L'interface Map
et
son implémentation
par une TreeMap

Université de Montpellier
Faculté des sciences
Mars 2020

L'interface Map (rappel)

- Map = Dictionnaire associatif
- Type abstrait de données
- Modéliser des fonctions (mathématiques) partielles discrètes d'un ensemble K (clefs) dans un ensemble V (valeurs)
  - Cela associe à un élément de K une valeur de V
- Opérations principales
  - **put (clef, valeur)** Insérer une association (clef, valeur)
  - **get(clef)** Rechercher par la clef (pour obtenir la valeur)
  - **remove(clef)** Supprimer une association (clef, valeur)

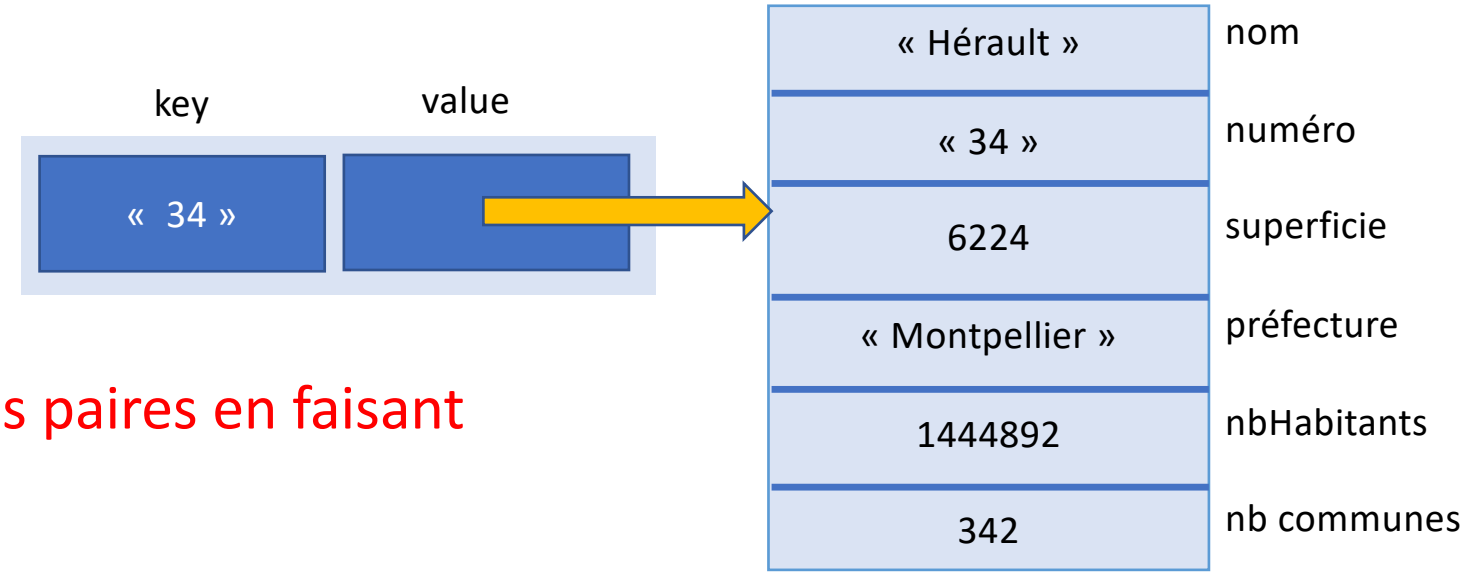
Exemple de dictionnaire associatif (rappel)

- Associer la description des départements français à leur numéro
- 101 départements
  - Numéro à 2 ou 3 chiffres
  - Pouvant commencer par 0
  - Pouvant contenir une lettre (ex. 2A, 2B en Corse, 69D, 69M à Lyon)
  - Tous les numéros n'existent pas
    - N'existent pas : 20, 985, rien entre 95 et 971, rien entre 978 et 984, rien après 989
- Clef : numéro de département (String)
- Valeur : description du département (Département)

- 01 : Ain
- 02 : Aisne
- 03 : Allier
- 04 : Alpes-de-Haute-Provence
- 05 : Hautes-Alpes
- 06 : Alpes-Maritimes
- 07 : Ardèche
- 08 : Ardennes
- 09 : Ariège
- 10 : Aube
- 11 : Aude
- 12 : Aveyron
- 13 : Bouches-du-Rhône
- 14 : Calvados
- 15 : Cantal
- 16 : Charente
- 17 : Charente-Maritime
- 18 : Cher
- 19 : Corrèze
- 2A : Corse-du-Sud
- 2B : Haute-Corse
- 21 : Côte-d'Or
- 22 : Côtes-d'Armor
- 23 : Creuse
- 24 : Dordogne
- 25 : Doubs
- 26 : Drôme
- 27 : Eure
- 28 : Eure-et-Loir
- 29 : Finistère
- 30 : Gard
- 31 : Haute-Garonne
- 32 : Gers
- 33 : Gironde
- 34 : Hérault
- 35 : Ille-et-Vilaine
- 36 : Indre
- 37 : Indre-et-Loire
- 38 : Isère
- 39 : Jura
- 40 : Landes
- 41 : Loir-et-Cher
- 42 : Loire
- 43 : Haute-Loire
- 44 : Loire-Atlantique
- 45 : Loiret
- 46 : Lot
- 47 : Lot-et-Garonne
- 48 : Lozère
- 49 : Maine-et-Loire
- 50 : Manche
- 51 : Marne
- 52 : Haute-Marne
- 53 : Mayenne
- 54 : Meurthe-et-Moselle
- 55 : Meuse
- 56 : Morbihan
- 57 : Moselle
- 58 : Nièvre
- 59 : Nord
- 60 : Oise
- 61 : Orne
- 62 : Pas-de-Calais
- 63 : Puy-de-Dôme
- 64 : Pyrénées-Atlantiques
- 65 : Hautes-Pyrénées
- 66 : Pyrénées-Orientales
- 67 : Bas-Rhin
- 68 : Haut-Rhin
- 69D : Rhône
- 69M : Métropole de Lyon
- 70 : Haute-Saône
- 71 : Saône-et-Loire
- 72 : Sarthe
- 73 : Savoie
- 74 : Haute-Savoie
- 75 : Paris
- 76 : Seine-Maritime
- 77 : Seine-et-Marne
- 78 : Yvelines
- 79 : Deux-Sèvres
- 80 : Somme
- 81 : Tarn
- 82 : Tarn-et-Garonne
- 83 : Var
- 84 : Vaucluse
- 85 : Vendée
- 86 : Vienne
- 87 : Haute-Vienne
- 88 : Vosges
- 89 : Yonne
- 90 : Territoire de Belfort
- 91 : Essonne
- 92 : Hauts-de-Seine
- 93 : Seine-Saint-Denis
- 94 : Val-de-Marne
- 95 : Val-d'Oise
- 971 : Guadeloupe
- 972 : Martinique
- 973 : Guyane
- 974 : La Réunion
- 975 : Saint-Pierre-et-Miquelon
- 976 : Mayotte
- 977 : Saint-Barthélemy
- 978 : Saint-Martin
- 984 : Terres australes et antarctiques françaises
- 986 : Wallis-et-Futuna
- 987 : Polynésie française
- 988 : Nouvelle-Calédonie
- 989 : Île de Clipperton

Les entrées seront des paires (MyEntry) (numéro de département, département)

- Clef : numéro de département (String)
- Valeur : description du département (Département)



- On enrichira les paires en faisant une sous-classe

Représentation des Départements (valeurs)

```
public class Departement {
    private String nom;
    private String numero;
    private double superficie;
    private String préfecture;
    private int nbHabitants;
    private int nbCommunes;
// …..
```

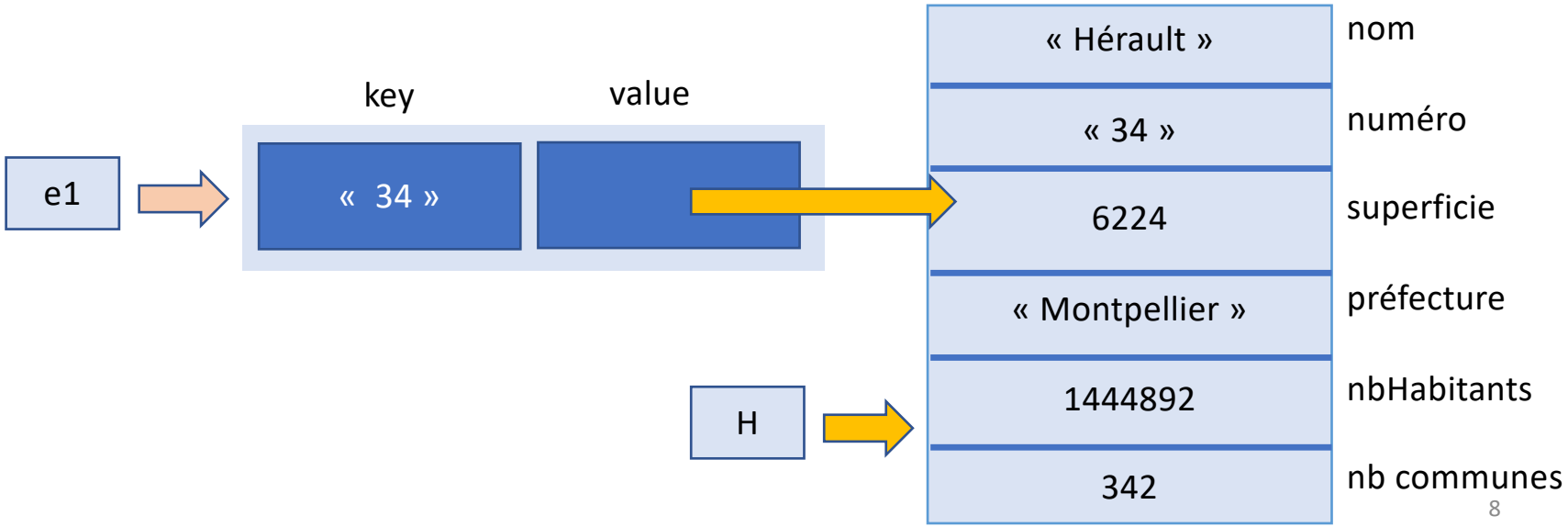
Représentation des entrées de base (associations)

```
public class MyEntry<K,V> {
        K key;
        V value;

        public MyEntry(K key, V value) {
          this.key = key;
          this.value = value;
        }
}
```

Exemple d'entrée pour ranger des départements dans une Map

```
Departement H = new Departement("Hérault","34",
                                6224,"Montpellier",1444892,342);
MyEntry<String,Departement> e1 = new MyEntry<>("34",H);
```



Principe des TreeMap

- Rangement des entrées dans un arbre
  - binaire
  - ordonné par les clefs (arbre binaire dit « de recherche »)
  - auto-équilibré (les branches croissent à peu près à la même vitesse)

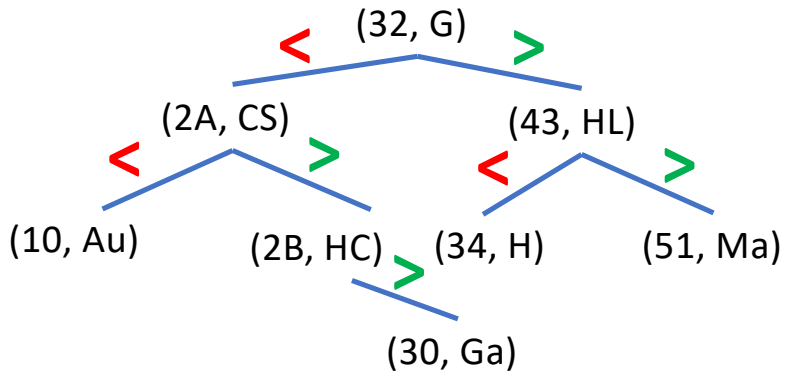
Principe des TreeMap

- Rangement des entrées dans un arbre
  - binaire
  - ordonné par les clefs (arbre binaire dit « de recherche »)
  - auto-équilibré (les branches croissent à peu près à la même vitesse)
- Arbre binaire
  - type abstrait de données défini récursivement par :
    - une racine
    - un sous-arbre gauche
    - un sous-arbre droit
  - structure de données composée d'arbres ou de cellules chaînées
  - Ici : nous choisissons une structure de cellules chaînées entre elles

Principe des TreeMap

- Rangement des entrées dans un arbre binaire
- Les clefs sont ordonnées
  - les clefs de la branche GAUCHE d'un noeud sont **INFERIEURES** à la clef du noeud lui-même.
  - les clefs de la branche DROITE d'un noeud sont **SUPERIEURES** à la clef du noeud lui-même.

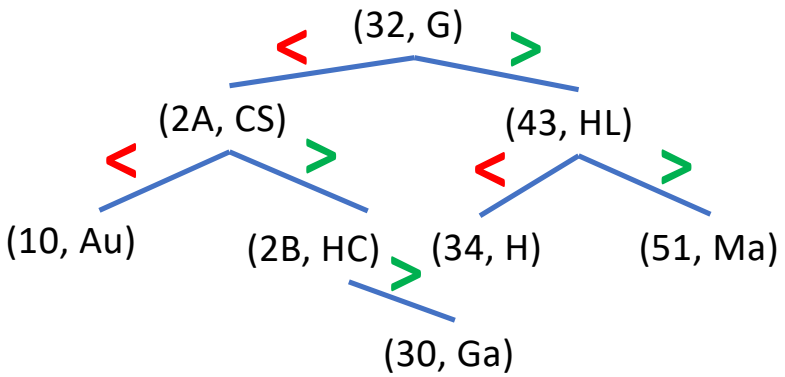
2A < 32



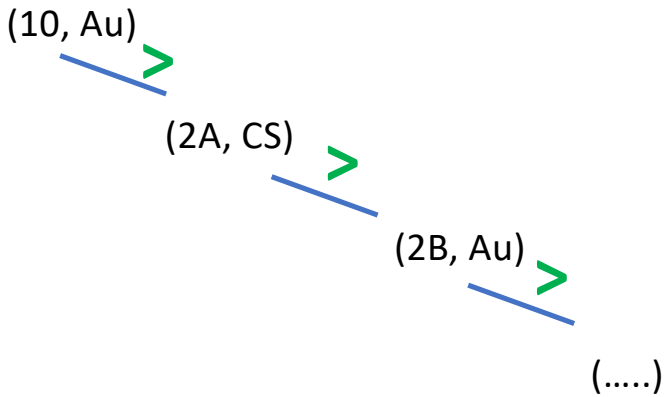
43 > 32

Principe des TreeMap

- L'arbre est équilibré : les branches sont « à peu près » de la même longueur

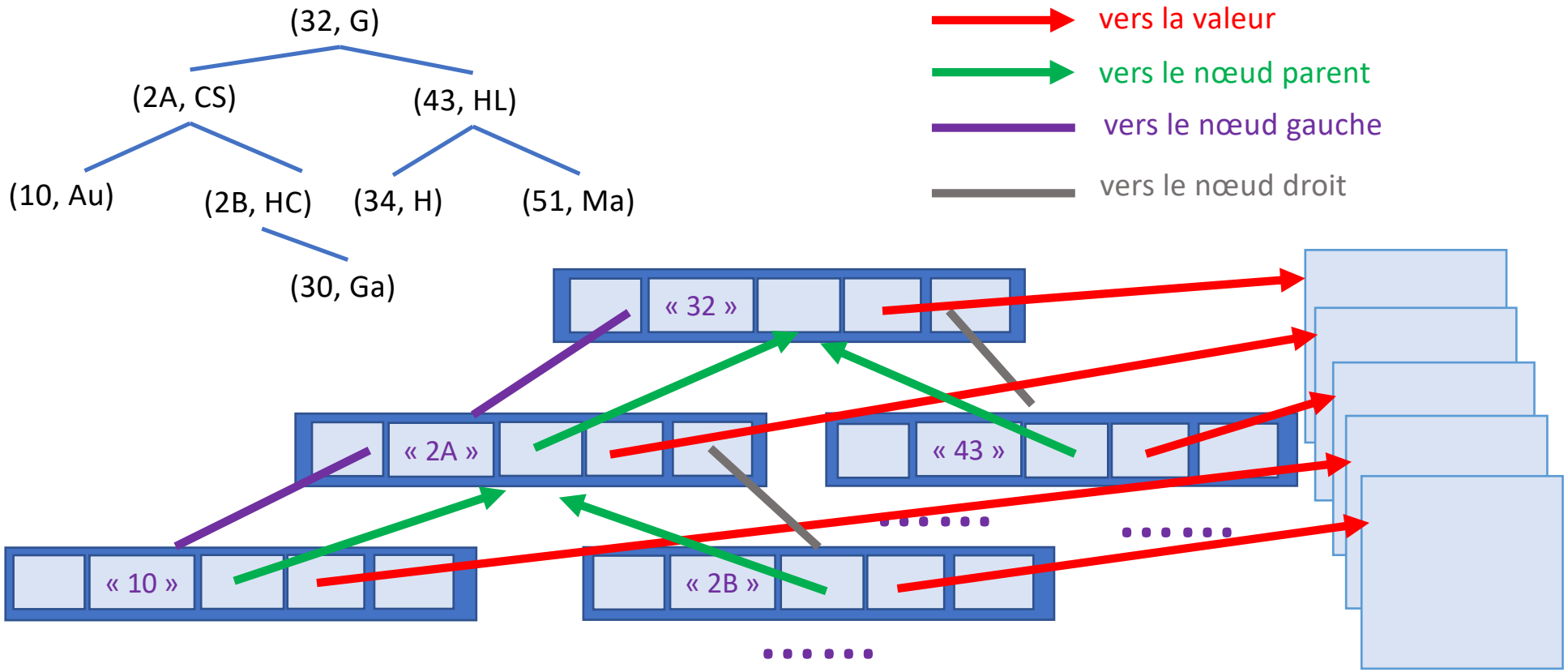


Arbre équilibré



Arbre non équilibré

Représentation par des cellules chaînées



Les nœuds sont des entrées enrichies (sous-classe de MyEntry)

```
public class MyNodeEntry
                <K extends Comparable<K>,V>
                extends MyEntry<K,V> {

MyNodeEntry<K,V> left = null;
MyNodeEntry<K,V> right = null;
MyNodeEntry<K,V> parent;

public MyNodeEntry(K key, V value,
                        MyNodeEntry<K,V> parent) {
super(key,value);
this.parent=parent;
}
```

La TreeMap contient le nœud racine (root)

```
public class MyTreeMap<K extends Comparable<K>, V>
                                    implements Map<K, V> {

    private MyNodeEntry<K,V> root;

    private int nbNodes;

    public MyTreeMap() {}

…
}
```

Principe de l’ajout : put(key, value)

Arbre vide

- root = null
- nbNodes = 0

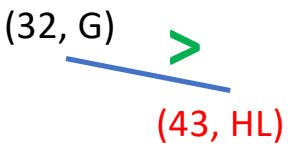
Principe de l'ajout : put(« 32 », G)

(32, G)

L'arbre est réduit à un nœud contenant

- la clef « 32 »
- la valeur G
- parent = null
- left = null
- right = null

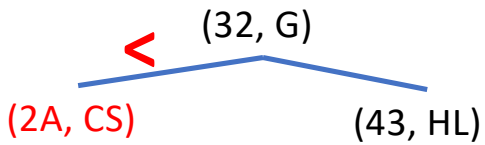
Principe de l'ajout : put(« 43 », HL)



Ajout de (« 43 », HL)

- « 43 » est supérieur à « 32 » (ordre alphabétique)
- on crée un nœud avec :
  - clef « 43 »
  - valeur HL
  - parent : le nœud portant « 32 »
  - left = null
  - right = null
- le nœud « 32 » a pour nœud droit (right) le nœud « 43 »

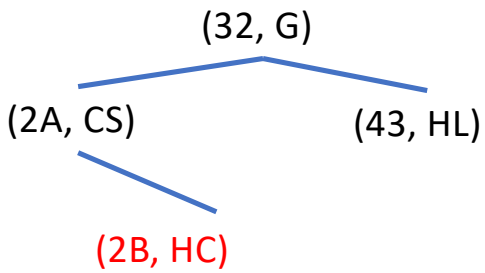
Principe de l'ajout : put(« 2A », CS)



Ajout de (« 2A », CS)

- « 2A » est inférieur à « 32 » (ordre alphabétique)
- on crée un nœud avec :
  - clef « 2A »
  - valeur CS
  - parent : le nœud portant « 32 »
  - left = null
  - right = null
- le nœud « 32 » a pour nœud gauche (left) le nœud « 2A »

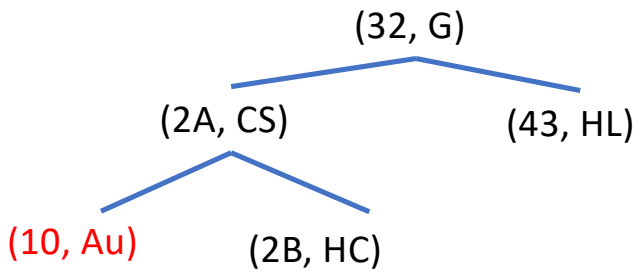
Principe de l'ajout : put(« 2B », HC)



Ajout de (« 2B », HC)

- « 2B » est inférieur à « 32 », on se déplace à gauche
- « 2B » est supérieur à « 2A »
- il est rangé dans la droite du nœud « 2A »

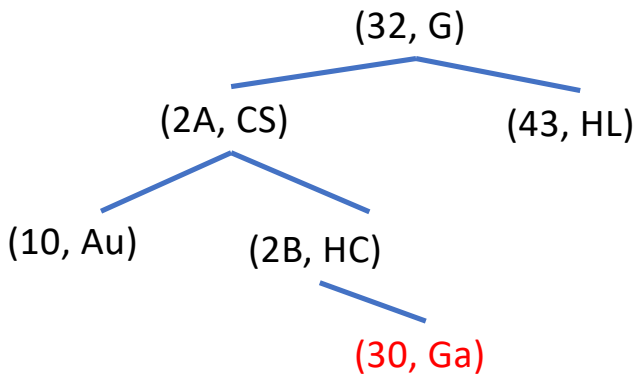
Principe de l'ajout : put(« 10 », Au)



Ajout de (« 10 », Au)

- « 10 » est inférieur à « 32 », on se déplace à gauche
- « 10 » est inférieur à « 2A »
- il est rangé dans la gauche du nœud « 2A »

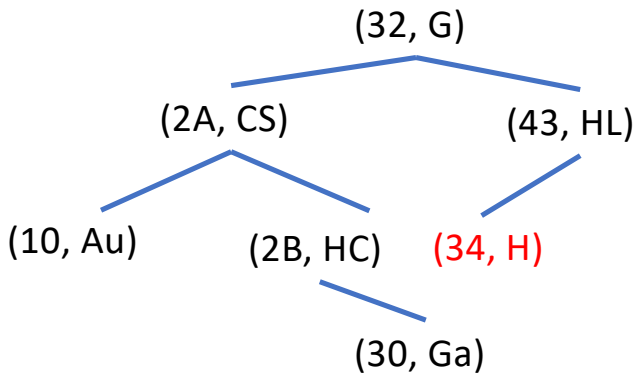
Principe de l'ajout : put(« 30 », Ga)



Ajout de (« 30 », Ga)

- « 30 » est inférieur à « 32 », on se déplace à gauche
- « 30 » est supérieur à « 2A », on se déplace à droite
- « 30 » est supérieur à « 2B »
- il est rangé dans la droite du nœud « 2B »

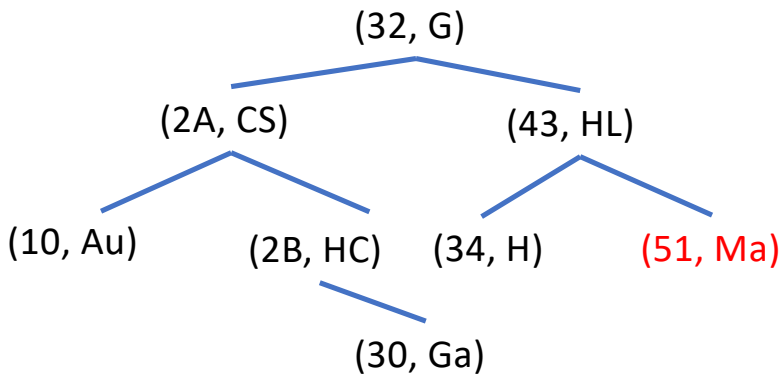
Principe de l'ajout : put(« 34 », H)



Ajout de (« 34 », H)

- « 34 » est supérieur à « 32 », on se déplace à droite
- « 34 » est inférieur à « 43 »
- il est rangé dans la gauche du nœud « 43 »

Principe de l'ajout : put(« 51 », Ma)



Ajout de (« 51 », Ma)

- « 51 » est supérieur à « 32 », on se déplace à droite
- « 51 » est supérieur à « 43 »
- il est rangé dans la droite du nœud « 43 »

```
public V put(K key, V value) {
   if (this.isEmpty()){ // si l'arbre est vide, on crée une racine
       root = new MyNodeEntry<>(key, value, null); this.nbNodes++;  return value;
    }
    // sinon on cherche l'emplacement adapté
    int cmp;  MyNodeEntry<K,V> courant, parent;  courant = root;
    do {  parent = courant;   cmp = key.compareTo(courant.key);
          if (cmp < 0)      courant = courant.left;  // descendre à gauche
          else if (cmp > 0)     courant = courant.right; // descendre à droite
          else { // la clef est déjà dedans, on change simplement la valeur associée
                courant.setValue(value);
                return value;     }
     } while (courant != null);
     // si on arrive ici il faut insérer à gauche ou à droite
     MyNodeEntry<K,V> e = new MyNodeEntry<>(key, value, parent);
     if (cmp < 0)  parent.left = e;
     else    parent.right = e;
     this.nbNodes++;
     return null;
}
```

Equilibrage

- Ce point ne sera pas vu dans ce cours
- Il s'agit de rééquilibrer l'arbre pendant l'ajout ou le retrait
- Pour ceux qui veulent aller plus loin

https://www.irif.fr/~carton/Enseignement/Algorithmique/Programmation/RedBlackTree/

- Référence Cormen, Thomas; Leiserson, Charles; Rivest, Ronald; Stein, Clifford (2009). "13". Introduction to Algorithms (3rd ed.). MIT Press. pp. 308–309. ISBN 978-0-262-03384-8.

Noms des structures imitées en Java

- MyEntry (association key-value)
  - interface Map.Entry<K,V>
- MyTreeNode (association key-value + left-right-parent)
  - class Tree.Entry<K,V>
- MyTreeMap
  - interface Map<K,V>
  - abstract class AbstractMap<K,V>
  - class TreeMap<K,V>
- http://www.docjar.com/html/api/java/util/TreeMap.java.html

Synthèse

- Un dictionnaire (avec arbre ordonné) est fait pour être assez efficace (en temps de calcul) dans :
  - L'ajout
  - Le retrait
  - Dans les 2 cas, il faut descendre sur une branche de l'arbre
- Il reste assez efficace pour des parcours séquentiels
- Pour l'implémentation TreeMap, la place occupée est l'ensemble des valeurs et des clefs + les pointeurs left, right, parent
- Base de code à compléter : https://repl.it/@mariannehuchard/TreeMap