



Bre^{zh} C@mp
Mix de technologies



Hashons peu, mais hashons bien

#HPHB #BzhCmp

@OlivierBourgain & @OlivierCroisier



Olivier BOURGAIN

Freelance
OBMG

@OlivierBourgain

Olivier CROISIER

Freelance
Moka Technologies

@OlivierCroisier
thecodersbreakfast.net



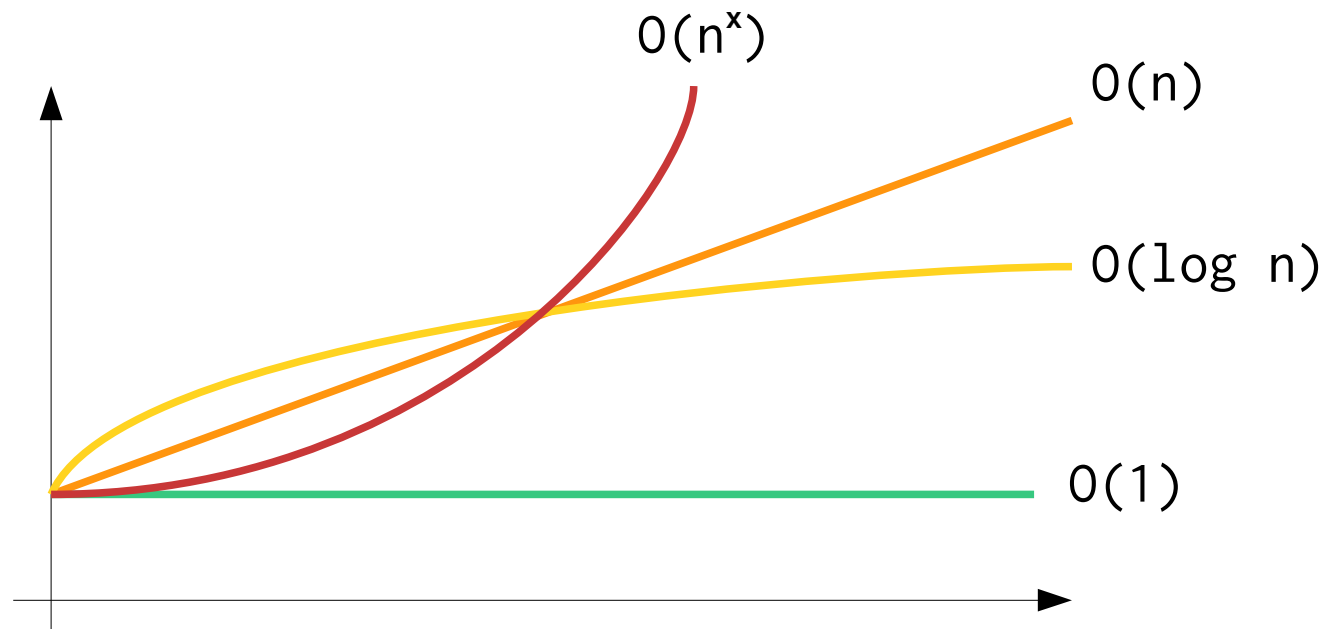


Plan

- Identité
- Hashcode
- Hashcode en action
- Performances



Classes de complexité





2 identités

- Identité *physique*
 - Adresse mémoire
 - Opérateur ==
- Identité *logique*
 - Modélisation métier
 - Méthode equals()



Contrat

- Réflexif
- Symétrique
- Transitif
- Stable
- Non-null

- Cf. javadoc



Guide d'implémentation

- Test ==
 - Test instanceof
 - Conversion
 - Comparaison
-
- `java.util.Objects.equals()`



Guide d'implémentation

```
public class BankAccount {  
  
    private final String bankId;  
    private final long accountId;  
    private BigDecimal amount;  
  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (!(o instanceof BankAccount)) return false;  
        BankAccount other = (BankAccount) o;  
        return Objects.equals(this.bankId, other.bankId)  
            && this.accountId == other.accountId;  
    }  
}
```




Identité immuable

- Champs `final`
- Autre identité, autre instance
- Construction cohérente



Identité composite

- Classe dédiée
- Immuable
- Value-type
- Atomicité



Identité composite

```
public class BankAccount {  
  
    private final BankAccountId id;  
    private BigDecimal amount;  
  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (!(o instanceof BankAccount)) return false;  
        BankAccount other = (BankAccount) o;  
        return this.id.equals(other.id);  
    }  
}
```



Recherche

- Opération fréquente
 - Insertion dans les Sets & Maps
- Recherche linéaire en $O(n)$
- Optimisation ?



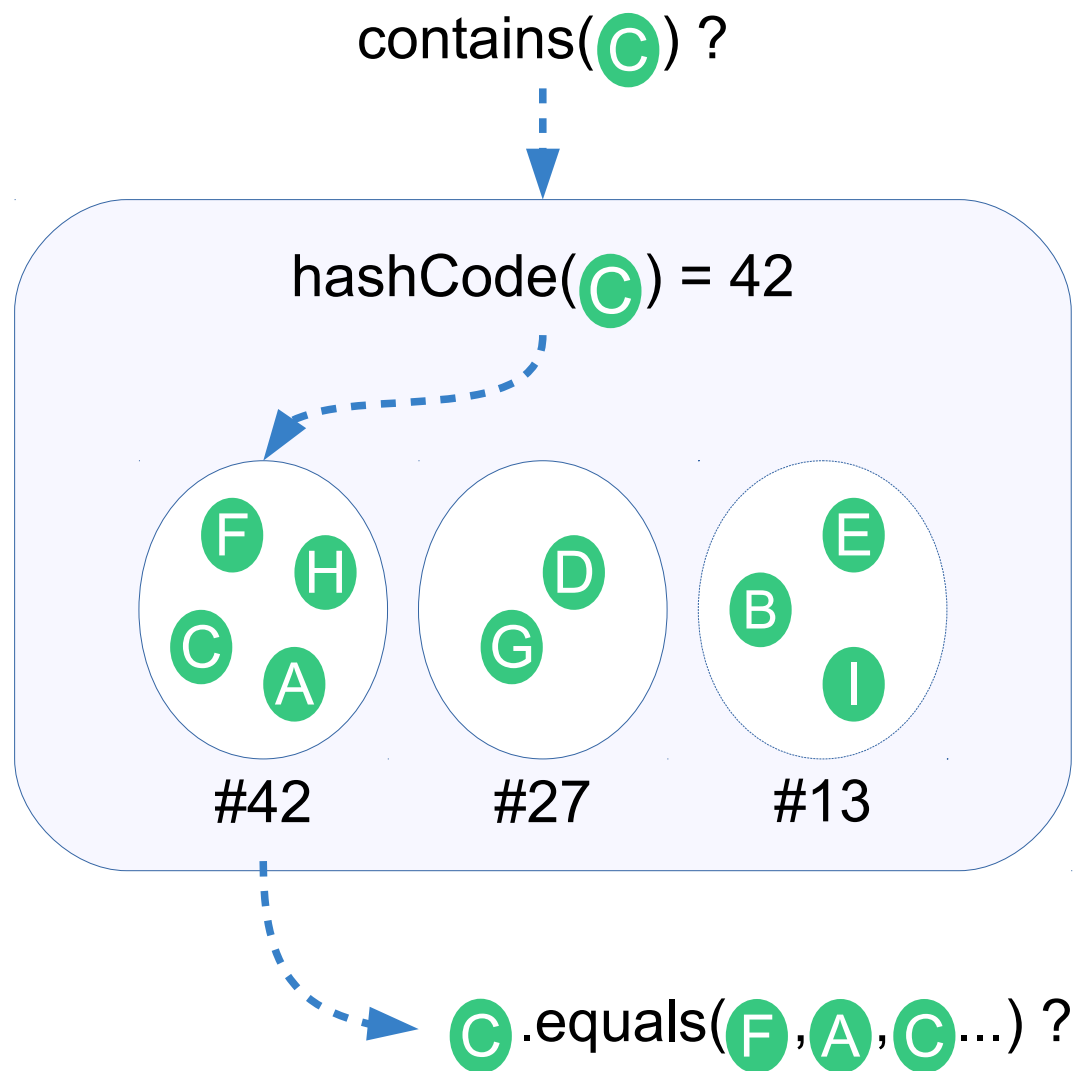
Recherche

- Les structures de données à la rescousse !
 - Arbres
 - Tables de hachage
 - Heaps
 - ...



HashCode

- Fonction de classification
- Méthode hashCode()
- Recherche par identité au sein du groupe
- Performance en $O(1)$
 - Sensible à la qualité de la fonction





Contrat

- Stable
- Cohérente avec l'identité
- Cf. javadoc

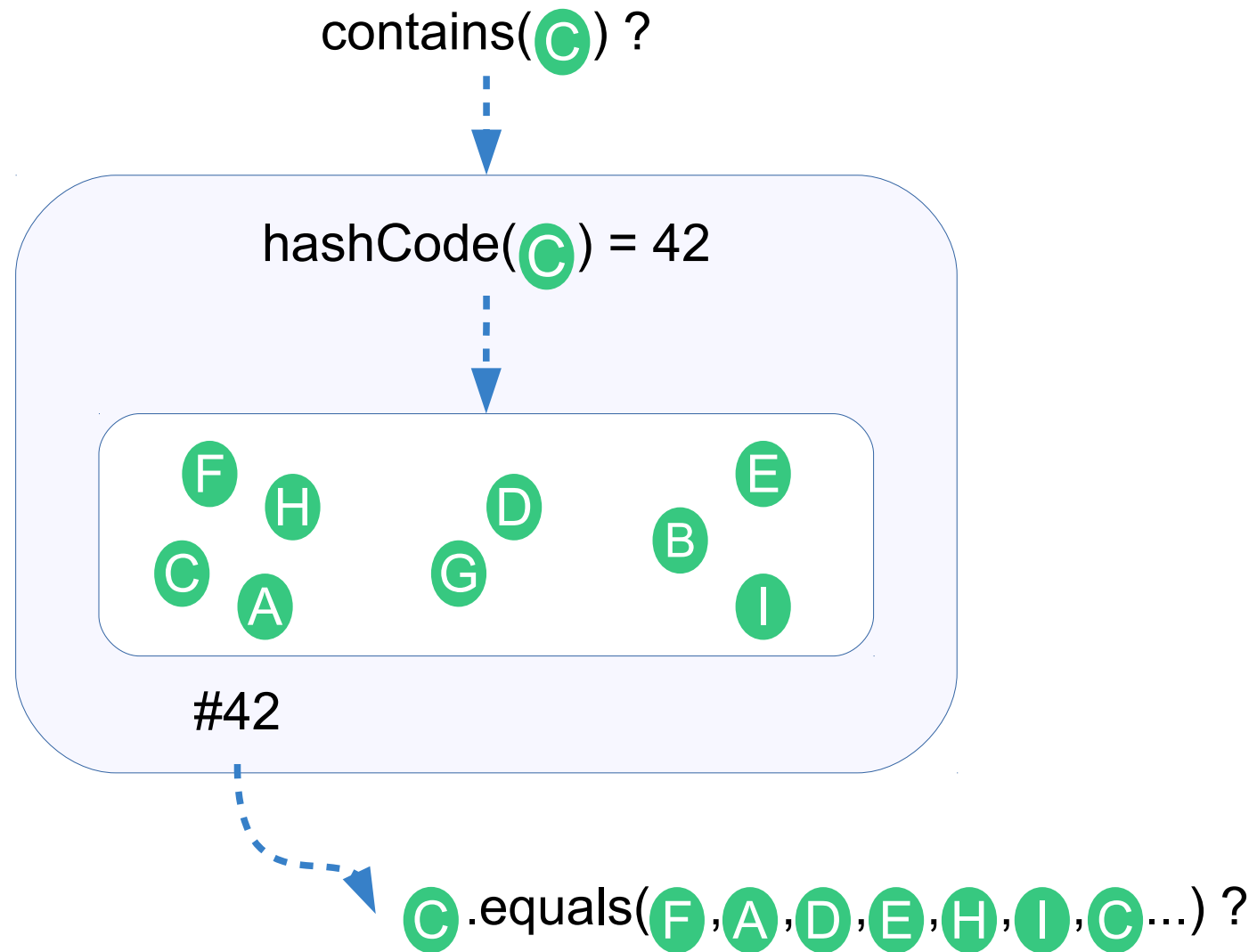


Contrat

- Sous-ensemble de l'identité
 - Cohérence avec `equals()`
- BankAccount (id, balance)
 - `equals()` : id
 - `hashCode()` : id + balance
- `account(42, 1000€)` → groupe A
`account(42, 2000€)` → groupe B !



Cas du hashcode constant





Guide d'implémentation (Java 7+)

- `java.util.Objects.hashCode()`
- `java.util.Objects.hash(Object... fields)`

```
public int hashCode() {  
    return Objects.hash(bankId, accountId);  
}
```



Guide d'implémentation (Java 7+)

Arrays#hashCode

```
public static int hashCode(Object a[]) {  
    if (a == null) return 0;  
  
    int result = 1;  
  
    for (Object element : a) {  
        result = 31 * result +  
            (element == null ? 0 : element.hashCode());  
    }  
    return result;  
}
```



Guide d'implémentation (autres)

- Génération par un IDE ou une librairie
- Bien choisir les champs utilisés
- Attention aux performances
 - Apache Commons ReflectionHashCode



Dans le JDK

- Collections Hash*
- 2 types de structures
 - En buckets (HashMap)
 - Open addressing (IdentityHashMap)



HashMap

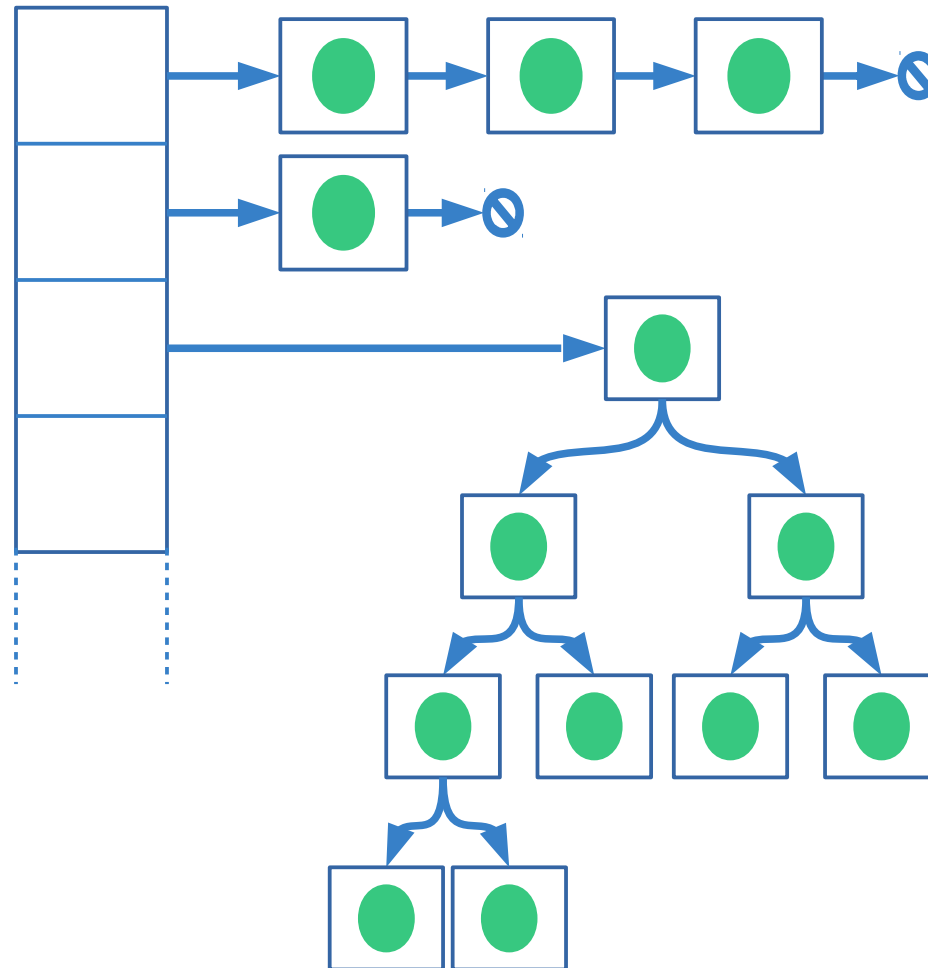
- Structure interne : tableau de buckets
- Bucket de destination
 - Re-hash & bit-shifting

```
int reHash = 0;
if (key != null) {
    int hash = key.hashCode();
    reHash = hash ^ (hash >>> 16);
}
int pos = (size - 1) & reHash;
```



HashMap

- Structure d'un bucket
 - 0-7 éléments : liste chaînée (Nodes)
 - 8+ éléments : arbre binaire (TreeNodes)
- Transformation bi-directionnelle





HashMap

- 16 buckets initiaux
- Load factor 0.75
- Redimensionnement
 - Taille x 2
 - Re-hachage



HashMap

- Performance moyenne en $O(1)$
 - Insertion, recherche
- Dégénérée en $O(n)$ ou $O(\log n)$
- Parcours \propto capacité



IdentityHashMap

- Distinction physique des instances
- Egalité avec `==`
- Hashcode avec `System.identityHashCode()`



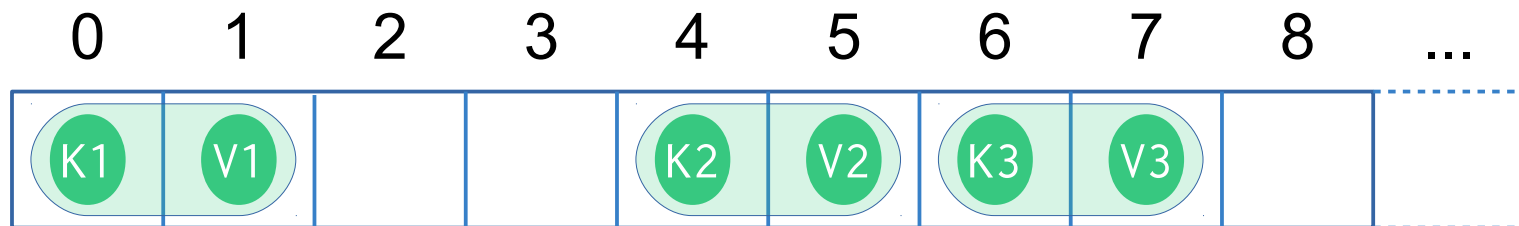
IdentityHashMap

- Structure interne : tableau d'éléments
 - (clé,valeur) aux indices (N,N+1)
- Indice de destination
 - Re-hash
 - Indice suivant si collision (*linear probing*)

```
int hash = System.identityHashCode(element);  
int pos = ((hash << 1) - (hash << 8)) & (length - 1);
```




IdentityHashMap





IdentityHashMap

- Tableau initial de 32 cases (16 éléments)
- Load factor 0.66
- Redimensionnement
 - Taille x 2
 - Re-hachage



IdentityHashMap

- Performance en $O(1)$
 - Insertion, recherche
- Dégénérée en $O(n)$
- Parcours \propto capacité



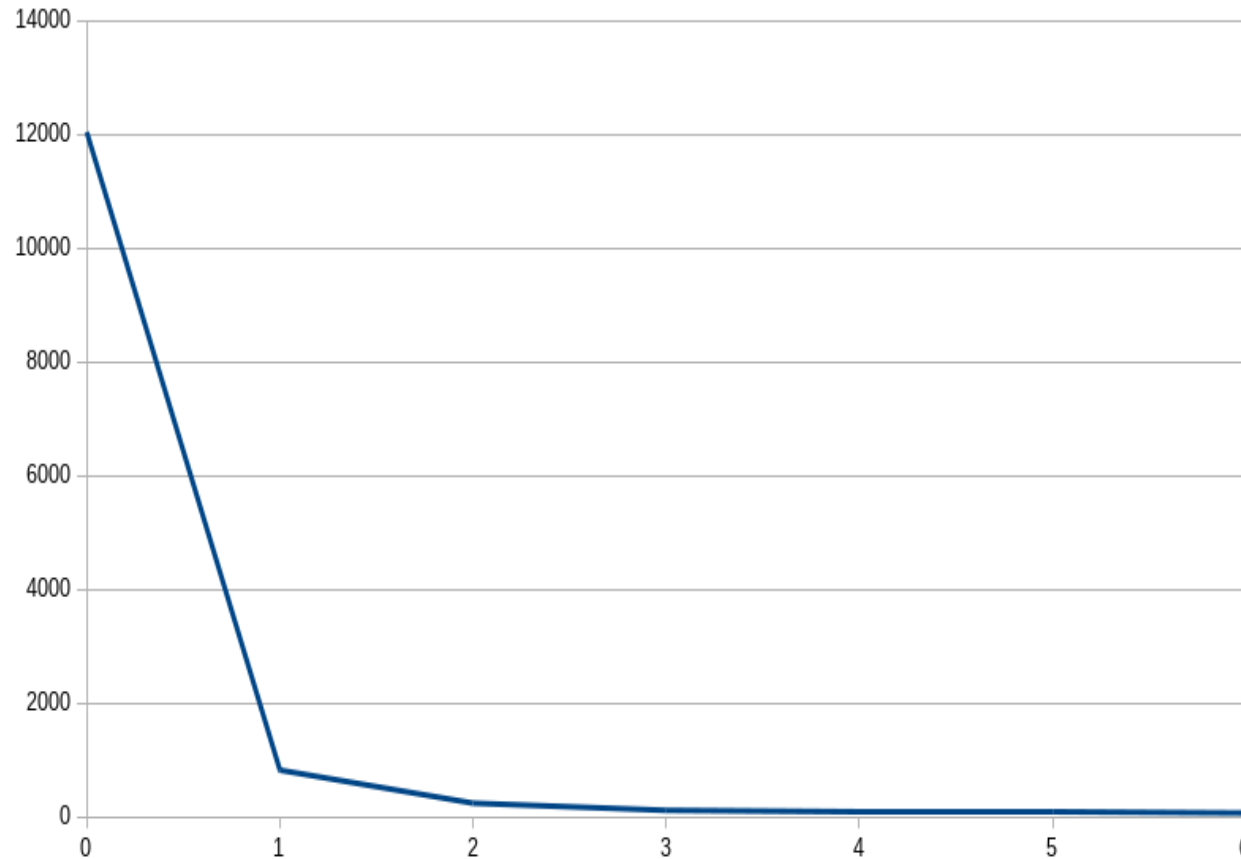
Performances des tables de hachage

- Directement liées à la qualité de la fonction
- Fonction idéale
 - Dispersion importante
 - Très rapide à calculer
- A éviter : constante



Dispersion

- Insertion de 2M Strings dans un HashSet



chars	time(ms)
0	12046
1	828
2	247
3	122
4	92
5	82
6	75

https://github.com/obourgain/hashcode_benchmarks



HashCode par défaut

- `Object.hashCode()`
 - Méthode native
- Mythe : Adresse mémoire ?
 - 6 algorithmes dans OpenJDK 8
 - `-XX:hashCode={0-5}`
 - Par défaut : `-XX:hashCode=5`



Hashcode par défaut

- 0 : Random
- 1 : Manipulation des bits de l'adresse xor random
- 2 : Constante : 1
- 3 : Séquence
- 4 : Adresse mémoire
- 5 : Marsaglia xor-shift



HashCode par défaut

- Est écrit dans le header de l'objet de manière thread-safe
- A des impacts sur la synchronisation
 - Biased locking
 - `System.identityHashCode()`



Pré-dimensionnement

- Minimise les collisions
 - Dégradation \propto taux de collision
- Evite le redimensionnement
 - Garbage
 - Re-hachage

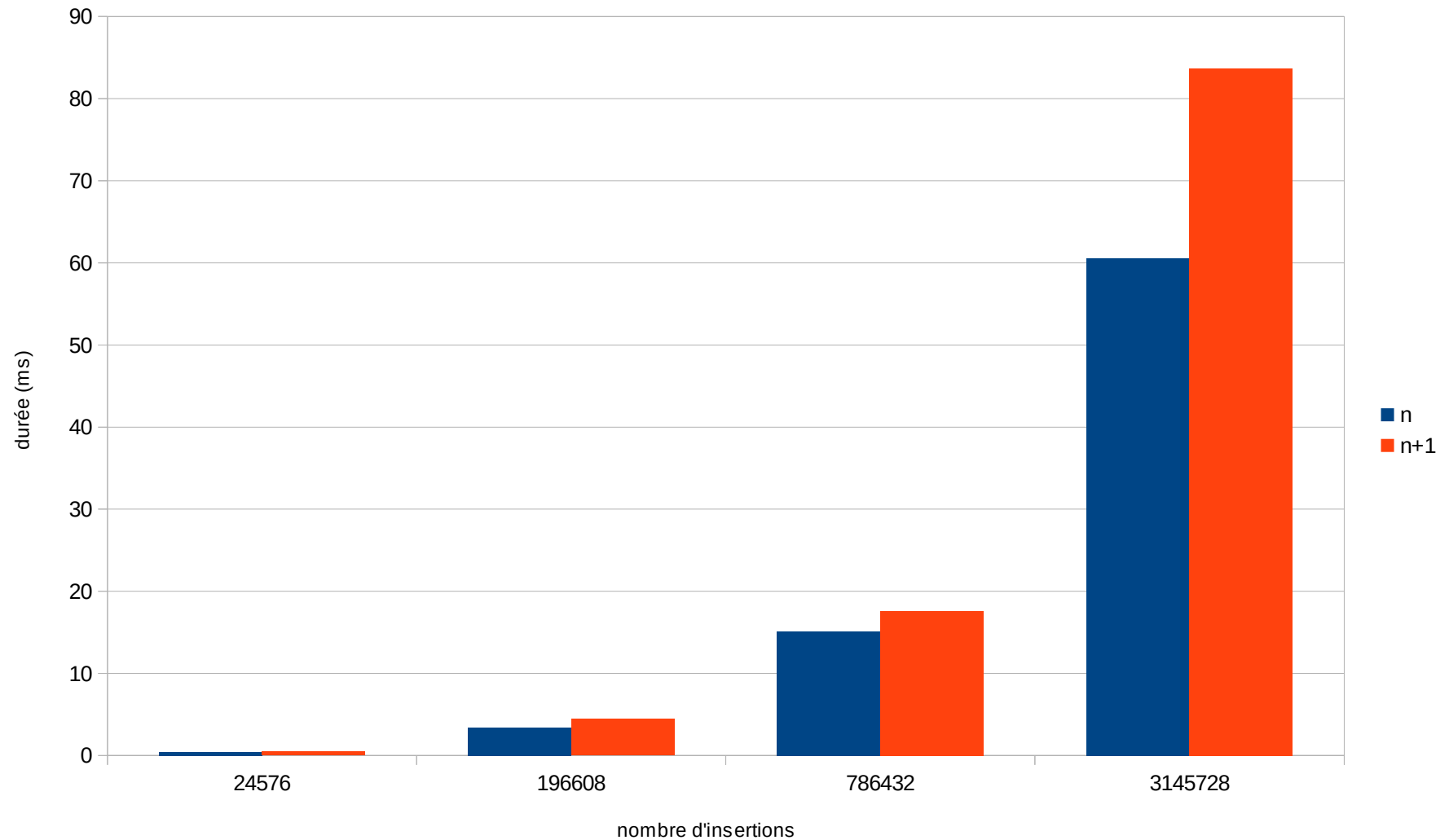


Pré-dimensionnement

- Estimer par excès
 - Eviter le redimensionnement
- Tenir compte des implémentations
 - Load factor
 - Alignement aux puissances de 2



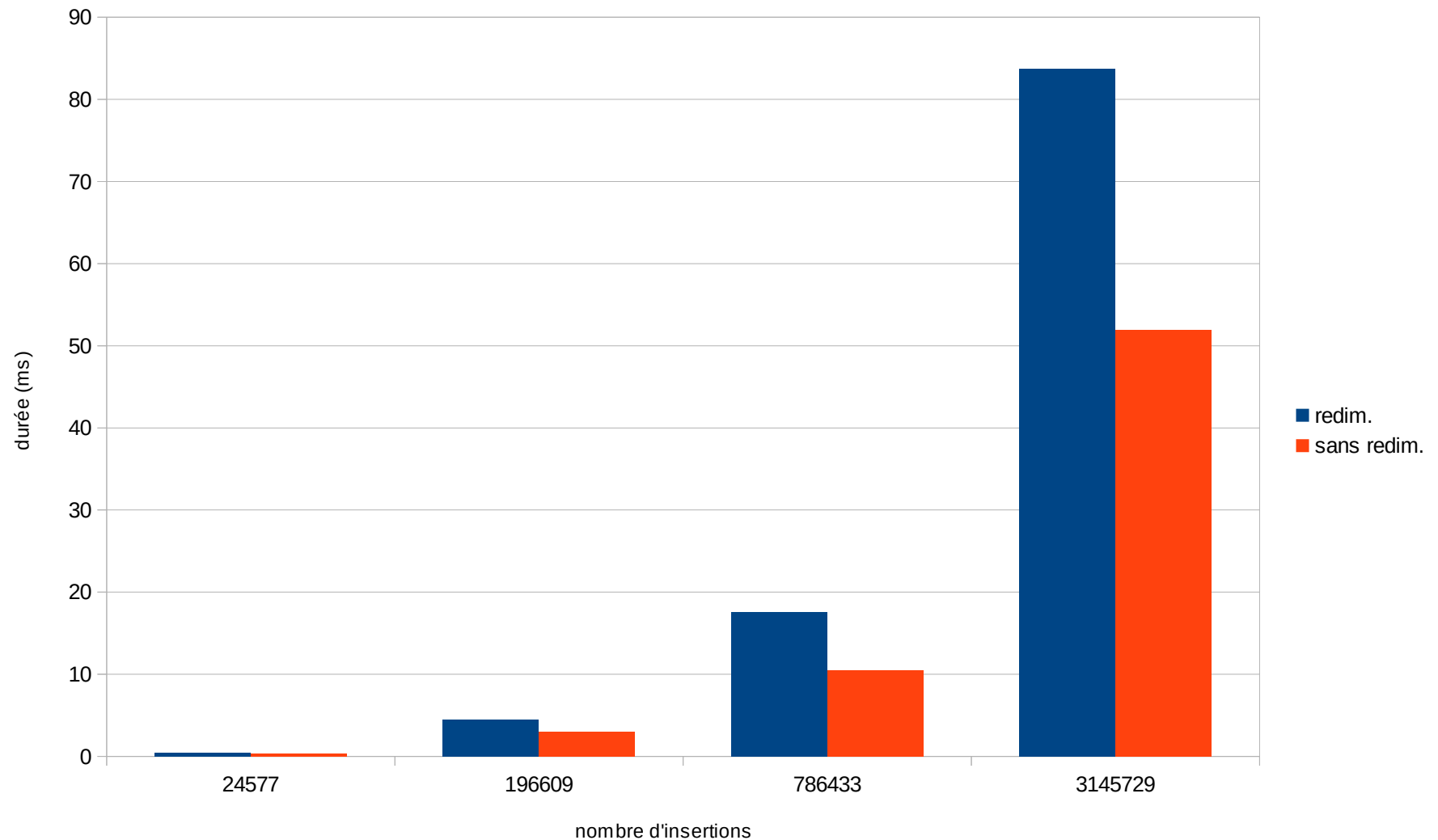
Pré-dimensionnement



https://github.com/obourgain/hashcode_benchmarks



Pré-dimensionnement



https://github.com/obourgain/hashtable_benchmarks



Résolution des collisions

- Chaînage
 - Avec ou sans nœud initial
- Open addressing
 - Linear / quadratic probing
 - Double hashing
- Cuckoo hashing...



Conclusion

- Les méthodes `equals()` et `hashCode()` sont importantes
 - Identité métier
 - Performance et cohérence des collections
- Respectez les contrats
- Etudiez les structures de données
 - Dans le JDK et en-dehors



Olivier BOURGAIN

Freelance

OBMG

@OlivierBourgain

Olivier CROISIER

Freelance

Moka Technologies

@OlivierCroisier

Questions ?

Hashons peu, mais hashons bien