# Chapitre 4 : Structures de données

## Table des matières

# 1 Structures, types et modularité

## 1.1 Programmation modulaire

### 1.1.1 Principe

Le principe de la modularité est de décomposer un programme en éléments aussi indépendants et réutilisables que possible.

Intérêts de la modularité :

- Faciliter l'implémentation en se focalisant sur des tâches précises et simples ;
- Faciliter la lecture du code ;
- Éviter de coder plusieurs fois la même chose ;
- Faciliter le débogage et les tests ;
- Permettre des optimisations locales ;
- Permettre le travail d'équipe ;
- etc.

### 1.1.2 Outils

Nous avons déjà évoqué deux types d'outils pour la programmation modulaire :

- Les fonctions : ce sont des sous-programmes que l'on peut réutiliser pour exécuter le même algorithme sur des données différentes. Il est donc important de décomposer ses programmes en fonctions élémentaires ;
- Les bibliothèques, ou modules : ce sont des ensembles de définitions de constantes et de fonctions, regroupées pour leur appartenance à un même thème afin de fournir les fonctionnalités nécessaires à la programmation dans le cadre de ce thème. Rappel : l'usage d'une bibliothèque en C nécessite l'inclusion de son/ses fichier(s) d'entête par une directive `#include <fichier.h>` ou `#include "fichier.h"`, la première option étant réservée à la bibliothèque standard. En OCaml, on peut utiliser directement un objet en préfixant son nom par celui du module qui le contient (ex : `Module.fonction`, comme `List.hd`), ou bien on peut inclure le contenu d'un module avec une expression de la forme `open Module`, et utiliser les éléments déclarés directement via leur nom. L'usage de bibliothèques est important car il permet de séparer l'implémentation de ses fonctionnalités de leur usage. En particulier, un utilisateur n'a pas besoin de connaître les détails d'implémentations d'un module, et deux implémentations fournissant les mêmes fonctionnalités sont interchangeables.

### 1.1.3 Interfaces

L'interchangeabilité des implémentations de mêmes fonctionnalités est rendue possible grâce aux fichiers d'interface (d'extension .h en C et .mli en OCaml) qui contiennent une description des noms déclarés dans un module. Le mécanisme de la compilation séparée permet alors d'écrire un programme sans avoir accès à une implémentation d'un module et, au moment de la production d'un exécutable, de choisir n'importe quelle implémentation qui fournit les objets requis.

Le principe des interfaces et de l'interchangeabilité des implémentations n'est pas limité aux modules, mais s'applique également au stockage et à la manipulation de données.

## 1.2 Structures de données

### 1.2.1 Définition

Une structure de données abstraite est la donnée d'une interface représentant un ensemble de données et l'ensemble des opérations permettant de manipuler ces données.

### 1.2.2 Exemple

Une structure d'ensemble dont les éléments sont pris dans un ensemble $E$ est la donnée d'un type représentant des collections d'éléments de $E$ et d'opérations sur ce type permettant la construction d'ensembles élémentaires (ensemble vide, singletons, $E$), l'union, l'intersection, la différence de deux ensembles et les tests d'inclusion et d'égalité.

Ce que nous avons fait dans le $\text{TP}_{02}$ consistait à fournir une implémentation concrète de cette structure.

### 1.2.3 Vocabulaire

Étant donné une structure de données abstraite, on appelle :

- *Constructeur* une opération permettant d'initialiser un élément de cette structure (ex : une opération singleton, qui prend un élément $e \in E$ et qui construit l'ensemble $\{e\}$) ;
- *Accesseur* (ou *getter*) une opération permettant d'accéder à des données stockées dans la structure (ex : une opération représentant, qui prend une structure de classe d'équivalence et qui renvoie un élément canonique de cette classe, par exemple les classes de congruence modulo $n$ avec pour représentants les entiers de $[\![0\ ;\ n-1]\!]$) ;
- *Transformateur* (ou *setter*) une opération permettant de modifier l'état de la structure (ex : une opération d'ajout d'un élément à un ensemble, si cela est autorisé par la structure de données, cf. ci-après).

On oppose structures de données *statiques* et *dynamiques*. Une structure de données dynamique est une structure dont on peut modifier la quantité de données stockées (par ajout ou suppression, par exemple le type list en Python). Une structure est statique si elle n'est pas dynamique.

On oppose également structures de données *mutables* et *immuables*. Une structure est dite mutable si l'on peut changer la valeur de l'un de ses éléments. Dans le cas contraire, elle est immuable (on dit aussi fonctionnelle). Par exemple, en OCaml, les listes sont l'implémentation d'une structure immuable tandis que les tableaux implémentent une structure mutable.

### 1.2.4 Concrétisations

Une structure de données abstraite est une manière d'exprimer un besoin pour résoudre plusieurs problèmes similaires. Une implémentation / concrétisation de cette structure est une manière de répondre à ce besoin. Étant donné un problème, le choix d'une structure de données pour le résoudre est important, car il conditionne la manière dont la résolution s'effectuera (le problème du parcours d'un graphe nous fournira plus tard un exemple).

Il peut également être nécessaire de sélectionner une implémentation de cette structure selon les opérations qui seront le plus utilisées dans la résolution du problème. En effet, une structure de données abstraite peut avoir plusieurs implémentations / concrétisations ne fournissant pas les fonctionalités avec la même efficacité. Certains informaticiens considèrent que la définition d'une structure de données abstraite doit contenir des contraintes de complexité pour les opérations. Cependant, des contraintes de complexité trop précises peuvent limiter trop drastiquement les possibilités dans le choix de l'implémentation.

En pratique, une implémentation d'une structure de données est la donnée d'un type concret et de fonctions manipulant ce type et implémentant les opérations requises.

## 1.3 Déclaration de types

### 1.3.1 Structures et types en C

On peut déclarer un nom de type comme synonyme d'un type existant (on parle d'alias).

On utilise pour cela la syntaxe :

```
typedef <type> <nom>;
```

Par exemple :

```
typedef int entier_relatif;
entier_relatif n = -1;
```

Les déclarations de types synonymes sont particulièrement utiles en combinaison avec les déclarations de structures. Dans une structure, on regroupe des données dans des champs nommés. La structure peut porter un nom.

Syntaxe :

```
struct [nom_structure] {
    <type_champ_1> <nom_champ_1>;
    ...
    <type_champ_n> <nom_champ_n>;
};
```

Par exemple :

```
struct couple {
    int x;
    int y;
};
```

Il ne faut pas oublier le mot-clé `struct` lors de la déclaration d'un élément d'une structure.

Par exemple :

```
struct couple c;
```

Pour initialiser un tel élément, cela fonctionne comme pour les tableaux : on peut l'initialiser directement au moment de la déclaration en fournissant les valeurs des champs dans l'ordre entre accolades, ou bien on initialise les champs un à un par affectation, sachant que l'accès à un champ se fait via la syntaxe :

`<nom_variable_structure>.<nom_champ>`.

Par exemple :

```
struct couple c = {0, 1};
//ou bien
struct couple c;
c.x = 0;
c.y = 1;
```

On peut utiliser une déclaration de type éviter d'avoir à répéter le mot-clé `struct` :

```
typedef struct couple couple_entier;
```

En une seule fois, sans nommer la structure intermédiaire :

```
typedef struct {
    int x;
    int y;
} couple_entier;
```

Remarque : lorsque l'on a un pointeur vers une structure, l'accès aux champs peut s'écrire `nom_ptr->nom_champ` plutôt que `(*nom_ptr).nom_champ`.

### 1.3.2 Types en OCaml

Pour déclarer un type en OCaml, on peut utiliser la syntaxe suivante :

```
type <nom_type> = <type_concret>
```

Par exemple, pour un type synonyme :

```
type entier_relatif = int
```

On peut définir l'équivalent d'une structure en C grâce aux types produits/enregistrements.

Syntaxe :

```
type <nom_type> = {
    <nom_champ_1> : <type_champ_1>;
    ...
    <nom_champ_n> : <type_champ_n>;
}
```

Par exemple :

```
type couple_entier = {x : int; y : int}
```

On déclare alors un élément de ce type en fournissant la valeur de chacun des champs. Par exemple :

```
let c = {x = 0; y = 1}
```

Attention, les champs sont *immuables*, sauf si l'on ajoute le mot-clé `mutable` aux champs que l'on souhaite pouvoir modifier. Une affectation sur un champ s'écrit avec une flèche, à la manière des tableaux. L'accès aux champs s'écrit comme en C. Par exemple :

```
type couple_mutable = {mutable x : int; mutable y : int};;
let c = {x = 0; y = 1};;
c.x <- 1;;
```

Remarques :

– On a la compatibilité avec le polymorphisme :

```
type 'a couple = {x : 'a; y : 'a};;
let c : int couple = {x = 0; y = 1};;
```

– On peut déconstruire un élément d'un type enregistrement par filtrage, y compris dans un `let` :

```
let {x = abscisse; y = _} = c in print_int abscisse
(* voire *)
let {x; y} = c in print_int x
(* voire meme, puisqu'on n'utilise que x *)
let {x} = c in print_int x
```

En OCaml, on dispose d'un format de type supplémentaire : le type somme.

Syntaxe :

```
type <nom_type> =
    | <Nom_constructeur> (* constructeur sans argument *)
    | <Nom_constructeur> of <type_argument > (* constructeur avec
        arguments *)
    | ...
```

Par exemple :

```
type carte = Roi | Dame | Valet | Nombre of int
let carte_as = Nombre 1
```

Un constructeur avec plusieurs arguments prend en fait un unique argument qui est le $n$-uplet des arguments. Par exemple :

```
type carte_avec_couleur =
    | Roi of string
    | Dame of string
    | Valet of string
    | Nombre of int * string
let as_pique = Nombre (1, "pique")
```

Remarque : comme pour les types enregistrement, on la compatibilité avec le polymorphisme et le filtrage. On peut même définir des types récursifs. Par exemple :

```
type 'a liste = Nil | Cons of 'a * 'a liste;;
let tete (l : 'a liste) : 'a =
    match l with
    | Nil -> failwith "Liste vide"
    | Cons (t, q) -> t
```

# 2 Structures de données séquentielles

## 2.1 Listes

### 2.1.1 Introduction

La structure de liste est une structure de donnée (SD) abstraite représentant un ensemble fini et ordonné de valeurs. Il n'y a pas vraiment d'interface standard pour cette SD.

Les primitives possibles pour cette SD sont :

- Opération de création de liste vide (commun à la plupart des SD) ;
- Un test de vacuité (commun à la plupart des SD) ;
- Une opération pour ajouter un élément (début / fin) ;
- Une opération pour accéder aux éléments (1$^{er}$, dernier, successeur, prédécesseur d'un élément) ;
- ...

Le plus souvent, le terme *liste* désigne la structure de liste simplement chaînée, qui est la notion principale de liste en programmation fonctionnelle (surtout en Lisp) et en théorie des types.

### 2.1.2 Listes simplement chaînées

• Définition :

La structure de liste simplement chaînée est une structure de liste dont les primitives sont les accès au 1$^{er}$ élément et au successeur d'un élément donné.

• Implémentation en OCaml : c'est le type `'a list` que l'on peut définir récursivement comme en 1.3.2. Les primitives sont les constructeurs du type, `Nil ([])` et `Cons (::)`, et les fonctions `List.hd` (accès au premier élément) et `List.tl` (accès au sucesseur). Ces opérations sont garanties en temps constant, contrairement à d'autres opérations qu'il faut implémenter.

Exemple : ajout en fin de liste :

```
let rec append (x : 'a) (l : 'a list) : 'a list =
    match l with
    | [] -> [x] (*x::[]*)
    | t::q -> t::(append x q)
```

• Implémentation en C : on définit un structure représentant un maillon de la chaîne, contenant une valeur et (on doit connaître le type à l'avance) un pointeur vers le maillon suivant (la structure est récursive).

```
struct maillon {
    int valeur;
    struct maillon* suivant;
};
typedef struct maillon* liste;
```

On utilise un pointeur car la liste vide ne peut être représentée comme un maillon. Ici, la liste vide sera le pointeur `NULL`.

Exemples de primitives :

```
int tete(liste l) {
    return l->valeur;
}

liste queue(liste l) {
    return l->suivant;
}
```

Attention, une liste doit être allouée dynamiquement.

```
liste list_of_arr(int* t, int n) {
    liste l = NULL;
    liste q = NULL;

    for (int k = n - 1 ; k >= 0 ; k--) {
        l = (liste) malloc(sizeof(struct maillon));
        assert(l != NULL);
        l->valeur = t[k];
        l->suivant = q;
        q = l;
    }

    return l;
}
```

### 2.1.3 Implémentation par tableau

On peut stocker les valeurs d'une liste dans un tableau (comme en Python).

Idée : on fixe une taille et on remplit un tableau de cette taille avec les éléments de la liste, du dernier au premier.

Ex, en OCaml :

```
type 'a liste = {valeur : 'a array; mutable nb_elts : int};;

let liste_vide (x : 'a) : 'a liste =
    {valeurs = Array.make 42 x; nb_elts = 0};;

let add (x : 'a) (l : 'a liste) : unit =
    l.valeurs.(l.nb_elts) <- x;
    l.nb_elts <- l.nb_elts + 1;;
```

Avantages : accès en temps constant à tous les éléments (premier, dernier, par index, successeur, prédécesseur), calcul de la taille en temps constant.

Problème : si on sature le tableau, on a une erreur à l'ajout suivant.

Solution : on utilise des tableaux dynamiques : on alloue un tableau plus grand après saturation. En OCaml, le champ `valeur` devient mutable.

Exo : implémentation de la fonction `add` en C dans laquelle on alloue un tableau de taille double après saturation.

```c
typedef struct {
    int* valeurs;
    int taille;
    int nb_elts;
} liste;

void add(int x, liste* l)
```

Rq : le choix de doubler la taille n'est pas annodin : `add` est de complexité constante dans le meilleur cas, linéaire dans pire cas (allocation d'un tableau de taille $2n$ + copie des $n$ premiers éléments), et de complexité amortie constante.

### 2.1.4 Analyse amortie

L'analyse amorite permet de déterminer le coût moyen d'une opération au sein d'une série d'opérations : on calcule la moyenne de la complexité dans le pire cas d'une série de $n$ opérations. Ainsi, une opération coûteuse ponctuelle est compensée par un grand nombre d'opérations peu coûteuses.

Attention, ne pas comfondre avec la complexité moyenne, qui est la moyenne du coût d'une seule opération sur toutes les entrées d'une taille donnée.

Méthodes pour effectuer une analyse amortie :

- Méthode de l'agrégat : on calcule un majorant $M(n)$ du coût de $n$ opérations. Le coût amorti est alors $\frac{M(n)}{n}$

  Ex : la fonction `add`. La $i$-ème opération coûte $O(1)$ sauf si $i$ est une puissance de 2 (si on part d'un tableau de taille 1). Dans ce cas, l'opération coûte $O(i)$.

  Le coût de $n$ opérations est alors inférieur à $n + \sum_{i=0}^{\log_2(n)} 2^i < n + 2n = 3n$

  Le coût amorti de l'opération d'ajout est alors $\frac{3n}{n} = 3$

- Méthode comptable : on surévalue le coût des opérations de faible complexité pour obtenir des "crédits" servant à "payer" les opérations de forte complexité. Les crédits sont associés à des objets stockés dans la SD : ils servent à payer les opérations sur ces objets.

  Ex : fonction `add` : chaque élément ajouté vient avec 3 crédits :
  - Un crédit pour l'ajout de l'élément ;
  - Un crédit pour sa prochaine copie ;
  - Un crédit pour la copie d'un autre élément déjà présent, et qui n'a plus de crédit.

  Juste après une extension, on a un tableau de taille $m$ sans crédit. Il contient $\frac{m}{2}$ éléments. Chacun des $\frac{m}{2}$ éléments qu'on peut ajouter paye pour son insertion, sa copie et la copie de l'un des $\frac{m}{2}$ premiers éléments.

• Méthode du potentiel : on associe une fonction de potentiel à la SD et on définit le coût amorti d'une opération comme son coût réel + la variation de potentiel induite par l'opération.

Si le potentiel final est supérieur au potentiel initial, on obtient bien un majorant pour la somme des coûts.

$\sum$ Coûts amortis = $\sum$ coûts réels + (potentiel final - potentiel initial)

Ex : pour les tableaux dynamiques : $\Phi(t) = 2\texttt{nb\_elts} - \texttt{taille}(t)$

Juste après une extension, $\Phi(t) = 0$

De plus, $\Phi(t) \geqslant 0$ car le tableau est toujours au moins à moitié rempli donc le potentiel final est supérieur au potentiel initial.

Coût amorti d'une opération :

– S'il n'y a pas d'extension : $1 + ((2\underbrace{(n+1)}_{\texttt{nb\_elts}} - \underbrace{c}_{\text{taille}}) - (2n - c)) = 3$

– S'il y a une extension : $n + 1 + ((2(n+1) - 2c) - (2n - c)) = n + 3 - c = 3$ car $n = c$

## 2.2 Piles et files

### 2.2.1 Motivations

Voici quelques problèmes que l'on souhaite résoudre :

(1) On veut implémenter la fonction retour arrière dans un navigateur / éditeur de texte.

Besoins :

– Stocker les états précédants
– Ajouter un nouvel état (nouvelle page)
– Obtenir l'état courant (affichage)
– Extraire l'état courant (retour arrière)

(2) On veut gérer les appels de fonctions dans un programme.

Besoins :

– Stocker les blocs d'activation des fonctions
– Ajouter un bloc (appel de fonction)
– Retirer un bloc (retour de fonction)

(3) On veut gérer les tâches d'un serveur d'impression.

Besoins :

– Stocker les tâches en attente
– extraire la tâche la plus ancienne
– Ajouter des tâches

(4) On veut gérer les entrées au clavier sur un ordinateur.

Besoins :

– Stocker les caractères entrés (dans un buffer)
– Ajouter des caractères au buffer (nouvelle entrée)
– Extraire les caractères les plus anciens, au besoin

Tous ces problèmes exprimment un besoin commun : stocker des données en attente d'être traités et les extraire dans un ordre précis. C'est le signe que l'élaboration d'une structure de données adaptée permettrait de les résoudre efficacement.

Ici, on peut concevoir deux structures selon l'ordre d'extraction :

– La structure de pile : dernier entré, premier sorti (LIFO, *last in, first out*), utile pour (1) et (2) ;
– La structure de file : premier entré, premier sorti (FIFO, *first in, first out*), utile pour (3) et (4).

Ce sont deux cas particuliers de la structure de file de priorité, que nous verrons plus tard.

### 2.2.2 Définition (structure de pile)

Une pile est une structure de données dont les opérations sont les suivantes :

– `pileVide` : crée une pile ;
– `estVide` : teste si une pile est vide ;
– `empiler` : ajoute un élément sur le sommet de la pile ;
– `dépiler` : extrait, *i.e* retire et renvoie, le sommet de la pile ;
– `sommet` (facultatif) : renvoit la valeur du sommet sans l'extraire.

Rq : Ceci est une structure dynamique (la SD peut changer) immuable (on ne peut pas modifier chaque bloc), qui peut être impérative persistante / fonctionnelle selon le type des opérations empiler / dépiler.

### 2.2.3 Concrétisation de la structure de pile

La structure de liste simplement chaînée est assez naturelle pour implémenter les piles :

– `pileVide` : renvoit la liste vide ;
– `estVide` : teste l'égalité avec la liste vide ;
– `empiler` : ajoute en tête de liste ;
– `dépiler` : extrait la tête ;
– `sommet` : renvoit la tête.

On peut donc utiliser une structure de mailles chaînées ou des tableaux pour implémenter les piles.

Exo :

• Implémenter en OCaml un type `'a stack` et ses primitives à partir du type `'a list`
• Maillons chaînés en C :

```c
struct maillon {
    int val;
    struct maillon* next;
};
typedef struct maillon* stack;

stack push(int x, stack s) { //structure persistante
    stack top = (stack) malloc(sizeof(struct maillon));
    top->val = x;
    top->next = s;
    return top;
}

stack pop(int* x, stack s) {
    assert(!estVide(s));
    *x = s->val;
    stack res = s->next;
    free(s);
    return res;
}

//Or
void pop(stack s, int* x, stack* res) {
    //Exo
}
```

• Tableau : comme pour les listes, on peut utiliser un tableau statique (en imposant une taille maximale) ou dynamique.

Différence : la réallocation (version dynamique) et la création de pile vide (versions statique) sont plus coûteuses.

Exo : implémentation en C et en OCaml.

• Module `Stack` d'OCaml : dans la bibliothèque standard, on dispose des objets suivants :

- type `'a Stack.t` des piles ;
- `Stack.create : unit -> 'a Stack.t` qui crée une pile vide ;
- `Stack.is_empty : 'a Stack.t -> bool` qui teste la vacuité de la pile ;
- `Stack.push : 'a -> 'a Stack.t -> unit` (donc structure impérative) ;
- `Stack.pop : 'a Stack.t -> 'a` qui extrait le sommet de la pile, et lève l'exception `Stack.Empty` si la pile est vide ;
- H.P `Stack.tp : 'a Stack.t -> 'a` qui renvoit le sommet de la pile, même exception si la pile est vide.

Concrètement, la type `t` du module `Stack` est défini comme `unit` :

```
type 'a t = {mutable C : 'a list}
```

### 2.2.4 Définition (structure abstraite de file)

Une file est une structure de données dont les opérations sont :

- `fileVide` : crée une file vide ;

– `estVide` : teste si une file est vide ;
– `enfiler` : insère un élément en fin de file ;
– `défiler` : extrait l'élément en début de file ;
– `premier` (facultatif) : renvoit l'élément en début de file.

### 2.2.5 Concrétisation de la structure de file

On a besoin d'accéder efficacement au premier et au dernier élément.

• Tableaux : on peut utiliser une structure de tableau associée à un indice de début et un indice de fin (*cf* TD)

• Maillon chaînés : on peut utiliser une structure de liste simplement chaînée associée à des pointeurs vers le premier et le dernier maillon.
C'est le principe de l'implémentation du module `Queue` d'OCaml, qui fournit les objets suivants :

– type `'a Queue.t`, des files ;
– `Queue.create : unit -> 'a Queue.t` ;
– `Queue.is_empty : 'a Queue.t -> bool` ;
– `Queue.add : 'a -> 'a Queue.t -> unit` Il y a un alias `Queue.push`, favorisé par le programme ;
– `Queue.take : 'a Queue.t -> 'a` (alias `Queue.pop`) ;
– H.P `Queue.peek : 'a Queue.t -> 'a` (alias `Queue.top`) qui lèvent l'exception `Queue.Empty` si la file est vide.

Implémentation :

```
type 'a cell =
    | Nil
    | Cons of {content : 'a; mutable next : 'a cell}

type 'a t = {
    mutable length : int;
    mutable first : 'a cell;
    mutable last : 'a cell;
}
```

Exo : savoir le faire en C.

## 2.3 Tableau associatifs

### 2.3.1 Définition (structure abstraite de tableau associatif)

Un tableau associatif, ou dictionnaire, est une structure de données permettant de stocker des associations entre des clés, prises dans un ensemble $K$, et des valeurs, prises dans un ensemble $V$. Pour toute clé $k \in K$, il ne peut y avoir qu'au plus une valeur.

Les opérations de la structure sont :

- Création d'un dictionnaire vide ;
- Test de vacuité ;
- Test de présence d'une association pour une clé donnée ;
- Recherche de la valeur associée à une clé ;
- Insertion / modification / supression de la valeur associée à une clé.

### 2.3.2 Remarque

Cette structure généralise celle de tableau (où les clés sont les indices).

C'est une structure mutable (on peut modifier les éléments), dynamique (on peut ajouter ou retirer des éléments).

### 2.3.3 Implémentation par maillons chaînés ou par tableaux

N'importe quelle implémentation de la structure de liste convient en stockant les couples (clés, valeurs).

Complexité des opérations

- Création : $O(1)$ pour les maillons chaînés, $O(t)$ pour les tableaux ($t$ la taille initiale) ;
- Test de vacuité : $O(1)$ ;
- Test de présence / recherche de valeur : $O(n)$ ;
- Insertion : $O(1)$ (pire cas pour les maillon, amorti pour les tableaux), en supposant que l'on utilise une nouvelle clé ;
- Modification, supression : $O(n)$ (il faut déjà trouver l'élément).

Les opérations les plus coûteuses le sont en raison de la recherche. On peut ramener en tête de liste les clés auxquelles on accède pour placer en début de liste les clés fréquemment recherchées afin d'améliorer la complexité moyenne (si l'on suppose une bonne distribution des clés).

On appelle cette structure une *liste auto-organisée*.

Si les clés appartiennent à un ensemble totalement ordonné (*cf* chap 5), on peut rendre la recherche plus efficace dans l'implémentation par tableau en utilisant l'algorithme de la recherche dichotomique, si on conserve le tableau trié selon les clés.

Les complexités qui changent sont :

- Test de présence / recherche de valeur : $O(\log n)$ ;
- Insertion : $O(n)$, car $O(\log n)$ pour la recherche de l'emplacement, et $O(n)$ pour le décalage ;
- Modification : $O(\log n)$ ;

Cette structure est efficace si on effectue peu d'insertions et de supressions.

Remarque : on a des fonctions dans le module `List` d'OCaml pour l'implémentation par maillon chaînés (H.P) :

– `List.assoc : 'a -> ('a * 'b) list -> 'b` qui lève l'exception `Not_found` si la clé passée en argument n'est pas dans la liste ;

– `List.assoc_opt : 'a -> ('a * 'b) list -> 'b option` ;

– `List.mem_assoc : 'a -> ('a * 'b) list -> bool`

– `List.remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list`, qui ne lève pas d'exception si l'association n'existe pas.

Exo : code.

### 2.3.4 Implémentation par table de hachage

Les problèmes de performance des implémentations précédantes sont liés à la recherche d'une clé dans la structure. Dans l'idéal, on voudrait accéder à la valeur associée à une clé en temps constant, comme dans un tableau indicé par les clés.

Si l'ensemble $K$ est assez petit, on peut numéroter les clés, grâce à une bijection

$$h : K \longrightarrow [\![0 \ ; \ \mathrm{card}(K) - 1]\!]$$

et utiliser un tableau (la valeur associée à la clé $k$ est dans la case d'indice $h(k)$).

Toutes les opérations se font en temps constant, sauf la création du tableau en $O(\mathrm{card}(K))$.

Problème : c'est trop coûteux en espace.

Idée : on réduit la taille du tableau, mais il y aura moins d'indices que de clés.

On utilise donc un tableau associé à une fonction

$$h : K \longrightarrow [\![0 \ ; \ m]\!]$$

où $m$ n'est pas trop grand, que l'on appelle *fonction de hachage*. On appelle $h(k)$ le *haché* de la clé $k$, $m$ la longueur de la table de hachage, et les cases du tableau les *alvéoles*.

Lorsque l'on a $k, k' \in K \mid h(k) = h(k')$, on parle de *collision*.

Les collisions empêchent de stocker directement la valeur associée à une clé dans l'alvéole d'indice le haché de la clé. Plusieurs techniques existent pour concevoir des fonctions de hachage limitant le nombre de collisions et pour gérer ces collisions. Elles sont toutes H.P. L'une d'entre elles est très classique : la résolution des collisions par chaînage : on stocke dans une alvéole d'indice $i$ une liste chaînée des associations (clé, valeur) telles que le haché de la clé vaut $i$.

Exemple : Prenons $m = 5$ et

$$h(k) = \left\lfloor mk\frac{\sqrt{5}-1}{2} \right\rfloor \mod m$$

et considérons les opérations suivantes :

– Création d'une table de hachage de taille $m$ ;

– Insertion de $v_0$ de clé 4 ($h(4) = 2$) ;

– Insertion de $v_1$ de clé 3 ($h(3) = 4$)

– Insertion de $v_2$ de clé 9 ($h(9) = 2$)

0 ☐ []
1 ☐ []
2 ☐ $[(9, v_2), (4, v_0)]$
3 ☐ []
4 ☐ $[(3, v_1)]$

Le coût des opérations est en $O$(longueur de l'alvéole concernée). On peut montrer que si la fonction de hachage est bien choisie, *i.e* si les clés sont uniformément réparties sur les alvéoles, la complexité moyenne des opérations est en $O(1 + \alpha)$, où $\alpha = \frac{n}{m}$ est le taux de remplissage de la table de hachage.

En conservant $m$ proportionnel à $n$, on obtient une complexité moyenne constante pour les opérations de dictionnaire.

Une possibilité est d'utiliser le double hachage si le nombre de clés devient trop grand : chaque alvéole est elle-même une table de hachage. Une autre possibilité est d'agrandir la table de hachage, donc de changer de fonction de hachage, à la manière des tableaux dynamiques (*cf* TD)

Remarque : le module `Hashtbl` d'OCaml fournit une implémentation des tables de hachage. On y trouve :

– le type `('a, 'b) Hashtbl.t` des tables de hachage dont les clés sont de type `'a` est les valeurs de type `'b` ;

– `Hashtbl.create : int -> ('a, 'b) Hashtbl.t` qui crée une table de hachage dont le nombre d'alvéoles est donné en paramètre ;

– `Hashtbl.add : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` ;

– `Hashtbl.replace : ('a, 'b) Hashtbl.t -> 'a -> 'b -> unit` H.P ;

– `Hashtbl.remove : ('a, 'b) Hashtbl.t -> 'a -> unit` ;

– `Hashtbl.mem : ('a, 'b) Hashtbl.t -> 'a -> bool` ;

– `Hashtbl.find : ('a, 'b) Hashtbl.t -> 'a -> 'b` qui lève l'exception `Not_found` si la clé n'est pas dans la table ;

– `Hashtbl.find_opt : ('a, 'b) Hashtbl.t -> 'a -> 'b opt` ;

– `Hashtbl.iter : ('a -> 'b -> unit) -> ('a, 'b) Hashtbl.t -> unit` Attention, l'ordre d'itération n'est pas spécifié ;