# Programmation avancée

## Dictionnaires

ENSIIE

Semestre 2 — 2011–12

# 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)

- `rechercher(creer(i),k) = ERR` :

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

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

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

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

-

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

  $k \neq k'$ :

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

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

# Correction (OCaml, suite)

- `rechercher(supprimer(d,k),k) = ERR` :
  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)

- `rechercher(supprimer(d,k),k) = ERR` :
  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) = ERR`

# 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