# *Chương 12* – BẢNG VÀ TRUY XUẤT THÔNG TIN

Chương này tiếp tục nghiên cứu về cách tìm kiếm truy xuất thông tin đã đề cập ở chương 7, nhưng tập trung vào các bảng thay vì các danh sách. Chúng ta bắt đầu từ các bảng hình chữ nhật thông thường, sau đó là các dạng bảng khác và cuối cùng là bảng băm.

## 12.1. Dẫn nhập: phá vỡ rào cản lgn

Trong chương 7 chúng ta đã thấy rằng, bằng cách so sánh khóa, trung bình việc tìm kiếm trong n phần tử không thể có ít hơn lg n lần so sánh. Nhưng kết quả này chỉ nói đến việc tìm kiếm bằng cách so sánh các khóa. Bằng một vài phương pháp khác, chúng ta có thể tổ chức các dữ liệu sao cho vị trí của một phần tử có thể được xác định nhanh hơn.

Thật vậy, chúng ta thường làm thế. Nếu chúng ta có 500 phần tử khác nhau có các khóa từ 0 đến 499, thì chúng ta sẽ không bao giờ nghĩ đến việc tìm kiếm tuần tự hoặc tìm kiếm nhị phân để xác định vị trí một phần tử. Đơn giản chúng ta chỉ lưu các phần tử này trong một mảng kích thước là 500, và sử dụng chỉ số n để xác định phần tử có khóa là n bằng cách tra cứu bình thường đối với một bảng.

Việc tra cứu trong bảng cũng như  việc tìm kiếm có chung một mục đích, đó là truy xuất thông tin. Chúng ta bắt đầu từ một khóa và mong muốn tìm một phần tử chứa khóa này

Trong chương này chúng ta tìm hiểu cách hiện thực và truy xuất các bảng trong vùng nhớ liên tục, bắt đầu từ các bảng hình chữ nhật thông thường, sau đó đến các bảng có một số vị trí hạn chế như các bảng tam giác, bảng lồi lõm. Sau đó chúng ta chuyển sang các vấn đề mang tính tổng quát hơn, với mục đích tìm hiểu cách sử dụng các mảng truy xuất và các bảng băm để truy xuất thông tin.

Chúng ta sẽ thấy rằng, tuỳ theo hình dạng của bảng, chúng ta cần có một số bước để truy xuất một phần tử, tuy vậy, thời gian cần thiết vẫn là 0(1) - có nghĩa là, thời gian có giới hạn là một hằng số và độc lập với kích thước của bảng- và do đó việc tra cứu bảng có thể đạt hiệu quả hơn nhiều so với bất kỳ phương pháp tìm kiếm nào.

Các phần tử của các bảng mà chúng ta xem xét được đánh chỉ số bằng một mảng các số nguyên, tương tự cách đánh chỉ số của mảng. Chúng ta sẽ hiện thực các bảng được định nghĩa trừu tượng bằng các mảng. Để phân biệt giữa khái niệm trừu tượng và các hiện thực của nó, chúng ta có một quy ước sau:

Chỉ số xác định một phần tử của một bảng định nghĩa trừu tượng được bao bởi cặp dấu ngoặc đơn, còn chỉ số của một phần tử trong mảng được bao bởi cặp dấu ngoặc vuông.

Ví dụ, T(1,2,3) là phần tử của bảng T được đánh chỉ số bởi dãy số 1, 2, 3, và A[1][2][3] tương ứng phần tử với chỉ số trong mảng A của C++.

## 12.2. Các bảng chữ nhật

Do tầm quan trọng của các bảng chữ nhật, hầu hết các ngôn ngữ lập trình cấp cao đều cung cấp mảng hai chiều để chứa và truy xuất chúng, và nói chung người lập trình không cần phải bận tâm đến cách hiện thực chi tiết của nó. Tuy nhiên, bộ nhớ máy tính thường có tổ chức cơ bản là một mảng liên tục (như một mảng tuyến tính có phần tử này nằm kế phần tử kia), đối với mỗi truy xuất đến bảng chữ nhật, máy cần phải có một số tính toán để chuyển đổi một vị trí trong hình chữ nhật sang một vị trí trong mảng tuyến tính. Chúng ta hãy xem xét điều này một cách chi tiết hơn.

### 12.2.1. Thứ tự ưu tiên hàng và thứ tự ưu tiên cột

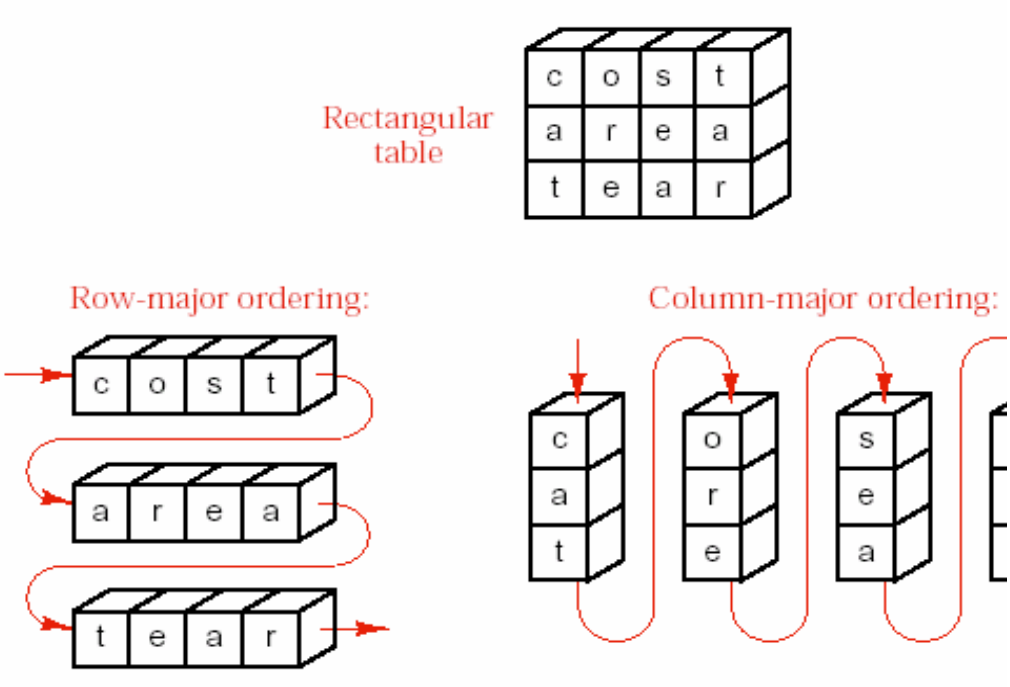
Cách tự nhiên để đọc một bảng chữ nhật là đọc các phần tử ở hàng thứ nhất trước, từ trái sang phải, sau đó đến các phần tử hàng thứ hai, và cứ thế tiếp tục cho đến khi hàng cuối đã được đọc xong. Đây cũng là thứ tự mà đa số các trình biên dịch lưu trữ bảng chữ nhật, và được gọi là thứ tự ưu tiên hàng (*row-major ordering*). Chẳng hạn, một bảng trừu tượng có hàng được đánh số là từ 1 đến 2, và cột được đánh số từ 5 đến 7, thì thứ tự của các phần tử theo thứ tự ưu tiên hàng như sau:

(1,5) (1,6) (1,7) (2,5) (2,6) (2,7)

Đây cũng là thứ tự được dùng trong C++ và hầu hết các ngôn ngữ lập trình cấp cao để lưu trữ các phần tử của một mảng hai chiều. FORTRAN chuẩn lại sử dụng thứ tự ưu tiên cột, trong đó các phần tử của cột thứ nhất được lưu trước, rồi đến cột thứ hai,v.v...Ví dụ thứ tự ưu tiên cột như sau:

(1,5) (2,5) (1,6) (2,6) (1,7) (2,7)

Hình 12.1 minh họa các thứ tự ưu tiên cho một bảng có 3 hàng và 4 cột.




**Hình 12.1** – Biểu diễn nối tiếp cho mảng chữ nhật

### 12.2.2. Đánh chỉ số cho bảng chữ nhật

Một cách tổng quát, trình biên dịch có thể bắt đầu từ chỉ số (i,j) để tính vị trí trong một mảng nối tiếp mà một phần tử tương ứng trong bảng được lưu trữ. Chúng ta sẽ đưa ra công thức tính toán sau đây. Để đơn giản chúng ta chỉ sử dụng thứ tự ưu tiên hàng cùng với giả thiết là hàng được đánh số từ 0 đến m-1, và cột từ 0 đến n-1. Trường hợp các hàng và các cột được đánh số không phải từ 0 được xem như bài tập. Số phần tử của bảng sẽ là mn, và đó cũng là số phần tử trong hiện thực liên tục trong mảng. Chúng ta đánh số các phần tử trong mảng từ 0 đến mn –1. Để có công thức tính vị trí của phần tử (i,j) trong mảng, trước hết chúng ta quan sát một vài trường hợp đặc biệt. Phần tử (0,0) nằm tại vị trí 0, các phần tử thuộc hàng đầu tiên trong bảng rất dễ tìm thấy: (0,j) nằm tại vị trí j. Phần tử đầu của hàng thứ hai (1,0) nằm ngay sau phần tử (0,n-1), đó là vị trí n. Tiếp theo, chúng ta thấy phần tử (1,j) nằm tại vị trí n+j. Các phần tử của hàng kế tiếp cũng sẽ nằm sau số phần tử của hai hàng trước đó (2n phần tử). Do đó phần tử (2,j) nằm tại vị trí 2n+j. Một cách tổng quát, các phần tử thuộc hàng i có n i phần tử phía trước, nên công thức chung là:

*Phần tử (i,j) trong bảng chữ nhật nằm tại vị trí n i + j trong mảng nối tiếp.*

Công thức này cho biết vị trí trong mảng nối tiếp mà một phần tử trong bảng chữ nhật được lưu trữ, và được gọi là hàm chỉ số (*index function*).

### 12.2.3. Biến thể: mảng truy xuất

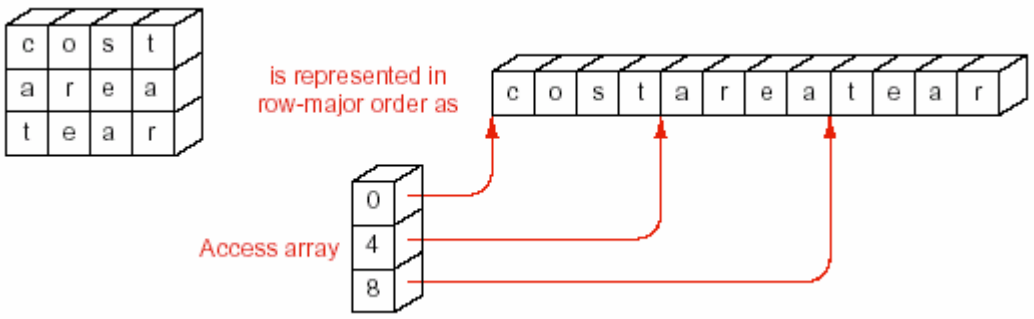
Việc tính toán cho các hàm chỉ số của các bảng chữ nhật thật ra không khó lắm, các trình biên dịch của hầu hết các ngôn ngữ cấp cao sẽ dịch hàm này sang ngôn ngữ máy thành một số bước tính toán cần thiết. Tuy nhiên, trên các máy tính nhỏ, phép nhân thường thực hiện rất chậm, một phương pháp khác có thể được sử dụng để tránh phép nhân.

Phương pháp này lưu một mảng phụ trợ chứa một phần của bảng nhân với thừa số là n:

0, n, 2n, 3n, ..., (m-1)n.

Lưu ý rằng mảng này nhỏ hơn bảng chữ nhật rất nhiều, nên nó có thể được lưu thường trực trong bộ nhớ. Các phần tử của nó chỉ phải tính một lần (và chúng có thể được tính chỉ bằng phép cộng). Khi gặp một yêu cầu tham chiếu đến bảng chữ nhật, trình biên dịch có thể tìm vị trí của phần tử (i,j) bằng cách lấy phần tử thứ i trong mảng phụ trợ cộng thêm j để đến vị trí cần có.

Mảng phụ trợ này cung cấp cho chúng ta một ví dụ đầu tiên về một mảng truy xuất (*access mảng*) (Hình 12.2). Nói chung, một mảng truy xuất là một mảng phụ trợ được sử dụng để tìm một dữ liệu được lưu trữ đâu đó. Mảng truy xuất có khi còn được gọi là *vector* truy xuất (*access vector*).




**Hình 12.2** – Mảng truy xuất cho bảng chữ nhật

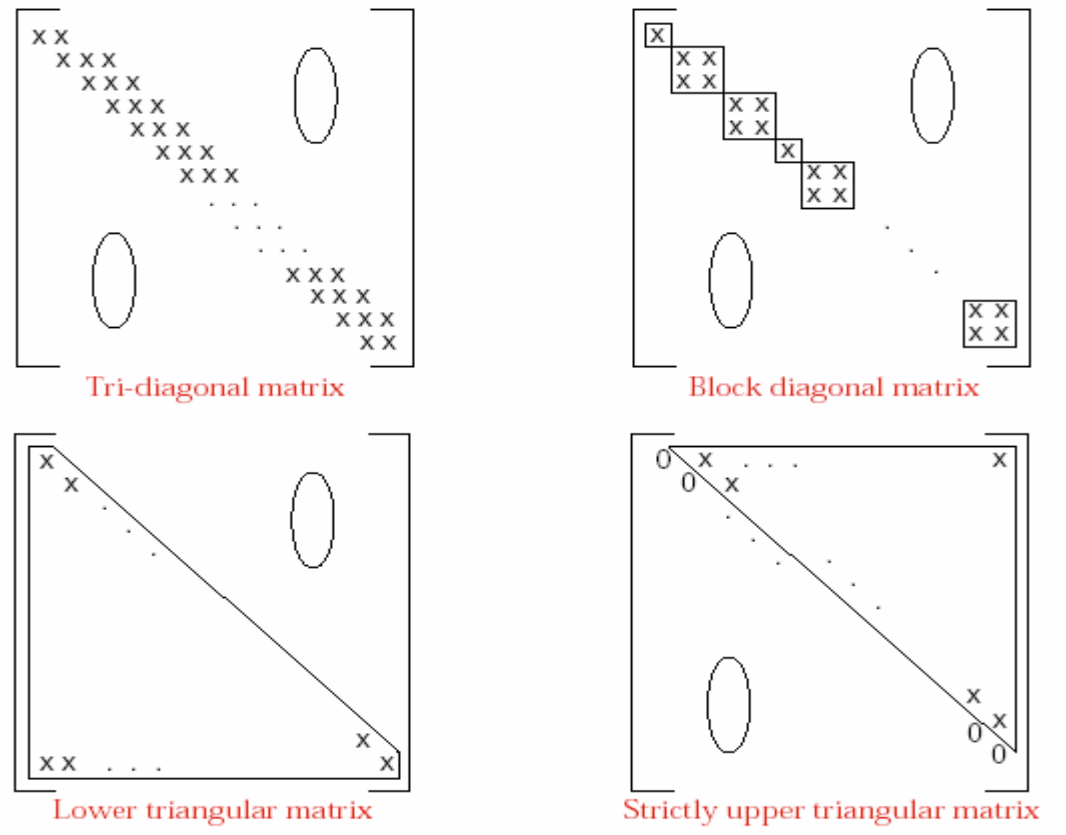
## 12.3. Các bảng với nhiều hình dạng khác nhau

Thông tin thường lưu trong một bảng chữ nhật có thể không cần đến mọi vị trí trong hình chữ nhật đó. Nếu chúng ta định nghĩa ma trận là một bảng chữ nhật gồm các con số, thì thường là một vài vị trí trong ma trận đó mang trị 0. Một vài ví dụ như thế được minh họa trong hình 12.3. Ngay cả khi các phần tử trong một bảng không phải là những con số, các vị trí được sử dụng thực sự cũng có thể không phải là tất cả hình chữ nhật, và như vậy có thể có cách hiện thực khác hay hơn thay vì sử dụng một bảng chữ nhật với nhiều chỗ trống. Trong phần này, chúng ta tìm hiểu các cách hiện thực các bảng với nhiều hình dạng khác nhau,

những cách này sẽ không đòi hỏi vùng nhớ cho những phần tử vắng mặt như trong bảng chữ nhật.

### 12.3.1. Các bảng tam giác

Chúng ta hãy xem xét cách biểu diễn bảng tam giác dưới như trong hình vẽ 12.3. Một bảng như vậy chỉ cần các chỉ số (i,j) với i≥j. Chúng ta có thể hiện thực một bảng tam giác trong một mảng liên tục bằng cách trượt mỗi hàng ra sau hàng nằm ngay trên nó, như cách biểu diễn ở hình 12.4.

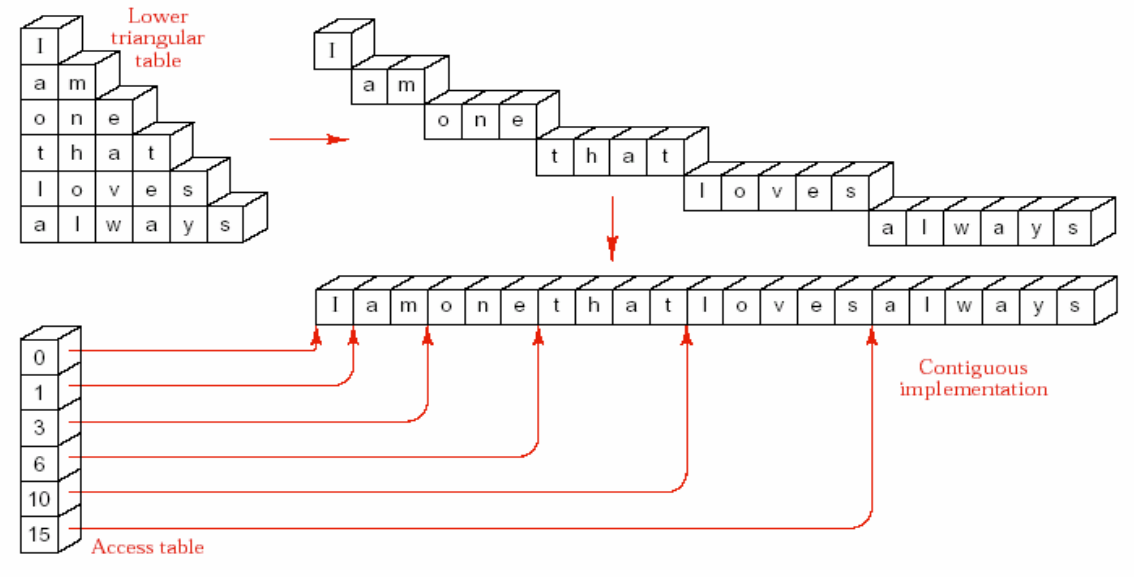



**Hình 12.3** – Các bảng với nhiều dạng khác nhau.

Để xây dựng hàm chỉ số mô tả cách ánh xạ này, chúng ta cũng giả sử rằng các hàng và các cột đều được đánh số bắt đầu từ 0. Để tìm vị trí của phần tử (i,j) trong mảng liên tục chúng ta cần tìm vị trí bắt đầu của hàng i, sau đó để tìm cột j chúng ta chỉ việc cộng thêm j vào điểm bắt đầu của hàng i. Nếu các phần tử của mảng liên tục cũng được đánh số bắt đầu từ 0, thì chỉ số của điểm bắt đầu của hàng thứ i cũng chính là số phần tử nằm ở các hàng trên hàng i. Rõ ràng là trên hàng thứ 0 có 0 phần tử, và chỉ có một phần tử của hàng 0 là xuất hiện trước hàng 1. Đối với hàng 2, có 1 + 2 = 3 phần tử đi trước, và trong trường hợp tổng quát chúng ta thấy số phần tử có trước hàng i chính xác là:

$$1 + 2 + \ldots + i = \frac{1}{2} i(i + 1).$$

Vậy phần tử (i,j) trong bảng tam giác tương ứng phần tử $\frac{1}{2}$ i(i + 1) + j của mảng liên tục.

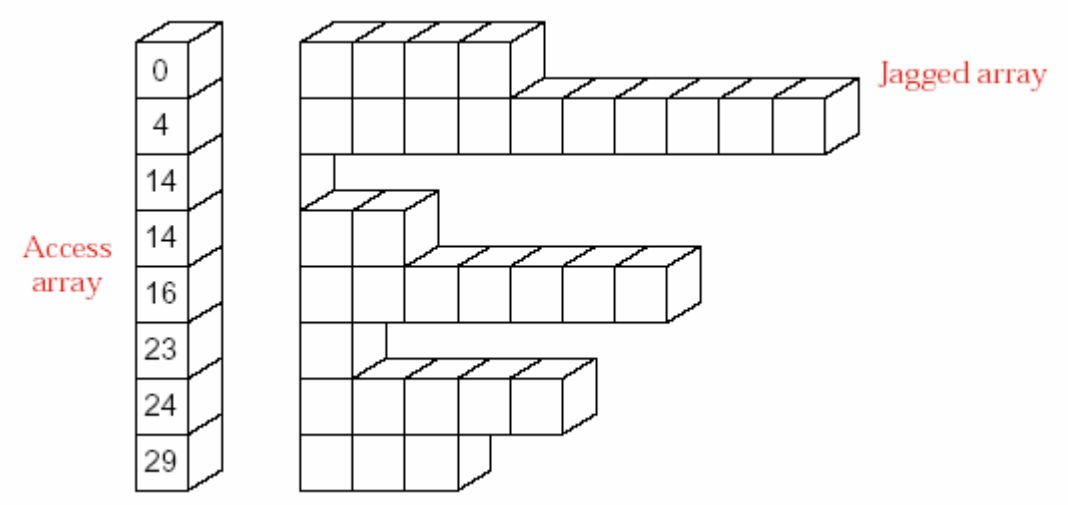



**Hình 12.4** – Hiện thực liên tục của bảng tam giác.

Cũng như chúng ta đã làm cho các bảng chữ nhật, chúng ta cũng tránh mọi phép nhân và chia bằng cách tạo một mảng truy xuất chứa các phần tử tương ứng với các chỉ số của các hàng trong bảng tam giác. Vị trí i trong mảng truy xuất mang trị $\frac{1}{2}$i (i + 1). Mảng truy xuất được tính toán chỉ một lần khi bắt đầu chương trình, và được sử dụng lặp lại cho mỗi truy xuất đến bảng tam giác. Chú ý rằng ngay cả việc tính toán ban đầu cũng không cần đến phép nhân hoặc chia mà chí có phép cộng theo thứ tự sau mà thôi:

0, 1, 1+2, (1 + 2) + 3, . . .

### 12.3.2. Các bảng lồi lõm




**Hình 12.5** – Mảng truy xuất cho bảng lồi lõm.

Trong cả hai ví dụ đã đề cập trước chúng ta đã xem xét một bảng được tạo từ các hàng của nó. Trong các bảng chữ nhật thông thường, tất cả các hàng đều có cùng chiều dài; trong bảng tam giác, chiều dài mỗi hàng có thể được tính dựa vào một công thức đơn giản. Bây giờ chúng ta hãy xem xét đến trường hợp của các bảng lồi lõm tựa như hình 12.5, không có một mối quan hệ có thể đoán trước nào giữa vị trí của một hàng và chiều dài của nó.

Một điều hiển nhiên được nhìn thấy từ sơ đồ rằng, tuy chúng ta không thể xây dựng một hàm thứ tự nào để ánh xạ một bảng lồi lõm sang vùng nhớ liên tục, nhưng việc sử dụng một mảng truy xuất cũng dễ dàng như các ví dụ trước, và các phần tử của bảng lồi lõm có thể được truy xuất một cách nhanh chóng. Để tạo mảng truy xuất, chúng ta phải xây dựng bảng lồi lõm theo thứ tự vốn có của nó, bắt đầu từ hàng đầu tiên. Phần tử 0 của mảng truy xuất, cũng như trước kia, là bắt đầu của mảng liên tục. Sau khi mỗi hàng của bảng lồi lõm được xây dựng xong, chỉ số của vị trí đầu tiên chưa được sử dụng tới của vùng nhớ liên tục chính là trị của phần tử kế tiếp trong mảng truy xuất và được sử dụng để bắt đầu xây dựng hàng kế của bảng lồi lõm.

### 12.3.3. Các bảng chuyển đổi

Tiếp theo, chúng ta hãy xem xét một ví dụ minh họa việc sử dụng nhiều mảng truy xuất để tham chiếu cùng lúc đến một bảng các phần tử qua một vài khóa khác nhau.

Chúng ta xem xét nhiệm vụ của một công ty điện thoại trong việc truy xuất đến các phần tử về các khách hàng của họ. Để in danh mục điện thoại, các phần tử cần sắp thứ tự tên khách hàng theo *alphabet*. Tuy nhiên, để xử lý các cuộc gọi đường dài, các phần tử lại cần có thứ tự theo số điện thoại. Ngoài ra, để tiến hành bảo trì định kỳ, danh sách các khách hàng sắp thứ tự theo địa chỉ sẽ có ích cho các nhân viên bảo trì. Như vậy, công ty điện thoại cần phải lưu cả ba, hoặc nhiều hơn, danh sách các khách hàng theo các thứ tự khác nhau. Bằng cách này, không những tốn kém nhiều vùng lưu trữ mà còn có khả năng thông tin bị sai lệch do không được cập nhật đồng thời.

Chúng ta có thể tránh được việc phải lưu nhiều lần cùng một tập các phần tử bằng cách sử dụng các mảng truy xuất, và chúng ta có thể tìm các phần tử theo bất kỳ một khóa nào một cách nhanh chóng chẳng khác gì chúng đã được sắp thứ tự theo khóa đó. Chúng ta sẽ tạo một mảng truy xuất cho tên các khách hàng. Phần tử đầu tiên của mảng này chứa vị trí của khách hàng đứng đầu danh sách theo alphabet. Phần tử thứ hai chứa vị trí khách hàng thứ hai, và cứ thế. Trong mảng truy xuất thứ hai, phần tử đầu tiên chứa vị trí của khách hàng có số điện thoại nhỏ nhất. Tương tự, mảng truy xuất thứ ba có các phần tử chứa vị trí của các khách hàng theo thứ tự  địa chỉ của họ. (Hình 12.6)

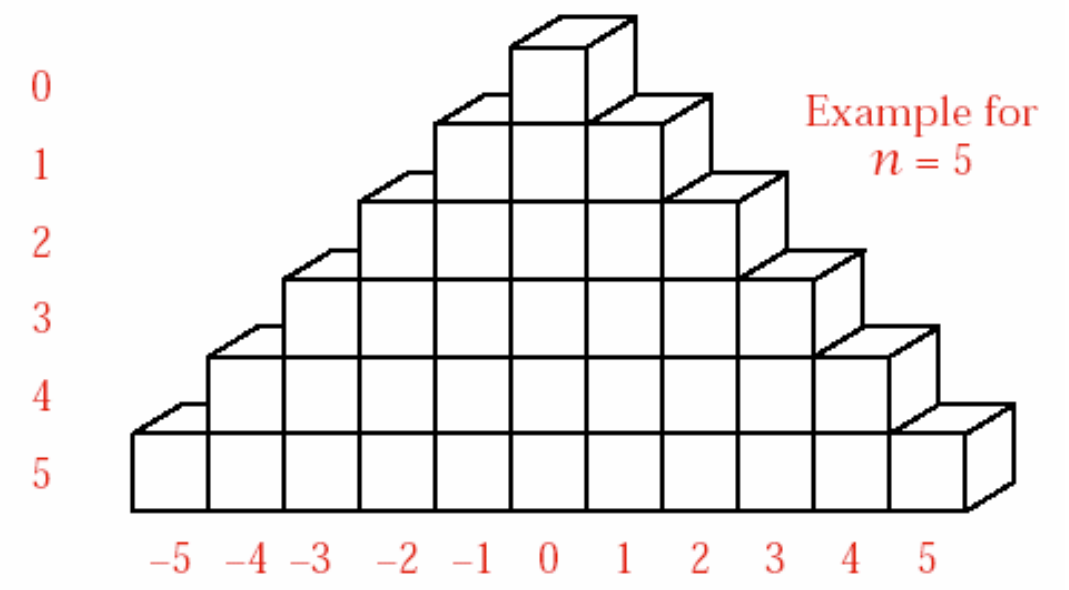
| *Index* | *Name* | *Address* | *Phone* |
|---|---|---|---|
| 1 | Hill, Thomas M. | High Towers #317 | 2829478 |
| 2 | Baker, John S. | 17 King Street | 2884285 |
| 3 | Roberts, L. B. | 53 Ash Street | 4372296 |
| 4 | King, Barbara | High Towers #802 | 2863386 |
| 5 | Hill, Thomas M. | 39 King Street | 2495723 |
| 6 | Byers, Carolyn | 118 Maple Street | 4394231 |
| 7 | Moody, C. L. | High Towers #210 | 2822214 |

*Access Tables*

| *Name* | *Address* | *Phone* |
|---|---|---|
| 2 | 3 | 5 |
| 6 | 7 | 7 |
| 1 | 1 | 1 |
| 5 | 4 | 4 |
| 4 | 2 | 2 |
| 7 | 5 | 3 |
| 3 | 6 | 6 |

**Hình 12.6** – Mảng truy xuất cho nhiều khóa: bảng chuyển đổi

Chúng ta lưu ý rằng trong phương pháp này các thành phần dữ liệu được xem như là khóa đều được xử lý cùng một cách. Không có lý do gì buộc các phần tử phải có thứ tự vật lý ưu tiên theo khóa này mà không theo khóa khác. Các phần tử có thể được lưu trữ theo một thứ tự tùy ý, có thể nói đó là thứ tự mà chúng được nhập vào hệ thống. Cũng không có sự khác nhau giữa việc các phần tử được lưu trong một danh sách liên tục là mảng (mà các phần tử của các mảng truy xuất chứa các chỉ số của mảng này) hay các phần tử đang thuộc một danh sách liên kết (các phần tử của các mảng truy xuất chứa các địa chỉ đến từng phần tử riêng). Trong mọi trường hợp, chính các mảng truy xuất được sử dụng để truy xuất thông tin, và, cũng giống như các mảng liên tục thông thường, chúng có thể được sử dụng trong việc tra cứu các bảng, hoặc tìm kiếm nhị phân, hoặc với bất kỳ mục đích nào khác thích hợp với cách hiện thực liên tục.



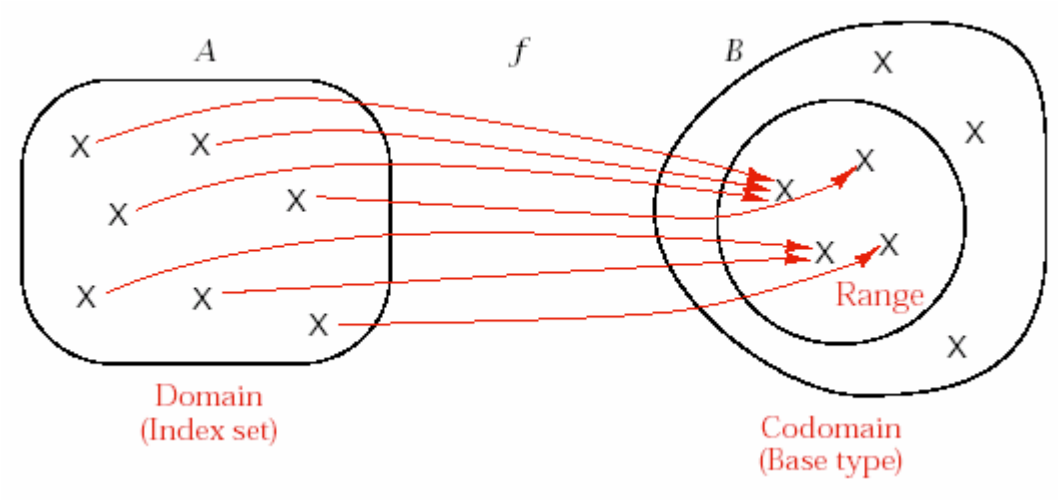

**Hình 12.7** – Ví dụ về bảng tam giác đối xứng qua 0.

## 12.4. Bảng: Một kiểu dữ liệu trừu tượng mới

Từ đầu chương này chúng ta đã biết đến một số hàm chỉ số được dùng để tìm kiếm các phần tử trong các bảng, sau đó chúng ta cũng đã gặp các mảng truy xuất là các mảng được dùng với cùng một mục đích như các hàm chỉ số. Có một sự giống nhau rất lớn giữa các hàm với việc tra cứu bảng: với một hàm, chúng ta bắt đầu bằng một thông số để tính một giá trị tương ứng; với một bảng, chúng ta bắt đầu bằng một chỉ số để truy xuất một giá trị dữ liệu tương ứng được lưu trong bảng. Chúng ta hãy sử dụng sự tương tự này để xây dựng một định nghĩa hình thức cho thuật ngữ **bảng**.

### 12.4.1. Các hàm

Trong toán học, một hàm được định nghĩa dựa trên hai tập hợp và sự tương ứng từ các phần tử của tập thứ nhất đến các phần tử của tập thứ hai. Nếu f là một hàm từ tập A sang tập B, thì f gán cho mỗi phần tử của A một phần tử duy nhất của B. Tập A được gọi là *domain* của f, còn tập B được gọi là *codomain* của f. Tập con của B chỉ chứa các phần tử là các trị của f được gọi là *range* của f. Định nghĩa này được minh họa trong hình 12.8.




**Hình 12.8** – *Domain*, *codomain* và *range* của một hàm

Việc truy xuất bảng bắt đầu bằng một chỉ số và bảng được sử dụng để tra cứu một trị tương ứng. Đối với một bảng chúng ta gọi *domain* là tập chỉ số (*index set*), và *codomain* là kiểu cơ sở (*base type*) hoặc kiểu trị (*value type*). Lấy ví dụ, chúng ta có một khai báo mảng như sau:

```
double array[n];
```

thì tập chỉ số là tập các số nguyên từ 0 đến n-1, và kiểu cơ sở là tập tất cả các số thực. Lấy ví dụ thứ hai, chúng ta hãy xét một bảng tam giác có m hàng, mỗi phần tử có kiểu `item`. Kiểu cơ sở sẽ là kiểu `item` và tập chỉ số là tập các cặp số nguyên

$$\{(i,j) \mid 0 \le j \le i < m\}$$

### 12.4.2. Một kiểu dữ liệu trừu tượng

Chúng ta đang đi đến một định nghĩa cho bảng như một kiểu dữ liệu trừu tượng mới, đồng thời trong các chương trước chúng ta đã biết rằng để hoàn tất một định nghĩa cho một cấu trúc dữ liệu, chúng ta cần phải đặc tả các tác vụ đi kèm.

*Định nghĩa*: Một bảng với tập chỉ số I và kiểu cơ sở T là một hàm từ I đến T kèm các tác vụ sau:
1. *Access* (truy xuất bảng): Xác định trị của hàm theo bất kỳ một chỉ số trong I.
2. *Assignment* (ghi bảng): Sửa đổi hàm bằng cách thay đổi trị của nó tại một chỉ số nào đó trong I thành một trị mới được chỉ ra trong phép gán.

Hai tác vụ này là tất cả những gì được cung cấp bởi các mảng trong C++ hoặc một vài ngôn ngữ khác, nhưng đó không phải là lý do để có thể ngăn cản chúng ta thêm một số tác vụ khác cho một bảng trừu tượng. Nếu so sánh với định nghĩa

của một danh sách (*list*), chúng ta đã có các tác vụ như thêm phần tử, xóa phần tử cũng như truy xuất hoặc cập nhật lại. Vậy chúng ta có thể làm tương tự đối với bảng.

*Các tác vụ bổ sung cho bảng*:
1. *Creation* (Tạo): Tạo một hàm từ I vào T.
2. *Clearing* (Dọn dẹp): Loại bỏ mọi phần tử trong tập chỉ số I, *domain* sẽ là một tập rỗng.
3. *Insertion* (Thêm): Thêm một phần tử x vào tập chỉ số I và xác định một trị tương ứng của hàm tại x.
4. *Deletion* (Xóa): Loại bỏ một phần tử x trong tập chỉ số I và hạn chế chỉ cho hàm xác định trên tập chỉ số còn lại.

### 12.4.3. Hiện thực

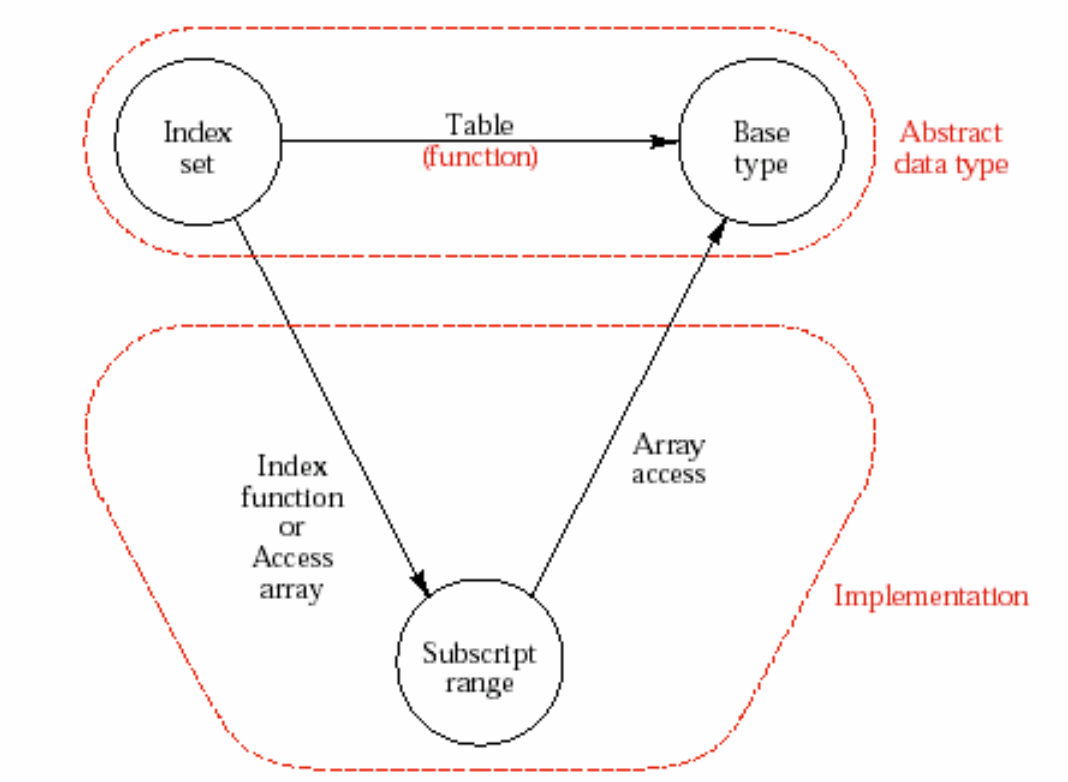
Định nghĩa trên chỉ mới là định nghĩa của một kiểu dữ liệu trừu tượng mà chưa nói gì đến cách hiện thực. Nó cũng không hề nhắc đến các hàm chỉ số hay các mảng truy xuất. Chúng ta hãy xem hình minh họa trong hình 12.9. Phần trên của hình này cho chúng ta thấy một sự trừu tượng trong định nghĩa, truy xuất bảng đơn giản chỉ là một ánh xạ từ một tập chỉ số sang một kiểu cơ sở. Phần dưới của hình là ý tưởng tổng quát của phần hiện thực. Một hàm chỉ số hoặc một mảng truy xuất nhận thông số từ một tập chỉ số theo một dạng đã được đặc tả nào đó. Chẳng hạn (i,j) trong bảng 2 chiều hoặc (i,j,k) trong bảng 3 chiều với i, j, k đã có miền xác định đã định. Kết quả của hàm chỉ số hoặc mảng truy xuất sẽ là một trong các trị trong miền các chỉ số, chẳng hạn tập con của tập các số nguyên. Miền trị này có thể được sử dụng trực tiếp như chỉ số cho mảng và được cung cấp bởi ngôn ngữ lập trình.

Đến đây xem như chúng ta đã giới thiệu xong một kiểu cấu trúc dữ liệu mới, đó là bảng. Tùy từng mục đích của các ứng dụng, bảng có thể có nhiều phiên bản khác nhau. Phần định nghĩa chi tiết hơn cho các phiên bản này cũng như các cách hiện thực của chúng được xem như bài tập. Phần tiếp theo đây trình bày sự giống và khác nhau giữa danh sách và bảng. Sau đó chúng ta sẽ tiếp tục làm quen với một cấu trúc dữ liệu khá đặc biệt và rất phổ biến, đó là bảng băm. Cấu trúc dữ liệu bảng băm cũng xuất phát từ ý tưởng sử dụng bảng như phần này đã giới thiệu.

### 12.4.4. So sánh giữa danh sách và bảng

Chúng ta hãy so sánh hai kiểu dữ liệu trừu tượng danh sách và bảng. Nền tảng toán học cơ bản của danh sách là **một chuỗi nối tiếp các phần tử**, còn đối với bảng, đó là **tập hợp và hàm**. Chuỗi nối tiếp có một trật tự ngầm trong đó, đó là phần tử đầu tiên, phần tử thứ hai, v.v..., còn tập hợp và hàm không có thứ tự.

Việc truy xuất thông tin trong một danh sách thường liên quan đến việc tìm kiếm, nhưng việc truy xuất thông tin trong bảng đòi hỏi những phương pháp khác, đó là các phương pháp có thể đi thẳng đến phần tử mong muốn. Thời gian cần thiết để tìm kiếm trong danh sách nói chung phụ thuộc vào n là số phần tử trong danh sách và ít nhất là bằng lg n, nhưng thời gian để truy xuất bảng thường không phụ thuộc vào số phần tử trong bảng, và thường là O(1). Vì lý do này, trong nhiều ứng dụng, việc truy xuất bảng thực sự nhanh hơn việc tìm kiếm trong một danh sách.




**Hình 12.9** – Hiện thực của bảng

Mặt khác, **duyệt là một tác vụ tự nhiên đối với một danh sách, nhưng đối với bảng thì không**. Việc di chuyển xuyên suốt một danh sách để thực hiện một tác vụ nào đó lên từng phần tử của nó nói chung là dễ dàng. Điều này đối với bảng không dễ dàng chút nào, đặc biệt trong trường hợp có yêu cầu trước về một trật tự nào đó của các phần tử được duyệt.

Cuối cùng, chúng ta cần làm rõ sự khác nhau giữa bảng và mảng. Nói chung, chúng ta dùng từ bảng như là chúng ta đã định nghĩa trong phần vừa rồi và giới hạn từ mảng chỉ với nghĩa như là một phương tiện dùng để lập trình của C++ và phần lớn các ngôn ngữ cấp cao, các mảng này thường được sử dụng để hiện thực cả hai: bảng và danh sách liên tục.

## 12.5. Bảng băm

Khi giới thiệu tổng quát về bảng cũng như cách sử dụng hàm chỉ số và mảng truy xuất, chúng ta cần nhận ra một điều rằng, thông số cho hàm chỉ số hoặc mảng truy xuất phần nào phản ánh vị trí, hay nói rõ hơn, đó là trật tự của phần tử cần truy xuất trong bảng. Chẳng hạn trật tự theo chỉ số hàng và cột trong bảng (i,j), hay trường hợp danh sách các khách hàng sử dụng điện thoại: tên của các khách hàng có thứ tự theo *alphabet*. Bảng băm mà chúng ta sẽ nghiên cứu tiếp theo mang một đặc điểm hoàn toàn khác. Việc truy xuất bảng bắt đầu từ giá trị của khóa trong phần tử dữ liệu, và thông thường khóa này không liên quan đến trật tự trong hàng hoặc cột của bảng để có thể sử dụng một hàm chỉ số đơn giản cho ra vị trí của nó trong bảng như ở phần trên đã giới thiệu.

### 12.5.1. Các bảng thưa

#### 12.5.1.1. Các hàm chỉ số

Điều chúng ta có thể làm là xây dựng sự tương ứng một – một giữa các khóa và các chỉ số mà chúng ta sử dụng để truy xuất bảng. So với các phần trước, hàm chỉ số mà chúng ta xây dựng ở đây sẽ phức tạp hơn, vì có khi chúng ta cần đến sự biến đổi của các khóa, chẳng hạn từ các chữ cái sang các số nguyên. Theo nguyên tắc, điều này luôn  có thể làm được.

Khó khăn thực sự chỉ là khi số các khóa có thể có vượt ra ngoài không gian của bảng. Lấy ví dụ, nếu các khóa là các từ có 8 ký tự, thì có thể có đến $26^8 \approx 2 \times 10^{11}$ khóa khác nhau, và đây cũng là con số lớn hơn rất nhiều dung lượng cho phép của một bộ nhớ tốc độ cao. Tuy nhiên trong thực tế, chỉ có một số không lớn các khóa này là thực sự xuất hiện. Điều đó có nghĩa là bảng chứa sẽ rất thưa thớt. Chúng ta có thể xem bảng được đánh chỉ số bằng một tập rất lớn, nhưng chỉ có một số tương đối ít vị trí là thực sự có phần tử.

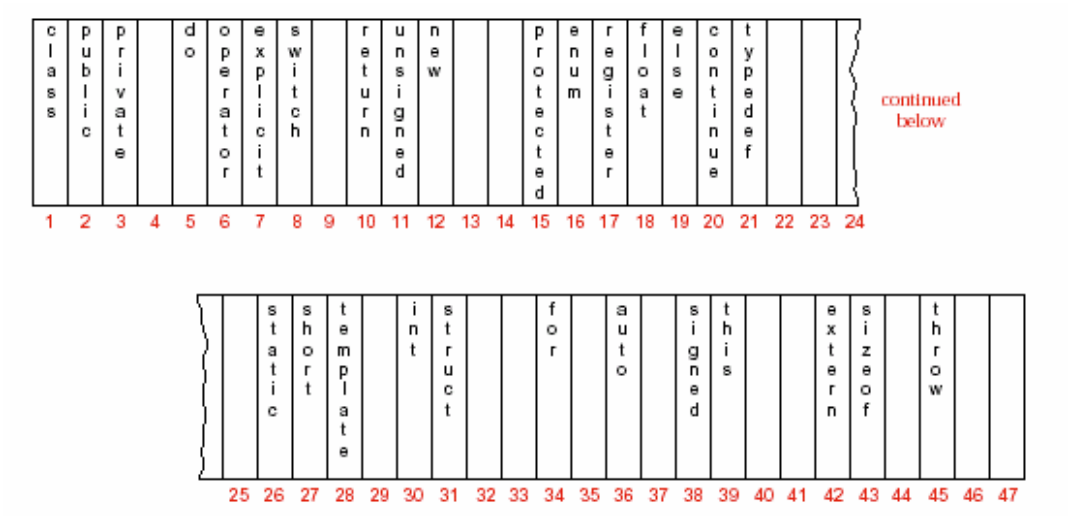
#### 12.5.1.2. Khái niệm băm

Nhằm tránh một bảng quá thưa thớt có nhiều vị trí không bao giờ được dùng đến, chúng ta làm quen với khái niệm băm. Ý tưởng của bảng băm (hình 12.10) là cho phép ánh xạ một tập các khóa khác nhau vào các vị trí trong một mảng với kích thước cho phép. Gọi kích thước mảng này là `hash_size`, mỗi khóa sẽ được ánh xạ vào một chỉ số trong khoảng [0, `hash_size`-1]. Ánh xạ này được gọi là hàm băm (*hash function*). Một cách lý tưởng, hàm này cần có cách tính đơn giản và phân bổ các khóa sao cho hai khóa khác nhau luôn vào hai vị trí khác nhau. Nhưng do kích thước mảng là giới hạn và miền trị của các khóa là rất lớn, điều này là không thể được. Chúng ta chỉ có thể hy vọng rằng một hàm băm tốt thì sẽ phân bổ được các khóa vào các chỉ số một cách khá đồng đều và tránh được hiện tượng gom tụ.

Hàm băm nói chung luôn ánh xạ một vài khóa khác nhau vào cùng một chỉ số. Nếu phần tử cần tìm đang nằm tại chỉ số được ánh xạ đến, vấn đề của chúng ta xem như đã được giải quyết; ngược lại, chúng ta cần sử dụng một phương pháp nào đó để giải quyết đụng độ. Việc đụng độ (*collision*) xảy ra khi hai phần tử cần được chứa trong cùng một vị trí của bảng.

Trên đây là ý tưởng cơ bản của việc sử dụng bảng băm. Có ba vấn đề chúng ta cần xem xét khi sử dụng phương pháp băm:

- Tìm hàm băm tốt.
- Xác định phương pháp giải quyết đụng độ.
- Xác định kích thước bảng băm.




**Hình 12.10** – Bảng băm

### 12.5.2. Lựa chọn hàm băm

Hai tiêu chí cơ bản để chọn lựa một hàm băm là:

- Hàm băm cần được tính toán dễ dàng và nhanh chóng.
- Việc phân phối các khóa có thể xuất hiện rải đều trên bảng băm.

Nếu chúng ta biết trước chính xác những khóa nào sẽ xuất hiện, thì chúng ta có thể xây dựng một hàm băm thật hiệu quả, nhưng nói chung chúng ta thường không biết trước điều này.

Chúng ta cần lưu ý rằng một hàm băm không hề có tính ngẫu nhiên. Khi tính nhiều lần cho cùng một khóa, một hàm băm phải cho cùng một trị, có như vậy thì khóa mới có thể được truy xuất sau khi được lưu trữ.

#### 12.5.2.1. Chia lấy phần dư (*modular arithmetic*)

Trước hết chúng ta hãy xem xét một trường hợp thật đơn giản. Nếu các khóa là các số nguyên, hàm băm đơn giản và phổ biến được dùng là phép chia cho hash_size để lấy phần dư, vì như vậy chúng ta sẽ có các chỉ số thuộc [0, hash_size -1]. Tuy nhiên cũng cần lưu ý những trường hợp các khóa tập trung vào một số giá trị đặc biệt nào đó. Chẳng hạn nếu hash_size = 10, mà phần lớn các khóa lại có con số ở hàng đơn vị là 0. Sự phân tán các khóa phụ thuộc nhiều vào phép chia lấy phần dư, đó chính là kích thước của bảng băm. Nếu kích thước đó là một bội số của các số nguyên nhỏ như 2 hoặc 10, thì rất nhiều khóa sẽ cho cùng chỉ số như nhau, trong khi đó có một số chỉ số rất ít được sử dụng đến. Cách chọn phép chia lấy phần dư tốt nhất thường là chia cho một số nguyên tố (nhưng không phải là luôn luôn), kết quả sẽ rải đều các khóa trong bảng băm hơn. Như vậy, thay vì chọn bảng băm kích thước 1000, chúng ta nên chọn kích thước 997 hoặc 1009; cách chọn $2^{10} = 1024$ là một cách chọn rất dở.

Thông thường, các khóa là các chuỗi ký tự. Một cách tự nhiên, người ta thường lấy một số nguyên bằng với tổng của các mã ASCII của các ký tự trong khóa làm đại diện cho nó. Hàm băm với cách viết của C chuẩn sau đây thật đơn giản và tính cũng rất nhanh:

```
index Hash(const char *Key, int hash_size)
{
     unsigned int HashVal = 0;
     while (*Key != '\0')
     {
          HashVal += *Key;
          Key++;
     }
             return HashVal % hash_size;
}
```

Tuy nhiên, nếu hash_size lớn, hàm sẽ không phân bổ các khóa tốt. Lấy ví dụ với hash_size =10007 (một số nguyên tố). Giả sử các khóa có chiều dài 8 ký tự hoặc ít hơn. Mỗi ký tự có mã ASCII <=127. Giá trị của hàm băm chỉ có thể từ 0 đến 127 x 8 = 1016.

Một cải tiến khác của hàm băm như sau: với giả thiết rằng các khóa đều có ít nhất 3 ký tự, số 27 được dùng vì đó là số ký tự trong bảng chữ cái tiếng Anh (tính cả khoảng trắng).

```
index Hash(const char *Key, int hash_size)
{
   return (Key[0] + 27*Key[1] + 27*27*Key[2]) %hash_size;
}
```

Hàm này chỉ quan tâm 3 ký tự đầu của các khóa, nhưng nếu chúng là ngẫu nhiên và hash_size là 10007 như trên, thì sự phân bổ khá đồng đều. Điều không may ở đây là các từ trong tiếng Anh không phải là một sự ghép các ký tự một cách ngẫu nhiên. Mặc dù có đến $26^3 = 17576$ khả năng ghép 3 ký tự, thực tế trong từ điển cho thấy chỉ có 2851 khả năng xảy ra. Ngay cả khi không có sự đụng độ xảy ra giữa từng cặp trong các khả năng này, thì cũng chỉ có 28% vị trí trong bảng là được sử dụng.

Thêm một cải tiến khác như sau đây:

```
index Hash(const char *Key, int hash_size)
{
     unsigned int HashVal = 0;
     while (*Key != '\0')
     {
          HashVal = (HashVal << 5 ) + *Key);
          Key++;
     }
             return HashVal % hash_size;
}
```

Hàm này quan tâm đến mọi ký tự trong khóa và nói chung có thể phân bổ các khóa đồng đều trong một bảng kích thước tương đối lớn. Trị của hàm được tính $\sum_{i=0}^{KeySize-1}$ Key[KeySize-i-1].$32^i$. Đây là đa thức với hệ số là 32 và sử dụng công thức Horner. Ví dụ, để tính $h_k = k_1 + 27k_2 + 27^2k_3$, người ta tính $h_k = ((k_3)*27 + k_2)*27 + k_1$. Việc dùng số 32 thay số 27 là vì với 32 thì không cần làm phép nhân mà chỉ đơn giản là phép dịch chuyển bit ($32 = 2^5$), và thực tế là dùng phép XOR.

Hàm trên đây chưa phải là hàm tốt nhất khi xét đến tiêu chí phân bổ đồng đều, nhưng nó cho phép việc tính toán được thực hiện rất nhanh chóng. Nếu khóa quá dài thì nó cũng lộ nhược điểm là phải tính quá lâu. Hơn nữa quá trình dịch bit sẽ làm mất đi tác dụng của các ký tự đã được xét trước. Thực tế khắc phục điều này bằng cách không sử dụng tất cả các ký tự có trong khóa.

#### 12.5.2.2. Cắt xén (*truncation*)

Phương pháp cắt xén bỏ qua một phần của khóa, phần còn lại được xem như chỉ số (các dữ liệu không phải số thì lấy theo bảng mã của chúng). Ví dụ, nếu khóa là một số nguyên 8 ký số và bảng băm có 1000 vị trí, thì việc lấy từ vị trí thứ nhất, thứ hai và thứ năm kể từ phải sang sẽ là hàm băm. Có nghĩa là khóa 21**2**96**87**6 có chỉ số là 976. Cắt xén là một phương pháp cực nhanh, nhưng nó thường không phân phối các khóa đều khắp bảng băm.

#### 12.5.2.3. Xáo trộn (*folding*)

Ý tưởng xáo trộn (*folding*) dưới đây giúp cho các bộ phận của khóa đều có thể tham gia vào việc xác định kết quả cuối cùng của hàm băm. Từ băm ở đây có nghĩa là kết quả sinh ra có phần giống với khóa ban đầu. Ngoài ra, sự xáo trộn cho phép chúng ta hy vọng rằng mọi khuôn mẫu hoặc sự lặp lại có thể xuất hiện trong các khóa (hậu quả của tính thiếu ngẫu nhiên của dữ liệu trong thực tế) sẽ bị triệt tiêu. Có như vậy thì các kết quả mới được phân phối theo cùng một quy luật như nhau mà không có sự trùng lặp của từng nhóm kết quả và chúng ta tránh được hiện tượng gom tụ. Ở đây chúng ta thấy rằng thuật ngữ "băm" mang tính mô tả rõ nhất. Tuy nhiên trong một số tài liệu khác người ta dùng các các từ mang tính kỹ thuật hơn như "bộ nhớ phân tán" (*scatter-storage*) hoặc "phép biến đổi khóa" (*key-transformation*).

Phương pháp xáo trộn chia khóa làm nhiều phần và kết nối các phần này lại theo một cách thích hợp (thường sử dụng phép cộng hoặc phép nhân). Lấy ví dụ, một số nguyên 8 ký số có thể được chia làm 3 nhóm gồm 3, 3, và 2 ký số, các nhóm này được cộng lại với nhau, sau đó có thể được cắt xén bớt nếu cần thiết để cho ra các chỉ số phù hợp kích thước bảng băm. Khóa 21296876 sẽ được băm thành 212 + 968 + 76 = 1256, cắt ngắn còn 256. Do mọi dữ liệu trong khóa đều có ảnh hưởng đến kết quả hàm băm nên phương pháp này làm cho các khóa rải đều trên bảng băm hơn là phương pháp cắt xén nêu trên.

Tóm lại, chúng ta đã xem xét một số phương pháp mà chúng ta có thể kết hợp lại theo nhiều cách khác nhau để xây dựng hàm băm. Lấy phần dư thường là một cách tốt để kết thúc việc tính toán của một hàm băm, do nó vừa có thể đạt được sự rải đều các khóa trong bảng băm vừa bảo đảm kết quả nhận được luôn nằm trong miền các chỉ số cho phép.

### 12.5.3. Phác thảo giải thuật cho các thao tác dữ liệu trong bảng băm

Trước hết, chúng ta cần khai báo một mảng để chứa bảng băm. Sau đó, các vị trí trong mảng cần được khởi tạo là trống. Giá trị khởi tạo phụ thuộc vào ứng dụng, thông thường chúng ta cho các vị trí trống này chứa một giá trị đặc biệt nào đó. Chẳng hạn, với các khóa là các chữ cái, một trị chứa toàn ký tự trống có thể biểu diễn một vị trí trống.

Để thêm một phần tử vào bảng băm, cần tính hàm băm cho khóa của nó. Nếu vị trí tìm thấy còn trống, phần tử sẽ được thêm vào; nếu đã có phần tử tại vị trí này và khóa của nó trùng với khóa của phần tử cần thêm thì việc thêm sẽ không được thực hiện; trường hợp cuối cùng, nếu tại vị trí tìm thấy đã có một phần tử nhưng của một khóa khác, chúng ta sẽ áp dụng một phương pháp giải quyết đụng độ nào đó để tìm đến một vị trí khác cho việc thêm phần tử mới của chúng ta.

Việc truy xuất một phần tử với khóa cho trước được làm tương tự. Trước tiên, hàm băm được tính cho khóa cho trước. Nếu phần tử cần tìm đang nằm tại vị trí được chỉ bởi hàm băm, thì việc truy xuất sẽ được thực hiện thành công; ngược lại, trong khi mà vị trí đang xét không trống và mọi vị trí chưa được xét đến, cần tiến hành các bước tương tự như các bước đã được sử dụng khi giải quyết đụng độ trong quá trình thêm vào. Nếu trong khi tìm kiếm gặp một vị trí trống, hoặc khi mọi vị trí đã được xét đến, thì có thể kết luận việc tìm kiếm thất bại: không có phần tử với khóa cần tìm trong bảng băm.

### 12.5.4. Ví dụ trong C++

Như một ví dụ đơn giản, chúng ta sẽ viết một hàm băm trong C++ để chuyển đổi một khóa gồm 8 ký tự chữ cái sang một số nguyên trong miền

```
0 . . hash_size – 1.
```

Chúng ta có một lớp **Key** với các phương thức như sau:

```
class Key: public String{
public:
    char key_letter(int position) const;
    void make_blank();
    //  Các constructor và các phương thức khác.
};
```

Để giảm công sức lập trình khi hiện thực lớp, chúng ta chọn cách thừa kế các phương thức của lớp **String** trong chương 5. Chúng ta sẽ đỡ phải viết lại các tác vụ so sánh. Phương thức **key_letter(int position)** trả về ký tự tại vị trí **position** trong khóa, hoặc trả về khoảng trắng nếu khóa có chiều dài nhỏ hơn n. Phương thức **make_blank** tạo một khóa trống.

```
int hash(const Key &target)
/*
post: Hàm băm trên target trả về trị trong miền 0 .. hash_size-1.
uses: Các phương thức của lớp Key.
*/
{
   int value = 0;
   for (int position = 0; position < 8; position++)
      value = 4 * value + target.key_letter(position);
   return value % hash_size;
}
```

Hàm băm trên đơn giản chỉ cộng dồn các mã của mỗi ký tự trong khóa sau khi đã nhân với 4. Chúng ta không thể lý giải được rằng phương pháp này là tốt hơn (hoặc xấu hơn) một vài phương pháp khác. Chúng ta cũng có thể lấy mã của ký tự trừ đi một số nào đó, rồi nhân từng cặp với nhau, hoặc bỏ qua một vài ký tự nào

đó. Đôi khi một ứng dụng sẽ cho thấy một hàm băm này là tốt hơn một hàm băm khác, đôi khi cần phải có sự khảo sát bằng thực nghiệm mới chỉ ra được hàm nào là tốt hơn.

### 12.5.5. Giải quyết đụng độ bằng phương pháp địa chỉ mở

Có hai nhóm phương pháp giải quyết đụng độ: nhóm phương pháp địa chỉ mở và nhóm phương pháp nối kết. Nhóm phương pháp địa chỉ mở chỉ sử dụng các mảng cấp phát tĩnh. Nhóm phương pháp nối kết có sử dụng những vùng nhớ cấp phát động được quản lý bởi các con trỏ nối kết.

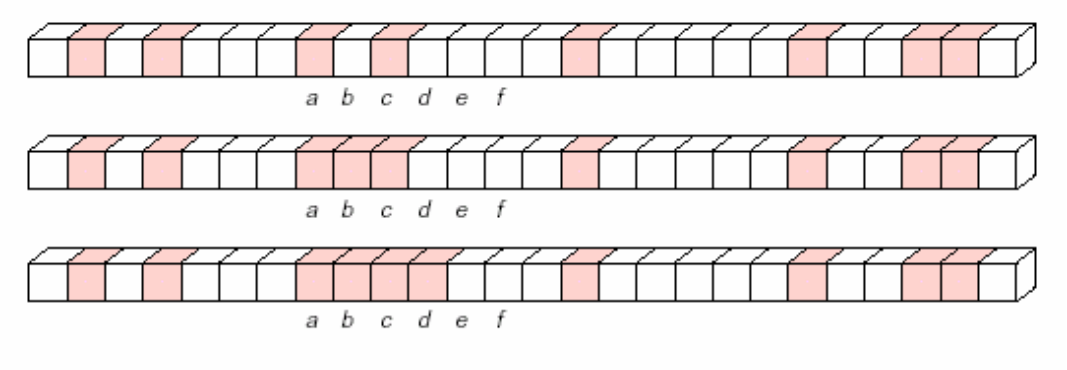
Dưới đây là các phương pháp dùng địa chỉ mở.

#### 12.5.5.1. Thử tuyến tính

Phương pháp đơn giản nhất để giải quyết đụng độ là bắt đầu từ vị trí trả về từ hàm băm có xảy ra đụng độ, việc tìm kiếm sẽ tiếp tục một cách tuần tự ở các vị trí kế trong bảng cho đến khi gặp khóa mong muốn hoặc một vị trí trống. Phương pháp này được gọi là phương pháp thử tuyến tính (*linear probing*). Bảng được xem như một mảng vòng, khi việc tìm kiếm đạt đến vị trí cuối của bảng thì sẽ quay về vị trí đầu của bảng.

**Hiện tượng gom tụ**

Nhược điểm chính của phương pháp thử tuyến tính là khi có khoảng một nửa số vị trí trong bảng đã chứa dữ liệu, khuynh hướng gom tụ sẽ xuất hiện; nghĩa là, các phần tử sẽ nằm trong các chuỗi liên tục các vị trí, giữa các chuỗi này là những lỗ hổng. Việc tìm kiếm tuần tự một vị trí trống trong bảng sẽ ngày càng lâu hơn. Chúng ta hãy xem ví dụ ở hình 12.11. Giả sử mảng có n vị trí thì xác suất mà hàm băm chọn một vị trí nào đó là 1/n. Ban đầu việc phân phối được thực hiện khá đều trong bảng (phần trên của hình). Giả sử cần thêm dữ liệu mới mà hàm băm trả về vị trí b thì dữ liệu được thêm vào tại đây, nhưng nếu hàm băm trả về vị trí a mà vị trí này đã có dữ liệu, việc thêm vào sẽ được thực hiện tại b. Như vậy xác suất để vị trí b nhận dữ liệu là 2/n. Tại bước tiếp theo, khi dữ liệu cần thêm vào một trong các vị trí a, b, c, hoặc d thì chỗ trống thực sự để thêm vào chỉ là d, như vậy xác suất dữ liệu thêm vào d là 4/n. Sau đó, xác suất dữ liệu thêm vào vị trí e lại là 5/n. Và cứ như thế, khi dữ liệu càng được thêm vào nhiều thì chuỗi liên tục các vị trí đã có dữ liệu bắt đầu từ a ngày càng dài ra. Như vậy cách thực hiện của bảng băm bắt đầu suy thoái dần tới sự tìm kiếm tuần tự.




**Hình 12.11** – Hiện tượng gom tụ trong bảng băm.

Hiện tượng gom tụ chính là nguyên nhân của tính thiếu ổn định. Nếu một số ít các khóa ngẫu nhiên nằm kế nhau, thì sau đó các khóa khác bỗng trở nên kết dính với chúng, có nghĩa là vị trí chứa chúng phụ thuộc lẫn nhau, và sự phân phối dần dần trở nên thiếu cân bằng.

### 12.5.5.2. Hàm gia tăng

Để tránh hiện tượng gom tụ, chúng ta phải sử dụng phương pháp phức tạp hơn để chọn ra chuỗi các vị trí cần xem xét đến để thêm một dữ liệu mới nào đó khi có xảy ra đụng độ. Có nhiều cách để thực hiện. Ý tưởng chung là sử dụng **một hoặc một vài hàm gia tăng để xác định khoảng cách từ vị trí vừa đụng độ đến một vị trí mới**. Cần lưu ý rằng kết quả của hàm gia tăng không được phép trả về trị 0.

Hàm gia tăng có thể phụ thuộc vào khóa, hoặc vào số lần đã thử, sao cho có thể tránh được hiện tượng gom tụ.

Trường hợp thứ nhất, **khi hàm gia tăng phụ thuộc vào khóa**, chúng ta có khái niệm băm lại. Đó là cách sử dụng một hàm băm thứ hai. Kết quả của hàm băm này là số vị trí cần di chuyển kể từ vị trí đã bị đụng độ trước đó. Nếu vị trí này lại đụng độ, chúng ta lại dùng một hàm băm khác nữa để tìm đến vị trí thứ ba, và cứ thế. Cũng có khi từ kết quả tính của hàm băm thứ hai người ta dùng luôn số này để di chuyển giữa hai lần thử kế tiếp.

Trong trường hợp thứ hai, **hàm gia tăng phụ thuộc vào số lần đã thử**, có thể kể ra đây phương pháp thử bậc hai.

#### Thử bậc hai

Nếu có sự đụng độ tại địa chỉ băm được h, phương pháp thử bậc hai (*quadratic probing*) thử các vị trí kế tiếp là `h+1, h+4, h+9`, ... trong bảng, có nghĩa là các vị trí $h + i^2$, với i là lần thử. Nói cách khác, hàm gia tăng là $i^2$.

Phương pháp thử tuyến tính đã nêu trên cũng có thể xem như một trường hợp sử dụng hàm gia tăng là i.

Phương pháp thử bậc hai thực sự có làm giảm hiện tượng gom tụ, nhưng thực tế thường nó không thể thử hết mọi vị trí trong bảng. Đối với một vài giá trị của hash_size, hàm $i^2$ sẽ thử một số tương đối ít các vị trí trong bảng. Lấy ví dụ, khi hash_size là một bội số lớn của 2, chỉ khoảng một phần sáu số các vị trí trong bảng băm là được thử. Khi hash_size là một số nguyên tố, thử bậc hai sẽ đạt được một nửa số vị trí trong bảng băm.

Để chứng minh điều trên, giả sử rằng hash_size là một số nguyên tố. Giả sử chúng ta cùng đạt một vị trí khi thử lần thứ i và lần thứ i + j với j là một số nguyên > 0. Giả sử j là một số nguyên nhỏ nhất theo điều kiện trên. Giá trị tính được bởi hàm băm lần thứ i và lần thứ i + j khác nhau bởi một bội số của hash_size. Nói cách khác,

$$h + i^2 \equiv h + (i + j)^2 \pmod{hash\_size}$$

Biến đổi biểu thức trên ta có:

$$j^2 + 2ij = j(j + 2i) \equiv 0 \pmod{hash\_size}.$$

Biểu thức này có nghĩa là j(j + 2i) chia hết cho hash_size. Một tích chia hết cho một số nguyên tố chỉ khi một trong các thừa số của tích đó chia hết cho số nguyên tố đó. Vậy hoặc j chia hết cho hash_size, hoặc j+2i chia hết cho hash_size. Trong trường hợp thứ nhất, chúng ta đã phải thử j=hash_size lần trước khi gặp lại vị trí đã thử với i (chúng ta nhớ rằng j là số nhỏ nhất theo giả thiết). Tuy nhiên trường hợp thứ hai sẽ xảy ra sớm hơn, khi j=hash_size –2i, hoặc khi biểu thức tăng thêm hash_size nếu biểu thức này âm. Do đó tổng số vị trí khác nhau được thử sẽ là

$$(hash\_size + 1) / 2.$$

Khi đã thử với số lần như trên chúng ta có thể xem như bảng đã đầy.

Chú ý rằng phương pháp thử bậc hai có thể được thực hiện mà không cần phép nhân: Sau lần thử thứ nhất tại vị trí h, biến tăng được gán là 1. Tại mỗi lần thử thành công, biến tăng sẽ tăng thêm 2 sau khi nó đã được thêm vào vị trí trước đó.

Do $1 + 3 + 5 + \ldots + (2i - 1) = i^2$

đối với mọi i≥1 , lần thử i sẽ tìm tại vị trí h +1 +. . . +(2i-1)=h+$i^2$, theo như mong muốn.

### 12.5.5.3. Thử ngẫu nhiên

Phương pháp cuối cùng là sử dụng số ngẫu nhiên được sinh ra để làm biến gia tăng. Chúng ta chỉ được dùng một bộ sinh số ngẫu nhiên để từ một số bắt đầu cho trước nó luôn luôn sinh ra cùng một chuỗi các số ngẫu nhiên kế tiếp. Đây là một phương pháp rất tốt để tránh hiện tượng gom tụ, nhưng nó có thể chậm hơn các phương pháp khác.

### 12.5.5.4. Giải thuật C++

Để kết thúc việc nghiên cứu về phương pháp địa chỉ mở, chúng ta có một ví dụ C++ với các khóa là các ký tự chữ cái. Chúng ta giả sử rằng lớp Key và lớp Record có các đặc tính mà chúng ta vừa sử dụng trong hai phần cuối. Lớp Key có phương thức **key_letter(int position)** để trả về ký tự tại **position**, lớp Record có phương thức để lấy một khóa của một phần tử.

Bảng băm của chúng ta sẽ có khai báo như sau:

```
const int hash_size = 997;   //  Số nguyên tố
class Hash_table {
public:
   Hash_table();
   void clear();
   Error_code insert(const Record &new_entry);
   Error_code retrieve(const Key &target, Record &found) const;
private:
   Record table[hash_size];
};
```

Bảng băm sẽ được khởi tạo sao cho tất cả các phần tử trong mảng đều chứa khóa đặc biệt gồm 8 khoảng trắng. Đây là nhiệm vụ của *constructor*:

```
Hash_table:: Hash_table();
// post: Bảng băm được tạo và được khởi tạo là rỗng.
```

Phương thức clear cần để loại tất cả các dữ liệu hiện có trong bảng băm:

```
void Hash_table::clear();
// post: Bảng băm đã được dọn dẹp và trở thành bảng băm rỗng.
```

Mặc dù chúng ta đã bắt đầu đặc tả các phương thức của bảng băm, chúng ta sẽ không tiếp tục phát triển thành một gói tổng quát và đầy đủ. Do việc chọn một hàm băm tốt phụ thuộc nhiều vào loại của khóa sẽ được sử dụng, các phương thức

của bảng băm thường phụ thuộc mạnh mẽ vào từng ứng dụng riêng, một gói tổng quát cho một bảng băm là không có lợi.

Để minh họa cách viết các hàm tiếp theo, chúng ta sẽ sử dụng phương pháp thử bậc hai để giải quyết đụng độ. Chúng ta đã chứng minh rằng số lần thử tối đa có thể thực hiện theo phương pháp này là `(hash_size+1) / 2`, nên sẽ dùng biến đếm **`probe_count`** để kiểm tra giới hạn này.

Với những quy ước như trên, chúng ta có phương thức thêm một phần tử **`new_entry`** vào bảng băm như sau:

```
Error_code Hash_table::insert(const Record &new_entry)
/*
post: Nếu bảng băm đầy, phương thức trả về overflow.
      Nếu bảng băm đã chứa phần tử có khóa trùng khóa trong new_entry thì ph7ơng thức trả
      về duplicate_error. Ngược lại, phần tử new_entry đươc thêm vào bảng băm và
      phương thức trả về success.
uses: Các phương thức của các lớp Key và Record, hàm hash.
*/
{
   Error_code result = success;
   int probe_count,     //  Đếm số lần thử để phát hiện bảng đầy.
       increment,       //  Số gia tăng bở phép thử bậc hai.
       probe;           //  Vị trí thử hiện thời.
   Key null;            //  Giá trị NULL của khóa dùng cho phép so sánh.
   null.make_blank();

   probe = hash(new_entry);
   probe_count = 0;
   increment = 1;

   while (table[probe] != null                 // Vị trí thử có trống hay không?
     && table[probe] != new_entry              // Khóa đã có trong bảng băm?
     && probe_count < (hash_size + 1) / 2) {// Bảng đầy hay chưa?
      probe_count++;
      probe = (probe + increment) % hash_size;
      increment += 2;           // Tính lại độ dời cho lần thử kế tiếp.
   }
   if (table[probe]==null) table[probe]= new_entry;
   else if (table[probe] == new_entry) result = duplicate_error;
   else result =  overflow;
   return result;
}
```

Phương thức để truy xuất một phần tử với một khóa cho trước có dạng tương tự, chúng ta dành lại như bài tập. Đặc tả của nó như sau:

```
Error_code Hash_table:: retrieve(const Key &target, Record &found) const;
//post: Nếu một phần tử trong bảng băm có khóa giống target, thì found sẽ được gán trị của
        phần tử đó, phương thức trả về success. Ngược lại, trả về not_present.
```

#### 12.5.5.5. Loại bỏ một phần tử

Cho đến bây giờ, chúng ta vẫn chưa nói gì đến việc loại một phần tử trong bảng băm. Thoạt nhìn, dường như đó là một việc dễ dàng, chỉ cần gán lại cho vị trí cần loại một trị đặc biệt của khóa để chỉ ra rằng đó là vị trí trống. Tuy nhiên cách này không thể áp dụng được. Lý do là một vị trí trống được xem như một dấu hiệu để kết thúc quá trình tìm kiếm một khóa. Giả sử như trước khi loại, đã xảy ra một hoặc hai lần đụng độ và một phần tử nào đó lẽ ra phải được thêm vào tại vị trí đang xét lại phải dời đến một vị trí đâu đó trong bảng. Nếu bây giờ chúng ta cần truy xuất đến phần tử này thì chỗ trống mới được tạo ra sẽ kết thúc việc tìm kiếm, và chúng ta không thể tìm thấy nó, mặc dù nó vẫn tồn tại trong bảng.

Một phương pháp để ngăn ngừa tình huống trên là sử dụng một khóa đặc biệt để đặt vào các vị trí cần loại đi phần tử. Khóa đặc biệt này chỉ ra rằng đó là một vị trí trống có thể thêm phần tử mới vào, nhưng nó không được dùng để kết thúc quá trình tìm kiếm một khóa nào khác trong bảng. Tuy nhiên cách sử dụng khóa đặc biệt này sẽ làm cho giải thuật phức tạp hơn và chậm hơn. Còn một số phương pháp khác có thể áp dụng trong việc loại bỏ một phần tử khỏi bảng băm. Tuy nhiên chúng ta cần nhớ rằng phương pháp loại bỏ nào cũng phải tương thích với chiến lược thêm và tìm kiếm phần tử, để hai tác vụ này luôn hoạt động một cách chính xác. Danh sách liên kết trong mảng liên tục trong phần 4.5 cũng thường được sử dụng làm bảng băm, và cũng thuộc nhóm phương pháp địa chỉ mở để giải quyết đụng độ. Các phần tử có cùng giá trị hàm băm sẽ được nối kết trong cùng một danh sách liên kết. Và trong bảng băm có nhiều danh sách liên kết như vậy.

### 12.5.6. Giải quyết đụng độ bằng phương pháp nối kết

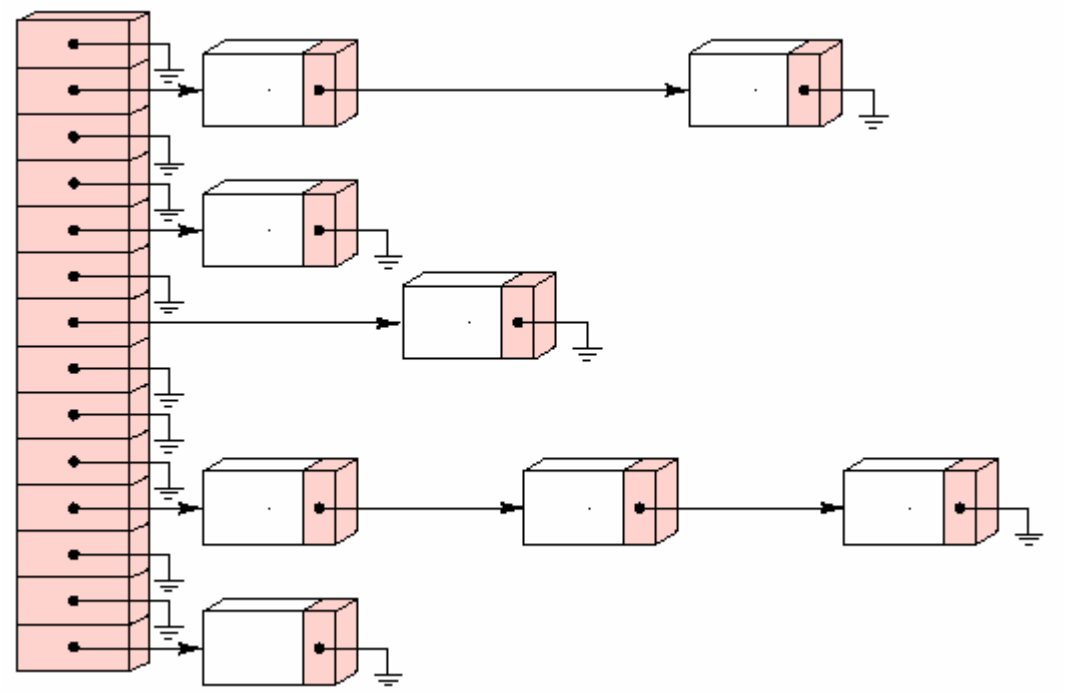
Cho đến bây giờ chúng ta vẫn cứ ngầm hiểu rằng chúng ta chỉ sử dụng vùng nhớ liên tục để chứa bảng băm. Thật vậy, vùng nhớ liên tục là cách chọn tự nhiên để hiện thực bảng băm, do chúng ta cần truy xuất một vị trí ngẫu nhiên trong bảng một cách nhanh chóng, mà vùng nhớ liên kết thì không hỗ trợ việc truy xuất ngẫu nhiên. Tuy nhiên, điều đó không có nghĩa là vùng nhớ liên kết không thể được dùng để chứa các phần tử. Chúng ta có thể dùng bảng băm là một mảng các danh sách liên kết. Chúng ta hãy xem hình 12.12.

Người ta thường quen gọi các danh sách liên kết từ bảng băm là các chuỗi mắc xích nối kết (*chain*) nên phương pháp giải quyết đụng độ này còn được gọi là phương pháp nối kết (*chaining*).

#### 12.5.6.1. Ưu điểm của phương pháp nối kết

Ưu điểm thứ nhất và cũng là ưu điểm quan trọng nhất của phương pháp này là nó có thể tiết kiệm vùng nhớ khi bản thân các phần tử khá lớn. Do bảng băm là một mảng, chúng ta cần khai báo trước một số lượng phần tử  khá lớn để tránh

hiện tượng tràn. Nếu để các phần tử nằm trong bảng băm thì khi chưa có nhiều dữ liệu, có quá nhiều vị trí để trống, trong khi chương trình của chúng ta có thể cần nhiều vùng nhớ cho những biến khác nữa. Ngược lại, nếu bảng băm chỉ chứa các con trỏ mà mỗi con trỏ chỉ cần chiếm số byte bằng số byte của một từ thì kích thước bảng băm giảm đáng kể.



**Hình 12.12** – Bảng băm nối kết

Ưu điểm chính thứ hai của việc lưu các danh sách liên kết kèm với bảng băm là nó cho phép xử lý đụng độ một cách đơn giản và hiệu quả. Với một hàm băm tốt, chỉ có một số ít khóa là trùng địa chỉ băm, và như vậy các danh sách liên kết đều ngắn và các khóa đều có thể được tìm kiếm nhanh chóng. Gom tụ không còn là vấn đề phải quan tâm bởi vì các khóa có các địa chỉ băm khác nhau luôn nằm trong các danh sách khác nhau.

Ưu điểm thứ ba là kích thước bảng băm không nhất thiết phải lớn hơn số phần tử có thể có, chúng ta không còn phải lo vấn đề tràn. Khi số phần tử nhiều hơn kích thước bảng băm thì chỉ có nghĩa rằng, chắc chắn là có một vài danh sách liên kết nào đó có nhiều hơn một phần tử. Ngay cả khi số phần tử nhiều gấp vài lần kích thước bảng băm thì chiều dài trung bình của mỗi danh sách liên kết vẫn nhỏ và việc tìm kiếm tuần tự trên từng danh sách vẫn còn hiệu quả.

Cuối cùng, việc loại một phần tử trở thành một công việc dễ dàng và nhanh chóng đối với bảng băm theo phương pháp nối kết. Việc loại bỏ này được tiến

hành hoàn toàn giống với việc loại một phần tử ra khỏi một danh sách liên kết đơn.

#### 12.5.6.2. Nhược điểm của phương pháp nối kết

Các ưu điểm của bảng băm theo phương pháp nối kết thực sự là rất có lợi. Chúng ta nên tin rằng phương pháp nối kết luôn là phương pháp tốt hơn so với phương pháp địa chỉ mở. Tuy vậy, chúng ta hãy xét đến một nhược điểm quan trọng của nó: mọi mối liên kết đều chiếm vùng nhớ. Nếu phần tử có kích thước lớn thì kích thước của các con trỏ sẽ không đáng kể, ngược lại sẽ là điều không hay.

Giả sử rằng mỗi mối liên kết chiếm một từ (một *word* chiếm 2 hoặc 4 *bytes*) và mỗi phần tử cũng chỉ chiếm một từ. Những ứng dụng như vậy cũng tương đối phổ biến, trong đó chúng ta sử dụng bảng băm chỉ để trả lời một vài câu hỏi yes-no về các khóa. Giả sử chúng ta dùng bảng băm theo phương pháp nối kết và khai báo một mảng nhỏ để chứa bảng băm với n là số phần tử của mảng mà cũng là số phần tử sẽ có. Chúng ta sẽ phải sử dụng 3n từ trong bộ nhớ: n cho bảng băm, n cho các khóa, và n cho các mối liên kết để tìm đến phần tử kế trong các danh sách liên kết. Do bảng băm gần như đầy nên đụng độ sẽ xảy ra nhiều hơn, một số danh sách liên kết sẽ có vài phần tử. Việc tìm kiếm sẽ chậm. Mặt khác, giả sử như chúng ta dùng phương pháp địa chỉ mở. Cũng với 3n từ của bộ nhớ, nếu chúng ta chứa trực tiếp các phần tử trong bảng băm có kích thước 3n này thì chỉ có một phần ba bảng là có dữ liệu, như vậy số đụng độ cũng sẽ tương đối ít và việc tìm một phần tử sẽ nhanh hơn rất nhiều.

#### 12.5.6.3. Các giải thuật trong C++

Bảng băm theo phương pháp nối kết trong C++ có định nghĩa đơn giản như sau:

```
class Hash_table {
public:
   //  Specify methods here.
private:
   List<Record> table[hash_size];
};
```

Ở đây lớp `List` có thể là bất kỳ một hiện thực liên kết tổng quát nào của một danh sách đã học trong chương 4**.** Để được nhất quán, các phương thức của bảng băm nối kết sẽ chứa mọi phương thức của hiện thực bảng băm trước kia của chúng ta. *Constructor* của bảng băm chỉ đơn giản gọi các *constructor* cho từng danh sách của mảng. Trong khi đó việc dọn dẹp xóa sạch các phần tử trong bảng băm nối kết lại là một việc hoàn toàn khác, chúng ta cần dọn dẹp từng danh sách tại mỗi vị trí trong bảng băm. Việc này có thể được thực hiện nhờ phương thức **`clear()`** của `List`.

Chúng ta còn có thể sử dụng các phương thức trong đóng gói `List` để truy xuất bảng băm. Bản thân hàm băm không khác so với bảng băm theo phương pháp địa chỉ mở. Để truy xuất dữ liệu, chúng ta có thể đơn giản sử dụng phiên bản liên kết của hàm **`sequential_search`** trong phần 7.2. Cốt lõi của phương thức **`Hash_table::retrieve`** là

```
sequential_search( table[hash(target)], target, position);
```

Chi tiết của việc chuyển đổi hàm này thành một hàm đầy đủ được xem như bài tập. Tương tự, cốt lõi của việc thêm dữ liệu vào bảng băm là

```
table[hash(new_entry)].insert( 0, new_entry );
```

Ở đây chúng ta chọn cách khi thêm phần tử mới vào thì nó sẽ được đứng tại vị trí đầu của danh sách liên kết, do đây là cách dễ nhất. Như chúng ta đã thấy, cả hai việc thêm vào và truy xuất phần tử này đều đơn giản hơn là phương pháp địa chỉ mở, do việc giải quyết đụng độ không còn là vấn đề nữa.

Việc loại phần tử ra khỏi bảng băm nối kết cũng đơn giản hơn rất nhiều so với bảng băm địa chỉ mở. Để loại một phần tử với một khóa cho trước, chúng ta chỉ cần tìm tuần tự phần tử đó trong danh sách liên kết có chứa nó và loại nó ra khỏi danh sách này. Đặc tả của phương thức loại bỏ như sau:

```
Error_code Hash_table::remove(const Key &target, Record &x);
```
*post*: Nếu bảng có chứa phần tử có khóa bằng `target`, thì phần tử này được chép vào `x` và được loại khỏi bảng băm, phương thức trả về `success`. Ngược lại phương thức trả về `not_present`.

Hiện thực của phương thức này cũng được dành lại như bài tập.

## 12.6. Phân tích bảng băm

### 12.6.1. Điều ngạc nhiên về ngày sinh

Khả năng xảy ra đụng độ trong việc băm có liên quan đến một chuyện vui khá nổi tiếng trong toán học: nếu chọn một cách ngẫu nhiên từng người để đưa vào một căn phòng thì sẽ được bao nhiêu người trước khi có thể xảy ra việc hai người trong số đó có cùng một ngày sinh. Do mỗi năm có `365` ngày, nhiều người đoán rằng câu trả lời phải lên đến con số hàng trăm, nhưng lời giải thực ra là chỉ có `23` người.

Chúng ta có thể định ra xác suất cho câu hỏi này bằng cách trả lời theo hướng ngược lại: Với m người được chọn một cách ngẫu nhiên để đưa vào phòng, xác suất để hai người có cùng ngày sinh là bao nhiêu? Chúng ta hãy bắt đầu với bất kỳ

người nào và đáng dấu loại ngày sinh của họ trên lịch. Xác suất để người thứ hai có ngày sinh khác với người đã chọn là `364/365`. Tiếp tục đánh dấu loại ngày sinh của người này chúng ta có xác suất để người thứ ba có ngày sinh khác sẽ là `363/365`. Tiếp tục tiến hành theo cách này, chúng ta sẽ thấy nếu m-1 người đầu tiên có các ngày sinh khác nhau, thì xác suất người thứ m có ngày sinh khác nữa là `(365 – m + 1) / 365`. Do các ngày sinh của những người khác nhau là độc lập nhau, các xác suất này sẽ được nhân với nhau, và chúng ta có được xác suất để m người có các ngày sinh không trùng nhau là

$$\frac{364}{365} \text{ x } \frac{363}{365} \text{ x } \frac{362}{365} \text{ x } \ldots \text{ x } \frac{365\text{-m+1}}{365}$$

Biểu thức này sẽ nhỏ hơn `0.5` khi $m \geq 23$.

Khi xét đến bảng băm, điều đáng ngạc nhiên về các ngày sinh trên đây cho chúng ta biết rằng với bất kỳ một kích thước nào thì hầu như sự đụng độ cũng chắc chắn sẽ xảy ra. Vì thế, cách tiếp cận của chúng ta không nên chỉ dừng lại ở việc làm giảm số lần đụng độ, mà còn phải xử lý chúng càng hiệu quả càng tốt.

### 12.6.2. Đếm số lần thử

Cũng như những phương pháp truy xuất thông tin khác, chúng ta muốn biết số lần so sánh trung bình của các khóa trong cả hai trường hợp tìm kiếm thành công và không thành công đối với một khóa cho trước. Chúng ta sẽ dùng từ thử (*probe*) cho việc xem xét một phần tử và so sánh khóa của nó với khóa cần tìm.

Số lần thử cần thiết phụ thuộc vào mức độ đầy của bảng. Do đó (cũng như các phương pháp tìm kiếm), chúng ta gọi n là số phần tử trong một bảng và t (cũng là `hash_size`) là số vị trí trong mảng chứa bảng băm. **Hệ số tải** (*load factor*) của bảng sẽ là $\lambda = n/t$; $\lambda = 0$ có nghĩa là bảng rỗng; $\lambda = 0.5$ là bảng chứa một nửa số phần tử. Đối với bảng địa chỉ mở, $\lambda$ không bao giờ có thể vượt quá 1, nhưng đối với bảng nối kết sẽ không có giới hạn cho $\lambda$. Chúng ta sẽ xem xét riêng từng bảng trên.

### 12.6.3. Phân tích phương pháp nối kết

Với một bảng nối kết chúng ta đi trực tiếp đến một trong các danh sách liên kết trước khi thực hiện bất kỳ một phép thử nào. Giả sử như danh sách có chứa khóa cần tìm có k phần tử. Chú ý rằng k có thể bằng 0.

Nếu việc tìm kiếm không thành công, thì khóa cần tìm sẽ phải được so sánh với tất cả k khóa của k phần tử tương ứng. Do các phần tử được phân phối một cách như nhau trên tất cả t danh sách (xác suất xuất hiện bằng nhau trên mọi

danh sách), số phần tử được mong đợi trong danh sách đang được tìm kiếm là $\lambda = n/t$. Do đó số lần thử trung bình của một lần tìm kiếm không thành công là $\lambda$.

Bây giờ chúng ta hãy giả sử là việc tìm kiếm sẽ thành sông. Từ phân tích của việc tìm tuần tự, chúng ta đã biết rằng số lần so sánh trung bình là $\frac{1}{2}(k+1)$, với $k$ là chiều dài của danh sách chứa phần tử cần tìm. Nhưng chiều dài mong đợi của danh sách này không lớn hơn $\lambda$, và chúng ta biết trước là nó chứa ít nhất một phần tử (phần tử cần tìm). Ngoại trừ phần tử cần tìm, $n-1$ phần tử còn lại được phân phối như nhau trên tất cả $t$ danh sách; vậy số phần tử mong đợi trên danh sách có chứa phần tử cần tìm là $1+(n-1)/t$. Không kể các bảng có kích thước nhỏ, chúng ta lấy xấp xỉ $(n-1)/t$ bằng $n/t=\lambda$. Vậy số lần thử trung bình cho một lần tìm kiếm thành công gần với

$$\frac{1}{2}(k+1) \approx \frac{1}{2}(1 + \lambda + 1) = 1 + \frac{1}{2}\lambda.$$

Tóm lại, việc truy xuất một bảng băm nối kết có hệ số tải $\lambda$ trung bình cần đến $1 + \frac{1}{2}\lambda$ lần thử cho một lần tìm kiếm thành công và $\lambda$ lần thử cho một lần tìm kiếm không thành công.

### 12.6.4. Phân tích phương pháp địa chỉ mở

Để phân tích số lần thử trong bảng băm địa chỉ mở, trước hết chúng ta bỏ qua vấn đề gom tụ với giả thiết rằng không chỉ lần thử đầu tiên là ngẫu nhiên mà ngay cả sau khi xảy ra đụng độ, lần thử kế tiếp cũng ngẫu nhiên trên khắp các vị trí còn lại của bảng. Nói cách khác, giả sử rằng bảng băm có kích thước rất lớn sao cho mọi lần thử có thể được xem là độc lập nhau. Kết quả tính được như sau:

Việc truy xuất từ bảng băm địa chỉ mở, với phép thử ngẫu nhiên và hệ số tải $\lambda$, có số lần thử trung bình xấp xỉ bằng

$$\frac{1}{\lambda} \ln \frac{1}{1-\lambda}$$

trong trường hợp tìm kiếm thành công và $1/(1-\lambda)$ trong trường hợp tìm kiếm không thành công.

Trong trường hợp bảng băm địa chỉ mở với phép thử tuyến tính, lưu ý rằng giả thiết các lần thử độc lập nhau là không thể chấp nhận. Kết quả tính được như sau:

Việc truy xuất bảng băm địa chỉ mở với phép thử tuyến tính và hệ số tải $\lambda$ cần số lần thử trung bình xấp xỉ bằng

$$\frac{1}{2}\left[1 + \frac{1}{1 - \lambda}\right]$$

trong trường hợp thành công và

$$\frac{1}{2}\left[1 + \frac{1}{(1 - \lambda)^2}\right]$$

trong trường hợp không thành công.

### 12.6.5. Các so sánh lý thuyết

| *Load factor* | 0.10 | 0.50 | 0.80 | 0.90 | 0.99 | 2.00 |
|---|---|---|---|---|---|---|
| *Successful search, expected number of probes:* | | | | | | |
| *Chaining* | 1.05 | 1.25 | 1.40 | 1.45 | 1.50 | 2.00 |
| *Open, Random probes* | 1.05 | 1.4 | 2.0 | 2.6 | 4.6 | — |
| *Open, Linear probes* | 1.06 | 1.5 | 3.0 | 5.5 | 50.5 | — |
| *Unsuccessful search, expected number of probes:* | | | | | | |
| *Chaining* | 0.10 | 0.50 | 0.80 | 0.90 | 0.99 | 2.00 |
| *Open, Random probes* | 1.1 | 2.0 | 5.0 | 10.0 | 100. | — |
| *Open, Linear probes* | 1.12 | 2.5 | 13. | 50. | 5000. | — |

**Hình 12.13** – So sánh lý thuyết các phương pháp băm

Hình 12.13 cho thấy các giá trị của các biểu thức trên với các trị khác nhau của hệ số tải λ.

Chúng ta có thể thấy được một vài kết luận từ bảng này. Trước hết, rõ ràng là bảng băm nối kết cần ít lần thử hơn bảng băm địa chỉ mở. Mặt khác, việc duyệt các danh sách liên kết thường chậm hơn là truy xuất mảng, điều này làm giảm ưu điểm của bảng băm nối kết, nhất là trong những trường hợp mà việc so sánh khóa có thể được thực hiện rất nhanh. Phương pháp nối kết trở nên hợp lý khi các phần tử có kích thước lớn và thời gian cần để so sánh các khóa là nhiều. Ngoài ra, nó còn tỏ ra có lợi thế khi việc tìm kiếm không thành công thường xảy ra, do những lúc quá trình tìm kiếm gặp được một danh sách rỗng hoặc một danh sách thật ngắn và có thể kết thúc nhanh với rất ít lần so sánh khóa.

Đối với việc tìm kiếm thành công trong bảng băm địa chỉ mở, phương pháp thử tuyến tính đơn giản không làm chậm quá trình tìm kiếm đi nhiều so với các phương pháp giải quyết đụng độ phức tạp khác, ít nhất là cho đến khi bảng gần

như đầy. Tuy nhiên, đối với việc tìm kiếm không thành công, hiện tượng gom tụ nhanh chóng làm cho phép thử tuyến tính suy thoái thành quá trình tìm kiếm tuần tự. Vì thế, chúng ta có thể kết luận rằng nếu việc tìm kiếm thường thành công, và hệ số tải vừa phải, thì phép thử tuyến tính sẽ đáp ứng được; còn trong những trường hợp khác, nên sử dụng những phương pháp giải quyết đụng độ khác như phương pháp thử bậc hai chẳng hạn.

### 12.6.6. Các so sánh thực nghiệm

Một điều quan trọng cần nhớ là các tính toán trong hình 12.13 chỉ là các con số xấp xỉ, và trong thực tế không có gì là hoàn toàn ngẫu nhiên, do đó chúng ta luôn biết rằng sẽ có một vài điều khác nhau giữa các kết quả lý thuyết và việc tính toán thực sự. Vì vậy, để so sánh, hình 12.14 cho thấy kết quả của việc nghiên cứu bằng thực nghiệm với 900 khóa lấy ngẫu nhiên giữa 0 và 1.

Nếu so sánh các con số trong hai bảng 12.15 và 12.16, chúng ta thấy rằng kết quả thực nghiệm trên bảng băm nối kết gần giống với kết quả lý thuyết. Các kết quả của phép thử bậc hai lại gần giống với kết quả lý thuyết của việc thử ngẫu nhiên; sự khác nhau có thể được giải thích dễ dàng là vì thử bậc hai chưa thật sự ngẫu nhiên. Đối với thử tuyến tính, các kết quả tương tự khi bảng còn tương đối trống, nhưng khi bảng gần như đầy thì các con số xấp xỉ được tính bằng lý thuyết khác nhiều so với thực nghiệm. Đó là hậu quả của các giả thiết đã được đơn giản hóa trong toán học.

| *Load factor* | 0.1 | 0.5 | 0.8 | 0.9 | 0.99 | 2.0 |
|---|---|---|---|---|---|---|
| *Successful search, average number of probes:* | | | | | | |
| *Chaining* | 1.04 | 1.2 | 1.4 | 1.4 | 1.5 | 2.0 |
| *Open, Quadratic probes* | 1.04 | 1.5 | 2.1 | 2.7 | 5.2 | — |
| *Open, Linear probes* | 1.05 | 1.6 | 3.4 | 6.2 | 21.3 | — |
| *Unsuccessful search, average number of probes:* | | | | | | |
| *Chaining* | 0.10 | 0.50 | 0.80 | 0.90 | 0.99 | 2.00 |
| *Open, Quadratic probes* | 1.13 | 2.2 | 5.2 | 11.9 | 126. | — |
| *Open, Linear probes* | 1.13 | 2.7 | 15.4 | 59.8 | 430. | — |

**Hình 12.14** – So sánh thực nghiệm các phương pháp băm.

So sánh với các phương pháp truy xuất thông tin khác, điều quan trọng cần lưu ý về tất cả những con số này là chúng chỉ phụ thuộc vào hệ số tải, mà không phụ thuộc vào số phần tử thực sự có trong bảng. Việc truy xuất từ bảng băm có 20,000 phần tử trong 40,000 vị trí có thể có của bảng, xét trung bình, không chậm hơn

việc tìm kiếm trong 20 phần tử trong 40 vị trí có thể có. Với việc tìm tuần tự, một danh sách có kích thước lớn gấp 1000 lần sẽ làm cho quá trình tìm lâu hơn 1000 lần. Với tìm kiếm nhị phân, tỉ lệ này giảm xuống 10 (chính xác hơn là lg1000), nhưng thời gian tìm kiếm vẫn luôn phụ thuộc vào kích thước của danh sách, điều này không có ở bảng băm.

Chúng ta có thể tổng kết những khảo sát về việc truy xuất từ n phần tử như sau:

- Tìm tuần tự là $\theta(n)$.
- Tìm nhị phân là $\theta(\log n)$.
- Truy xuất bảng băm là $\theta(1)$.

Cuối cùng, chúng ta nhấn mạnh về tầm quan trọng của việc lựa chọn một hàm băm tốt, một hàm băm thực hiện tính toán nhanh và rải đều các khóa trong bảng. Nếu hàm băm không tốt thì băm sẽ suy thoái về tìm kiếm tuần tự.

## 12.7. Kết luận: so sánh các phương pháp

Trong chương này và chương 7, chúng ta đã xem xét bốn phương pháp khác nhau để truy xuất thông tin:

- Tìm tuần tự,
- Tìm nhị phân,
- Tra cứu bảng, và
- Băm.

Nếu được hỏi rằng phương pháp nào là tốt nhất, trước hết chúng ta cần chọn ra các tiêu chí để đánh giá. Các tiêu chí gồm các yêu cầu của ứng dụng, và các mối quan tâm khác có ảnh hưởng lên sự chọn lựa cấu trúc dữ liệu, do hai phương pháp đầu chỉ có thể áp dụng với các danh sách còn hai phương pháp sau chỉ dành cho các bảng. Trong nhiều ứng dụng, chúng ta cũng được tự do trong việc chọn lựa giữa danh sách và bảng.

Về mặt tốc độ cũng như tính thuận lợi, việc tra cứu theo thứ tự trong các bảng chắc chắn là tốt nhất, nhưng có nhiều ứng dụng mà điều này lại không áp dụng được do tập các khóa khá thưa thớt hoặc đối với chúng danh sách tỏ ra ưu thế hơn. Một điều không thích ứng nữa là khi việc thêm và loại phần tử xảy ra thường xuyên, trong vùng nhớ liên tục các tác vụ này đòi hỏi phải di chuyển một số lớn dữ liệu.

Trong ba phương pháp còn lại, phương pháp nào là tốt nhất phụ thuộc vào tiêu chí khác như dạng của dữ liệu.

Tìm tuần tự là phương pháp mềm dẻo nhất trong các phương pháp. Dữ liệu có thể được lưu theo bất kỳ thứ tự nào, trong hiện thực liên tục hoặc liên kết. Tìm nhị phân đòi hỏi nhiều hơn, các khóa phải lưu theo thứ tự và dữ liệu phải lưu trong vùng nhớ cho phép truy xuất ngẫu nhiên (vùng nhớ liên tục). Băm còn đòi hỏi nhiều hơn nữa, tuy thứ tự khác thường của các khóa vẫn đáp ứng được việc truy xuất từ bảng băm, nhưng nói chung nó không có lợi cho bất kỳ một mục đích nào khác. Nếu như dữ liệu cần phải luôn sẵn sàng cho một sự khảo sát thì một thứ tự nào đó là cần thiết, và như vậy bảng băm không đáp ứng.

Cuối cùng là vấn đề liên quan đến việc tìm kiếm không thành công. Tìm tuần tự và băm khi không thành công thì xem như không có kết quả gì. Trong khi đó, nếu thất bại thì tìm nhị phân sẽ cho biết dữ liệu có khóa gần với khóa cần tìm, và như vậy nó có thể cung cấp thông tin hữu ích. Trong chương 9 và 10 chúng ta đã nghiên cứu các phương pháp lưu trữ dữ liệu dựa trên cơ sở cây, có kết hợp tính hiệu quả của tìm nhị phân với sự mềm dẻo của các cấu trúc liên kết.