Chapitre 3

Tables

3.1 Définition

Définition 1 (Table)**.** *Une table t est une fonction partielle de E dans S, où E est un type d'entrées (aussi dit index, indice ou clé) et S un type de valeurs.*

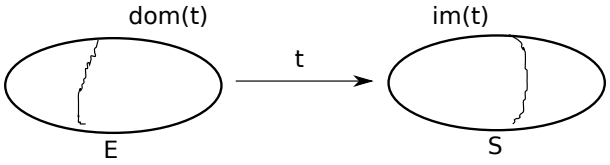

FIGURE 3.1 – Schéma général d'une table.

On note $dom(t)$ l'ensemble (ou domaine) de définition de la table t (Fig.3.1). On l'appelle également ensemble d'entrées dans t. On note $im(t)$ l'image de t dans S.

Exemple : une table t : chaine $\rightarrow$ chaine fait correspondre à un nom de département le code postal. Par exemple, pour Strasbourg le code 67.
On peut éventuellement compléter une table avec un élément particulier de E noté ω tel que $\omega \notin im(t)$, en posant $t'(i) =$ **si** $i \in dom(t)$ **alors** $t(i)$ **sinon** ω.

Les opérations usuelles sur les tables sont :
- ADJONCTION$(t, e, s) \rightarrow t$: adjonction d'un élément à la table, de clé e et de valeur s ;
- SUPPRESSION$(t, e) \rightarrow t$: suppression de l'élément de clé e ;
- ELEMENT$(t, e) \rightarrow s$: accès à la valeur de l'élément de clé e ;

3.2 Spécification algébrique

Nous proposons ci-dessous une spécification de la sorte table, telle que le type d'entrée et celui des valeurs sont resp. symbolisés par des sortes e resp. s, et avec $\omega : \rightarrow s$.

sorte table(**sortes** e,s **tq** $\omega : \rightarrow s$)
utilise bool
opérations
tabnouv : $\rightarrow$ *table*
adj : $table \times e \times s \rightarrow table$
sup : $table \times e \rightarrow table$
mod : $table \times e \times s \rightarrow table$

elem : $table \times e \rightarrow s$

vide : *table* $\rightarrow$ **bool**

axiomes
elem(tabnouv,i) = ω
vide(tabnouv) = **true**
elem(adj(t,i,x),j)= **si** $i = j$ **alors** x **sinon** elem(t,j)
vide(adj(t,i,x),j) = **false**
sup(adj(t,i,x),j) = **si** $i = j$ **alors** t **sinon** adj(sup(t,j),i,x)
mod(t,i,x) = adj(sup(t,i),i,x)

préconditions
pré(adj(t,i,x)) $\equiv$ $x \neq \omega \wedge elem(t,i) = \omega$
pré(sup(t,i)) $\equiv$ $elem(t,i) \neq \omega$
pré(mod(t,i,x)) $\equiv$ $x \neq \omega \wedge elem(t,i) \neq \omega$

Notons que elem(t,i) peut être noté plus simplement t(i), voire t[i] selon le contexte.

3.3 Représentation des tables

Il est important de ne pas confondre table et tableau. Un tableau est un cas particulier de la table avec comme entrée (index ou clé) un entier et comme type de valeur, le type du tableau. Evidemment, le tableau peut servir de moyen de modéliser des tables, mais ce n'est pas la seule approche. Par exemple, dans certains cas une table peut être représentée par une fonction. La table qui associe aux n premiers entiers leur carré, est directement donnée sous la forme d'une fonction mathématique du calcul de carré.
Pour modéliser des tables, on passe souvent par un ensemble d'adresses intermédiaires :

E $\rightarrow$ A $\rightarrow$ S
i $\rightarrow$ a(i) $\rightarrow$ f(a(i))=t(i)

On appelle $a()$ la fonction d'adressage et $f()$ la fonction d'accès à la valeur recherchée.

3.3.1 fonction d'accès

Accès direct

Dans ce cas f donne le contenu d'une cellule dont on connaît l'adresse. Pour une entrée i, $a(i)$ donne directement l'indice de la case du tableau (ou une adresse mémoire) contenant $t(i) = f(a(i))$.
Exemple : La table du nombre des jours dans le mois. On utilise un tableau $v[12]$. Avec $a(i) = i-1$, on obtient $t(i) = v[i-1]$.

Accès indirect

Ici la fonction f peut être vue comme donnant le contenu du contenu d'une cellule dont on connaît l'adresse. On peut utiliser un tableau ta pour ranger les adresses des valeurs de la table t. Celles-ci peuvent être mémorisées dans un autre tableau v.
Exemple : Soit t la table des noms de département à partir de leurs numéros, allant de 1 à 95. $a(i) = i-1$. Dans le schéma ci-dessus, on voit que ta donne l'indice du caractère dans le tableau v des chaînes. Cet exemple est utile dans le cas où le tableau est statique, car les modifications ne sont pas très aisées.

3.3.2 Modes d'adressage

Adressage calculé

Le principe consiste à calculer $a(i)$ à l'aide d'une fonction injective (c'est une fonction telle que $a(i) = a(j)$ implique $i = j$). C'était déjà le cas dans les exemples précédents.

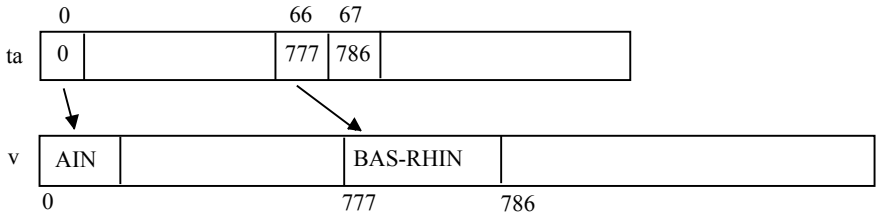

FIGURE 3.2 – Table des noms de département.

Exemple : En langage ADA, un tableau à plusieurs dimensions se déclare de la manière suivante : t : array ($a_1..b_1$,$a_2..b_2$, ... , $a_p..b_p$) of TYPE. Un tableau de ce type peut être vu comme une table associant à des p-uplets une valeur de TYPE. Cette table peut elle même être implémentée sous forme de tableau v dont les éléments sont indicés de 0 à $n-1$, avec $n = (b_1 - a_1 + 1)(b_2 - a_2 + 1)...(b_p - a_p + 1)$.

Il faut passer d'un p-uplet d'indices $i = (i_1, i_2, ..., i_p)$. à l'indice $a_p(i)$ correspondant dans v. Le calcul se fait par relation de récurrence : $a_p(i) = (b_p - a_p + 1)a_{p-1}(i_1, ..., i_{p-1}) + i_p - a_p$. En développant cette formule, on obtient une combinaison linéaire des $i_k - a_k$.

Adressage associatif

Dans un adressage associatif, on représente uniquement les entrées effectives par une liste les adresses sont associées.

Exemple : Soit une table t qui associe à un nom de département le nom du chef-lieu.

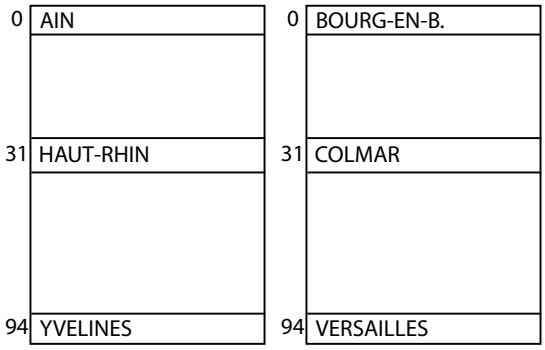

FIGURE 3.3 – Table des chef-lieux par département.

$a(i)$ est l'indice de la case contenant i. La valeur de $t(i)$ est obtenue par un tableau associé $v[a(i)]$. Dans ce cas, $a(i)$ est obtenu par recherche dichotomique (les départements sont triés par ordre alphabétique). La fonction d'accès est directe.

En pratique, il n'est pas nécessaire d'utiliser deux tableaux avec deux déclarations séparées : on peut utiliser une structure qui associe les éléments. Pour l'exemple de la table des chef-lieux par département, on peut définir une structure couple qui définit deux chaines de caractères :

```
typedef struct { char *cheflieux; char *departement; } couple;
couple tablecheflieux[N];
```

Le problème général de ce type de représentation est que les opérations d'adjonctions ou suppressions sont lourdes. Il devient alors parfois interessant pour ces traitements, d'utiliser une représentation mixte, contiguë et chaînée.

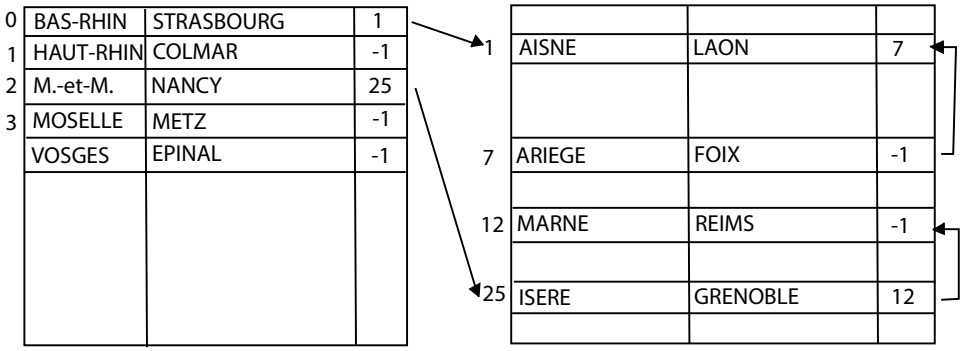

FIGURE 3.4 – Table des chef-lieux par département, avec une structure mixte contiguë et chaînée.

Dans l'exemple précédent, on utilise deux tableaux de structure pour réaliser la table. Le premier tableau contient les éléments de manière contiguë et triée. Dans le second les éléments sont chaînés. Le chaînage est réalisé par le biais d'indices. Le second tableau peut être considéré comme une zone de débordement.

La seule difficulté avec ce type de représentation est la gestion d'un nombre éventuellement important de débordements. En pratique, quand on s'aperçoit que l'on utilise beaucoup d'éléments en débordement, il faut réorganiser les listes en déplaçant des éléments du second tableau vers le premier tableau.

3.3.3 Partage de tables

Comme nous venons de le voir, il peut être efficace de partitionner les entrées en sous-ensembles, c'est-à-dire d'avoir des sous-tables (qui peuvent d'ailleurs être chainées à plusieurs niveaux). A deux niveaux, on parle de table majeure et table mineure.

Exemple : Nous voulons réaliser une table qui associe à une personne (nom et prénom) un numéro de téléphone. Il se peut que plusieurs personnes portent le même noms. Nous faisons un partage de table où nous associons toutes les personnes portant le même nom dans la table majeure. Dans la table mineure nous plaçons le prénom et le numéro de téléphone. On parle dans ce cas d'un **rangement partitionné**, avec adressage associatif.

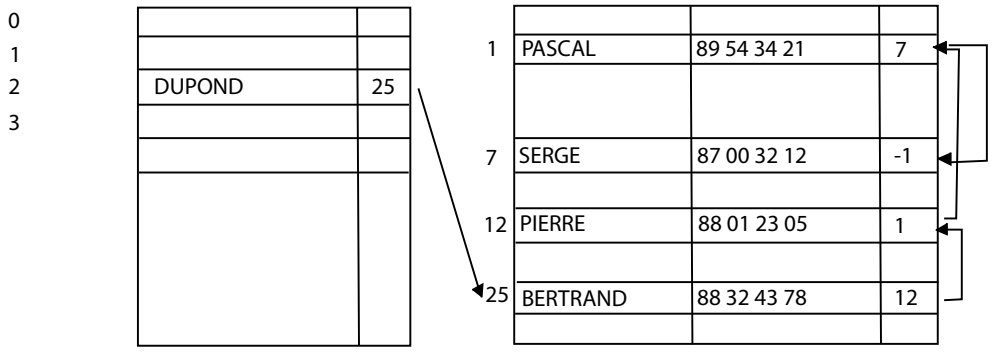

FIGURE 3.5 – Exemple de partage de table.

Lorsque le tableau des entrées est totalement ordonné, et que le partage en sous-tables est fait en partageant en **intervalles**, chacun correspondant à une table mineure, alors on parlera plutôt de **rangement indexé**.

Dans l'exemple précédent les éléments de la table mineure sont chainés. (notons que cela revient aussi à définir une liste chainée, sans nécessairement avoir besoin de recourir à une allocation de tableau). Les éléments ne sont

pas ordonnés. Or , il est tout à fait possible de définir une table mineure dans laquelle les éléments sont aussi rangés de façon contigue et triée, comme pour la table majeure. Ce processus récursif peut même être défini à plus de deux niveaux de tables : la table mineure point elle–même vers une table mineure de niveau 2 etc. Ce type d'organisation est très utilisé, en particulier à plus de deux niveaux, pour les rangements dans les bases de données : la structure résultant d'un chainage sur plusieurs niveaux, tel que, en chaque niveau, les éléments sont triés et partitionnés en intervalles, est celle d'un **B–arbre de recherche** que nous étudierons ultérieurement.

3.4 La rangement dispersé

Le rangement dispersé (hachage ou hashing, hash–coding) est un rangement où la fonction d'adressage est directement calculée quelque soit le type d'entrée. Le hachage a la particularité de permettre un temps de recherche constant, c'est-à-dire indépendant du nombre de données.

Exemple : On veut gérer une table de personnes indexées par leur prénom. A chaque prénom on associe un entier $h(i)$ de 0 à 12 en procédant comme suit :

— Attribuer aux lettres leur rang dans l'alphabet (A->1, B->2, etc.)
— Ajouter les valeurs des rangs pour chaque lettre composant le nom i.
— Ajouter au nombre obtenu le nombre de lettre de i.
— Prendre le reste de la division de ce nombre par 13.

On obtient pour les prénoms : h("serge")=7, h("odile")=11, h("luc")=0, h("anne")=12, h("basile")=2, h("elise")=3. On peut placer les prénoms dans un tableau en utilisant comme position la valeur de $h(i)$. Le problème est que si l'on veut insérer paule, on constate que h("paule")=2. On dit qu'il y a collision primaire entre paule et basile.

Toute la difficulté du hachage consiste :
— à trouver une fonction de hachage appropriée pour éviter le plus possible ces collisions
— quoi qu'il en soit il est impossible d'éviter toujours les collisions (surtout lorsque l'ensemble d'entrées est grand), de ce fait il faut trouver un moyen de résoudre les collisions sans pour autant augmenter inconsidérément l'espace de mémorisation.

3.4.1 Les fonctions de hachage

Le choix d'une bonne fonction de hachage dépend de l'ensemble d'entrées sur lequel on travaille réellement. Par exemple la fonction qui associe à une chaîne le rang de son initiale est uniforme sur l'ensemble des chaînes possibles mais pas sur l'ensemble des noms communs français. Il y a par exemple plus de mots qui commencent par B que par Z.

Voici quelques types et techniques, basé sur le codage binaire. En effet, quelque soit le type d'entrée, le codage de l'information sera binaire. On peut donc simplement considérer la séquence binaire pour effectuer un calcul de hachage.

par extraction

On extrait un certain nombre de bits de la représentation binaire.

Exemple : extraire les bits 1, 2, 7 et 8 en comptant à partir de la droite "ET" => 00101 10100, h("ET")=1000=8, "IL" => 01001 01100, h("IL")=0000=0

Mais ce type de fonction n'est pas vraiment bon, car elle ne dépend que d'une partie des données.

Compression

On utilise tous les bits pour obtenir l'indice dans le tableau. On coupe les chaînes de bits en paquets d'égales longueurs et on les additionne. Pour éviter des débordements il arrive de remplacer l'addition par un XOR :

h("ET")=00101 XOR 10100 = 10001 = (17)10, h("IL")= 01001 XOR 01100 = 00101 = (5)10

Pour éviter de hacher de la même façon toutes les permutations d'un même mot, ici h("IL")=h("LI"), on effectue par exemple des décalages circulaires (rotations). Par exemple : en décalant de 1 rang le premier paquet, de deux le deuxième, etc.

Division

On calcule le reste de la division par un entier mn (un nombre d'entrée dans le tableau) de la codification binaire :

Exemple avec n=37 : 'ET' = 00101 10100 = $(180)_{10}$, h("ET") = 180 mod 37 = 32

Malheureusement ce type de fonction dépend fortement de n. Si n est pair, alors tous les éléments pairs vont donner des indices pairs. Si en entrée il y plus de pairs que d'impairs cela produit des accumulations. On est alors amené à choisir n premier.

multiplication

Etant donné un nombre réel q, $0 < q < 1$, on pose : $h(i)$ = partie entière de (partie décimale de $(i * q) * n$), avec $q = 0.6125423371$ et $n = 30$.

h("ET") = partie entière de (partie décimale de $(180 \times 0.6125423371) \times 30$) = partie entière de (partie décimale de $(110.2576) \times 30$) = partie entière de (0.2576×30) = partie entière de ($7.7\ldots$) = 7
On remarque que statistiquement on obtient les meilleurs résultats pour $q = (\sqrt{5}-1)/2$ ou $1-(\sqrt{5}-1)/2$.

3.4.2 Résolution des collisions

En utilisant une table mineure

On procède comme pour le partage de table. Le schéma ci-dessous illustre cette approche.

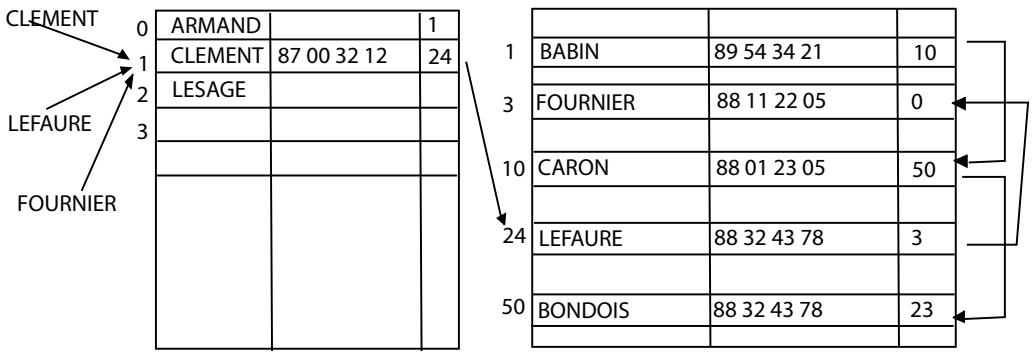

FIGURE 3.6 – Exemple de partage de table pour la résolution de collisions.

Sans utiliser de table mineure

Lorsqu'un élément j est en collision avec i, c'est-à-dire $h(i) = h(j)$, on essaie de placer j dans une autre case libre. Pour mémoriser le chainage, il est possible d'ajouter à la structure un indice vers le suivant.

Mais une alternative consiste à ne pas utiliser explicitement de pointeur mais à résoudre les collisions par calculs d'adresses successifs dans un espace mémoire unique. Par exemple, si l'espace $h(i)$ est déjà occupé, on regarde en $h(j)+1$ (à la case suivante). Si celle-ci est également occupée, la suivante etc., jusqu'à trouver un emplacement libre. A présent, lorsque l'on recherche un élément x, ce dernier ne se trouve pas nécessairement en $h(x)$. Il peut se trouver plus bas. On cherche donc séquentiellement jusqu'à tomber sur x, ou sur une case vide.

Un problème se pose lorsque l'on veut supprimer un élément. En effet, marquer simplement la case vide risque d'interrompre "la chaine" pour l'accès à d'autres données. Pour cela, on utilise un marqueur spécial, de case libérée. Il faut donc distinguer trois états : occupé, libre (vide) et libéré.

Lorsqu'un élément i est en collision et qu'on place i à un autre endroit du tableau (par exemple pemière case v libre en dessous de $u = h(i)$), on risque d'engendrer de nouvelles collisions en cet endroit v, par exemple avec un élément j, non encore inséré, tel que $v = h(j)$. En pratique, cela revient à fusionner des chaines de placements. Ici les chaines de $h(i)$ et $h(j)$ se retrouvent fusionnées en une seule chaine, c'est pour cela que l'on parle de **hachage coalescent**.

Une façon de réduire les risques de regroupements d'éléments, consiste à essayer de disperser plus ces éléments quand il y a collision. Ceci peut être fait en appliquant une fonction d de décalage $h(i) + k \times d(i)$, k étant le niveau de collision. On parle alors de double hachage. Mais il faut s'assurer que la redistribution engendrée par d balaie tous les emplacements du tableau.

De façon générale, l'approche par coalescence permet d'optimiser l'espace mémoire occupé, mais allonge les temps de traitement. La complexité des opérations de recherche ou d'insertion sont difficiles à évaluer. On peut cependant constater que cette méthode reste très performante.