# CHƯƠNG 9
# BẢNG BĂM

Trong chương này, chúng ta sẽ nghiên cứu bảng băm. Bảng băm là cấu trúc dữ liệu được sử dụng để cài đặt KDLTT từ điển. Nhớ lại rằng, KDLTT từ điển là một tập các đối tượng dữ liệu được xem xét đến chỉ với ba phép toán tìm kiếm, xen vào và loại bỏ. Đương nhiên là chúng ta có thể cài đặt từ điển bởi danh sách, hoặc bởi cây tìm kiếm nhị phân. Tuy nhiên bảng băm là một trong các phương tiện hiệu quả nhất để cài đặt từ điển. Trong chương này, chúng ta sẽ đề cập tới các vấn đề sau đây:

- Phương pháp băm và hàm băm.
- Các chiến lược giải quyết sự va chạm.
- Cài đặt KDLTT từ điển bởi bảng băm.

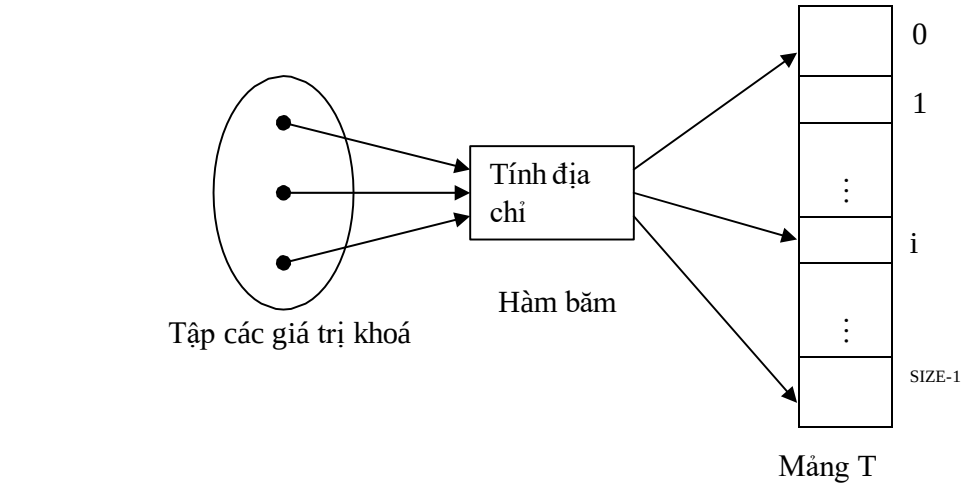
## 9.1 PHƯƠNG PHÁP BĂM

Vấn đề được đặt ra là, chúng ta có một tập dữ liệu, chúng ta cần đưa ra một CTDL cài đặt tập dữ liệu này sao cho các phép toán tìm kiếm, xen, loại được thực hiện hiệu quả. Trong các chương trước, chúng ta đã trình bày các phương pháp cài đặt KDLTT tập động (từ điển là trường hợp riêng của tập động khi mà chúng ta chỉ quan tâm tới ba phép toán tìm kiếm, xen, loại). Sau đây chúng ta trình bày một kỹ thuật mới để lưu giữ một tập dữ liệu, đó là phương pháp băm.

Nếu như các giá trị khoá của các dữ liệu là số nguyên không âm và nằm trong khoảng [0..SIZE-1], chúng ta có thể sử dụng một mảng data có cỡ SIZE để lưu tập dữ liệu đó. Dữ liệu có khoá là k sẽ được lưu trong thành phần data[k] của mảng. Bởi vì mảng cho phép ta truy cập trực tiếp tới từng thành phần của mảng theo chỉ số, do đó các phép toán tìm kiếm, xen, loại

được thực hiện trong thời gian O(1). Song đáng tiếc là, khoá có thể không phải là số nguyên, thông thường khoá còn có thể là số thực, là ký tự hoặc xâu ký tự. Ngay cả khoá là số nguyên, thì các giá trị khoá nói chung không chạy trong khoảng [0..SIZE-1].

Trong trường hợp tổng quát, khi khoá không phải là các số nguyên trong khoảng [0..SIZE-1], chúng ta cũng mong muốn lưu tập dữ liệu bởi mảng, để lợi dụng tính ưu việt cho phép truy cập trực tiếp của mảng. Giả sử chúng ta muốn lưu tập dữ liệu trong mảng T với cỡ là SIZE. Để làm được điều đó, với mỗi dữ liệu chúng ta cần định vị được vị trí trong mảng tại đó dữ liệu được lưu giữ. Nếu chúng ta đưa ra được cách tính chỉ số mảng tại đó lưu dữ liệu thì chúng ta có thể lưu tập dữ liệu trong mảng theo sơ đồ hình 9.1.




**Hình 9.1. Lược đồ phương pháp băm**.

Trong lược đồ hình 9.1, khi cho một dữ liệu có khoá là k, nếu tính địa chỉ theo k ta thu được chỉ số i, 0 <= i <= SIZE-1, thì dữ liệu sẽ được lưu trong thành phần mảng T[i].

Một hàm ứng với mỗi giá trị khoá của dữ liệu với một địa chỉ (chỉ số) của dữ liệu trong mảng được gọi là **hàm băm** (hash function). Phương pháp lưu tập dữ liệu theo lược đồ trên được gọi là phương pháp băm (hashing). Trong lược đồ 9.1, mảng T được gọi là **bảng băm** (hash table).

Như vậy, hàm băm là một ánh xạ h từ tập các giá trị khoá của dữ liệu vào tập các số nguyên {0,1,…, SIZE-1}, trong đó SIZE là cỡ của mảng dùng để lưu tập dữ liệu, tức là:

h : K → {0,1,…,SIZE-1}

với K là tập các giá trị khoá. Cho một dữ liệu có khoá là k, thì h(k) được gọi là **giá trị băm** của khoá k, và dữ liệu được lưu trong T[h(k)].

Nếu hàm băm cho phép ứng các giá trị khoá khác nhau với các chỉ số khác nhau, tức là nếu $k_1 \neq k_2$ thì $h(k_1) \neq h(k_2)$, và việc tính chỉ số h(k) ứng với mỗi khoá k chỉ đòi hỏi thời gian hằng, thì các phép toán tìm kiếm, xen, loại cũng chỉ cần thời gian O(1). Tuy nhiên, trong thực tế một hàm băm có thể ánh xạ hai hay nhiều giá trị khoá tới cùng một chỉ số nào đó. Điều đó có nghĩa là chúng ta phải lưu các dữ liệu đó trong cùng một thành phần mảng, mà mỗi thành phần mảng chỉ cho phép lưu một dữ liệu ! Hiện tượng này được gọi là **sự va chạm** (collision). Vấn đề đặt ra là, giải quyết sự va chạm như thế nào? Chẳng hạn, giả sử dữ liệu $d_1$ với khoá $k_1$ đã được lưu trong T[i], $i = h(k_1)$; bây giờ chúng ta cần xen vào dữ liệu $d_2$ với khoá $k_2$, nếu $h(k_2)$
$= i$ thì dữ liệu $d_2$ cần được đặt vào vị trí nào trong mảng?

Như vậy, một hàm băm như thế nào thì được xem là tốt. Từ những điều đã nêu trên, chúng ta đưa ra các tiêu chuẩn để thiết kế một hàm băm tốt như sau:

1. Tính được dễ dàng và nhanh địa chỉ ứng với mỗi khoá.
2. Đảm bảo ít xảy ra va chạm.

## 9.2 CÁC HÀM BĂM

Trong các hàm băm được đưa ra dưới đây, chúng ta sẽ ký hiệu k là một giá trị khoá bất kỳ và SIZE là cỡ của bảng băm. Trước hết chúng ta sẽ xét trường hợp các giá trị khoá là các số nguyên không âm. Nếu không phải là trường hợp này (chẳng hạn, khi các giá trị khoá là các xâu ký tự), chúng ta chỉ cần chuyển đổi các giá trị khoá thành các số nguyên không âm, sau đó băm chúng bằng một phương pháp cho trường hợp khoá là số nguyên.

Có nhiều phương pháp thiết kế hàm băm đã được đề xuất, nhưng được sử dụng nhiều nhất trong thực tế là các phương pháp được trình bày sau đây:

### 9.2.1 Phương pháp chia

Phương pháp này đơn giản là lấy phần dư của phép chia khoá k cho cỡ bảng băm SIZE làm giá trị băm:

h(k) = k mod SIZE

Bằng cách này, giá trị băm h(k) là một trong các số 0,1,…, SIZE-1. Hàm băm này được cài đặt trong C++ như sau:

```
unsigned int hash(int k, int SIZE)
{
    return k % SIZE;
}
```

Trong phương pháp này, để băm một khoá k chỉ cần một phép chia, nhưng hạn chế cơ bản của phương pháp này là để hạn chế xảy ra va chạm, chúng ta cần phải biết cách lựa chọn cỡ của bảng băm. Các phân tích lý thuyết đã chỉ ra rằng, để hạn chế va chạm, khi sử dụng phương pháp băm này chúng ta nên lựa chọn SIZE là số nguyên tố, tốt hơn là số nguyên tố có dạng đặc biệt, chẳng hạn có dạng 4k+3. Ví dụ, có thể chọn SIZE = 811, vì 811 là số nguyên tố và 811 = 4 . 202 + 3.

### 9.2.2 Phương pháp nhân

Phương pháp chia có ưu điểm là rất đơn giản và dễ dàng tính được giá trị băm, song đối với sự va chạm nó lại rất nhạy cảm với cỡ của bảng băm. Để hạn chế sự va chạm, chúng ta có thể sử dụng phương pháp nhân, phương pháp này có ưu điểm là ít phụ thuộc vào cỡ của bảng băm.

Phương pháp nhân tính giá trị băm của khoá k như sau. Đầu tiên, ta tính tích của khoá k với một hằng số thực $\alpha$, $0 < \alpha < 1$. Sau đó lấy phần thập phân của tích $\alpha k$ nhân với SIZE, phần nguyên của tích này được lấy làm giá trị băm của khoá k. Tức là:

$$h(k) = \lfloor (\alpha k - \lfloor \alpha k \rfloor) \, . \, SIZE \rfloor$$

(Ký hiệu $\lfloor x \rfloor$ chỉ phần nguyên của số thực x, tức là số nguyên lớn nhất <=x, chẳng hạn $\lfloor 3 \rfloor = 3$, $\lfloor 3.407 \rfloor = 3$).

Chú ý rằng, phần thập phân của tích $\alpha k$, tức là $\alpha k - \lfloor \alpha k \rfloor$, là số thực dương nhỏ hơn 1. Do đó tích của phần thập phân với SIZE là số dương nhỏ hơn SIZE. Từ đó, giá trị băm h(k) là một trong các số nguyên 0,1,…, SIZE- 1.

Để có thể phân phối đều các giá trị khoá vào các vị trí trong bảng băm, trong thực tế người ta thường chọn hằng số $\alpha$ như sau:

$$\alpha = \Phi^{-1} \approx 0{,}61803399$$

Chẳng hạn, nếu cỡ bảng băm là SIZE = 1024 và hằng số $\alpha$ được chọn như trên, thì với k = 1849970, ta có

$$h(k) = \lfloor (1024.(\alpha.1849970 - \lfloor \alpha.1849970 \rfloor) \rfloor = 348.$$

### 9.2.3 Hàm băm cho các giá trị khoá là xâu ký tự

Để băm các xâu ký tự, trước hết chúng ta chuyển đổi các xâu ký tự thành các số nguyên. Các ký tự trong bảng mã ASCII gồm 128 ký tự được đánh số từ 0 đến 127, đo đó một xâu ký tự có thể xem như một số trong hệ đếm cơ số 128. Áp dụng phương pháp chuyển đổi một số trong hệ đếm bất kỳ sang một số trong hệ đếm cơ số 10, chúng ta sẽ chuyển đổi được một xâu ký tự

thành một số nguyên. Chẳng hạn, xâu "NOTE" được chuyển thành một số nguyên như sau:

"NOTE" → 'N'.$128^3$ + 'O'.$128^2$ + 'T'.128 + 'E' =

= 78.$128^3$ + 79.$128^2$ + 84.128 + 69

Vấn đề nảy sinh với cách chuyển đổi này là, chúng ta cần tính các luỹ thừa của 128, với các xâu ký tự tương đối dài, kết quả nhận được sẽ là một số nguyên cực lớn vượt quá khả năng biểu diễn của máy tính.

Trong thực tế, thông thường một xâu ký tự được tạo thành từ 26 chữ cái và 10 chữ số, và một vài ký tự khác. Do đó chúng ta thay 128 bởi 37 và tính số nguyên ứng với xâu ký tự theo luật Horner. Chẳng hạn, số nguyên ứng với xâu ký tự "NOTE" được tính như sau:

"NOTE" → 78.$37^3$ + 79.$37^2$ + 84.37 + 69=

= ((78.37 + 79).37 +84).37 +69

Sau khi chuyển đổi xâu ký tự thành số nguyên bằng phương pháp trên, chúng ta sẽ áp dụng phương pháp chia để tính giá trị băm. Hàm băm các xâu ký tự được cài đặt như sau:

```
unsigned   int   hash(const string &k, int SIZE)
{
        unsigned  int  value = 0;
        for (int i=0; i< k.length(); i++)
                value = 37 * value + k[i];
        return    value % SIZE;
}
```

## 9.3 CÁC PHƯƠNG PHÁP GIẢI QUYẾT VA CHẠM

Trong mục 9.2 chúng ta đã trình bày các phương pháp thiết kế hàm băm nhằm hạn chế xẩy ra va chạm. Tuy nhiên trong các ứng dụng, sự va chạm là không tránh khỏi. Chúng ta sẽ thấy rằng, cách giải quyết va chạm ảnh hưởng trực tiếp đến hiệu quả của các phép toán từ điển trên bảng băm. Trong mục này chúng ta sẽ trình bày hai phương pháp giải quyết va chạm. Trong phương pháp thứ nhất, mỗi khi xảy ra va chạm, chúng ta tiến hành thăm dò để tìm một vị trí còn trống trong bảng và đặt dữ liệu mới vào đó. Một phương pháp khác là, chúng ta tạo ra một cấu trúc dữ liệu lưu giữ tất cả các dữ liệu được băm vào cùng một vị trí trong bảng và "gắn" cấu trúc dữ liệu này vào vị trí đó trong bảng.

### 9.3.1 Phương pháp định địa chỉ mở

Trong phương pháp này, các dữ liệu được lưu trong các thành phần của mảng, mỗi thành phần chỉ chứa được một dữ liệu. Vì thế, mỗi khi cần xen một dữ liệu mới với khoá k vào mảng, nhưng tại vị trí h(k) đã chứa dữ liệu, chúng ta sẽ tiến hành thăm dò một số vị trí khác trong mảng để tìm ra một vị trí còn trống và đặt dữ liệu mới vào vị trí đó. Phương pháp tiến hành thăm dò để phát hiện ra vị trí trống được gọi là phương pháp định địa chỉ mở (open addressing).

Giả sử vị trí mà hàm băm xác định ứng với khoá k là i, i=h(k). Từ vị trí này chúng ta lần lượt xem xét các vị trí

$i_0, i_1, i_2, \ldots, i_m, \ldots$

Trong đó $i_0 = i$, $i_m (m=0,1,2,\ldots)$ là vị trí thăm dò ở lần thứ m. Dãy các vị trí này sẽ được gọi là dãy thăm dò. Vấn đề đặt ra là, xác định dãy thăm dò như thế nào? Sau đây chúng ta sẽ trình bày một số phương pháp thăm dò và phân tích ưu khuyết điểm của mỗi phương pháp.

**Thăm dò tuyến tính**.

Đây là phương pháp thăm dò đơn giản và dễ cài đặt nhất. Với khoá k, giả sử vị trí được xác định bởi hàm băm là i=h(k), khi đó dãy thăm dò là

i , i+1, i+2 , …

Như vậy thăm dò tuyến tính có nghĩa là chúng ta xem xét các vị trí tiếp liền nhau kể từ vị trí ban đầu được xác định bởi hàm băm. Khi cần xen vào một dữ liệu mới với khoá k, nếu vị trí i = h(k) đã bị chiếm thì ta tìm đến các vị trí đi liền sau đó, gặp vị trí còn trống thì đặt dữ liệu mới vào đó.

**Ví dụ**. Giả sử cỡ của mảng SIZE = 11. Ban đầu mảng T rỗng, và ta cần xen lần lượt các dữ liệu với khoá là 388, 130, 13, 14, 926 vào mảng. Băm khoá 388, h(388) = 3, vì vậy 388 được đặt vào T[3]; h(130) = 9, đặt 130 vào T[9]; h(13) = 2, đặt 13 trong T[2]. Xét tiếp dữ liệu với khoá 14, h(14) = 3, xẩy ra va chạm (vì T[3] đã bị chiếm bởi 388), ta tìm đến vị trí tiếp theo là 4, vị trí này trống và 14 được đặt vào T[4]. Tương tự, khi xen vào 926 cũng xảy ra va chạm, h(926) = 2, tìm đến các vị trí tiếp theo 3, 4, 5 và 92 được đặt vào T[5]. Kết quả là chúng ta nhận được mảng T như trong hình 9.2.

| T | | | 13 | 388 | 14 | 926 | | | | 130 | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

**Hình 9.2. Bảng băm sau khi xen vào các dữ liệu 38, 130, 13, 14 và 926**

Bây giờ chúng ta xét xem, nếu lưu tập dữ liệu trong mảng bằng phương pháp định địa chỉ mở thì các phép toán tìm kiếm, xen, loại được tiến hành như thế nào. Các kỹ thuật tìm kiếm, xen, loại được trình bày dưới đây có thể sử dụng cho bất kỳ phương pháp thăm dò nào. Trước hết cần lưu ý rằng, để tìm, xen, loại chúng ta phải sử dụng cùng một phương pháp thăm dò, chẳng hạn thăm dò tuyến tính. Giả sử chúng ta cần tìm dữ liệu với khoá là k. Đầu tiên cần băm khoá k, giả sử h(k)=i. Nếu trong bảng ta chưa một lần nào thực hiện phép toán loại, thì chúng ta xem xét các dữ liệu chứa trong mảng tại vị trí i và các vị trí tiếp theo trong dãy thăm dò, chúng ta sẽ phát

hiện ra dữ liệu cần tìm tại một vị trí nào đó trong dãy thăm dò, hoặc nếu gặp một vị trí trống trong dãy thăm dò thì có thể dừng lại và kết luận dữ liệu cần tìm không có trong mảng. Chẳng hạn chúng ta muốn tìm xem mảng trong hình 9.2 có chứa dữ liệu với khoá là 47? Bởi vì h(47) = 3, và dữ liệu được lưu theo phương pháp thăm dò tuyến tính, nên chúng ta lần lượt xem xét các vị trí 3, 4, 5. Các vị trí này đều chứa dữ liệu khác với 47. Đến vị trí 6, mảng trống. Vậy ta kết luận 47 không có trong mảng.

Để loại dữ liệu với khoá k, trước hết chúng ta cần áp dụng thủ tục tìm kiếm đã trình bày ở trên để định vị dữ liệu ở trong mảng. Giả sử dữ liệu được lưu trong mảng tại vị trí p. Loại dữ liệu ở vị trí p bằng cách nào? Nếu đặt vị trí p là vị trí trống, thì khi tìm kiếm nếu thăm dò gặp vị trí trống ta không thể dừng và đưa ra kết luận dữ liệu không có trong mảng. Chẳng hạn, trong mảng hình 9.2, ta loại dữ liệu 388 bằng cách xem vị trí 3 là trống, sau đó ta tìm dữ liệu 926, vì h (926) = 2 và T[2] không chứa 926, tìm đến vị trí 3 là trống, nhưng ta không thể kết luận 926 không có trong mảng. Thực tế 926 ở vị trí 5, vì lúc đưa 926 vào mảng các vị trí 2, 3, 4 đã bị chiếm. Vì vậy để đảm bảo thủ tục tìm kiếm đã trình bày ở trên vẫn còn đúng cho trường hợp đã thực hiện phép toán loại, khi loại dữ liệu ở vị trí p chúng ta đặt vị trí p là vị trí đã loại bỏ. Như vậy, chúng ta quan niệm mỗi vị trí i trong mảng (0 <= i
<= SIZE-1) có thể là vị trí trống (EMPTY), vị trí đã loại bỏ (DELETED), hoặc vị trí chứa dữ liệu (ACTIVE). Đương nhiên là khi xen vào dữ liệu mới, chúng ta có thể đặt nó vào vị trí đã loại bỏ.

Việc xen vào mảng một dữ liệu mới được tiến hành bằng cách lần lượt xem xét các vị trí trong dãy thăm dò ứng với mỗi khoá của dữ liệu, khi gặp một vị trí trống hoặc vị trí đã được loại bỏ thì đặt dữ liệu vào đó.

Sau đây là hàm thăm dò tuyến tính

```
int     Probing (int i, int m, int SIZE)
// SIZE là cỡ của mảng
// i là vị trí ban đầu được xác định bởi băm khoá k, i = h(k)
```

```
// hàm trả về vị trí thăm dò ở lần thứ  m= 0, 1, 2,…
{
        return    (i + m)  %   SIZE;
}
```

Phương pháp thăm dò tuyến tính có ưu điểm là cho phép ta xem xét tất cả các vị trí trong mảng, và do đó phép toán xen vào luôn luôn thực hiện được, trừ khi mảng đầy. Song nhược điểm của phương pháp này là các dữ liệu tập trung thành từng đoạn, trong quá trình xen các dữ liệu mới vào, các đoạn có thể gộp thành đoạn dài hơn. Điều đó làm cho các phép toán kém  hiệu quả, chẳng hạn nếu i = h(k) ở đầu một đoạn, để tìm dữ liệu với khoá k chúng ta cần xem xét cả một đoạn dài.

**Thăm dò bình phương**

Để khắc phục tình trạng dữ liệu tích tụ thành từng cụm trong phương pháp thăm dò tuyến tính, chúng ta không thăm dò các vị trí kế tiếp liền nhau, mà thăm dò bỏ chỗ theo một quy luật nào đó.

Trong thăm dò bình phương, nếu vị trí ứng với khoá k là i = h(k), thì dãy thăm dò là

$i , i + 1^2, i + 2^2,\dots , i + m^2,\dots$

Ví dụ. Nếu cỡ của mảng SIZE = 11, và i = h(k) = 3, thì thăm dò bình phương cho phép ta tìm đến các địa chỉ 3, 4, 7, 1, 8 và 6.

Phương pháp thăm dò bình phương tránh được sự tích tụ dữ liệu thành từng đoạn và tránh được sự tìm kiếm tuần tự trong các đoạn. Tuy nhiên nhược điểm của nó là không cho phép ta tìm đến tất cả các vị trí trong mảng, chẳng hạn trong ví dụ trên, trong số 11 vị trí từ 0, 1, 2, …, 10, ta chỉ tìm đến các vị trí 3, 4, 7, 1, 8 và 6. Hậu quả của điều đó là, phép toán xen vào có thể không thực hiện được, mặc dầu trong mảng vẫn còn các vị trí không chứa dữ liệu. Chúng ta có thể dễ dàng chứng minh được khẳng định sau đây:

Nếu cỡ của mảng là số nguyên tố, thì thăm dò bình phương cho phép ta tìm đến một nửa số vị trí trong mảng. Cụ thể hơn là, các vị trí thăm dò $h(k) + m^2$ (mode SIZE) với $m = 0, 1, \ldots, \lfloor SIZE/2 \rfloor$ là khác nhau.

Từ khẳng định trên chúng ta suy ra rằng, nếu cỡ của mảng là số nguyên tố và mảng không đầy quá 50% thì phép toán xen vào luôn luôn thực hiện được.

**Băm kép**

Phương pháp băm kép (double hashing) có ưu điểm như thăm dò bình phương là hạn chế được sự tích tụ dữ liệu thành cụm; ngoài ra nếu chúng ta chọn cỡ của mảng là số nguyên tố, thì băm kép còn cho phép ta thăm dò tới tất cả các vị trí trong mảng.

Trong thăm dò tuyến tính hoặc thăm dò bình phương, các vị trí thăm dò cách vị trí xuất phát một khoảng cách hoàn toàn xác định trước và các khoảng cách này không phụ thuộc vào khoá. Trong băm kép, chúng ta sử dụng hai hàm băm $h_1$ và $h_2$:

- Hàm băm $h_1$ đóng vai trò như hàm băm h trong các phương pháp trước, nó xác định vị trí thăm dò đầu tiên
- Hàm băm $h_2$ xác định bước thăm dò.

Điều đó có nghĩa là, ứng với mỗi khoá k, dãy thăm dò là:

$$h_1(k) + m\, h_2(k), \text{ với } m = 0, 1, 2, \ldots$$

Bởi vì $h_2(k)$ là bước thăm dò, nên hàm băm $h_2$ phải thoả mãn điều kiện $h_2(k) \neq 0$ với mọi k.

Có thể chứng minh được rằng, nếu cỡ của mảng và bước thăm dò $h_2(k)$ nguyên tố cùng nhau thì phương pháp băm kép cho phép ta tìm đến tất cả các vị trí trong mảng. Khẳng định trên sẽ đúng nếu chúng ta lựa chọn cỡ của mảng là số nguyên tố.

**Ví dụ**. Giả sử SIZE = 11, và các hàm băm được xác định như sau:

$h_1(k) = k \% 11$

$h_2(k) = 1 + (k \% 7)$

với k = 58, thì bước thăm dò là $h_2(58) = 1 + 2 = 3$, do đó dãy thăm dò là: $h_1(58) = 3, 6, 9, 1, 4, 7, 10, 2, 5, 8, 0$. còn với k = 36, thì bước thăm dò là $h_2(36) = 1 + 1 = 2$, và dãy thăm dò là 3, 5, 7, 9, 0, 2, 4, 6, 8, 10.

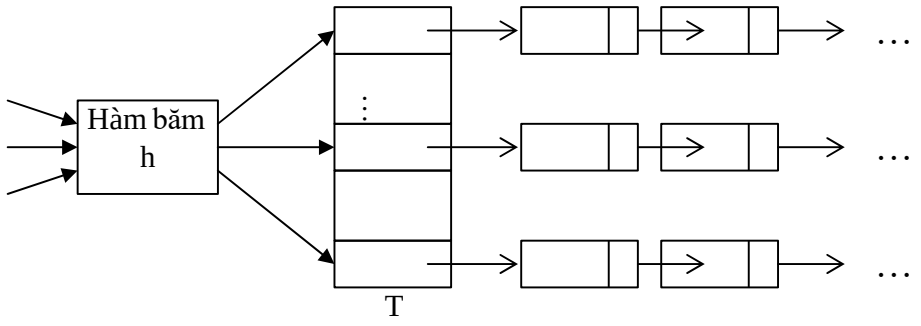
Trong các ứng dụng, chúng ta có thể chọn cỡ mảng SIZE là số nguyên tố và chọn M là số nguyên tố, M < SIZE, rồi sử dụng các hàm băm

$h_1(k) = k \% SIZE$

$h_2(k) = 1 + (k \% M)$

### 9.3.2 Phương pháp tạo dây chuyền

Một cách tiếp cận khác để giải quyết sự va chạm là chúng ta tạo một cấu trúc dữ liệu để lưu tất cả các dữ liệu được băm vào cùng một vị trí trong mảng. Cấu trúc dữ liệu thích hợp nhất là danh sách liên kết (dây chuyền). Khi đó mỗi thành phần trong bảng băm T[i], với i = 0, 1, …, SIZE – 1, sẽ chứa con trỏ trỏ tới đầu một DSLK. Cách giải quyết va chạm như trên được gọi là phương pháp **tạo dây chuyền** (separated chaining). Lược đồ lưu tập dữ liệu trong bảng băm sử dụng phương pháp tạo dây chuyền được mô tả trong hình 9.3.




**Hình 9.3. Phương pháp tạo dây chuyền**.

Ưu điểm của phương pháp giải quyết va chạm này là số dữ liệu được lưu không phụ thuộc vào cỡ của mảng, nó chỉ hạn chế bởi bộ nhớ cấp phát động cho các dây chuyền.

Bây giờ chúng ta xét xem các phép toán từ điển (tìm kiếm, xen, loại) được thực hiện như thế nào. Các phép toán được thực hiện rất dễ dàng, để xen vào bảng băm dữ liệu khoá k, chúng ta chỉ cần xen dữ liệu này vào đầu DSLK được trỏ tới bởi con trỏ T[h(k)]. Phép toán xen vào chỉ đòi hỏi thời gian O(1), nếu thời gian tính giá trị băm h(k) là O(1). Việc tìm kiếm hoặc loại bỏ một dữ liệu với khoá k được quy về tìm kiếm hoặc loại bỏ trên DSLK T[h(k)]. Thời gian tìm kiếm hoặc loại bỏ đương nhiên là phụ thuộc vào độ dài của DSLK.

Chúng ta có nhận xét rằng, dù giải quyết va chạm bằng cách thăm dò, hay giải quyết va chạm bằng cách tạo dây chuyền, thì bảng băm đều không thuận tiện cho sự thực hiện các phép toán tập động khác, chẳng hạn phép toán Min (tìm dữ liệu có khoá nhỏ nhất), phép toán DeleteMin (loại dữ liệu có khoá nhỏ nhất), hoặc phép duyệt dữ liệu.

Sau này chúng ta sẽ gọi bảng băm với giải quyết va chạm bằng phương pháp định địa chỉ mở là **bảng băm địa chỉ mở**, còn bảng băm giải quyết va chạm bằng cách tạo dây chuyền là **bảng băm dây chuyền**.

## 9.4 CÀI ĐẶT BẢNG BĂM ĐỊA CHỈ MỞ

Trong mục này chúng ta sẽ nghiên cứu sự cài đặt KDLTT từ điển bởi bảng băm địa chỉ mở. Chúng ta sẽ giả thiết rằng, các dữ liệu trong từ điển có kiểu Item nào đó, và chúng chứa một trường dùng làm khoá tìm kiếm (trường key), các giá trị khoá có kiểu keyType. Ngoài ra để đơn giản cho viết ta giả thiết rằng, có thể truy cập trực tiếp trường key. Như đã thảo luận trong mục 9.3.1, trong bảng băm T, mỗi thành phần T[i], 0 <= i <= SIZE -1 , sẽ chứa hai biến: biến data để lưu dữ liệu và biến state để lưu trạng thái của vị trí i, trạng thái của vị trí i có thể là rỗng (EMPTY), có thể chứa dữ liệu (ACTIVE), hoặc có thể đã loại bỏ (DELETED). Chúng ta sẽ cài đặt KDLTT bởi lớp OpenHash phụ thuộc tham biến kiểu Item, lớp này sử dụng một hàm băm Hash và một hàm thăm dò Probing đã được cung cấp. Lớp OpenHash được khai báo trong hình 9.4.

```
typedef   int   keyType;
const   int   SIZE = 811;
template <class  Item>
class       OpenHash
{
public:
        OpenHash(); // khởi tạo bảng băm rỗng.
        bool     Search(keyType    k, Item & I)  const;
        // Tìm dữ liệu có khoá là k.
        // Hàm trả về true (false) nếu tìm thấy (không tìm thấy).
        // Nếu tìm kiếm thành công, biến I ghi lại dữ liệu cần tìm.
        void     Insert(const Item &  object, bool &  Suc)
        // Xen vào dữ liệu object. biến Suc nhận giá trị true
        // nếu phép xen thành công, và false nếu thất bại.
        void     Delete(keyType    k);
        // Loại khỏi bảng băm dữ liệu có khoá k.
        enum   stateType {ACTIVE, EMPTY, DELETED};
private:
        struct     Entry
        {
            Item     data;
            stateType    state;
        }
```

```
    Entry   T[SIZE];
    bool    Find(keyType k, int & index, int & index1)    const;
    // Hàm thực hiện thăm dò tìm dữ liệu có khoá k.
    // Nếu thành công, hàm trả về true và biến index ghi lại chỉ
    // số tại đó chứa dữ liệu.
    // Nếu thất bại, hàm trả về false và biến index1 ghi lại
    // chỉ số ở trạng thái EMPTY hoặc DELETED nếu thăm dò
    // phát hiện ra.
};
```

**Hình 9.4. Định nghĩa lớp OpenHash**.

Bây giờ chúng ta cài đặt các hàm thành phần của lớp OpenHash. Hàm kiến tạo bảng băm rỗng được cài đặt như sau:

```
template <class Item>
OpenHash<Item>::OpenHash()
{
    for ( int i = 0 ; i < SIZE ; i++ )
        T[i].state = EMPTY;
}
```

Chú ý rằng, các phép toán tìm kiếm, xen, loại đều cần phải thực hiện thăm dò để phát hiện ra dữ liệu cần tìm hoặc để phát hiện ra vị trí rỗng (hoặc bị trí đã loại bỏ) để đưa vào dữ liệu mới. Vì vậy, trong lớp OpenHash chúng ta đã đưa vào hàm ẩn Find. Sử dụng hàm Find ta dễ dàng cài đặt được các hàm Search, Insert và Delete. Trước hết chúng ta cài đặt hàm Find. Trong hàm Find khi mà quá trình thăm dò phát hiện ra vị trí rỗng thì có nghĩa là bảng không chứa dữ liệu cần tìm, song trước khi đạt tới vị trí rỗng có thể ta

đã phát hiện ra các vị trí đã loại bỏ, biến index1 sẽ ghi lại vị trí đã loại bỏ đầu tiên đã phát hiện ra . Còn nếu phát hiện ra vị trí rỗng, nhưng trước đó ta không gặp vị trí đã loại bỏ nào, thì biến index1 sẽ ghi lại vị trí rỗng. Hàm Find được cài đặt như sau:

```
template <class Item>
bool OpenHash<Item>::Find(keyType k, int & index, int & index1)
{
        int i = Hash(k);
        index = 0;
        index1 = i;
        for (int m = 0 ; m <  SIZE ; m++)
        {
                int   n = Probing(i,m); // vị trí thăm dò ở lần thứ m.
                if (T[n].state = = ACTIVE && T[n].data.key = = k )
                  {
                     index = n;
                     return   true;
                   }
                else if (T[n].state = = EMPTY)
                     {
                       if (T[index1].state != DELETED)
                                index1 = n;
                        return   false;
                      }
```

```
            else if (T[n].state = = DELETED && T[index1].state
                                        != DELETED)
                index1 = n;
        }
        return  false; // Dừng thăm dò mà vẫn không tìm ra dữ liệu
                    //  và cũng không phát hiện ra vị trí rỗng.
    }
```

Sử dụng hàm Find, các hàm tìm kiếm, xen, loại được cài đặt như sau:

```
template<class Item>
bool   OpenHash<Item>:: Search(keyType k, Item &I)
{
        int   ind, ind1;
        if (Find(k,ind,ind1))
            {
               I = T[ind].data;
               return    true;
            }
        else
            {
               I = *(new Item); // giá trị của I là giả
               return   false;
            }
}
```

```
template <class Item>
void   OpenHash<Item>::  Insert(const Item & object, bool &  Suc)
{
        int    ind, ind1;
        if (!Find(object.key, ind, ind1))
        if (T[ind1].state = = DELETED || T[ind1].state = = EMPTY)
            {
                T[ind1].data  =  object;
                T[ind1].state  = ACTIVE;
                Suc  =  true;
             }
        else    Suc  =  false;
}

template <class Item>
void   OpenHash<Item>:: Delete(keyType    k)
{
        int   ind, ind1;
        if (Find(k, ind, ind1))
                T[ind].state = DELETED;
}
```

Trên đây chúng ta đã cài đặt bảng băm địa chỉ mở bởi mảng có cỡ cố định. Hạn chế của cách này là, phép toán Insert có thể không thực hiện được do mảng đầy hoặc có thể mảng không đầy nhưng thăm dò không phát hiện ra vị trí rỗng hoặc vị trí đã loại bỏ để đặt dữ liệu vào. Câu hỏi đặt ra là,

chúng ta có thể cài đặt bởi mảng động như chúng ta đã làm khi cài đặt KDLTT tập động (xem 4.4). Câu trả lời là có, tuy nhiên cài đặt bảng băm bởi mảng động sẽ phức tạp hơn, vì các lý do sau:

- Cỡ của mảng cần là số nguyên tố, do đó chúng ta cần tìm số nguyên tố tiếp theo SIZE làm cỡ của mảng mới.
- Hàm băm phụ thuộc vào cỡ của mảng, chúng ta không thể sao chép một cách đơn giản mảng cũ sang mảng mới như chúng ta đã làm trước đây, mà cần phải sử dụng hàm Insert để xen từng dữ liệu của bảng cũ sang bảng mới

## 9.5 CÀI ĐẶT BẢNG BĂM DÂY CHUYỀN

Trong mục này chúng ta sẽ cài đặt KDLTT từ điển bởi bảng băm dây chuyền. Lớp ChainHash phụ thuộc tham biến kiểu Item với các giả thiết như trong mục 9.4. Lớp này được định nghĩa trong hình 9.5.

---

```
template<class Item>
class       ChainHash
{
        public:
             static   const  int  SIZE =  811;
             ChainHash(); // Hàm kiến tạo mặc định.
             ChainHash(const ChainHash  &  Table);//Kiến tạo copy.
             ~ChainHash();//Hàm huỷ
             void Operator=(const ChainHash & Table);//Toán tử gán
             // Các phép toán từ điển:
             bool   Search(keyType k, Item & I)  const;
```

```
        void    Insert(const Item & object, bool & Suc);
        void    Delete(keyType   k);
    private:
        struct     Cell
        {
            Item    data;
            Cell*   next;
        };  // Cấu trúc tế bào trong dây chuyền.
        Cell*   T[SIZE];// Mảng các con trỏ trỏ  đầu các dây chuyền
};
```

**Hình 9.5. Lớp ChainHash**.

Sau đây chúng ta cài đặt các hàm thành phần của lớp ChainHash. Để khởi tạo ra bảng băm rỗng, chúng ta chỉ cần đặt các thành phần trong mảng T là con trỏ NULL.

Hàm kiến tạo mặc định như sau:

```
template<class Item>
ChainHash<Item>::ChainHash()
{
    for ( int i = 0 ; i < SIZE ; i++ )
        T[i] = NULL;
}
```

Các hàm tìm kiếm, xen, loại được cài đặt rất đơn giản, sau khi băm chúng ta chỉ cần áp dụng các kỹ thuật tìm kiếm, xen, loại trên các DSLK. Các hàm Search, Insert và Delete được xác định dưới đây:

```
template<class Item>
bool ChainHash<Item>::Search(keyType k, Item & I)
{
        int  i = Hash(k);
        Cell*  P = T[i];
        while (P ! = NULL)
                if (P →data.key = = k)
                        {
                           I = P →data;
                           return   true;
                        }
                else P= P →next;
        I = *(new Item); // giá trị của biến I là giả.
        return    false;
}
template<class Item>
void   ChainHash<Item>::Insert(const Item &  object, bool &  Suc)
{
        int   i = Hash(k);
        Cell*   P = new Cell;
        If (P != NULL)
        {
                P→data = object;
                P→next = T[i];
```

```
            T[i] = P;  //Xen vào đầu dây chuyền.
            Suc = true;
        }
        else    Suc  = false;
}

template <class Item>
void   ChainHash<Item>::Delete(keyType  k)
{
        int   i = Hash(k);
        Cell*   P;
        If (T[i] != NULL)
        If (T[i]→data.key = = k)
            {
                P = T[i];
                T[i] = T[i]→next;
                delete   P;
             }
        else
            {
                P = T[i];
                Cell*  Q = P→next;
                while (Q != NULL)
                   if (Q→data.key = = k)
```

```
                {
                    P→next = Q→next;
                    delete  Q;
                Q = NULL;
                }
            else
                {
                    P  =  Q;
                    Q  =  Q→next;
                }
        }
    }
```

Ưu điểm lớn nhất của bảng băm dây chuyền là, phép toán Insert luôn luôn được thực hiện, chỉ trừ khi bộ nhớ để cấp phát động đã cạn kiệt. Ngoài ra, các phép toán tìm kiếm, xen, loại, trên bảng băm dây chuyền cũng rất đơn giản. Tuy nhiên, phương pháp này tiêu tốn bộ nhớ giành cho các con trỏ trong các dây chuyền.

## 9.6 HIỆU QUẢ CỦA PHƯƠNG PHÁP BĂM

Trong mục này, chúng ta sẽ phân tích thời gian thực hiện các phép toán từ điển (tìm kiếm, xen, loại) khi sử dụng phương pháp băm. Trong trường hợp xấu nhất, khi mà hàm băm băm tất cả các giá trị khoá vào cùng một chỉ số mảng để tìm kiếm chẳng hạn, chúng ta cần xem xét từng dữ liệu giống như tìm kiếm tuần tự, vì vậy thời gian các phép toán đòi hỏi là O(N), trong đó N là số dữ liệu.

Sau đây chúng ta sẽ đánh giá thời gian trung bình cho các phép toán từ điển. Đánh giá này dựa trên giả thiết hàm băm phân phối đều các khoá

vào các vị trí trong bảng băm (uniform hashing). Chúng ta sẽ sử dụng một tham số $\alpha$, được gọi là **mức độ đầy** (load factor). Mức độ đầy $\alpha$ là tỷ số giữa số dữ liệu hiện có trong bảng băm và cỡ của bảng, tức là

$$\alpha = \frac{N}{SIZE}$$

trong đó, N là số dữ liệu trong bảng. Rõ ràng là, khi $\alpha$ tăng thì khả năng xẩy ra va chạm sẽ tăng, điều này kéo theo thời gian tìm kiếm sẽ tăng. Như vậy hiệu quả của các phép toán phụ thuộc vào mức độ đầy $\alpha$. Khi cỡ mảng cố định, hiệu quả sẽ giảm nếu số dữ liệu N tăng lên. Vì vậy, trong thực hành thiết kế bảng băm, chúng ta cần đánh giá số tối đa các dữ liệu cần lưu để lựa chọn cỡ SIZE sao cho $\alpha$ đủ nhỏ. Mức độ đầy $\alpha$ không nên vượt quá 2/3.

Thời gian tìm kiếm cũng phụ thuộc sự tìm kiếm là thành công hay thất bại. Tìm kiếm thất bại đòi hỏi nhiều thời gian hơn tìm kiếm thành công, chẳng hạn trong bảng băm dây chuyền chúng ta phải xem xét toàn bộ một dây chuyền mới biết không có dữ liệu trong bảng.

D.E. Knuth (trong The art of computer programming, vol3) đã phân tích và đưa ra các công thức đánh giá hiệu quả cho từng phương pháp giải quyết va chạm như sau.

**Thời gian tìm kiếm trung bình trên bảng băm địa chỉ mở sử dụng thăm dò tuyến tính**. Số trung bình các lần thăm dò cho tìm kiếm xấp xỉ là:

Tìm kiếm thành công $\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$

Tìm kiếm thất bại $\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$

Trong đó $\alpha$ là mức độ đầy và $\alpha < 1$.

Ví dụ. Nếu cỡ bảng băm SIZE = 811, bảng chứa N = 649 dữ liệu, thì mức độ đầy là $\alpha = \frac{649}{811} \approx 80\%$. Khi đó, để tìm kiếm thành công một dữ liệu, trung bình chỉ đòi hỏi xem xét 3 vị trí mảng, vì

$$\frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) = \frac{1}{2}\left(1 + \frac{1}{1-0,8}\right) = 3$$

**Thời gian tìm kiếm trung bình trên bảng băm địa chỉ mở sử dụng thăm dò bình phương (hoặc băm kép)**. Số trung bình các lần thăm dò cho tìm kiếm được đánh giá là

Tìm kiếm thành công $\frac{-\ln(1-\alpha)}{\alpha}$

Tìm kiếm thất bại $\frac{1}{1-\alpha}$

Phương pháp thăm dò này đòi hỏi số lần thăm dò ít hơn phương pháp thăm dò tuyến tính. Chẳng hạn, giả sử bảng đầy tới 80%, để tìm kiếm thành công trung bình chỉ đòi hỏi xem xét 2 vị trí mảng,

$$\frac{-\ln(1-\alpha)}{\alpha} = \frac{-\ln(1-0,8)}{0,8} = \frac{1,6}{0,8} = 2$$

**Thời gian tìm kiếm trung bình trên bảng băm dây chuyền**. Trong bảng băm dây chuyền, để xen vào một dữ liệu mới, ta chỉ cần đặt dữ liệu vào đầu một dây chuyền được định vị bởi hàm băm. Do đó, thời gian xen vào là O(1).

Để tìm kiếm (hay loại bỏ) một dữ liệu, ta cần xem xét các tế bào trong một dây chuyền. Đương nhiên là dây chuyền càng ngắn thì tìm kiếm càng nhanh. Độ dài trung bình của một dây chuyền là $\frac{N}{SIZE} = \alpha$ (với giả thiết hàm băm phân phối đều).

Khi tìm kiếm thành công, chúng ta cần biết dây chuyền có rỗng không,

rồi cần xem xét trung bình là một nửa dây chuyền. Do đó, số trung bình các vị trí cần xem xét khi tìm kiếm thành công là

$1+\frac{\alpha}{2}$

Nếu tìm kiếm thất bại, có nghĩa là ta đã xem xét tất cả các tế bào trong một dây chuyền nhưng không thấy dữ liệu cần tìm, do đó số trung bình các vị trí cần xem xét khi tìm kiếm thất bại là $\alpha$.

Tóm lại, hiệu quả của phép toán tìm kiếm trên bảng băm dây chuyền là:

Tìm kiếm thành công $1+\frac{1}{\alpha}$

Tìm kiếm thất bại

| Mức độ đầy $\alpha$ | Bảng băm địa chỉ mở với thăm dò tuyến tính | Bảng băm địa chỉ mở với thăm dò bình phương | Bảng băm dây chuyền |
|---|---|---|---|
| 0,5 | 1,50 | 1,39 | 1,25 |
| 0,6 | 1,75 | 1,53 | 1,30 |
| 0,7 | 2,17 | 1,72 | 1,35 |
| 0,8 | 3,00 | 2,01 | 1,40 |
| 0,9 | 5,50 | 2,56 | 1,45 |
| 1,0 | | | 1,50 |
| 2,0 | | | 2,00 |
| 3,0 | | | 3,00 |

**Hình 9.6. Số trung bình các vị trí cần xem xét trong tìm kiếm thành công.**

Các con số trong bảng ở hình 9.6, và thực tiễn cũng chứng tỏ rằng, phương pháp băm là phương pháp rất hiệu quả để cài đặt từ điển.

## BÀI TẬP.

1. Hãy cài đặt hàm băm sử dụng phương pháp nhân (mục 9.2.2).

2. Hãy cài đặt hàm thăm dò sử dụng phương pháp băm kép.

3. Giả sử cỡ của bảng băm là SIZE = s và $d_1, d_2, \ldots, d_{s-1}$ là hoán vị ngẫu nhiên của các số 1, 2, …, s-1. Dãy thăm dò ứng với khoá k được xác định như sau:
   $i_0 = i = h(k)$
   $i_m = (i + d_i)$ % SIZE , $1 \le m \le s - 1$
   Hãy cài đặt hàm thăm dò theo phương pháp trên.

4. Cho cỡ bảng băm SIZE = 11. Từ bảng băm rỗng, sử dụng hàm băm chia lấy dư, hãy đưa lần lượt các dữ liệu với khoá:
   32 , 15 , 25 , 44 , 36 , 21
   vào bảng băm và đưa ra bảng băm kết quả trong các trường hợp sau:
   a. Bảng băm được chỉ mở với thăm dò tuyến tính.
   b. Bảng băm được chỉ mở với thăm dò bình phương.
   c. Bảng băm dây chuyền.

5. Từ các bảng băm kết quả trong bài tập 4, hãy loại bỏ dữ liệu với khoá là 44 rồi sau đó xen vào dữ liệu với khoá là 65.

6. Bảng băm chỉ cho phép thực hiện hiệu quả các phép toán tập động nào? Không thích hợp cho các phép toán tập động nào? Hãy giải thích tại sao?

7. Giả sử khoá tìm kiếm là từ tiếng Anh. Hãy đưa ra ít nhất 3 cách thiết kế hàm băm. Bình luận về các cách thiết kế đó theo các tiêu chuẩn hàm băm tốt.