Hw6_22000123_NguyenDucSi_

Nguyễn Đức Sĩ

November 2024

1 Băm

1.1 Hàm Băm

Hàm băm là một hàm ánh xạ dữ liệu đầu vào (khóa) thành một chỉ số trong bảng băm, qua đó phân tán đều các khóa vào các ô trong bảng băm để giảm thiểu đụng độ.
Các thành phần chính :

- Hàm băm.
- Bảng băm.
- Cách thức xử lý va chạm.

Một hàm băm tốt là hàm băm có chia đều các giá trị vào bảng, tính toán với tốc độ nhanh, hạn chế việc xảy ra đụng độ (tức hai hoặc nhiều giá trị khác nhau cùng được ánh xạ vào một ô trong bảng). Và với một giá trị đầu vào, hàm phải luôn cho cùng một kết quả.

1.2 Một số phương pháp thường gặp

1.2.1 Hàm băm chia dư (Division Hashing)

Hàm băm chia dư sử dụng phép chia lấy phần dư để ánh xạ khóa thành một chỉ số trong bảng băm:

$$h(k) = k \tag{1}$$

Với k là khóa và m là kích thước bảng băm.

1.2.2 Hàm băm Folding (Folding Hashing)

Hàm băm folding chia khóa thành nhiều phần nhỏ có kích thước cố định, sau đó cộng tất cả các phần này lại và lấy phần dư của tổng với kích thước bảng:

$$h(k) = \text{sum of segments of } k \mod m$$

Ví dụ: nếu khóa là số lớn như 987654321, ta chia thành các đoạn nhỏ như 987, 654, và 321, sau đó cộng lại và lấy phần dư khi chia cho kích thước bảng.

1.2.3 Hàm băm Mid-Square (Mid-Square Hashing)

Hàm băm này nhân khóa với chính nó, sau đó lấy một số bit nhất định từ giữa kết quả để làm chỉ số cho bảng băm:

$$h(k) = \text{middle bits of } k^2$$

1.2.4 Hàm băm Radix Transformation

Phương pháp này chuyển đổi các giá trị không phải số nguyên thành số nguyên trước khi áp dụng một hàm băm khác. Ví dụ, ta có thể chuyển đổi các ký tự thành giá trị ASCII rồi áp dụng phép toán băm chia dư.

1.3 Đụng độ

Đụng độ xảy ra khi hai khóa khác nhau được ánh xạ vào cùng một ô trong bảng băm.
Các cách sử lý đụng độ:

1.4 Separate Chaining (Chuỗi liên kết)

Phương pháp chuỗi liên kết sử dụng một danh sách liên kết (linked list) tại mỗi ô trong bảng băm. Khi xảy ra đụng độ, khóa đang được thêm vào sẽ được liên kết với khóa ở ô đó thông qua một danh sách liên.
Cách cài đặt:

```java
// Java code example
import java.util.LinkedList;

class HashTableWithChaining {
    private LinkedList<Entry>[] table;
    private final int SIZE = 10;

    public HashTableWithChaining() {
        table = new LinkedList[SIZE];
        for (int i = 0; i < SIZE; i++) {
            table[i] = new LinkedList<>();
        }
    }

    private int hash(String key) {
        return key.length() % SIZE;
    }

    public void put(String key, int value) {
        int index = hash(key);
        for (Entry entry : table[index]) {
            if (entry.key.equals(key)) {
                entry.value = value;
```

```
                return;
            }
        }
        table[index].add(new Entry(key, value));
    }

    public Integer get(String key) {
        int index = hash(key);
        for (Entry entry : table[index]) {
            if (entry.key.equals(key)) {
                return entry.value;
            }
        }
        return null;
    }

    class Entry {
        String key;
        int value;

        public Entry(String key, int value) {
            this.key = key;
            this.value = value;
        }
    }
}
```

1.4.1 Linear Probing (Tìm kiếm tuyến tính)

Linear Probing tìm vị trí trống tiếp theo trong bảng băm khi xảy ra đụng độ bằng cách di chuyển tuần tự qua các ô cho đến khi tìm được ô trống. Phương pháp này không cần sử dụng danh sách liên kết.

```
// Java code example
class HashTableWithLinearProbing {
    private Entry[] table;
    private final int SIZE = 10;

    public HashTableWithLinearProbing() {
        table = new Entry[SIZE];
    }

    private int hash(String key) {
        return key.length() % SIZE;
    }

    public void put(String key, int value) {
        int index = hash(key);
```

```
        while (table[index] != null && !table[index].key.
            equals(key)) {
            index = (index + 1) % SIZE;
        }
        table[index] = new Entry(key, value);
    }

    public Integer get(String key) {
        int index = hash(key);
        while (table[index] != null && !table[index].key.
            equals(key)) {
            index = (index + 1) % SIZE;
        }
        return table[index] != null ? table[index].value :
            null;
    }

    class Entry {
        String key;
        int value;

        public Entry(String key, int value) {
            this.key = key;
            this.value = value;
        }
    }
}
```

1.4.2 Quadratic Probing (Tìm kiếm bậc hai)

Phương pháp này tìm vị trí trống bằng cách tăng khoảng cách giữa các lần tìm kiếm theo lũy thừa bậc hai (ví dụ: $1^2, 2^2, 3^2, ...$).

```
class HashTableWithQuadraticProbing {
    private Entry[] table;
    private final int SIZE = 10;

    public HashTableWithQuadraticProbing() {
        table = new Entry[SIZE];
    }

    private int hash(String key) {
        return key.length() % SIZE;
    }

    public void put(String key, int value) {
        int index = hash(key);
        int i = 0;
```

```
        while (table[(index + i * i) % SIZE] != null && !
            table[(index + i * i) % SIZE].key.equals(key)) {
            i++;
        }
        table[(index + i * i) % SIZE] = new Entry(key, value
            );
    }

    public Integer get(String key) {
        int index = hash(key);
        int i = 0;
        while (table[(index + i * i) % SIZE] != null) {
            if (table[(index + i * i) % SIZE].key.equals(key
                )) {
                return table[(index + i * i) % SIZE].value;
            }
            i++;
        }
        return null;
    }

    class Entry {
        String key;
        int value;

        public Entry(String key, int value) {
            this.key = key;
            this.value = value;
        }
    }
}
```

1.4.3 Double Hashing (Băm kép)

Double Hashing sử dụng hai hàm băm khác nhau để tính toán chỉ số cho khóa. Khi xảy ra đụng độ, hàm băm thứ hai sẽ xác định khoảng cách nhảy giữa các lần kiểm tra ô trống.
Hàm băm thứ hai thường được chọn để tránh lặp lại chỉ số băm, ví dụ:

$$h_2(k) = 7 - (k\%7) \tag{2}$$

```
class HashTableWithDoubleHashing {
    private Entry[] table;
    private final int SIZE = 10;

    public HashTableWithDoubleHashing() {
        table = new Entry[SIZE];
    }
```

```
8
9       private int hash1(String key) {
10          return key.length() % SIZE;
11      }
12
13      private int hash2(String key) {
14          return 1 + (key.hashCode() % (SIZE - 1));
15      }
16
17      public void put(String key, int value) {
18          int index = hash1(key);
19          int stepSize = hash2(key);
20          int i = 0;
21          while (table[(index + i * stepSize) % SIZE] != null
                && !table[(index + i * stepSize) % SIZE].key.
                equals(key)) {
22              i++;
23          }
24          table[(index + i * stepSize) % SIZE] = new Entry(key
                , value);
25      }
26
27      public Integer get(String key) {
28          int index = hash1(key);
29          int stepSize = hash2(key);
30          int i = 0;
31          while (table[(index + i * stepSize) % SIZE] != null)
                {
32              if (table[(index + i * stepSize) % SIZE].key.
                    equals(key)) {
33                  return table[(index + i * stepSize) % SIZE].
                        value;
34              }
35              i++;
36          }
37          return null;
38      }
39
40      class Entry {
41          String key;
42          int value;
43
44          public Entry(String key, int value) {
45              this.key = key;
46              this.value = value;
47          }
48      }
49  }
```