UIT
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

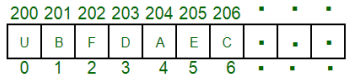
CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

Data Structures & Algorithms

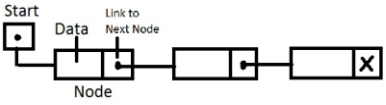
BẢNG BĂM– HASHTABLE

Các cấu trúc dữ liệu đã được học ?

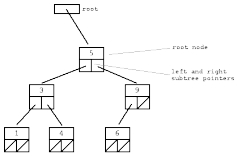
Array



List



Tree

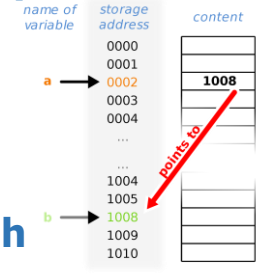


2

TÌM KIẾM ?

Tìm kiếm giá trị **key=x** ?? ?

Duyệt & **So sánh**



3

ĐỘ PHỨC TẠP TÌM KIẾM

- Sequential search: O(n)
- Binary search: $O(\log_2 n)$

4

ĐỘ PHỨC TẠP TÌM KIẾM

Size	Binary	Sequential (Average)	Sequential (Worst Case)
16	4	8	16
50	6	25	50
256	8	128	256
1,000	10	500	1,000
10,000	14	5,000	10,000
100,000	17	50,000	100,000
1,000,000	20	500,000	1,000,000

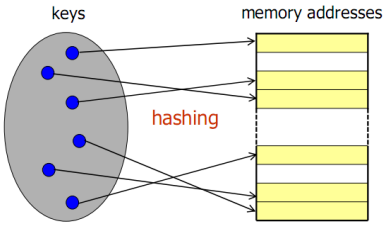
5

TÌM KIẾM

Phương pháp để tìm kiếm có độ phức tạp **O(1)** ?

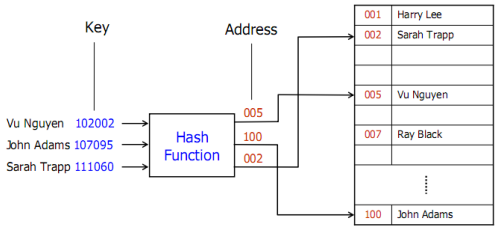
6

CÁI NHÌN ĐẦU TIÊN



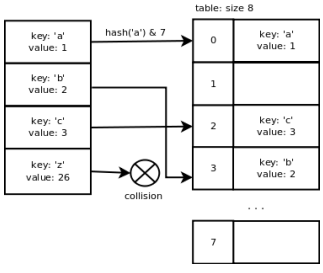
Mỗi key chỉ tới địa chỉ lưu thông tin

CHI TIẾT THÀNH PHẦN



CÁC KHÁI NIỆM LIÊN QUAN

- HashTable - Hash Map
- Collision
- Hash funtion

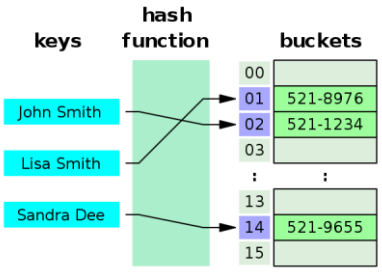


Hash Table - Hash Map

- Bảng băm là một CTDL trong đó **mỗi phần tử là một cặp (khóa, giá trị)** (key - value)

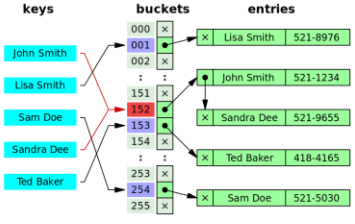
key	value
hello	hola
red	roja
blue	azul

Hash Table - Hash Map



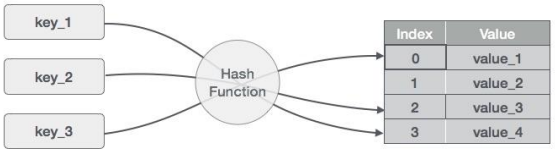
Đụng độ - Colission

Sự đụng độ là hiện tượng **các khóa khác nhau nhưng băm cùng địa chỉ** như nhau.

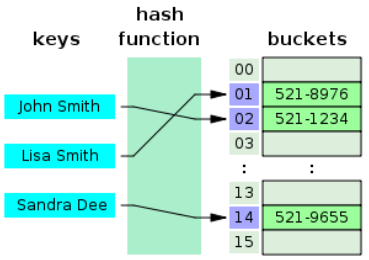


Hàm băm - HASH Funtion

Hàm được dùng để **ánh xạ một khoá – Key vào một dãy các số nguyên** và dùng các giá trị nguyên này để truy xuất dữ liệu



13



14

Yêu cầu của hash funtion

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 (Colission)**.
- Giải quyết vấn đề băm với các khoá không phải là số nguyên.

15

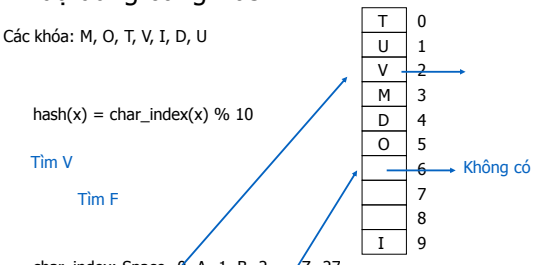
Ví dụ dùng bảng Hash

Các khóa: M, O, T, V, I, D, U

hash(x) = char_index(x) % 10

Tìm V

Tìm F



char_index: Space=0, A=1, B=2, ... Z=27

M	O	T	V	I	D	U	F
3	5	0	2	9	4	1	6

HF: Phương pháp chia

$k \bmod 2^8$ chọn các bits

01100101110**00011010**

- Dùng số dư:
  - $h(k) = k \bmod m$
  - k là khoá, m là kích thước của bảng.

Vấn đề chọn giá trị *m*

❖$m = 2^n$ **(không tốt)**

❖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 $m = 4093$

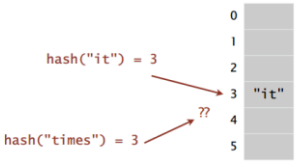
HF: Phương pháp chia

$h(k) = k \bmod m$

- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

Key	Hash	Chỉ mục mảng
1	1 % 20 = 1	1
2	2 % 20 = 2	2
42	42 % 20 = 2	2
4	4 % 20 = 4	4
12	12 % 20 = 12	12
14	14 % 20 = 14	14
17	17 % 20 = 17	17
13	13 % 20 = 13	13
37	37 % 20 = 17	17

Kỹ thuật giải quyết đụng độ (Colission)



Colission - ánh xạ của nhiều khoá khác nhau vào **cùng một địa chỉ**

19

Phương pháp giải quyết đụng độ (Colission)

(1) Phương pháp nối kết (**chaining method**): các phần tử bị **băm cùng địa chỉ** (các phần tử bị xung đột) được gom **thành một DSLK**.

(2) Phương pháp băm lại (rehash function): Nếu băm lần đầu bị xung đột thì băm lại lần 1, nếu bị xung đột nữa thì băm lai lần 2,... Quá trình **băm lại diễn ra cho đến khi không còn xung đột nữa**

20

(1) Phương pháp nối kết (chaining method)

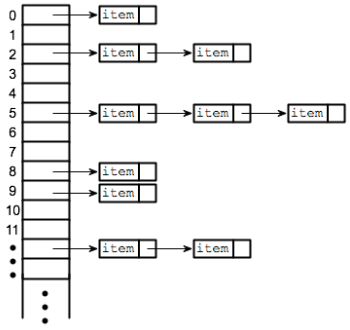
Mỗi phần tử của bảng băm **là một danh sách liên kết (bucket)**.

- **Chèn một phần tử** vào bảng băm ta **phải chèn nó vào trong một danh sách liên kết**.

-Trường hợp **hai phần tử có chung giá trị (hash code) thì ta sẽ chèn chúng vào chung một danh sách liên kết**

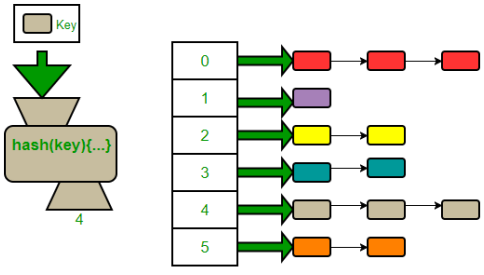
21

Kỹ thuật Separate Chaining



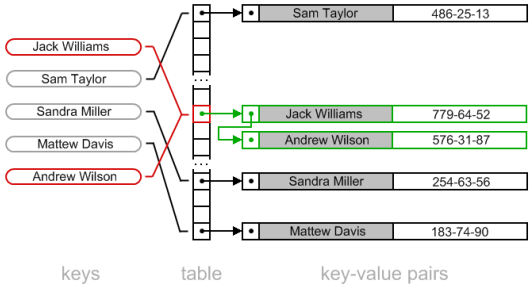
22

Kỹ thuật Separate Chaining



23

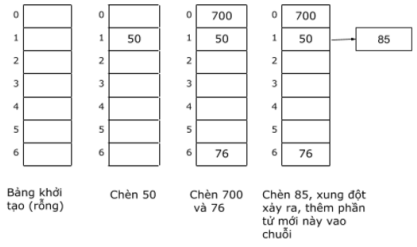
Kỹ thuật Separate Chaining



24

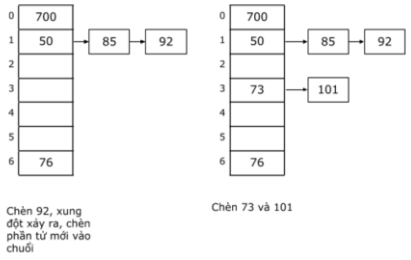
Kỹ thuật Separate Chaining

Giả sử ta có hàm băm chuyển đổi **các khóa 50, 700, 76, 85, 92, 73, 101** bằng cách **chia cho 7 rồi lấy số dư**.



Kỹ thuật Separate Chaining

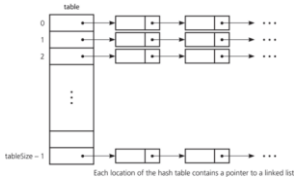
Giả sử ta có hàm băm chuyển đổi **các khóa 50, 700, 76, 85, 92, 73, 101** bằng cách **chia cho 7 rồi lấy số dư**.



Kỹ thuật Separate Chaining

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

//khai bao kieu con tro chi nut
typedef NODE *PHNODE;
/* Khai báo mảng HASHTABLE chứa M con trỏ đầu cua MHASHTABLE */
typedef PHNODE HASHTABLE[M];
```



Thao tác cơ bản trong Hash Table

Tìm kiếm: tìm kiếm một phần tử trong cấu trúc dữ liệu HashTable.

Chèn: chèn một phần tử vào trong cấu trúc dữ liệu HashTable.

Xóa: xóa một phần tử từ cấu trúc dữ liệu HashTable.

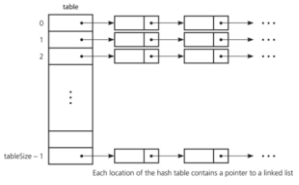
Kỹ thuật Separate Chaining

Hàm băm
Giả sử chúng ta chọn hàm băm dạng %: **f(key)=key % M**.

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

Khởi tạo bảng băm:

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



Kỹ thuật Separate Chaining

Kiểm tra bảng băm rỗng tại ví trí i HASHTABLE[i]

```
int emptyHASHTABLE (int i)
{
      return(HASHTABLE[i] ==NULL ?1 :0);
}
```

Kiểm tra bảng băm có rỗng không

```
int empty( ){
      for (int i = 0;i<M;i++)
            if(HASHTABLE[i] !=NULL)
            return 0;
      return 1;
}
```

Kỹ thuật Separate Chaining

Thêm phần tử có khóa k vào HASH:

- Giả sử các phần tử trên các HASHTABLE là có thứ tự.
- **Tìm vị trí cần thêm** trong HASH thông qua **hàm băm**.
- **Thêm vào bảng băm bằng cách thêm vào DSLK**

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

Kỹ thuật Separate Chaining

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)
        cout<< " khong co nut co khoa <<k;
        else if (p == HASHTABLE [b])
                pop(b);
        else
                delafter(q);
}
```

Kỹ thuật Separate Chaining

Phép toán tìm kiếm:

```
PHNODE search(int k)
{
    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);
}
```

Kỹ thuật Separate Chaining

Phép toán duyệt HASHTABLE[i]:

Duyệt các phần tử trong HASHTABLE b.

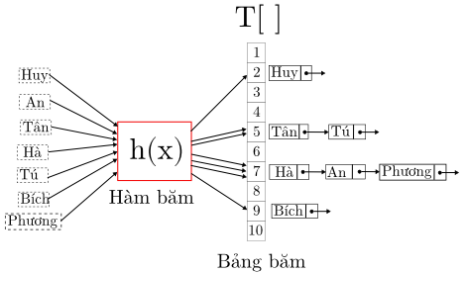
```
void traverseHASHTABLE (int b)
{
        PHNODE p;
        p= HASHTABLE[b];
        while (p !=NULL)
        {
                printf("%3d", p->key);
                p= p->next;
        }
}
```

Kỹ thuật Separate Chaining

Duyệt toàn bộ bảng băm:

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

Kỹ thuật Separate Chaining



Kỹ thuật Separate Chaining

Ưu điểm:

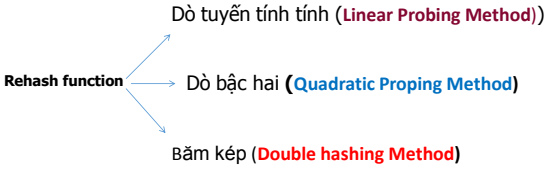
- **Cài đặt đơn giản**
- Không phải lo tới kích thước bảng băm, và ta luôn có thể **thêm dữ liệu vào bảng bằng cách thêm vào các danh sách liên kết**.

Nhược điểm:

- Phải sử **dụng vùng nhớ ngoài**.
- Khi mà **chuỗi (danh sách liên kết) trở nên quá dài, lúc đó thời gian cho các thao tác tìm kiếm, xóa phần tử có thể rất tốn thời gian**.

(2) Phương pháp (rehash function)

Nếu băm lần đầu bị xung đột thì băm lại lần 1, nếu bị xung đột nữa thì băm lai lần 2,... Quá trình **băm lại diễn ra cho đến khi không còn xung đột nữa**



(2.1) Dò tuyến tính (Linear Probing Method)

- Nếu **chưa bị xung đột** thì **thêm nút mới vào địa chỉ qua hàm băm**.
- **Nếu bị xung đột** thì hàm **băm lại lần 1- f1 sẽ xét địa chỉ kế tiếp**,**nếu lại** bị xung đột thì hàm băm thì **hàm băm lại lần 2 - f2 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(key) **sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1**, **tìm nút khoá key trong khối đặc chứa các nút xuất phát từ địa chỉ i.**

(2.1) Dò tuyến tính (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 diên bằng công thức sau:

$$f(key)=(f(key)+i)\ \%M$$

với **f(key) là hàm băm chính** của bảng băm.

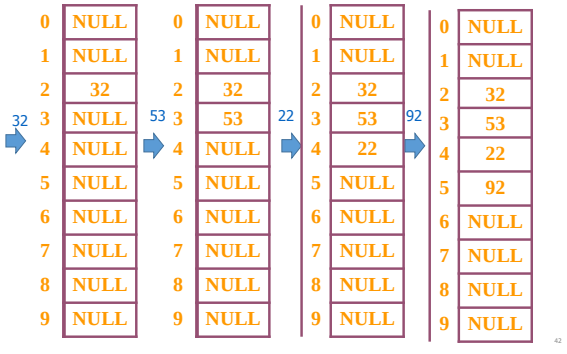
- Lưu ý địa chỉ dò tìm kế tiếp là địa chỉ 0 nếu đã dò đến cuối bảng

(2.1) Dò tuyến tính (Linear Probing Method)

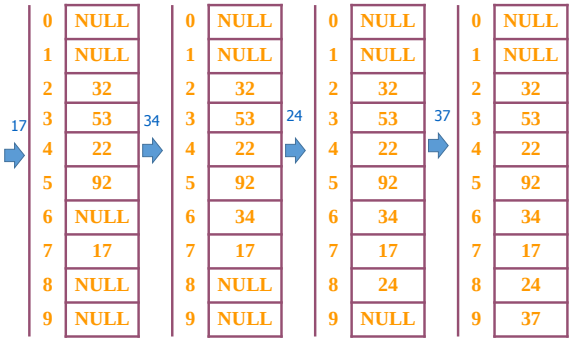
- Được cài đặt bằng **danh sách kề có M nút.**
- Mỗi **nút của bảng băm** là một mẫu tin có **một trường key** để chứa khoá của nút.
- **Khi khởi tạo** bảng băm thì **tất cả trường key được gán NULLKEY**.
- **Khi thêm nút có khoá key** vào bảng băm, **hàm băm f(key) sẽ xác định địa chỉ i** trong khoảng từ 0 đến M-1:

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 bằng phương pháp dò tuyến tính

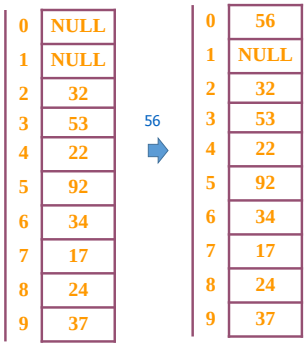


Thêm các nút 32, 53, 22, 92, 17, 34, 24, 37, 56 vào bảng băm bằng phương pháp dò tuyến tính



43

Thêm các nút 32, 53, 22, 92, 17, 34, 24, 37, 56 vào bảng băm bằng phương pháp dò tuyến tính



44

(2.1) Linear Probing Method – Cài đặt

```
//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*/
```

(2.1) Linear Probing Method – Cài đặt

Hàm băm:
Giả sử chúng ta chọn hàm băm dạng %:f(key_0) = 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;
}
```

(2.1) Linear Probing Method – Cài đặt

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.

(2.1) Linear Probing Method – Cài đặt

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);
}
```

(2.1) Linear Probing Method – Cài đặt

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;
    }
    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);
}
```

(2.1) Nhận xét - Linear Probing Method

➢ 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 chưa sử dụng xen kẽ nhau**, tốc độ truy xuất lúc này có bậc 0(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 0(n).**

(2.2) Dò bậc hai (Quadratic Proping 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 - 2^2** ,…,
- Quá trình **cứ thế cho đến khi nào tìm được trống và thêm nút vào địa chỉ này**.
- **Khi tìm môt nút có khóa key** trong bảng băm thì xét nút tại địa chỉ i=f(key), nếu chưa tìm thấy thì **xét nút cách i 1^2,2^2,…,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).

(2.2) Dò bậc hai (Quadratic Proping 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:

$$f_i(key)=(f(key) + i^2) \% M$$

với f(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 do bậc hai **nên chọn số địa chỉ M là số nguyên tố**.

(2.2) Dò bậc hai (Quadratic Proping Method)

➢Khắc phục phương pháp dò tuyến tính **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** kề có M nút, **mỗi nút của bảng băm là một mẫu tin có một trường key để chứa khóa các nút**

➢ Khi khởi tạo bảng băm thì **tất cả trường key bị gán NULLKEY**.

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

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

fi(key)=(f(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

(2.2) Quadratic Proping Method – cài đặt

```
//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
```

(2.2) Quadratic Proping Method – cài đặt

Hàm băm:
Giả sử chúng ta chọn hàm băm dạng%: f(key)=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
}
```

(2.2) Quadratic Proping Method – cài đặt

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;
    retiurn M;
}
```

(2.2) Quadratic Proping Method – cài đặt

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;
    }
```

(2.2) Quadratic Proping Method – cài đặt

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

(2.2) Quadratic Proping Method – Nhận xét

•**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 0(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.

(2.2) Băm kép -Double hashing Method

- 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).

(2.2) Băm kép -Double hashing Method

- 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**.
- **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**.

(2.2) Băm kép -Double hashing Method

- Bảng băm **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.**
- Khởi tạo bảng băm: **tất cả trường key được gán NULL**.
- **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., …, quá trình cứ thế cho đến khi nào tìm được khoá**Xét phần tử tại địa chỉ i, nếu chưa tìm thấy thì xét tiếp phần tử i+ji+2j** (trường hợp tìm thấy) hoặc bị rơi vào địa chỉ trống (trường hợp không tìm thấy).

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

(2.2) Double hashing Method – Cài đặt

```
//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
```

(2.2) Double hashing Method – Cài đặt

Hàm băm:
Giả sử chúng ta chọn hai hàm băm dạng %:
f1(key0=key %M va f2(key) =M-2-key%(M-2).

```
//Ham bam thu nhat
int HF(int key)
{
        return(key%M);
}
//Ham bam thu hai
int HF2(int key)
{
        return(M-2 – key%(M-2));
}
```

(2.2) Double hashing Method – Cài đặt

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
}
```

(2.2) Double hashing Method – Cài đặt

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.

(2.2) Double hashing Method – Cài đặt

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 ; //bam lai (theo phuong phap bam kep)
        if (HASHTABLE [i]).key == k) // tim thay
                return (i) ;
        else                    // khong tim thay
                return (M) ;
}
```

(2.2) Double hashing Method – Cài đặt

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) ;
        }
```

(2.2) Double hashing Method – Cài đặt

```
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 sửdụng phương pháp nối kết trực tiếp.
2. Hãy cài đặt hàm băm sửdụng phương pháp băm kép.
3. Giả sử kích thước của bảng băm là SIZE = s và d1, d2, …, ds-1 là hoán vị ngẫu nhiên của các số 1, 2, …, s-1. Dãy thăm dò ứng với khoá k được xác định nhưsau:

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

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

Hãy cài đặt hàm thăm dò theo phương pháp trên.

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 dữliệu với khoá là 65.

Slide được tham khảo từ

- **Slide được tham khảo từ:**
  - Slide CTDL GT, Khoa Khoa Học Máy Tính, ĐHCNTT
  - Congdongcviet.com
  - Cplusplus.com