Structures Intermédiaires
TP02

Exercice 1.)

1.1.)

Un passager contient un numéro de client un nom, prénom, âge et numéro de siège sur un vol. On utilise donc un Type TPassager qui contient un TPersonne ayant un int numéro de client et un int numéro de siège. Le prénom et nom étant stocké dans le TPersonne.

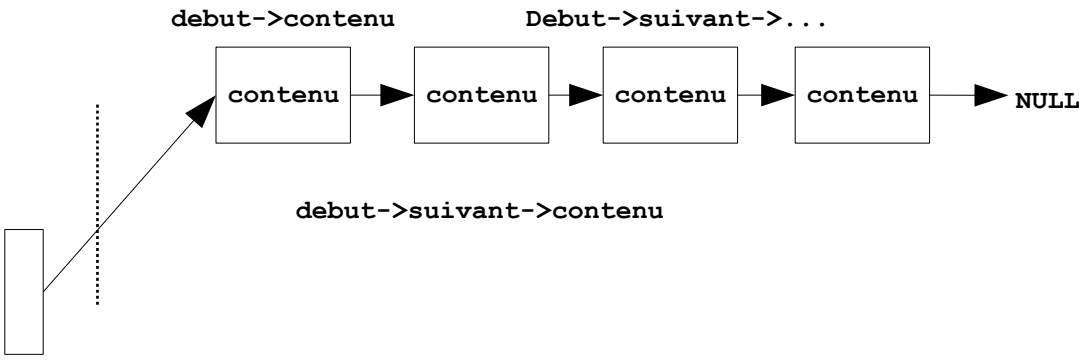
1.2.)

Une liste chaînée est une structure linéaire récursive qui fonctionne de la façon suivante : on initialise un pointeur pointant sur un TContenu qui contient un pointeur sur un autre contenu et un TPassager.

1.3.)

Pour l'implémentation on se servira du fichier « passagers.txt » en le lisant ligne par ligne afin de remplir notre liste chaînée. Le code nous permettant de lire le fichier texte à été récupéré sur tutorialspoint[1] et adapté à mes besoins. J'utilise le compitateur GDB[2], qui fonctionne sans problème pour mon code, attention selon le compilateur utilisé ça peut ne pas marcher.

L'implémentation est en pièce jointe, voici un schéma rappelant le principe d'une liste chaînée :



1 https://www.tutorialspoint.com/c_standard_library/c_function_fscanf.htm

2 https://www.onlinegdb.com/

Exercice 2.)

2.1.)

Étant donné que c'est le sujet de cette semaine, nous allons utiliser une structure d'**arbre de tri binaire**. Il faudra donc implémenter une insertion qui respecte l'ordre de tri. L'implémentation est très technique, j'ai en effet eu besoin d'utiliser des pointeurs vers des pointeurs, qui utilisent une syntaxe compliquée lorsque l'on est pas habitués, et le pseudo-code du cours porte à confusion, car il ne précise pas ce fait. Je me permetterait donc de mettre du C dans certains des schémas UML, afin que la transition implémentation-rapport soit plus douce.

La fiche technique d'un arbre, dans le cadre de ce TP, se résume à la structure suivante :

But:
Stocker des informations conservant une hiérachie.
Création:
On spécifie un type de base L'arbre est initialement vide
Modification:
Insérer un élément
Accès:
Rechercher un élément et le renvoyer.

Étant donné que c'est un arbre binaire, chaque nœud aura deux descendants, gauche et droite, qui seront des pointeurs sur un nœud. C'est donc une définition récusive, le pseudo-code d'un nœud est le suivant :

```
TNoeud struct
     TPassager : contenu
     ^Tnoeud : droite
     ^Tnoeud : gauche
```

2.2.)

Une clef de tri, dans le cadre des arbres de tri, est une **propriété** du type de base de la collection d'éléments que l'on veut trier sur laquelle on peut établir une **relation d'ordre.** La **politique d'insertion** dans un arbre de tri est la suivante :

Si la clef de l'élément à insérer est plus petite que l'élément sur lequel on se trouve, on insère à gauche. Sinon, on insère à droite.

Cette politique est fondamentale à suivre, car si l'on veut parcourir un arbre de tri en utilisant un parcours en ordre, de façon à parcourir les éléments en ordre selon leur clef de tri, il faut qu'ils aient étés insérés suivant celle-ci.

La clef de tri à utiliser est bien entendu le numéro de siège, étant donné que l'on trie les gens en fonction de celui-ci dans le cadre de ce TP.

2.3.)

Je vais détailler comment fonctionne mon implémentation au niveau des pointeurs, afin de lever toute possible confusion sur le sujet. Dans la fonction **main** j'initialise un pointeur racine, en utilisant :

```
struct Tnoeud *racine = NULL;
```

Ce pointeur pointe sur du nul initialement, mais il est lui même stocké dans la mémoire à une certaine addresse, à laquelle on peut accéder via &racine. La fonction insert doit pouvoir avoir accès au pointeur racine et allouer les membres gauche et droite de la valeur que racine pointe, afin de construire l'arbre, je passe donc en paramètre l'addresse du pointeur racine :

```
insert(current, &racine); //Ou current est le passager à insérer
```

La signature de insert sera donc :

```
insert(struct TPassager newELement, struct Tnoeud **courant);
```

Car, &racine est une addresse qui indique ou se trouve un pointeur, c'est donc un ***pointeur vers un pointeur*** et la signature requiert donc **TNoeud **courant.** Dans insert, pour accéder aux valeurs pointées, il faut faire du **déférencage**, c'est à dire accéder aux valeurs pointées, cela se fait en plaçant une * à **GAUCHE** du pointeur. Il faut également rajouter des parenthèses, car les → indiquant le

champ de la structure pointée ont priorité sur les étoiles, on écrira donc :

(*courant)→contenu = newElement;

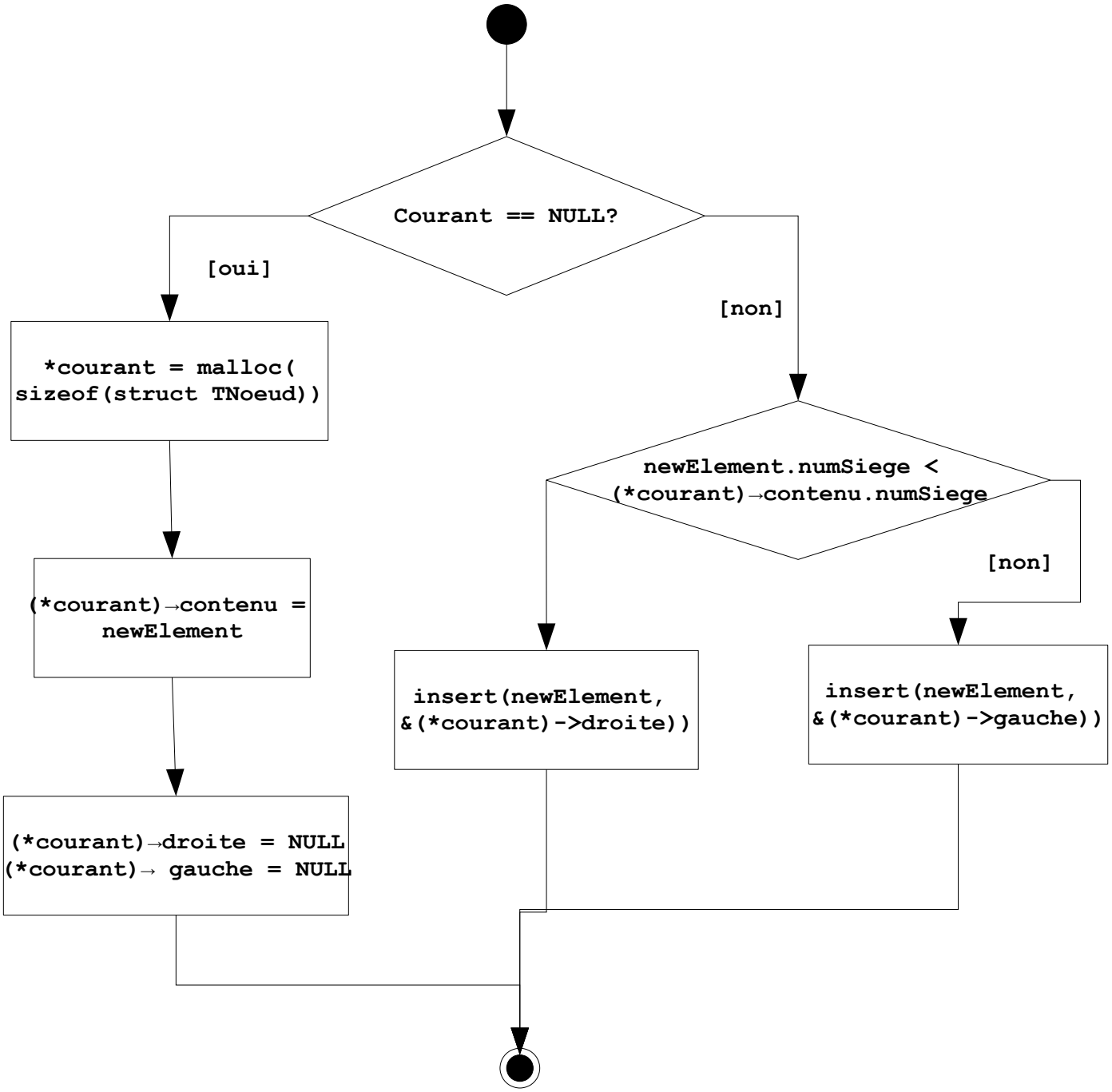
pour, par exemple assigner une valeur au contenu. En récapitulation : (*courant) est un déférencage de &addresse qui est une addresse sur un pointeur, (*courant) est donc un pointeur sur un Tnoeud, on se sert donc de → pour accéder aux membres de la structure pointée. Finalement, lorsque l'on veut insérer un nouveaux passager mais que l'on est sur un pointeur non NULL, il faut parcourir soit à gauche soit à droite suivant la politique d'insertion. Après avoir comparé les valeurs de numéro de siège, on appele insert avec (si on insère à droite) :

insert(current, &(*courant)->droite);

(*courant) est un pointeur qui pointe sur le nœud courant (sur lequel on se trouve), courant→droite est le pointeur droit du nœud courant qui pointe sur un autre nœud alloué ou non, alors &(*courant)→droite est donc l'addresse en mémoire du pointeur de droite du nœud courant.

Voici l'UML tant attendu, avec le pseudo-code écrit directement en C.

void insert(struct TPassager newElement, struct TNoeud **courant)



2.4.)

L'implémentation est en pièce jointe.

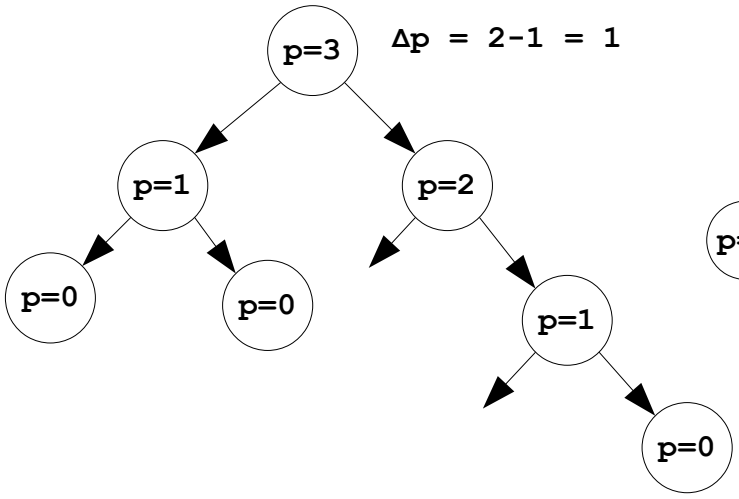
Exercice 3.)

3.1.)

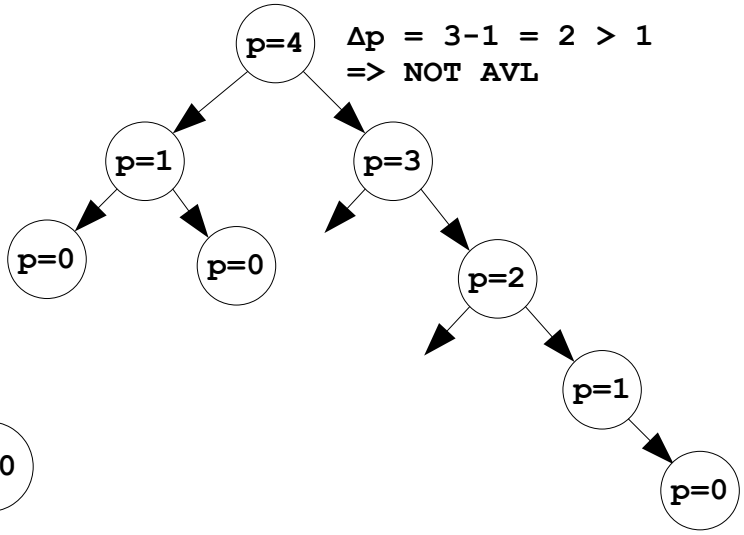
Un arbre AVL est un arbre de tri tel que la profondeur des sous-arbres d'un nœud diffère au plus de 1. Dans le cas d'un arbre binaire, c'est un arbre tel que pour tout nœud le sous-arbre droit et le sous-arbre gauche de ce nœud ont une profondeur qui diffère au plus de 1.

3.2.)

Arbre AVL



Arbre NON-AVL



3.3.)

Le facteur d'équilibrage est la condition qui permet de savoir si l'arbre est équilibré ou pas, c'est à dire, la différence de profondeur des sous-arbres d'un nœud doit être plus petite ou égale à 1. Le rééquilibrage sert, lorsque suite à une insertion on viole la condition d'AVL et on se retrouve avec un déséquilibre. Il y a plusieurs procédures à suivre pour rééquilibrer un arbre.

3.4.)

Commençons par la rotation simple droite et gauche. La rotation simple droite s'effectue lorsque l'on effectue une insertion à gauche d'un arbre, créant un déséquilibre. Pour rétablir l'équilibre, il faut réduire la profondeur de l'arbre, on utilise les procédures demandées, car elles ont la propriété fondamentale de conserver le tri.

Supposons que l'on ait inséré un nouveau nœud à gauche dans le sous-arbre gauche du fils gauche de la racine qui crée un désiquilibre, alors, pour y rémédier, on fera en sorte que :

1. Par rapport au nœud déséquilibré, le fils gauche devient la nouvelle racine.
2. L'ancienne racine devient le fils le plus à droite de la nouvelle racine. Et on retourne la nouvelle racine.

C'est symétrique à droite on inverse tous les « gauche » par « droite ». Il faut prendre du papier et un crayon et faire des essais et se convaincre que ça marche. Ce n'est pas le même algorithme que dans le cours, en lisant les diapositives correspondantes, je ne suis pas totalement sûr d'avoir compris comment appliquer l'algorithme dans certaines situations, j'ai donc développé cette méthode qui j'espère est équivalente. (Elle l'est du moins pour les exemples que j'ai essayé.)

La rotation double sert lorsque l'on insère un nœud à droite de du fils de gauche de la racine ou à gauche du fils de droite de la racine et que cela crée un désiquilibre. Pour y rémédier, dans le cas d'une insertion à gauche du fils de droite de la racine :

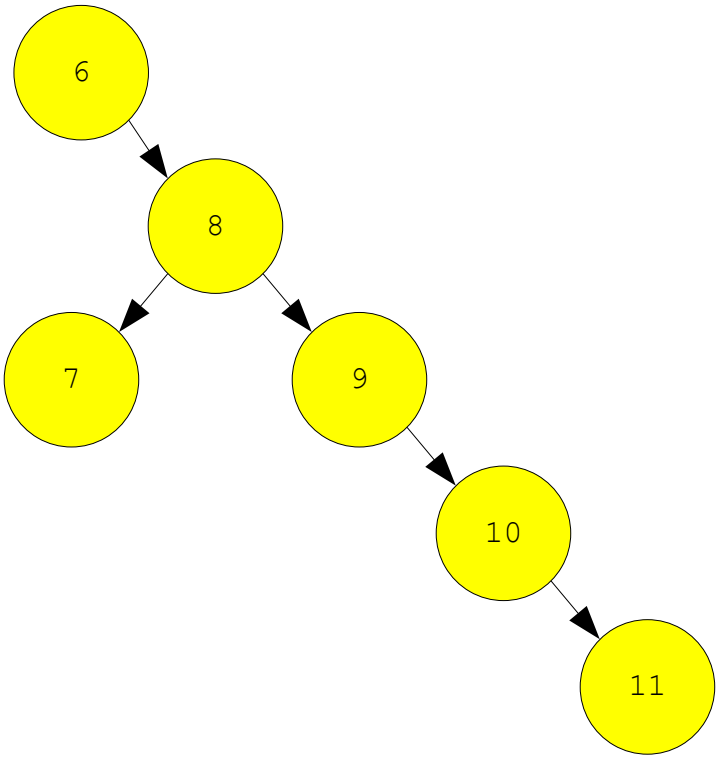
1. Le nœud inséré devient la nouvelle racine.
2. L'ancienne racine devient le fils gauche du nœud inséré.

C'est de nouveau l'inverse si on insère à droite. Pour illustrer, j'ai joint un exemple sur la page suivante.

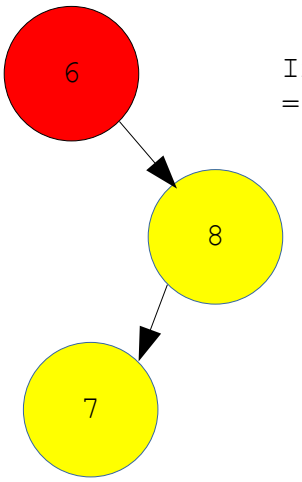
Exemple :

Soit la séquence : 6 8 7 9 10 11

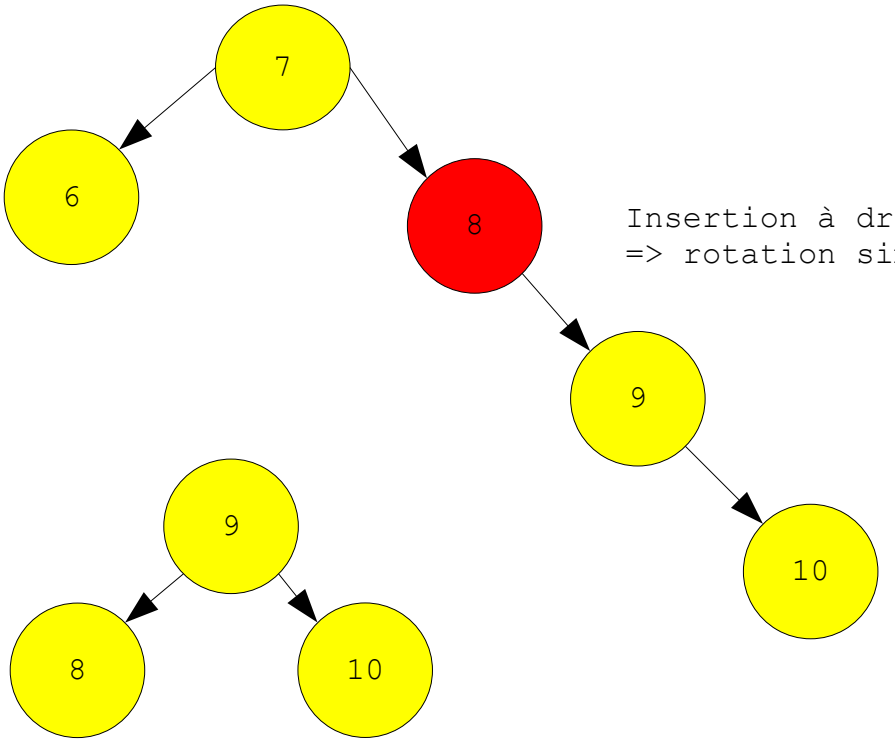
En suivant l'insertion simple...



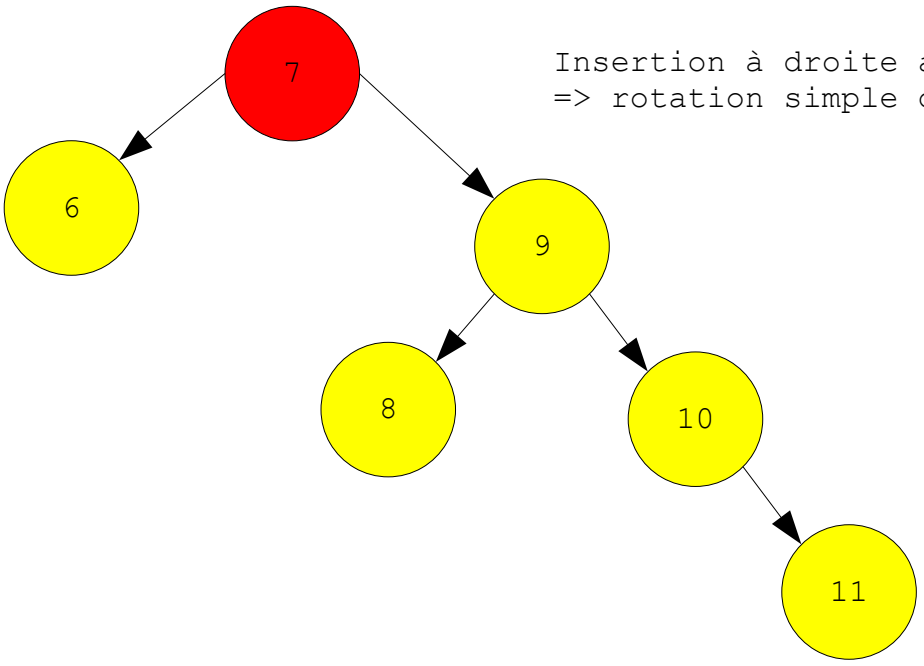
… c'est totalement déséquilibré!
Faisons alors avec les méthodes détaillées ci-dessus...



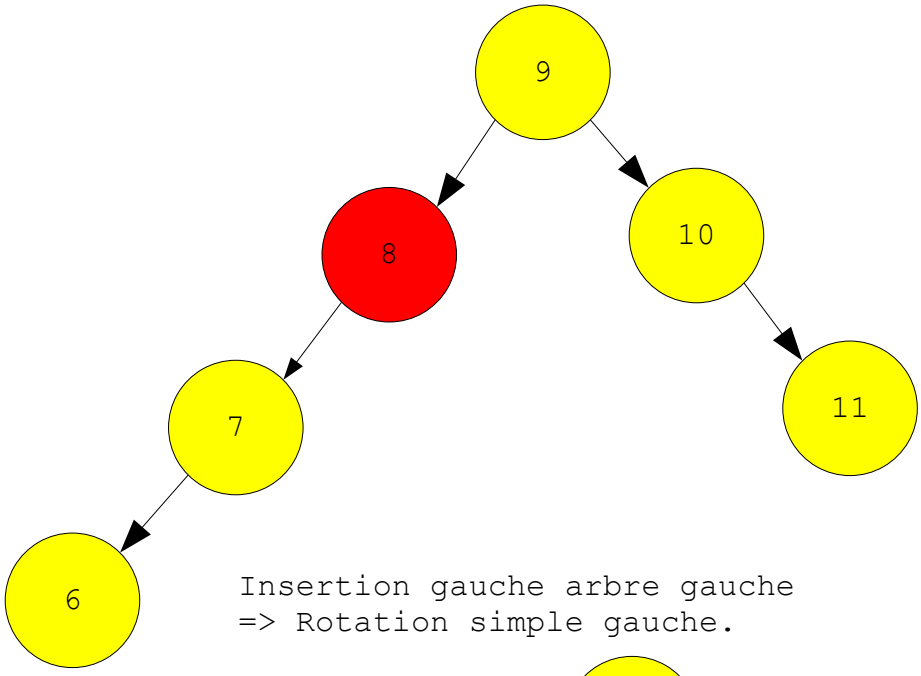
Insertion à gauche arbre droit.
=> rotation double gauche.



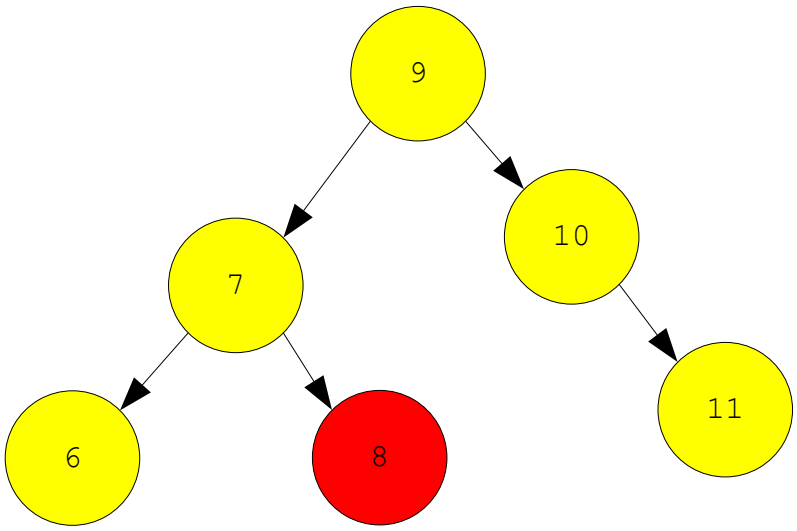
Insertion à droite arbre droit.
=> rotation simple droite.



Insertion à droite arbre droit.
=> rotation simple droite.



Insertion gauche arbre gauche
=> Rotation simple gauche.



...c'est un arbre équilibré ! Ce qui est rudement mieux.