Programmation impérative avancée

ENSIIE

Semestre 2 — 2015–16

Objectifs du cours

- Compilation séparée
- Modularité
- Complexité
- Structures de données avancées
  - arbres bien équilibrés
  - tables de hachage
  - ...

Évaluation

Note du module « Programmation impérative avancée » :

- 1 TP noté (1/3, non rattrapable)
- 1 examen écrit (2/3)

Note de l'UE :

- Programmation impérative avancée (1/2)
- Logique (1/2)

Compilation séparée et Modularité

Modularité

GCC : $>$ 14,5 millions de lignes de code
Noyau Linux : $>$ 19 millions de lignes de code
Windows Vista : 50 millions de lignes de code

- besoin de partage des tâches entre développeurs
- besoin de maintenance
- besoin de réutilisabilité

Fonction

Grain le plus fin de modularité :

- fonction/procédure

Entités indépendantes

Lien avec le reste du code :

- Prototype + contrat (requires/assigns/ensures)

Modules

Grain plus gros
Regroupe

- Structures de données
- Fonctions sur celles-ci

Lien avec le reste du code = interface

- indique les fonctionnalités proposées
- = contrat

En dehors du module, seul ce qui est déclaré dans l'interface peut être utilisé (boîte noire)

Interface

Contient :

- Définitions de types concrets
- Déclarations de types abstraits
- Déclarations de prototypes de fonctions

Les modules extérieurs utilisent ces types et ces fonctions, et uniquement ceux-là, pour pouvoir utiliser le module.
Pour les types abstraits, ils ne peuvent les créer/manipuler qu'à travers les fonctions proposées.

Compilation séparée

Chaque module peut être compilé indépendamment des autres modules

- puisque seule leurs interfaces l'intéresse
- il n'est donc pas nécessaire de tout recompiler à chaque modification
- ni d'attendre qu'un module soit complètement implémenté pour compiler un autre

Une fois chaque module compilé, l'éditeur de liens (*linker*) se charge de « coller » les morceaux les uns avec les autres.

Éventuellement, l'éditeur de lien peut combiner des programmes écrits dans différents langages

Interfaces en C

En C, pas de mécanisme spécifique du langage pour les interfaces, mais pratique standard :

- L'interface est écrite dans un fichier `truc.h`
  - elle contient des prototypes de fonctions
  - des définitions de type (éventuellement abstraite)

```
typedef struct type_concret {
  int x;
  double y;
} type_concret;

typedef struct non_defini* type_abstrait;

int une_fonction(char, type_abstrait);
void une_autre(type_concret*,int);
```

Implémentation

- Le contenu du module est écrit dans un fichier `truc.c`
  - il inclut l'interface (`#include "truc.h"`)
  - il définit le corps des fonctions et les types restés abstraits dans l'interface
  - éventuellement il définit d'autre fonctions (non accessibles en dehors du module)

```
struct non_defini {
  type_concret a;
  char b;  };

int une_fonction(char c, type_abstrait a) { ... }
void une_autre(type_concret* tp, int i) { ... }
type_concret fonction_locale(double l) { ... }
```

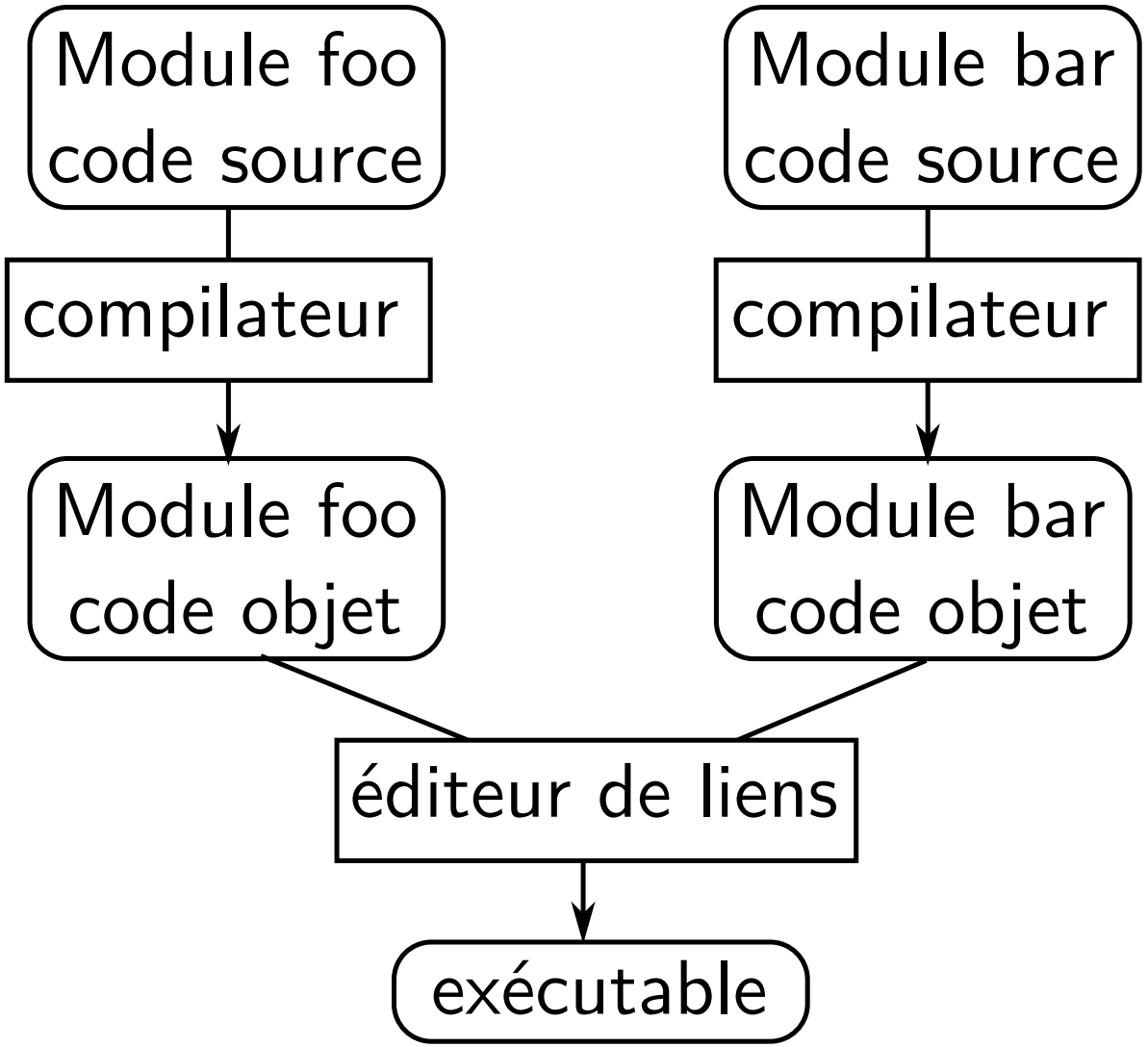
Utilisation

- un autre module qui a besoin de `truc` a uniquement besoin d'inclure l'interface (`#include "truc.h"`)
- il ne peut donc accéder qu'à ce qui est déclaré/défini dans l'interface

```
#include "truc.h"

void machin(type_abstrait x, type_concret* y)
  une_autre(y, une_fonction('z', x));
  /* x->a INTERDIT */
  /* fonction_locale(x) INTERDIT */
}
```

Chaîne de compilation



Chaîne de compilation : C

- source `foo.c` → code objet `foo.o` (code natif)

  `gcc -Wall -Wextra -ansi -c foo.c`

- code objets `foo.o bar.o` → exécutable

  `gcc -Wall -Wextra -ansi foo.o bar.o`

  produit un exécutable `a.out`, si on veut un autre nom : `-o mon_executable`

Il doit y avoir une et une seule fonction `main` dans l'ensemble des modules, qui est appelée au lancement de l'exécutable.

L'ordre des modules n'importe pas.

Exemple

Affichage en base 2 et piles

Modularité

Permet :

- séparation des tâches
- création d'un espace de nom
- réutilisabilité (par exemple, bibliothèques)
- encapsulation
- abstraction (en particulier des types)
- maintenabilité

Séparation des tâches

- Chaque module peut être implémenté par un développeur différent
- Le seul point sur lequel ils doivent s'entendre est l'interface des modules
  (et la sémantique des fonctions...)

Création d'un espace de nom

Dans certains langages (pas en C), chaque module crée un nouvel espace de nom :

- On peut donner le même nom de fonction dans des modules différents
  ex. en Python : `array.fromstring`, `numpy.fromstring`, `...`

Réutilisabilité

Chaque module est indépendant, il peut donc être réutilisé dans un autre projet sans avoir à tout reprendre.
Ex. : Structure de donnée pour stocker des chaînes de caractères
Permet de ne pas réinventer la roue

On peut regrouper des modules pour créer des bibliothèques
En particulier, on dispose en général pour chaque langage d'une bibliothèque standard

- pour C, définie dans le standard C ANSI, implémentée par exemple par la GNU C Library

Encapsulation

- Seules les fonctions déclarées dans l'interface permettent d'accéder aux objets
- Le développeur n'a pas à se soucier d'objets mal formés créés à l'extérieur du module
- Permet de maintenir des invariants (via des constructeurs intelligents par exemple)

Abstraction

Les structures de données utilisées pour implémenter tel ou tel objet peuvent rester abstraites

- définition de type abstrait dans l'interface

  En C :
  `typedef struct contenu_type* type_abstrait`
  dans le `.h`
  la structure `struct contenu_type` n'est pas définie dans l'interface, elle n'est donc pas visible de l'extérieur

Pas besoin de connaître l'implémentation concrète depuis l'extérieur

Maintenabilité

Permet de pouvoir changer l'implémentation concrète sans avoir à changer tout le code

Exemple : utilisation de tableaux au lieu de listes pour implémenter des piles

Exemple : changement du tri par bulle (complexité $O(n^2)$) en tri fusion ($O(n \log n)$)

Invisible depuis les autres modules

Extensions

Suivant les langages de programmation, modules plus avancés :

- modules imbriqués
- modules paramétrés par des modules (foncteurs)
  - généricité
- modules de première classe : peuvent être manipulés comme des valeurs du langage

Dépendances

Un module dépend de son implémentation, mais également des interfaces des modules qu'il utilise

- si celles-ci changent, il faut le recompiler

Par contre, il ne dépend pas de l'implémentation des modules qu'il utilise

- même si celle-ci change, il n'est pas nécessaire de le recompiler

Il peut être difficile de savoir quel fichier a besoin d'être recompilé après un changement

- Utilisation d'un `Makefile` pour gérer automatiquement les dépendances

Makefile

Un fichier `Makefile` est une suite de règles

```
cible: dependance1 dependance2 ... dependancer
	commande_pour_creer_cible
```

(attention, tabulation au début de la deuxième ligne)

- `cible` : fichier à créer
- `dependancei` : fichiers dont `cible` dépend
- `commande_pour_creer_cible;` optionnel
  peut utiliser les raccourcis suivants :
  - `$@` nom de la cible
  - `$^` dépendances
  - `$<` première dépendance

Utilisation d'un Makefile

Quand on appelle `make cible` dans le shell, on vérifie récursivement que les dépendances n'ont pas besoin d'être recompilées, puis on appelle `commande_pour_creer_cible` si une des dépendances est plus récente que `cible` (ou si `cible` n'existe pas).

Si `make` est appelé sans argument, utilise la première règle.

- Convention : première règle

  ```
  all: executable1 executable2
  ```

  sans commande de production (pas de création de `all`, donc règle toujours active)

Règle avec filtrage

```
%.o: %.c
	gcc -Wall -Wextra -ansi -c $<
```

Attention, spécifique GNU make, sinon :

```
.SUFFIXES: .c .o
```

```
.c.o:
	gcc -Wall -Wextra -ansi -c $<
```

Cette règle (`.c` en `.o`) est ajoutée par défaut.

Variables

```
CC=gcc -Wall -Wextra -ansi

%.o: %.c
	$(CC) -c $<

OBJ=dep1.o dep2.o

prog: $(OBJ)
	$(CC) -o $@ $^
```

CC définie par défaut à cc

Makefile générique

Au final, entre les règles et les variables définies par défaut :

```
CC=gcc -Wall -Wextra -ansi
OBJ=module1.o module2.o ... modulen.o

all: prog

module1.o: modulei.h modulej.h
module2.o: modulek.h
...

prog: $(OBJ)
	$(CC) -o $@ $^
```

Complexité

Motivation

On cherche à modéliser les ressources dont a besoin un programme :

- temps
- mémoire

- en moyenne
- dans le pire des cas

Permet de comparer différents algorithmes pour une même spécification

Algorithme VS. programme

Algorithme

- Ensemble d'instructions explicites qui résolve un problème
- Abstrait
- Indépendant d'un langage de programmation

Programme

- Réalisation dans un langage de programmation donné d'un algorithme

Pour la complexité, on s'intéresse principalement aux algorithmes

Pour ce cours, on se contentera des algorithmes décrit par des programmes en C

Opération/ressource élémentaire

On mesure complexité algorithme en fonction du nombre de ressources élémentaires nécessaire au fonctionnement de l'algorithme

$$\text{algo } A, \text{ entrée } e, \text{ complexité } c(A, e)$$

ressources élémentaires :

- nombre d'instruction effectuées
- mémoire utilisée
- comparaisons effectuées
- ...

Modélisation !

- programme compilé
- phénomènes liés au matériel (e.g. mémoire cache)
- interaction avec le système d'exploitation, les autres programmes, ...

On a un modèle

- du programme
- de la machine
- de l'exécution

Complexité allocation de mémoire

`malloc` = instruction élémentaire ?

- recherche d'un bloc libre de bonne taille
  - assez grand
  - pas trop pour limiter "trous"
- éviter fragmentation quand libère mémoire
- compromis recherche efficace/utilisation mémoire optimale

`malloc/free` = mini "garbage collector"

Difficile de considérer comme étape élémentaire

- dépend de l'état courant du tas
- non prévisible
- beaucoup plus d'instructions processeur qu'une simple addition

Complexité dans le pire des cas

Pour pouvoir analyser complexité

- en fonction de la taille de l'entrée

$$c_{max}(A, n) = \max_{e \text{ entrée de taille } n} c(A, e)$$

$\max \rightarrow$ pire des cas

- vision pessimiste
- peut être bien meilleur en pratique
- impossible de faire mieux sans en savoir plus sur les entrées

Complexité en moyenne

Si on connaît distribution des entrées, possibilité de calculer complexité en moyenne
$p(e)$: probabilité que l'entrée e soit passée à l'algorithme parmi celles de même taille

$$c_{moy}(A, n) = \sum_{e \text{ entrée de taille } n} p(e) \times c(A, e)$$

Attention !

- distribution pas forcément équiprobable
- dépend du contexte
- pas toujours possible

Algorithme VS. problème

Complexité d'un problème P

- complexité minimale d'un algorithme qui résout P

$$c(P, n) = \inf_{A \text{ algo qui résout } P} c_{max}(A, n)$$

Algorithme optimal :

- $c(A, n) = c(P, n)$

Complexité d'un problème pas toujours connue

Complexité asymptotique

En général on ne s'intéresse à la complexité que pour des entrées de grande taille

- comportement à la limite $+\infty$
- utilisation des notations de Landau

Rappel de notations

f et g fonctions de $\mathbb{N}$ dans $\mathbb{R}^+$

- On note $f = \mathcal{O}(g)$ s'il existe une constante K et un rang N tel que pour tout $n \geq N$ on a $f(n) \leq Kg(n)$.
  C'est une relation de préordre (réflexive, transitive).
- On note $f = \Theta(g)$ si $f = \mathcal{O}(g)$ et $g = \mathcal{O}(f)$.

Exemple : le tri fusion

On cherche à trier une liste :

- on coupe la liste en deux ;
- on trie récursivement ces deux listes ;
- on fusionne les listes obtenues en comparant les éléments en tête.

Complexité du tri rapide

Soit $C(n)$ la complexité du tri rapide, où n est la taille du tableau, en fonction du nombre de comparaisons.
Pour séparer la liste en deux, on n'a pas besoin de comparaisons.
Pour trier chaque sous-liste, on a besoin de $C(\frac{n}{2})$.
Pour fusionner les sous-listes triées, on a besoin au maximum de $n-1$ comparaisons.
On a donc la relation de récurrence

$$C(n) = n - 1 + C\left(\frac{n}{2}\right) + C\left(\frac{n}{2}\right)$$

Résolution de la récurrence

$$C(n) = n - 1 + 2C(n/2)$$

On pose $T(k) = C(2^k) - 1$

$$\begin{aligned} T(k) &= 2^k - 1 + 2C(2^k/2) - 1 \\ &= 2^k + 2(C(2^{k-1}) - 1) \\ &= 2^k + 2T(k-1) \end{aligned}$$

$$T(k) = 2^k + 2\,T(k-1)$$

Suite récurrente linéaire non homogène. La partie homogène vérifie :

$$U(k) = 2\,U(k-1)$$

ce qui se résoud en $U(k) = 2^k$.
On pose $V(k) = T(k)/2^k$. On a

$$\begin{aligned} V(k) &= \frac{2^k + 2\,T(k-1)}{2^k} \\ &= 1 + T(k-1)/2^{k-1} \\ &= 1 + V(k-1) \end{aligned}$$

ce qui se résoud en $V(k) = k$.
On a donc $T(k) = 2^k\,V(k) = k2^k$. Par conséquent $C(n) = T(\log_2(n)) + 1 = \mathcal{O}(n \log n)$.

Remarques

- Pour un tri, on ne peut pas espérer mieux que $\mathcal{O}(n \log n)$
- Ici, on a pu donner une complexité exacte. En pratique, on doit souvent faire des approximations.

Décidabilité

Un problème est dit :

- décidable s'il existe un algorithme qui permet de le résoudre
- indécidable s'il n'existe pas de tel algorithme

Exemple de problème indécidable :

Problème de l'arrêt

Entrées Le code source d'un programme p et une entrée e

Sortie Vrai ssi p termine sur l'entrée e

Classes de complexité

- $\mathcal{O}(\log n)$ logarithmique
- $\mathcal{O}(n)$ linéaire
- $\mathcal{O}(n \log n)$ quasi-linéaire
- $\mathcal{O}(n^2)$ quadratique
- $\mathcal{O}(n^k)$ (k constante) polynomiale
- 2^n exponentielle

Comparaison

$T(n)$	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$	$n = 60$
$\log n$	1μs	1,3μs	1,5μs	1,6μs	1,7μs	1,8μs
n	10μs	20μs	30μs	40μs	50μs	60μs
$n \log n$	10μs	26μs	44μs	64μs	85μs	107μs
n^2	100μs	400μs	900μs	1,6ms	2,5ms	3,6ms
n^5	0,1s	3s	24s	1,7min	5min	13min
2^n	1ms	1s	18min	13j	36a	36 600a

Algorithme déterministe/non déterministe

Algorithme non déterministe :

- Peut faire un choix à certains points de l'algo
- Résout le problème ssi un des ensembles de choix donne une réponse positive

Exemple

3-coloriage

Entrée un graphe

Sortie donner une couleur parmi 3 à chaque sommet
deux voisins ne doivent pas avoir la même couleur

```
pour tout sommet s
  si les voisins de s ont au plus 2 couleurs ≠
    donner une couleur possible à s
  sinon
    échouer
```

Classes de problèmes

Un problème appartient à

- P s'il existe un algorithme (déterministe) polynomial qui permet de le résoudre
- NP s'il existe un algorithme non-déterministe polynomial (équivalent à dire qu'on peut vérifier une solution du problème en temps polynomial)

On ne sait pas si P = NP (1M$!)

Le problème est dit NP-complet si la résolution de n'importe quel problème NP peut se ramener à la résolution de ce problème particulier.
3-coloriage est NP-complet

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.

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

Spécification d'un dictionnaire – Interface

```
typedef int clef;
typedef char* valeur;

typedef struct dict_base* dict;

dict creer(int);
valeur rechercher(dict, clef);
void inserer(dict*, clef, valeur);
valeur supprimer(dict*, clef);
```

Spécification d'un dictionnaire – Propriétés

- `rechercher` retourne la dernière valeur associée à la clef via `inserer` ;
- `rechercher` retourne `NULL` sinon ;
- `inserer` ajoute une nouvelle association entre la clef et la valeur ; elle masque d'autres associations éventuellement présentes ;
- `supprimer` retire la dernière association ajoutée avec cette clef ;
- `supprimer` ne fait rien et retourne `NULL` si rien n'est associé à la clef ;
- `supprimer` retourne la valeur de l'association qui est supprimée sinon.

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 premier 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 C

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

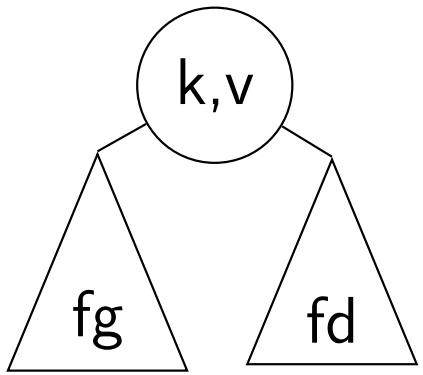
mais

- 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.

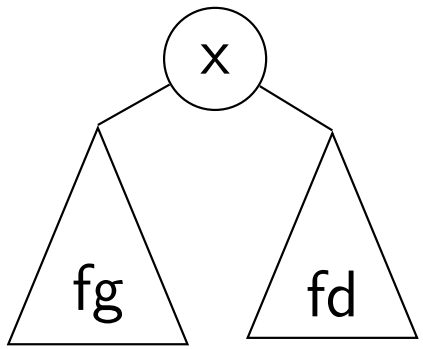


Invariant : pour un nœud

- 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

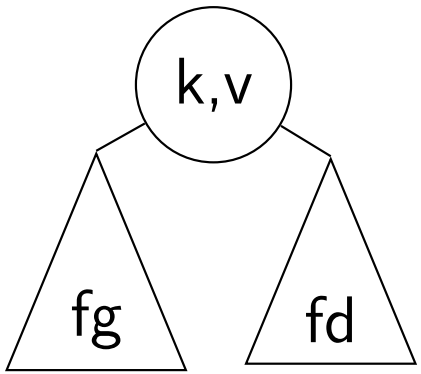
Pour montrer que P est vrai pour tout arbre binaire, il suffit de montrer que

- 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 

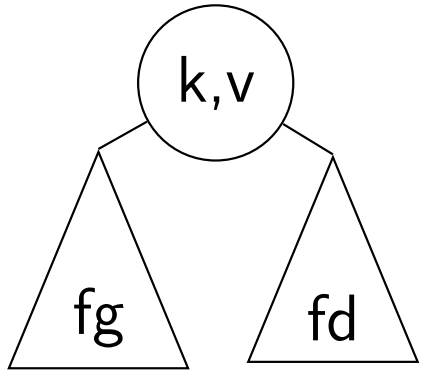
- si k' $=$ k, retourner v
- si k' $<$ k et fg est non-vide, rechercher k' dans fg
- si k' $>$ k et fd est non-vide, rechercher k' dans fd

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
- si k' $=$ k, mettre v' à la place de v et insérer k',v dans fg

Complexité : $O(h)$

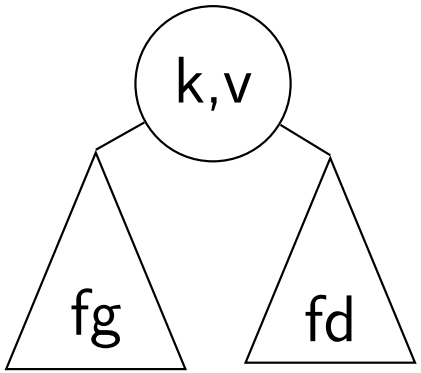
- à chaque appel récursif, on diminue la hauteur de l'arbre d'au moins 1
- on peut avoir à descendre jusqu'au nœud le plus bas

Suppression

Supprimer le noeud avec la clef la plus grande :

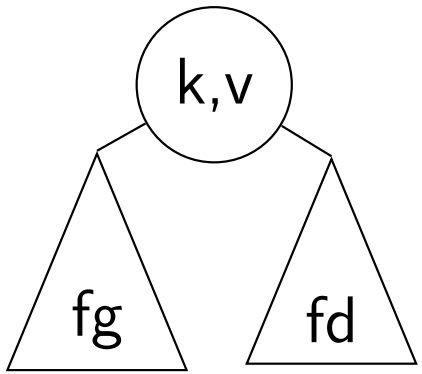
- Si pas de fils droit, on retourne le sous-arbre gauche
- Sinon on va à droite tant qu'il existe un fils droit
- On relie le père du noeud à supprimer à son fils gauche

Suppression

Supprimer la racine de 

- si fg vide, on retourne fd
- si fd vide, on retourne fg
- sinon,
  - on supprimer le noeud contenant (k',v') avec la plus grande clef dans fg
  - on le met à la place de la racine

Suppression

Supprimer k' dans 

avec k $\neq$ k' :

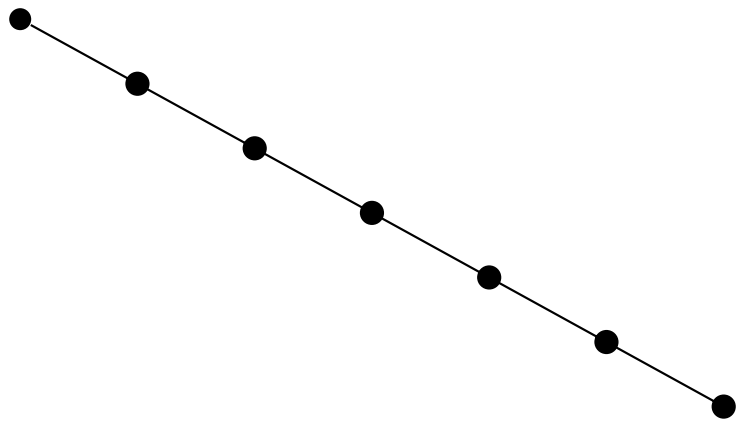
- si k' $<$ k, supprimer k' dans fg
- si k' $>$ k, supprimer k' dans fd
- si k $=$ k', supprimer la racine

Complexité : $O(h)$

- supprimer la clef la plus grande : $O(h)$
- donc supprimer la racine : $O(h)$
- 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 (C)

Appels (récursifs)

Lors d'appel de fonctions, utilisation de la pile pour stocker l'environnement de la fonction :

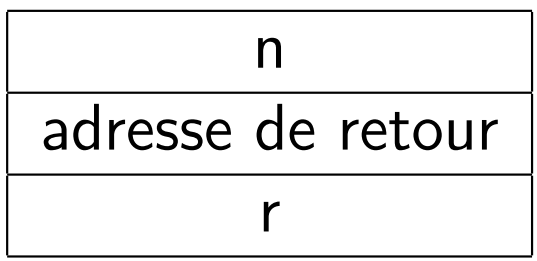
- paramètres
- variables locales
- adresse de retour

Permet d'imbriquer les appels et de revenir à l'environnement initial à la fin d'un sous-appel

Exemple

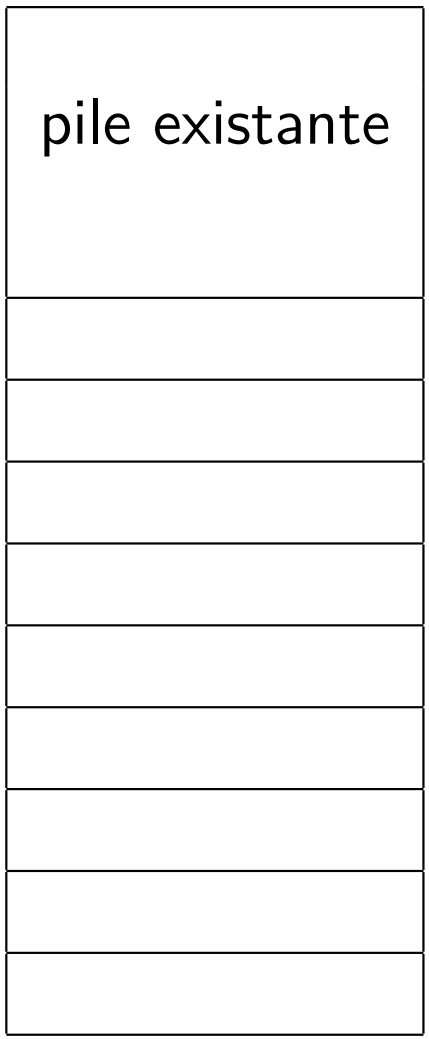
```
int fib(int n) {
  int r = 0;
  if (n <= 1)
    r = 1;
  else {
    r = fib(n-1);
    r = r + fib(n-2);
  };
  return r;
}
```

Trame de `fib` :



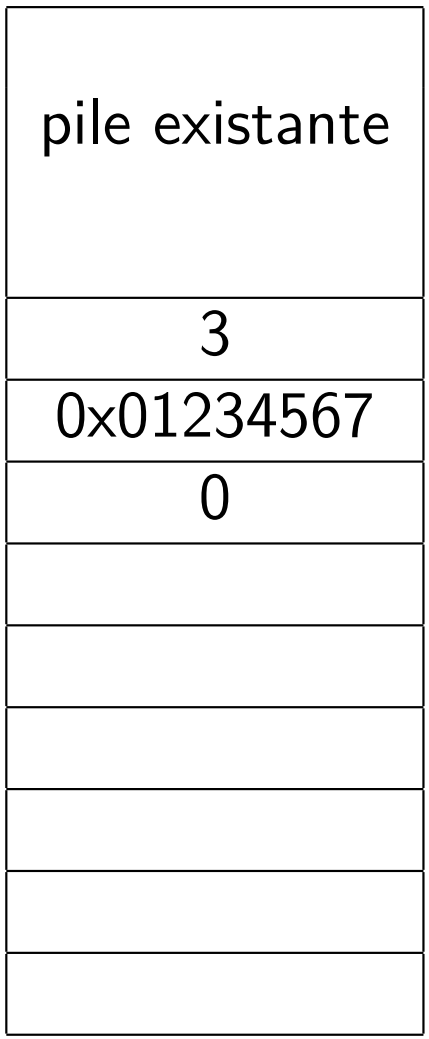
Exemple

Appel à `fib(3)`



Exemple

Appel à `fib(3)`



Exemple

Appel à `fib(3)`

pile existante
3
0x01234567
0
2
0x89012345
0

Exemple

Appel à `fib(3)`

pile existante
3
0x01234567
0
2
0x89012345
0
1
0x89012345
0

Exemple

Appel à `fib(3)`

pile existante
3
0x01234567
0
2
0x89012345
0
1

Exemple

Appel à `fib(3)`

pile existante
3
0x01234567
0
2
0x89012345
1
0
0x89012347
0

Exemple

Appel à `fib(3)`

pile existante
3
0x01234567
0
2
0x89012345
1
1

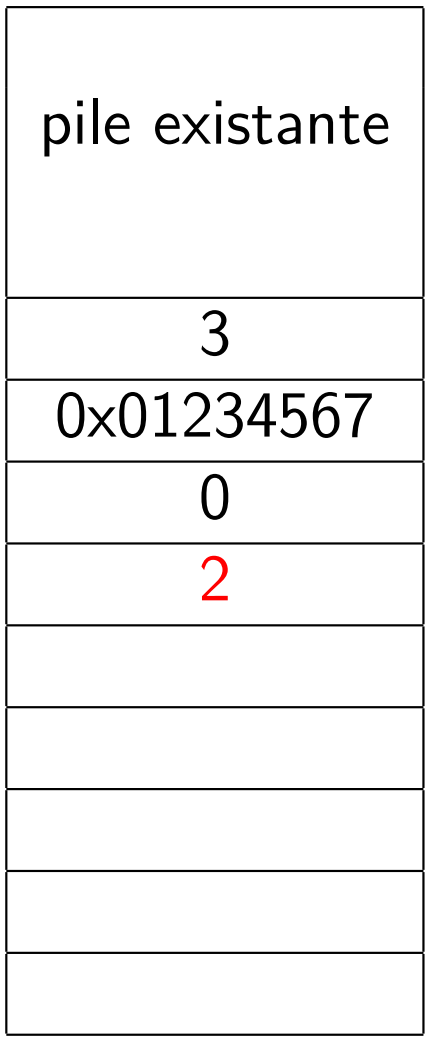
Exemple

Appel à `fib(3)`

pile existante
3
0x01234567
0
2
0x89012345
2

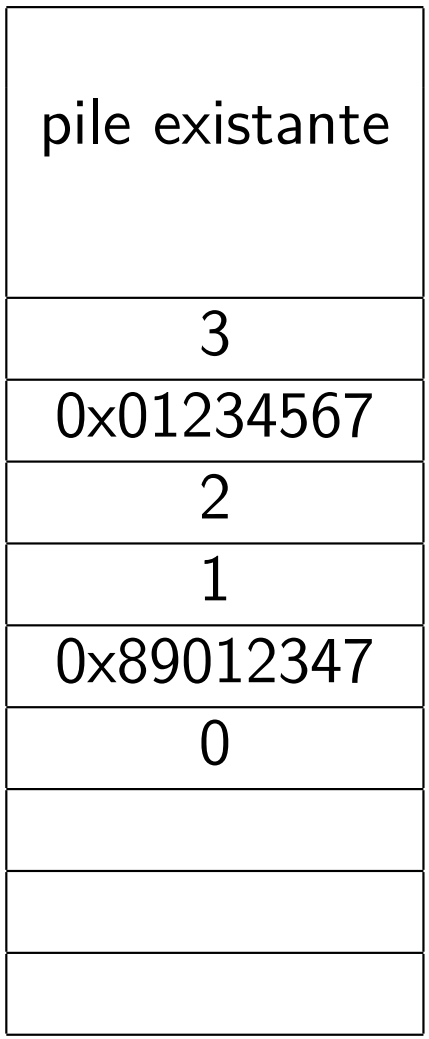
Exemple

Appel à `fib(3)`



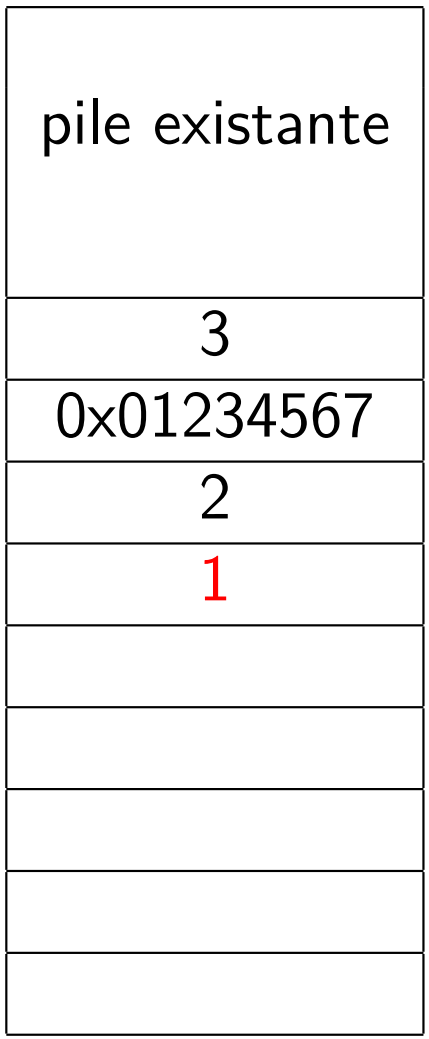
Exemple

Appel à `fib(3)`



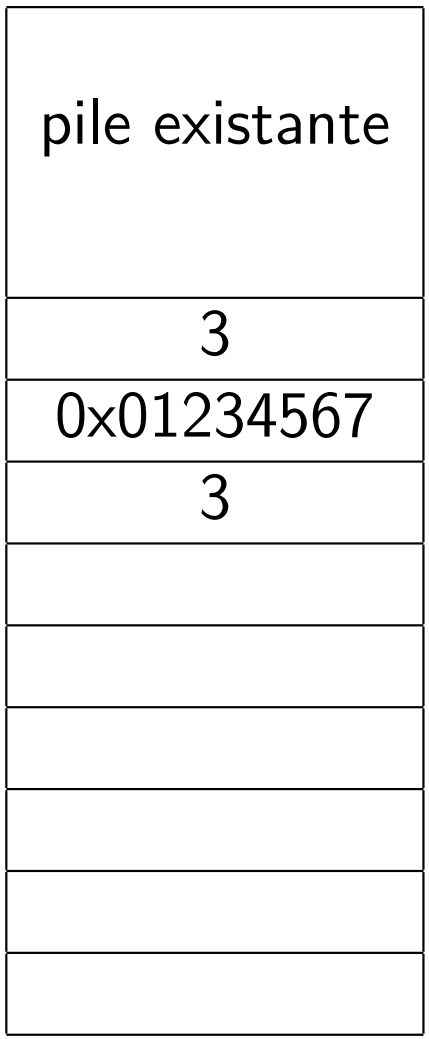
Exemple

Appel à `fib(3)`



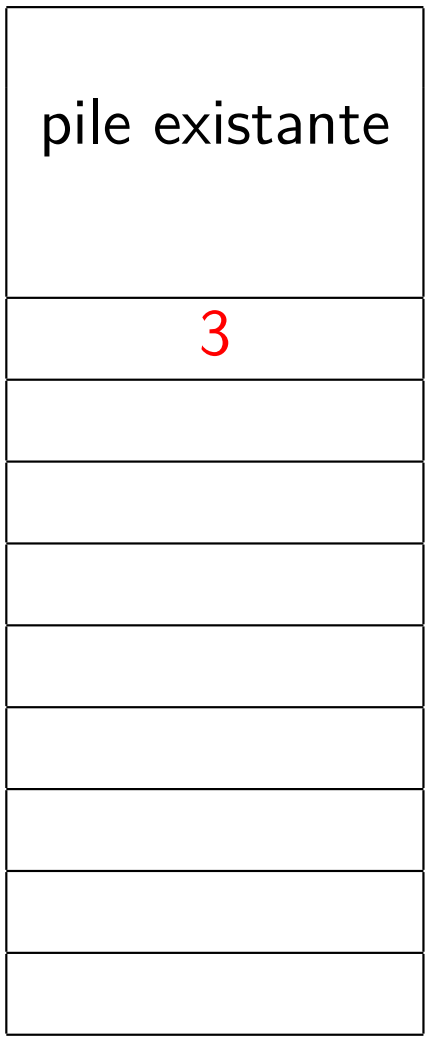
Exemple

Appel à `fib(3)`



Exemple

Appel à `fib(3)`



Dépassement de pile

Si trop d'appel (en particulier récursifs)

- la pile grandit trop et dépasse la place réservée
- le programme s'arrête avec une erreur (e.g. *Stack overflow* ou *Segmentation fault*)

Optimisation des appels récursifs terminaux

Appel terminal : dernière instruction effectuée par la fonction

```
ret fonction(t1 arg1, t2 arg2) {
  if (cond) {
    [...1...];
    return fonction(exp1, exp2);
  };
  [...2...];
  return r;
}
```

```
ret fonction(t1 arg1, t2 arg2) {
  while (cond) {
    [...1...];
    arg1 = exp1;
    arg2 = exp2;
  };
  [...2...];
  return r;
}
```

Optimisation des appels terminaux

Certains compilateurs optimisent automatiquement les appels terminaux

- Par exemple `ocamlc`
- Cf. cours de compilation en S3

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

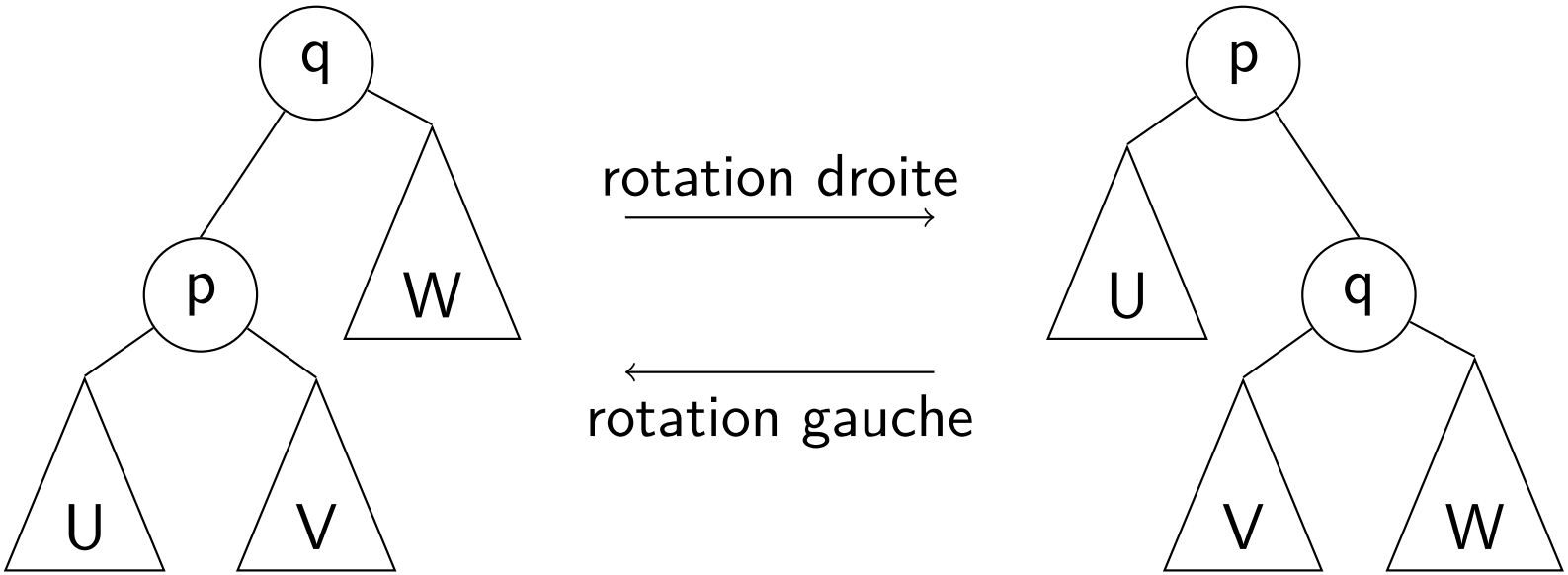
Arbres auto-équilibrants

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

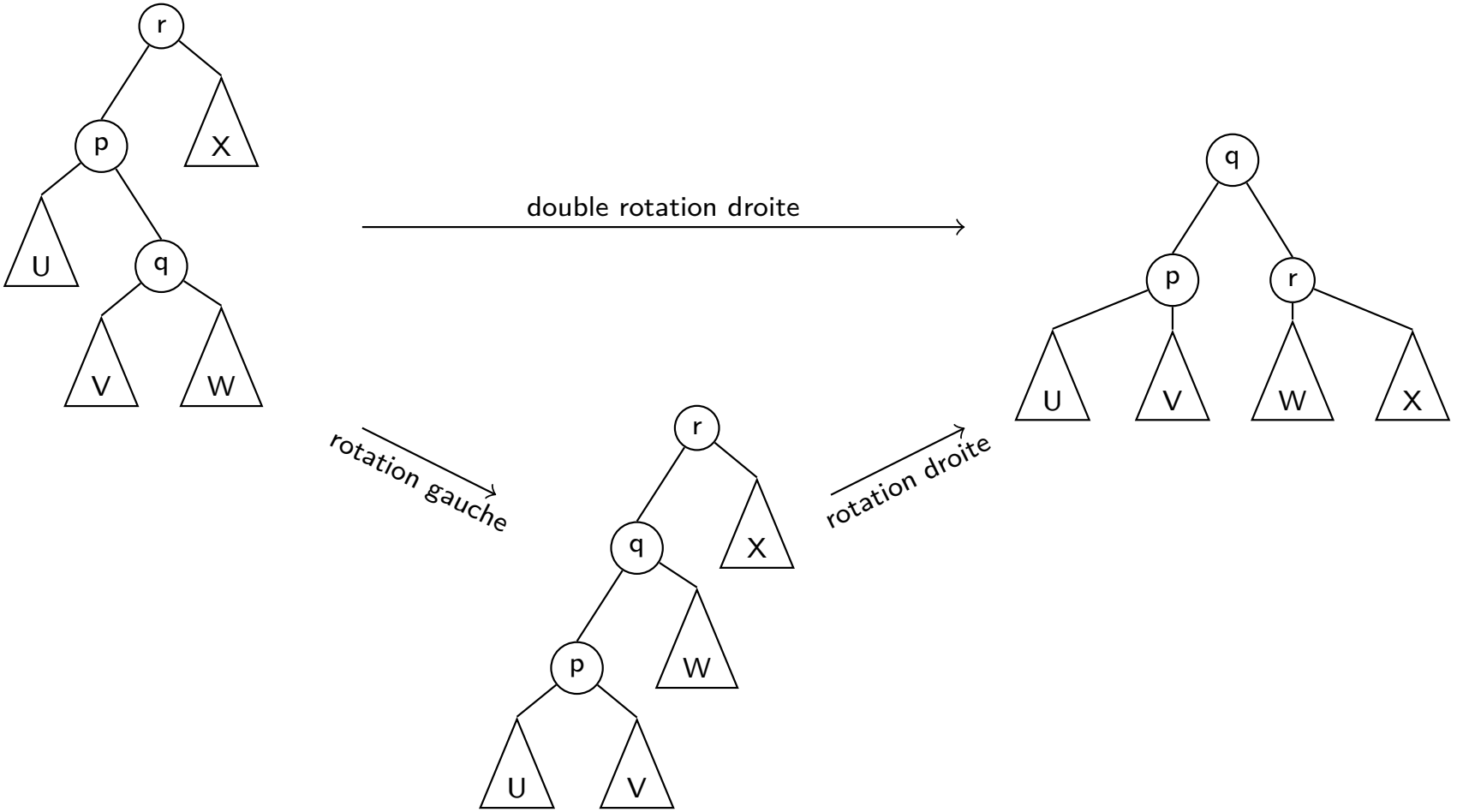
Rotations



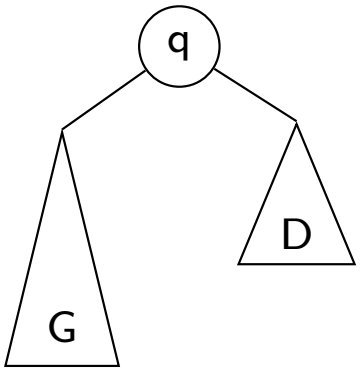
Les rotations préservent l'invariant des ABR

Elles peuvent être réalisées en temps constant

Double rotations

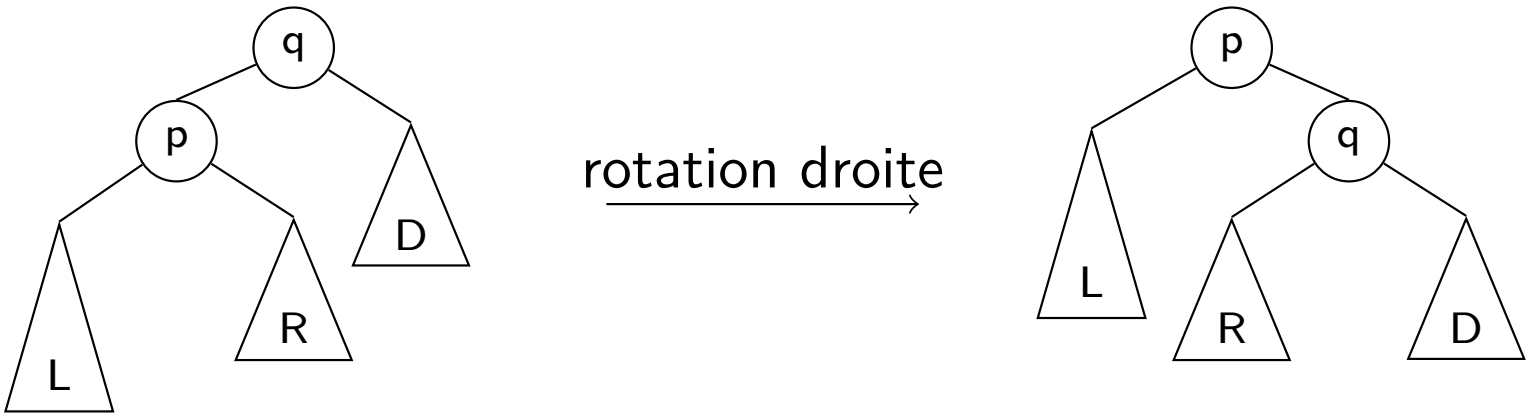


Rééquilibrage



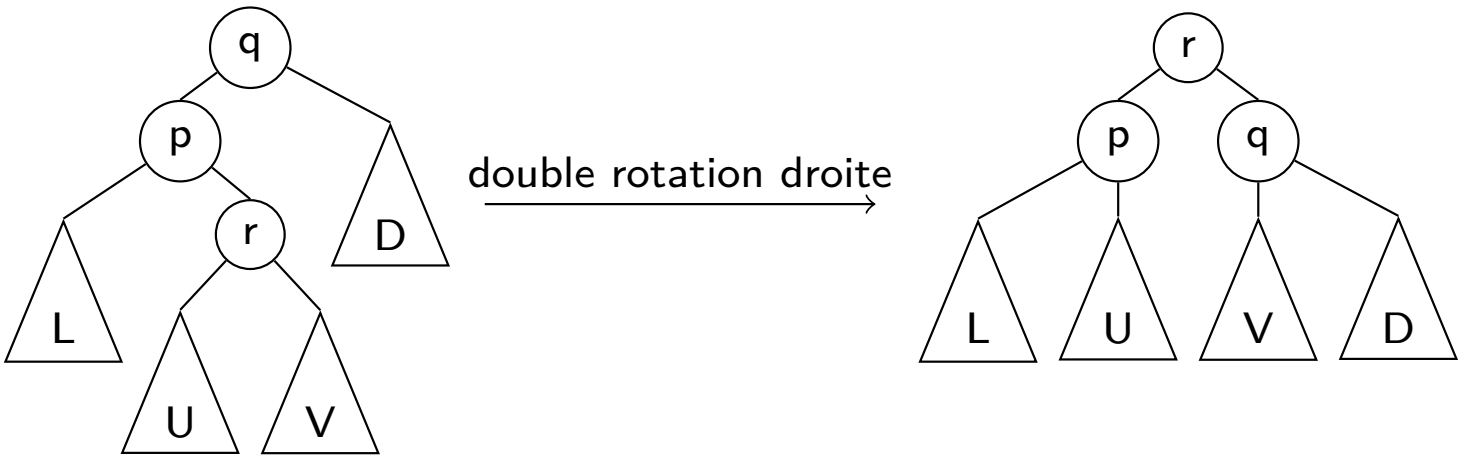
Une insertion ou une suppression perturbe d'au plus 1 les hauteurs des sous-arbres
Par conséquent la différence de hauteur entre deux sous-arbres est d'au plus 2 si l'arbre était bien équilibré avant
On suppose $h(G) = h(D) + 2$ (l'autre cas par symétrie)

Rééquilibrage (suite)



- Si $h(L) \geq h(R)$: rotation droite

Rééquilibrage (suite)



- Si $h(L) \geq h(R)$: rotation droite
- Si $h(L) < h(R)$: double rotation droite

Hauteur

Pour savoir quelles rotations effectuer, il faut connaître la hauteur

- stockée dans les nœuds

À chaque modification des sous-arbres, mais uniquement à ces moments-là, il faut mettre à jour la hauteur
Calcul de hauteur en temps constant

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)

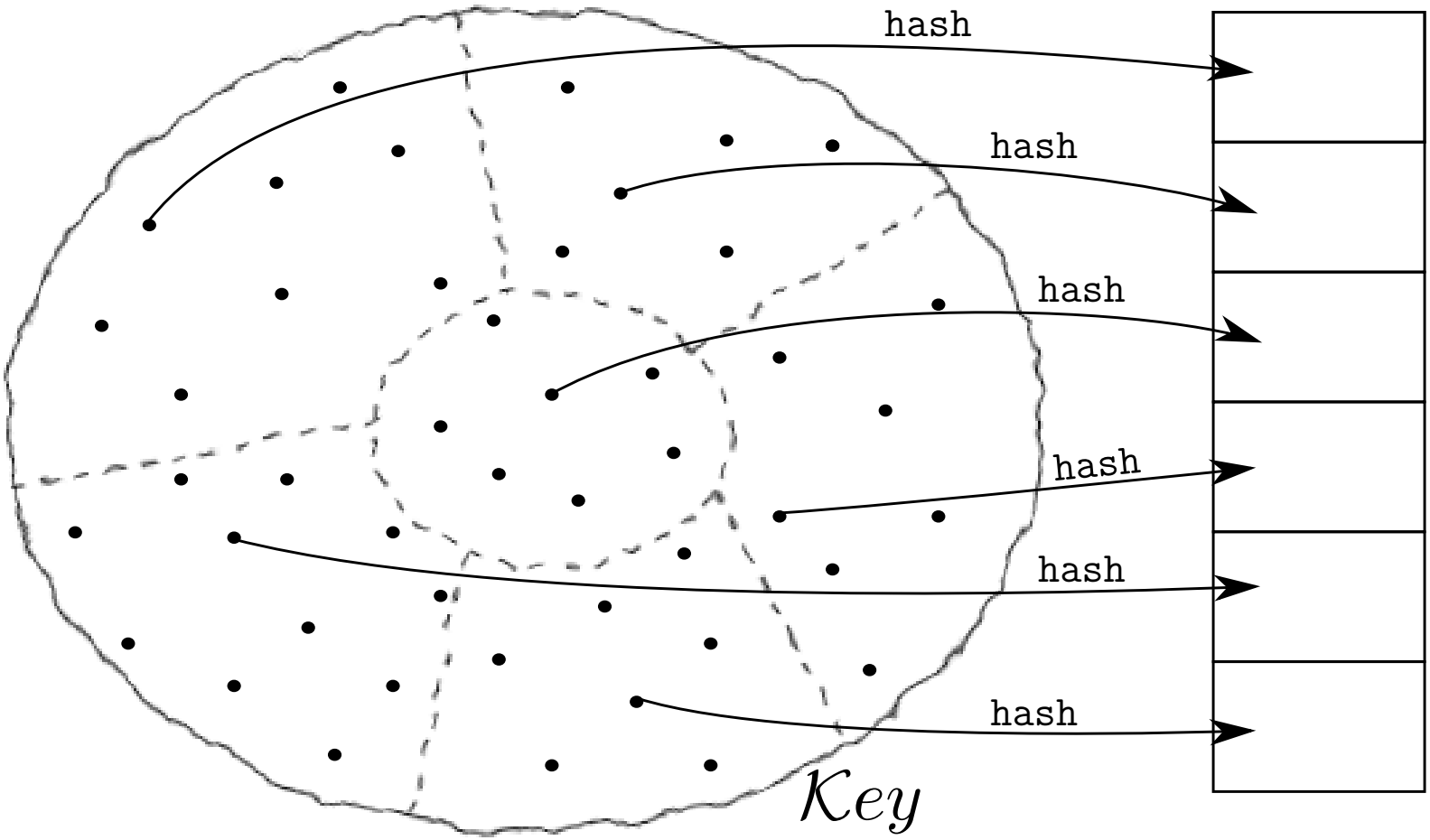
Hachage

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



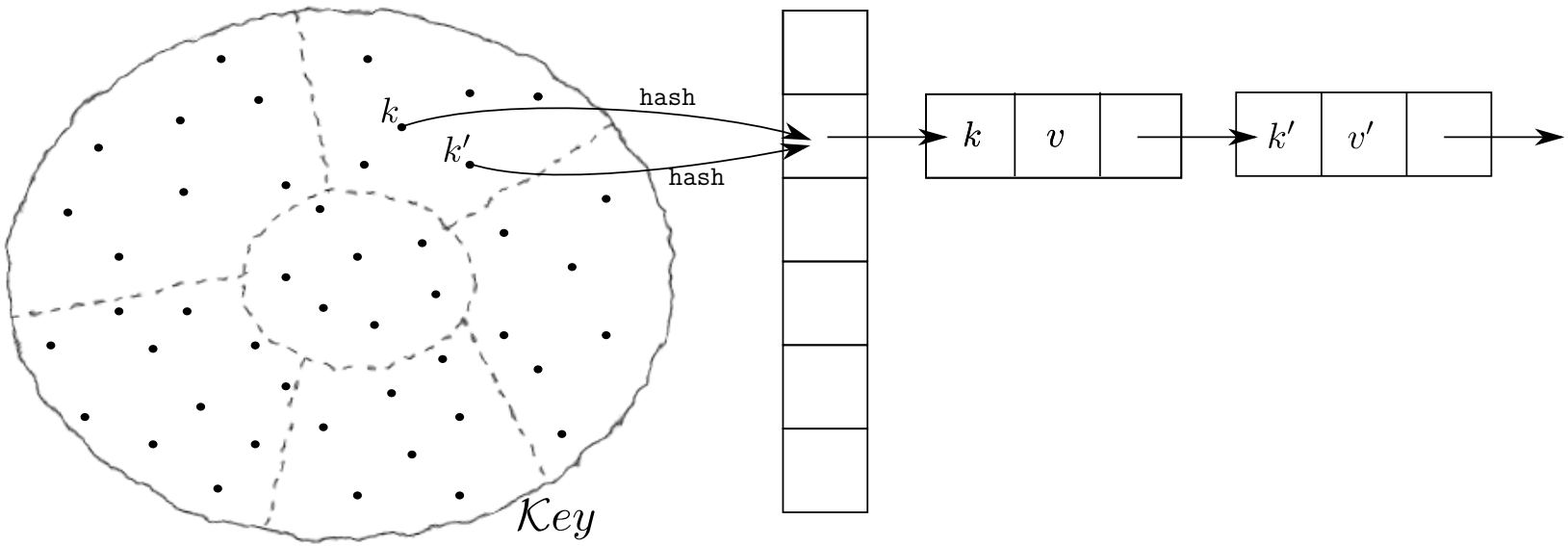
Collision

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

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

typedef struct bucket* list;

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

Création

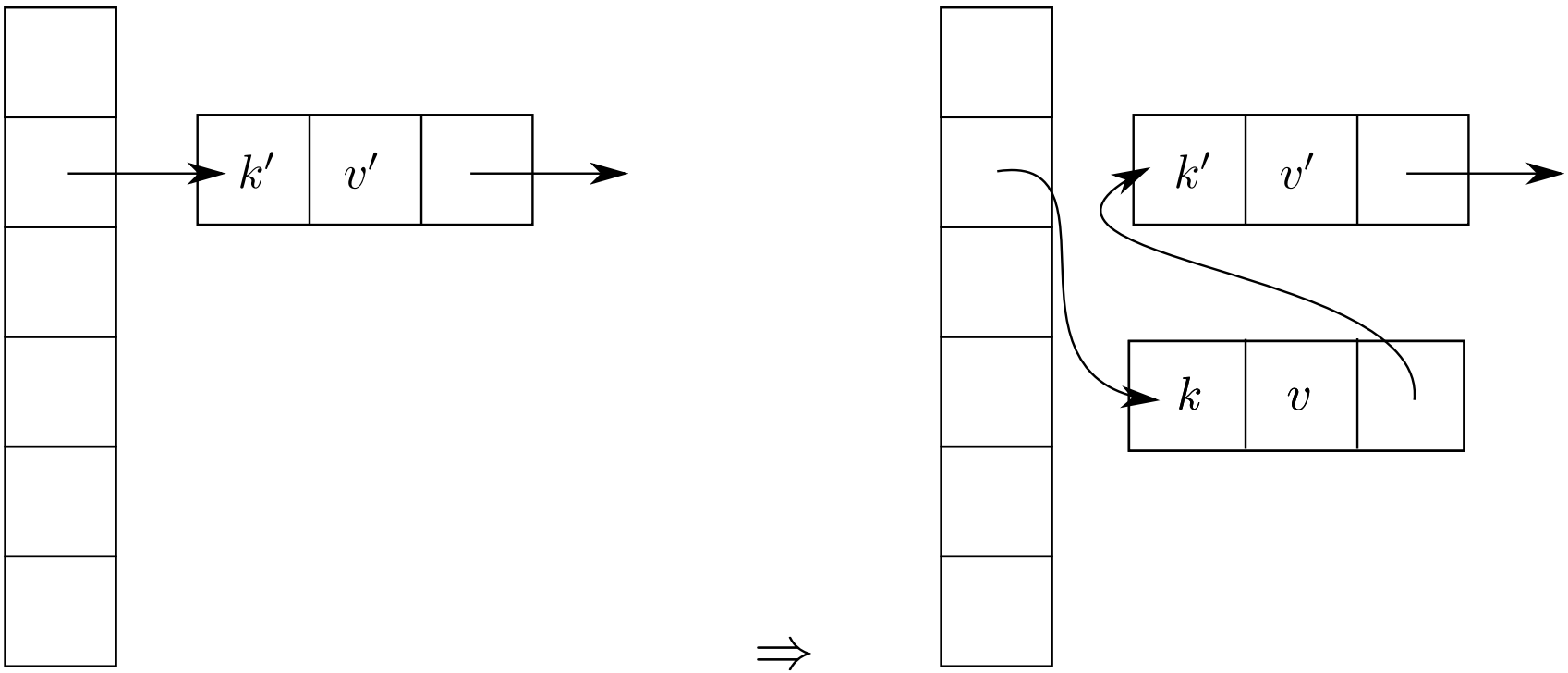
`creer(i)`

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

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



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

Suppression

`supprimer(d,k)`

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

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	$O(n)$	$O(n)$	$O(n)$