Licence MIASHS

INF F5 — Hachage

UFR SHS Université Grenoble Alpes

Daniel.Bardou@univ-grenoble-alpes.fr

Sommaire

- Introduction
- Dictionnaire à adressage direct
- Dictionnaire haché

Introduction

Introduction

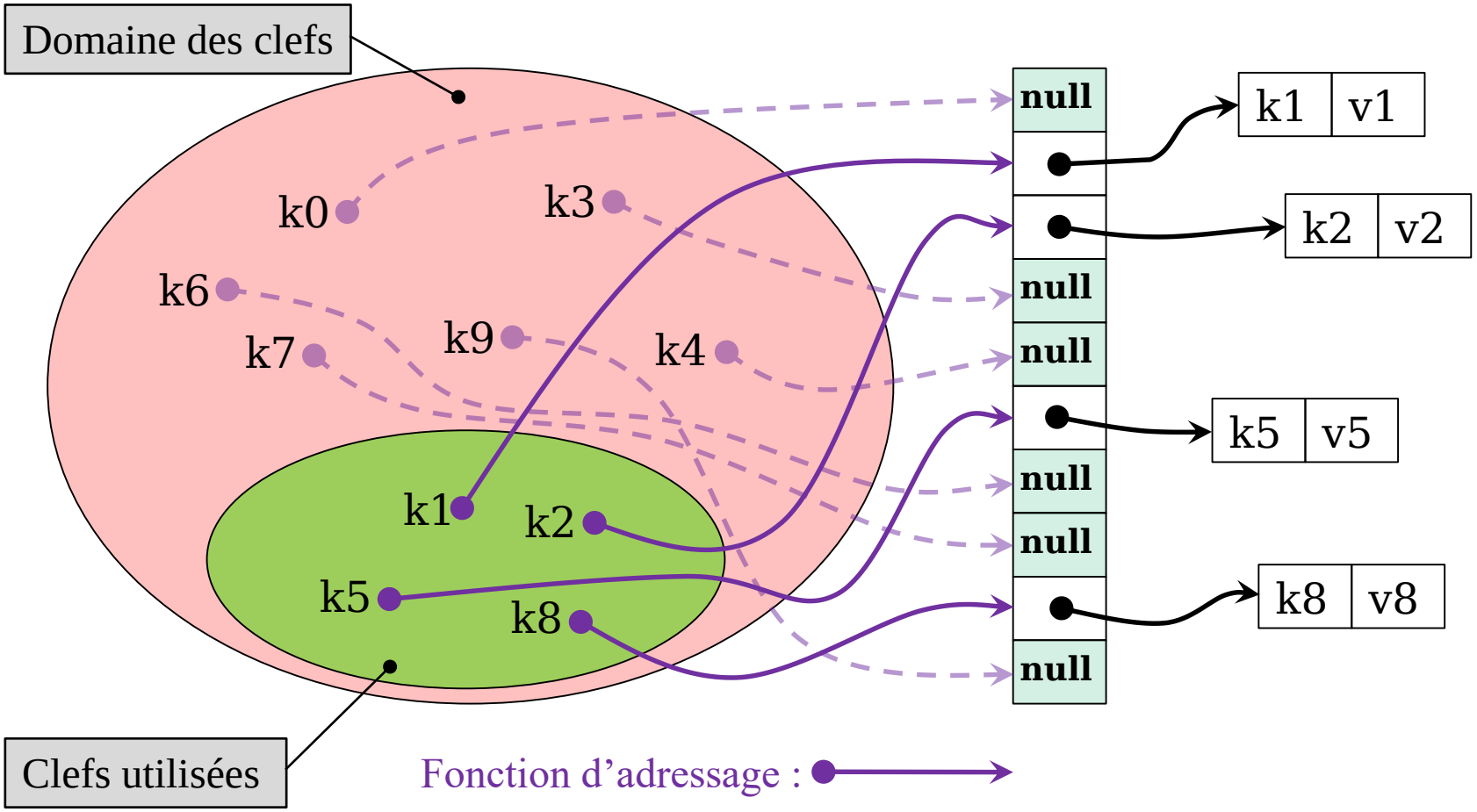
- Afin d'accélérer la recherche de clefs dans un dictionnaire (une `Map` en Java), une idée simple est d'utiliser un tableau pour stocker les associations clef-valeur.
- Si le domaine des clefs (l'ensemble des valeurs possibles pour une clef) est suffisamment petit, on peut mettre en place de l'adressage direct.
  - Il suffit de disposer d'une fonction qui nous donne pour chaque clef un indice unique dans le tableau.
  - On peut alors vérifier la présence d'une clef dans le tableau par un simple accès indicé.
- Si le domaine des clefs est trop grand, on peut mettre en place une technique de hachage et stocker les associations clef-valeur dans une table hachée.

Dictionnaire à adressage direct

Dictionnaire à adressage direct

- Les associations clef-valeur sont stockées dans une table à adressage direct.
- Le domaine des clefs doit être suffisamment petit car la taille de la table correspond à la taille du domaine des clefs.
- On doit disposer d'une fonction d'adressage qui garantit l'unicité des adresses : 2 clefs différentes doivent obligatoirement avoir des adresses différentes.

Table à adressage direct



Exemple de programmation

```
public class DictionnaireAdressageDirect<K, V> {
    private Entree<K, V>[] table;
    private int taille;

    public DictionnaireAdressageDirect(int capacite) {
        if (capacite < 0)
            throw new IllegalArgumentException("capacité incorrecte");
        table = (Entree<K, V>[]) new Entree[capacite];
        taille = 0;
    }

    // ...

}
```

Une classe imbriquée

```
public class DictionnaireAdressageDirect<K, V> {
    // ...

    private class Entree<T, U> {
        T clef;
        U valeur;

        Entree(T clef, U valeur) {
            this.clef = clef;
            this.valeur = valeur;
        }

        public String toString() {
            return clef + "=" + valeur;
        }
    }
}
```

Fonction d'adressage

```
public class DictionnaireAdressageDirect<K, V> {
    // ...

    private int adresse(K clef) {
        int adresse;
        /* un calcul ici qui permet de donner une valeur à adresse en
           fonction de clef */
        if (adresse >= table.length)
            throw new IllegalArgumentException("clef interdite");
        return adresse;
    }

}
```

Recherche

```
public class DictionnaireAdressageDirect<K, V> {
    // ...

    public V get(K clef) {
        Entree<K, V> e = table[adresse(clef)];
        if (e == null)
            return null;
        return e.valeur;
    }

    // ...
}
```

Ajout/Modification

```
public class DictionnaireAdressageDirect<K, V> {
    // ...

    public V put(K clef, V valeur) {
        int i = adresse(clef);
        Entree<K, V> e = table[i];
        table[i] = new Entree<K, V>(clef, valeur);
        if (e != null)
            return e.valeur;
        taille ++;
        return null;
    }

    // ...
}
```

Suppression

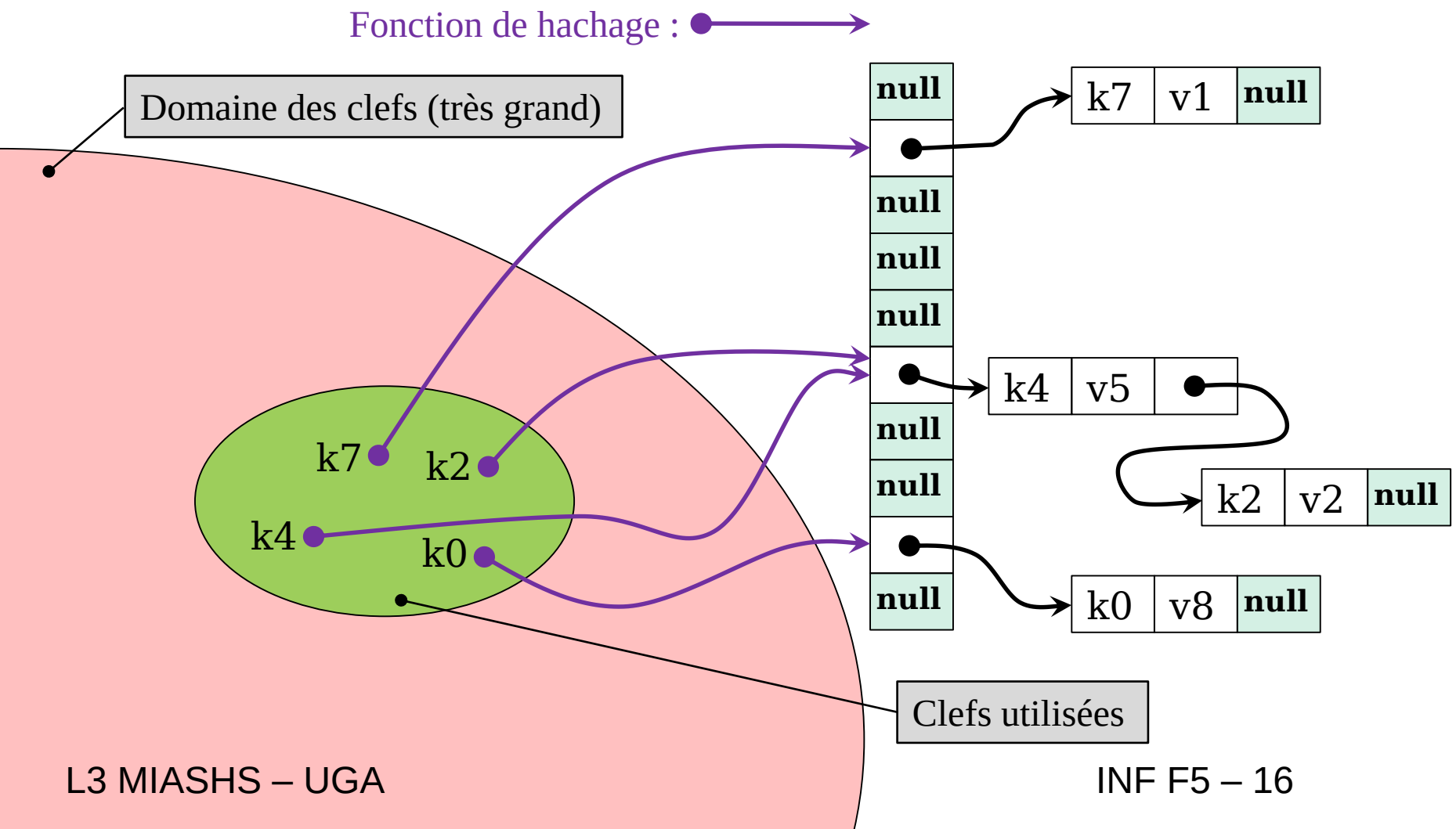
```
public class DictionnaireAdressageDirect<K, V> {
    // ...

    public V remove(K clef) {
        int i = adresse(clef);
        Entree<K, V> e = table[i];
        if (e == null)
            return null;
        table[i] = null;
        taille --;
        return e.valeur;
    }

    // ...
}
```

Dictionnaire haché

Dictionnaire haché

- Les associations clef-valeur sont stockées dans une table hachée.
- Le domaine des clefs peut être beaucoup plus grand que la taille de la table.
- On se repose sur une fonction de hachage qui ne garantit pas l'unicité des adresses : 2 clefs différentes peuvent avoir la même adresse donnée par la fonction de hachage.
- On peut gérer les collisions (inévitables en théorie) par chaînage des entrées.

Table hachée



Paramètres d'une table hachée

- La capacité *C* : c'est la taille de la table.
- Le nombre d'éléments *N* présents dans la table.
- Le facteur de charge maximum *F*. On appelle facteur de charge le rapport entre le nombre d'éléments et la capacité.
- Le seuil *S* est le nombre maximum d'éléments pouvant être présents dans la table avant redimensionnement (re-hachage). $S = C * F$.
- On considère que 0,75 est une bonne valeur pour *F* :
  - si *F* est trop petit, on risque de faire des re-hachages trop fréquents,
  - si *F* est trop grand, on risque d'avoir trop de collisions.

Exemple de programmation

```
public class DictionnaireHache<K, V> {
    private Entree<K, V>[] table;
    private int taille;
    private int seuil;
    private float facteurDeCharge;

    public DictionnaireHache(int capacite, float facteurDeCharge) {
        // gestion des valeurs incorrectes
        table = (Entree<K, V>[]) new Entree[capacite];
        taille = 0;
        seuil = (int) (capacite * facteurDeCharge);
        this.facteurDeCharge = facteurDeCharge;
    }

    // ...

}
```

Une classe imbriquée

```
public class DictionnaireHache<K, V> {
    // ...

    private class Entree<T, U> {
        T clef;
        U valeur;
        Entree<T, U> suivant;

        Entree(T clef, U valeur, Entree<T, U> suivant) {
            this.clef = clef;
            this.valeur = valeur;
            this.suivant = suivant;
        }

        public String toString() {
            return clef + "=" + valeur;
        }
    }
}
```

Fonction de hachage

```
public class DictionnaireHache<K, V> {
    // ...

    private int hachage(K clef) {
        if (clef == null)
            return 0;
        return (clef.hashCode() & 0x7FFFFFFF) % table.length;
    }

}
```

Une méthode utilitaire

```
public class DictionnaireHache<K, V> {
    // ...

    private Entree<K, V> getEntree(K clef) {
        int index = hachage(clef);
        for(Entree<K, V> e = table[index]; e != null; e = e.suivant)
            if ((clef == null && e.clef == null) ||
                    (clef != null && clef.equals(e.clef)))
                return e;
        return null;
    }

}
```

Recherche

```
public class DictionnaireHache<K, V> {
    // ...

    public V get(K clef) {
        Entree<K, V> e = getEntree(clef);
        if (e == null)
            return null;
        return e.valeur;
    }

    // ...
}
```

Ajout/Modification

```
public class DictionnaireHache<K, V> {
    // ...

    public V put(K clef, V valeur) {
        int i = hachage(clef);
        Entree<K, V> e = getEntree(clef);
        if (e != null) { // clef déjà présente, modification
            V res = e.valeur;
            e.valeur = valeur;
            return res;
        }
        // clef absente, ajout
        if (taille >= seuil) {
            rehachage();
            i = hachage(clef);
        }
        table[i] = new Entree<K, V>(clef, valeur, table[i]);
        taille ++;
        return null;
    }
}
```

Rehachage

```
public class DictionnaireHache<K, V> {
    // ...

    private void rehachage() {
        int ancienneCap = table.length;
        int nouvelleCap = ancienneCap * 2 + 1;
        Entree<K, V>[] ancienneTable = table;
        table = (Entree<K, V>[]) new Entree[nouvelleCap];
        seuil = (int) (nouvelleCap * facteurDeCharge);
        for (int i = 0; i < ancienneCap; i ++)
            for (Entree<K, V> e = ancienneTable[i]; e != null; ) {
                Entree<K, V> mobile = e;
                e = e.suivant;
                int index = hachage(mobile.clef);
                mobile.suivant = table[index];
                table[index] = mobile;
            }
    }
}
```

Suppression

```
public class DictionnaireHache<K, V> {
   // ...

   public V remove(K clef) {
      int i = hachage(clef);
      for (Entree<K, V> e = table[i], precedent = null;
                        e != null;
                        precedent = e, e = e.suivant)
         if ((clef == null && e.clef == null) ||
                           (clef != null && clef.equals(e.clef))) {
            if (precedent == null)
               table[i] = e.suivant;
            else
               precedent.suivant = e.suivant;
            taille --;
            return e.valeur;
         }
      return null;
   }
}
```