Bảng băm – Hash table

Hashing, hash table, separate chaining, open addressing

Nội dung

- Bảng băm
- Các kỹ thuật thiết kế hàm băm
- Các phương án xử lý đụng độ
- Băm kép
- Băm lại
- So sánh bảng băm và cây tìm kiếm

Khái niệm cơ bản

Bảng băm

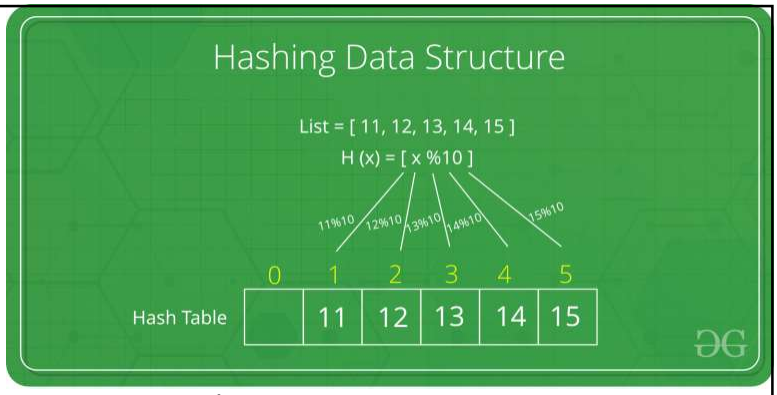
- Bài toán lưu trữ và tìm kiếm trên danh sách
  - Tháo tác tìm kiếm phần tử dựa trên các thông tin liên quan
  - Thao tác thêm phần tử mới
  - Thao tác xóa phần tử
- Cần tổ chức dữ liệu và thuật toán sao cho hiệu quả
  - Danh sách liên kết/mảng với tìm kiếm tuần tự: O(n)/O(n)
  - Mảng với tìm kiếm nhị phân: O(n)/O(logn)
  - Cây nhị phân tìm kiếm cân bằng: O(logn)/O(logn)
- Cần thực hiện các thao tác nhanh hơn?
  - Truy cập trực tiếp các phần tử dựa trên bản thân giá trị khóa
  - Không cần xét mối quan hệ giữa khóa tìm kiếm với giá trị khóa các phần tử

implementation	guarantee			average case			ordered iteration?
	search	insert	delete	search	insert	delete	
unordered array	N	N	N	N/2	N/2	N/2	no
ordered array	lg N	N	N	lg N	N/2	N/2	yes
unordered list	N	N	N	N/2	N	N/2	no
ordered list	N	N	N	N/2	N/2	N/2	yes
BST	N	N	N	1.39 lg N	1.39 lg N	?	yes
randomized BST	7 lg N	7 lg N	7 lg N	1.39 lg N	1.39 lg N	1.39 lg N	yes
red-black tree	3 lg N	3 lg N	3 lg N	lg N	lg N	lg N	yes

Bảng băm

- **Ý tưởng bảng truy cập trực tiếp – direct access table:**
  - Dùng mảng kích thước lớn (bảng)
  - Mỗi phần tử sẽ được lưu tại một ô duy nhất dựa vào giá trị khóa
  - Nếu khóa không tồn tại thì ô tương ứng sẽ là rỗng - NULL
  - Khi tìm kiếm thì căn cứ vào khóa cần tìm ta tìm đến ô tương ứng
  - Nếu ô đó có chứa phần tử thì tìm thấy, ngược lại là không tìm thấy
- Nhận xét
  - Thời gian xử lý (tìm kiếm, thêm, xóa): O(1)
  - Cần mảng kích thước rất lớn.
    VD. lưu trữ danh sách sinh viên có khóa là số SHSV dạng XXXX XXXX thì cần mảng kích thước 10^8
  - Khó xử lý nếu khóa là dạng số thực hoặc xâu ký tự

Bảng băm



- Bảng băm: Ý tưởng cũng tương tự bảng truy cập trực tiếp, tuy nhiên dùng thêm 1 hàm băm
- Hàm băm
  - Chuyển giá trị khóa ban đầu về một tập giá trị số nguyên nhỏ hơn
  - Có thể nhận đầu vào khóa kiểu số thực, xâu ký tự, object,…
  - Thời gian tìm kiếm sẽ phụ thuộc vào hàm băm
  - Thời gian có thể O(1) tới O(n)
- Chọn hàm băm đủ tốt
  - Thời gian tính toán đủ nhanh
  - Có phân phối đều với giá trị khóa

Bảng băm

- VD. Sinhvien(1222, ‘Nguyễn văn A’, ‘Hà Nội’)
  Sinhvien(1101,’Trần văn C’,’Thái Bình’)
  Sinhvien(0097, ‘Nguyễn Thi D’, ‘Hà Nội’)
  Sinhvien(1331,’Trần văn C’,’Thái Bình’)
  Sinhvien(1345, ‘Nguyễn văn A’, ‘Hà Nội’)
- Sử dụng bảng kích thước 100 để lưu các phần tử

Hàm băm
h = k%100

Chỉ số mảng

1	**Sinhvien(1101,’Trần văn C’,’Thái Bình’)**
…	
22	**Sinhvien(1222, ‘Nguyễn văn A’, ‘Hà Nội’)**
…	
31	**Sinhvien(1331,’Trần văn C’,’Thái Bình’)**
…	
45	**Sinhvien(1345, ‘Nguyễn văn A’, ‘Hà Nội’)**
…	
97	**Sinhvien(0097, ‘Nguyễn thi D’, ‘Hà Nội’)**

Bảng băm

- Tìm kiếm:
  - Tính địa chỉ ô lưu trữ tương ứng với giá trị của khóa qua hàm băm
  - Tìm tại ô lưu trữ xem có phần tử hay là rỗng (NULL)
- Thêm/xóa
  - Tính địa chỉ ô lưu trữ thông qua hàm băm (dựa trên giá trị của khóa)
  - Thêm/xóa phần tử tại ô tương ứng
- Hàm băm sẽ ảnh hưởng lớn đến hiệu quả của bảng băm
  - Hàm băm hoàn hảo: không đụng độ giữa các khóa→ thời gian O(1) ?
  - Hàm băm đủ tốt
    - Tính toán đủ nhanh
    - Phân phối đều các khóa
    - Ít đụng độ

T(n)= T(tính hàm băm) + T(tìm kiếm do đụng độ)

  - Mỗi kiểu khóa lại khác nhau ➔ cần thiết kế hàm băm tùy vào trường hợp cụ thể

Bảng băm

- **Đụng độ**? hai khóa khác nhau lại cùng băm ra một giá trị chỉ số

  VD: hai khóa 21296876, 11391176

  Trong hàm băm dùng phương pháp cắt bỏ
- Không gian giá trị khóa gần như vô hạn, trong khi địa chỉ lưu trữ lại giới hạn → **không thể tránh khỏi đụng độ**
- Giải pháp xử lý đụng độ:
  - Phương pháp đánh địa chỉ mở (Open Addressing): lưu trữ tại các ô lân cận khi đụng độ
    - Phương pháp dò
    - Băm kép (double hashing): Dùng hàm băm thứ 2 khi hàm băm thứ nhất bị đụng độ
  - Phương pháp đánh địa chỉ đóng: xích ngăn cách (Separate Chaining): lưu trữ các khóa bị đụng độ trong một danh sách

Hàm băm lưu trữ KHÁC hoàn toàn mục đích của hàm băm bảo mật (md4, md5, SHA256,..)

Một số kỹ thuật thiết kế hàm băm

Một số kỹ thuật thiết kế hàm băm

- **Cắt bỏ (Truncation)** : dùng một phần của khóa làm chỉ số

  VD: khóa có 8 chữ số và bảng băm kích thước 1000
  lấy chữ số thứ 4, 7 và 8 làm chỉ số
  21296876→ 976
  Nhanh nhưng phân bố khóa không đều!

- **Gấp (folding)**: chia khóa thành nhiều phần sau đó kết hợp các phần lại (thường dùng cộng hoặc nhân)

  VD. 21296876 chia thành 3 phần 212, 968 và 76
  kết hợp 212+968+76 = 1256, cắt bỏ được 256

  Giá trị các thành phần trong khóa đều ảnh hưởng tới chỉ số. Cho phân phối tốt hơn phương pháp cắt bỏ

Một số kỹ thuật thiết kế hàm băm

- **Phương pháp chia module**: lấy số dư của phép chia giá trị khóa cho kích thước của bảng băm để làm địa chỉ $h(k) = k \% m$
- Thường chọn m là số nguyên tố nhỏ hơn, gần với kích thước bảng băm.
  - Bảng băm kích thước 1000 thì chọn m=997
- **Phương pháp nhân**: giá trị khóa được nhân với chính nó, sau đó lấy một phần kết quả để làm địa chỉ băm

Giá trị khóa	địa chỉ
5402	417
367	367
1246	249
2983	989

Giá trị khóa k	k*k	địa chỉ
5402	29181604	181
367	00134689	134
1246	01552516	552
2983	08898289	898

Một số kỹ thuật thiết kế hàm băm

- Khóa kiểu số thực? Chuyển về số nguyên
- Khóa là kiểu ký tự? dồn ký tự (accumulation) hoặc hàm băm DJB2 để chuyển về số nguyên
  - Dồn ký tự (accumulation) khi khóa không quá dài: VD. Khóa "ABC"
    - Cộng mã ASCII: "ABC" ➔ 65+66+67 = 198
      Khá đơn giản và không chia đều các khóa
    - Nhân theo vị trí và kích thước từ điển: "ABC" với từ điển kích thước 128 (nửa đầu bảng mã ascii)
      "ABC" ➔ $65*128^2 + 66*128^1 + 67*128^0 = 1073475$
      Tính toán mất thời gian và dễ tràn số nếu kích thước từ điển lớn hoặc khóa dài ➔ kết hợp thêm chia module %M
  - Khóa dài: VD. Đoạn văn?
    - Lấy 1 phần ở 1 số vị trí (đầu, giữa, cuối,…)
- Khóa là kiểu object: File ảnh, audio, video…
  - Cần có thuật toán chuyển đổi riêng tùy từng trường hợp cụ thể

Một số kỹ thuật thiết kế hàm băm

Với khóa là số thực

```
#define TABLE_SIZE 10
#define SCALE_FACTOR 1000

// Hàm băm khóa là số thực
int hashFunction(double key) {
    int intKey = (int)(key * SCALE_FACTOR);
    return abs(intKey % TABLE_SIZE);
}
```

```
// Hàm băm (sử dụng thuật toán DJB2)
unsigned long hashFunction(const char* str) {
    unsigned long hash = 5381; // Giá trị khởi tạo
    int c;
    while ((c = *str++)) {
        hash = ((hash << 5) + hash) + c; // hash * 33 + c
    }
    return hash % TABLE_SIZE;
}
```

Với khóa là xâu ký tự với thuật toán DJB2
hash(i) = hash(i - 1) * 33 + str[i].

Một số kỹ thuật thiết kế hàm băm

- Một số hàm băm thực tế trên xâu ký tự
  - Dồn ký tự (Accumulation): $hash\_value = \sum_{i=0}^{n-1} ASCII(str[i])$
  - Hàm băm DJB2 (do Daniel J. Bernstein đề xuất):

```
hash = 5381
hash = ((hash≪5)+hash) + ASCII(str[i])
(Tương đương hash = hash×33 + ASCII(str[i]))
```

  - Hàm băm SDBM (dùng trong cơ sở dữ liệu **SDBM (Simple Database Manager)**)

```
hash = 0
hash = ASCII(str[i]) + (hash≪6)+(hash≪16)−hash
```

  - Hàm băm BKDR (Brian Kernighan and Dennis Ritchie)

```
hash = 0
hash = hash×seed + ASCII(str[i])
Thường dùng seed = 31, 131, 1313, 13131, ...
```

Một số kỹ thuật thiết kế hàm băm

- Một số hàm băm thực tế trên xâu ký tự
  - Hàm băm Fowler–Noll–Vo (FNV)
  - Hàm băm CRC32 (Cyclic Redundancy Check)
  - Hàm băm MurmurHash

Công thức (FNV-1a):

$$\text{hash} = \text{offset\_basis}$$
$$\text{hash} = (\text{hash} \oplus \text{ASCII}(\text{str}[i])) \times \text{FNV\_prime}$$

Giá trị chuẩn:

- $\text{offset\_basis} = 2166136261$
- $\text{FNV\_prime} = 16777619$

Thuật toán	Độ phức tạp	Tốc độ	Phân phối	Độ khó cài đặt
Dồn ký tự	$O(n)$	Nhanh	Kém	Dễ
DJB2	$O(n)$	Trung bình	Tốt	Dễ
SDBM	$O(n)$	Trung bình	Rất tốt	Vừa phải
BKDR	$O(n)$	Nhanh	Tốt	Dễ
FNV	$O(n)$	Trung bình	Rất tốt	Vừa phải
CRC32	$O(n)$	Chậm	Rất tốt	Khó
MurmurHash	$O(n)$	Rất nhanh	Xuất sắc	Khó

Lựa chọn thuật toán phù hợp
1.Ứng dụng nhỏ:
1. Dùng **dồn ký tự**, **BKDR**, hoặc **DJB2**.

2.Ứng dụng lớn, cần hiệu quả cao:
1. Dùng **MurmurHash** hoặc **FNV**.

3.Tương thích giao thức mạng:
1. Dùng **CRC32**.

Một số kỹ thuật thiết kế hàm băm

- Ví dụ 1. Hàm băm tương ứng của số có 3 chữ số (a1 a2 a3) là **mod** (a1 + a2 + a3 , 5). Với các tổ hợp số dưới đây, giá trị băm bị xung đột là trường hợp nào?
  A. 881 và 509
  B. 913 và 426
  C. 731 và 429
  D. 102 và 297
  E. 677 và 388

mod: chia lấy phần dư mod(5,3) = 2

Một số kỹ thuật thiết kế hàm băm

- Ví dụ 2. Thiết kế hàm băm cho khóa là các SDT
  - điện thoại cố định: 02xx xxx xxxx,
  - điện thoai di động 09x-xxxxxxx hoặc dạng đầu số mới: 03x, 05x, 07x, 08x (0xxx xxx xxx (mobile)) VD. 098 932 3880, 096 543 2999
- Cách tạo? Chia thành 3 nhóm số A: 4 số đầu, B và C là nhóm 3 số tiếp
  - H = A+B+C % M có đủ tốt
  - Nhân thêm hệ số H = A*a1+B*a2+C % M với a1 và a2 là 2 số ngẫu nhiên (số nguyên tố)
  - Phương án khác?
    - Cộng dồn, CRC32, …


```
int hashPhoneNumber(const char* phone, int table_size) {
    int hash = 0;
    for (int i = 0; phone[i] != '\0'; i++) {
        hash += phone[i] - '0'; // Chuyển ký tự sang số
    }
    return hash % table_size;
}
```

Phương pháp cộng dồn

```
int hashPhoneNumber(const char* phone, int table_size) {
    unsigned long long num = 0;
    for (int i = 0; phone[i] != '\0'; i++) {
        num = num * 10 + (phone[i] - '0'); // Chuyển thành số nguyên lớn
    }
    double A = 0.6180339887; // Hằng số vàng
    double fractional_part = (num * A) - (unsigned long long)(num * A);
    return (int)(fractional_part * table_size);
}
```

Phương pháp nhân

```c
int hashPhoneNumber(const char* phone, int table_size) {
    unsigned long hash = 5381;
    for (int i = 0; phone[i] != '\0'; i++) {
        hash = ((hash << 5) + hash) + (phone[i] - '0'); // hash * 33 + digit
    }
    return hash % table_size;
}
```

Hàm băm DJB2

```c
int hashPhoneNumber(const char* phone, int table_size) {
    unsigned long hash = 2166136261;
    for (int i = 0; phone[i] != '\0'; i++) {
        hash = (hash ^ (phone[i] - '0')) * 16777619;
    }
    return hash % table_size;
}
```

Hàm băm Fowler–Noll–Vo (FNV)

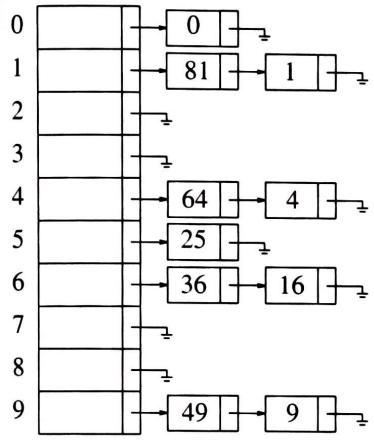
Một số kỹ thuật thiết kế hàm băm

- Ví dụ 3. Thiết kế hàm băm cho khóa là các từ tiếng anh được tách từ bài báo
  VD. "*Facebook CEO Mark Zuckerberg jumped into fourth place, his highest rank ever, with a net worth of $55.5 billion. However, Oracle founder Larry Ellison landed at No. 5 for the first time since 2007. His net worth is $49.3 billion.*"
- Vấn đề
  - Độ dài tối đa của khóa. VD. 30 ký tự
  - Kích thước từ điển: 26, 36, 54, 64, 128 (số lượng ký tự có nghĩa trong hàm băm)
  - Xử lý với trường hợp ngoại lệ.
    VD. URL
    https://www.geeksforgeeks.org/c-program-hashing-chaining/
    https://www.geeksforgeeks.org/hashing-set-2-separate-chaining/
    https://www.geeksforgeeks.org/c-program-swap-two-numbers/?ref=leftbar-rightbar

Xử lý xung đột

Xử lý đụng độ

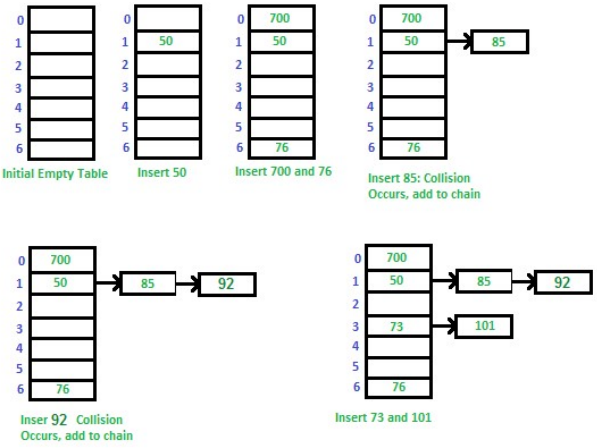
- Đánh địa chỉ đóng – **Phương pháp xích ngăn cách (Separate Chaining)**
  - Dùng mảng của (danh sách móc nối/ cây nhị phân tìm kiếm/mảng động,…)
- **Ưu điểm**:
  - Cài đặt đơn giản
  - không bị giới hạn số phần tử,
  - Dễ thêm và xóa
  - xử lý đụng độ tốt, không bị quá phụ thuộc vào hàm băm hoặc tỉ lệ nạp của bản – load factor
  - Thường được áp dụng khi không biết số lượng khóa hoặc phân phối tần số các khóa
- **Nhược điểm** :
  - Phải lưu nhiều con trỏ NULL
  - Thực hiện tìm kiếm tuyến tính nên thời gian tìm kiếm chậm (nếu đụng độ diễn ra nhiều)
  - Khó thực hiện cache dữ liệu của bảng băm



Hàm băm h=k%10

Separate Chaining

- Bảng băm dùng đánh địa chỉ đóng:
  - Kích thước n, số phần tử hiện có trong bảng băm là m
  - Hệ số nạp – load factor $\alpha = m/n$
  - Thời gian thêm: O(1)
  - Thời gian xóa: O(1+ α)
  - Thời gian tìm kiếm: O(1+ α)
  - Nếu α = O(1) thì thời gian các thao tác sẽ là O(1)



Hàm băm h=k%7

Separate Chaining

- Cài đặt bảng băm với các cách lưu trữ chainning khác nhau

Dùng danh sách liên kết	Dùng mảng cấp phát động (mảng kích thước biến đổi)	Dùng cây tìm kiếm cân bằng (AVL, RB tree)
Search: O(L) where L = length of linked list Delete: O(L) Insert: O(1) Not cache friendly	Search: O(L) where L = length of array Delete: O(1) → Mảng không thứ tự Insert: O(1) Cache friendly Mảng phải co/giãn nên thời gian tồi nhất là O(L) Mảng có thứ tự + tìm kiếm nhị phân Search: O(logL) where L = length of array Delete: O(L) Insert: O(L)	Search: O(log(L)) Delete: O(log(L)) Insert: O(log(L)) Not cache friendly Java 8 dùng cách này cho HashMap L là số phần tử trên cây

```c
// Cấu trúc nút trong danh sách liên kết
typedef struct Node {
    double key;           // Khóa (số thực)
    struct Node* next;    // Liên kết đến nút tiếp theo
} Node;

// Bảng băm là mảng các danh sách liên kết
Node* hashTable[TABLE_SIZE];

// Hàm khởi tạo bảng băm
void initHashTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashTable[i] = NULL;
    }
}
```

```c
// Hàm băm
int hashFunction(double key) {
    int intKey = (int)(key * SCALE_FACTOR); // Chuyển đổi khóa thực thành số nguyên
    return abs(intKey % TABLE_SIZE);        // Trả về chỉ số trong bảng băm
}
```



```c
// Hàm chèn khóa vào bảng băm
void insert(double key) {
    int index = hashFunction(key);
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->key = key;
    newNode->next = hashTable[index];
    hashTable[index] = newNode; // Thêm vào đầu danh sách liên kết
}
```

```c
// Hàm tìm kiếm khóa trong bảng băm
int search(double key) {
    int index = hashFunction(key);
    Node* current = hashTable[index];
    while (current != NULL) {
        if (fabs(current->key - key) < 1e-6) { // So sánh số thực với độ chính xác
            return 1; // Tìm thấy
        }
        current = current->next;
    }
    return 0; // Không tìm thấy
}
```

Open Addressing – đánh địa chỉ mở

- Đánh địa chỉ mở - Open Addressing
  - Các phần tử sẽ được lưu tại 1 ô nào đó trong bảng
  - Địa chỉ lưu trữ và địa chỉ tính được bởi hàm băm có thể khác nhau
  - Kích thước bảng băm luôn > số lượng khóa
  - Insert(K): tiếp tục dò cho đến khi tìm được ô trống để thêm khóa mới
  - Search(K): tiếp tục dò cho đến khi tìm thấy khóa hoặc tới khi tìm tới ô trống (khóa không tồn tại)
  - Delete(K): phần tử bị xóa sẽ là xóa trễ - lazy deletion: đánh dấu ô đó là xóa, khi thêm vào sẽ tương đương ô trống, nhưng khi tìm kiếm vẫn coi ô đó là có phần tử để dò tiếp.
    Một ô sẽ có 3 trạng thái: Ô trống/có phần tử/đánh dấu là xóa

Open Addressing – đánh địa chỉ mở

Phương pháp đánh địa chỉ mở - Open Addressing: **Có 3 phương pháp**

- **Dò tuyến tính**(Linear Probing): tại vị trí đụng độ, thực hiện tìm kiếm tuần tự để tìm ra khóa (khi tìm kiếm) hoặc vị trí trống (khi thêm mới)
- Hàm băm dạng h= k%M + i (%M) với i=0,1,2,3,...

  VD: hàm băm

  `h=k%10`

  Các khóa {89, 18, 49, 58, 69} được thêm lần lượt vào bảng băm ban đầu rỗng

Dò tuyến tính - Linear Probing

- Ô bị đụng độ được đánh dấu *
- Khi đến cuối mảng sẽ quay trở về đầu để dò tiếp

stt	Ban đầu	Thêm 89	Thêm 18	Thêm 49	Thêm 58	Thêm 69
0				49	49(*)	49(*)
1					58	58(*)
2						69
3						
4						
5						
6						
7						
8			18	18	18(*)	18
9		89	89	89(*)	89(*)	89(*)

Dò tuyến tính - Linear Probing

- Vấn đề của dò tuyến tính
  - Việc tìm kiếm vẫn là tìm kiếm tuần tự, vì vậy nếu đụng độ nhiều sẽ dẫn tới việc tìm kiếm suy biến về O(n)
  - Đụng độ sẽ dẫn tới hình thành cụm sơ cấp - **Primary Clustering** (các phần tử đụng độ liên tiếp nhau tạo thành 1 nhóm dẫn tới tìm kiếm tuần tự lâu hơn), và cụm thứ cấp - **Secondary Clustering** (ít xảy ra hơn, hai khóa khác nhau nhưng có cùng chuỗi dò khi vị trí khởi tạo của chúng trùng nhau).
  - Các phần tử khi xóa chỉ được đánh dấu chứ không thực sự bị xóa vì sẽ ảnh hưởng tới tìm kiếm các phần tử có cùng chuỗi dò

Open Addressing – đánh địa chỉ mở

- **Dò bậc hai -** Quadratic Probing: nếu xảy ra đụng độ tại địa chỉ h, thì ta sẽ tìm kiếm tiếp theo tại các địa chỉ h+1, h+4, h+9,.. ; là các địa chỉ dạng

$$h = k\%M + i^2 \ (\%M) \ \text{với } i=0,1,2,3,...$$

VD: hàm băm

$$\boldsymbol{h = k\%10 + i^2}$$

Các khóa {89, 18, 49, 58, 69} được thêm lần lượt vào bảng băm ban đầu rỗng


```c
// Struct để lưu một cặp key-value
typedef struct {
    char* key;
    char* value;
} HashEntry;

// Bảng băm
HashEntry* hashTable[TABLE_SIZE];

// Khởi tạo bảng băm
void initHashTable() {
    for (int i = 0; i < TABLE_SIZE; i++) {
        hashTable[i] = NULL;
    }
}
```

```c
// Hàm băm (cộng dồn ký tự)
unsigned int hashFunction(const char* key) {
    unsigned int hash = 0;
    for (int i = 0; key[i] != '\0'; i++) {
        hash += key[i];
    }
    return hash % TABLE_SIZE;
}
```

```c
// Thêm một cặp key-value vào bảng băm
void insert(const char* key, const char* value) {
    unsigned int index = hashFunction(key);
    // Tìm vị trí trống bằng dò tuyến tính
    for (int i = 0; i < TABLE_SIZE; i++) {
        unsigned int newIndex = (index + i) % TABLE_SIZE;
        // Nếu ô trống hoặc đã bị xóa
        if (hashTable[newIndex] == NULL || hashTable[newIndex]->key == NULL) {
            if (hashTable[newIndex] == NULL) {
                hashTable[newIndex] = malloc(sizeof(HashEntry));
            }
            hashTable[newIndex]->key = strdup(key);
            hashTable[newIndex]->value = strdup(value);
            return;
        }
        // Nếu khóa đã tồn tại, cập nhật giá trị
        else if (strcmp(hashTable[newIndex]->key, key) == 0) {
            free(hashTable[newIndex]->value);
            hashTable[newIndex]->value = strdup(value);
            return;
        }
    }
    printf("Bảng băm đầy, không thể chèn '%s'!\n", key);
}
```

```c
// Tìm giá trị theo khóa
char* search(const char* key) {
    unsigned int index = hashFunction(key);

    for (int i = 0; i < TABLE_SIZE; i++) {
        unsigned int newIndex = (index + i) % TABLE_SIZE;

        // Nếu tìm thấy khóa
        if (hashTable[newIndex] != NULL && hashTable[newIndex]->key != NULL &&
            strcmp(hashTable[newIndex]->key, key) == 0) {
            return hashTable[newIndex]->value;
        }

        // Nếu gặp ô trống
        if (hashTable[newIndex] == NULL) {
            break;
        }
    }

    return NULL; // Không tìm thấy
}
```

```c
// Xóa một khóa khỏi bảng băm
void deleteKey(const char* key) {
    unsigned int index = hashFunction(key);

    for (int i = 0; i < TABLE_SIZE; i++) {
        unsigned int newIndex = (index + i) % TABLE_SIZE;

        if (hashTable[newIndex] != NULL && hashTable[newIndex]->key != NULL &&
            strcmp(hashTable[newIndex]->key, key) == 0) {
            // Giải phóng bộ nhớ và đánh dấu là xóa
            free(hashTable[newIndex]->key);
            free(hashTable[newIndex]->value);
            hashTable[newIndex]->key = NULL;
            hashTable[newIndex]->value = NULL;
            return;
        }

        // Nếu gặp ô trống
        if (hashTable[newIndex] == NULL) {
            break;
        }
    }
    printf("Không tìm thấy khóa '%s' để xóa!\n", key);
}
```

Dò bậc hai - Quadratic Probing

- Ô bị đụng độ được đánh dấu *
- Khi đến cuối mảng sẽ quay trở về đầu để dò tiếp

stt	Ban đầu	Thêm 89	Thêm 18	Thêm 49	Thêm 58	Thêm 69
0				49	49	49(*)
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18(*)	18
9		89	89	89(*)	89(*)	89(*)

Dò bậc hai - Quadratic Probing

- **Dò bậc hai:**
  - Giảm được sự phân nhóm
  - Không phải tất cả các vị trí trong bảng đều được dò
    VD. Khi kích thước bảng là mũ của 2 thì 1/6 vị trí được dò, là số nguyên tố thì 1/2 được dò
  - Kích thước n, số phần tử hiện có trong bảng băm là m
  - Hệ số nạp – load factor $\alpha = m/n$
  - Thời gian tìm kiếm/thêm/xóa: $1/(1-\alpha)$

Open Addressing – đánh địa chỉ mở

- **Băm kép – double hashing:**
  - dùng hàm băm thứ 2 nếu hàm băm thứ nhất xảy ra đụng độ
  - Tìm theo vị trí dạng i*hash2(x) với i=0,1,2,3,…
  - Hàm băm tổng quát *: h = (hash1(key) + i * hash2(key)) % TABLE_SIZE*
  - Hàm băm thứ 2 tốt
    - Không tạo ra giá trị 0
    - Tất cả các ô trong bảng đều được dò
  - Băm kép tạo ra phân phối độc nhất cho mỗi khóa vì vậy nó sẽ không tạo ra các cụm
  - Mất thêm thời gian tính toán hàm băm

Băm kép – double hashing

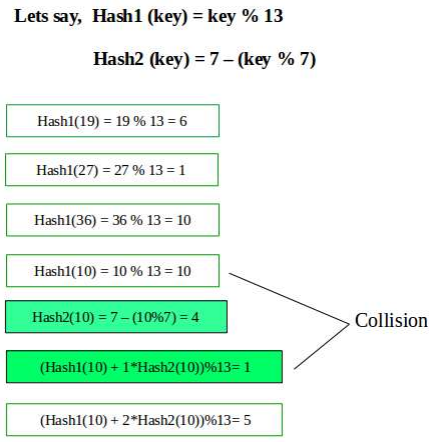
```
bool isFull()
{

    // if hash size reaches maximum size
    return (curr_size == TABLE_SIZE);
}


// function to calculate first hash
int hash1(int key)
{
    return (key % TABLE_SIZE);
}


// function to calculate second hash
int hash2(int key)
{
    return (PRIME - (key % PRIME));
}


DoubleHash()
{
    hashTable = new int[TABLE_SIZE];
    curr_size = 0;
    for (int i = 0; i < TABLE_SIZE; i++)
    hashTable[i] = -1;
}
```



ing

```
// function to insert key into hash table
void insertHash(int key)
{
    // if hash table is full
    if (isFull())
        return;


    // get index from first hash
    int index = hash1(key);


    // if collision occurs
    if (hashTable[index] != -1) {
        // get index2 from second hash
        int index2 = hash2(key);
        int i = 1;
        while (1) {
            // get newIndex
            int newIndex = (index + i * index2) % TABLE_SIZE;


            // if no collision occurs, store
            // the key
            if (hashTable[newIndex] == -1) {
                hashTable[newIndex] = key;
                break;
            }
            i++;
        }
    }


    // if no collision occurs
    else
        hashTable[index] = key;
    curr_size++;
}
```

```
// function to search key in hash table
void search(int key)
{
    int index1 = hash1(key);
    int index2 = hash2(key);
    int i = 0;
    while (hashTable[(index1 + i * index2) % TABLE_SIZE] != key) {
        if (hashTable[(index1 + i * index2) % TABLE_SIZE] == -1) {
            cout << key << " does not exist" << endl;
            return;
        }
        i++;
    }
    cout << key << " found" << endl;
}
```

Open Addressing – đánh địa chỉ mở

- **Dò theo khóa**: trong trường hợp đụng độ tại h thì dò tiếp tại vị trí cách vị trí đó khoảng cách bằng giá trị ký tự đầu tiên trong khóa (là số hoặc là mã ASCII).

  Ví dụ: nếu khóa **2918160** xảy ra đụng độ thì dò tại ô tiếp theo cách ô đụng độ 2 vị trí

- **Dò ngẫu nhiên** (Random Probing): sử dụng cách sinh số giả "ngẫu nhiên" để tạo ra vị trí dò tiếp. Cách sinh này phải là duy nhất với 1 giá trị khóa.

  Ví dụ: khóa 123245 thì sinh ra số ngẫu nhiên duy nhất là 5

Bảng băm

Đánh địa chỉ đóng - Separate Chaining	Đánh địa chỉ mở - open addressing
Cài đặt đơn giản	Cài đặt phức tạp hơn
Bảng băm không bao giờ đầy (trừ khi hết RAM), luôn có thể thêm phần tử	Bảng băm có thể đầy khi số phần tử >= MAX_SIZE
Ít nhạy cảm với hàm băm và hệ số nạp	Cần thêm các kỹ thuật xử lý để tránh tạo thành cụm, nhạy cảm với hệ số nạp, khi bảng đầy thời gian xử lý tăng lên nhiều
Thường dùng trong trường hợp không biết số lượng và tần xuất của các khóa	Dùng trong trường hợp biết số lượng khoá và tần số các khóa
Khó cache (do lưu trong danh sách liên kết)	Cache bảng dễ dàng hơn
một phần bảng có thể không được dùng	Mọi ô trong bảng đều được dùng (có thể dùng khi dò)
Tốn bộ nhớ phụ cho cho con trỏ	Không tốn bộ nhớ phụ

Bảng băm

- **Băm lại – rehashing**
  - Bảng băm dùng đánh địa chỉ mở, khi số lượng phần tử tăng – hệ số nạp tăng thì thời gian thực hiện tồi dần. $T(n) = 1/(1-\alpha)$
  - Giải pháp: giữ hệ số nạp đủ nhỏ, VD. dùng dò bậc 2 thì hệ số nạp nên $\alpha < 1/2$, còn tối đa có thể tới 3/4
  - Khi thêm phần tử mà hệ số nạp tới ngưỡng ➔ nhân đôi mảng cũ
  - Khi thay đổi mảng ➔ thay đổi hàm băm ➔ băm lại – rehashing
  - Băm lại: duyệt toàn bộ bảng băm cũ, thêm phần tử sang bảng mới dùng hàm băm mới
  - Thời gian $T(n) = O(n)$

Bảng băm

- Bảng băm VS Cây nhị phân tìm kiếm (AVL, RB tree)

Bảng băm	Cây nhị phân tìm kiếm cân bằng
Thời gian thêm/xóa/tìm kiếm cỡ O(1) – trong trường hợp tổng quát	Thời gian thêm/xóa/tìm kiếm luôn cỡ O(logn)
Thời gian tồi nhất có thể tới O(n) – khi đụng độ hoặc băm lại	Luôn là O(logn)
Không hỗ trợ duyệt danh sách các khóa theo thứ tự	Có thể duyệt danh sách các khóa theo thứ tự tăng dần
Không hỗ trợ	Có thể hỗ trợ tìm phần tử lớn nhất, nhỏ nhất,…
Thường dựa trên các cài đặt sẵn trong thư viện đi kèm ngôn ngữ lập trình	Dễ cài đặt và tùy biến
Không hỗ trợ tìm kiếm gần đúng	Hỗ trợ tìm kiếm gần đúng (ví dụ: phần tử lớn nhất nhỏ hơn x).
Cần thêm bộ nhớ cho mảng và xử lý xung đột.	Không cần bộ nhớ phụ (ngoài các nút cây).
Có thể xảy ra xung đột, cần giải quyết bằng chaining hoặc open addressing.	Không xảy ra xung đột

Bảng băm trong C++

- Trong STL hỗ trợ
  - Unordered_set : tìm xem 1 phần tử có trong tập hợp hay không
  - Unordered_map: tìm 1 cặp dạng <Key,Value>

Methods of unordered_set:

•insert()– Insert a new {element} in the unordered_set container.
•begin()– Return an iterator pointing to the first element in the unordered_set container.
•end()– Returns an iterator pointing to the past-the-end-element.
•count()– Count occurrences of a particular element in an unordered_set container.
•find()– Search for an element in the container.
•clear()– Removes all of the elements from an unordered_set and empties it.
•cbegin()– Return a const_iterator pointing to the first element in the unordered_set container.
•cend()– Return a const_iterator pointing to past-the-end element in the unordered_set container or in one of it's bucket.
•bucket_size()– Returns the total number of elements present in a specific bucket in an unordered_set container.
•erase()– Remove either a single element or a range of elements ranging from start(inclusive) to end(exclusive).
•size()– Return the number of elements in the unordered_set container.
•swap()– Exchange values of two unordered_set containers.
•emplace()– Insert an element in an unordered_set container.
•max_size()– Returns maximum number of elements that an unordered_set container can hold.
•empty()– Check if an unordered_set container is empty or not.
•hash_function() – This hash function is a unary function which takes asingle argument only and returns a unique value of type size_t based on it.
•reserve()– Used to request capacity change of unordered_set.
•bucket()– Returns the bucket number of a specific element.
•bucket_count() – Returns the total number of buckets present in an unordered_set container.
•load_factor()– Returns the current load factor in the unordered_set container.
•rehash()– Set the number of buckets in the container of unordered_set to given size or more.

Bảng băm trong C++

Methods of unordered_map :

•at(): This function in C++ unordered_map returns the reference to the value with the element as key k.
•begin(): Returns an iterator pointing to the first element in the container in the unordered_map container
•end(): Returns an iterator pointing to the position past the last element in the container in the unordered_map container
•bucket(): Returns the bucket number where the element with the key k is located in the map.
•bucket_count: bucket_count is used to count the total no. of buckets in the unordered_map. No parameter is required to pass into this function.
•bucket_size: Returns the number of elements in each bucket of the unordered_map.
•count(): Count the number of elements present in an unordered_map with a given key.
•equal_range: Return the bounds of a range that includes all the elements in the container with a key that compares equal to k.


```
#include <bits/stdc++.h>
using namespace std;

int main()
{
   unordered_set <int> mySet;

   printf("Nhap vao n phan tu:\n");
   int n,x;
   printf("n= ");
   scanf("%d",&n);
   for(int i=0; i<n;i++)
   {
      printf("Nhap vao so tiep theo: ");
      scanf("%d",&x);
      mySet.insert(x);
   }
   cout << "Cac phan tu KHONG trung trong day vua nhap : \n";
   unordered_set<int> :: iterator itr;
   for (itr = mySet.begin(); itr != mySet.end(); itr++)
      cout << (*itr) << endl;
   return 0;
}
```



```
#include <iostream>
#include <unordered_map>
using namespace std;

int main() {
   unordered_map<int, int> mylist;
   int n, x;
   cout<<"So luong so can nhap: ";
   cin>>n;
   printf("Nhap vao cac so: ");
   unordered_map<int, int>:: iterator p;
   for(int i=0; i<n; i++)
   {
      cin >>x;
      p = mylist.find(x);
      if(p!=mylist.end())  p->second++;
      else mylist[x]=1;
   }
   cout<<"danh sach cac gia tri va tan so tuong ung"<<endl;
   for (auto x : mylist)
     cout << x.first << " : " << x.second << endl;

   return 0;
}
```

Bảng băm

Bài 2. Cho bảng băm với kích thước 13, chỉ số các phần tử từ 0 đến 12, và dãy khóa 10, 100, 32, 45, 58, 126, 3, 29, 200, 400, 0

a) Sử dụng hàm băm i=k%13, vẽ các bước khi thêm các khóa vào bảng sử dụng phương pháp xử lý đụng độ là dò tuyến tính và dò bậc hai.
b) Sử dụng hàm băm là tổng của các chữ số trong khóa chia lấy dư cho 13, vẽ lại bảng băm với hai phương pháp xử lý đụng độ như phần a
c) Tìm một hàm băm mà không xảy ra đụng độ cho dãy khóa trên

Bài 3. Tìm phần giao của 2 danh sách liên kết đơn

Bài 4. Đếm tần số xuất hiện của các từ trong 1 đoạn văn với thời gian nhanh nhất

Bài 5. Tìm 2 phần tử a,b trong dãy n số nguyên phân biệt sao cho a+b = k

Bài 6. Tìm phần tử có tần số xuất hiện lớn nhất trong dãy, hoặc trong stream

Bài 7. Kiểm tra xem 2 tập có độc lập với nhau hay không

Bảng băm

- Bài 8. Cho dãy gồm n phần tử số nguyên, tìm cửa sổ kích thước k (chứa k phần tử) lớn nhất mà các phần tử không trùng lặp
- Bài 9. Cho danh sách người quản lý trực tiếp dạng (quản lý, nhân viên)

```
{ "A", "C" }, {"A", "B" }, { "C", "F" }, { "D", "E" }, { "E", "F" }, { "F", "G" }
```

Hãy tìm xem ai là người quản lý nhiều nhân viên nhất
VD. trong VD trên A sẽ là quản lý trực tiếp của 4 người

- Bài 10. Xây dựng chương trình kiểm tra chính tả cho đoạn văn dùng từ điển