

## BẢNG BẮM



# Các thuật ngữ thường dùng

- Hash table
- Hash map
- Hash function
- Collision resolution
- Chaining
- Open addressing



# Giới thiệu

- Phép Băm (Hashing): Là quá trình ánh xạ một giá trị khóa vào một vị trí trong bảng.
- Một hàm băm ánh xạ các giá trị khóa đến các vị trí ký hiệu:  $h$
- Bảng băm (Hash Table) là mảng lưu trữ các record, ký hiệu: HT
- HT có  $M$  vị trí được đánh chỉ mục từ 0 đến  $M-1$ ,  $M$  là kích thước của bảng băm.
- Bảng băm thích hợp cho các ứng dụng được cài đặt trên đĩa và bộ nhớ, là một cấu trúc dung hòa giữa thời gian truy xuất và không gian lưu trữ.

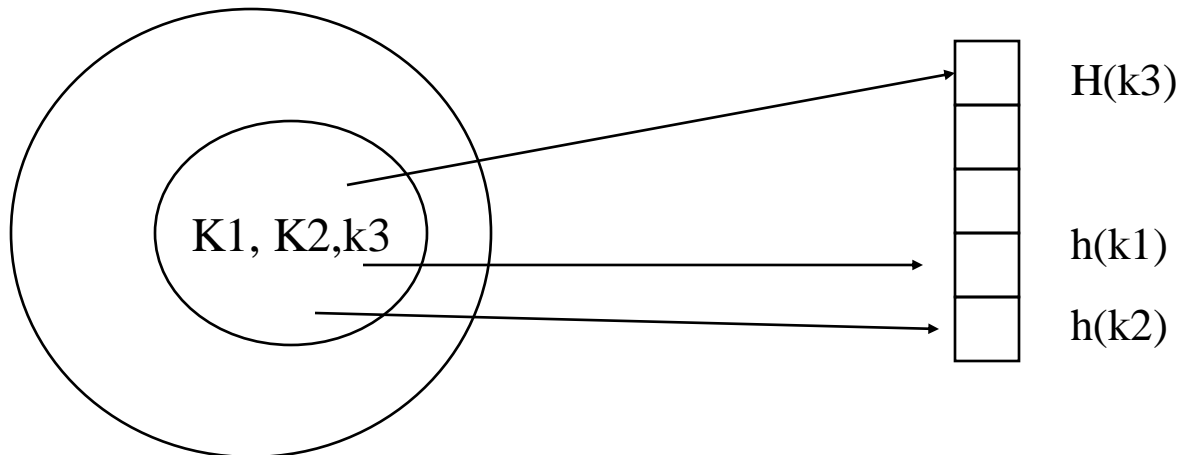


# Hàm băm (Hash Function)

- Hàm băm biến đổi một khóa vào một bảng các địa chỉ.



- Khóa có thể là dạng số hay số dạng chuỗi.
- Địa chỉ tính ra được là số nguyên trong khoảng 0 đến  $M-1$  với  $M$  là số địa chỉ có trên bảng băm



# Hàm băm (Hash Function)

- Hàm băm tốt thỏa mãn các điều kiện sau:
  - ↳ Tính toán nhanh.
  - ↳ Các khoá được phân bố đều trong bảng.
  - ↳ Ít xảy ra đụng độ.
- Giải quyết vấn đề băm với các khoá không phải là số nguyên:
  - ↳ Tìm cách biến đổi khoá thành số nguyên
    - Trong ví dụ trên, loại bỏ dấu '-' 9635-8904 đưa về số nguyên 96358904
    - Đối với chuỗi, sử dụng giá trị các ký tự trong bảng mã ASCII
  - ↳ Sau đó sử dụng các hàm băm chuẩn trên số nguyên.



# HF: Phương pháp chia

## ➤ Dùng số dư:

↪  $h(k) = k \bmod m$

↪  $k$  là khoá,  $m$  là kích thước của bảng.

0110010111000011010

$k \bmod 2^8$  chọn các bits

## ➤ vấn đề chọn giá trị $m$

### ➤ $m = 2^n$ (không tốt)

nếu chọn  $m = 2^n$  thông thường không tốt

$h(k) = k \bmod 2^n$  sẽ chọn cùng  $n$  bits cuối của  $k$

### ➤ $m$ là nguyên tố (tốt)

↪ Gia tăng sự phân bố đều

↪ Thông thường  $m$  được chọn là số nguyên tố gần với  $2^n$

↪ Chẳng hạn bảng ~4000 mục, chọn  $m = 4093$



# HF: Phương pháp nhân

## ➤ Sử dụng

↪  $h(k) = \lfloor m (k A \bmod 1) \rfloor$

↪  $k$  là khóa,  $m$  là kích thước bảng,  $A$  là hằng số:  $0 < A < 1$

## ➤ Chọn $m$ và $A$

↪ M thường chọn  $m = 2^p$

↪ Sự tối ưu trong việc chọn  $A$  phụ thuộc vào đặc trưng của dữ liệu.

↪ Theo Knuth chọn  $A = \frac{1}{2}(\sqrt{5} - 1) \approx 0.618033987$  được xem là tốt.



# Phép băm phổ quát

- Việc chọn hàm băm không tốt có thể dẫn đến xác suất đụng độ lớn.
- Giải pháp:
  - ↳ Lựa chọn hàm băm  $h$  ngẫu nhiên.
  - ↳ Chọn hàm băm độc lập với khóa.
  - ↳ Khởi tạo một tập các hàm băm  $H$  phổ quát và từ đó  $h$  được chọn ngẫu nhiên.
- Một tập các hàm băm  $H$  là phổ quát (*universal*) nếu với mọi  $\forall f, k \in H$  và 2 khóa  $k, l$  ta có xác suất:  $\Pr\{f(k) = f(l)\} \leq 1/m$





# Sự đụng độ (Collision)

- Sự đụng độ là hiện tượng các khóa khác nhau nhưng băm cùng địa chỉ như nhau.
- Khi  $key1 \neq key2$  mà  $f(key1) = f(key2)$  chúng ta nói nút có khóa  $key1$  đụng độ với nút có khóa  $key2$ .
- Thực tế người ta giải quyết sự đụng độ theo hai phương pháp: phương pháp nối kết và phương pháp băm lại.



# Giải quyết xung đột- Collision Resolution

Gồm các phương pháp:

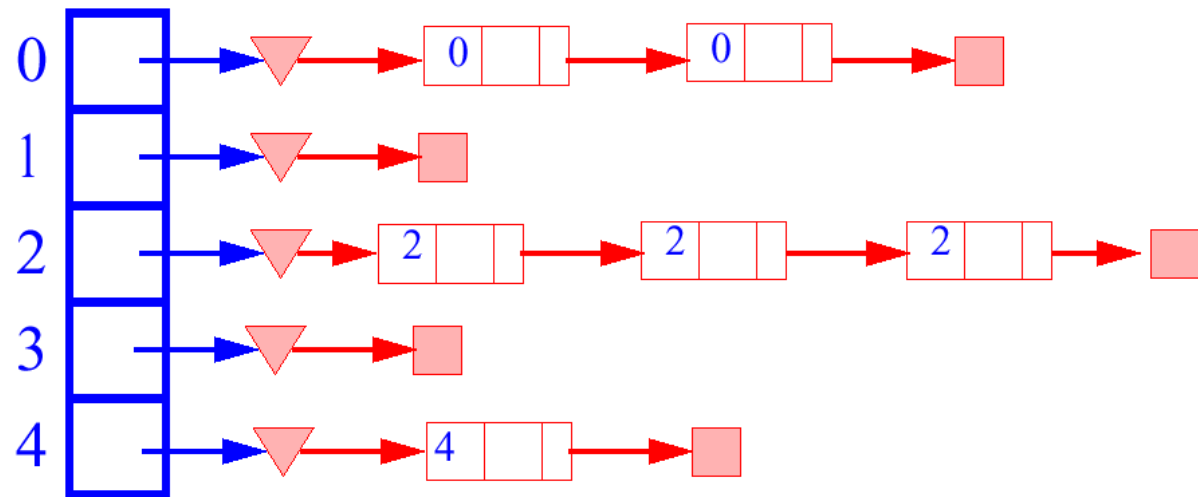
- Các loại bảng băm giải quyết sự xung đột bằng phương pháp nối kết như:
  - ↪ Nối kết trực tiếp - Direct Chaning
  - ↪ Nối kết hợp nhất – Coalesced Chaning
- Các loại bảng băm giải quyết sự xung đột bằng phương pháp băm lại như:
  - ↪ Dò tuyến tính - Linear probing
  - ↪ Dò bậc hai - Quadratic probing
  - ↪ Băm kép - Double hashing



# Direct Chaning

## I. Phương pháp nối kết trực tiếp:

- Các nút bị băm cùng địa chỉ (các nút bị xung đột) được gom thành một danh sách liên kết.
- các nút trên bảng băm được *băm* thành các danh sách liên kết. Các nút bị xung đột tại địa chỉ  $i$  được nối kết trực tiếp với nhau qua danh sách liên kết  $i$ .



# Direct Chaning

```
#define M 100
typedef struct nodes
{
    int key;
    struct nodes *next
}NODE;
```

```
//khai bao kieu con tro chi nut
typedef NODE *PHNODE;
```

```
/* khai bao mang HASHTABLE chua M con tro dau
cua MHASHTABLE */
typedef PHNODE HASHTABLE[M];
```



# Direct Chaning

## Hàm băm

Giả sử chúng ta chọn hàm băm dạng %:  $f(\text{key}) = \text{key} \% M$ .

```
int HF (int key)
{
    return (key % M);
}
```

## Phép toán khởi tạo bảng băm:

Khởi động các HASHTABLE.

```
void InitHASHTABLE( )
{
    for (int i=0;<M;i++)
        HASHTABLE[i]=NULL;
}
```



# Phương pháp nối kết trực tiếp(tt)

**Phép toán kiểm tra bảng băm rỗng HASHTABLE[i]:**

```
int emptyHASHTABLE (int i)
```

```
{
```

```
    return(HASHTABLE[i] ==NULL ?1 :0);
```

```
}
```

**Phép toán toàn bộ bảng băm rỗng:**

```
int empty( ){
```

```
    for (int i = 0;i<M;i++)
```

```
        if(HASHTABLE[i] !=NULL)
```

```
            return 0;
```

```
    return 1;
```

```
}
```



# Direct Chaning

## Phép toán thêm vào:

Thêm phần tử có khóa k vào bảng băm.

Giả sử các phần tử trên các HASHTABLE là có thứ tự để thêm một phần tử khóa k vào bảng băm trước tiên chúng ta xác định HASHTABLE phù hợp, sau đó dùng phép toán place của danh sách liên kết để đặt phần tử vào vị trí phù hợp trên HASHTABLE.

```
void insert(int k)
{
    int i= HF(k)
    InsertList(HASHTABLE[i],k); //phép toán thêm khoá k
                                //vào danh sách liên kết HASHTABLE[i]
}
```



# Direct Chaning

## Phép toán loại bỏ:

Xóa phần tử có khóa k trong bảng băm.

```
void remove ( int k){
    int b;
    PHNODE q,  p;
    b = HF(k);
    p = HASHTABLE(b);
    q=p;
    while(p !=NULL && p->key !=k)
    {
        q=p;
        p=p->next;
    }
    if (p == NULL) printf("\n khong co nut co khoa %d" ,k);
    else if (p == HASHTABLE [b])
        pop(b);
    else
        delafter(q);
}
```





# Direct Chaning

## Phép toán tìm kiếm:

PHNODE p;

int b;

b = HF (k);

p = HASHTABLE[b];

while(k != p->key && p !=NULL)

    p=p->next;

if (p == NULL || k !=p->key)           // khong tim thay

    return(NULL);

else   //tim thay

    return(p);

}



# Direct Chaning

## Phép toán duyệt HASHTABLE[i]:

Duyệt các phần tử trong HASHTABLE b.

```
void traverseHASHTABLE (int b)
```

```
{
```

```
    PHNODE p;
```

```
    p= HASHTABLE[b];
```

```
    while (p !=NULL)
```

```
    {
```

```
        printf("%3d", p->key);
```

```
        p= p->next;
```

```
    }
```

```
}
```



# Direct Chaning

## Phép toán duyệt toàn bộ bảng băm:

Duyệt toàn bộ bảng băm.

```
void traverse( )  
{  
    int b;  
    for (b = 0; n < M; b++)  
    {  
        printf("\nButket %d:", b);  
        traverseHASHTABLE(b);  
    }  
}
```



# Direct Chaning

- **Nhận xét:**
- Bảng băm dùng phương pháp nối kết trực tiếp sẽ "băm" n nút vào M danh sách liên kết (M bucket).
- Tốc độ Truy xuất phụ thuộc vào việc lựa chọn hàm băm sao cho băm đều n nút của bảng băm cho M bucket.
- Nếu chọn M càng lớn thì tốc độ thực hiện các phép toán trên bảng băm càng nhanh tuy nhiên không hiệu quả về bộ nhớ. Chúng ta có thể điều chỉnh M để dung hòa giữa tốc độ truy xuất và dung lượng bộ nhớ;
- Nếu chọn  $M=n$  thời gian truy xuất tương đương với truy xuất trên mảng(có bậc  $O(1)$ ), song tốn bộ nhớ.
- Nếu chọn  $M = n / k (k = 2, 3, 4, \dots)$  thì ít tốn bộ nhớ hơn k lần, nhưng tốc độ chậm đi k lần.



# Coalesced Chaining

## II. Bảng băm với phương pháp nối kết hợp nhất (*Coalesced Chaining Method*):

- Bảng băm trong trường hợp này được cài đặt bằng danh sách liên kết dùng mảng, có M nút. Các nút bị xung đột địa chỉ được nối kết nhau qua một danh sách liên kết.
- Mỗi nút của bảng băm là một mẫu tin có 2 trường:
  - ↪ Trường key: chứa các khóa node
  - ↪ Trường next: con trỏ chỉ node kế tiếp nếu có xung đột.
- Khi khởi động bảng băm thì tất cả trường key được gán NULLKEY, tất cả trường next được gán -1.

Key	next
NULL	-1
...	...
NULL	-1



# Coalesced Chaining

- Khi thêm một nút có khóa key vào bảng băm, hàm băm  $f(\text{key})$  sẽ xác định địa chỉ  $i$  trong khoảng từ 0 đến  $M-1$ .
- Nếu chưa bị xung đột thì thêm nút mới vào địa chỉ này.
- Nếu bị xung đột thì nút mới được cấp phát là nút trống phía cuối mảng. Cập nhật liên kết next sao cho các nút bị xung đột hình thành một danh sách liên kết.
- Khi tìm một nút có khóa key trong bảng băm, hàm băm  $f(\text{key})$  sẽ xác định địa chỉ  $i$  trong khoảng từ 0 đến  $M-1$ , tìm nút khóa key trong danh sách liên kết xuất phát từ địa chỉ  $i$ .



# Coalesced Chaning

Minh họa cho bảng băm có tập khóa là tập số tự nhiên, tập địa chỉ có 10 địa chỉ ( $M=10$ ) (từ địa chỉ 0 đến 9), chọn hàm băm  $f(\text{key}) = \text{key} \% 10$ .

Key=10, 42, 20, 109, 62

0	10	-1
1		-1
2		-1
...	null	-1
.		
8		-1
9		-1



# Coalesced Chaning

a. Khai báo cấu trúc bảng băm:

```
//Khai bao cau truc mot nut cua bang bam
typedef struct
{
    int key; //khoa cua nut tren bang bam
    int next;
    //con tro chi nut ke tiep khi co xung dot
}NODE;
//Khai bao bang bam
typedef NODE HASHTABLE[M];
int avail;
```





# Coalesced Chaning

**Hàm băm:**

```
int HF(int key){  
    return(key % 10);  
}
```

**Phép toán khởi tạo (Initialize):**

```
void initialize(){  
    int i;  
    for(i = 0; i < M; i++)  
    {  
        HASHTABLE[i].key = NULLKEY;  
        HASHTABLE[i].Next = -1;  
    }  
    avail = M-1; /* nút M-1 là nút o cuối bang chuan bi cap phat neu  
co xung dot */  
}
```



# Coalesced Chaning

## Phép toán tìm kiếm (search):

```
int search(int k)
{
    int i;
    i=HF(k);
    while(k !=HASHTABLE[i].key && i !=-1)
        i=HASHTABLE[i].next;
    if(k== HASHTABLE[i]key)
        return(i);        //tim thay
    return(M);             //khong tim thay
}
```



# Coalesced Chaning

## Phép toán lấy phần tử trống (Getempty):

Chọn phần tử còn trống phía cuối bản băm để cấp phát khi xảy ra xung đột.

```
int getempty()  
{  
    while(HASHTABLE[avail].key != NULLKEY)  
        avail - -;  
    return(avail);  
}
```



# Coalesced Chaning

```
if(HASHTABLE[i].key == NULLKEY)
    //Nut i con trong thi cap nhat
j = i;
else
    //Neu nut i la nut cuoi cua DSLK
{
    j = getempty();
    if(j < 0)
    {
        printf("\n Bang bam bi ");    return(j);
    }
    else
        HASHTABLE[i].next = j;
    }
    HASHTABLE[j].key = k;
    return(j);
}
```



# Coalesced Chaning

```
int insert(int k)
{
    int l;
    //con tro lan theo danh sach lien ket chua cac nut //bi xung dot
    int j;
    //dia chi nut trong duoc cap phat
    i = search(k);
    if(i != M)
    {
        printf("\n khoa %d bi trung,khong them nut nay duoc",k);
        return(i);
    }
    i=HF(k);
    while(HASHTABLE[i].next >=0)
    i=HASHTABLE[i].next;
```



# Open addressing

$$H(\text{key}, i) = (H(\text{key}) + f(i)) \% \text{TableSize}$$

Trong đó:

- *key*: từ khóa cần tìm
- *H(key, f(i))*: hàm băm lại lần thứ *i*.
- *H(key)* : hàm băm chính.
- *TableSize*: kích thước bảng băm.
- *%*: phép lấy dư (mod)
- *f(i)*: hàm thể hiện của các phương pháp xử lý đụng độ: dò tuyến tính, dò bậc hai, băm kép,  $f(0)=0$



# Linear Probing Method

$$H(\text{key}, i) = (H(\text{key}) + f(i)) \% \text{TableSize}$$

Với phương pháp dò tuyến tính thì  $f(i) = i$   
 $\Rightarrow H(\text{key}, i) = (H(\text{key}) + i) \% \text{TableSize}$



# Linear Probing Method

## III. Bảng băm với phương pháp dò tuyến tính (Linear Probing Method):

- Bảng băm trong trường hợp này được cài đặt bằng danh sách kê có M nút, mỗi nút của bảng băm là một mẫu tin có một trường key để chứa khoá của nút.
- Khi khởi động bảng băm thì tất cả trường key được gán NULLKEY.
- Khi thêm nút có khoá key vào bảng băm, hàm băm  $f(\text{key})$  sẽ xác định địa chỉ  $i$  trong khoảng từ 0 đến  $M-1$ :
- **Thêm các nút 32, 53, 22, 92, 17, 34, 24, 37, 56 vào bảng băm.**

0

nullkey

1

nullkey

2

nullkey

3

nullkey

...

nullkey

M-1

nullkey





Thêm các nút 32, 53, 22, 92, 17, 34, 24, 37, 56 vào bảng băm

0	NULL	0	NULL	0	NULL	0	NULL	0	56
1	NULL	1	NULL	1	NULL	1	NULL	1	NULL
2	32	2	32	2	32	2	32	2	32
3	53	3	53	3	53	3	53	3	53
4	NULL	4	22	4	22	4	22	4	22
5	NULL	5	92	5	92	5	92	5	92
6	NULL	6	NULL	6	34	6	34	6	34
7	NULL	7	NULL	7	17	7	17	7	17
8	NULL	8	NULL	8	NULL	8	24	8	24
9	NULL	9	NULL	9	NULL	9	37	9	37



# Linear Probing Method

- Nếu chưa bị xung đột thì thêm nút mới vào địa chỉ này.
- Nếu bị xung đột thì hàm băm lại lần 1-  $f_1$  sẽ xét địa chỉ kế tiếp, nếu lại bị xung đột thì hàm băm thì hàm băm lại lần 2 -  $f_2$  sẽ xét địa chỉ kế tiếp.
- Quá trình cứ thế cho đến khi nào tìm được địa chỉ trống và thêm nút vào địa chỉ này.
- Khi tìm một nút có khoá key vào bảng băm, hàm băm  $f(\text{key})$  sẽ xác định địa chỉ  $i$  trong khoảng từ 0 đến  $M-1$ , tìm nút khoá key trong khối đặt chứa các nút xuất phát từ địa chỉ  $i$ .



# Linear Probing Method

- Hàm băm lại của phương pháp dò tuyến tính là truy xuất địa chỉ kế tiếp. Hàm băm lại lần  $i$  được biểu diễn bằng công thức sau:
- $f(\text{key}) = (f(\text{key}) + i) \% M$  với  $f(\text{key})$  là hàm băm chính của bảng băm.
- Lưu ý địa chỉ dò tìm kế tiếp là địa chỉ 0 nếu đã dò đến cuối bảng

## **Nhận xét:**

- Chúng ta thấy bảng băm này chỉ tối ưu khi băm đều, tốc độ truy xuất lúc này có bậc  $O(1)$ .
- Trường hợp xấu nhất là băm không đều hoặc bảng băm đầy, lúc này hình thành một khối đặc có  $n$  nút trên tốc độ truy xuất lúc này có bậc  $O(n)$ .



# Linear Probing Method

```
//khai bao cau truc mot node cua bang bam
typedef struct
{
    int key; //khoa cua nut tren bang bam
}NODE;
//Khai bao bang bam co M nut
NODE HASHTABLE[M];
int N; /*bien toan cuc chi so nut hien co tren
bang bam*/
```



# Linear Probing Method

## Hàm băm:

Giả sử chúng ta chọn hàm băm dạng  $f(\text{key}_0) = \text{key} \% 10$ .

```
int HF(int key){  
    return(key% 10);  
}
```

Chúng ta có thể dùng một hàm băm bất kì thay cho hàm băm dạng % trên.

## Phép toán khởi tạo (initialize):

Gán tất cả các phần tử trên bảng có trường key là NULL.

Gán biến toàn cục N=0.

```
void initialize(){  
    int i;  
    for(i=0;i<M;i++)  
        HASHTABLE[i].key=NULLKEY;  
    N=0;  
}
```



# Linear Probing Method

## **Phép toán kiểm tra trống (empty):**

Kiểm tra bảng băm có trống hay không.

```
int empty( );  
{  
    return(N==0 ? TRUE;FALSE);  
}
```

## **Phép toán kiểm tra đầy (full):**

Kiểm tra bảng băm đã đầy chưa.

```
int full( )  
{  
    return (N==M-1 ? TRUE; FALSE);  
}
```

*Lưu ý* bảng băm đầy khi  $N=M-1$ , chúng ta nên dành ít nhất một phần tử trống trên bảng băm.



# Linear Probing Method

## Phép toán search:

```
int search(int k){
    int i;
    i=HF(k);
    while(HASHTABLE[i].key!=k &&
          HASHTABLE[i].key!=NULKEY)
    {
        i=i+1;
        if(i>=M) i=i-M;
    }
    if(HASHTABLE[i].key==k) //tim thay
        return(i);
    else //khong tim thay
        return(M);
}
```



# Linear Probing Method

## Phép toán insert:

Thêm phần tử có khoá k vào bảng băm.

```
int insert(int k)
{
    int i, j;
    if(search(K)<M) return M;//Trùng khoá
    if(full( ))
    {
        printf("\n Bang bam day khong them nut co
                khoa %d duoc",k);
        return;
    }
}
```





# Linear Probing Method

```
i=HF(k);  
while(HASHTABLE[i].key !=NULLKEY)  
{  
    //Bam lai (theo phuong phap do tuyen tinh)  
    i ++;  
    if(i >=M)  
        i= i-M;  
}  
HASHTABLE[i].key=k;  
N=N+1;  
return(i);  
}
```



# Linear Probing Method

## ➤ Nhận xét bảng băm dùng phương pháp dò tuyến tính:

Bảng băm này chỉ tối ưu khi băm đều, nghĩa là trên bảng băm các khối đặc chứa vài phần tử và các khối phần tử chứa sử dụng xen kẽ nhau, tốc độ truy xuất lúc này có bậc  $O(1)$ .

Trường hợp xấu nhất là băm không đều hoặc bảng băm đầy, lúc này hình thành một khối đặc có  $n$  phần tử, nên tốc độ truy xuất lúc này có bậc  $O(n)$ .



# Quadratic Probing Method

$$H(\text{key}, i) = (H(\text{key}) + f(i)) \% \text{TableSize}$$

Với phương pháp dò bậc 2 thì  $f(i) = i^2$

$$\Rightarrow H(\text{key}, i) = (H(\text{key}) + i^2) \% \text{TableSize}$$



## IV. Bảng băm với phương pháp dò bậc hai (Quadratic Probing Method)

➤ Bảng băm dùng phương pháp dò tuyến tính bị hạn chế do rải các nút không đều, bảng băm với phương pháp dò bậc hai rải các nút đều hơn.

➤ Bảng băm trong trường hợp này được cài đặt bằng danh sách kê có  $M$  nút, mỗi nút của bảng băm là một mẫu tin có một trường key để chứa khóa các nút

Khi khởi động bảng băm thì tất cả trường key bị gán NULLKEY.

Khi thêm nút có khóa key vào bảng băm, hàm băm  $f(\text{key})$  sẽ xác định địa chỉ  $i$  trong khoảng từ 0 đến  $M-1$ .



# Quadratic Probing Method

- Nếu chưa bị xung đột thì thêm nút mới vào địa chỉ  $i$ .
- Nếu bị xung đột thì hàm băm lại lần 1  $f_1$  sẽ xét địa chỉ cách  $i^2$ , nếu lại bị xung đột thì hàm băm lại lần 2  $f_2$  sẽ xét địa chỉ cách  $i \cdot 2^2, \dots$ , quá trình cứ thế cho đến khi nào tìm được trống và thêm nút vào địa chỉ này.
- Khi tìm một nút có khóa key trong bảng băm thì xét nút tại địa chỉ  $i=f(\text{key})$ , nếu chưa tìm thấy thì xét nút cách  $i \cdot 1^2, 2^2, \dots$ , quá trình cứ thế cho đến khi tìm được khóa (trường hợp tìm thấy) hoặc rơi vào địa chỉ trống (trường hợp không tìm thấy).



# Quadratic Probing Method

Hàm băm lại của phương pháp dò bậc hai là truy xuất các địa chỉ cách bậc 2.

Hàm băm lại hàm  $f_i$  được biểu diễn bằng công thức sau:

$f_i(\text{key}) = (f(\text{key}) + i^2) \% M$  với  $f(\text{key})$  là hàm băm chính của bảng băm.

Nếu đã dò đến cuối bảng thì trở về dò lại từ đầu bảng.

Bảng băm với phương pháp dò bậc hai nên chọn số địa chỉ  $M$  là số nguyên tố.



# Ví dụ

Thêm vào các khóa 10, 15, 16, 20, 30, 25, 26, 36

$$f_i(\text{key}) = (f(\text{key}) + i2) \% M$$

0	10	0	10	0	10	0	10	0	10
1	NULL	1	20	1	20	1	20	1	20
2	NULL	2	NULL	2	NULL	2	NULL	2	36
3	NULL	3	NULL	3	NULL	3	NULL	3	NULL
4	NULL	4	NULL	4	30	4	30	4	30
5	15	5	15	5	15	5	15	5	15
6	16	6	16	6	16	6	16	6	16
7	NULL	7	NULL	7	NULL	7	26	7	26
8	NULL	8	NULL	8	NULL	8	NULL	8	NULL
9	NULL	9	NULL	9	25	9	25	9	25



# Quadratic Probing Method

```
//Khai bao nut cua bang bam
typedef struct
{
    int key;           //Khoa cua nut tren bang
    bam
}NODE;
//Khai bao bang bam co M nut
NODE HASHTABLE[M];
int N;
//Bien toan cuc chi so nut hien co tren bang bam
```





# Quadratic Probing Method

## Hàm băm:

Giả sử chúng ta chọn hàm băm dạng%:  $f(\text{key}) = \text{key} \% 10$ .

Tương tự các hàm băm nói trên, chúng ta có thể dùng một hàm băm bất kì cho hàm băm dạng % trên.

## Phép toán initialize

Gán tất cả các phần tử trên bảng có trường key là NULLKEY.

Gán biến toàn cục  $N=0$ .

```
void initialize(){
    int i;
    for(i=0; i<M;i++)
        HASHTABLE[i].key = NULLKEY;
    N=0; //so nut hien co khoi dong bang 0
}
```



# Quadratic Probing Method

## Phép toán search:

Tìm phần tử có khóa k trên bảng băm, nếu không tìm thấy hàm này trả về trị M, nếu tìm thấy hàm này trả về địa chỉ tìm thấy.

```
int search(int k)
{
    int i, d;
    i = hashfuns(k);
    d = 1;
    while(HASHTABLE[i].key!=k&&HASHTABLE[i].key!=NULLKEY)
    {
        //Bam lai (theo phuong phap bac hai)
        i = (i+d) % M;
        d = d+2;
    }
    if(HASHTABLE[i].key ==k)
        return i;
    return M;
}
```



# Quadratic Probing Method

## Phép toán insert:

Thêm phần tử có khoá k vào bảng băm.

```
int insert(int k)
{
    int i, d;
    i = hashfuns(k);
    d = 1;
    if(search(K)<M) return M;//Trùng khoá
    if(full( ))
    {
        printf("\n Bang bam bi day khong them nut co
                khoa %d duoc",k);
        return;
    }
}
```



# Quadratic Probing Method

```
i=HF(k);  
  
while(HASHTABLE[i].key!=NULLKEY)  
{  
    i = (i+d) % M;  
    d = d+2;  
}  
HASHTABLE[i].key=k;  
N=N+1;  
return(i);  
}
```



# Quadratic Probing Method

## ➤ Nhận xét bảng băm dùng phương pháp dò bậc hai:

⇒ Nên chọn số địa chỉ  $M$  là số nguyên tố. Khi khởi động bảng băm thì tất cả  $M$  trường key được gán NULL, biến toàn cục  $N$  được gán 0.

⇒ Bảng băm đầy khi  $N = M-1$ , và nên dành ít nhất một phần tử trống trên bảng băm.

⇒ Bảng băm này tối ưu hơn bảng băm dùng phương pháp dò tuyến tính do rải rác phần tử đều hơn, nếu bảng băm chưa đầy thì tốc độ truy xuất có bậc  $O(1)$ . Trường hợp xấu nhất là bảng băm đầy vì lúc đó tốc độ truy xuất chậm do phải thực hiện nhiều lần so sánh.



# Double Hash Method

$$H(\text{key}, i) = (H(\text{key}) + f(i)) \% \text{TableSize}$$

Với phương pháp băm kép thì  $f(i) = i * H2(\text{key})$

$$\Rightarrow H(\text{key}, i) = (H1(\text{key}) + i * H2(\text{key})) \% \text{TableSize}$$



# Double Probing Method

## Mô tả:

- Cấu trúc dữ liệu: Bảng băm này dùng hai hàm băm khác nhau với mục đích để rải rác đều các phần tử trên bảng băm.
- Chúng ta có thể dùng hai hàm băm bất kì, ví dụ chọn hai hàm băm như sau:  
$$f1(key) = key \% M.$$
$$f2(key) = (M-2) - key \% (M-2).$$
- Bảng băm trong trường hợp này được cài đặt bằng danh sách kề có M phần tử, mỗi phần tử của bảng băm là một mẫu tin có một trường key để lưu khoá các phần tử.



# Double Probing Method

- - Khi khởi động bảng băm, tất cả trường key được gán NULL.
- - Khi thêm phần tử có khoá key vào bảng băm, thì  $i=f1(key)$  và  $j=f2(key)$  sẽ xác định địa chỉ  $i$  và  $j$  trong khoảng từ 0 đến  $M-1$ :
- Nếu chưa bị xung đột thì thêm phần tử mới tại địa chỉ  $i$  này.
- Nếu bị xung đột thì hàm băm lại lần 1  $f1$  sẽ xét địa chỉ mới  $i+j$ , nếu lại bị xung đột thì hàm băm lại lần 2 là  $f2$  sẽ xét địa chỉ  $i+2j$ , ..., quá trình cứ thế cho đến khi nào tìm được địa chỉ trống và thêm phần tử vào địa chỉ này.





Thêm vào các khóa 10, 15, 16, 20, 30, 25, 26, 36 :

$$f1(key) = key \% M.$$
$$f2(key) = (M-2) - key \% (M-2).$$

0	10	0	10	0	10	0	10	0	10
1	NULL	1	NULL	1	NULL	1	20	1	20
2	NULL	2	NULL	2	30	2	30	2	26
3	NULL	3	NULL	3	NULL	3	NULL	3	36
4	NULL	4	20	4	NULL	4	NULL	4	20
5	15	5	15	5	15	5	15	5	15
6	16	6	16	6	16	6	16	6	16
7	NULL	7	NULL	7	NULL	7	NULL	7	NULL
8	NULL	8	NULL	8		8	26	8	NULL
9	NULL	9	NULL	9	25	9	25	9	25



# Double Probing Method

- Khi tìm kiếm một phần tử có khoá key trong bảng băm, hàm băm  $i=f1(key)$  và  $j=f2(key)$  sẽ xác định địa chỉ  $i$  và  $j$  trong khoảng từ 0 đến  $M-1$ . Xét phần tử tại địa chỉ  $i$ , nếu chưa tìm thấy thì xét tiếp phần tử  $i+j$ ,  $i+2j$ , ..., quá trình cứ thế cho đến khi nào tìm được khoá (trường hợp tìm thấy) hoặc bị rơi vào địa chỉ trống (trường hợp không tìm thấy).



# Double Probing Method

```
//Khai bao phan tu cua bang bam  
typedef struct  
{  
    int key;    //khoa cua nut tren bang bam  
}NODE;  
//khai bao bang bam co M nut  
struct node HASHTABLE[M];  
int N;  
//bien toan cuc chi so nut hien co tren bang bam
```



# Double Probing Method

## Hàm băm:

Giả sử chúng ta chọn hai hàm băm dạng %:

$f1(key) = key \% M$  và  $f2(key) = M - 2 - key \% (M - 2)$ .

//Hàm băm thu nhất

```
int HF(int key)
```

```
{
```

```
    return(key%M);
```

```
}
```

//Hàm băm thu hai

```
int HF2(int key)
```

```
{
```

```
    return(M-2 - key%(M-2));
```

```
}
```



# Double Probing Method

## Phép toán initialize :

Khởi động bảng băm.

Gán tất cả các phần tử trên bảng có trường key là NULL.

Gán biến toàn cục  $N = 0$ .

```
void initialize()
{
    int i;
    for (i = 0 ; i < M ; i++ )
        HASHTABLE [i].key = NULLKEY;
    N = 0; // so nut hien co khoi dong bang 0
}
```



# Double Probing Method

## Phép toán empty :

Kiểm tra bảng băm có rỗng không.

int empty() .

{

return (N == 0 ? TRUE : FALSE) ;

}

## Phép toán full :

Kiểm tra bảng băm đã đầy chưa.

int full() .

{

return (N == M-1 ? TRUE : FALSE) ;

}

Lưu ý bảng băm đầy khi  $N=M-1$ , chúng ta nên dành ít nhất một phần tử trống trên bảng băm.



# Double Probing Method

## Phép toán search :

```
int search(int k)
{
    int i, j ;
    i = HF (k);
    j = HF2 (k);
    While (HASHTABLE [i].key!=k &&HASHTABLE [i] .key !=
NULLKEY)
        i = (i+j) % M ; //băm lại (theo phương pháp băm kép)
    if (HASHTABLE [i].key == k) // tìm thấy
        return (i) ;
    else
        return (M) ; // không tìm thấy
}
```



# Double Probing Method

## Phép toán insert :

Thêm phần tử có khoá k vào bảng băm.

```
int insert(int k)
{
    int i, j;
    if(search(k)<M) return M;//trùng khóa
    if (full () )
    {
        printf ("Bang bam bi day") ;
        return (M) ;
    }
}
```





# Double Probing Method

```
if (search (k) < M)
{
    printf ("Da co khoa nay trong bang bam");
    return (M) ;
}
i = HF (k) ;
j = HF 2 (k) ;
while (HASHTABLE [i].key != NULLEY)
    // Bam lai (theo phuong phap bam kep)
    i = (i + j) % M;
    HASHTABLE [i].key = k ;
    N = N+1;
return (i) ;
}
```



# Câu hỏi và Bài tập

1. Hãy cài đặt hàm băm sử dụng phương pháp nối kết trực tiếp.
2. Hãy cài đặt hàm băm sử dụng phương pháp băm kép.
3. Giả sử kích thước của bảng băm là  $SIZE = s$  và  $d_1, d_2, \dots, d_{s-1}$  là hoán vị ngẫu nhiên của các số  $1, 2, \dots, s-1$ . Dãy thăm dò ứng với khoá  $k$  được xác định như sau:

$$i_0 = i = h(k)$$

$$i_m = (i + d_i) \% SIZE, \quad 1 \leq m \leq s - 1$$

Hãy cài đặt hàm thăm dò theo phương pháp trên.



# Câu hỏi và Bài tập

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:

b. Bảng băm được chỉ mở với thăm dò tuyến tính.

c. Bảng băm được chỉ mở với thăm dò bình phương.

d. 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.

