Programmation avancée

Dictionnaires

ENSIIE

Semestre 2 — 2014–15

Dictionnaire

Types abstraits

Au moment de la conception du programme, on est amené à définir des types abstraits en spécifiant quelles propriétés ils doivent vérifier (spécification fonctionnelle)

- Le développeur du type devra respecter ces propriétés, mais sera libre de l'implémentation concrète
- L'utilisateur pourra supposer que les propriétés sont vérifiées pour son code

De la spécification à l'implémentation

prototypes $\longrightarrow$ interface d'un module

comment vérifier que l'implémentation vérifie les propriétés ?

- test
  - à la main
  - génération de tests

  exhaustivité ?
- certification (preuve)
  - à la main
  - formelle (interactive ou automatique)

  $\pm$ facile suivant le langage

Dictionnaire

Exemple concret :
les dictionnaires (aussi appelés tableaux associatifs ou tables d'association)

On veut associer des clefs $k \in \mathcal{K}ey$ à des valeurs $v \in \mathcal{V}al$

Il est possible

- de créer un dictionnaire vide
- d'insérer une nouvelle association
- de rechercher à quelle valeur est associée une clef
- de supprimer une association

En général, $\mathcal{K}ey$ est supposé totalement ordonné.
Dans la suite, on supposera qu'à une clef n'est associée qu'une seule valeur au maximum (le dictionnaire est une fonction au sens mathématique)

Exemples d'utilisation

Omniprésent en informatique :

- Table des symboles dans un compilateur
  $\mathcal{K}ey=$ symboles, $\mathcal{V}al=$ informations (type, visibilité, ...)
- Système de fichier
  $\mathcal{K}ey=$ chemins, $\mathcal{V}al=$ emplacements disque
- Mémoïsation
  $\mathcal{K}ey=$ arguments, $\mathcal{V}al=$ résultats
- Moteur de recherche
  $\mathcal{K}ey=$ mots-clefs, $\mathcal{V}al=$ pages associées
- Représentation d'un ensemble
  $\mathcal{K}ey=$ élément, $\mathcal{V}al=\{est\_dedans\}$
- ...

Spécification d'un dictionnaire – Interface

```
type ('k,'v) dict

creer : int -> dict
inserer : dict -> 'k -> 'v -> dict
rechercher : dict -> 'k -> 'v
supprimer : dict -> 'k -> dict
```

Spécification d'un dictionnaire – Propriétés

- une recherche dans un dictionnaire vide renvoie une erreur
- une recherche de la clef `k` dans le dictionnaire où l'on vient d'ajouter l'association $\texttt{k} \mapsto \texttt{v}$ renvoie `v`
- une recherche de la clef `k` dans le dictionnaire où l'on vient d'ajouter l'association $\texttt{k'} \mapsto \texttt{v}$ renvoie avec $\texttt{k} \neq \texttt{k'}$ renvoie la même valeur que la recherche avant l'ajout
- une recherche de la clef `k` dans le dictionnaire où l'on vient de retirer l'association à `k` renvoie une erreur
- une recherche de la clef `k` dans le dictionnaire où l'on vient de retirer l'association à `k'` avec $\texttt{k} \neq \texttt{k'}$ renvoie la même valeur que la recherche avant la suppression

Interface : C

```
typedef int key;
typedef char* value;

typedef struct dict_base *dict;

dict creer(int);

value rechercher(dict, key);

dict inserer(dict, key, value);

dict supprimer(dict, key);
```

Interface : OCaml

```
type ('key,'value) dict

val creer : int -> ('key,'value) dict

val rechercher :
  ('key,'value) dict -> 'key -> 'value

val inserer :
  ('key,'value) dict -> 'key -> 'value
    -> ('key,'value) dict

val supprimer :
  ('key,'value) dict -> 'key -> ('key,'value) dict
```

Implémentations

Dans ce cours, trois implémentations

- listes d'association
- arbres bien équilibrés
- table de hachage

Comparaison de la complexité en temps de chacune des fonctions

- en moyenne
- dans le pire des cas

Listes d'association

Implémentation par liste d'association

Type concret : liste contenant des couples (clef, valeur)

- créer : retourner la liste vide
- insérer : ajoute le couple (clef,valeur) en tête de liste
- rechercher : parcourir la liste jusqu'à trouver un couple avec la bonne clef
- supprimer : parcourir la liste et supprimer le couple avec la clef correspondante s'il existe

Complexité

Complexité	Moyenne	Pire
inserer	$O(1)$	$O(1)$
rechercher	$O(n)$	$O(n)$
supprimer	$O(n)$	$O(n)$

Implémentation en OCaml

```
type ('key,'value) dict = ('key * 'value) list

let creer _ = []

let rec rechercher d k =
  match d with
      [] -> raise Not_found
    | (k',v)::_ when k' = k -> v
    | _::q -> rechercher q k

let inserer d k v = (k,v)::d
```

Implémentation en OCaml (suite)

```
let supprimer d k =
  match d with
    [] -> []
  | (k',_)::q when k = k' -> supprimer q k
  | a::q -> a::supprimer q k
```

Implémentation en OCaml (suite)

```
let supprimer d k =
  match d with
    [] -> []
  | (k',_)::q when k = k' -> supprimer q k
  | a::q -> a::supprimer q k
```

Version récursive terminale :

```
let supprimer d k =
  let rec aux accu = function
    | [] -> accu
    | (k',_)::q when k = k' -> aux accu q
    | x::q -> aux (x::accu) q
  in aux [] d
```

Implémentation en C

```
struct dict_base {
  key key;
  value data;
  dict next; };

dict creer(int size) { return NULL; }

value rechercher (dict d, key key) {
  while (d != NULL) {
    if (key == tab->key) return d->data;
    d = d->next;
  };
  return NULL;
}
```

Insérer en C

```
dict inserer(dict d, key k, value v) {
  dict new = malloc(sizeof(struct dict_base));
  new->key = k;
  new->data = v;
  new->next = d;
  return new;
}
```

Supprimer en C

```
dict supprimer(dict d, key k) {
  dict accu = NULL;
  while (d) {
    if (d->key != k)
      accu = inserer(accu, d->key, d->data);
    d = d->next;
  };
  return accu;
}
```

Supprimer en C (version destructive)

```
dict supprimer(dict d, key k) {
  dict i = d;
  dict tmp;
  if (i == NULL) return NULL;
  if (i->key == k) return i->next;
  while(i->next) {
    if (i->next->key==key) {
      tmp = i->next->next;
      free(i->next);
      i->next = tmp;
    }
    else
      i = i->next;  };
  return d; }
```

Correction (OCaml)

- une recherche dans un dictionnaire vide renvoie une erreur

  `creer(i)` est la liste vide, donc `rechercher` renvoie effectivement une erreur

- une recherche de la clef `k` dans le dictionnaire où l'on vient d'ajouter l'association $\texttt{k} \mapsto \texttt{v}$ renvoie `v`

```
inserer d k v = (k,v)::d

rechercher ((k,v)::d) k = v
```

Correction (OCaml)

- une recherche de la clef `k` dans le dictionnaire où l'on vient d'ajouter l'association `k'`$\mapsto$`v` renvoie avec `k`$\neq$`k'` renvoie la même valeur que la recherche avant l'ajout

```
inserer d k v = (k,v)::d

rechercher ((k,v)::d) k' = rechercher d k'
```

Correction (OCaml, suite)

- une recherche de la clef `k` dans le dictionnaire où l'on vient de retirer l'association à `k` renvoie une erreur

  Impossible de poursuivre sans connaître le contenu de d

Preuve par induction

Pour montrer que $P(l)$ est vrai pour toute liste l, il suffit de montrer que

- $P([])$ est vrai
- $P(x :: q)$ est vrai si on suppose que $P(q)$ est vrai

Plus approprié qu'une récurrence sur la longueur de la liste

Peut-être utilisé pour tous les types sommes usuels (arbres, expressions arithmétiques, ...)

Assistant à la démonstration Coq (Inria)

Correction (OCaml, suite)

- une recherche de la clef `k` dans le dictionnaire où l'on vient de retirer l'association à `k` renvoie une erreur

  On procède par induction sur la liste `d` :

  - `d = []` : `supprimer d k = []`
    `rechercher [] k` renvoie une erreur
  - `d = (k',v)::q` :
    - si `k` = `k'`, `supprimer d k = supprimer q k`

      ```
      rechercher (supprimer d k) k
      = rechercher (supprimer q k) k
      ```

    - sinon `supprimer d k = (k',v):: supprimer q k`

      ```
      rechercher (supprimer d k) k
      = rechercher ((k',v) :: supprimer q k) k
      = rechercher (supprimer q k) k
      ```

  Or par hypothèse d'induction,
  `rechercher(supprimer(q,k),k)` renvoie une erreur

Correction (C)

Plus difficile, besoin de prouver un invariant :

- Toutes les listes obtenues uniquement à l'aide des fonctions `creer`, `inserer` et `supprimer` sont bien fondées, c'est-à-dire qu'elles ne comportent pas de cycle.

Outil FramaC (plugin Jessie) (CEA/Inria) : permet d'annoter les programmes avec les propriétés à prouver
Les preuves sont ensuite déléguées à des prouveurs automatiques et/ou interactifs

Arbres bien équilibrés

Recherche par dichotomie

on aimerait avoir des opérations de recherche, d'insertion et de suppression efficaces en moyenne et dans le pire des cas

tableau trié

- recherche en $O(\log n)$ par dichotomie

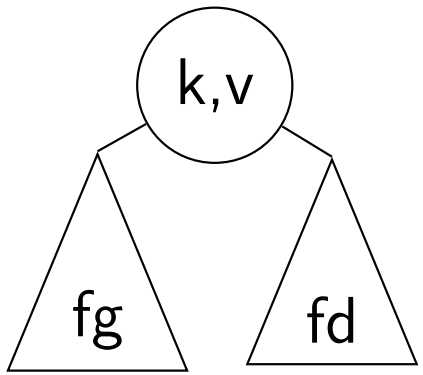
mais

- tri du tableau en $O(n \log n)$
- insertion et suppression coûteuses ($O(n)$, décalages)

Arbres binaires de recherche (ABR)

Structure de données pour exploiter la recherche par dichotomie

Arbre dont les nœuds contiennent des paires (clef, valeur), possédant au plus deux fils.

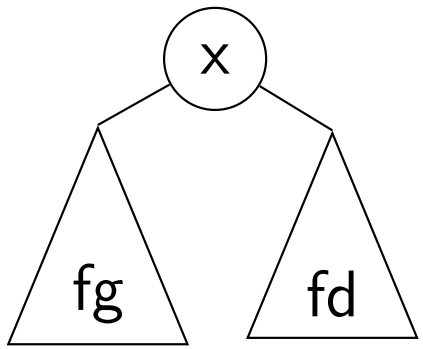


Invariant : pour un nœud

- les clefs dans le sous-arbre gauche fg sont toutes plus petites que k
- les clefs dans le sous-arbre droit fd sont toutes plus grandes que k
- fg et fd vérifie l'invariant

Preuve par induction

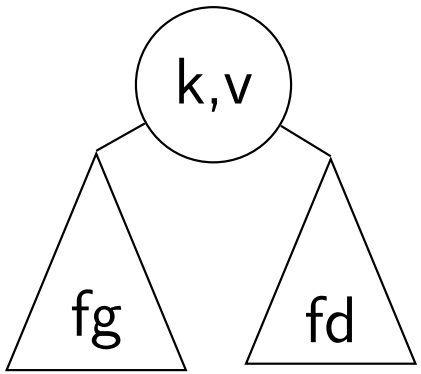
Pour montrer que P est vrai pour tout arbre binaire, il suffit de montrer que

- P est vrai pour l'arbre vide
- P est vrai pour 

si on suppose que P est vrai pour fg et fd

Plus approprié qu'une récurrence sur la hauteur ou la taille de l'arbre

Recherche

rechercher k' dans 

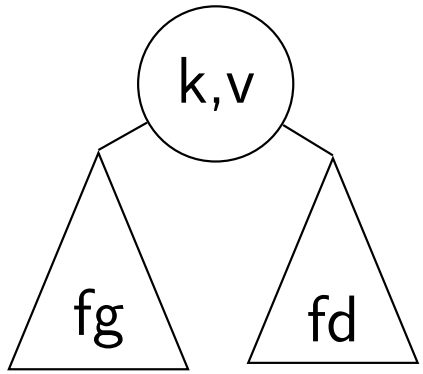
- si k' $=$ k, retourner v
- si k' $<$ k et fg est non-vide, rechercher k' dans fg
- si k' $>$ k et fd est non-vide, rechercher k' dans fd

Complexité : $O(h)$

- à chaque appel récursif, on diminue la hauteur de l'arbre d'au moins 1
- si la bonne clef se trouve au plus bas de l'arbre, il faudra h étapes, où h est la hauteur de l'arbre

Insertion

insérer k',v' dans l'arbre vide : retourner (k',v')

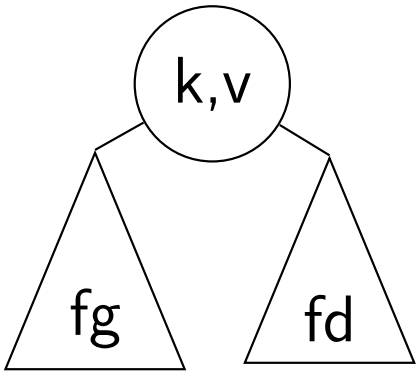
insérer k',v' dans 

- si k' $<$ k, insérer k',v' dans fg
- si k' $>$ k, insérer k',v' dans fd

Complexité : $O(h)$

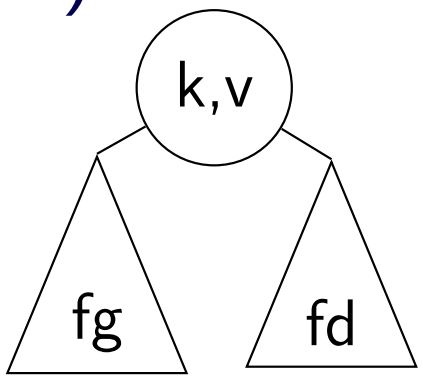
- à chaque appel récursif, on diminue la hauteur de l'arbre d'au moins 1
- on peut avoir à descendre jusqu'au nœud le plus bas

Suppression

Supprimer k dans 

- si fg vide, on retourne fd
- si fd vide, on retourne fg
- sinon,
  - on recherche l'association (k',v') avec la plus grande clef dans fg (celle la plus à droite)
  - on la met à la place de la racine
  - on supprime k' dans fg

Suppression (suite)

Supprimer k' dans 

avec k $\neq$ k' :

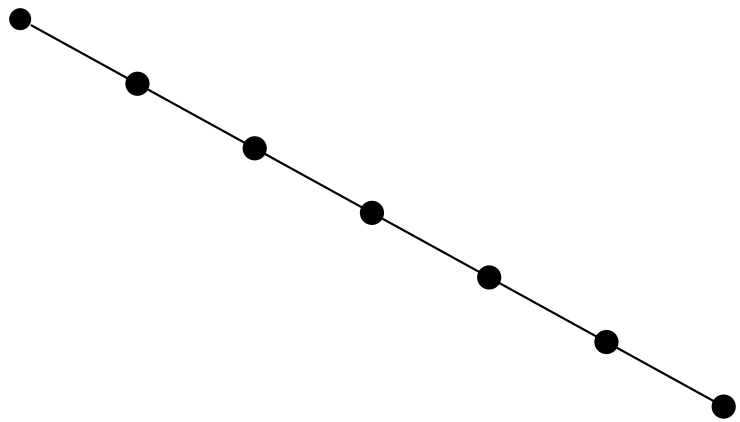
- si k' $<$ k, supprimer k' dans fg
- si k' $>$ k, supprimer k' dans fd

Complexité : $O(h)$

- si la clef à supprimer est dans le nœud le plus à droite, $O(h)$
- donc si la clef à supprimer est à la racine, recherche du plus à droite en $O(h)$ et suppression en $O(h) \Rightarrow O(h)$
- si la clef n'est pas à la racine, on fait un appel récursif sur un arbre de hauteur $h-1$ au maximum $\Rightarrow O(h)$

Complexité dans le pire des cas

La hauteur d'un arbre à n nœuds est n dans le pire des cas :



En particulier, on obtient un arbre de ce type si on insère les éléments par ordre croissant

La complexité dans le pire des cas est $O(n)$ pour rechercher, insérer et supprimer

Complexité en moyenne

La hauteur moyenne d'un arbre à n nœuds est $\sqrt{n}$

Toutefois, si on considère les arbres obtenus en insérant les éléments de $[0 \cdots n-1]$ suivant toutes les permutations possibles, la hauteur moyenne d'un arbre est $\log n$

La complexité en moyenne est donc $O(\log n)$ pour rechercher, insérer et supprimer

Implémentation (OCaml)

```
type ('k,'v) dict =
    Nil
  | Bin of ('k,'v) dict * ('k * 'v) * ('k,'v) dict

let creer _ = Nil

let rec rechercher d k =
  match d with
     Nil -> raise Not_found
  | Bin(_, (k',v), _) when k = k' -> v
  | Bin(fg,(k',_), _) when k < k' -> rechercher fg k
  | Bin(_, (k',_),fd)  -> rechercher fd k
```

Implémentation (OCaml, suite)

```
let rec inserer d k v =
  match d with
      Nil -> Bin(Nil,(k,v),Nil)
    | Bin(_, (k', _), _) when k = k' ->
        failwith "cas non traité"
    | Bin(fg,(k',v'),fd) when k < k' ->
        Bin(inserer fg k v,(k',v'),fd)
    | Bin(fg,(k',v'),fd) ->
        Bin(fg,(k',v'),inserer fd k v)
```

Correction (OCaml)

Une recherche de la clef `k` dans le dictionnaire `d` où l'on vient d'ajouter l'association `k`$\mapsto$`v` renvoie `v`

Pour pouvoir continuer, il faut connaître la forme de `d`
On procède par induction sur `d`

- `d = Nil` : alors `inserer d k v = Bin(Nil,(k,v),Nil)`
  `rechercher (Bin(Nil,(k,v),Nil)) k = v` CQFD

- `d = Bin(fg, k',fd)`
  Par hypothèse d'induction,
  `rechercher(inserer(fg,k,v),k) = v` et
  `rechercher(inserer(fd,k,v),k) = v`
  - si $k < k'$,
    `inserer d k v = Bin(inserer fg k v, k', fd)`
    `rechercher (Bin(inserer fg k v, k', fd)) =`
    `rechercher fg k v`
  - si $k > k'$, symétrique
  - $k \neq k'$ par hypothèse

Implémentation (C)

```
struct dict_base { key key;
                   value val;
                   dict fg;
                   dict fd; };

dict creer(int size) { return NULL; }

dict bin(dict fg, key k, value v, dict fd) {
  dict p = malloc(sizeof(dict_base));
  p->key = k;       p->val = v;
  p->fg = fg;       p->fd = fd;
  return p;
}
```

Implémentation (C, suite)

```
value rechercher(dict d, key k) {
  while(d != NULL) {
    if (d->key == k) return d->val;
    if (d->key > k) d = d->fg;
    else d = d->fd;   };
  return NULL;
}
```

Implémentation (C, version destructive)

```
dict inserer(dict d, key k, value v) {
  if (!d) return bin(NULL,k,v,NULL);
  if (d->key > k)
       d->fg = inserer(d->fg, k, v);
  if (d->key < k)
       d->fd = inserer(d->fd, k, v);
  return d;
}
```

Implémentation (C, version non destructive)

```
dict inserer(dict d, key k, value v) {
  dict res;
  if (!d) res = bin(NULL,k,v,NULL);
  else {
    if (d->key > k)
       res = bin(inserer(d->fg, k, v),
                 d->key, d->val, d->fd);
    if (d->key < k)
       res = bin(d->fd, d->key, d->val,
                 inserer(d->fg, k, v));
  };
  return res;
}
```

Arbres bien équilibrés

Pour rendre les ABR efficaces, il faut minimiser la hauteur par rapport au nombre de nœuds

Définition

Un arbre est bien équilibré si :

- *la différence de hauteur entre ses deux sous-arbres est d'au plus 1*
- *chacun de ses sous-arbres est bien équilibré*

Hauteur d'un arbre bien équilibré

Théorème

Pour un arbre bien équilibré à n nœuds et de hauteur h

$$\log_2(1+n) \leq h \leq \alpha \log_2(2+n)$$

avec $\alpha = \frac{1}{\log_2\left(\frac{1+\sqrt{5}}{2}\right)} < 1,44$

Par conséquent, rechercher, insérer et supprimer sont en $O(\log n)$ dans les arbres bien équilibrés

Arbres auto-équilibrants

Problème : l'insertion et la suppression peuvent déséquilibrer un arbre bien équilibré

- Modifier les fonctions d'insertion et de suppression pour garantir un invariant de bon équilibre

Arbres AVL (Adelson-Velskii et Landis, 1962) : rééquilibrage par rotations

Encapsulation

On veut ne manipuler que des arbres bien équilibrés

- Il faut interdire la création d'arbres non équilibrés à l'extérieur du module

On utilise l'encapsulation liée à la modularité : la seule façon de construire des AVL sera via le constructeur intelligent `bal`

Intérêt : dans `bal` on peut supposer que les arguments sont bien équilibrés, puisqu'ils proviennent des appels aux constructeurs intelligents

Complexité

Après une insertion, une seule rotation (éventuellement double) est nécessaire

Après une suppression, la rotation à effectuer peut entraîner un déséquilibre au-dessus, il y a donc au plus h rotations à effectuer

Comme les rotations sont effectuées en temps constant, les opérations pour rééquilibrer l'arbre sont en $O(\log n)$

Par conséquent, dans un arbre AVL, la recherche, l'insertion et la suppression sont en $O(\log n)$, y compris dans le pire des cas

Tables de hachage

Recherche en temps constant

Pour n très grand, $O(\log n)$ peut être encore trop

Idée : utiliser un tableau en utilisant les clefs comme indices (accès aux éléments en temps constant)

Problème : nombre de clefs possibles potentiellement trop grand
Exemple : Dictionnaire de la langue française
$26^{25} > 10^{35}$ entrées potentielles (en supposant que le mot le plus long de la langue francaise est anticonstitutionnellement)

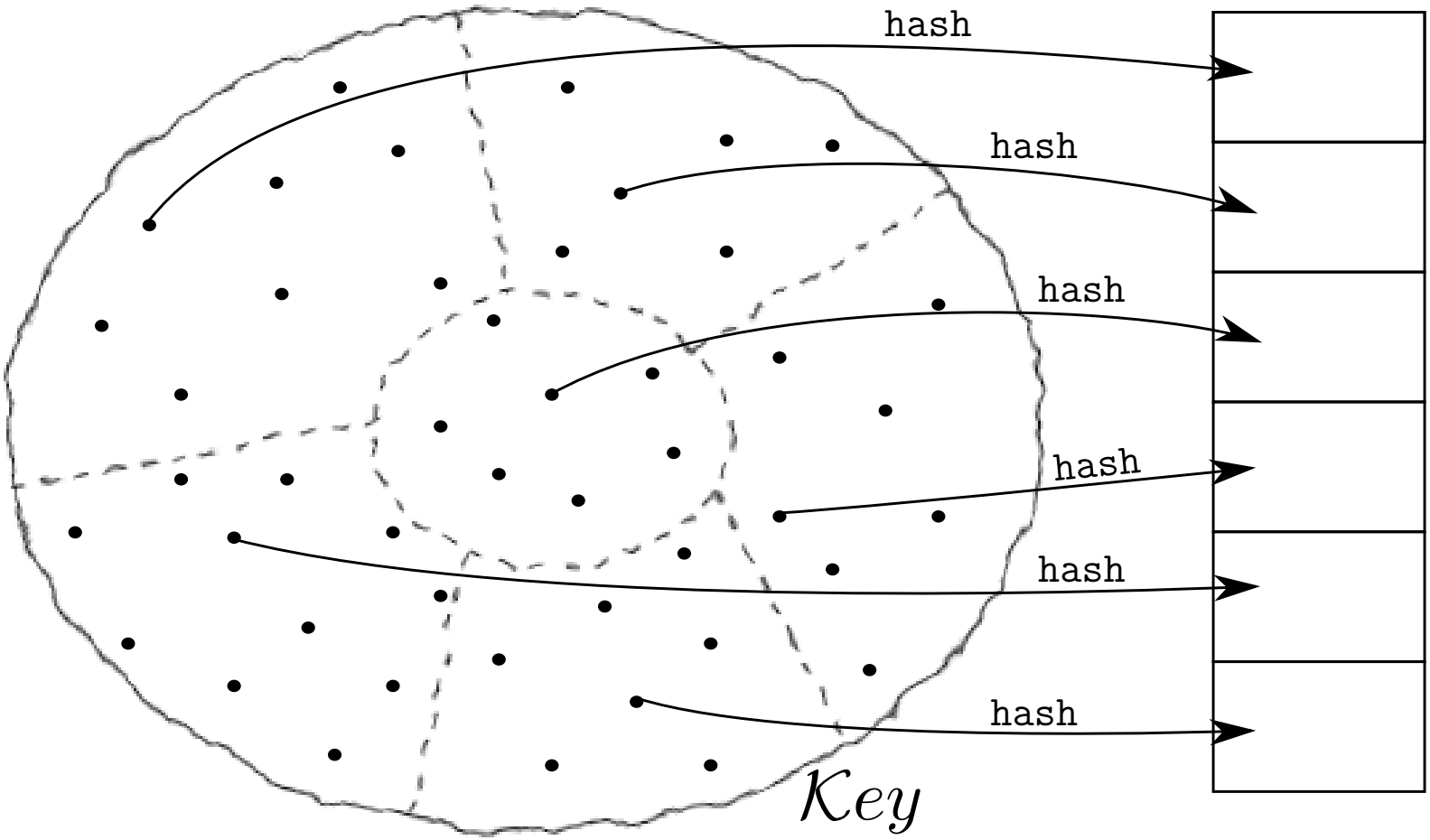
Hachage

L'idée est de restreindre les indices possible en regroupant les clefs.

On considére une fonction `hash` qui va de l'ensemble des clefs dans [0..m-1] pour m approximativement égal au nombre de clés à stocker dans le dictionnaire

On stocke le couple `(key,value)` dans la case `hash(key)` d'un tableau de m éléments

Tables de hachage



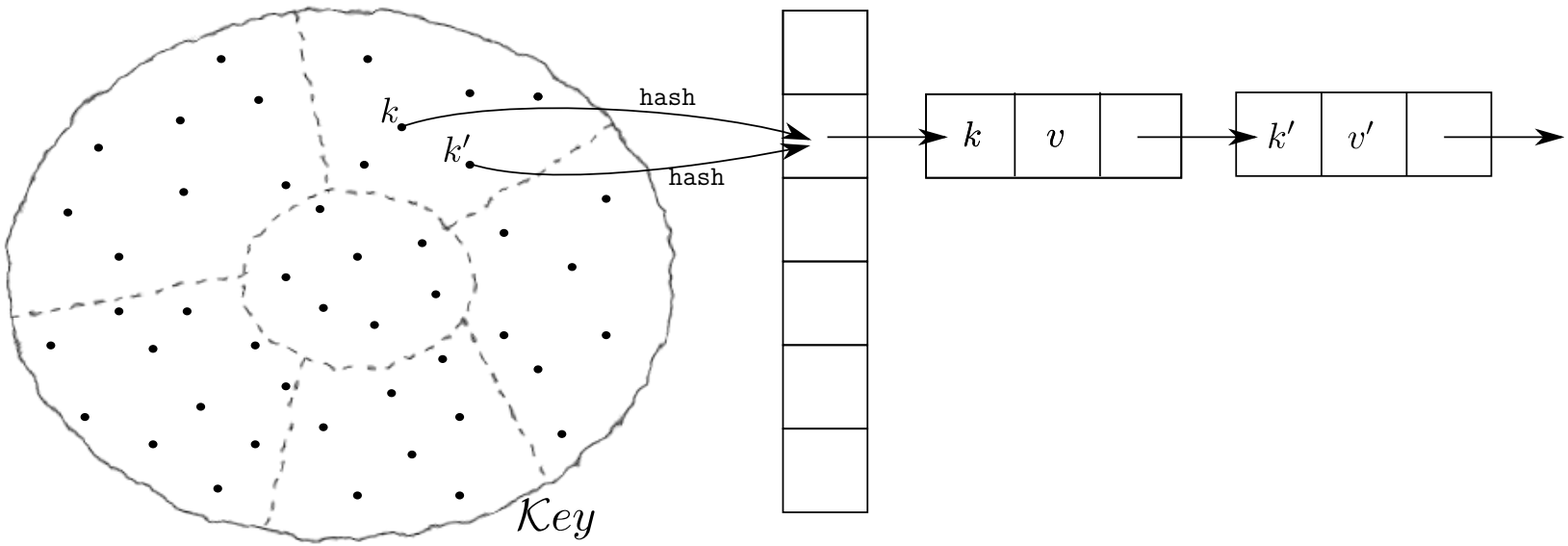
Collision

Comme les clefs potentielles sont bien plus nombreuses que la taille du tableau (c'est qui a motivé la fonction de hachage), la fonction `hash` ne peut être injective :

- plusieurs clefs pour une case du tableau

Solution : mettre dans les cases une liste d'association au lieu d'un unique couple

Tables de hachage



Choix de la fonction de hachage

Pour vraiment gagner par rapport aux listes d'association, il faut limiter les collisions

- le choix de la fonction de hachage est essentiel

Exemple : dictionnaire de la langue française, $\sim 2^{16}$ entrées
si on prend comme fonction de hachage la valeur en ASCII des deux premières lettres, il y aura beaucoup de collisions !
(beaucoup de mots en `ch`, peu en `zx`)

Le choix de la fonction de hachage dépend des clefs et de leur répartition dans l'ensemble des clefs potentielles

Hachage uniforme : pour toute clef k et tout $i \in [0 \cdots m-1]$, la probabilité que $hash(k) = i$ est de $\frac{1}{m}$

Exemple de bonnes fonctions de hachage

Dans le cas où les clefs sont des entiers répartis de façon homogène, on peut utiliser les fonctions de hachage suivantes :

- **Méthode de la division :** on prend $hash(k) = k$ mod m
  Problème : ne marche bien que si m est un nombre premier éloigné d'une puissance de 2

- **Méthode de la multiplication**
  On considère une constante réelle $0 < A < 1$
  On prend la partie fractionnaire $f = kA - \lfloor kA \rfloor$ de $k \times A$
  On retourne la partie entière de m $\times f$

  En pratique, on choisit pour m une puissance de 2 pour avoir une version plus efficace de l'algorithme ci-dessus
  La valeur $A = \frac{\sqrt{5}-1}{2}$ donne de bons résultats

Structure de données

En OCaml :

```
type ('k,'v) dict = (('k,'v) Liste_assoc.dict) array
```

- Réutilisation

En C :

```
struct bucket = {
  key key;
  value val;
  struct bucket* next;  };

struct dict_base {
  unsigned int taille;
  struct bucket** contenu;  };
```

Création

`creer(i)`

- On crée un tableau de taille `i` dont les éléments sont des listes chaînées contenant des couples clef, valeur

En OCaml :

```
let creer i = Array.make i (Liste_assoc.creer 2)
```

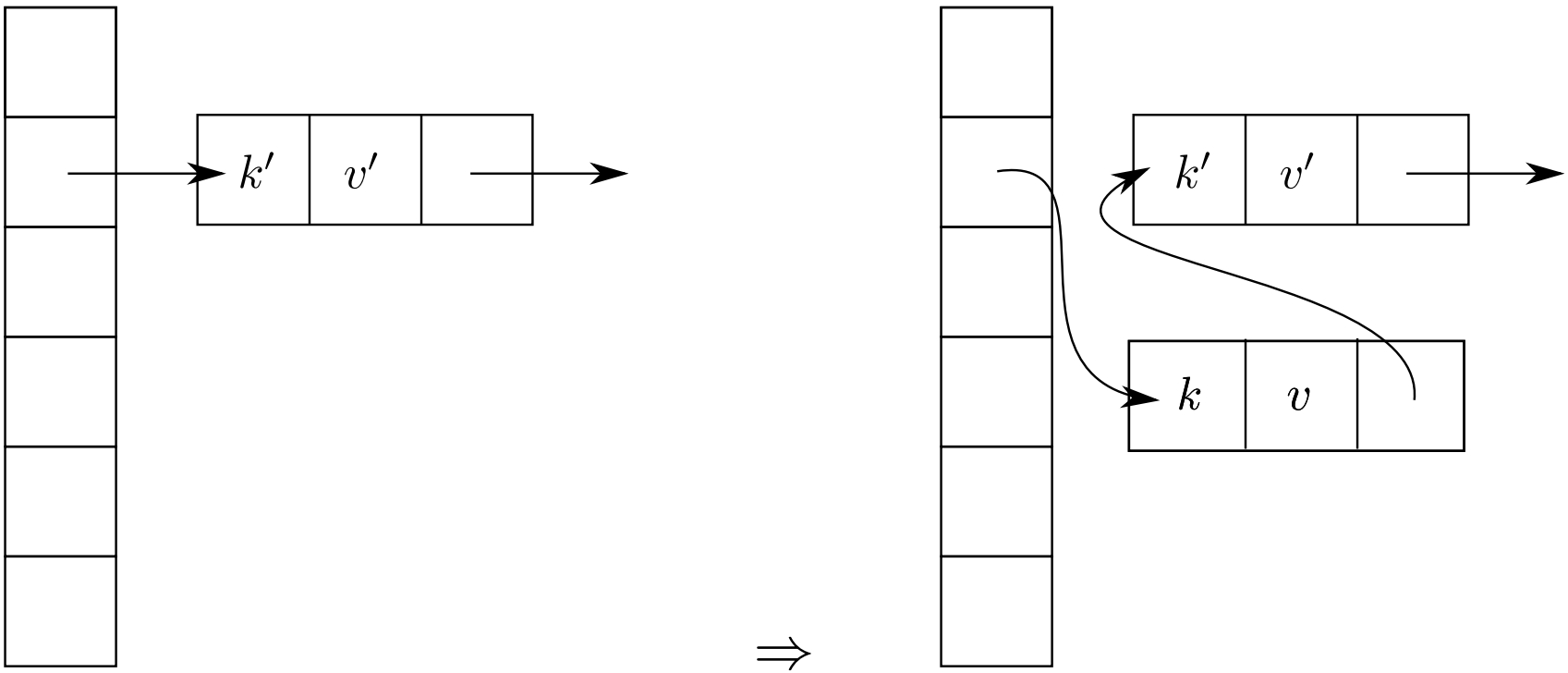
En C :

```
dict creer(int i) {
  int j;
  dict res = malloc(sizeof(struct dict_base));
  res->taille = i;
  res->contenu = calloc(i, sizeof(struct bucket*));
  for (j=0; j<i; j++) res->contenu[j] = NULL;
  return res;  }
```

Insertion

`inserer(d,k,v)`

- on calcule `hash(k)`
- on ajoute le couple `k,v` en tête de la liste chaînée à la position `hash(k)` du tableau



Implémentation (insertion)

En OCaml :

```
let inserer d k v =
  let h = hash k mod Array.length d in
  d.(h) <- Liste_assoc.inserer d.(h) k v;
  d
```

En C :

```
dict inserer(dict d, key k, value v) {
  unsigned int h = hash(k) % d->taille;
  d->contenu[h] = cons(k, v, d->contenu[h]);
  return d;
}
```

Recherche

`rechercher(d,k)`

- on calcule `hash(k)`
- on recherche un couple `k,v` dans la liste chaînée à la position `hash(k)` du tableau

Implémentation (recherche)

En OCaml :

```
let rechercher d k =
  let h = hash k mod Array.length d in
  Liste_assoc.rechercher d.(h) k
```

En C :

```
dict rechercher(dict d, key k) {
  unsigned int h = hash(k) % d->taille;
  struct bucket* b = d->contenu[h];
  while (b != NULL) {
    if (b->key == k) return b->val;
    b = b->next;   }
  return NULL;
}
```

Suppression

`supprimer(d,k)`

- on calcule `hash(k)`
- on supprimer les couple `k,v` dans la liste chaînée à la position `hash(k)` du tableau

En OCaml :

```
let supprimer d k =
  let h = hash k mod Array.length d in
  d.(h) <- Liste_assoc.supprimer d.(h) k
```

En C :

```
dict supprimer(dict d, key k) {
  unsigned int h = hash(k) % d->taille;
  struct bucket* b = d->contenu[h];
  if (b == NULL) return d;
  if (b->key == k)
    d->contenu[h] = b->next;
  while (b->next != NULL) {
    if (b->next->key == k)  b->next = b->next->next;
    else  b = b->next;   };
  return d;  }
```

Complexité

Complexité	Moyenne	Pire
inserer	$O(1)$	$O(1)$
rechercher	$O(1+\alpha)$	$O(n)$
supprimer	$O(1+\alpha)$	$O(n)$

où $\alpha = \frac{n}{\mathsf{m}}$

Le cas le pire est quand on n'a que des collisions

Pour la complexité en moyenne, on suppose que la fonction de hachage est uniforme

Redimensionnement dynamique

Pour obtenir une complexité constante en moyenne, on peut faire grossir le tableau quand les entrées sont trop nombreuses (typiquement quand $n >$m)

- On créer un nouveau tableau de taille 2m
- On insère les anciennes associations dans le nouveau tableau, à l'aide d'une fonction de hachage sur [0..2m-1]

Coût de la copie en $O(n)$, mais n'est nécessaire que pour $n = 2^k$

- en moyenne, coût de l'insertion, de la recherche et de la suppression en $O(1)$

Résumé

	en moyenne		
	rechercher	insérer	supprimer
listes d'association	$O(n)$	$O(1)$	$O(n)$
ABR	$O(\log n)$	$O(\log n)$	$O(\log n)$
arbres AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$
tables de hachage	$O(1)$	$O(1)$	$O(1)$

	dans le pire des cas		
	rechercher	insérer	supprimer
listes d'association	$O(n)$	$O(1)$	$O(n)$
ABR	$O(n)$	$O(n)$	$O(n)$
arbres AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$
tables de hachage	$O(n)$	$O(n)$	$O(n)$

Compactage

L'utilisation de listes chaînées est assez gourmande en mémoire, et nécessite des allocations dynamiques coûteuses en temps

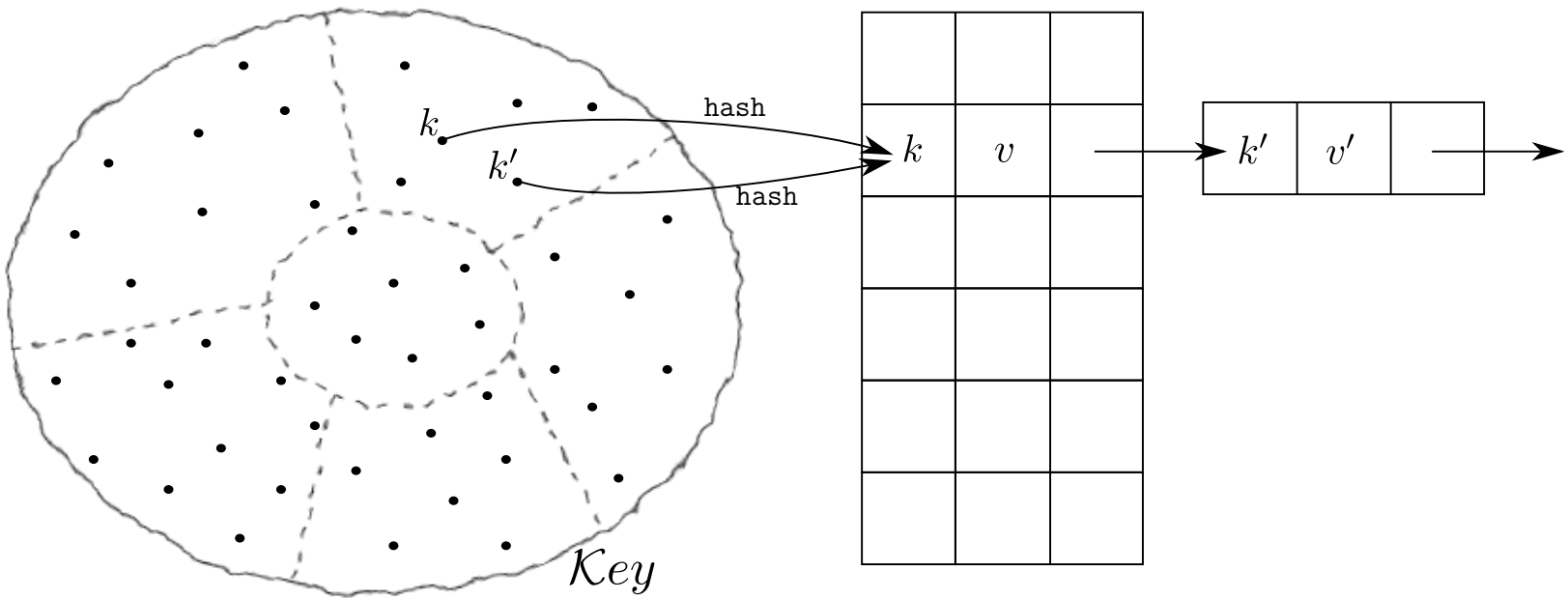
Si on a n éléments dans une table de hachage de taille n, on aimerait ne pas avoir à utiliser plus de n fois la taille d'une clef et de la valeur correspondante.

- On utilise le tableau lui même pour stocker les associations qui seraient à la suite dans la liste chaînée

On parle alors de **table de hachage fermée**

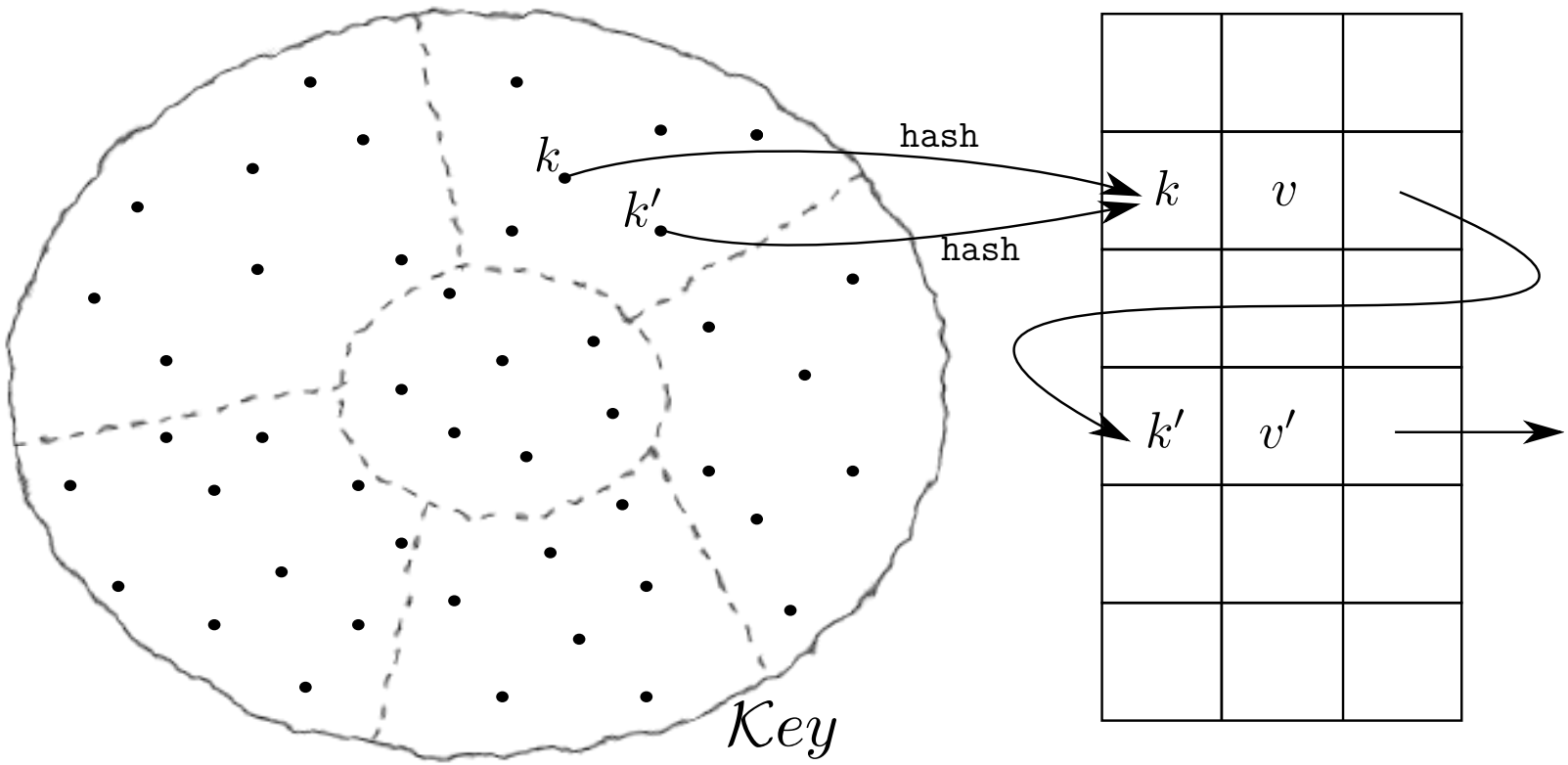
1re idée

Le tableau ne contient pas des pointeurs vers des listes chaînées, mais les premiers éléments de la liste



2e idée

On utilise le tableau pour les éléments suivants de la liste chaînée

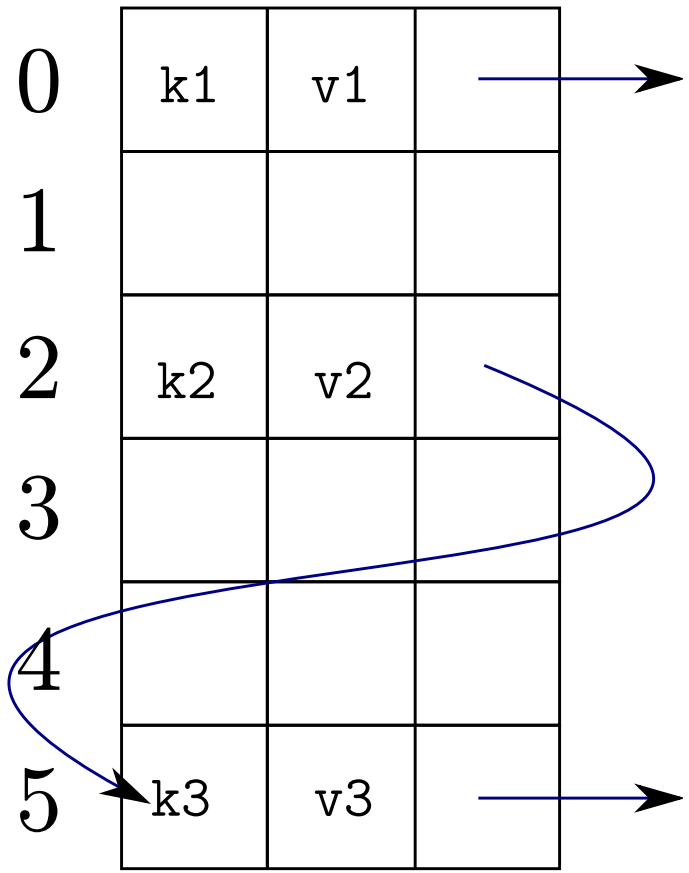


Rechercher : exemple 1

`rechercher(d,k)`

- `hash(k) = 1`

Pas trouvée

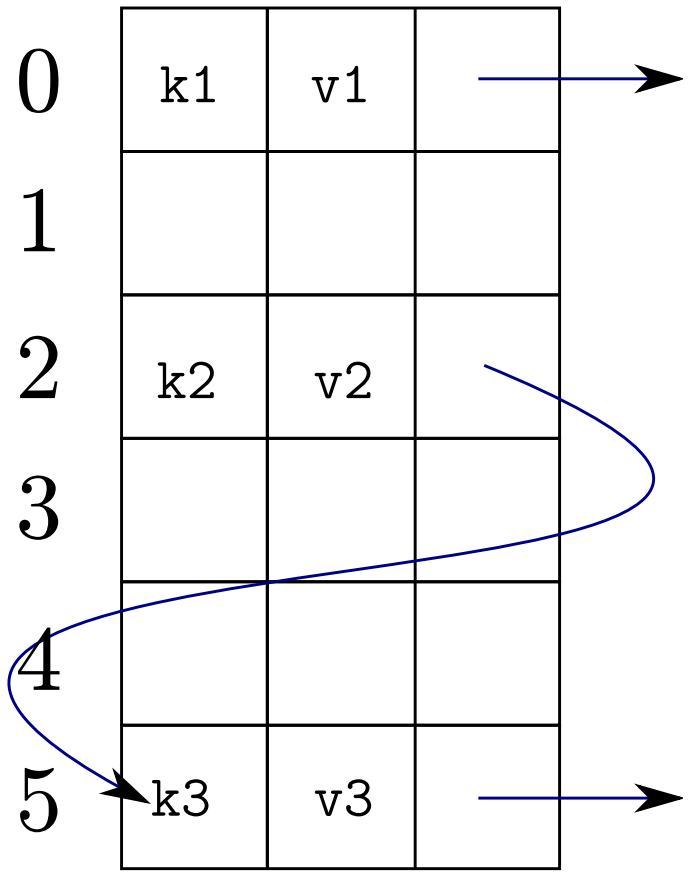


Rechercher : exemple 2

`rechercher(d,k3)`

- `hash(k3) = 2`

On suit les liens jusqu'à trouver la clef (ou non)

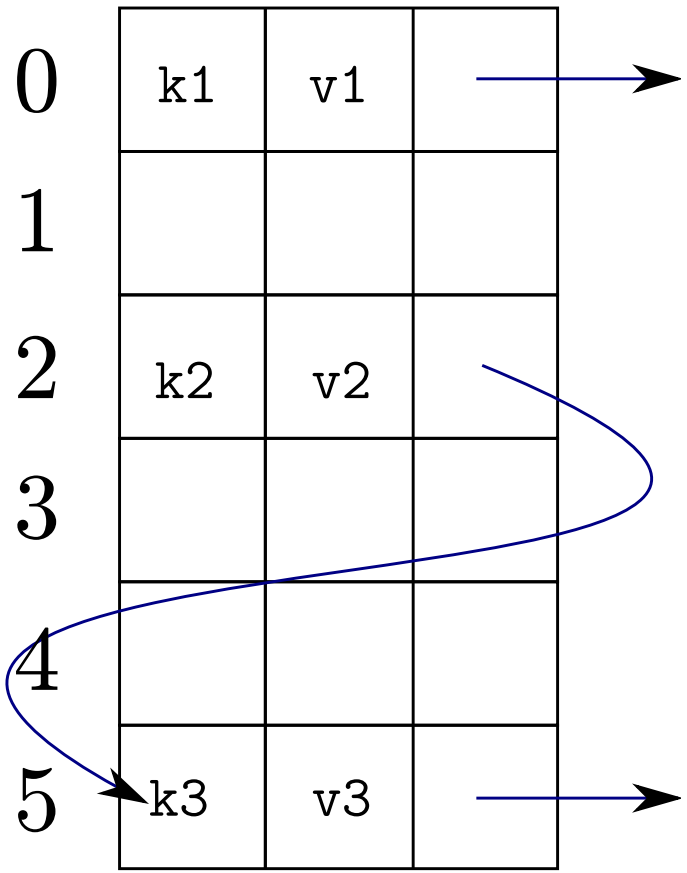


Rechercher : exemple 3

`rechercher(d,k')`

- `hash(k')` $= 5 \neq$ `hash(k3)`

Pas trouvée



Orbite

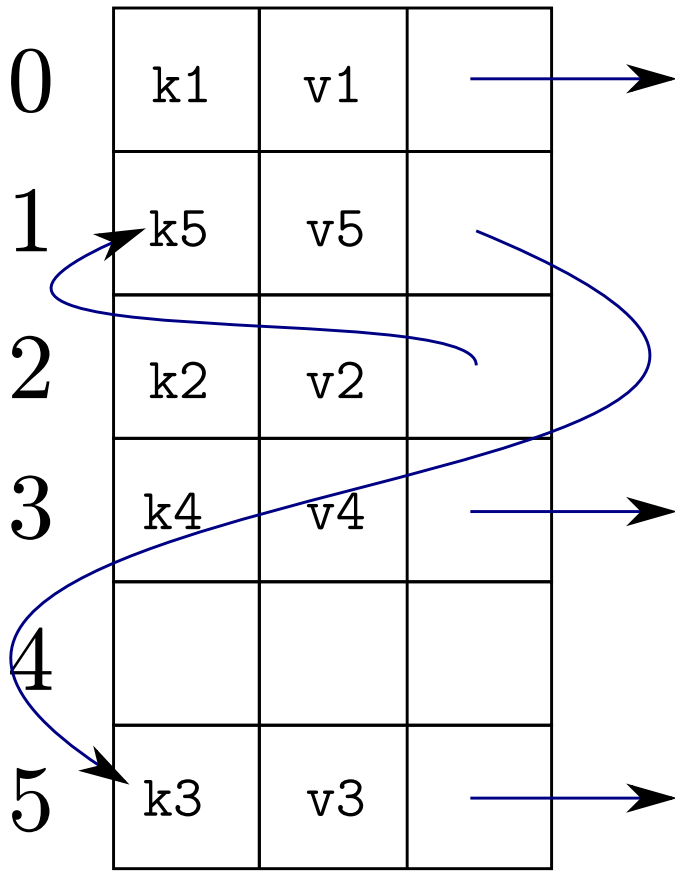
Deux associations `(k,v)` et `(k',v')` sont dans la même **orbite** si `hash(k) = hash(k')`

Orbites :

`{(k1,v1)}`

`{(k2,v2);(k5,v5);(k3,v3)}`

`{(k4,v4)}`



Recherche

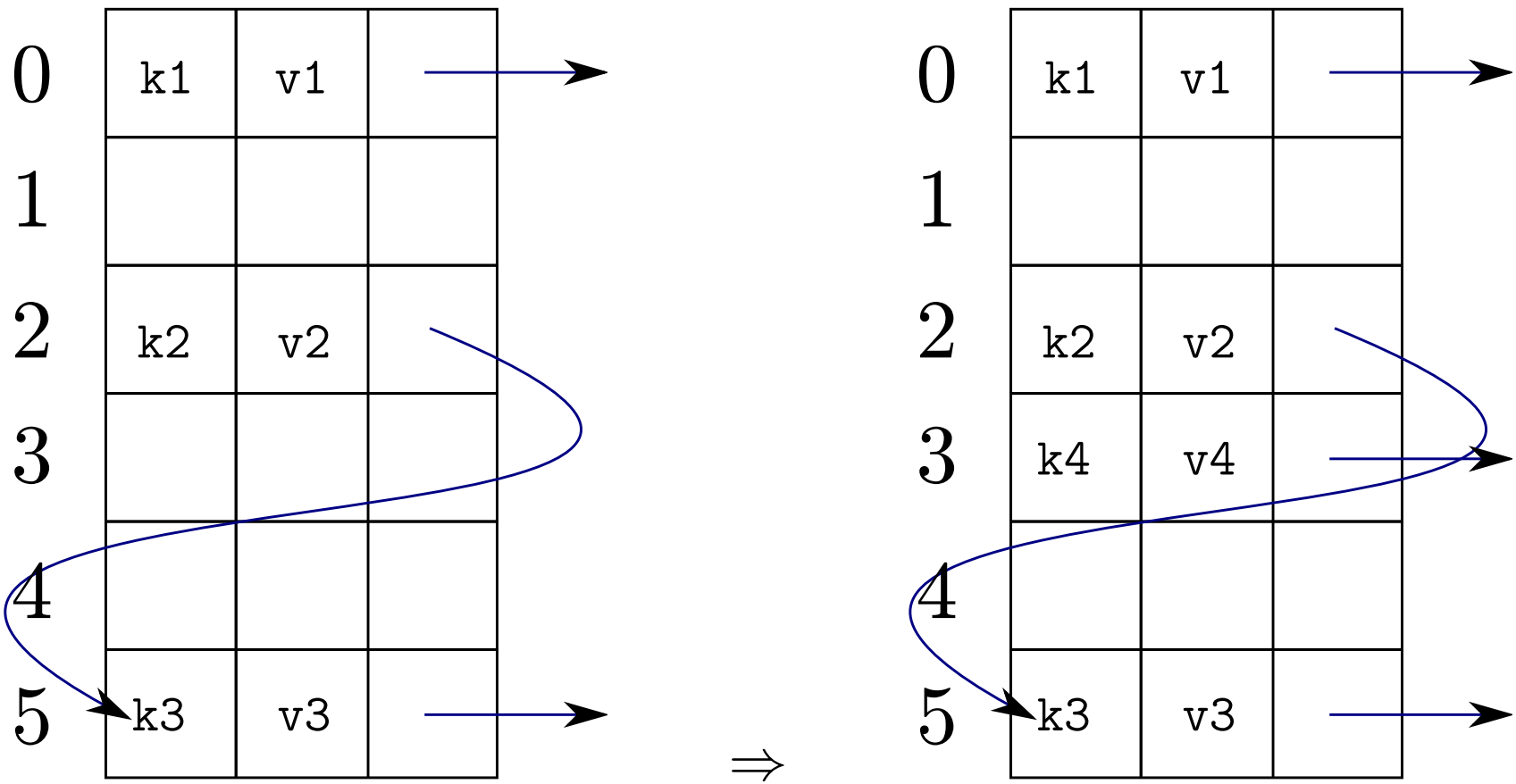
`rechercher(d,k)`

- on calcule `hash(k)`
- si la case en `hash(k)` est vide
  - clef non trouvée
- si la case en `hash(k)` contient une clef de hachage différent de `hash(k)`
  - clef non trouvée
- sinon
  - si la clef est `k`, on renvoie la valeur
  - sinon on va à la case indiquée dans le successeur

Insertion : exemple 1

`inserer(d,k4,v4)`

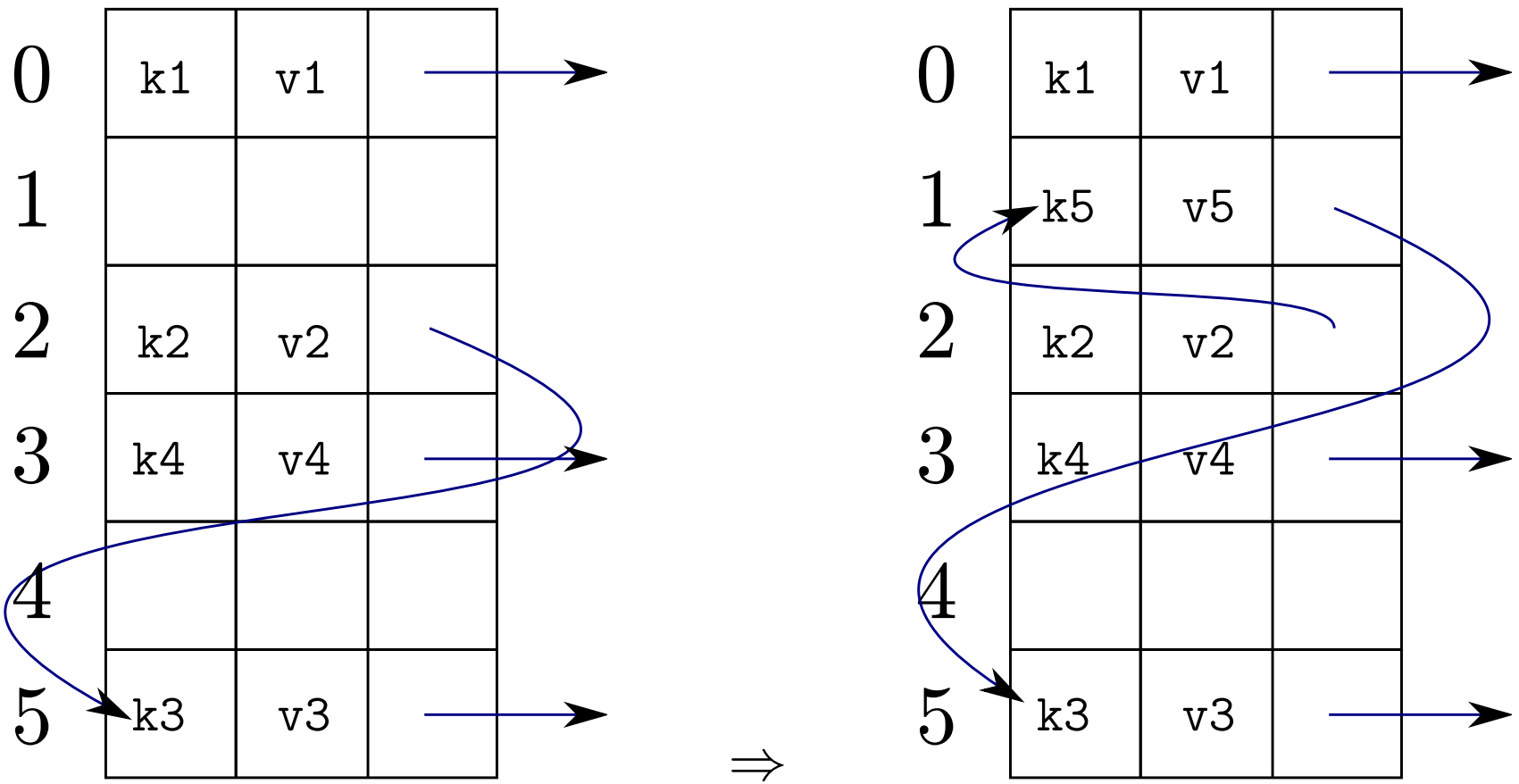
- `hash(k4) = 3`



Insertion : exemple 2

`inserer(d,k5,v5)`

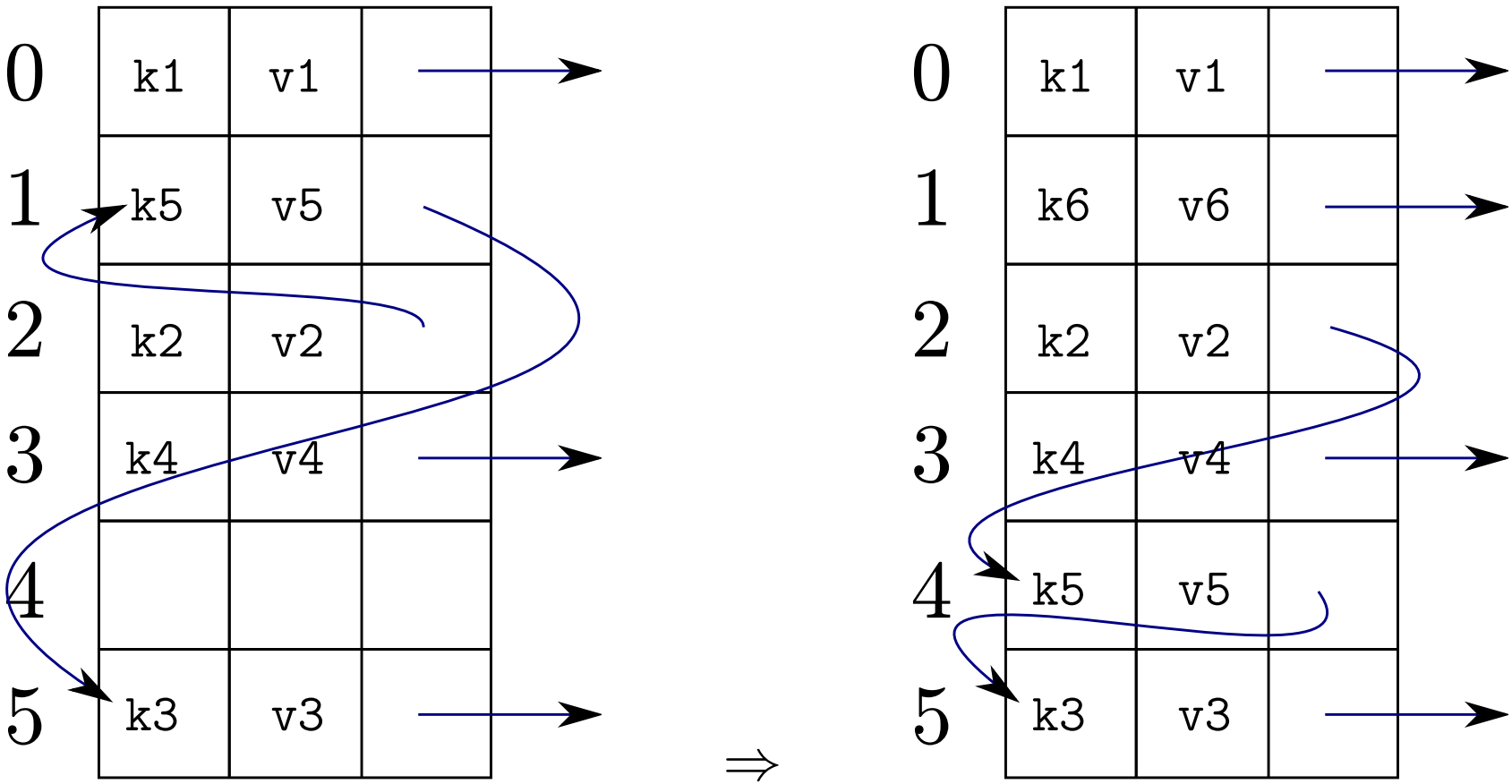
- `hash(k5) = 2`



Insertion : exemple 3

`inserer(d,k6,v6)`

- `hash(k6) = 1`



Insertion

`inserer(d,k,v)`

- on calcule `hash(k)`
- si la case `hash(k)` est libre, on y met `k,v`
- si la case `hash(k)` est occupée par une clef `k'` avec `hash(k')=hash(k)`
  - on trouve une case vide à la position j
  - on rempli la case j avec `k,v`
  - on met comme successeur en j le successeur en `hash(k)`
  - on met j comme successeur en `h(k)`

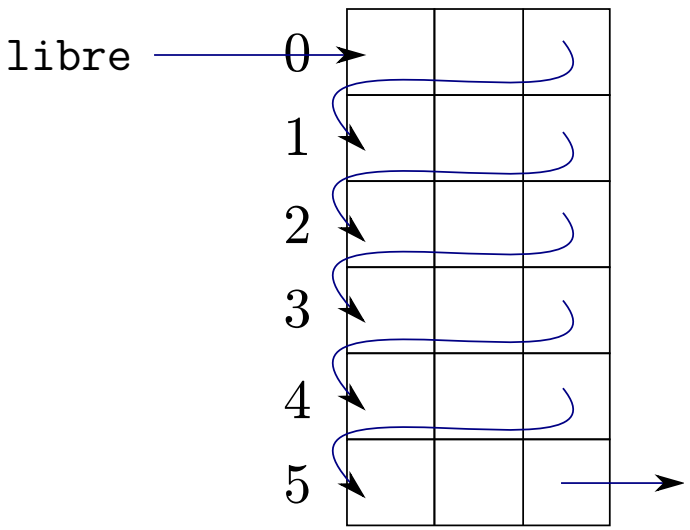
Insertion (suite)

- sinon on a `k',v',s'` en `h(k)` avec `hash(k')` $\neq$ `hash(k)`
  - on met `k,v` en `hash(k)`
  - on suit les successeurs à partir de `hash(k')` jusqu'au prédécesseur j de `hash(k)`
  - on met `k',v',s'` dans une case libre
  - on met à jour le successeur de j sur cette case

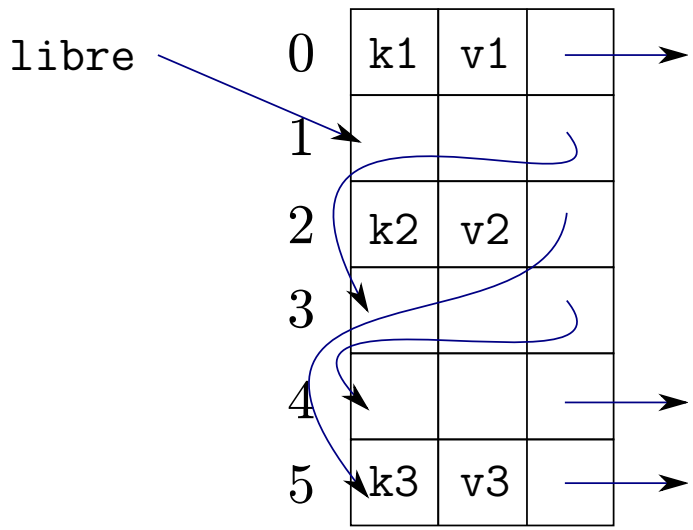
Gestion des cases libres

On a un pointeur vers la première case libre, et on utilise le successeur pour avoir les suivantes

Initialement



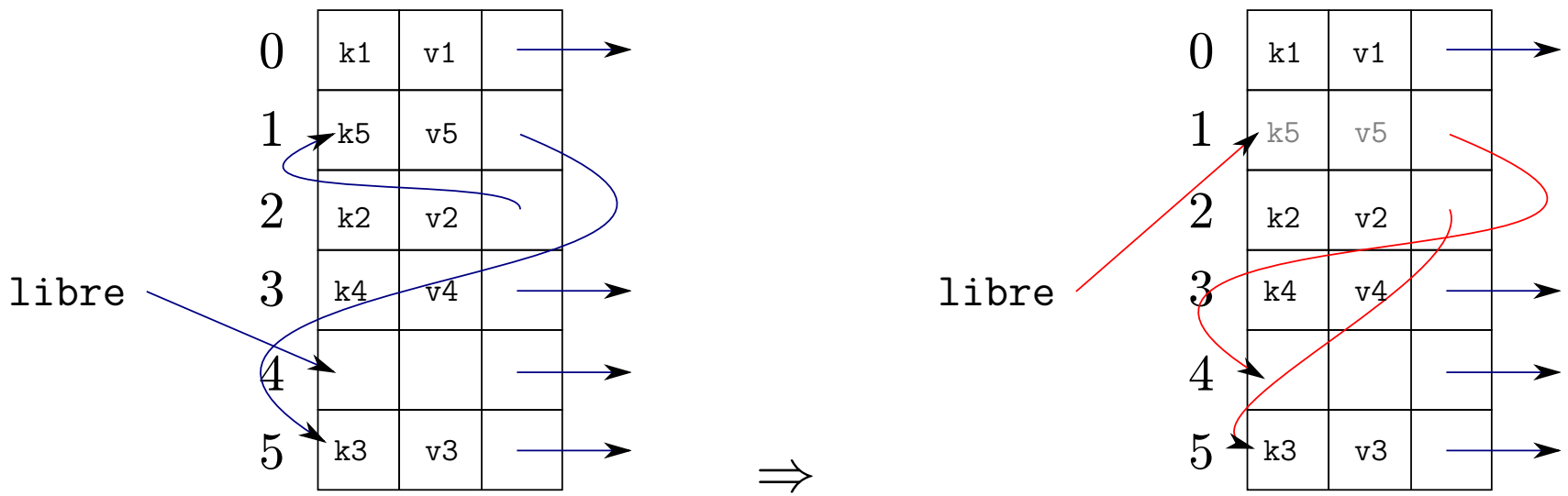
Après insertions/suppressions



Suppression : exemple

`supprimer(d,k5)`

- `hash(k5) = 2`



Suppression

`supprimer(d,k)`

- on trouve `k` en i
- on met à jour le prédécesseur éventuel en j :
  - on met comme successeur en j le successeur en i
- on libère la case correspondante :
  - on fait pointer `libre` vers i
  - on met comme successeur en i l'ancienne valeur de `libre`