Option informatique, cours 6

Structures de données 3/3 : piles, files et dictionnaires

C. MULLAERT

Lycée Saint-Louis

Année 2024-2025

Plan du cours

1. Introduction
2. Piles mutables et persistantes, applications
3. Files mutables et persistantes, applications
4. Dictionnaires, applications

Introduction

La structure de liste

Précédemment dans le cours, on a vu le type `'a list` comme structure de données récursive et linéaire qui permet l'accès à la tête en **temps constant**.

À l'aide d'algorithmes récursifs, on sait réaliser les opérations suivantes en **temps linéaire** :

- trouver la longueur de la liste
- accéder au n-ième élément de la liste
- insérer ou supprimer un élément à la n-ième place

La structure de tableau

Précédemment dans le cours, on a vu le type `'a array` comme structure de données mutable et linéaire qui permet l'accès à n'importe quel élément en **temps constant**.

Si l'on souhaite réaliser et coder un tri, la structure de tableau est plus pertinente que celle de liste car le caractère mutable permet de réaliser des tris en place.

Si l'on souhaite rechercher si un élément est présent dans un ensemble trié, la structure de tableau est plus intéressante car l'accès en temps constant à l'élément médian permet de mettre en œuvre la dichotomie.

Mais le tableau a une taille fixée au moment de sa création. Cela peut être une contrainte.

Structure de données

En fonction du problème algorithmique à résoudre, une structure de données peut être plus ou moins adaptée et permettre une résolution efficace.

Lorsque l'on parle de "structure de données", il s'agit de préciser les éléments suivants :

- sa définition,
- son caractère mutable ou non,
- les opérations élémentaires que l'on peut faire avec, ainsi que leurs complexités.

Résumé liste/tableau

Le type `'a list` est non mutable. Il permet :

- l'ajout ou la suppression de la tête en temps constant,
- l'accès à un élément quelconque en temps linéaire.

Le type `'a array` est mutable. Il permet :

- l'accès à un élément quelconque en temps constant,
- l'ajout ou la suppression d'un élément nécessite de recopier le tableau. Ces opérations se font en temps linéaire.

Piles mutables et persistantes, applications

La structure de pile

On appelle **pile** une structure de données linéaire mutable, munie de deux opérations `push` et `pop` qui ajoutent et retirent un élément en temps constant.

L'élément retiré est le dernier à avoir été ajouté : c'est le principe LIFO (last in first out)

Si l'on exclut le caractère mutable, une liste peut être une implémentation d'une structure de pile.

Le module Stack

Ocaml propose une implémentation de la pile mutable avec le module `Stack`. Les trois actions que l'on peut faire sur une pile, dont le type est noté `'a t` sont :

- ajouter un élément avec la fonction `push`, de type `'a -> 'a t -> unit`, cette fonction modifie la structure en place et agit uniquement par effet de bord;
- retirer un élément avec la fonction `pop`, de type `'a t -> 'a`, cette fonction modifie la structure en place et renvoie la valeur du dernier élément.
- accéder au dernier élément ajouté, sans modification de la pile, avec la fonction `top`, de type `'a t -> 'a`.

Exemple d'utilisation du module Stack

```
open Stack;;
let pile = create ();;
for i=0 to 8 do push i pile done;;
length pile;;
for i=0 to 8 do print_int(pop pile) done;;
```

Le code précédent créée une pile, la remplit avec les entiers de 0 à 8, puis affiche ces nombres dans l'ordre décroissant (principe last in first out).

Notez l'effet de bord de `pop pile` qui, à la fois, renvoie la tête de la pile et la modifie alors qu'elle est passée en argument.

Exemple d'utilisation du module Stack

Si l'on ne connait pas la taille d'une pile et que l'on souhaite tout dépiler, il faut écrire une boucle while et rattraper l'exception `Empty` :

```
open Stack;;
let pile = create ();;
for i=0 to 8 do push i pile done;;
try while true do print_int(pop pile) done
   with | Empty -> ();;
```

Exemple d'implémentation d'une pile persistante

Comme on l'a vu, la structure de liste permet naturellement d'implémenter une pile persistante. Voici un exemple d'interface pour une telle structure :

```
type 'a pile = 'a list ;;
let create () = [];;
let push x p = x::p;;
let pop = function
        |[] -> failwith "Empty"
        |h::q -> q;;
let top = function
        |[] -> failwith "Empty"
        |h::q -> h;;
```

Comme on a une structure persistante, on ne peut pas modifier ce type de pile par effet de bord.

Exemple d'implémentation d'une pile mutable

Pour réaliser une structure de pile mutable, on peut par exemple utiliser une référence vers une liste :

```
type 'a pile = 'a list ref ;;
let create () = ref [] ;;
let push x p = p := x::(!p);;
let pop p = match !p with
        |[] -> failwith "Empty"
        |h::q -> p := q; h;;
```

Notez l'utilisation d'effets de bord pour les fonctions `push` et `pop`.

Application 1 : renversement d'une liste

On dispose d'une liste, et on souhaite construire une nouvelle liste dans laquelle les éléments apparaissent dans l'ordre inverse. Pour ce faire, on peut empiler tous les éléments de la liste, puis tous les dépiler.

```
let rev l =
  let p = create () in
  let rec aux = function
          | [] -> ()
          | h::q -> push h p; aux q;
  and aux2 () = try let h = pop p in h::aux2 ()
    with | Empty -> []
  in aux l; aux2 ();;
```

Bien sur, la fonction `List.rev` le fait déjà en temps linéaire, sans utiliser de pile.

Application 2 : évaluation d'une expression en notation postfixe

On rappelle que la notation postfixe (ou polonaise inversée) d'une expression correspond au parcours postfixe de son arbre. En voici un exemple : "2 3 + 5 -" correspond, en notation infixe à $(2 + 3) - 5$.

Pour évaluer une telle expression, on utilise naturellement une structure de pile. Après avoir créé une pile vide, on parcourt les termes de l'expression de gauche à droite. Pour chaque élément :

- s'il s'agit d'une opérande, on l'empile et on passe au terme suivant;
- s'il s'agit d'un opérateur, on dépile le bon nombre d'opérandes et on empile le résultat de l'opération élémentaire;
- lorsqu'il n'y a plus de terme à traiter, la pile contient le résultat.

Application 2 : évaluation d'une expression en notation postfixe

```
let is_operator s = (s="+") || (s="-");;
let apply s a b = match s with
  | "+" -> a+b
  | "-" -> a-b;;
let eval l =
  let p = create () in
  let rec aux = function
    | [] -> pop p
    | h::q -> if (is_operator h) then
      let a = pop p and b = pop p
      in push (apply h b a) p
      else push (int_of_string h) p; aux q
  in aux l;;
eval ["2";"3";"+";"5";"-"];; # renvoie 0
```

Application 3 : Conversion entre notation infixe et postfixe

On se donne une expression parenthésée en notation infixe, par exemple $(2+3)-5$ et on cherche à l'écrire en notation postfixe. On utilise pour cela l'algorithme de la "**gare de triage**" de Dijkstra.

On va lire les caractères de l'entrée un par un et réaliser des actions sur une pile de "triage" et une pile de "sortie". A la fin de l'opération, la pile de sortie contiendra l'expression sous forme postfixe.

Application 3 : Conversion entre notation infixe et postfixe

L'algorithme est le suivant :

- s'il n'y a plus d'éléments dans l'entrée, alors il faut dépiler toute la pile de triage dans la pile de sortie;
- si le premier élément de l'entrée est un nombre, on le place sur la pile de sortie;
- si le premier élément de l'entrée est une parenthèse gauche, on la place sur la pile de triage;
- si le premier élément de l'entrée est une parenthèse droite, on dépile tous les éléments de la pile de triage jusqu'à arriver à une parenthèse gauche et on place ces éléments dans la sortie. Les parenthèses ne sont pas placées dans la sortie;
- si le premier élément de l'entrée est un opérateur, on le place sur la pile de triage, après avoir déplacé tous les opérateurs déjà présents.

Application 3 : Conversion entre notation infixe et postfixe

On commence par définir un type de données adapté pour les éléments d'une expression.

```
open Stack;;
type token = P of string | Op of string | Num of int;;

let l =[P "(";Num 2;Op "+";Num 5;P ")";Op "-";Num 5];;
```

On créée les piles de triage et de sortie :

```
let triage = create () and sortie = create ();;
```

La fonction suivante convertit une pile en liste, de sorte que le haut de la pile soit en fin de liste :

```
let rec to_list acc = try let h = pop sortie
  in to_list (h::acc) with | Empty -> acc;;
```

Application 3 : Conversion entre notation infixe et postfixe

```
let rec aux = function
  | [] -> (try while true do push (pop triage) sortie
          done with | Empty -> ()); to_list []
  | h::q -> match h with
    | P "(" -> push (P "(") triage; aux q;
    | P ")" -> while (top triage <> (P "(")) do
            push (pop triage) sortie done;
            pop triage; aux q;
    | Op f -> while (try (match top triage with
            | Op _ -> true | _ -> false)
            with | Empty -> false) do
            push (pop triage) sortie done;
            push (Op f) triage; aux q;
    | Num k -> push (Num k) sortie; aux q;;
```

Application 3 : Conversion entre notation infixe et postfixe

Les points importants à noter sur le programme précédent sont :

- La fonction `to_list`, qui permet de vider une pile et de renvoyer la liste de ses éléments, de sorte que le sommet de la pile soit en dernière position dans la liste renvoyée.
- L'utilisation d'exception pour réaliser des opérations sur une pile jusqu'à la vider complètement sans générer d'erreur
- Attention, lors de l'utilisation de `try`, le type de l'expression qui vient après doit correspondre avec celui renvoyé en cas d'exception. Le code suivant ne compile pas :

```
try while true do push (pop triage) sortie done
  with | Empty -> to_list [];;
```

En effet, `to_list []` est de type `list` alors que `push (pop triage) sortie` est de type `unit`.

Files mutables et persistantes, applications

Le type file

On appelle file une structure de donnée linéaire de taille variable qui permet l'ajout d'un élément à sa tête en temps constant. Il est aussi possible, toujours en temps constant, d'accéder et de supprimer l'élément située à sa queue. On parle de structure FIFO (first in first out).

Il s'agit d'une structure habituellement **mutable** : l'accès et la suppression de l'élément de queue peut se faire avec une seule fonction qui renvoie la queue et modifie la file par effet de bord.

Le module Queue

Ocaml propose un module qui implémente la structure de file : le module `Queue`.

Comme pour le module `Array`, il faudra préalablement exécuter l'instruction `open Queue;;`, ou bien préfixer toutes les fonctions par `Queue.`

On créée une file vide, dont le type est noté `'a t` avec la fonction `create` qui est de type `unit-> 'a t`.

Le module Queue

Les trois actions que l'on peut faire sur une file sont :

- Ajouter un élément en tête avec la fonction `push`, de type `'a -> 'a t -> unit`. Cette fonction modifie la structure en place et agit uniquement par effet de bord.
- Accéder au dernier élément et le retirer avec la fonction `pop`, de type `'a t -> 'a`. Cette fonction modifie la structure en place et renvoie la valeur du dernier élément.
- Accéder au dernier élément sans le retirer avec la fonction `top`, de type `'a t -> 'a`.

L'exception `Empty` est levée si l'on tente de retirer un élément à une file vide.

Exemple d'utilisation du module Queue

```
open Queue;;
let file = create ();;
for i=0 to 8 do push i file done;;
length file;;
for i=0 to 8 do print_int(pop file) done;;
```

Le code précédent créée une file, la remplit avec les entiers de 0 à 8, puis affiche ces nombres dans le **même ordre** (principe first in first out).

Notez l'effet de bord de `pop file` qui, à la fois, renvoie la queue de la file et la modifie alors qu'elle est passée en argument.

Implémentation d'une structure de file persistante

Une implémentation courante des files se fait en utilisant un couple de listes : la première sert à enfiler les éléments et la deuxième à défiler. Lorsque la deuxième liste est vide, il faut retourner la première.

```
type 'a file = 'a list * 'a list;;
let create () = [],[];;
let push x p = let l1,l2 = p in (x::l1),l2;;
let rec pop = function
        |[],[] -> failwith "Empty"
        |l1,h::q -> l1,q
        |l1,[] -> pop ([],(List.rev l1)) ;;
let rec top = function
        |[],[] -> failwith "Empty"
        |l1,h::q -> h
        |l1,[] -> top ([],(List.rev l1)) ;;
```

Implémentation d'une structure de file persistante

Avec cette implémentation non mutable, les effets de bord ne sont pas autorisés.

Le point important de cette implémentation est la complexité des opérations `top` et `pop`. Si la deuxième liste n'est pas vide, elle s'exécute en temps constant. Sinon, il faut appeler `List.rev` qui a un temps d'exécution linéaire en la longueur de la première liste, mais ce cas intervient rarement.

Implémentation d'une structure de file persistante

On parle de complexité **amortie** : pour empiler, puis dépiler n éléments, il faudra utiliser une seule fois `List.rev` sur une liste à n éléments. On dit que le coût amorti d'une opération est **constant**.

On remarque que ce coût amorti constant ne dépend en fait pas de l'ordre dans lequel on réalise les opération d'emfilement et de défilement.

Implémentation d'une structure de file mutable

Une première implémentation d'une file mutable utilise la construction précédente en remplaçant les listes par des références. Les complexités sont inchangées.

```
type 'a file = 'a list ref * 'a list ref;;
let create () =  (ref [],ref []);;
let push x f = let (e,s)=f in e := x::(!e);;
let pop f = let (e,s)=f in (match (!e=[], !s=[]) with
  | true,true -> failwith "Empty"
  | _,true -> s := List.rev !e; e:=[]
  | _,_ -> ());
  let h::q = !s in s:=q; h;;
```

Implémentation d'une structure de file mutable

Une deuxième implémentation utilise les listes "doublement chainées" :

```
type 'a boite = {valeur: 'a;
                 mutable avant: 'a pointeur;
                 mutable apres: 'a pointeur}
and 'a pointeur = Vide | Boite of 'a boite ;;
type 'a file = {mutable debut: 'a pointeur;
                mutable fin: 'a pointeur };;
```

Avec cette définition, une file est un enregistrement comportant deux pointeurs mutables vers le début et la fin de la file. Ainsi, il est possible de modifier à la fois la tête (pour ajouter un élément) et la queue (pour en retirer). Le constructeur `Vide` sert à indiquer la fin de la file, côté tête comme côté queue.

Plus complexe, cette deuxième implémentation garantit un temps constant pour les opérations `push` et `pop`.

Implémentation d'une structure de file mutable

Avec cette implémentation, les fonctions qui manipulent les files s'écrivent :

```
let create () =  {debut=Vide ; fin=Vide};;
let push x f =
  let b = Boite {valeur=x; avant=Vide; apres=f.debut} in
  match f.debut with
    |Vide -> f.debut <- b; f.fin <- b
    |Boite c -> c.avant <- b; f.debut <- b;;
let pop f = match f.fin with
  |Vide -> failwith "Empty"
  |Boite b -> match b.avant with
    |Vide -> f.debut <- Vide; f.fin <- Vide; b.valeur
    |Boite c -> c.apres <- Vide; f.fin <- Boite c;
                b.valeur;;
```

Application 1 : Nombres de Hamming

Un nombre de Hamming est un entier natuel dont les seuls diviseurs sont 2, 3 et 5. Il s'écrit donc $2^a 3^b 5^c$ pour un certain triple $(a, b, c) \in \mathbb{N}^3$. Les premiers nombres de Hamming sont $1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ...$

On souhaite écrire une fonction qui calcule le n-ième nombre de Hamming. Il est possible de le faire en créant trois files, qui contiennent initialement la valeur 1. Pour générer un nouveau nombre de Hamming, on réalise l'opération suivante :

- on multiplie le nombre courant par 2 et on l'ajoute à la première file. On fait de même pour les autres files en multipliant par 3 et par 5;
- on défile le plus petit des trois nombres en tête des files : c'est le prochain nombre de Hamming;
- on défile également les têtes des autres files, si celles-ci sont égales au nombre de Hamming que l'on vient de générer.

Application 1 : Nombres de Hamming

```
let hamming n =
 let f2=create () and f3=create () and f5=create () in
  push 1 f2; push 1 f3; push 1 f5;
  for i=1 to (n-1) do
    let x2 = top f2 and x3 = top f3 and x5 = top f5 in
    let x = min x2 (min x3 x5) in
    if x=x2 then (let t = pop f2 in ());
    if x=x3 then (let t = pop f3 in ());
    if x=x5 then (let t = pop f5 in ());
    push (2*x) f2; push (3*x) f3; push (5*x) f5;
  done;
  let x2 = top f2 and x3 = top f3 and x5 = top f5 in
  min x2 (min x3 x5);;
```

Application 1 : Nombres de Hamming

La fonction précédente pourrait être optimisée : à chaque itération, trois nombres sont ajoutés aux files et il peut y avoir des doublons.

Application 2 : Parcours d'un arbre en largeur

La structure de file permet de réaliser simplement le **parcours en largeur** (ou hiérarchique) d'un arbre. Pour rappel, il s'agit de parcourir les nœuds de l'arbre niveaux par niveaux. On se donne la structure d'arbre suivante :

```
type arbre = Nil | Noeud of int * arbre * arbre;;
let ar = Noeud(1,
         Noeud(2,Noeud(4,Nil,Nil), Noeud(5,Nil,Nil)),
         Noeud(3,Nil,Nil));;
```

Si l'on souhaite afficher la valeur de tous les nœuds, le parcours de cet arbre donne :

- 12453 dans l'ordre préfixe
- 42513 dans l'ordre infixe
- 45231 dans l'ordre postfixe
- et 12345 dans l'ordre hiérarchique

Application 2 : Parcours d'un arbre en largeur

Pour réaliser un parcours en largeur, on va initialiser une file avec un élément : l'arbre dont on veut le parcours en largeur. À chaque étape, on retire un élément de la file, on affiche la valeur du nœud et on enfile les fils droit et gauche :

```
let parcours arbre =
 let file = create() in add arbre file;
 try while true do match pop file with
  | Nil -> ()
  | Noeud (a,fg,fd) -> print_int a;
          push fg file; push fd file;
 done with | Empty ->();;
```

Notez l'utilisation d'effet de bord avec `pop file`, et l'utilisation d'exceptions pour terminer une boucle `while` qui traite un à un les éléments d'une file.

Dictionnaires, applications

Le type dictionnaire

On appelle dictionnaire (ou table d'association) une structure de données mutable qui permet de stocker des couples $(k, v) \in K \times V$ (clé,valeur), avec les fonctionnalités suivantes :

- l'ajout d'un nouveau couple (clé,valeur) à un dictionnaire, ou sa suppression, se fait en temps constant
- l'accès à la valeur correspondant à une clé donnée se fait en temps constant.

L'ensemble K des clés est supposé totalement ordonné et chaque clé est présente au plus une fois dans le dictionnaire.

Cette structure généralise les tableaux (qui peuvent être considéré comme des dictionnaires finis et dont l'ensemble des clés est $[\![0, n]\!]$).

Le module Hashtbl

Ocaml dispose du module `Hashtbl` qui implémente une structure de dictionnaire. Les fonctions disponibles sont :

1. `create`, de type `bool->int->('a, 'b) t`, renvoit un dictionnaire vide à partir d'une taille initiale et d'un booléen qui indique comment construire la table de hachage (voir détails plus loin dans le cours)
2. `add`, de type `('a, 'b) t -> 'a -> 'b -> unit` modifie le dictionnaire par effet de bord en ajoutant un nouvel élément
3. `remove`, de type `('a, 'b) t -> 'a -> unit` modifie le dictionnaire par effet de bord en supprimant l'entrée associée à une clé
4. `find`, de type `('a, 'b) t -> 'a -> 'b` renvoie la valeur correspondant à une clé dans un dictionnaire. L'exception `Not_found` est appelée si la clé recherchée est absente.

Implémentation d'une structure de dictionnaire

Pour implémenter un dictionnaire, plusieurs idées peuvent être utilisées :

- Une liste de couples clé/valeur
- Un arbre binaire de recherche
- Une table de hachage

On va les étudier un à un, et examiner les complexités des opérations élémentaires pour chacun des cas.

Liste d'association

Une première idée simple pour implémenter un dictionnaire est de considérer une liste de couples clé/valeur. On obtient, par exemple, la spécification suivante (dans une version non mutable) :

```
type ('k,'v) dico = ('k * 'v) list;;

let create () = [];;
let add d k v = (k,v)::d;;
let rec find d k = match d with
  | [] -> raise Not_found
  | (kk,v)::_ when kk=k -> v
  | _::q -> find q k;;
```

Liste d'association

Avec la définition précédente, quelle est la complexité de l'ajout d'un élément ? de la recherche d'un élément ?

Proposer une implémentation de la fonction `remove`. Quelle est sa complexité ?

On remarque que cette solution, simple, ne permet pas d'obtenir les performances souhaitées. Par ailleurs, on n'a pas exploité le caractère ordonné de l'ensemble des clés, et il est possible pour une liste d'association de contenir plusieurs éléments avec la même clé. On va voir par la suite une implémentation plus sophistiquée.

Liste d'association

```
(* version 1 *)
let remove d k = match d with
  | [] -> []
  | (kk,_)::q when k=kk -> remove q k
  | a::q -> a::remove q k;;

(* version récursive terminale *)
let remove d k =
  let rec aux acc = function
    | [] -> acc
    | (kk,_)::q when k=kk -> aux acc q
    | x::q -> aux (x::acc) q
  in aux [] d;;
```

Arbre binaire de recherche

On considère un arbre binaire dont les nœuds sont étiquetés par les couples clé/valeur. On impose que les clés du fils gauche sont toutes inférieures à celle du nœud, qui est elle-même inférieure à toutes les clés du fils droit.

```
type ('k,'v) dico = Nil
 | N of ('k*'v) * ('k,'v) dico * ('k,'v) dico;;
```

Cette structure permet d'avoir une recherche plus efficace puisque l'on sait quel arbre fils explorer. On réalise une sorte de **dichotomie**.

Arbre binaire de recherche

L'implémentation de la recherche dans un arbre binaire de recherche se fait de la façon suivante :

```
let rec find d k = match d with
  | Nil -> raise Not_found
  | N((kk,v),_,_) when k=kk -> v
  | N((kk,_),fg,fd) ->
    if k<kk then find fg k else find fd k;;
```

Cette recherche a pour complexité la hauteur de l'arbre. Dans le cas d'un arbre bien équilibré (i.e. la différence de hauteur entre ses deux fils est au plus 1 et les arbres fils sont également bien équilibrés), on montre que la hauteur est un $O(\log n)$.

Arbre binaire de recherche

L'implémentation de l'ajout dans un arbre binaire de recherche se fait de la façon suivante :

```
let rec add d k v = match d with
  | Nil -> N((k,v),Nil,Nil)
  | N((kk,_),_,_) when k=kk -> failwith "clé présente"
  | N((kk,vv),fg,fd)  -> if k<kk
     then N((kk,vv),add fg k v,fd)
     else N((kk,vv),fg,add fd k v);;
```

Ici encore, la complexité est en $O(h)$ et, dans le cas d'un arbre équilibré, en $O(\log n)$.

Exercice : écrire la fonction qui réalise la suppression d'une entrée dans un arbre binaire de recherche. On écrira d'abord une fonction qui renvoit le couple clé/valeur correspondant à la plus grande clé d'un arbre

Arbre binaire de recherche

```
(* Fonction renvoyant la plus grande clé d'un arbre *)
let rec aux = function
 | Nil -> raise Error
 | N((k,v),_,Nil) -> (k,v)
 | N(_,_,fd) -> aux fd;;

let rec remove d k = match d with
 | Nil -> raise Not_found
 | N((kk,_),fg,fd) when k=kk -> (match (fg,fd) with
   | (Nil,fd) -> fd
   | (fg,Nil) -> fg
   | _ -> let (kk,vv) = aux fg
          in N((kk,vv),remove fg kk,fd) )
 | N((kk,vv),fg,fd) -> if k<kk
    then N((kk,vv),remove fg k v,fd)
    else N((kk,vv),fg,remove fd k v);;
```

En résumé, un arbre binaire de recherche permet de réaliser les opérations de recherche, d'ajout et de suppression en $O(\log n)$, sous réserve que l'arbre reste bien équilibré.

On va maintenant voir une troisième implémentation qui permet de réaliser ces opérations en temps constant.

Table de hachage

Le module `hashtbl` d'Ocaml utilise une troisième structure de données pour implémenter un dictionnaire : la **table de hachage**.

On se donne un entier $n \in \mathbb{N}$ et on souhaite stocker les valeurs du dictionnaire dans un tableau de taille n. Pour cela, on se donne une fonction de hachage $h : K \to [\![0, n-1]\!]$, et on range à l'indice $h(k)$ du tableau la valeur v correspondant au couple (k, v).

On convient qu'une valeur particulière v_0 est réservée pour les clés absentes du dictionnaire.

Ainsi, l'accès à la valeur correspondant à la clé k se fait en **temps constant**. La suppression et l'ajout sont également très simples : il suffit de mettre à jour la valeur.

En revanche, il y a un problème si la fonction h n'est pas injective car, alors, les valeurs associées aux clés k et k' et telles que $h(k) = h(k')$ sont rangées au même endroit. Ce phénomène s'appelle une **collision**.

Ces collisions sont inévitables car, bien souvent, l'ensemble des clés possibles est bien plus grand que n. Un choix judicieux de la fonction de hachage permet de limiter les collisions, ou, au moins, de bien les répartir sur l'ensemble $[\![0, n-1]\!]$.

Table de hachage

Une manière de traiter ces collisions consiste à stocker dans le tableau à l'indice $h(k)$ une liste d'association contenant tous les couples (k', v) tels que $h(k) = h(k')$.

```
type ('k,'v) dico = ('k*'v) list ref array;;
let create () = [| [] |];;
```

Il reste à implémenter les fonction d'ajout, de suppression et de recherche, et d'étudier leur complexité.

Table de hachage

Pour insérer un élément (k, v) dans une table de hachage, il faut calculer $h(k)$ et ajouter l'entrée (k, v) à la liste située à l'indice $h(k)$. Pour rechercher un élément, on recherche dans la liste à l'indice $h(k)$.

```
let add d k v = let h = hash k in
  d.(h) <- (k,v)::d.(h);;

let find d k = let h = hash k in
  let rec aux l k = match l with
    | [] -> raise Not_found
    | (kk,v)::_ when k=kk -> v
    | _::q -> aux q k
  in aux d.(h) k;;
```

L'implémentation de la suppression suit le même modèle.

Table de hachage

La complexité de l'ajout est toujours en $O(1)$. Celle des opérations de recherche et de suppression dépend du nombre de collisions :

- en l'absence de collision, la recherche et la suppression sont en temps constant
- dans le pire des cas, seule une valeur de hachage est utilisée et on retrouve la complexité linéaire des listes d'association.

On perçoit l'intérêt d'une fonction de hachage qui utilise uniformément l'ensemble des indices. Ainsi, le nombre de collisions est minimal. Avec une telle fonction, la longueur des listes d'associations est de m/n, avec m le nombre de clés utilisées.

Ainsi, en choisissant n de l'ordre du nombre de clés utilisées et une bonne fonction de hachage, on garantit une **complexité moyenne constante**.

Application : la mémoïsation

Une des applications les plus importante de la structure de dictionnaire est la **mémoïsation**. Il s'agit de stocker dans un dictionnaire les résultats des évaluations d'une fonction afin de n'exécuter qu'une seule fois le calcul pour chaque combinaison d'argument.

Prenons l'exemple du calcul du coefficient binomial et de son implémentation récursive naïve (on suppose $n \geq p$) :

```
let rec binom n p = match (n,p) with
 | (n,0) -> 1
 | (n,p) when n = p -> 1
 | (n,p) -> binom (n-1) p + binom (n-1) (p-1);;
```

Application : la mémoïsation

```
let rec binom n p = match (n,p) with
 | (n,0) -> 1
 | (n,p) when n = p -> 1
 | (n,p) -> binom (n-1) p + binom (n-1) (p-1);;
```

Cette fonction comporte deux appels récursifs et nécessite $\binom{n}{p}$ appels récursifs pour produire le résultat. Beaucoup de ces appels concerne l'évaluation de la fonction avec des arguments identiques.

On a vu que la **programmation dynamique** permet de minimiser les calculs et aboutir à une complexité en $O(np)$.

Application : la mémoïsation

Une version mémoïsée de la fonction `binom` peut s'écrire de la façon suivante :

```
let binomemo n p =
 let ht = create (n*p) in
 let rec  aux n p = match (n,p) with
  | (n,0) -> 1
  | (n,p) when n = p -> 1
  | (n,p) -> try find ht (n,p) with
      | Not_found -> let v = aux (n-1) p + aux (n-1) (p-1)
                     in add ht (n,p) v; v;
 in aux n p;;
```

Notez l'utilisation de la fonction `find` associée à la gestion des exceptions qui permet de dissocier le cas où le calcul a déjà été fait de celui où il reste à faire.

Écrire une fonction récursive "classique" et une fonction récursive utilisant un dictionnaire renvoyant le terme d'indice (n, p) de la fonction d'Ackermann définie sur $\mathbb{N}^2$ par

$$A : (n,p) \mapsto \begin{cases} p+1 & \text{si } n = 0 \\ A(n-1,1) & \text{si } p = 0 \\ A(n-1, A(n,p-1)) & \text{sinon} \end{cases}$$

Comparez-les pour $(n, p) = (3, 10)$.