent en temps constant, et sont de simples changements de parentalité sur les noeuds.

Rotations droite (→) et gauche (←)



Les méthodes définies précédemment doivent à présent être redéfinies. L'insertion et la suppression [2] demandent notamment beaucoup plus d'attention, dans la mesure où le résultat de cette opération doit maintenir un arbre rouge et noir. Pour l'insertion, différentes étapes sont nécessaires :

```
/* type */ inserer(/* parametres */) {
// Insertion d'un nouveau noeud dans l'arbre
inserer(racine, n);

// Reparation de l'arbre au cas ou les proprietes rouge-noir seraient violees
reparer(n);

// Recherche de la nouvelle racine a renvoyer
racine = n;

while (parent(racine) != NULL)
  racine = parent(racine);

  return n;
}
```

Étape #1 - `inserer` La première partie de l'insertion consiste à insérer le noeud en respectant l'arbre binaire de recherche. Cette insertion peut cela dit violer les propriétés d'arbre rouge et noir, et il faut donc réparer l'arbre obtenu en fonction de la position du noeud **courant**.

Étape #2 - `reparer` Il s'agit de la partie la plus délicate, qui demande d'étudier de nombreux cas :

Cas 1. le parent de n est `NULL`

Cas 2. le parent de n est noir

Cas 3. le parent de n est noir et son oncle est rouge

Cas 4. le parent de n est rouge et son oncle est noir

La procédure de réparation de l'arbre se présente donc comme suit :

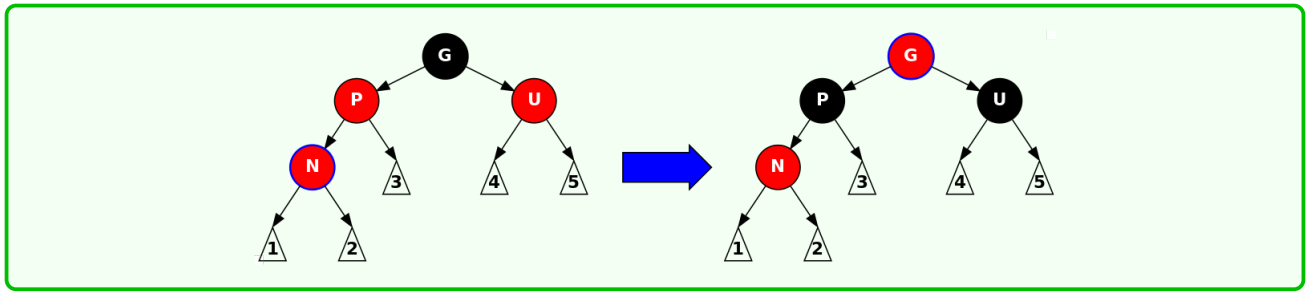
```
/* type */ reparer(/* parametres */) {
  if (parent(n) == NULL)
    insertion_cas1(n);
  else if (parent(n)->couleur == NOIR)
    insertion_cas2(n);
  else if (oncle(n)->couleur == ROUGE)
    insertion_cas3(n);
  else
    insertion_cas4(n);
}
```

2. Laissée en exercice libre.

Les différentes opérations à réaliser en fonction des cas présentés sont maintenant expliquées. Dans tous les cas, N représente la position du noeud **courant** (qui peut être différent du noeud inséré).

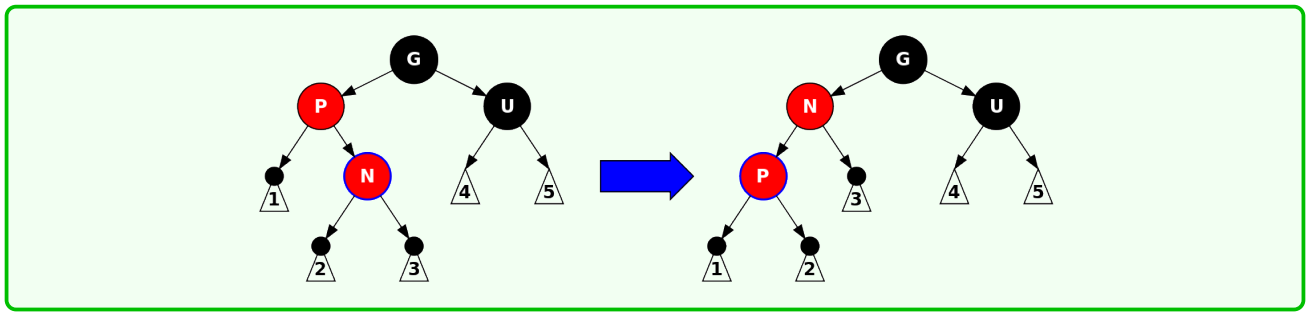
Cas 1 et 2. Le noeud se retrouve à la racine de l'arbre et doit être coloré noir. Cette opération suffit à obtenir un arbre rouge et noir. Dans le Cas 2, il n'y a rien à faire.

Cas 3. Le parent **et** l'oncle sont rouge, ils peuvent être tous deux recolorés en noir, et le grand-parent G devient rouge. Ainsi, la Propriété 5 est toujours vérifiée. En revanche, la Propriété 2 ou la Propriété 4 peuvent ne plus être vérifiées. Il est donc nécessaire d'appeler à nouveau la procédure `reparer` sur le noeud grand-parent.



Cas 4. Le dernier cas (le parent est rouge mais l'oncle est noir) est le plus délicat à gérer. L'objectif dans ce cas est d'effectuer une rotation pour amener le noeud courant N à la place du grand-parent. Cette rotation peut néanmoins fausser certaines propriétés, notamment si N est *à l'intérieur* du sous-arbre du grand-parent G, c'est-à-dire fils droit du fils gauche de G (resp. fils gauche du fils droit de G).

Dans un premier temps, il est nécessaire d'effectuer une rotation gauche (resp. droite) sur le noeud parent P de N. À la fin de cette opération, le noeud courant devient le fils gauche (resp. droit) de N et la procédure `etape_2` est appelée.



La procédure `etape_2` effectue une rotation droite (resp. gauche) sur G, obtenant un argre *quasiment* rouge et noir. En inversant les couleurs de P et G, l'arbre résultant est bien rouge et noir.