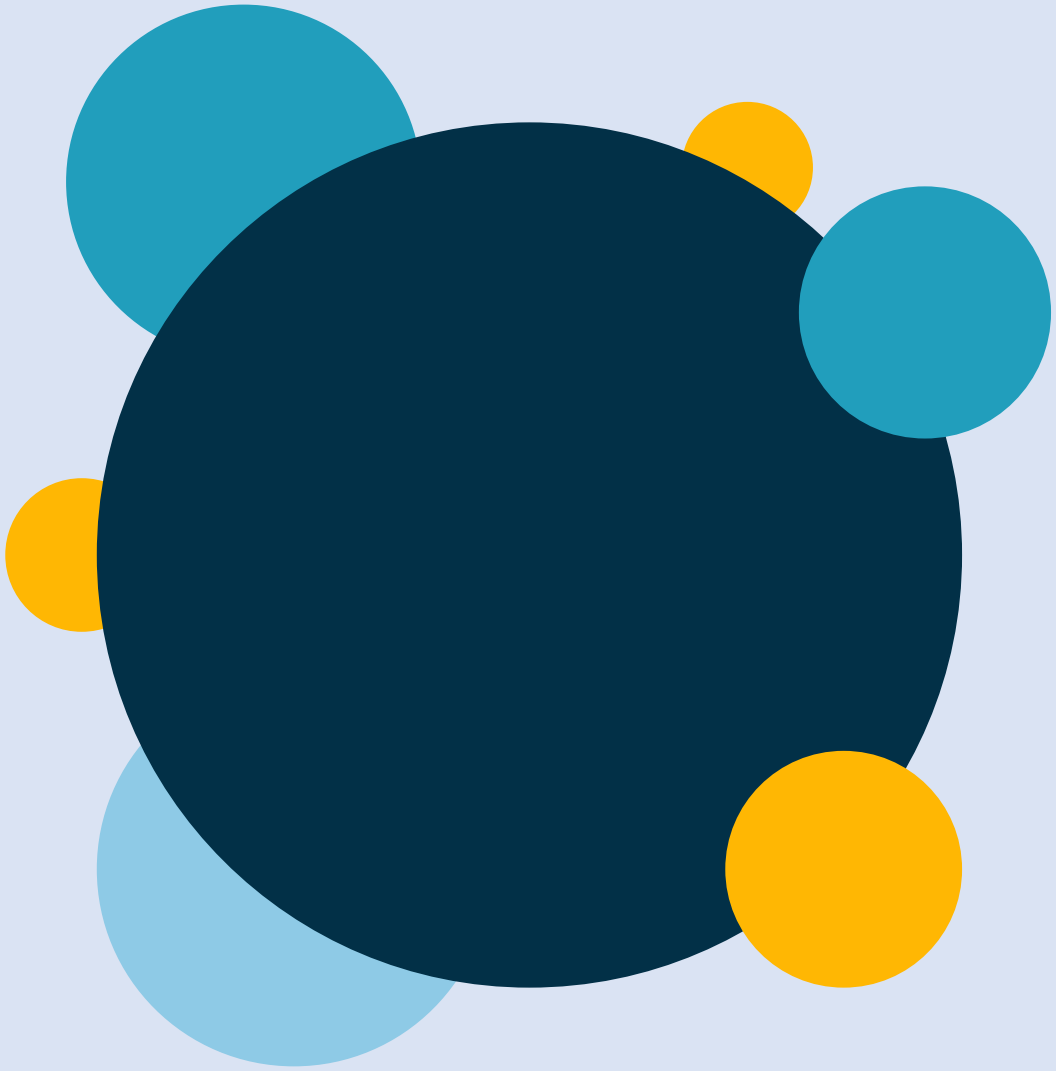
Data Structure and Algorithms



Nhóm 18

Hash Table

Thành viên trong nhóm

Nguyễn Thắng Lợi

Tôn Khánh An

Nguyễn Nghĩa Trung Kiên

NỘI DUNG THUYẾT TRÌNH

Giới thiệu về bảng băm (Hash Table)

Hàm băm (Hash Function)

Sự đụng độ (Collision)

Các phương pháp giải quyết đụng độ

Collision Resolutions

- Nối kết trực tiếp – Direct Chaining
- Nối kết hợp nhất – Coalesced Chaining
- Dò tuyến tính – Linear Probing
- Dò bậc hai – Quadratic Probing
- Băm kép – Double Hashing

Đặt vấn đề

- Giả sử chúng ta muốn tạo một danh sách lưu trữ thông tin của các bạn sinh viên trong lớp (các sinh viên phân biệt nhau theo số điện thoại) và thực hiện các thao tác cơ bản sau trên danh sách:
  - Thêm sinh viên
  - Tìm kiếm một sinh viên
  - Xóa một sinh viên khỏi danh sách

Đặt vấn đề

- Giả sử chúng ta muốn tạo một danh sách lưu trữ thông tin của các bạn sinh viên trong lớp (các sinh viên phân biệt nhau theo số điện thoại) và thực hiện các thao tác cơ bản sau trên danh sách:
  - Thêm sinh viên
  - Tìm kiếm một sinh viên
  - Xóa một sinh viên khỏi danh sách

MSSV	Name
23520031	An
23520789	Kiên
23520872	Lợi
…	…

- Câu hỏi đặt ra là tạo danh sách này như thế nào sao cho mỗi thao tác được thực hiện theo cách hiệu quả nhất?

Solutions

- Sử dụng mảng 1 chiều:
  - Thao tác tìm kiếm có độ phức tạp tuyến tính O(n).
  - Nếu mảng được xếp theo thứ tự, thì việc tìm kiếm có thể thực hiện trong O(logn) nhưng khi đó thao tác thêm và xóa lại tốn thời gian hơn.

Solutions

- Sử dụng mảng 1 chiều:
  - Thao tác tìm kiếm có độ phức tạp tuyến tính O(n).
  - Nếu mảng được xếp theo thứ tự, thì việc tìm kiếm có thể thực hiện trong O(logn) nhưng khi đó thao tác thêm và xóa lại tốn thời gian hơn.
- Sử dụng Linked List:
  - Thao tác tìm kiếm có độ phức tạp tuyến tính O(n).

Solutions

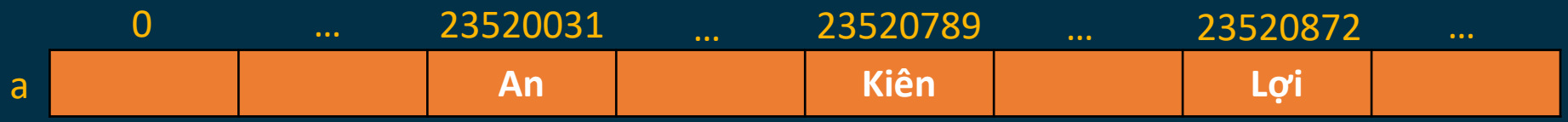
- Sử dụng mảng 1 chiều:
  - Thao tác tìm kiếm có độ phức tạp tuyến tính O(n).
  - Nếu mảng được xếp theo thứ tự, thì việc tìm kiếm có thể thực hiện trong O(logn) nhưng khi đó thao tác thêm và xóa lại tốn thời gian hơn.
- Sử dụng Balanced BST:
  - Tìm kiếm: O(logn).
  - Thêm: O(logn).
  - Xóa: O(logn).
- Sử dụng Linked List:
  - Thao tác tìm kiếm có độ phức tạp tuyến tính O(n).

Solutions

- Sử dụng mảng 1 chiều:
  - Thao tác tìm kiếm có độ phức tạp tuyến tính O(n).
  - Nếu mảng được xếp theo thứ tự, thì việc tìm kiếm có thể thực hiện trong O(logn) nhưng khi đó thao tác thêm và xóa lại tốn thời gian hơn.
- Sử dụng Balanced BST:
  - Tìm kiếm: O(logn).
  - Thêm: O(logn).
  - Xóa: O(logn).
- Sử dụng Linked List:
  - Thao tác tìm kiếm có độ phức tạp tuyến tính O(n).
- Sử dụng bảng truy cập trực tiếp (Direct Access Table):
  - Tìm kiếm: O(1).
  - Thêm: O(1).
  - Xóa: O(1).

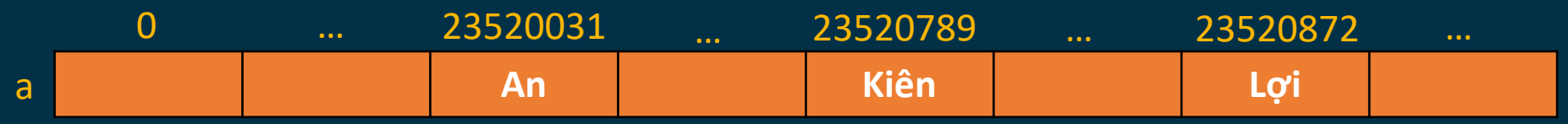
Solutions

- Sử dụng bảng truy cập trực tiếp (Direct Access Table):
  - Tìm kiếm: O(1).
  - Thêm: O(1).
  - Xóa: O(1).



Solutions

- Sử dụng bảng truy cập trực tiếp (Direct Access Table):
  - Tìm kiếm: O(1).
  - Thêm: O(1).
  - Xóa: O(1).



- Hạn chế:
  - Kích thước mảng lớn nhưng không sử dụng hết, gây lãng phí bộ nhớ. (cần mảng có n = m + 1 phần tử với m là giá trị MSSV lớn nhất).
  - Khi giá trị key quá lớn, tiêu tốn nhiều bộ nhớ.
  - Giá trị key phải là số tự nhiên (không dùng được trong trường hợp key là string, floating point…).

Solutions

→ Bảng băm (Hash table) - cải tiến dựa trên Bảng truy cập trực tiếp.

Solutions

→ Bảng băm (Hash table) - cải tiến dựa trên Bảng truy cập trực tiếp.

- Ví dụ: H(k) = k mod 4
  k = 23520031 => H(k) = 3
  k = 23520872 => H(k) = 0
  k = 23520789 => H(k) = 1

Index	MSSV	Name
0	23520872	Lợi
1	23520789	Kiên
3	23520031	An

Giới thiệu về bảng băm (Hash Table)

- Bảng băm (Hash table) là một cấu trúc dữ liệu lưu trữ các cặp dữ liệu theo định dạng (key, value). Bảng băm cho phép tham chiếu trực tiếp đến một phần tử trong bảng dữ liệu thông qua việc biến đổi số học trên khóa của phần tử để có được địa chỉ tương ứng của phần tử đang xét ở trong bảng.

Giới thiệu về bảng băm (Hash Table)

- Bảng băm có thể xem như sự mở rộng của mảng thông thường. Việc địa chỉ hóa trực tiếp trong mảng cho phép truy nhập đến phần tử bất kỳ trong thời gian O(1). Mặc dù, trong tình huống xấu nhất, thao tác tìm kiếm đòi hỏi thời gian O(N) giống như trên danh sách liên kết, nhưng trên thực tế bảng băm làm việc hiệu quả hơn nhiều.

Các thuật ngữ thường dùng

- Hash Table (bảng băm)
- Hash function (hàm băm)
- Collision resolution (giải quyết đụng độ)
- Open addressing (địa chỉ mở)

Các thuật ngữ thường dùng

- Một hàm băm (Hash function): ánh xạ các giá trị khóa đến các vị trí. Ký hiệu: h hay HF
- Bảng băm (Hash Table): Là mảng lưu trữ các record, ký hiệu: HT
- HT có M vị trí được đánh chỉ mục (index) từ 0 đến M-1, với 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.

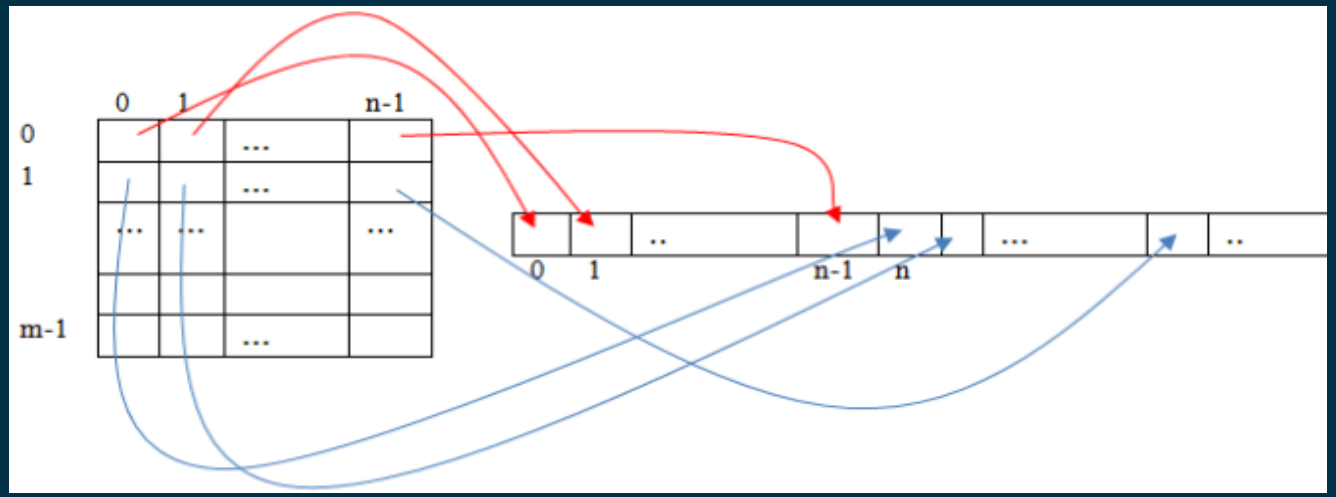
- Khóa không bị giới hạn về kiểu dữ liệu.
- Địa chỉ tính ra được là số nguyên trong khoảng 0 đến M-1 với M là số địa chỉ có thể có trên bảng băm.

Hàm băm (Hash Function)

- Hàm băm thường được tổ chức ở dạng công thức hoặc được tổ chức ở dạng bảng tra gọi là bảng truy xuất (access table).

Hàm băm (Hash Function)

- Hàm băm dạng công thức:



Hàm băm H(i,j) = ni+j

(Ở trường hợp này key là chỉ số hàng, cột của ma trận và value là giá trị của một ô trong ma trận)

Hàm băm (Hash Function)

- Hàm băm dạng bảng tra (bảng truy xuất - access table)

Khoá	Địa chỉ	Khóa	Địa chỉ	Khóa	Địa chỉ	Khóa	Địa chỉ
a	0	h	7	o	14	v	21
b	1	l	8	p	15	w	22
c	2	j	9	q	16	x	23
d	3	k	10	r	17	y	24
e	4	l	11	s	18	z	25
f	5	m	12	t	19	/	/
g	6	n	13	u	20	/	/

Khóa là bộ 26 kí tự, địa chỉ được đánh số từ 0 đến 25

Hàm băm (Hash Function)

- Hàm băm dạng công thức được sử dụng thông dụng và đa dạng nhất.
- Ví dụ:

MSSV	Name
23520031	An
23520043	Mai
23520789	Kiên
23520872	Lợi
23520856	Lộc
23520798	Dương

Hàm băm (Hash Function)

- Hàm băm dạng công thức được sử dụng thông dụng và đa dạng nhất.
- Ví dụ:

MSSV	Name	Index
23520031	An	3
23520043	Mai	3
23520789	Kiên	1
23520872	Lợi	0
23520856	Lộc	0
23520798	Dương	2

Hàm băm (Hash Function)

- Hàm băm dạng công thức được sử dụng thông dụng và đa dạng nhất.
- Ví dụ:

Index	Name	
0	Lợi	Lộc
1	Kiên	
2	Dương	
3	An	Mai

➔ Sự đụng độ (Collision) xảy ra.

Hàm băm (Hash Function)

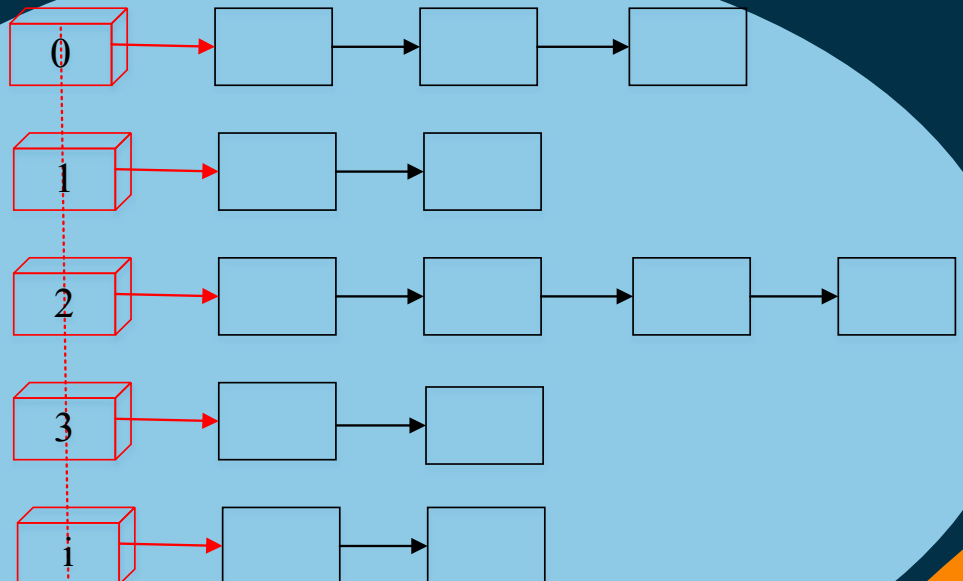
- Sự đụng độ là hiện tượng các khóa khác nhau nhưng băm cùng địa chỉ như nhau.
- Khi key1 != key2 mà f(key1)=f(key2), chúng ta nói nút có khóa key1 đụng độ với nút có khóa key2.

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
  - Xử lý được các loại khóa có kiểu dữ liệu khác nhau
- Giải quyết vấn đề băm với các khoá không phải là số nguyên:
  - Tìm cách biến đổi khóa thành số nguyên.
  - Ví dụ: loại bỏ dấu "-" trong "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.

Các phương pháp giải quyết đụng độ

- **Direct Chaining Method - PP 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.



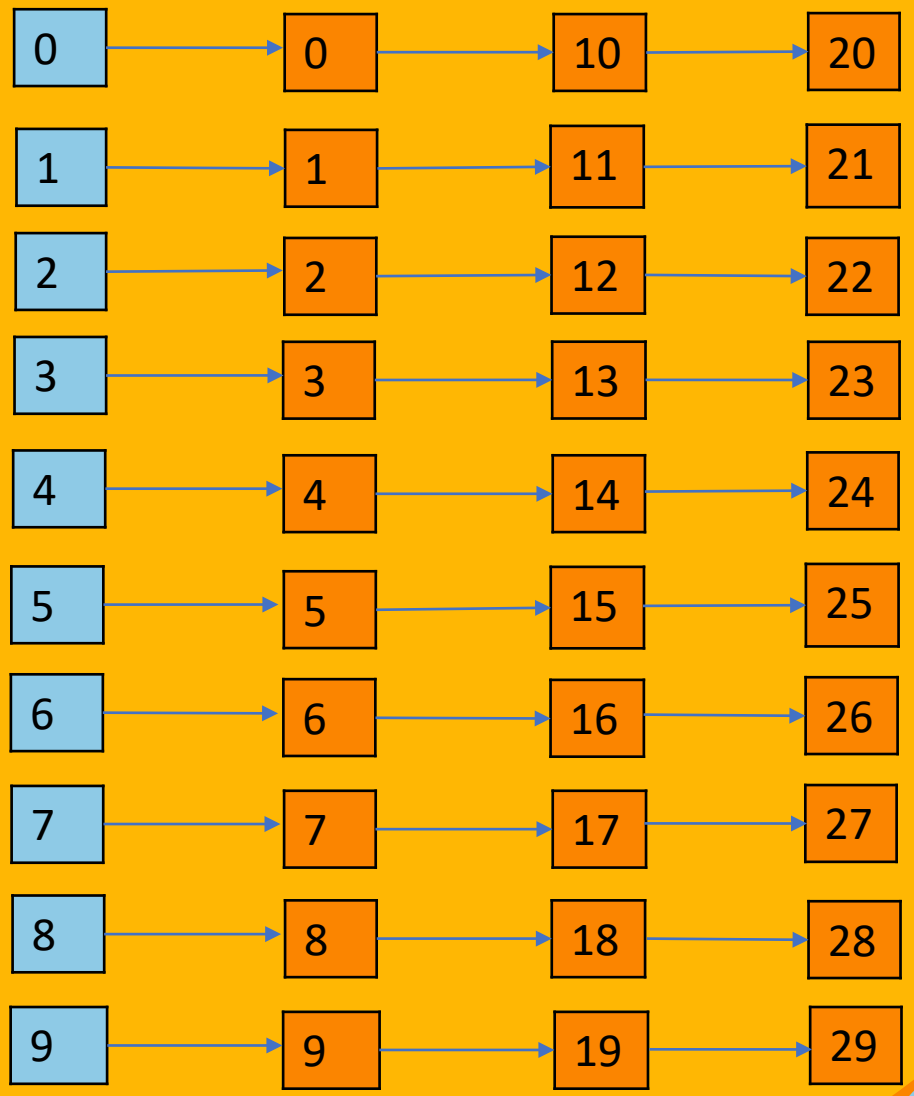
Direct Chaining Method

- ***Khi thêm một phần tử*** có khóa k vào bảng băm, hàm băm h(k) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1 ứng với danh sách liên kết i mà phần tử này sẽ được thêm vào.
- ***Khi tìm một phần tử*** có khóa k trong bảng băm, hàm băm h(k) cũng sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1 ứng với danh sách liên kết i có thể chứa phần tử này. Như vậy, việc tìm kiếm phần tử trên bảng băm sẽ được qui về bài toán tìm kiếm một phần tử trên danh sách liên kết.

XÉT BẢNG BĂM NHƯ SAU:

- Tập khóa K: các số tự nhiên từ {0,1,2,3,…,29}.
- Tập địa chỉ M: gồm 10 địa chỉ (M={0, 1, …, 9})
- Hàm băm h(key) = key % 10.

XÉT BẢNG BĂM NHƯ SAU:



- Hình bên minh họa bảng băm vừa mô tả. Theo hình vẽ, bảng băm đã "băm" phần tử trong tập khoá K theo 10 danh sách liên kết khác nhau, mỗi danh sách liên kết gọi là một bucket:
- Bucket i(i=0 | … | 9) gồm những phần tử có khóa tận cùng bằng i.
- Khi khởi động bảng băm, con trỏ đầu của các bucket là NULL.

Direct Chaining Method

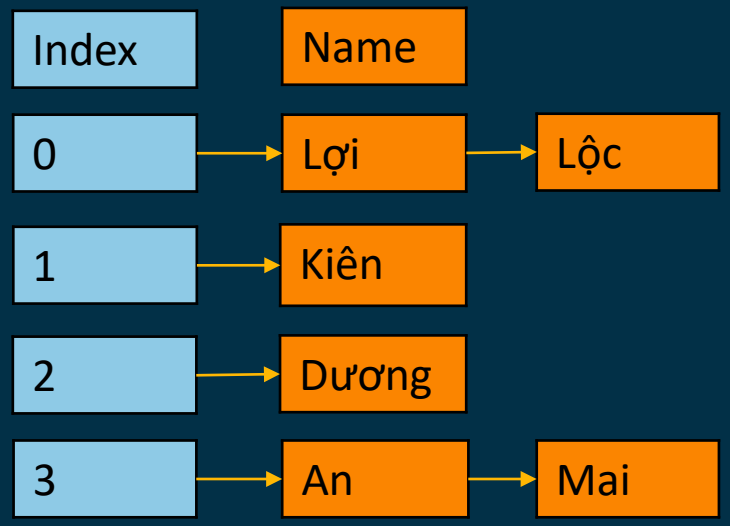
- Ví dụ: Hãy vẽ HashTable dựa trên phương pháp nối kết trực tiếp với dữ liệu từ bảng sau:

MSSV	Name	Index
23520031	An	3
23520043	Mai	3
23520789	Kiên	1
23520872	Lợi	0
23520856	Lộc	0
23520798	Dương	2

Direct Chaining Method

- Ví dụ: Hãy vẽ HashTable dựa trên phương pháp nối kết trực tiếp với dữ liệu từ bảng sau:

MSSV	Name	Index
23520031	An	3
23520043	Mai	3
23520789	Kiên	1
23520872	Lợi	0
23520856	Lộc	0
23520798	Dương	2



Direct Chaining Method: Cài đặt

```
#define M 100
struct Node {
    int key;
    Node *Next;
};

/* Khai báo mảng HASHTABLE chứa M
con trỏ đầu của HASHTABLE */
Node *HASHTABLE[M];.
```

Hàm băm: Giả sử chúng ta chọn hàm băm h(key) = key % 10.

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

Phép toán khởi tạo bảng băm: Khởi động các Bucket của HashTable bằng NULL.

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

Phép toán kiểm tra bucket thứ i rỗng:

```
bool isBucketEmpty(int i)
{
        return(HASHTABLE[i] == NULL ? true : false);
}
```

Phép toán kiểm tra toàn bộ bảng băm rỗng:

```
bool isHTEmpty()
{
    for (int i = 0; i<M; i++)
            if (HASHTABLE[i] != NULL)
                return false;
    return true;
}
```

- **Phép toán Insert:** Thêm phần tử có khóa k vào bảng băm:
  - Trước tiên, chúng ta xác định bucket của khóa k bằng hàm băm (Hash function).
  - Sau đó, thêm khóa k vào bucket như thực hiện chèn phần tử vào Linked List.

```
void Insert(int k)
{
        int i = HF(k);
        // Tạo một node mới chứa khóa
        Node *newNode = new Node;
        newNode->key = k;
        newNode->next = NULL;

}
```

Cài đặt

- **Phép toán Insert:** Thêm phần tử có khóa k vào bảng băm:
  - Trước tiên, chúng ta xác định bucket của khóa k bằng hàm băm (Hash function).
  - Sau đó, thêm khóa k vào bucket như thực hiện chèn phần tử vào Linked List.

```
void Insert(int k)
{
    int i = HF(k);
    // Tạo một node mới chứa khóa
    Node *newNode = new Node;
    newNode->key = k;
    newNode->next = NULL;
    // Nếu không có phần tử nào tại vị trí băm này, thêm phần tử mới làm đầu danh sách
    if (HASHTABLE[i] == NULL)
        HASHTABLE[i] = newNode;
    else
    {
    // Nếu có phần tử tại vị trí băm này, thêm phần tử mới vào cuối danh sách
    Node* temp = HASHTABLE[i];
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
```

- **Phép toán Remove:**

```
void Remove(int k) {
        int i = HF(k);
        // Nếu không có phần tử nào tại vị trí băm
này, không cần thực hiện gì cả
        if (HASHTABLE[i] == NULL) {
                return;
        }
        // Nếu phần tử cần xóa là phần tử đầu tiên
của danh sách tại vị trí băm này
        if (HASHTABLE[i]->key == k) {
                Node* temp = HASHTABLE[i];
                HASHTABLE[i]= HASHTABLE[i]->next;
                delete temp;
                return;
        }
```

- **Phép toán Remove:**

```
void Remove(int k) {
        int i = HF(k);
        // Nếu không có phần tử nào tại vị trí băm
này, không cần thực hiện gì cả
        if (HASHTABLE[i] == NULL) {
                return;
        }
        // Nếu phần tử cần xóa là phần tử đầu tiên
của danh sách tại vị trí băm này
        if (HASHTABLE[i]->key == k) {
                Node* temp = HASHTABLE[i];
                HASHTABLE[i]= HASHTABLE[i]->next;
                delete temp;
                return;
        }
// Nếu phần tử cần xóa không phải là phần tử đầu tiên
tìm và xóa phần tử
        Node* prev = HASHTABLE[i];
        Node* curr = HASHTABLE[i]->next;
        while (curr != NULL) {
            if (curr->key == k) {
                prev->next = curr->next;
                delete curr;
                return;
                }
            prev = curr;
            curr = curr->next;
        }
        }
```

Phép toán tìm kiếm:

```
Node* Search(int k)
{
	int i = HF(k);
	Node *p = HASHTABLE[i];

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

	return p;
}
```

Phép toán duyệt các Bucket[i]:

Duyệt các phần tử trong bucket thứ i:

```
void traverseBucket(int i) {
    Node *p = HASHTABLE[i];
    while (p != NULL) {
        cout << p->key << "\t";
        p = p->pNext;
    }
}
```

Phép toán duyệt toàn bộ bảng băm:

```
void traverseHashTable(){
    for (int i = 0; i<M; i++)
    {
        cout << endl << "Bucket " <<
        i << ": ";
        traverseBucket(i);
    }
}
```

Các phương pháp giải quyết đụng độ

- **Direct Chaining Method - PP nối kết trực tiếp**
- **Coalesced Chaining Method – PP nối kết hợp nhất**

Coalesced Chaining Method

- Bảng băm trong trường hợp này được cài đặt bằng mảng, với ý tưởng như là Linked List, có M nút. Mỗi nút của bảng băm là một node có 2 trường:
  - Trường key: chứa các khóa node.
  - Trường next: lưu chỉ số của node kế tiếp nếu xảy ra xung đột.
- Khi khởi động bảng băm thì tất cả trường key được gán nullkey, tất cả trường next được gán -1.

Coalesced Chaining Method

- Bảng băm trong trường hợp này được cài đặt bằng mảng, với ý tưởng như là Linked List, có M nút. Mỗi nút của bảng băm là một node có 2 trường:
  - Trường key: chứa các khóa node.
  - Trường next: lưu chỉ số của node kế tiếp nếu xảy ra xung đột.
- Khi khởi động bảng băm thì tất cả trường key được gán nullkey, tất cả trường next được gán -1.

Key	Next
nullkey	-1
…	…
nullkey	-1

Coalesced Chaining Method

- Khi thêm một nút có khóa key vào bảng băm, hàm băm H(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 H(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.

Xét bảng băm sau:

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

- Cho bảng băm có tập khóa là tập số tự nhiên, tập địa chỉ có 10 địa chỉ (từ địa chỉ 0 đến 9), chọn hàm băm h(key) = key % 10.
- Tập keys = {30, 24, 26, 10, 14, 54, 4}.

Xét bảng băm sau:

0	30	9
1	Nullkey	-1
2	Nullkey	-1
3	Nullkey	-1
4	24	8
5	4	-1
6	26	-1
7	54	5
8	14	7
9	10	-1

- 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 h(key) = key % 10.
- Tập keys = {30, 24, 26, 10, 14, 54, 4}.

Xét bảng băm sau:

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

- 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 h(key) = key % 10.
- Tập keys = {30, 24, 26, 10, 14, 54, 4, 8, 84}.

Xét bảng băm sau:

0	30	9
1	Nullkey	-1
2	84	-1
3	8	-1
4	24	8
5	4	2
6	26	-1
7	54	5
8	14	7
9	10	-1

- 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 h(key) = key % 10.
- Tập keys = {30, 24, 26, 10, 14, 54, 4, 8, 84}.

Xét bảng băm sau:

0		
1		
2		
3		
4		
5		
6		
7		
8		
9		

- 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 h(key) = key % 10.
- Tập keys = {20, 31, 10, 51, 84, 50, 1, 24, 90}.

Xét bảng băm sau:

0	20	9
1	31	8
2	Nullkey	-1
3	90	-1
4	84	5
5	24	-1
6	1	-1
7	50	3
8	51	6
9	10	7

- 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 h(key) = key % 10.
- Tập keys = {20, 31, 10, 51, 84, 50, 1, 24, 90}.

Coalesced Chaining Method

- Thực chất cấu trúc bảng băm này chỉ tối ưu khi băm đều, nghĩa là mỗi danh sách liên kết chứa một vài phần tử bị xung đột, tốc độ truy xuất lúc này có bậc O(1).
- Trường hợp xấu nhất là băm không đều vì hình thành một danh sách có n phần tử nên tốc độ truy xuất lúc này có bậc O(n).

Coalesced Chaining Method: Cài đặt

Cài đặt

Khai báo cấu trúc bảng băm:

```
#define nullkey -1
#define M 100
// Khai báo cấu trúc một node
trong bảng băm
struct Node{
    int key;
    int next;
};
// Khai báo bảng băm
Node HASHTABLE[M];
int avail; // biến toàn cục chỉ
nút trống ở cuối bảng băm được
cập nhật khi có xung đột
```

Cài đặt

Hàm băm:

```
// Giả sử chúng ta chọn hàm băm dạng
mod: h(key) = key % 10.
int HF(int key) {
    return key % 10;
}
```

Phép toán khởi tạo:

```
void Initialize() {
    for (int i = 0; i<M; i++){
        HASHTABLE[i].key = nullkey;
        HASHTABLE[i].next = -1;
    }
    avail = M - 1; // nút M-1 là nút ở
    cuối bảng chuẩn bị cấp phát nếu có
    xung đột.
}
```

Cài đặt

Phép toán tìm kiếm:

```
int Search(int k) {
    int i = HF(k);
    while (k != HASHTABLE[i].key &&
    i != -1)
            i = HASHTABLE[i].next;

    if (k == HASHTABLE[i].key)
            return i; //tìm thấy

    return M; //không tìm thấy
}
```

Phép toán lấy phần tử trống cuối bảng băm:
Chọn phần tử còn trống phía cuối bảng băm để cấp phát khi xảy ra xung đột.

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

Cài đặt

Phép toán Insert

```
int Insert(int k) {
        int i = Search(k), j; // j là địa chỉ nút trống
        được cấp phát
        if (i != M) {
            cout << "\n Khoa " << k << " bi trung,
            khong them nut nay duoc!"; return i;
        }
        i = HF(k);
        while (HASHTABLE[i].next >= 0)
              i = HASHTABLE[i].next;
        if (HASHTABLE[i].key == nullkey)
              j = i; // Nút i còn trống thì cập nhật
        else
        { // Nếu nút i là nút cuối của danh sách
             j = getEmpty();
             if (j < 0) { cout << "\n Bang bam bi day!";
             return j; }
             HASHTABLE[i].next = j;
        }
        HASHTABLE[j].key = k;
        return j;
}
```

Cài đặt

Phép toán Remove:

```
void Remove(int k) {
        int i = Search(k), prev = -1;
        while (index != -1 && HASHTABLE[i].key != k) {
                        prev = i;
                        i = HASHTABLE[i].next;
        }
        if (i == -1)
              return;
        if (prev == -1) {
                        HASHTABLE[i].key = -1;
                        int nextIndex = HASHTABLE[i].next;
                        HASHTABLE[i].next = -1;
                        if (nextIndex != -1) {
                                        HASHTABLE[i].key =
HASHTABLE[nextIndex].key;
                                        HASHTABLE[i].next =
HASHTABLE[nextIndex].next;
                                        HASHTABLE[nextIndex].key = -1;
                                        HASHTABLE[nextIndex].next = -1;
                                        avail = nextIndex;
                        }
                        return;  }
```

Cài đặt

Phép toán Remove:

```
void Remove(int k) {
        int i = Search(k), prev = -1;
        while (index != -1 && HASHTABLE[i].key != k) {
                        prev = i;
                        i = HASHTABLE[i].next;
        }
        if (i == -1)
              return;
        if (prev == -1) {
                        HASHTABLE[i].key = -1;
                        int nextIndex = HASHTABLE[i].next;
                        HASHTABLE[i].next = -1;
                        if (nextIndex != -1) {
                                        HASHTABLE[i].key =
HASHTABLE[nextIndex].key;
                                        HASHTABLE[i].next =
HASHTABLE[nextIndex].next;
                                        HASHTABLE[nextIndex].key = -1;
                                        HASHTABLE[nextIndex].next = -1;
                                        avail = nextIndex;
                        }
                        return;
```

```
// Xóa phần tử bằng cách cập nhật
con trỏ của phần tử trước nó
                HASHTABLE[prev].next =
HASHTABLE[i].next;
                HASHTABLE[i].key = -1;
                HASHTABLE[i].next = -1;
}
```

Các phương pháp giải quyết đụng độ

- **Direct Chaining Method - PP nối kết trực tiếp**
- **Coalesced Chaining Method – PP nối kết hợp nhất**
- **Linear Probing Method – PP dò tuyến tính**

Linear Probing Method

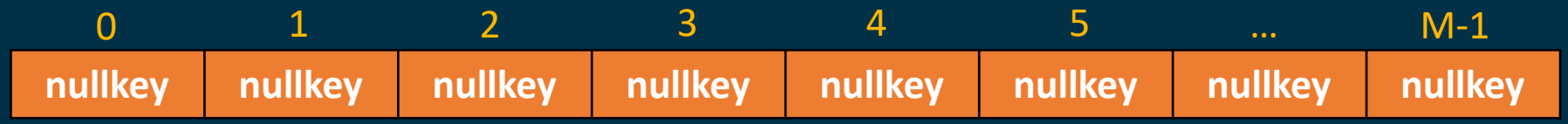
- Hàm băm lại của phương pháp dò tuyến tính là truy xuất địa chỉ kế tiếp. Hàm băm lại lần i được biểu diễn bằng công thức sau:

  $$H(key, i) = (h(key) + i) \bmod M$$

  với h(key) là hàm băm chính của bảng băm, f(i)=i.
- Lưu ý: Địa chỉ dò tìm kế tiếp là địa chỉ 0 nếu đã dò đến cuối bảng.

Linear Probing Method

- Bảng băm trong trường hợp này được cài đặt bằng danh sách kề có M nút, mỗi nút của bảng băm có một trường key để chứa khóa của nút.
- Khi khởi động bảng băm thì tất cả trường key được gán nullkey.



Linear Probing Method

- Khi thêm nút có khoá key vào bảng băm, hàm băm h(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1:
  - Nếu chưa bị xung đột thì thêm phần tử mới vào địa chỉ này.
  - Nếu bị xung đột thì hàm băm băm lại lần 1, hàm h_1 sẽ xét địa chỉ kế tiếp, nếu lại bị xung đột thì hàm băm băm lại lần 2, hàm h_2 sẽ xét địa chỉ kế tiếp nữa,…, và quá trình cứ thế cho đến khi nào tìm được địa chỉ trống và thêm phần tử mới vào địa chỉ này.

- Khi tìm một nút có khoá key trong bảng băm, hàm băm h(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.

Xét bảng băm sau:

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

- Giả sử, khảo sát bảng băm có cấu trúc như sau:
  - Tập khóa K: tập số tự nhiên
  - Tập địa chỉ: gồm 10 địa chỉ {0, 1, …, 9}, M=10.
  - Hàm băm h(key) = key mod 10 và H(key, i) = (h(key) + i) % M.
- Hình thể hiện thêm các giá trị 41, 45, 51, 30, 62, 80, 53, 89, 77, 19 vào bảng băm.

Xét bảng băm sau:

0	30
1	41
2	51
3	62
4	80
5	45
6	53
7	77
8	19
9	89

- Giả sử, khảo sát bảng băm có cấu trúc như sau:
  - Tập khóa K: tập số tự nhiên
  - Tập địa chỉ: gồm 10 địa chỉ {0, 1, .. 9}, M=10.
  - Hàm băm h(key) = key mod 10 và H(key, i) = (h(key) + i) % M.
- Hình thể hiện thêm các giá trị 41, 45, 51 30, 62, 80, 53, 89, 77, 19 vào bảng băm.

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 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).

Linear Probing Method: Cài đặt

Khai báo cấu trúc bảng băm:

```
#define nullkey -1
#define M 100
struct NODE
{
        int key; //khóa của node trên bảng băm
};
NODE HASHTABLE[M];
int N; //biến toàn cục chỉ số nút hiện có trên bảng băm
```

Hàm băm: Giả sử chúng ta chọn hàm băm dạng %: H(key) = key % 10.

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

Phép toán khởi tạo: Gán tất cả các phần tử trên bảng có trường key là nullkey.

```
void Initialize() {
    for (int i = 0; i<M; i++)
        HASHTABLE[i].key =  nullkey;
    N = 0; //số node hiện có khởi động bằng 0
}
```

Phép toán kiểm tra trống:

```
int isEmpty () {
    return N == 0 ? 1 : 0;
}
```

Phép toán kiểm tra đầy:

```
int isFull () {
    return N == M-1 ? 1 : 0;
}
```

Cài đặt

Phép toán Search: // Việc tìm kiếm phần tử có khoá k trên một khối đặc, bắt đầu từ một địa chỉ i = HF(k), nếu không tìm thấy phần tử có khoá k, hàm này sẽ trả về -1, còn nếu tìm thấy, hàm này trả về địa chỉ tìm thấy.

```
int Search (int k) {
   int pos = HF(k);
   while ( HASHTABLE[pos].key != k && HASHTABLE[pos].key != nullkey) {
            pos++;
            if (pos >= M) pos-=M;
   }
   if (HASHTABLE[pos].key == k) //tìm thấy
      return pos;
   return -1; // không tìm thấy
}
```

Cài đặt

Phép toán Insert: // Thêm phần tử có khoá k vào bảng băm.

```
int Insert (int k) {
   if (isFull()) return -2; // HashTable đầy
   int pos = HF(k), i=0;
   while (HASHTABLE[pos].key !=  nullkey) {
            pos++;
            if (pos >= M) pos-=M;
   }

   if (HASHTABLE[pos].key == k)
      return -1; // giá trị k đã tồn tại trong mảng

   HASHTABLE[pos].key = k;
   N+= 1;
   return pos;
}
```

Cài đặt

Phép toán Remove:

```
// xóa node tại vị trí i trên bảng băm.
void Remove(int i) {
    int j, r, a, cont=true;
    do {
        HASHTABLE[i].key = nullkey;
        j = i;
        do {
          i = i + 1;
          if (i >= M) i = i- M;
          if (HASHTABLE[i].key == nullkey) cont = false;
          else {
            r = HF(HASHTABLE[i].key);
            a = (j <r && r <= i) || (r <= i && i <j) || (i<j &&
            j <r);
          }
          } while (cont && a);
        if (cont) HASHTABLE[j].key = HASHTABLE[i].key;
    } while (cont);
}
```

Các phương pháp giải quyết đụng độ

- **Direct Chaining Method - PP nối kết trực tiếp**
- **Coalesced Chaining Method – PP nối kết hợp nhất**
- **Linear Probing Method – PP dò tuyến tính**
- **Quadratic Probing Method – PP dò bậc hai**

Quadratic Probing Method

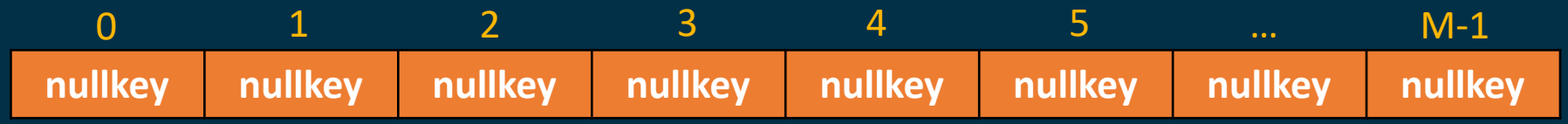
- Hàm băm lại của phương pháp dò bậc hai là truy xuất các địa chỉ cách bậc 2. Hàm băm lại hàm i được biểu diễn bằng công thức sau:

  $H(key, i) = (h(key) + i^2) \% M$

  với h(key) là hàm băm chính của bảng băm, $f(i) = i^2$
- Nếu đã dò đến cuối bảng thì trở về dò lại từ đầu bảng.

Quadratic Probing Method

- Bảng băm trong trường hợp này được cài đặt bằng danh sách kề có M nút, mỗi nút của bảng băm có một trường key để chứa khóa của nút.
- Khi khởi động bảng băm thì tất cả trường key được gán nullkey.



Quadratic Probing Method

- Khi thêm nút có khoá key vào bảng băm, hàm băm h(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1:
  - Nếu chưa bị xung đột thì thêm phần tử mới vào địa chỉ này.
  - Nếu bị xung đột thì hàm băm băm lại lần 1, sẽ xét địa chỉ cách 1^2, nếu lại bị xung đột thì hàm băm băm lại lần 2, sẽ xét địa chỉ cách 2^2,…, và quá trình cứ thế cho đến khi nào tìm được địa chỉ trống và thêm phần tử mới vào địa chỉ này.

- Khi tìm một nút có khoá key trong bảng băm, hàm băm h(key) sẽ xác định địa chỉ i trong khoảng từ 0 đến M-1, 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).

Xét bảng băm sau:

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

- Giả sử, khảo sát bảng băm có cấu trúc như sau:
  - Tập khóa K: tập số tự nhiên
  - Tập địa chỉ: gồm 10 địa chỉ {0, 1, …, 9}, M=10.
  - Hàm băm h(key) = key mod 10 và $H(key, i) = (h(key) + i^2) \% M$.
- Hình thể hiện thêm các giá trị 10, 15, 16, 20, 30, 25, 26, 36 vào bảng băm.

Xét bảng băm sau:

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

- Giả sử, khảo sát bảng băm có cấu trúc như sau:
  - Tập khóa K: tập số tự nhiên
  - Tập địa chỉ: gồm 10 địa chỉ {0, 1, …, 9}, M=10.
  - Hàm băm h(key) = key mod 10 và $H(key, i) = (h(key) + i^2) \% M$.
- Hình thể hiện thêm các giá trị 10, 15, 16, 20, 30, 25, 26, 36 vào bảng băm.

Quadratic Probing Method

- 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.

Quadratic Probing Method: Cài đặt

Cài đặt

```
#define nullkey -1
#define M 100

// Khai báo node của bảng băm
struct NODE {
        int key; // Khóa của node
trên bảng băm
};

// Khai báo bảng băm có M node
NODE HASHTABLE[M];

// Biến toàn cục chỉ số nút hiện
có trên bảng băm
int  N;
```

Hàm băm: Giả sử chúng ta chọn hàm băm dạng: H(key) = key % 10.

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

Phép toán khởi tạo: Gán tất cả các phần tử trên bảng có trường key là nullkey. Gán biến toàn cục N=0.

```
void Initialize() {
    for (int i = 0; i<M; i++)
                HASHTABLE[i].key =
    nullkey;
    N = 0; // Số node hiện có ban đầu
    bằng 0
}
```

Phép toán kiểm tra trống:

```
int isEmpty() {
        return N == 0 ? 1 : 0;
}
```

Phép toán kiểm tra đầy:

```
int isFull() {
        return N == M - 1 ? 1 : 0;
}
```

Phép toán tìm kiếm: 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 = HF(k), d = 1;
    while (HASHTABLE[i].key != k &&
    HASHTABLE[i].key != nullkey) {
       // Băm lại theo phương pháp dò
    bậc hai
        i = (i + d) % M;
        d = d + 2;
    }
    if (HASHTABLE[i].key == k)
            return i; //tìm thấy
    return M;  // không tìm thấy
}
```

Phép toán Insert: Thêm phần tử có khoá k vào bảng băm.

```
int Insert(int k) {
    int i , d;
    if (Search(k)<M)
        return M; // Trùng khoá
    if (isFull()) {
        cout << “Full!";
        return;
    }
    i = HF(k);
    d = 1;
    while (HASHTABLE[i].key != nullkey) {
        i = (i + d) % M;
        d = d + 2;
    }
    HASHTABLE[i].key = k;
    N = N + 1;
    return i;
}
```

Các phương pháp giải quyết đụng độ

- **Direct Chaining Method - PP nối kết trực tiếp**
- **Coalesced Chaining Method – PP nối kết hợp nhất**
- **Linear Probing Method – PP dò tuyến tính**
- **Quadratic Probing Method – PP dò bậc hai**
- **Double Probing Method – PP băm kép**

Double Probing Method

- Hàm băm lại lần i được biểu diễn bằng công thức sau:
  H(key, i) = (h1(key) + i* h2(key)) % M
  với h1(key) là hàm băm chính của bảng băm, f(i)= i* h2(key).
- Nếu đã dò đến cuối bảng thì trở về dò lại từ đầu bảng.

Double Probing 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 kỳ, ví dụ chọn hai hàm băm như sau:

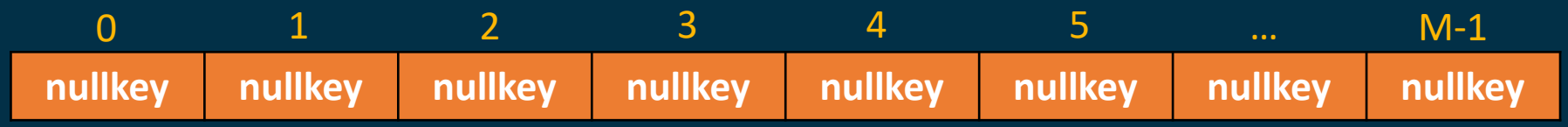
h1(key) = key % M

h2(key) = R- key % R (thường chọn R là số nguyên tố nhỏ hơn M)

- Bảng băm trong trường hợp này được cài đặt bằng danh sách kề có M phần tử, mỗi phần tử của bảng băm có một trường key để lưu khoá các phần tử.

Double Probing Method

- **Khi khởi động** bảng băm, tất cả trường key được gán nullkey.



Double Probing Method

- **Khi thêm phần tử** có khoá key vào bảng băm, thì **i=h1(key)** và **j=h2(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 chi này.

Double Probing Method

- **Khi thêm phần tử** có khoá key vào bảng băm, thì **i=h1(key)** và **j=h2(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, 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.

Double Probing Method

- **Khi tìm kiếm** một phần tử có khoá key trong bảng băm, hàm băm **i=h1(key)** và **j=h2(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).

Xét bảng băm sau:

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

- Giả sử, khảo sát bảng băm có cấu trúc như sau:
  - Tập khóa K: tập số tự nhiên
  - Tập địa chỉ: gồm 10 địa chỉ {0, 1, …, 9}, M=10.
  - Hàm băm h1(key) = key mod 10, h2(key) = R- key % R (R là số nguyên tố nhỏ hơn M) và H(key, i) = (h1(key) + i* h2(key)) % M.
- Hình thể hiện thêm các giá trị 89, 18, 49, 58, 69 vào bảng băm.

Xét bảng băm sau:

0	69
1	Nullkey
2	Nullkey
3	58
4	Nullkey
5	Nullkey
6	49
7	Nullkey
8	18
9	89

- Giả sử, khảo sát bảng băm có cấu trúc như sau:
  - Tập khóa K: tập số tự nhiên
  - Tập địa chỉ: gồm 10 địa chỉ {0, 1, …, 9}, M=10.
  - Hàm băm h1(key) = key mod 10, h2(key) = R- key % R (R là số nguyên tố nhỏ hơn M) và H(key, i) = (h1(key) + i* h2(key)) % M.
- Hình thể hiện thêm các giá trị 89, 18, 49, 58, 69 vào bảng băm.

Double Probing Method

- Bảng băm được cài đặt theo cấu trúc này linh hoạt hơn bảng băm dùng phương pháp dò tuyến tính và bảng băm dùng phương pháp dò bậc hai, do dùng hai hàm băm khác nhau nên việc rải phần tử mang tính ngẫu nhiên hơn, nếu bảng băm chưa đầy tốc độ truy xuất có bậc O(1).
- Trường hợp xấu nhất là bảng băm gần đầy, tốc độ truy xuất chậm do thực hiện nhiều lần so sánh.

Double Probing Method: Cài đặt

Cài đặt

```
#define nullkey -1
#define M 100
// Khai báo cấu trúc một node của
bảng băm
struct NODE{
        int key; // Khóa của nút
trên bảng băm
};

// Khai báo bảng băm có M nút
NODE HASHTABLE[M];

// Biến toàn cục chỉ số nút hiện
có trên bảng băm
int N;
```

Cài đặt

Hàm băm: Giả sử chúng ta chọn hai hàm băm dạng %:

h1(key)=key % M và h2(key) =(M-2) – key % (M-2).

```
//Hàm băm thứ nhất
int HF(int key) {
	return key % M;
}

//Hàm băm thứ hai
int HF2(int key) {
	return (M – 2) - key % (M-2);
}
```

Phép toán Khởi tạo:

- Khởi động bảng băm.
- Gán tất cả các phần tử trên bảng có trường key là nullkey.
- Gán biến toàn cục N = 0.

```
void Initialize() {
    for (int i = 0; i<M; i++)
            HASHTABLE[i].key =
    nullkey;
    N = 0; // số nút hiện có khởi
    động bằng 0
}
```

```
//Kiểm tra bảng băm có rỗng không
int isEmpty() {
	return N == 0 ? true : false;
}

// Kiểm tra bảng băm đã đầy chưa
int isFull() {
	return N == M - 1 ? true :
false;
}
```

Phép toán tìm kiếm:

```
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)

    // tìm thấy
    if (HASHTABLE[i].key == k)
            return i;

    // không tìm thấy
    return M;
}
```

Phép toán Insert: Thêm phần tử có khoá k vào bảng băm.

```
int Insert(int k) {
    int i, j;
    if (Search(k) < M) {
        cout << "Da co khoa nay
        trong bang bam!";
        return M;
    }

    if (isFull()) {
        cout << "Bang bam bi
        day!";
        return M;
    }
        i = HF(k);
        j = HF2(k);
        while (HASHTABLE[i].key
        != nullkey){
            // Băm lại theo
            phương pháp băm kép
            i = (i + j) % M;
            }
        HASHTABLE[i].key = k;
        N = N + 1;
        return i;
}
```

Cài đặt HashTable bằng thư viện

```cpp
#include <iostream>
#include <unordered_map>
using namespace std;
int main() {
    // Khai báo một unordered_map
    unordered_map<int, int> hashMap;
    // Thêm các phần tử vào bảng băm
    hashMap[5] = 50;
    hashMap[10] = 100;
    hashMap[15] = 150;
    // Truy cập và hiển thị giá trị của một phần tử
    cout << "Value at key 10: " << hashMap[10] << endl;
    // Xóa một phần tử
    hashMap.erase(10);
    // Kiểm tra xem một khóa có tồn tại trong bảng băm không
    if (hashMap.find(10) == hashMap.end()) {
        cout << "Key 10 not found in hash map." << endl;
    }
    // Duyệt qua tất cả các phần tử của bảng băm
    cout << "Hash map:" << endl;
    for (const auto& pair : hashMap) {
        cout << "Key: " << pair.first << ", Value: " << pair.second << endl;
    }
    return 0;
}
```

Thanks You