TP 7 et 8 - Le rouge et le noir

Dans le cadre de l'UE, vous êtes amenés à programmer en **C++** en utilisant les références pour réaliser vos passages de paramètres, les constructeurs, les destructeurs et les surcharges de l'opérateur d'affectation. Néanmoins, il sera important que vous sachiez toujours basculer dans un autre langage de programmation. Pour cela, il est important que vous établissiez votre raisonnement au niveau du pseudo-langage algorithmique et en utilisant les spécificités du langage utilisé seulement quand vous effectuez la mise en œuvre.

1 A la recherche d'un équilibre

Les arbres rouge et noir constituent des arbres binaires de recherche semi-équilibrés. Chaque nœud d'un arbre rouge et noir comporte une information supplémentaire dénotant sa *couleur* : soit rouge, soit noire. L'information de couleur est portée par les nœuds, mais elle peut être étendue aux arbres et aux sous-arbres, en adoptant les conventions suivante :

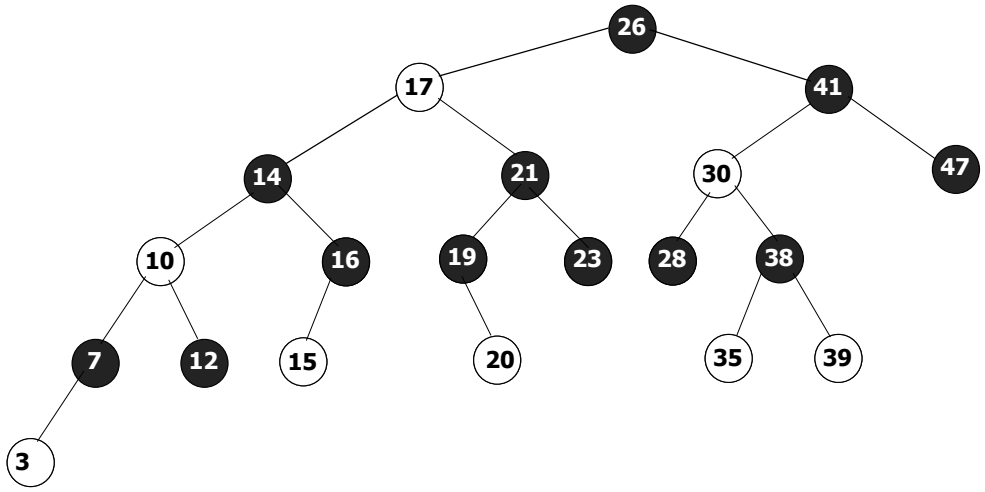
— un arbre ou un sous-arbre vide est noir,
— un arbre ou un sous-arbre non vide a la couleur de son nœud racine.

Les arbres rouge et noir sont dits semi-équilibrés au sens où le chemin de la racine à une feuille ne peut être 2 fois plus long que le chemin à une autre feuille.

Cette caractéristique des arbres rouge et noir est garantie grâce aux propriétés suivantes :

1. un nœud rouge ne peut pas avoir de fils rouge ;
2. les chemins [1] reliant un nœud aux feuilles descendantes de ce nœud doivent tous contenir le même nombre de nœuds noirs.

Le nombre de nœuds noirs entre la racine et une feuille est aussi appelé **hauteur noire** de l'arbre. Pour des considérations pratiques, nous imposerons de plus que le nœud racine d'un arbre non vide soit noir (ce qui n'est évidemment pas forcément le cas pour un sous-arbre).



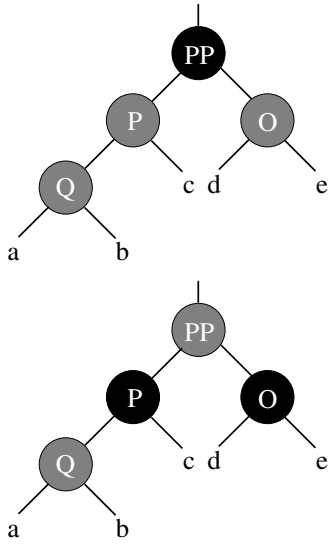
1. Par chemin, il faut ici entendre *chemin simple*, c'est-à-dire un chemin qui ne contient pas deux fois le même nœud.

1.1 Insertion d'un élément

L'insertion d'un nouvel élément dans un arbre rouge et noir se décompose en plusieurs étapes. Au départ, l'élément est inséré aux feuilles, comme dans n'importe quel arbre binaire de recherche, puis les propriétés des arbres rouges et noirs sont restaurées grâce à la modification des couleurs de certains nœuds et grâce à des opérations de rotation au niveau de certains nœuds de l'arbre.

Au moment de l'insertion d'un nouvel élément dans un arbre rouge et noir non vide, la couleur de la feuille nouvellement créée est rouge. Le nombre de nœuds noirs entre la racine et les feuilles demeure donc constant, mais la propriété 1 n'est plus forcément assurée : on peut se retrouver avec un nœud rouge dont le père est rouge. Le rééquilibrage consiste alors à rétablir la situation localement, quitte à propager la violation de la propriété 1 vers le haut de l'arbre, jusqu'à aboutir à une situation où la résorption locale la fasse totalement disparaitre (ne pas oublier que la racine de l'arbre est noire !).

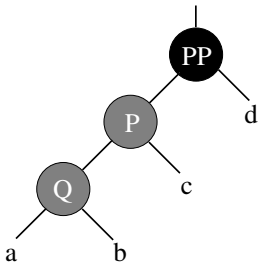
1. Si le nœud `Q` possède un oncle `O` rouge, le rééquilibrage local se fait de la manière suivante :

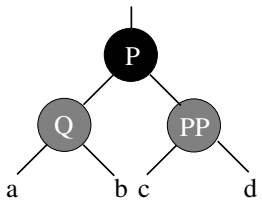


   On change la couleur du père `P` et de l'oncle `O` qui deviennent noirs, ainsi que celle du nœud grand-père `PP` qui devient rouge. Attention, dans ce cas, on résout le problème localement par changement de couleur, mais le conflit peut être remonté dans l'arbre si le père de `PP` est lui-même rouge. Le conflit va alors continuer à se propager de la même manière, jusqu'à rencontrer un oncle ou un arrière grand-père noir ou la racine. **Si le nœud `PP` est la racine, il est recolorié en noir à la fin (cela augmente de 1 la hauteur noire des chemins issus de la racine)**.

2. Si le nœud `Q` ne possède pas d'oncle rouge, on a plusieurs cas de figures suivant que `Q` est un fils gauche ou un fils droit et suivant que le père `P` est lui même un fils gauche ou droit de `PP` (nœud grand-père de `Q`).

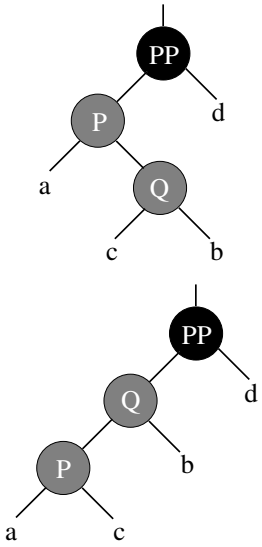
   (a) Si `Q` est le fils gauche de `P` qui est lui-même le fils gauche de `PP`, le rééquilibrage se fait selon l'opération de rotation suivante :





On dit que l'on fait une rotation droite de l'arbre au niveau du nœud `PP` (la même opération que l'on a découvert avec les AVL mais sans se soucier des questions de hauteur de sous-arbre). D'autre part, les couleurs de `P` et de `PP` sont échangées. Après cette opération de rééquilibrage local, toutes les propriétés des arbres rouge et noirs sont rétablies, et l'opération d'insertion est donc terminée.

(b) Si `Q` est le fils droit de `P` qui est cette fois-ci le fils gauche de `PP`, le rééquilibrage se fait par l'opération de rotation suivante, sans modifier la couleur des nœuds :



On dit que l'on fait une rotation gauche au niveau du nœud `P` et on aboutit à une situation similaire à celle du cas traité précédemment. Au final on se retrouve donc à faire une rotation double à droite.

(c) Les 2 derniers cas se déduisent des 2 précédents par symétrie.

Attention, lors de l'implantation, il est préférable d'identifier les différents cas nécessitant une réorganisation, depuis le nœud grand-père (du nœud rouge qui a lui même un père rouge). La structure de données ne permet pas facilement de remonter d'un nœud vers ses ancêtres, en revanche, il est possible de recourir à une procédure interne récursive s'appliquant à un nœud. Il est alors possible de profiter de la remontée récursive. Lorsqu'on sort d'un appel récursif sur un nœud fils, on se retrouve dans le corps de l'appel sur le nœud père.

2 Travail à réaliser

2.1 Module Collection

On se propose de réaliser un module `Collection` d'`Element`s implantée sous forme d'`Arbre Rouge et Noir`. Vous veillerez à ce que ce module `Collection` soit bien modulaire, c'est à dire que votre module `Collection` devra être indépendant de la nature des `Element`s stockés. Vous vous limiterez aux opérations d'initialisation (constructeur) et de testament (destructeur) d'une collection, ainsi qu'aux opérations d'insertion et de recherche d'un élément, sans aborder la question de sa suppression. Vous offrirez également une procédure de visualisation de l'état interne de

l'arbre modélisant une collection. Cette procédure pourra appeler la procédure interne d'affichage d'un sous-arbre.

```
void ARN::affichageSsArbre(const Noeud *p, int prof)
{
  if (p!=0)
  {
        this->affichageSsArbre(p->d, prof+1);
affichage de prof espaces;
        affichageCouleur(p->info,p->couleur);
        this->affichageSsArbre(a,p->g, prof+1);
  }
}
```

Réaliser un second module `Collection` implantée cette fois-ci sous forme d'`Arbre Binaire de Recherche` (non équilibré).

Etablir un profil du temps d'insertion dans une `Collection` implantée sous forme d'un `Arbre Binaire de Recherche` non équilibré, ainsi que le profil du temps d'insertion dans une collection implantée sous forme d'un `Arbre Rouge et Noir`. Ce profil sera une fonction du nombre `n` d'éléments dans la Collection au moment de l'insertion. Vous procéderez de manière similaire aux statistiques établies sur les SkipLists et les Tables de Hachage, à savoir que vous prendrez un nombre `NBC` de collections initialement vides, et que vous insérerez des éléments aléatoires (donc potentiellement différents) dans chacune des collections. Le temps d'insertion dans une Collection contenant `n` éléments s'obtient par moyenne sur les `NBC` collections de taille `n`.

3 Conditions de rendu

Vous déposerez une archive `Nom1_Nom2.tgz` (`Nom1` et `Nom2` étant les noms des 2 étudiants composant le binôme) de votre travail sur Tomuss avant le **mardi 23 novembre** à 23h (le dépôt sera ensuite fermé). **Les deux membres du binôme DOIVENT APPARTENIR AU MEME GROUPE DE TP et il est interdit de se greffer au travail d'un étudiant dans un autre groupe**.

Votre archive doit contenir un répertoire `Nom1_Nom2` avec les sources de votre application. Les sources sont bien évidemment tous les fichiers `.cpp` et `.hpp`, mais aussi le fichier `Makefile`, ainsi que tout autre fichier (`README`, etc.) utile à la compréhension de votre programme, ainsi qu'à sa compilation et à son exécution.

Le fichier `Makefile` doit modéliser clairement les dépendances entre les fichiers et permettre une recompilation partielle, en cas de modification partielle.

Attention !

- Votre archive NE DOIT PAS contenir de fichiers objets (.o) ni d'exécutable.
- Le travail rendu doit être issu de la collaboration entre 2 personnes et toute récupération flagrante du code d'autrui sera sanctionnée.
- Votre programme doit pouvoir être compilé sans erreur ni avertissement avec `g++` sans nécessiter l'installation de bibliothèques supplémentaires sur une des machines de la salle de TP.
- une attention particulière devra être consacrée à la mise en œuvre d'une programmation modulaire débouchant sur la proposition de véritables types abstraits.
- l'interface de vos modules doit offrir toutes les informations utiles à l'utilisation des types et des fonctionnalités offertes.
- soignez bien votre programme principal (`main.cpp`) qui doit illustrer que les opérations à coder ont été faites correctement.

Le **mercredi 24 novembre**, vous serez amenés à faire une petite présentation au chargé de TP au cours de laquelle des questions seront posées à chacun des 2 membres du binôme. Cette

présentation sera précédée d'une mini-interrogation sur papier portant sur le travail que vous aurez rendu la veille.

Profitez du mercredi 17 novembre après-midi pour avancer sur votre TP avec votre binôme. Il n'y aura alors pas de séance de TP encadrée pour que vous puissiez avancer sur votre travail.