Examen Informatique novembre 2007 – Proposition de correction (en couleur bleue)

TreeSet

Présentation de la structure *TreeSet*

Pour les besoins d'une application on doit utiliser le conteneur *TreeSet* appartenant à la librairie standard. On souhaite évaluer l'efficacité de cette structure en procédant expérimentalement.

Ce conteneur générique est basé sur l'ordonnancement des objets contenus.

Pour cela, les objets contenus devront être munis d'une méthode permettant de les comparer entre eux. C'est la méthode *compareTo* , proposée par l'interface standard *Comparable*, qui est utilisée pour cela (la convention concernant *compareTo*, inspirée du langage *C*, est la suivante: cette méthode retourne 0 en cas d'égalité, sinon retourne +1 ou -1 selon le sens de l'inégalité).

Elle est déclarée ainsi dans l'interface de *Comparable* :

```
public interface Comparable {
     public abstract int compareTo(Object arg0);  // compare this avec arg0
}
```

La classe à étudier, *TreeSet*, est déclarée ainsi :

```
public class TreeSet < T extends Comparable<T> > {
...
}
```

La déclaration < *T extends Comparable*<*T*> > signifie que la classe *TreeSet* ne pourra contenir que des objets *comparable,* cette obligation résultant logiquement du caractère trié du conteneur.

L'aide en ligne précise son positionnement dans le graphe d'héritage:

Class TreeSet<E>
java.lang.Object
  java.util.AbstractCollection<T>
    java.util.AbstractSet<T>
      java.util.TreeSet<T>
Type Parameters:
T - the type of elements maintained by this set
All Implemented Interfaces:
Serializable, Cloneable, Iterable<T>, Collection<T>, NavigableSet<T>, Set<T>, SortedSet<T>

- *Object* et *TreeSet* sont des classes concrètes, *AbstractCollection* et *AbstractSet* sont des classes abstraites et *Serializable*, *Cloneable* etc... sont des interfaces. Rappeler brièvement ce qui distingue ces trois catégories.

Classe concrète	Une classe concrète définit toutes les méthodes qu'elle expose. Elle peut posséder des données membres. Elle peut produire des instances. L'héritage est simple.

Classe Abstraite	Une classe abstraite possède parmi les méthodes qu'elle expose au moins une méthode non définie. Elle peut posséder des données membres. Elle ne peut pas produire d'instances. L'héritage est simple.
Interface	Une interface ne définit aucune des méthodes qu'elle expose. Elle ne possède pas de données membres. Elle ne peut pas produire d'instances. L'héritage d'interfaces peut être multiple.

Dans la suite on va essentiellement étudier la complexité de l'opération d'ajout d'un élément dans la structure (*add)*. Cette méthode est documentée ainsi dans l'aide en ligne:

```
public boolean add(E e)
        Adds the specified element to this set if it is not already present. More
        formally, adds the specified element e to this set if the set contains no element
        e2 such that (e==null ? e2==null : e.equals(e2)). If this set already contains
        the element, the call leaves the set unchanged and returns false.
Specified by:
        add in interface Set<E>
Overrides:
        add in class AbstractCollection<E>
Parameters:
        e - element to be added to this set
Returns:
        true if this set did not already contain the specified element
```

- D'après cette documentation la méthode *add* semble présente dans deux entités dont *hérite TreeSet : Set* et *AbstractCollection*. D'autre part les mots employés pour le signifier (*Specified by* et *Overrides*) sont différents. Pouvez-vous expliquer pourquoi ?

Specified by signifie ici que la méthode héritée est exposée dans la surclasse sans y être définie.
Overrides signifie que la méthode héritée est définie dans la surclasse et va être redéfinie dans la classe courante.

Une utilisation typique de *TreeSet*, qui utilise les facilités de l'*autoboxing* avec le type *Integer*, est la suivante :

```
final int MAX=10;
// création d'un conteneur d'Integer
TreeSet<Integer> treeSet=new TreeSet<Integer>(); // création d'un conteneur
// garnissage avec 10 Integer
for(int i=0;i<MAX;i++)
    treeSet.add(i);  // insertion de MAX objets Integer successifs par autoboxing
// parcours de la structure et affichage des éléments un à un (version foreach de for)
for(int value:treeSet)
    System.out.print(value +" ");
```

Affichage:

```
0 1 2 3 4 5 6 7 8 9
```

- Rappeler en quoi consiste dans cet exemple l'*autoboxing*

Facilité de conversion entre type de base et type objet.
L'autoboxing est une facilité permettant d'automatiser la production d'un *Integer* chaque fois qu'un *Integer* est attendu et qu'un *int* est fourni, et inversement.
Ici :

```
for(int i=0;i<MAX;i++) treeSet.add(i);
```

équivaut à :

```
for(int i=0;i<MAX;i++) treeSet.add(new Integer(i));
```

Mesure de la complexité de l'insertion par le temps de traitement

Dans un premier temps on souhaite évaluer expérimentalement la complexité de l'opération d'insertion d'un élément dans un *TreeSet*. Pour cela on souhaitait procéder ainsi :

- on part d'un conteneur vide.
- on mesure la durée d'une première insertion
- puis on mesure la durée de la deuxième insertion
- et ainsi de suite, jusqu'à par exemple 100000 ou 1000000 d'insertions. Au fur et à mesure de ces insertions, celles-ci s'effectue dans un conteneur comportant donc de plus en plus d'éléments.

Mais on renonce finalement à cette approche en raison d'une durée moyenne d'insertion trop faible et de la gène occasionnée par le *garbage collector*.

- En quoi le *garbage collector* peut-il se révéler gênant ici ?

Le GC va périodiquement suspendre le comportement normal de l'application pour procéder au nettoyage mémoire et va donc perturber toute mesure basée sur le temps.

On choisit donc de procéder autrement.

Mesure de la complexité de l'insertion par le nombre d'opérations de comparaisons effectuées

La méthode consiste maintenant à dénombrer le nombre de comparaisons opérées sur un objet donné lors d'une opération insertion.

- Dans le cas d'une liste chaînée monodirectionnelle de *n* éléments triés, quelle est la complexité (mesurée par le nombre de comparaisons effectuées) d'une opération d'insertion ?

O(n)

- Dans le cas d'un *ABR* équilibré de *n* éléments triés, quelle est la complexité (mesurée par le nombre de comparaisons effectuées) d'une opération d'insertion ?

O(log(n))

Pour dénombrer le nombre de comparaisons opérées par un objet *TreeSet* l'expérience consiste à créer une classe *Item* d'objets à insérer, et à munir cette classe de la méthode *compareTo*. On sait que cette méthode sera utilisée par *TreeSet* pour comparer l'*item* à insérer avec les *items* présents.

La méthode *compareTo*, explicitée ci-après, dénombre, pour un *item* donné, le nombre d'invocations de *compareTo* effectuées sur cet item.

Par ailleurs on utilise une variable statique (*value*) pour que chaque item reçoive un identifiant entier permettant de le caractériser (0, puis 1, puis 2 etc...). Enfin *toString* est redéfinie pour disposer d'un affichage commode de l'élément sous la forme <*id*,*nComp*>, c'est à dire identifiant de l'élément et nombre de comparaisons effectuées pour l'insérer.

```
class Item implements Comparable<Item> {
    public int id; // identificateur entier de cet item
    private static int value=0;
    public int nComp=0; // nombre de comparaison effectuée sur cet item
    public Item() { id=value++;  } // attribution d'un identifiant unique
    public int compareTo(Item client) { // definition de compareTo
```

```
        nComp++;     // incrémentation du nombre de comparaisons effectuées sur this
        return (client.id==id)?0:(client.id>id?1:-1); // retourne 0, -1 ou +1
    }
    @Override
    public String toString() { return "<"+id+","+nComp+">"; }
}
```

Le code de test est la suivant :

```
    public static void main(String[] args) {
        final int MAX = 4;
        // on commence par mettre les MAX items dans un tableau classique
        Item [] tab=new Item[MAX]; // creation d'un tableau de MAX Item
        for(int i=0;i<MAX;i++) tab[i]=new Item(); // garnissage du tableau
        // On insère ces MAX items dans le treeSet
        // création du treeSet
        TreeSet<Item> treeSet= new TreeSet<Item>();

        // parcours du tableau et insertion de chaque item dans le treeSet
        for(int i=0;i<MAX;i++) treeSet.add(tab[i]); // insertion de chaque Item
        // parcours du treeSet pour interroger chaque item
        for(Item item:treeSet)
              System.out.println(item); // affichage du nombre de comparaisons
                                        // effectuées sur chaque Item
    }
```

Par exemple pour 4 insertions successives (*MAX*=4) la sortie est la suivante (pour chaque couple à gauche *l'id* de l'item, à droite le nombre de comparaison qu'il a subi pour son insertion) *:*

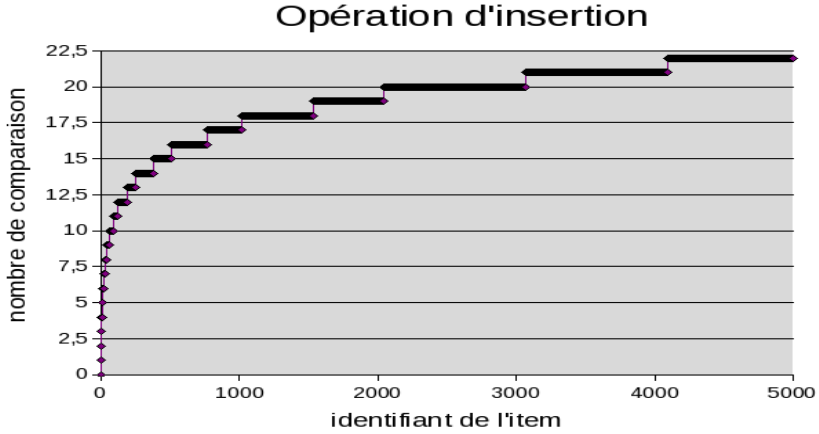
```
<3,2>
<2,2>
<1,1>
<0,0>
```

Cela signifie que l'item *0* (le premier à être inséré) ne doit se comparer à aucun autre item puisqu'il arrive dans un conteneur vide. L'item *1* est comparé une fois avant de trouver sa place. L'item *2* est comparé *2* fois. L'item *3* est comparé *2* fois.

- Sans le polymorphisme et la liaison retardée appliquée *à compareTo*, ce code de test serait complètement inopérant. Expliquer pourquoi.

Le conteneur générique TreeSet met en oeuvre sur les éléments qu'on lui confie la méthode qui permet de comparer ces éléments 2 à 2 (compareTo). Le polymorphisme permet de s'assurer que la méthode compareTo qui sera effectivement appelée ici est bien celle de Item (et non celle qui existe dans la classe mère Object, dont hérite Item, comme ça serait le cas sans polymorphisme).

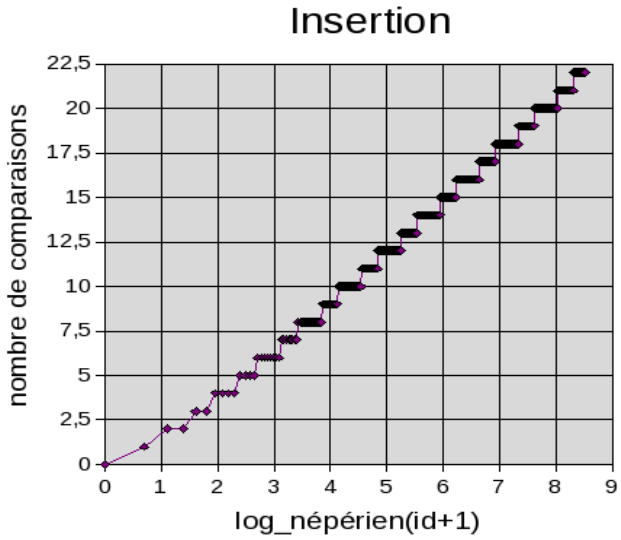
Le même code est maintenant utilisé mais avec une valeur de *MAX* plus significative (*5000*) pour une mesure expérimentale. La sortie standard analogue à ci-dessus (mais en plus long!) est redirigée vers un tableur et une représentation graphique en est tirée correspondant à la valeur *5000* pour *MAX*.



- Quelle est la nature de cette fonction ?

Un logarithme

Pour confirmer cette impression on trace *nComp=ln(id+1)*. On obtient le tracé suivant :



- En déduire la loi mathématique *f* qui établit avec une bonne approximation la relation *nComp=f(id)*

La pente de la droite est approximativement 20/8, c'est à dire 2,5. donc on peut écrire :
nComp=2,5*ln(id+1)

Analyse de la structure de donnée interne

- Les résultats précédents nous conduisent à penser que l'implémentation interne de *TreeSet* est un *ABR* équilibré. Rappeler ce qu'est un *ABR* et pourquoi on peut penser dans cet exemple qu'il est équilibré

Un *ABR* est un arbre binaire de recherche. La valeur du noeud d'un fils gauche est toujours inférieure à celle de son père, et la valeur du noeud d'un fils droit est toujours supérieure à celle de son père. Il peut être défini de la façon suivante :

Un ABR est :
- soit vide
- soit constitué ainsi : *<SAG, Valeur, SAD >* où *SAG* et *SAD* sont eux mêmes des *ABR*

Un *ABR* équilibré ne présente aucun déséquilibre entre chacun des ses *SAG* et *SAD* supérieur à 1.

Seul un *ABR* équilibré garantit une complexité d'insertion en *O(log(n))*.

(Au contraire un *ABR* totalement déséquilibré (un fil) présenterait une complexité d'insertion en *O(n)*)

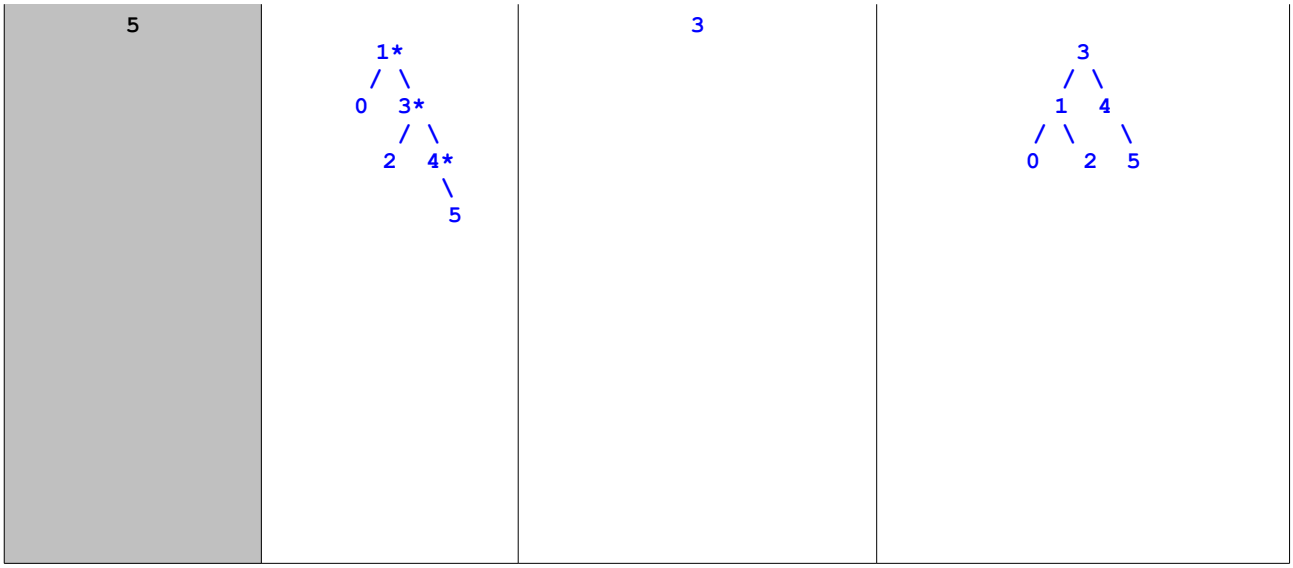
Un *ABR* équilibré maintient son équilibre à la volée. Cela signifie que si une opération d'insertion déclenche un déséquilibre (un décalage de +/- 2 dans un de ses arbres) ce déséquilibre est corrigé après l'insertion.
Pour préciser ce mécanisme sur l' *ABR* de *TreeSet* on revient à un petit nombre d'insertions et à l'analyse du nombre de comparaisons induites sur chaque élément, pour *MAX=6*:

<5,3>

<4,3>
<3,2>
<2,2>
<1,1>
<0,0>

- Compléter le tableau suivant (cases non grisées) en dessinant l'aspect de l'arbre après l'insertion de l'item *id* et après rééquilibrage éventuel (idem signifie qu'aucun rééquilibrage n'est nécessaire), de façon à retrouver les nombres de comparaisons annoncés ci-dessus. Marquer d'une croix comme ci-dessous les éléments avec lesquels l'item dernièrement inséré a été comparé:

Insertion de l'élément	*Etat de l'ABR juste après l'insertion*	*Nombre de comparaisons nécessitées par cette insertion*	*Etat de l'ABR après rééquilibrage éventuel*
0	0	0	(pas de rééquilibrage) 0
1	0* \ 1	1	(pas de rééquilibrage) 0* \ 1
2	0* \ 1* \ 2	2	1 / \ 0 2
3	1* / \ 0 2* \ 3	2	pas de rééquilibrage
4	1* / \ 0 2* \ 3* \ 4	3	1 / \ 0 3 / \ 2 4



Algorithme d'équilibrage de l'arbre

En langage C la structure de données d'un arbre AVL décrit dans les questions précédentes peut être représentée par :

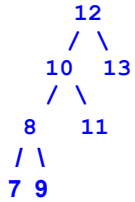
```
/* Déclaration du maillon ou du noeud de l'arbre*/
typedef struct node {
    int  element;   /* la valeur du  noeud*/
    struct node *fils_gauche; /* fils gauche*/
    struct node *fils_droit;  /* fils droit*/
    int height; /* hauteur de l'arbre*/
}avl_node;
/*définition du type AvlTree*/
typedef avl_node *AvlTree;
```

- Question : à votre avis à quoi sert le champ *height* ?

Gérer et contrôler la différence de hauteur entre *les fils gauches (SAG)* et les fils droits *(SAD), lors des opérations d'insertion et de suppression*

Dans le cas d'un déséquilibre nous devons rééquilibrer l'arbre. Étudions une transformation permettant de rééquilibrer localement un arbre lorsque, pour un nœud, le déséquilibre atteint +2, ses arbres gauche et droit étant équilibrés. On suppose que le déséquilibre du sous arbre gauche est de +1. Dans ce cas, nous devons effectuer une rotation à droite, décrite dans le cours.

- Donner un exemple d'arbre avec de tels déséquilibres



La fonction d'équilibrage d'un arbre (vue dans le cours), peut être écrite de la façon suivante :

```
AvlTree SimpleRotationDroite( AvlTree k2 ) {
    AvlTree k1;

    k1 = k2->fils_gauche;
    k2->fils_gauche = k1->fils_droit;
    k1->fils_droit = k2;

    k2->height = max( height(k2->fils_gauche), height(k2->fils_droit) ) + 1;
    k1->height = max( height(k1->fils_gauche), k2->height ) + 1;

    return k1;          /* La nouvelle racine*/
}
```

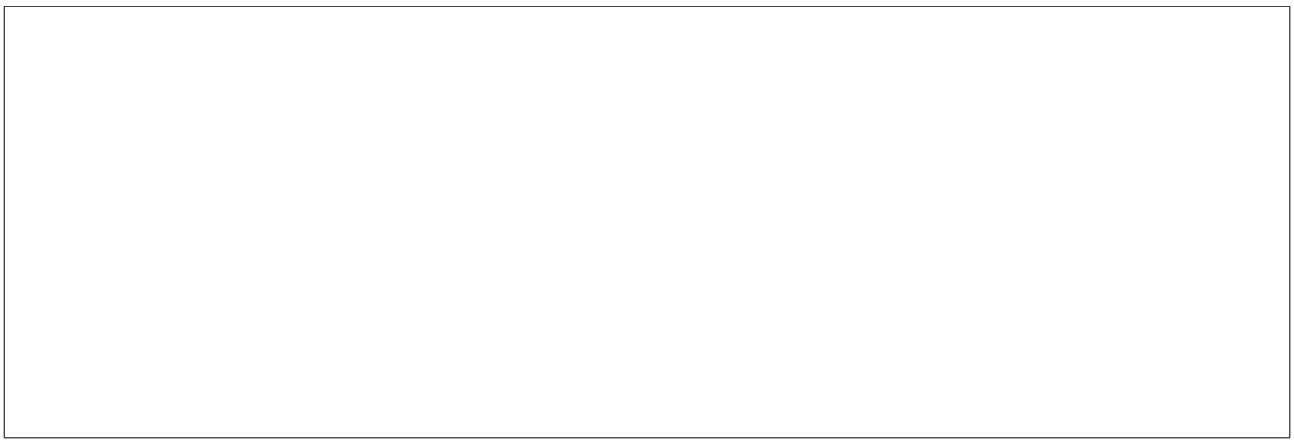
On se place dans le contexte de la plateforme *Turbo-C 8086* utilisée pendant le module. Toutes les adresses et pointeurs seront supposées intra-segment (= adresses/pointeurs *near*).

- On considère une variable globale `pNode` autorisant l'accès à un nœud d'un arbre. Que peut-on dire des 3 déclarations suivantes : `struct node *pNode;` `avl_node *pNode;` et `AvlTree pNode;` ?

On considère la portion de code suivante :

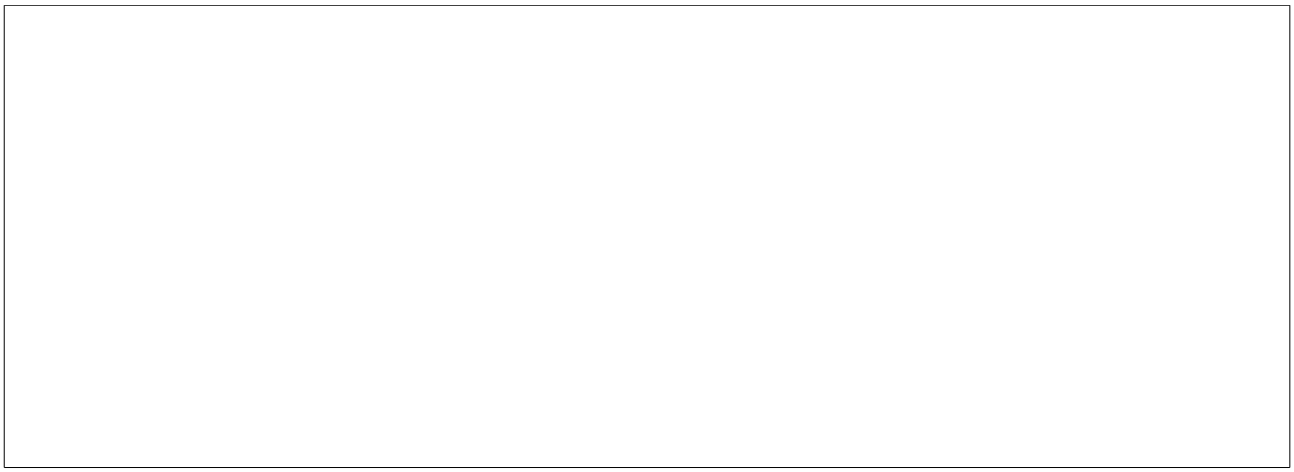
```
struct node myNode;
struct node *pNode = &myNode;
myNode.element = 5;
myNode.fils_gauche = myNode.fils_droit = NULL;
myNode.height = 0;
```

- A l'aide d'un schéma de mémoire, représentez les variables telles qu'elles sont après exécution de cette portion de code. Ne détaillez pas chaque étape mais simplement le résultat final.

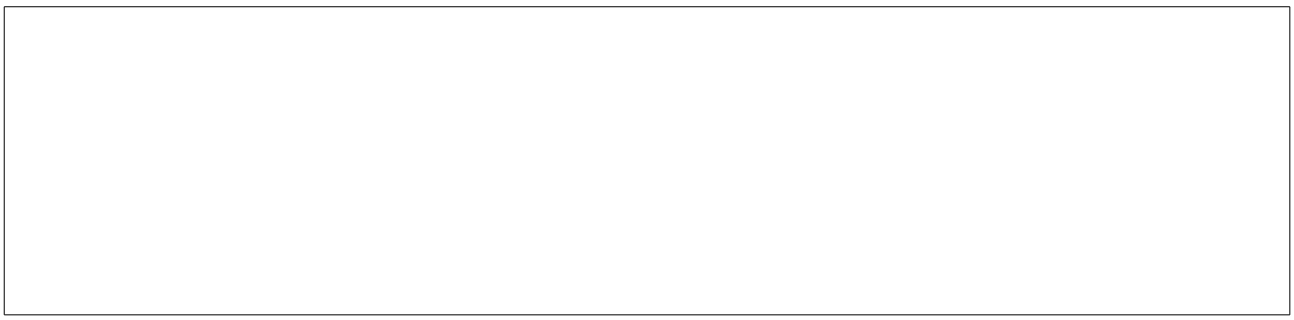


On suppose que la fonction *SimpleRotation* a été appelée grâce à `pNode = SimpleRotation(&myNode);` et que la ligne `k1=k2->fils_gauche;` vient d'être exécutée.

- Dessinez le contexte de pile de la fonction *SimpleRotation* à ce moment de l'exécution. Faites apparaître les registres *SP* et *BP* ainsi que les déplacements relatifs à *BP* pour l'accès aux paramètres et variables locales.



- Codez en assembleur la ligne `k1 = k2->fils_gauche;` On accèdera à `k1` et `k2` en utilisant l'adressage basé sur BP.



- Codez en assembleur la ligne `return k1;`

Architecture

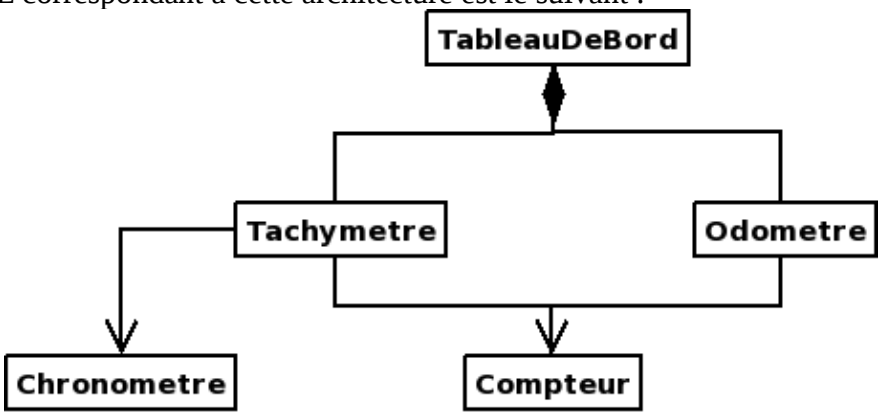
On souhaite reconsidérer l'architecture applicative de l'exemple du tableau de bord vu en cours. On avait vu que dans cette première version le tableau de bord est vu comme un objet agrégeant d'autres objets tels que l'odomètre ou le tachymètre. Ces derniers se partagent l'usage d'un compteur qui est un objet singleton délivrant le nombre de tour de roues effectués par le véhicule et qui sera appelé *Capteur* dans la suite. Un chronomètre est également utilisé par l'objet *Tachymetre*.
Un exemple de mise en oeuvre de ce dispositif est illustré par le programme de lancement suivant :

```
Compteur capteur=new Compteur(); // capteur de roue
final Chronometre chrono=new Chronometre();
final TableauDeBord tdb=new TableauDeBord(capteur,chrono);
// thread de scrutation du tableau de bord toutes les 5 secondes
Thread threadScrutation=new Thread(new Runnable() {
      public void run() {
            while (true) {
                  try {
                        Thread.sleep(5000);
                        System.out.printf("%5.1f km/h\t",tdb.getSpeed());
                        System.out.printf("%5.1f m\t\t",tdb.getDist());
                        System.out.println(chrono.getValue()/1000+" s");
                  } catch (InterruptedException e) {}
            }
      }
});
threadScrutation.start();
```

Cette portion de code peut donner le résultat suivant (on rappelle que le capteur est muni d'un *thread* qui incrémente toutes les secondes le nombre de tours de roue sur une base légèrement aléatoire):

```
 76,7 km/h    113,4 m             5 s
 76,4 km/h    220,4 m             10 s
 77,0 km/h    327,4 m             15 s
 77,0 km/h    434,4 m             20 s
 77,0 km/h    541,4 m             25 s
 76,9 km/h    648,4 m             30 s
```

Le schéma UML correspondant à cette architecture est le suivant :



On souhaite ajouter des éléments à ce tableau de bord, comme, par exemple, un *odomètre avec remise à zéro*, une *jauge d'essence*, un *indicateur de consommation instantanée de carburant*, un *compte tour* etc...

- L'architecture proposée en cours ne se révèle pas de ce point de vue très extensible. Expliquer pourquoi.

Le tableau de bord est conçu comme une entité qui ne peut contenir/référencer qu'un odomètre et un tachymètre. l'introduction d'une nouvelle jauge implique donc de modifier la structure et donc le code du tableau de bord.

Pour résoudre ce problème d'extensibilité on décide de capturer (dans une interface) le comportement commun de chacune des jauges évoquées. *Tachymetre*, *Odometre*, *Jauge d'essence*, *Jauge instantanée* et *Compte-tour* peuvent en effet être considérées chacun comme un cas particulier de *Jauge*. Une *Jauge* est définie ici comme une entité sur laquelle on peut invoquer à un instant donné le service *getValue* et obtenir une indication numérique qui sera respectivement une vitesse instantanée (en *km/h*), une distance (en *km*), un volume (en *litre*), un volume instantané (en *litre/100km*), un nombre de tour par minute (en mn^{-1}). On va considérer par la suite que ces valeurs numériques sont toutes de type *double*. Toutes les classes évoquées implémenteront donc l'interface *Jaugable* dont la sémantique vient d'être décrite.

- Écrire en java l'interface *Jaugable*

```
public interface Jaugable {
    public double getValue();
}
```

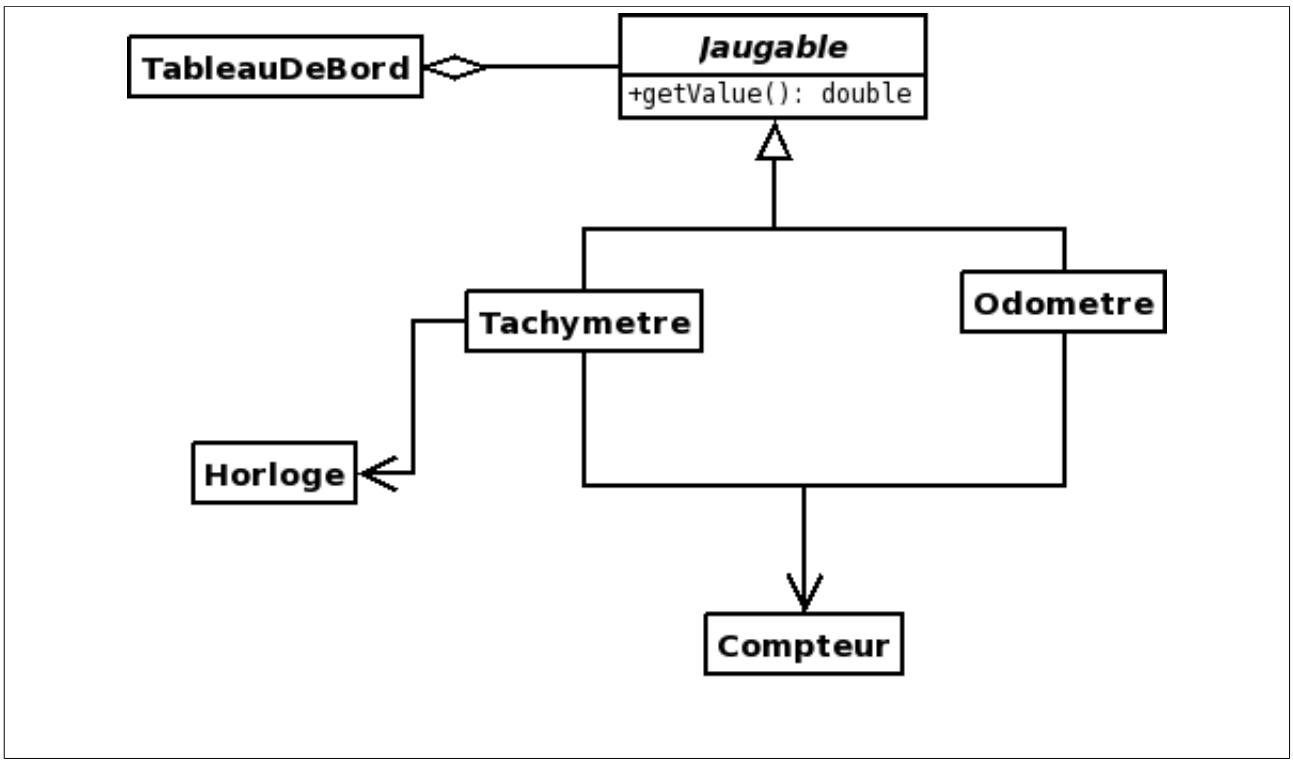
L'exemple de programme de test précédent serait réécrit, dans cette perspective, ainsi :

```
Compteur capteur=new Compteur(); // capteur de roue
Chronometre chrono=new Chronometre();

Odometre odo=new Odometre(capteur);
Tachymetre tachy=new Tachymetre(capteur, chrono);

final TableauDeBord tdb=new TableauDeBord();
tdb.add(odo);
tdb.add(tachy);
// thread de scrutation du tableau de bord toutes les 5 secondes
Thread threadScrutation=new Thread(new Runnable() {
      public void run() {
            while (true) {
                  try {
                        Thread.sleep(5000);
                        System.out.println(tdb);
                  } catch (InterruptedException e) {}
            }
      }
});
threadScrutation.start();
```

- Ce mode de construction et de mise en oeuvre d'un tableau de bord est différent de celui présenté précédemment. En déduire, en incluant *Jaugable*, la nouvelle version du schéma *UML* concernant *TableauDeBord*, *Tachymetre*, *Odometre*, *Compteur*, *Horloge* (et *Jaugable*).



- La classe *TableauDeBord* est ci-dessus sollicitée trois fois de façon différentes dans le code de test. En déduire les signatures du constructeur et des deux méthodes impliquées par ce code:

```
// non demandé
final TableauDeBord tdb=new TableauDeBord(); ==> constructeur sans param
tdb.add(odo) ou  tdb.add(tachy) ==> add(Jaugable)
System.out.println(tdb); ==>  Override de toString

// demandé
public class TableauDeBord {
  public TableaudeBord();
  public void add(Jaugable jauge);
  public String toString();
}
```

On suppose, pour la commodité, que chacune des *jauges* redéfinit la méthode *toString* de façon à ce que celle-ci produise une chaîne de caractère exprimant commodément la valeur considérée et son unité. Par exemple

```
System.out.println(tachy); // équivaut à System.out.println(tachy.toString());
```

affiche à l'écran

```
73,4 km/h
```

- Compte tenu de cette remarque implémenter complètement *TableauDeBord* en choisissant un tableau ou une structure générique de votre choix concernant la gestion des jauges et en implémentant les trois méthodes ou constructeurs concernés.

```
public class TableauDeBord {
  // structure pour la gestion de la liste des jauges... à compléter

  private List<Jaugable> list;

 // constructeur
```

```
 public TableauDeBord() {
     list=new List<Jaugable>()
 }

  // méthode1

public void add(Jaugable jauge) {
    list.add(jauge);
}


  // méthode 2

public String toString() {
  String info="";
  for(Jaugable jauge:list) info+=jauge.toString()+" ";
  return info;
}


}
```