HOA SEN UNIVERSITY

www.hoasen.edu.vn

Cấu Trúc Dữ Liệu & Giải Thuật

Data Structures & Algorithms

GV: Phan Hồng Trung

HOA SEN UNIVERSITY

www.hoasen.edu.vn

Bài 9 – Bảng Băm (Hash Table)

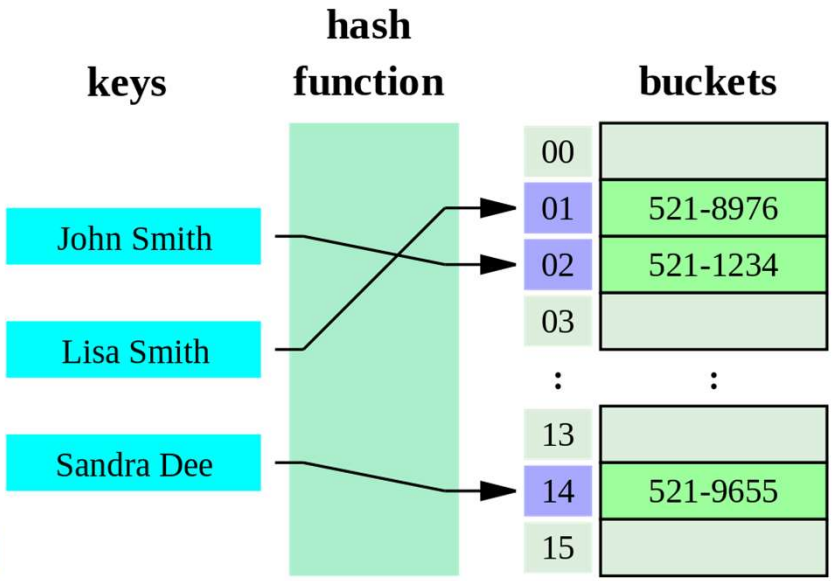
- Giới thiệu bảng băm
- Hàm băm và tính chất
- Xử lý xung đột
- Cài đặt bảng băm
- Phân tích độ phức tạp
- Ứng dụng thực tế

Tại sao phải băm?

- Trong mảng được sắp xếp:
  - Tìm kiếm nhị phân: O(log n).
  - Thêm. xóa: O(n).
- Trong cây AVL:
  - Thêm, xóa, tìm kiếm: O(log n).
- Có cấu trúc dữ liệu nào mà việc thêm, xóa và tìm kiếm các phần tử hiệu quả hơn không?
- Câu trả lời là "Có", đó là bảng băm (hash table).

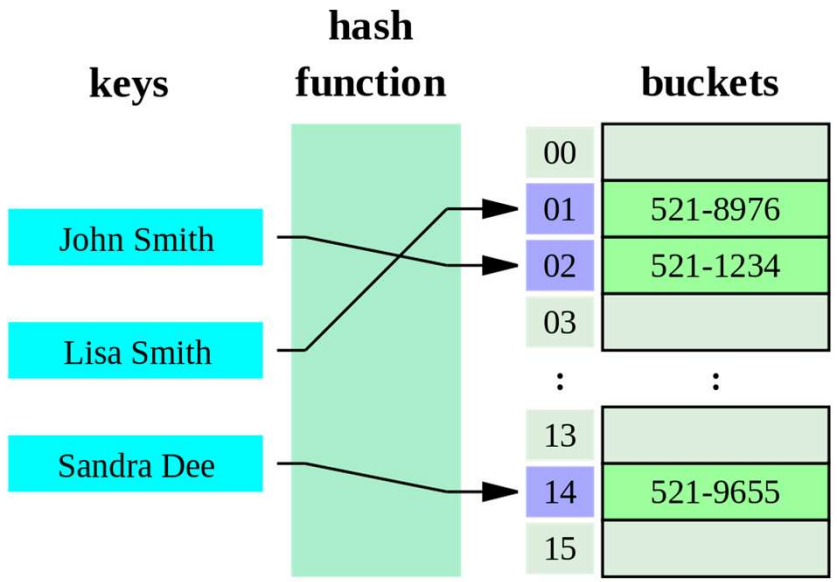
Giới thiệu bảng băm

- Bảng băm là cấu trúc dữ liệu ánh xạ (map) một khóa (key) thành một giá trị (value) bằng cách sử dụng hàm băm (hash function).
- Thực chất bảng băm là một mảng.
- Trong mảng: các phần tử được truy cập bằng index.
  - VD: a[0], a[4]
- Trong bảng băm: các phần tử được truy cập bằng key.
  - VD: ht["John Smith"], ht["Lisa Smith"]



Giới thiệu bảng băm

- Khóa có thể là một số nguyên, một chuỗi…
- Key –hash→ Index
- Ví dụ:
  - hash("John Smith") = 2
  - hash("Lisa Smith") → 1
- Key –Hash Table→ Value
- Ví dụ:
  - HT["John Smith"] → "521-1234"
  - HT["Lisa Smith"] → "521-8976"



Giới thiệu Hash Table

- Ưu điểm:
  - Truy cập nhanh: thêm, xóa, tìm kiếm: O(1) (trường hợp trung bình).
  - Lưu trữ linh hoạt.
- Hạn chế:
  - Phụ thuộc vào chất lượng của hàm băm.
  - Xử lý xung đột phức tạp.
  - Không hiệu quả trong các thao tác cần thông tin có thứ tự như: tìm min, tìm max, sắp thứ tự.

Hàm băm (Hash function)

- Hàm băm ánh xạ (map) một **khóa** (key) thành một **chỉ số** (index) trong mảng.
- Yêu cầu:
  - Xử lý nhanh.
  - Phân phối đều các khóa.
  - Hạn chế xung đột.
- Ví dụ hàm băm đơn giản: h(k) = k % m
- Trong đó:
  - k: Khóa
  - m: Kích thước bảng băm
- Ví dụ: m = 11
  - h(5) = 5
  - h(15) = 4
  - h(22) = 0

HOA SEN UNIVERSITY

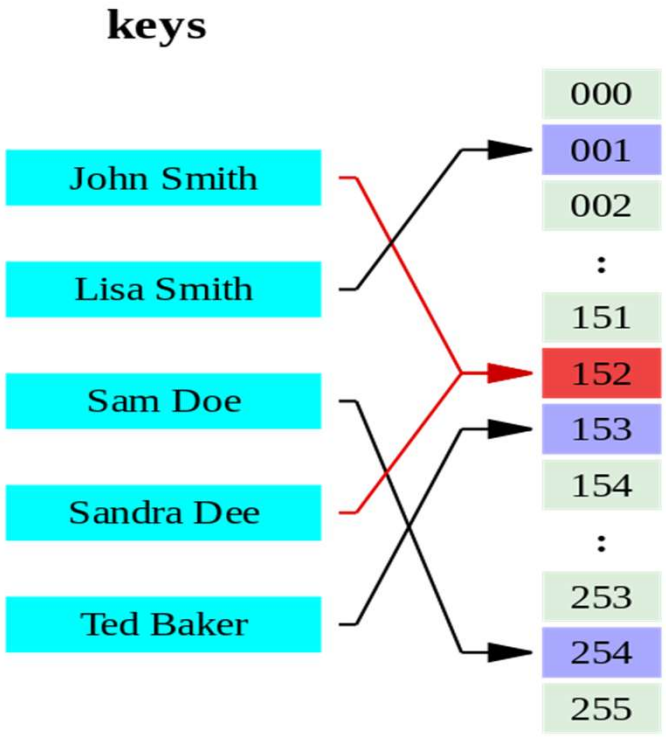
www.hoasen.edu.vn

Hàm băm (Hash function)

- Các vấn đề của hàm băm:
  - Các khóa có thể không phải là số.
  - Số lượng khóa có thể lớn hơn nhiều so với không gian có sẵn trong bảng.
  - Các khóa khác nhau có thể ánh xạ vào cùng một vị trí:
    - Hàm băm là ánh xạ nhiều-một → xung đột (collision).
    - Nếu có quá nhiều xung đột, hiệu suất của bảng băm sẽ bị ảnh hưởng đáng kể.
- Nếu các khóa là số nguyên thì chỉ cần dùng Key % TableSize.
- Nếu các khóa là chuỗi, hãy chuyển đổi khóa thành số.

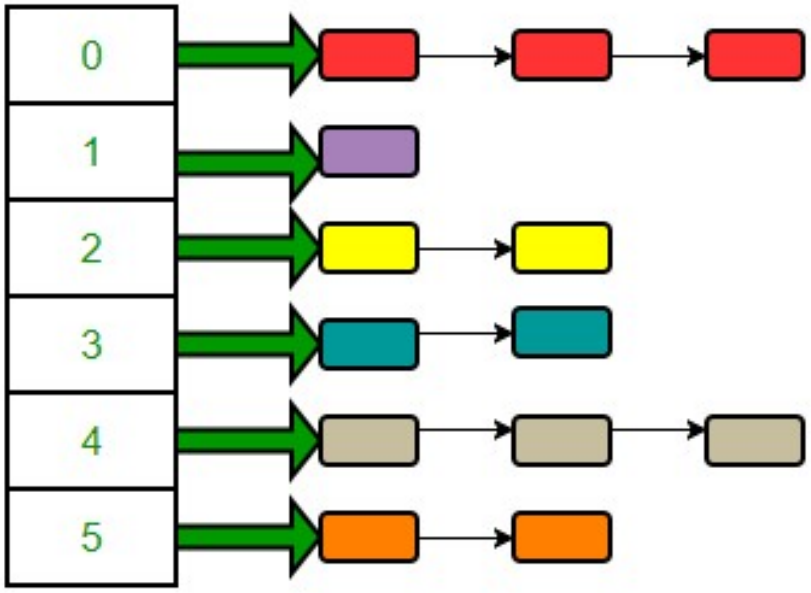
Giải quyết xung đột (Collision Resolution)

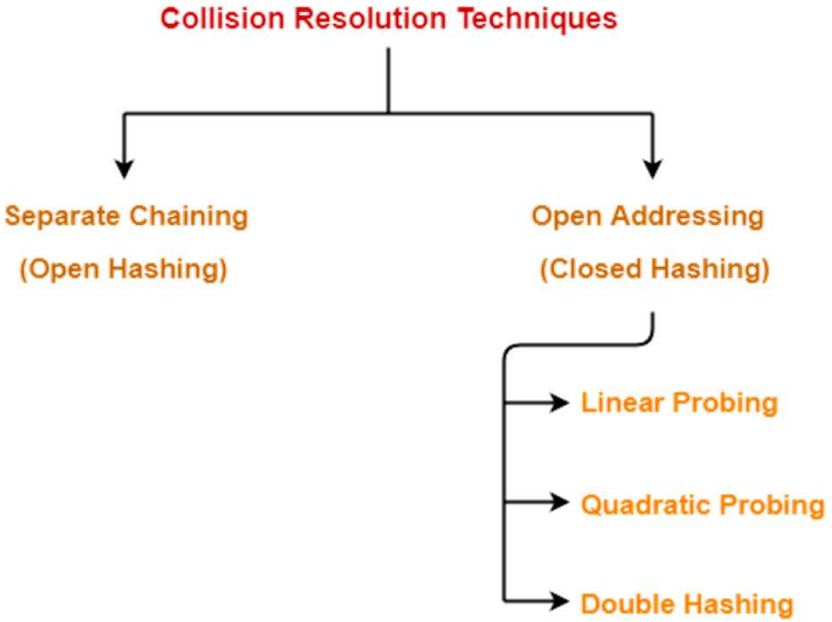
- Xung đột xảy ra khi hai khóa khác nhau biến đổi thành một chỉ số.
- Ví dụ:
  - Hàm băm h(k) = k % m
  - h(7) = 7 % 5 = 2
  - h(12) = 12 % 5 = 2



Giải quyết xung đột (Collision Resolution)

- Separate Chaining: tạo chuỗi riêng biệt.





Open Addressing

- Nếu vị trí k = h(x) được sử dụng, thì các vị trí sau đây được thử:
  - $k = h_i(x) = (h(x) + p(i)) \bmod M$, i = 1, 2,… (M = Table Size)
- Phương pháp thăm dò tuyến tính (đơn giản nhất):
  - p(i) = i
  - Vị trí cần thử là $(h(x) + i) \bmod M$, i = 1,2,…
- Phương pháp thăm dò bậc hai:
  - $p(i) = i^2$
  - Vị trí cần thử là $(h(x) + i^2) \bmod M$, i = 1,2,…

Open Addressing

- Ví dụ phương pháp thăm dò bậc hai:
  - Nếu vị trí k = h(x) được sử dụng, thì các vị trí sau đây được thử:
    - $k = h_i(x) = (h(x) + i^2) \bmod M$, i = 1, 2, ... (M = Table Size)
  - Ví dụ:
    - M = 10 và h(x) = x mod 10
    - $k = h_i(x) = (h(x) + i^2) \bmod M = ((x \bmod 10) + i^2) \bmod 10$
    - Thêm các khóa 89, 18, 49, 58, 69 theo thứ tự sẽ như sau:

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

Open Addressing

- Phương pháp băm kép:
  - Áp dụng hàm băm thứ hai khi xảy ra xung đột:
    - (h1(key) + i * h2(key)) % M (i=1,2,…), (M = Table Size)
  - Một hàm băm thứ hai phổ biến là:
    - h2(key) = PRIME – (key % PRIME)
    - PRIME là số nguyên tố nhỏ hơn M.
  - Ví dụ:
    - h1(key) = key % 7
    - h2(key) = 5 – (key % 5)
  - Một hàm băm thứ hai tốt là:
    - Nó không bao giờ có kết quả bằng 0.
    - Phải đảm bảo rằng tất cả các ô có thể được thăm dò.

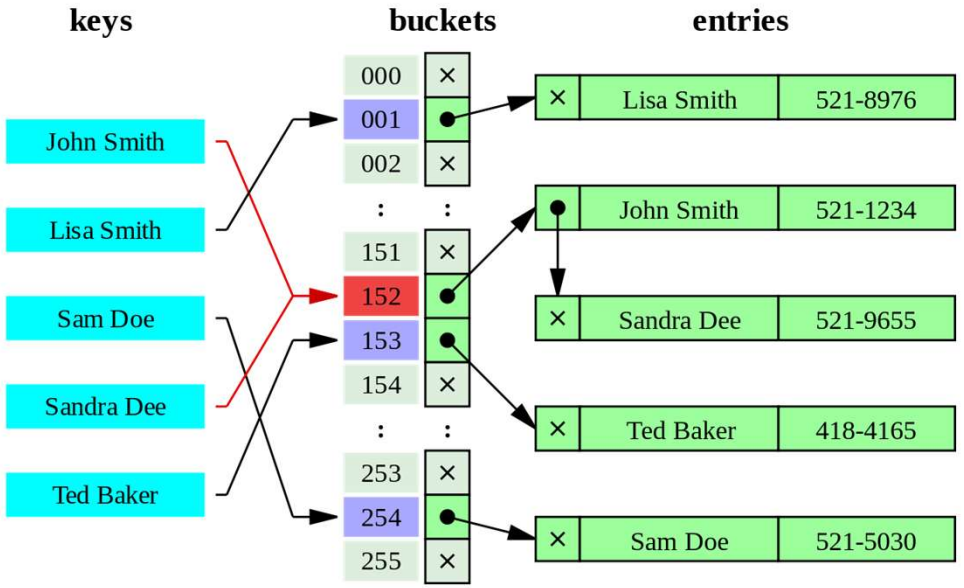
h1(21) = 21 % 7 = 0 (✗)
h2(21) = 5 – (21 % 5) = 4
(0 + 1 * 4) % 7 = 4 (✓)

h1(22) = 22 % 7 = 1 (✗)
h2(22) = 5 – (22 % 5) = 3
(1 + 1 * 3) % 7 = 4 (✗)
(1 + 2 * 3) % 7 = 0 (✗)
(1 + 3 * 3) % 7 = 3 (✓)

Key	Index	Key
14	0	14
15	1	15
16	2	16
21	3	22
22	4	21

Separate Chaining

- Mỗi vị trí của bảng được liên kết với một danh sách liên kết (hoặc một chuỗi các cấu trúc có chứa khóa hoặc tham chiếu đến khóa).
- Bảng tham chiếu (con trỏ) được gọi là bảng phân tán (scatter table).
- Bảng không bao giờ tràn vì danh sách liên kết có thể mở rộng.



HOA SEN UNIVERSITY

www.hoasen.edu.vn

Cài đặt Hash Table

```
#define TABLE_SIZE 11

typedef struct Node {
    int key;
    int value;
    struct Node* next; // Dành cho separate chaining
} Node;

typedef struct HashTable {
    Node* table[TABLE_SIZE];
} HashTable;
```

Cài đặt Hash Function

```
int hash(int key) {
    return key % TABLE_SIZE;
}
```

- Ví dụ:
  - key=15,TABLE_SIZE=11
  - hash(15) = 15 % 11 = 4

HOA SEN UNIVERSITY

www.hoasen.edu.vn

Thêm phần tử vào bảng băm

- Ý tưởng:
  - Xác định index của phần tử cần thêm bằng hàm băm.
  - Tạo node mới chứa phần tử cần thêm.
  - Gắn node mới vào đầu danh sách liên kết tại vị trí index.

Xóa phần tử khỏi bảng băm

- Ý tưởng:
  - Xác định index của phần tử cần xóa bằng hàm băm.
  - Duyệt danh sách liên kết tại vị trí index để tìm phần tử có khóa cần xóa.
  - Xóa phần tử khỏi danh sách liên kết bằng cách cập nhật con trỏ.
  - Trả về true/false.

Tìm kiếm trong bảng băm

- Ý tưởng:
  - Xác định index của phần tử cần tìm bằng hàm băm.
  - Duyệt danh sách liên kết tại vị trí index để tìm phần tử có khóa cần tìm.
  - Nếu tìm thấy thì trả về địa chỉ của node, ngược lại trả về NULL.

HOA SEN UNIVERSITY

www.hoasen.edu.vn

Độ phức tạp trong Hash Table

Trường hợp	Thêm	Xóa	Tìm kiếm
Tốt nhất (Best Case)	O(1)	O(1)	O(1)
Trung bình (Average)	O(1)	O(1)	O(1)
Xấu nhất (Worst Case)	O(n)	O(n)	O(n)

Giải thích các trường hợp

- Tốt nhất (Best Case):
  - Không có xung đột; mỗi khóa ánh xạ vào một chỉ số duy nhất.
  - Các thao tác chỉ cần thời gian để tính hàm băm và truy cập trực tiếp.
- Trung bình (Average Case):
  - Hàm băm hiệu quả, phân phối khóa đều, xung đột thấp.
  - Hệ số tải (α) duy trì ở mức thấp (≤ 0.75), giúp các thao tác gần O(1).
- Xấu nhất (Worst Case):
  - Tất cả khóa ánh xạ vào cùng một chỉ số, gây xung đột tối đa.
  - Với Separate Chaining: Duyệt toàn bộ danh sách liên kết tại một chỉ số (O(n)).
  - Với Open Addressing: Tìm kiếm hoặc kiểm tra tất cả các ô trong bảng (O(n)).

Cách cải thiện hiệu suất của Hash Table

1. Thiết kế hàm băm hiệu quả

- **Phân phối đều**: Hàm băm cần đảm bảo các giá trị được phân phối đều trong bảng, giảm thiểu xung đột.
- **Xử lý nhanh**: Hàm băm nên thực thi nhanh, tránh làm chậm quá trình xử lý dữ liệu.
- **Phù hợp với loại dữ liệu**: Lựa chọn hàm băm phù hợp với kiểu dữ liệu đầu vào (số, chuỗi...).
- Ví dụ: Đối với khóa là chuỗi, có thể dùng:

```
unsigned int hash(const char* str) {
    unsigned int hash_value = 0;
    while (*str) {
        // Sử dụng số nguyên tố để giảm xung đột
        hash_value = (hash_value * 31) + (*str);
        str++;
    }
    return hash_value;
}
```

Cách cải thiện hiệu suất của Hash Table

2. Chọn kích thước bảng băm phù hợp

- Kích thước bảng là số nguyên tố: Giúp giảm xung đột khi sử dụng hàm băm như key % table_size.
- Duy trì hệ số tải thấp (Load Factor):
  - Load Factor: $\frac{n}{m}$, trong đó:
    - n: Số phần tử đã lưu trữ.
    - m: Kích thước bảng (table size).
  - Duy trì $\alpha \leq 0.75$ để giảm xung đột và giữ hiệu suất tốt.

Cách cải thiện hiệu suất của Hash Table

3. Sử dụng kỹ thuật xử lý xung đột phù hợp

- **Separate Chaining:**
  - Hiệu quả khi Load Factor tăng cao.
  - Dễ cài đặt và mở rộng.
- **Open Addressing**:
  - Tốt với Load Factor thấp.
  - Cải thiện bằng các chiến lược:
    - **Linear Probing**: Đơn giản nhưng dễ gây hiện tượng cụm.
    - **Quadratic Probing**: Giảm hiện tượng cụm.
    - **Double Hashing**: Tối ưu khi có hàm băm phụ tốt.

Cách cải thiện hiệu suất của Hash Table

4. Điều chỉnh kích thước bảng động

- Khi Load Factor vượt ngưỡng (ví dụ: 0.75), cần **mở rộng bảng băm**:
  - Tăng kích thước bảng lên một số nguyên tố lớn hơn.
  - Rehash (tính lại chỉ số băm) tất cả phần tử trong bảng cũ.

5. Tối ưu bộ nhớ

- Sử dụng các cấu trúc nhẹ hơn nếu khóa và giá trị có kích thước lớn.
- Dùng cơ chế xóa phần tử không còn sử dụng để giải phóng bộ nhớ.

Cách cải thiện hiệu suất của Hash Table

6. Áp dụng kỹ thuật nâng cao

- **Cuckoo Hashing**: Dùng hai hàm băm để giảm xung đột.
- **Perfect Hashing**: Được sử dụng nếu biết trước tập hợp các khóa, tạo ra hàm băm không có xung đột.
- **Dynamic Hashing**: Thay đổi kích thước bảng băm linh hoạt theo số lượng phần tử.

7. Tối ưu truy cập

- **Sử dụng bộ nhớ đệm (Cache)**: Sắp xếp các chỉ số băm để phù hợp với cách CPU xử lý, giúp tăng tốc độ truy cập.
- **Giảm kích thước phần tử**: Chỉ lưu trữ thông tin cần thiết để giảm chi phí tìm kiếm và sử dụng bộ nhớ.

HOA SEN UNIVERSITY

www.hoasen.edu.vn

Tình huống không nên sử dụng Hash Table

1. Khi cần dữ liệu có thứ tự

2. Khi dữ liệu quá nhỏ

3. Khi bộ nhớ hạn chế

4. Khi các khóa không đồng nhất hoặc không thể băm hiệu quả

5. Khi cần truy cập tuần tự hoặc duyệt tất cả phần tử thường xuyên

6. Khi có quá nhiều xung đột

7. Khi cần độ an toàn cao trong môi trường đa luồng

8. Khi dữ liệu cần cập nhật thường xuyên

9. Khi cần so sánh khóa phức tạp

- Hash Table không duy trì thứ tự của các phần tử (trừ khi sử dụng các cấu trúc đặc biệt như LinkedHashMap hoặc OrderedDict trong Python.
- Tại sao không phù hợp:
  - Hash Table chỉ lưu trữ dữ liệu dựa trên hàm băm, không có ý nghĩa về thứ tự chèn hoặc thứ tự tự nhiên.
  - Nếu cần duyệt dữ liệu theo thứ tự tăng dần hoặc giảm dần, các cấu trúc như Tree Map, Red-Black Tree, hoặc Linked List sẽ phù hợp hơn.
- Ví dụ không phù hợp:
  - Lưu trữ danh sách sinh viên theo thứ tự điểm số hoặc theo thứ tự bảng chữ cái.

HOA SEN UNIVERSITY

www.hoasen.edu.vn

Tình huống không nên sử dụng Hash Table

1. Khi cần dữ liệu có thứ tự

2. Khi dữ liệu quá nhỏ

3. Khi bộ nhớ hạn chế

4. Khi các khóa không đồng nhất hoặc không thể băm hiệu quả

5. Khi cần truy cập tuần tự hoặc duyệt tất cả phần tử thường xuyên

6. Khi có quá nhiều xung đột

7. Khi cần độ an toàn cao trong môi trường đa luồng

8. Khi dữ liệu cần cập nhật thường xuyên

9. Khi cần so sánh khóa phức tạp

- Nếu chỉ cần lưu trữ một vài phần tử, Hash Table có thể không hiệu quả vì:
  - Bộ nhớ bị tiêu tốn không cần thiết (overhead).
  - Hiệu suất không cao hơn nhiều so với các cấu trúc đơn giản như array hoặc linked list.

Tình huống không nên sử dụng Hash Table

1. Khi cần dữ liệu có thứ tự

2. Khi dữ liệu quá nhỏ

3. Khi bộ nhớ hạn chế

4. Khi các khóa không đồng nhất hoặc không thể băm hiệu quả

5. Khi cần truy cập tuần tự hoặc duyệt tất cả phần tử thường xuyên

6. Khi có quá nhiều xung đột

7. Khi cần độ an toàn cao trong môi trường đa luồng

8. Khi dữ liệu cần cập nhật thường xuyên

9. Khi cần so sánh khóa phức tạp

- Hash Table tiêu tốn nhiều bộ nhớ hơn các cấu trúc dữ liệu khác, vì:
  - Nó cần không gian để lưu cả bảng băm và xử lý xung đột (danh sách liên kết, vùng tìm kiếm mở rộng).
  - Nếu hệ số tải cao ($\alpha > 1$), hiệu suất giảm, buộc phải mở rộng bảng, gây tiêu tốn thêm bộ nhớ.
- Ví dụ không phù hợp:
  - Ứng dụng nhúng hoặc thiết bị IoT có bộ nhớ hạn chế.

Tình huống không nên sử dụng Hash Table

1. Khi cần dữ liệu có thứ tự
2. Khi dữ liệu quá nhỏ
3. Khi bộ nhớ hạn chế
4. **Khi các khóa không đồng nhất hoặc không thể băm hiệu quả**
5. Khi cần truy cập tuần tự hoặc duyệt tất cả phần tử thường xuyên
6. Khi có quá nhiều xung đột
7. Khi cần độ an toàn cao trong môi trường đa luồng
8. Khi dữ liệu cần cập nhật thường xuyên
9. Khi cần so sánh khóa phức tạp

- Hash Table yêu cầu các khóa (keys) phải:
  - Có khả năng băm (hashable).
  - Phân phối đều qua bảng băm.
  - Nếu khóa không đồng nhất (ví dụ: các đối tượng phức tạp) hoặc hàm băm không phù hợp, xung đột sẽ xảy ra nhiều hơn, khiến Hash Table mất hiệu suất.
- Ví dụ không phù hợp:
  - Sử dụng các đối tượng lớn hoặc không thể băm (như danh sách, từ điển) làm khóa.
  - Dữ liệu không đồng nhất gây ra xung đột trong bảng băm.

Tình huống không nên sử dụng Hash Table

1. Khi cần dữ liệu có thứ tự

2. Khi dữ liệu quá nhỏ

3. Khi bộ nhớ hạn chế

4. Khi các khóa không đồng nhất hoặc không thể băm hiệu quả

5. Khi cần truy cập tuần tự hoặc duyệt tất cả phần tử thường xuyên

6. Khi có quá nhiều xung đột

7. Khi cần độ an toàn cao trong môi trường đa luồng

8. Khi dữ liệu cần cập nhật thường xuyên

9. Khi cần so sánh khóa phức tạp

- Hash Table không hỗ trợ duyệt tuần tự một cách hiệu quả. Khi cần duyệt qua toàn bộ phần tử, các cấu trúc như array, binary tree, hoặc linked list sẽ phù hợp hơn.
- Ví dụ không phù hợp:
  - Tìm kiếm tuần tự trong danh sách sản phẩm để áp dụng một chính sách giảm giá.

Tình huống không nên sử dụng Hash Table

1. Khi cần dữ liệu có thứ tự
2. Khi dữ liệu quá nhỏ
3. Khi bộ nhớ hạn chế
4. Khi các khóa không đồng nhất hoặc không thể băm hiệu quả
5. Khi cần truy cập tuần tự hoặc duyệt tất cả phần tử thường xuyên
6. **Khi có quá nhiều xung đột**
7. Khi cần độ an toàn cao trong môi trường đa luồng
8. Khi dữ liệu cần cập nhật thường xuyên
9. Khi cần so sánh khóa phức tạp

- Nếu dữ liệu có xu hướng tạo ra nhiều xung đột, hiệu suất của Hash Table giảm mạnh. Nguyên nhân có thể là:
  - Hàm băm không đủ tốt.
  - Tập dữ liệu bị gom cụm (clustered data).
  - Trong những trường hợp này, cấu trúc như cây balanced binary tree sẽ hiệu quả hơn.
- Ví dụ không phù hợp:
  - Ánh xạ các giá trị có cùng mẫu khóa, như tất cả các giá trị có dạng "user123".

HOA SEN UNIVERSITY

www.hoasen.edu.vn

Tình huống không nên sử dụng Hash Table

1. Khi cần dữ liệu có thứ tự

2. Khi dữ liệu quá nhỏ

3. Khi bộ nhớ hạn chế

4. Khi các khóa không đồng nhất hoặc không thể băm hiệu quả

5. Khi cần truy cập tuần tự hoặc duyệt tất cả phần tử thường xuyên

6. Khi có quá nhiều xung đột

7. Khi cần độ an toàn cao trong môi trường đa luồng

8. Khi dữ liệu cần cập nhật thường xuyên

9. Khi cần so sánh khóa phức tạp

- Hash Table không phải lúc nào cũng an toàn trong môi trường đa luồng (thread-safe).
- Nếu nhiều luồng cùng truy cập và thay đổi Hash Table mà không có cơ chế đồng bộ, lỗi dữ liệu hoặc hành vi không mong muốn có thể xảy ra.
- Giải pháp thay thế:
  - Sử dụng các phiên bản an toàn như ConcurrentHashMap (Java) hoặc khóa đồng bộ (synchronization locks).
- Ví dụ không phù hợp:
  - Hệ thống giao dịch ngân hàng yêu cầu tính nhất quán dữ liệu trong môi trường đa luồng.

Tình huống không nên sử dụng Hash Table

1. Khi cần dữ liệu có thứ tự
2. Khi dữ liệu quá nhỏ
3. Khi bộ nhớ hạn chế
4. Khi các khóa không đồng nhất hoặc không thể băm hiệu quả
5. Khi cần truy cập tuần tự hoặc duyệt tất cả phần tử thường xuyên
6. Khi có quá nhiều xung đột
7. Khi cần độ an toàn cao trong môi trường đa luồng
8. **Khi dữ liệu cần cập nhật thường xuyên**
9. Khi cần so sánh khóa phức tạp

- Hash Table hiệu quả khi các phần tử không thay đổi thường xuyên.
- Nếu dữ liệu thay đổi liên tục (thêm, xóa, sửa), bảng băm có thể cần tái tổ chức (rehashing), gây ra hiệu suất kém hơn.
- Ví dụ không phù hợp:
  - Hệ thống quản lý thời gian thực như theo dõi vị trí liên tục của xe trong GPS (Global Positioning System).

HOA SEN UNIVERSITY

www.hoasen.edu.vn

Tình huống không nên sử dụng Hash Table

1. Khi cần dữ liệu có thứ tự
2. Khi dữ liệu quá nhỏ
3. Khi bộ nhớ hạn chế
4. Khi các khóa không đồng nhất hoặc không thể băm hiệu quả
5. Khi cần truy cập tuần tự hoặc duyệt tất cả phần tử thường xuyên
6. Khi có quá nhiều xung đột
7. Khi cần độ an toàn cao trong môi trường đa luồng
8. Khi dữ liệu cần cập nhật thường xuyên
9. **Khi cần so sánh khóa phức tạp**

- Nếu việc so sánh khóa cần nhiều phép toán hơn là chỉ kiểm tra bằng hàm băm, Hash Table không phải lựa chọn tốt. Cấu trúc như binary search tree sẽ phù hợp hơn.
- Ví dụ không phù hợp:
  - So sánh các khóa dựa trên nhiều điều kiện phức tạp hoặc theo trọng số.

Ứng dụng thực tế

1. Hệ thống lưu trữ và truy xuất dữ liệu

- Ứng dụng:
  - Lập chỉ mục cho cơ sở dữ liệu (Database Indexing): Hash Table được dùng làm chỉ mục cho dữ liệu trong CSDL, giúp tăng tốc độ truy vấn.
  - Ví dụ: tìm kiếm bản ghi (record) dựa trên khóa chính.
  - Lưu trữ bộ nhớ cache (Caching): Các hệ thống như Redis hoặc Memcached sử dụng Hash Table để ánh xạ các khóa (keys) đến giá trị (values), tối ưu hóa việc truy xuất dữ liệu.
- Ưu điểm:
  - Tốc độ truy cập O(1) giúp xử lý khối lượng dữ liệu lớn nhanh chóng.
  - Hỗ trợ tra cứu nhanh trong các hệ thống thời gian thực.

Ứng dụng thực tế

2. Hệ thống kiểm tra và tra cứu

- Ứng dụng:
  - Bảng băm từ điển (Dictionary): Hash Table được dùng trong từ điển hoặc bộ kiểm tra chính tả để ánh xạ từ đến nghĩa hoặc cách sử dụng.
  - Bảng ký tự Unicode: Hash Table giúp tra cứu nhanh các mã ký tự (Unicode) và ánh xạ chúng tới biểu diễn thực tế.
- Ví dụ thực tế:
  - Từ điển Python (dict) và Java HashMap đều sử dụng Hash Table để ánh xạ khóa và giá trị.

Ứng dụng thực tế

3. Phát hiện và loại bỏ trùng lặp

- Ứng dụng:
  - Kiểm tra trùng lặp trong tập dữ liệu lớn: Hash Table lưu các giá trị đã duyệt qua, giúp phát hiện trùng lặp một cách hiệu quả.
  - Loại bỏ trùng lặp trong văn bản hoặc cơ sở dữ liệu: Áp dụng để xử lý dữ liệu trong các công cụ tìm kiếm hoặc hệ thống thông tin.
- Ưu điểm:
  - Tiết kiệm bộ nhớ và thời gian so với các phương pháp lặp thông thường.

Ứng dụng thực tế

4. Mạng máy tính

- Ứng dụng:
  - Bảng ARP (Address Resolution Protocol): Sử dụng Hash Table để ánh xạ địa chỉ IP với địa chỉ MAC, giúp định tuyến mạng hiệu quả.
  - Cấu trúc DNS (Domain Name System): Hash Table được dùng để ánh xạ tên miền (domain) tới địa chỉ IP tương ứng, giảm thời gian tra cứu.
- Ví dụ thực tế:
  - Hệ thống định tuyến và tra cứu DNS trong Internet.

Ứng dụng thực tế

5. Lập trình ngôn ngữ tự nhiên

- Ứng dụng:
  - Tạo chỉ mục trong công cụ tìm kiếm: Hash Table lưu chỉ mục của từ và các tài liệu chứa chúng, giúp tìm kiếm nhanh hơn.
  - Xây dựng bảng tần suất từ: Hash Table dùng để đếm tần suất từ trong một văn bản hoặc tập hợp tài liệu lớn.

Ứng dụng thực tế

6. Hệ thống mật mã và bảo mật

- Ứng dụng:
  - Bảng băm mật khẩu: Các hệ thống bảo mật sử dụng Hash Table để lưu trữ băm của mật khẩu thay vì lưu trữ mật khẩu gốc.
  - Phát hiện tấn công từ chối dịch vụ (DoS): Hash Table dùng để theo dõi các yêu cầu mạng và phát hiện hoạt động bất thường.

Ứng dụng thực tế

7. Lập trình trò chơi và đồ họa

- Ứng dụng:
  - Hệ thống quản lý tài nguyên: Hash Table ánh xạ tài nguyên (hình ảnh, âm thanh) tới các file tương ứng, giúp tối ưu hóa quản lý bộ nhớ.
  - Kiểm tra va chạm (Collision Detection): Dùng Hash Table để lưu trạng thái của các đối tượng trong trò chơi, hỗ trợ phát hiện va chạm nhanh chóng.

Ứng dụng thực tế

8. Các bài toán thuật toán

- Ứng dụng:
  - Tìm cặp số trong mảng có tổng bằng **k**: Sử dụng Hash Table để lưu các phần tử đã duyệt, giảm độ phức tạp từ $O(n^2)$ xuống $O(n)$.
  - Xử lý bài toán đếm tần suất: Hash Table giúp ánh xạ các phần tử trong mảng tới số lần xuất hiện của chúng.

Ứng dụng thực tế

9. Hệ thống phân tán

- Ứng dụng:
  - Bảng băm phân tán (Distributed Hash Table - DHT): Sử dụng trong các hệ thống như BitTorrent hoặc các mạng ngang hàng (P2P), giúp ánh xạ tài nguyên tới vị trí lưu trữ trong mạng phân tán.

Ứng dụng thực tế

10. Hệ thống quản lý session và state

- Ứng dụng:
  - Quản lý session trong web server:
    - Hash Table ánh xạ **Session ID** tới trạng thái hoặc dữ liệu của người dùng.
  - Lưu trạng thái trò chơi:
    - Hash Table ánh xạ người chơi tới trạng thái trò chơi của họ.

HOA SEN UNIVERSITY

www.hoasen.edu.vn

Tổng kết

- Hash Table là một cấu trúc dữ liệu mạnh mẽ với thời gian truy cập nhanh.
- Hiệu quả của Hash Table phụ thuộc vào thiết kế hàm băm và phương pháp xử lý xung đột.
- Ứng dụng rộng rãi trong nhiều lĩnh vực thực tế.


Q
A