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)

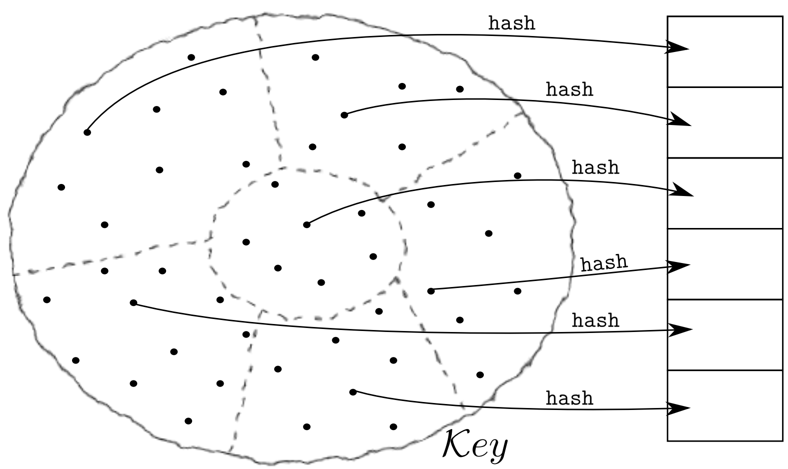
Problème : nombre de clefs possibles potentiellement trop grand
Exemple : Dictionnaire de la langue française
$26^{25} > 10^{35}$ entrées potentielles (en supposant que le mot le plus long de la langue francaise est anticonstitutionnellement)

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

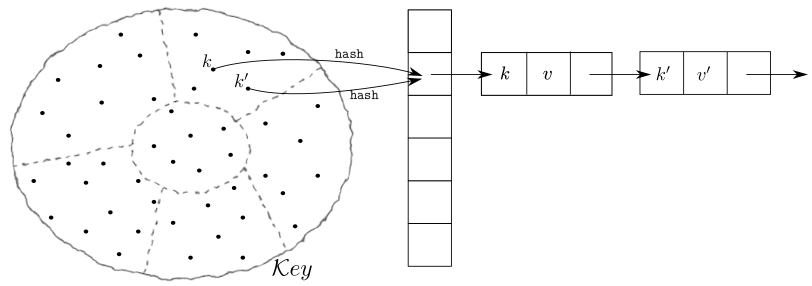


Collision

Comme les clefs 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



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)
```

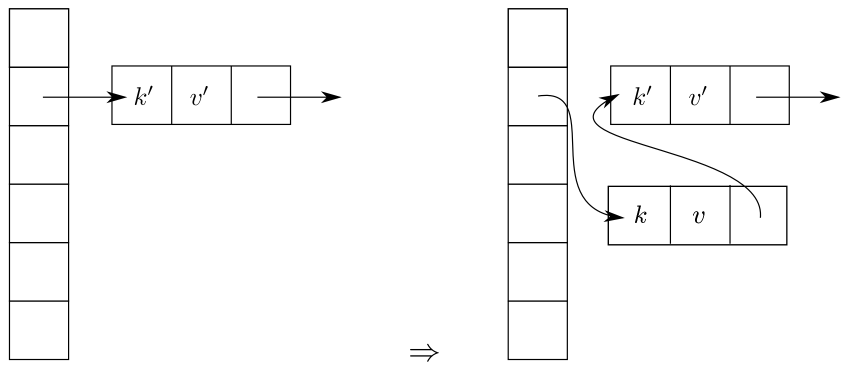
En C :

```
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 && b->next == NULL)
    d->contenu[h] = NULL;
  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	$O(n)$	$O(n)$	$O(n)$