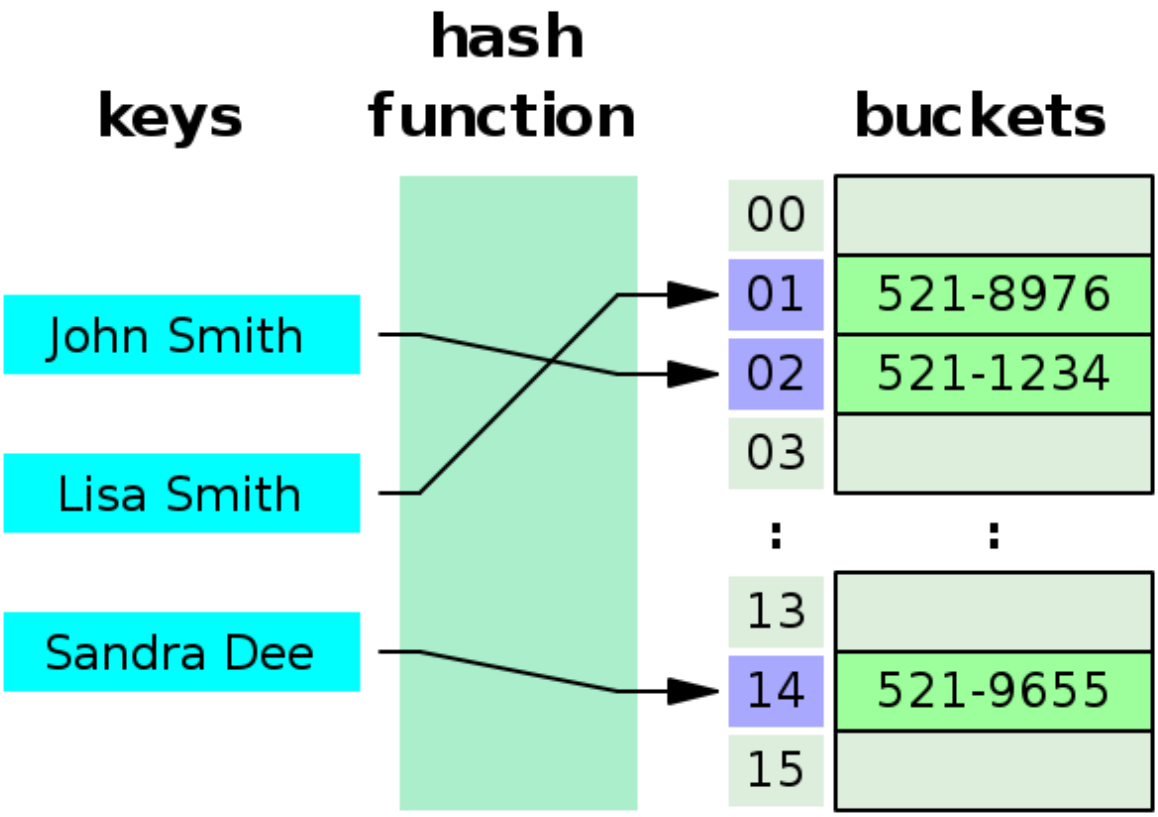
Bảng băm – Hash tables

Trong khoa học máy tính, bảng băm(Hash Tables) là một cấu trúc dữ liệu sử dụng hàm băm để ánh xạ từ giá trị xác định, được gọi là khóa (ví dụ như tên của một người), đến giá trị tương ứng (ví dụ như số điện thoại của họ). Do đó, bảng băm là một mảng kết hợp. Hàm băm được sử dụng để chuyển đổi từ khóa thành chỉ số (giá trị băm) trong mảng lưu trữ các giá trị tìm kiếm.

1. Lý thuyết về bảng băm

Băm là một kỹ thuật được sử dụng để định danh một đối tượng cụ thể trong một nhóm các đối tượng tương tự. Một số ví dụ về việc sử dụng bảng băm trên thực tế:

- Trong trường đại học, mỗi sinh viên được chỉ định một mã sinh viên không giống nhau và qua mã sinh viên đó có thể truy xuất các thông tin của sinh viên đó.
- Trong thư viện, mỗi một cuốn sách một mã số riêng và mã số đó có thể được dùng để xác định các thông tin của sách, chẳng hạn như vị trí chính xác của sách trong thư viện hay thể loại của sách đó,…



Mô phỏng bảng băm

Trong cả 2 ví dụ trên, các sinh viên và các cuốn sách được "băm" thành các mã số duy nhất(không trùng lặp).

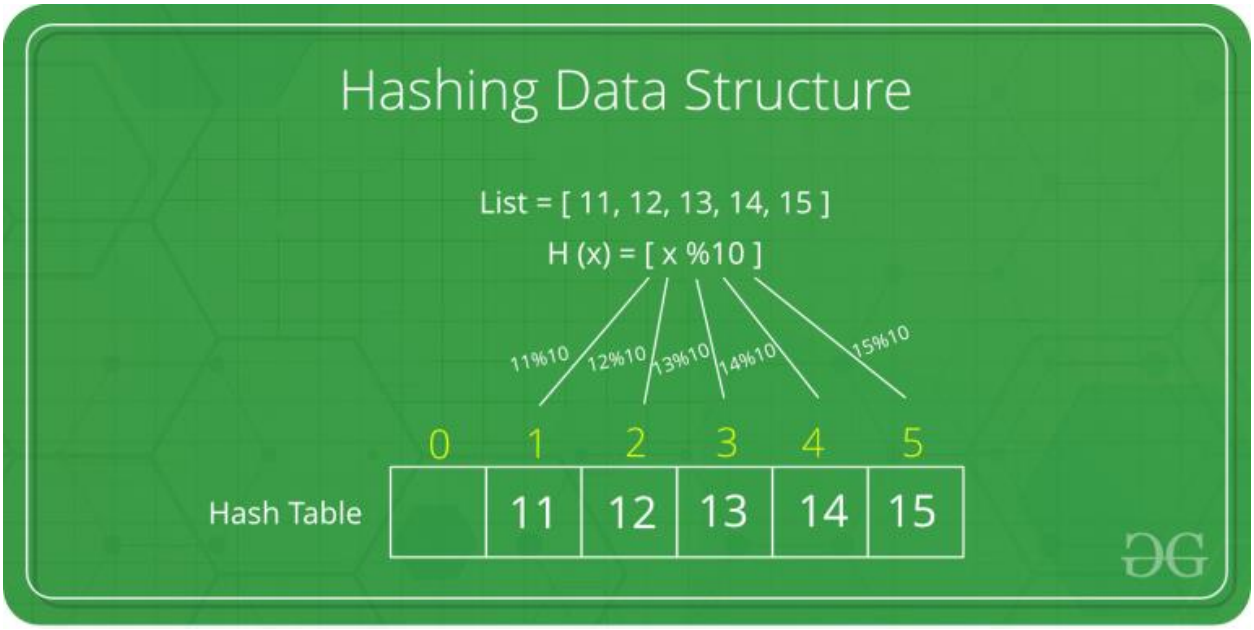
Giả sử rằng bạn có một đối tượng và bạn muốn gán cho nó một cái khóa(key) để giúp tìm kiếm dễ dàng hơn. Để lưu giữ cặp <khóa, giá trị>(nó thường được gọi là <key, value> hơn), bạn có thể sử dụng mảng bình thường để làm việc này; Với chỉ số mảng là khóa và giá trị tại chỉ số đó là giá trị tương ứng của khóa. Tuy nhiên, trong trường hợp phạm vi của khóa lớn và không thể sử dụng chỉ số mảng được, khi đó bạn sẽ cần tới "băm"(hashing).

Trong hashing, các key có giá trị lớn sẽ được đưa về giá trị nhỏ hơn bằng cách sử dụng **hàm băm(hash functions)**. Các giá trị sau đó được lưu trong một cấu trúc dữ liệu gọi là **bảng băm(hash tables)**. Ý tưởng của hashing là đưa các cặp <key, value> về một mảng thống nhất; Mỗi phần tử sẽ được gán một khóa định danh(khóa có được sau khi dùng hàm băm). Bằng việc sử dụng khóa định danh đó, chúng ta có thể truy cập trực tiếp tới nó với độ phức tạp O(1). Thuật toán băm sẽ sử dụng **khóa băm** để tính toán ra khóa định danh của các phần tử hoặc thêm vào bảng băm.

Việc hashing được thực hiện qua 2 bước:

1. Một phần tử sẽ được chuyển đổi thành 1 số nguyên bằng việc sử dụng hàm băm. Phần tử này được sử dụng như một chỉ mục để lưu trữ phần tử gốc và nó sẽ được đưa vào bảng băm.
2. Phần tử sẽ được lưu giữ trong bảng băm, nó có thể được truy xuất nhanh bằng khóa băm:
    - hash = hashfunc(key)
    - index = hash % array_size

Theo cách trên thì việc băm sẽ phụ thuộc vào kích thước mảng `array_size`. Chỉ số index sau đó được đưa về [0; array_size-1] bằng việc sử dụng toán tử chia lấy dư `%`.



Cách hàm băm hoạt động

2. Hàm băm

Hàm băm là bất kỳ hàm nào có thể được sử dụng để ánh xạ tập dữ liệu có kích thước tùy ý thành tập dữ liệu có kích thước cố định và đưa vào bảng băm. Các giá trị được trả về bởi hàm băm được gọi là giá trị băm.

Một hàm băm được đánh giá tốt nếu nó đạt được các yêu cầu cơ bản sau:

1. Dễ tính toán: Nó phải dễ tính toán và bản thân nó không phải là một thuật toán
2. Phân bố đồng đều: Nó cần phải phân phối đồng đều trên bảng băm, không xảy ra việc tập trung thành các cụm
3. Ít va chạm: Va chạm xảy ra khi các cặp phần tử được ánh xạ tới cùng một giá trị băm

Chú ý: Bất kể hàm băm có tốt đên đâu, va chạm vẫn có thể xảy ra. Vì vậy, để duy trì hiệu suất của bảng băm, điều quan trọng là phải quản lý va chạm thông qua *các kỹ thuật giải quyết va chạm*.

Tại sao phải cần hàm băm đủ tốt?

Hãy đi vào một ví dụ để dễ hình dung nhé. Giả sử rằng bạn cần lưu giữ các string sau đây trong bảng băm: {“abcdef”, “bcdefa”, “cdefab” , “defabc” }.

Để tính toán chỉ mục lưu giữ các string này, chúng ta dùng một hàm băm như sau: Chỉ mục của mỗi string sẽ được tính bằng tổng giá trị ASCII của các ký tự trong nó sau đó lấy dư với 599.

Do 5là số nguyên tố và nó lớn hơn số lượng phần tử, nó có khả năng lập ra các chỉ mục khác nhau(giảm va chạm). Số nguyên tố luôn là lựa chọn tốt nếu bạn muốn dùng phép lấy phần dư. Các giá trị ASCII của a, b, c, d, e và f lần lượt là 97, 98, 99, 100, 101 và 102.

Dễ nhận thấy, các string kia chỉ là các hoán vị của cùng 1 chuỗi, do đó chỉ mục được tạo ra của chúng đều giống nhau và đều bằng 597 % 5 = 2.

Lúc này, hàm băm sẽ tính toán và tất cả các string kia đều có cùng 1 chỉ mục. Khi đó, bạn có thể tạo 1 list tại chỉ số đó để lưu các string trong list đó. Như hình ảnh minh họa này:

Index				
0				
1				
2	abcdef	bcdefa	cdefab	defabc
3				
4				
-				
-				
-				
-				

Bạn thấy đó, bạn vẫn sẽ mất O(n) để tìm kiếm một string cụ thể. Điều đó có nghĩa hàm băm này chưa thực sự tốt.

Hãy thử với 1 hàm băm khác nhé. Chỉ mục của các string sẽ được tính bằng tổng của các mã ASCII của từng ký tự theo sau là vị trí của ký tự đó trong string. Sau đó đem chia dư cho số nguyên tố 2069.

String	Hash function	Index
abcdef	(97*1* + 98*2* + 99*3* + *100*4 + 101*5* + *102*6)%2069	38
bcdefa	(98*1* + 99*2* + 100*3* + *101*4 + 102*5* + *97*6)%2069	23
cdefab	(99*1* + *100*2 + 101*3* + *102*4 + 97*5* + *98*6)%2069	14
defabc	(100*1* + *101*2 + 102*3* + *97*4 + 98*5* + *99*6)%2069	11

Lúc này, các chỉ mục của mỗi string là không trùng lặp(không va chạm).

Hash Table

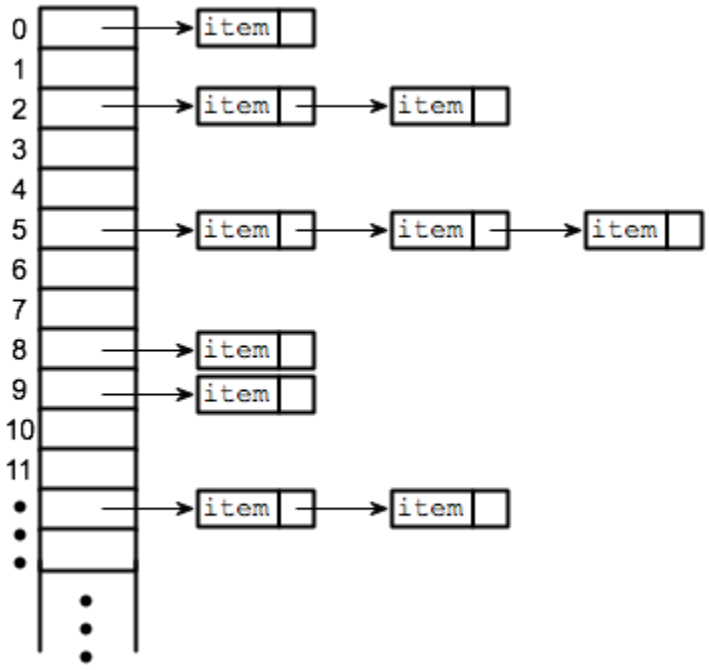
Here all strings are stored at different indices

Index	
0	
1	
-	
-	
-	
11	defabc
12	
13	
14	cdefab
-	
-	
-	
-	
23	bcdefa
-	
-	
-	
38	abcdef
-	
-	

3. Kỹ thuật xử lý va chạm

3.1. Separate chaining (open hashing)

Separate chaining là một kỹ thuật xử lý và chạm phổ biến nhất. Nó thường được cài đặt với danh sách liên kết. Để lưu giữ một phần tử trong bảng băm, bạn phải thêm nó vào một danh sách liên kết ứng với chỉ mục của nó. Nếu có sự va chạm xảy ra, các phần tử đó sẽ nằm cùng trong 1 danh sách liên kết(xem ảnh để hiểu hơn).



Như vậy, kỹ thuật này sẽ đạt được tốc độ tìm kiếm O(1) trong trường hợp tối ưu và O(N) nếu tất cả các phần tử ở cùng 1 danh sách liên kết duy nhất. Đó là do có điều kiện 3 trong tiêu chí hàm băm tốt.

Cài đặt bảng băm sử dụng Separate chaining

Giả định: Hàm băm sẽ trả về số int trong [0, 19].

```
vector <string> hashTable[20];
int hashTableSize=20;
```

Thêm vào bảng băm

```
void insert(string s)
{
    // Compute the index using Hash Function
    int index = hashFunc(s);
    // Insert the element in the linked list at the particular index
    hashTable[index].push_back(s);
}
```

Tìm kiếm

```
void search(string s)
{
    //Compute the index by using the hash function
    int index = hashFunc(s);
    //Search the linked list at that specific index
    for(int i = 0;i < hashTable[index].size();i++)
    {
        if(hashTable[index][i] == s)
        {
            cout << s << " is found!" << endl;
            return;
        }
    }
    cout << s << " is not found!" << endl;
}
```

3.2. Linear probing (open addressing or closed hashing)

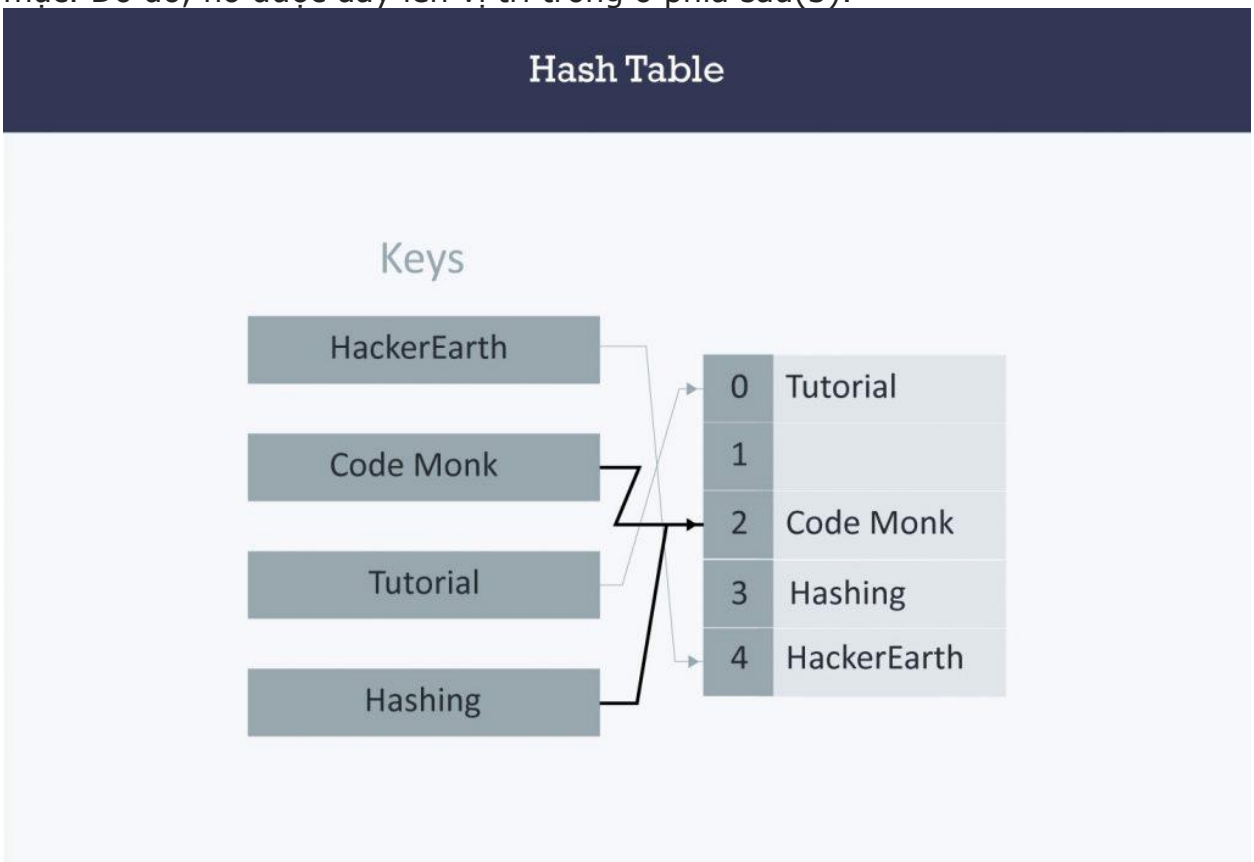
Trong kỹ thuật xử lý va chạm này, chúng ta sẽ không dùng linklist để lưu trữ mà chỉ có bản thân array đó thôi.

Khi thêm vào bảng băm, nếu chỉ mục đó đã có phần tử rồi; Giá trị chỉ mục sẽ được tính toán lại theo cơ chế tuần tự. Giả sử rằng chỉ mục là chỉ số của mảng, khi đó, việc tính toán chỉ mục cho phần tử được tính theo cách sau:

index = index % hashTableSize
index = (index + 1) % hashTableSize
index = (index + 2) % hashTableSize
index = (index + 3) % hashTableSize

Và cứ thế theo cách như vậy chừng nào index thu được chưa có phần tử được sử dụng. Tất nhiên, không gian chỉ mục phải được đảm bảo để luôn có chỗ cho phần tử mới.

Như trong ví dụ dưới đây, chuỗi Hashing bị trùng chỉ mục(2) ở lần đầu tính chỉ mục. Do đó, nó được đẩy lên vị trí trống ở phía sau(3).



Cài đặt bảng băm dùng Linear probing

Giả sử rằng:

- Không có nhiều hơn 20 phần tử trong tập dữ liệu
- Hàm băm sẽ trả về một số nguyên từ 0 đến 19
- Tập dữ liệu phải là các phần tử duy nhất

```
string hashTable[21];
int hashTableSize = 21;
```

Thêm vào bảng băm

```
void insert(string s)
{
    //Compute the index using the hash function
    int index = hashFunc(s);
    //Search for an unused slot and if the index will exceed the hashTableSize then roll back
    while(hashTable[index] != "")
        index = (index + 1) % hashTableSize;
    hashTable[index] = s;
}
```

Tìm kiếm

```
void search(string s)
{
    //Compute the index using the hash function
    int index = hashFunc(s);
     //Search for an unused slot and if the index will exceed the hashTableSize then roll ba
    while(hashTable[index] != s and hashTable[index] != "")
        index = (index + 1) % hashTableSize;
    //Check if the element is present in the hash table
    if(hashTable[index] == s)
        cout << s << " is found!" << endl;
    else
        cout << s << " is not found!" << endl;
}
```

3.3. Quadratic Probing

Ý tưởng cũng khá giống Linear Probing, nhưng cách tính chỉ mục có khác đôi chút:

index = index % hashTableSize
index = (index + 1^2) % hashTableSize
index = (index + 2^2) % hashTableSize
index = (index + 3^2) % hashTableSize

Và cứ thế cho tới khi tìm được chỉ mục trống.

Cài đặt bảng băm dùng Quadratic Probing

Giả sử rằng:

- Không có nhiều hơn 20 phần tử trong tập dữ liệu
- Hàm băm sẽ trả về một số nguyên từ 0 đến 19
- Tập dữ liệu phải là các phần tử duy nhất

```
string hashTable[21];
int hashTableSize = 21;
```

Thêm phần tử

```
void insert(string s)
{
    //Compute the index using the hash function
    int index = hashFunc(s);
    //Search for an unused slot and if the index will exceed the hashTableSize roll back
    int h = 1;
    while(hashTable[index] != "")
    {
        index = (index + h*h) % hashTableSize;
            h++;
    }
    hashTable[index] = s;
}
```

Tìm kiếm phần tử

```
void search(string s)
{
    //Compute the index using the Hash Function
    int index = hashFunc(s);
     //Search for an unused slot and if the index will exceed the hashTableSize roll back
   int h = 1;
    while(hashTable[index] != s and hashTable[index] != "")
    {
        index = (index + h*h) % hashTableSize;
            h++;
    }
    //Is the element present in the hash table
    if(hashTable[index] == s)
        cout << s << " is found!" << endl;
    else
        cout << s << " is not found!" << endl;
}
```

3.4. Double hashing

Vẫn giống 2 kỹ thuật ngay phía trên, chỉ khác ở công thức tính khi xảy ra va chạm như sau:

index = (index + 1 * indexH) % hashTableSize;
index = (index + 2 * indexH) % hashTableSize;

Và cứ tiếp tục cho tới khi tìm được chỉ mục chưa được sử dụng.

Cài đặt bảng băm dùng Double hashing

Giả sử rằng:

- Không có nhiều hơn 20 phần tử trong tập dữ liệu
- Hàm băm sẽ trả về một số nguyên từ 0 đến 19
- Tập dữ liệu phải là các phần tử duy nhất

```
string hashTable[21];
int hashTableSize = 21;
```

Thêm vào bảng băm

```
void insert(string s)
{
    //Compute the index using the hash function1
    int index = hashFunc1(s);
    int indexH = hashFunc2(s);
    //Search for an unused slot and if the index exceeds the hashTableSize roll back
    while(hashTable[index] != "")
        index = (index + indexH) % hashTableSize;
    hashTable[index] = s;
}
```

Tìm kiếm trên bảng băm

```
void search(string s)
{
    //Compute the index using the hash function
    int index = hashFunc1(s);
    int indexH = hashFunc2(s);
     //Search for an unused slot and if the index exceeds the hashTableSize roll back
    while(hashTable[index] != s and hashTable[index] != "")
        index = (index + indexH) % hashTableSize;
    //Is the element present in the hash table
    if(hashTable[index] == s)
        cout << s << " is found!" << endl;
    else
        cout << s << " is not found!" << endl;
}
```

4. Ứng dụng của bảng băm

Trong các bài toán thông thường, các bạn thường chỉ cần sử dụng cấu trúc dữ liệu được cài đặt sẵn ở các ngôn ngữ lập trình: map, set trong C/C++, Java; Dictionary trong C#, Python. Đó chính là các bảng băm cực kỳ hữu dụng mà chúng ta vẫn hay sử dụng.

Nếu các bạn để ý thì **multimap** và **multiset** trong C++ cho phép lưu trữ key trùng nhau với giá trị khác nhau đó. Đó là 2 thằng đã được thiết lập cơ chế xử lý va chạm.

Với các bài toán đặc thù, các bạn sẽ phải tự viết cho mình hàm băm và xây dựng cấu trúc dữ liệu bảng băm cho phù hợp. Dưới đây là một số ứng dụng của bảng băm:

- *Associative arrays:* Bảng băm thường được sử dụng để cài đặt nhiều loại in-memory tables. Chúng được sử dụng để thực hiện các mảng kết hợp (các mảng có chỉ số là các chuỗi tùy ý hoặc các đối tượng phức tạp khác).
- Lập chỉ mục CSDL: Các bảng băm cũng có thể được sử dụng làm cấu trúc dữ liệu để lập chỉ mục dữ liệu trong các CSDL.
- Caches: Bảng băm dùng để thiết lập cơ chế caches, thường được dùng để tăng tốc độ truy cập dữ liệu.
- Biểu diễn các đối tượng: Một số ngôn ngữ động, chẳng hạn như Perl, Python, JavaScript và Ruby sử dụng bảng băm để triển khai các đối tượng.
- Hàm băm được sử dụng trong các thuật toán khác nhau để tăng tốc độ tính toán.