**Bài tập lớn 02 - Group 3**

## **Implementation Details**

### **Description of the hash table implementation**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **LinearHashTable** |  |  |  | **ChainHashTable** |
| useFibonacci: bool |  |  |  | useFibonacci: bool |
| size, sizePower, count: unsigned int |  |  |  | size, sizePower, count: unsigned int |
| table: vector<optional<**Node**>> | **Node** |  | **Node** | table: vector<**Node**\*> |
| collisions: vector<optional<int>> | key: unsigned int |  | key: unsigned int | collisions: vector<optional<int>> |
| resetCollisions(): void | value: string |  | value: string | resetCollisions(): void |
| hashFunction(key: unsigned int): unsigned int |  |  | next: Node\* | hashFunction(key: unsigned int): unsigned int |
| insert(key: unsigned int, value: string): void |  |  |  | insert(key: unsigned int, value: string): void |
| remove(key: unsigned int): void |  |  |  | remove(key: unsigned int): void |
| search(key: unsigned int): string |  |  |  | search(key: unsigned int): string |
| resize(newSize: unsigned int): void |  |  |  | resize(newSize: unsigned int): void |
| input(path: string, n: unsigned int): void |  |  |  | input(path: string, n: unsigned int): void |
| outputStats(): void |  |  |  | outputStats(): void |
| output(n: unsigned int): void |  |  |  | output(n: unsigned int): void |
| LinearHashTable(size: unsigned int, useFibonacci: bool) |  |  |  | ChainHashTable(size: unsigned int, useFibonacci: bool) |
|  |  |  |  | ~ChainHashTable(); |

*Bảng 1: Các thuộc tính và phương thức của struct LinearHashTable và ChainHashTable*

* **Node**: là 1 struct riêng được định nghĩa bên trong 2 struct LinearHashTable và ChainHashTable, gồm có 2 thuộc tính là key - dùng để tính ra index thông qua hàm hashFunction(), value – lưu dữ liệu còn lại của phần tử, và con trỏ Node tiếp theo (chỉ dành cho struct ChainHashTable).
* **useFibonacci**: true – sử dụng phương pháp tỷ lệ vàng để tính index hoặc false – sử dụng phương pháp modulo thông thường để tính index.
* **size**: bằng 2sizePower, chứa độ lớn của mảng băm.
* **sizePower**: bằng log2(size), dùng khi tính index theo phương pháp tỉ lệ vàng.
* **count**: số lượng phần tử có trong mảng.
* **table**: mảng các Node (option).
* **collisions**: dùng để lưu số va chạm gặp được theo từng vị trí.
* ***resetCollisions*()**: dùng để reset số va chạm thành 0 cho tất cả các vị trí.
* ***hashFunction*()**: dùng để chọn 1 phương pháp đã được khai báo bên ngoài struct để tính ra index dựa trên useFibonacci.
* ***moduloHash*()**: sử dụng phương pháp chia modulo thông thường.
* ***fibonacciHash*()**: sử dụng phương pháp tỉ lệ vàng để tính ra index.
* ***insert*()**: dùng để thêm vào 1 phần tử vào mảng băm dựa trên index được tính từ key, nếu phần tử đã tồn tại thì cập nhật value mới cho phần tử đó.
* ***remove*()**: dùng để xóa bớt 1 phần tử của mảng dựa trên index được tính từ key.
* ***search*()**: dùng để tra cứu value của 1 phần tử bằng key, trả về “” nếu không tìm thấy phần tử đó.
* ***resize*()**: cập nhật lại size của mảng đồng thời rehash hết tất cả phần tử trong mảng.
* ***input*()**: đọc n dòng từ file tại path và thêm vào bảng băm.
* ***outputStats*()**: chỉ in ra size, count, loadFactor (count / size), số va chạm lớn nhất gặp trong 1 ô, tổng số va chạm, số va chạm trung bình của toàn mảng băm.
* ***output*()**: in ra tất cả những gì có trong outputStats, in ra tất cả các phần tử trong mảng băm thành từng dòng, tất cả số va chạm gặp được theo từng ô thành từng dòng.
  + **LinearHashTable**: <index>: <key>|<value> hoặc <index>: null.
  + **ChainHashTable**: <index>: <key1>|<value1> -> <key2>|<value2> -> …. hoặc <index>: null.
* ***LinearHashTable*(), *ChainHashTable*()**: Các constructor, dùng để khởi tạo các giá trị ban đầu cho các thuộc tính.
* **~*ChainHashTable*()**: destructor, dùng để giải phóng bộ nhớ cho tất cả các con trỏ trong từng ô của table trong struct ChainHashTable.

### What is the golden ratio?. The golden ratio, also known as the… | by Fahim Bin Omar | Bootcamp | Medium**Explanation of the Fibonacci hashing function**

*Hình 1: Các thuộc tính và phương thức của struct LinearHashTable và ChainHashTable*

* **Tỉ lệ vàng ϕ**: được tính bằng với Fn+1 = a, Fn = b đều thuộc dãy Fibonacci (Hình 1).

**≈ 1.6180339887...**

* + Người ta nhận thấy khi nhân 1 dãy số với **ϕ**, ta được 1 dãy số tương ứng với các giá trị được phân bố bao bọc dọc theo trục số một cách gần như đều đặn, gần như không có giá trị nào bằng nhau. Đặc điểm này khi được áp dụng vào mảng băm để tính ra các index sẽ tránh được va chạm– collisions và tập trung – clustering.
* **Cách tính ra index**:
  + Chỉ lấy phần thập phân của **ϕ** là **ϕ’ ≈ 0.6180339887...** rồi chuyển sang dạng số nguyên bằng cách nhân với 232 ta được hằng số để dùng cho Fibonacci hashing là:

**0.6180339887 × 232 = 2654435769**

* + Ta tính ra index bằng công thức:

**Index = (key × 2654435769) ≫ (wordSize − n)**

* + Trong đó: >> là phép dịch bit sang phải; n bằng log2 của độ lớn của mảng (log2 size) và wordSize là độ lớn theo bits của kiểu unsigned int (thường là 32 hoặc 64).

### **Explanation of collision resolution strategies**

Có 2 giải pháp để giải quyết va chạm trong mảng băm và cả 2 giải pháp này đều cần được rehash nếu có quá nhiều phần tử trong mảng, điều này sẽ được kiểm trả bằng giá trị:

**loadFactor = count / size**

Trong đó: count là số phần tử có trong mảng và size là độ lớn của mảng, loadFactor càng cao thì khả năng xảy ra va chạm càng lớn, vì thế cần rehash để giảm thiểu va chạm.

* Linear Probing (Open Addressing)
* Khi xảy ra va chạm (collisions), ta tìm ô kế tiếp trong bảng băm bằng cách tăng chỉ số lên 1 đơn vị cho đến khi tìm được ô trống:

**index = (index + 1) % tableSize**

* Hàm resize sẽ được gọi nếu **loadFactor > 0.7** để rehash hết tất cả phần tử trong mảng băm với độ lớn của mảng được nhân đôi.
* Separate Chaining (Closed Addressing)
* Mỗi ô trong bảng băm lưu một cấu trúc dữ liệu (thường là linked list) chứa các phần tử có cùng chỉ số băm.
* Hàm *resize()* sẽ được gọi nếu **loadFactor > 2.0** để rehash hết tất cả phần tử trong mảng băm với độ lớn của mảng được nhân đôi.

## **Experimental Results**

### **Benchmark methodology**

* Input sẽ là các cặp key, value tương ứng với ID người dùng và mật khẩu. Cả 2 giá trị này sẽ được đọc theo từng dòng trong các file .txt trong thư mục source/Input/
* Chương tình sẽ tiến hành do thời gian chạy (ms) và bộ nhớ sử dụng (Mb) và trả về theo pair<thời gian, bộ nhớ> thông qua hàm *measureExecutionTime*() mỗi khi tiến hành một loạt các *insert(), remove(), search()*.
* Các va chạm sẽ được cập nhật mỗi khi sử dụng các hàm *insert(), remove(), search()*. Chương trình cũng sẽ tính số va chạm nhiều nhất trong 1 ô, tổng số va chạm và số va chạm trung bình.
* Các key sẽ được lưu thành 3 file .txt riêng biệt dựa trên pattern (Random.txt: ngẫu nhiên, Cluster.txt: tập trung thành cụm, Sequential: dãy giá trị kề nhau) và người dùng cần chọn path của 1 trong 3 file để gán cho **INPUT\_FILE**.
* Mỗi file input đều chứa 500.000 keys, vì thế **LINE\_TO\_READ** sẽ được dùng để giới hạn số lượng phần tử mà chương trình sẽ đọc.
* **INITIAL\_SIZE** là giá trị cho size ban đầu và số lượng hàm *resize()* được gọi sẽ phụ thuộc vào hằng số này.
* **OPERATION\_SIZE** sẽ bằng 10% của LINE\_TO\_READ, chương trình sẽ khởi tạo số lượng key mới tương ứng để dùng cho các hàm *insert(), remove(), search()*.
* **USE\_FIBONACCI** chọn true nếu muốn dùng phương pháp tỉ lệ vàng để tính index, false nếu muốn dùng phương pháp modulo thông thường.
* Tất cả các thí nghiệm sẽ được tiến hành trên cùng một **SEED** được cho sẵn.
* Các file .exe đã được biên dịch sẵn và chứa trong thư mục build/ trong các thư mục con tương ứng với các key pattern của input, tên của các file .exe được đặt theo phương pháp hash và số lượng phần tử đã đọc từ input.

### **Performance comparison charts**

* **Open Addressing (Linear Probing)**
* Random Pattern
  + **INITIAL\_SIZE = 210 = 1024**
  + **INPUT\_FILE = “Random.txt”**
  + **LinearHashTable(*INITIAL\_SIZE, USE\_FIBONACCI)***
* Với thao tác *input(), insert(): số lần resize() được tính ra tương ứng.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Số phần tử random** | **30 000 (6 lần)** | **50 000 (7 lần)** | **100 000 (8 lần)** | **500 000 (10 lần)** |
| **Modulo (ms)** | 25.93 | 53.25 | 101.16 | 481.84 |
| **Fibonacci (ms)** | 25.22 | 46.19 | 110.87 | 458.8 |
| **Modulo (Mb)** | 5.58 | 10.57 | 21.53 | 84.59 |
| **Fibonacci (Mb)** | 5.57 | 10.56 | 21.59 | 84.35 |
| **Modulo (va chạm)** | 64171 | 122462 | 251896 | 1093674 |
| **Fibonacci (va chạm)** | 63793 | 119936 | 242717 | 1085369 |

*Bảng 2.1: Thời gian chạy (ms), bộ nhớ sử dụng (Mb), tổng số va chạm (va chạm) của các thao tác insert*

* Với thao tác *remove(), search()*: size của table sẽ bằng 10 lần số lượng phần tử tương ứng.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Số phần tử random** | **3 000 (30 000)** | **5 000 (50 000)** | **10 000 (100 000)** | **50 000 (500 000)** |
| **Modulo (ms)** | 0.49 | 0.65 | 1.63 | 10.95 |
| **Fibonacci (ms)** | 0.7 | 0.8 | 2.26 | 18.62 |
| **Modulo (Mb)** | 0 | 0 | 0 | 0 |
| **Fibonacci (Mb)** | 0 | 0 | 0 | 0 |
| **Modulo (va chạm)** | 3733 | 4242 | 8720 | 123067 |
| **Fibonacci (va chạm)** | 3414 | 3950 | 8126 | 66780 |

*Bảng 2.2: Thời gian chạy (ms), bộ nhớ sử dụng (Mb), tổng số va chạm (va chạm) của các thao tác search*

* Sequential Pattern
  + **INITIAL\_SIZE = 210 = 1024**
  + **INPUT\_FILE = “Sequential.txt”**
  + **LinearHashTable(*INITIAL\_SIZE, USE\_FIBONACCI)***
* Với thao tác *input(), insert(): số lần resize() được tính ra tương ứng.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Số phần tử random** | **30 000 (6 lần)** | **50 000 (7 lần)** | **100 000 (8 lần)** | **500 000 (10 lần)** |
| **Modulo (ms)** | 24.65 | 42.11 | 71.41 | 297.71 |
| **Fibonacci (ms)** | 20.93 | 48.38 | 83.73 | 398.55 |
| **Modulo (Mb)** | 5.57 | 10.56 | 21.53 | 84.59 |
| **Fibonacci (Mb)** | 5.57 | 10.56 | 21.59 | 84.35 |
| **Modulo (va chạm)** | 0 | 0 | 0 | 0 |
| **Fibonacci (va chạm)** | 1528 | 4620 | 23041 | 66850 |

*Bảng 3.1: Thời gian chạy (ms), bộ nhớ sử dụng (Mb), tổng số va chạm (va chạm) của các thao tác insert*

* Với thao tác *remove(), search()*: size của table sẽ bằng 10 lần số lượng phần tử tương ứng.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Số phần tử random** | **3 000 (30 000)** | **5 000 (50 000)** | **10 000 (100 000)** | **50 000 (500 000)** |
| **Modulo (ms)** | 0.3 | 0.66 | 1.19 | 3.99 |
| **Fibonacci (ms)** | 0.34 | 0.82 | 1.26 | 7.04 |
| **Modulo (Mb)** | 0 | 0 | 0 | 0 |
| **Fibonacci (Mb)** | 0 | 0 | 0 | 0 |
| **Modulo (va chạm)** | 0 | 0 | 0 | 0 |
| **Fibonacci (va chạm)** | 62 | 0 | 0 | 0 |

*Bảng 3.2: Thời gian chạy (ms), bộ nhớ sử dụng (Mb), tổng số va chạm (va chạm) của các thao tác search*

* Clustered Pattern
  + **INITIAL\_SIZE = 210 = 1024**
  + **INPUT\_FILE = “Clustered.txt”**
  + **LinearHashTable(*INITIAL\_SIZE, USE\_FIBONACCI)***
* Với thao tác *input(), insert(): số lần resize() được tính ra tương ứng.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Số phần tử random** | **30 000 (6 lần)** | **50 000 (7 lần)** | **100 000 (8 lần)** | **500 000 (10 lần)** |
| **Modulo (ms)** | 66.04 | 159.57 | 424.95 | 644.93 |
| **Fibonacci (ms)** | 31.96 | 45.15 | 87.38 | 437.07 |
| **Modulo (Mb)** | 5.58 | 10.56 | 21.59 | 84.34 |
| **Fibonacci (Mb)** | 5.57 | 10.57 | 21.53 | 84.35 |
| **Modulo (va chạm)** | 1441880 | 4509080 | 10212680 | 10212680 |
| **Fibonacci (va chạm)** | 70551 | 161082 | 238746 | 514208 |

*Bảng 4.1: Thời gian chạy (ms), bộ nhớ sử dụng (Mb), tổng số va chạm (va chạm) của các thao tác insert*

* Với thao tác *remove(), search()*: size của table sẽ bằng 10 lần số lượng phần tử tương ứng.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Số phần tử random** | **3 000 (30 000)** | **5 000 (50 000)** | **10 000 (100 000)** | **50 000 (500 000)** |
| **Modulo (ms)** | 5.69 | 12.37 | 9.21 | 45.03 |
| **Fibonacci (ms)** | 0.79 | 0.74 | 2.88 | 21.82 |
| **Modulo (Mb)** | 0 | 0 | 0 | 0 |
| **Fibonacci (Mb)** | 0 | 0 | 0 | 0 |
| **Modulo (va chạm)** | 160015 | 334893 | 205542 | 1053291 |
| **Fibonacci (va chạm)** | 4907 | 5381 | 9246 | 34720 |

*Bảng 4.2: Thời gian chạy (ms), bộ nhớ sử dụng (Mb), tổng số va chạm (va chạm) của các thao tác search*

* **Closed Addressing (Separate Chaining)**
* Random Pattern
  + **INITIAL\_SIZE = 210 = 1024**
  + **INPUT\_FILE = “Random.txt”**
  + **ChainHashTable(*INITIAL\_SIZE, USE\_FIBONACCI)***
* Với thao tác *input(), insert(): số lần resize() được tính ra tương ứng.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Số phần tử random** | **30 000 (5 lần)** | **50 000 (6 lần)** | **100 000 (7 lần)** | **500 000 (9 lần)** |
| **Modulo (ms)** | 16.39 | 28.86 | 69.14 | 417.81 |
| **Fibonacci (ms)** | 20.54 | 31.29 | 65.09 | 409.16 |
| **Modulo (Mb)** | 0.52 | 0.19 | 0 | 6.32 |
| **Fibonacci (Mb)** | 0.26 | 0.18 | 0 | 6.32 |
| **Modulo (va chạm)** | 73335 | 133104 | 269870 | 1259927 |
| **Fibonacci (va chạm)** | 73449 | 133500 | 270296 | 1261949 |

*Bảng 5.1: Thời gian chạy (ms), bộ nhớ sử dụng (Mb), tổng số va chạm (va chạm) của các thao tác insert*

* Với thao tác *remove(), search()*: size của table sẽ bằng 10 lần số lượng phần tử tương ứng.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Số phần tử random** | **3 000 (30 000)** | **5 000 (50 000)** | **10 000 (100 000)** | **50 000 (500 000)** |
| **Modulo (ms)** | 0.41 | 0.81 | 1 | 14.98 |
| **Fibonacci (ms)** | 0.72 | 0.59 | 1.19 | 16.8 |
| **Modulo (Mb)** | 0 | 0 | 0 | 0 |
| **Fibonacci (Mb)** | 0 | 0 | 0 | 0 |
| **Modulo (va chạm)** | 2678 | 564 | 2130 | 46332 |
| **Fibonacci (va chạm)** | 2646 | 625 | 2204 | 46453 |

*Bảng 5.2: Thời gian chạy (ms), bộ nhớ sử dụng (Mb), tổng số va chạm (va chạm) của các thao tác search*

* Sequential Pattern
  + **INITIAL\_SIZE = 210 = 1024**
  + **INPUT\_FILE = “Sequential.txt”**
  + **ChainHashTable(*INITIAL\_SIZE, USE\_FIBONACCI)***
* Với thao tác *input(), insert(): số lần resize() được tính ra tương ứng.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Số phần tử random** | **30 000 (5 lần)** | **50 000 (6 lần)** | **100 000 (7 lần)** | **500 000 (9 lần)** |
| **Modulo (ms)** | 15.69 | 25.4 | 170 | 263.25 |
| **Fibonacci (ms)** | 15.42 | 28.76 | 69.4 | 425.13 |
| **Modulo (Mb)** | 0.48 | 0.2 | 0.01 | 6.36 |
| **Fibonacci (Mb)** | 0.52 | 0.2 | 0 | 6.32 |
| **Modulo (va chạm)** | 59704 | 112474 | 228012 | 1021232 |
| **Fibonacci (va chạm)** | 59538 | 110981 | 224711 | 1028418 |

*Bảng 6.1: Thời gian chạy (ms), bộ nhớ sử dụng (Mb), tổng số va chạm (va chạm) của các thao tác insert*

* Với thao tác *remove(), search()*: size của table sẽ bằng 10 lần số lượng phần tử tương ứng.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Số phần tử random** | **3 000 (30 000)** | **5 000 (50 000)** | **10 000 (100 000)** | **50 000 (500 000)** |
| **Modulo (ms)** | 0.47 | 0.98 | 2.41 | 10.49 |
| **Fibonacci (ms)** | 0.45 | 0.63 | 2.54 | 24.48 |
| **Modulo (Mb)** | 0 | 0 | 0 | 0 |
| **Fibonacci (Mb)** | 0 | 0 | 0 | 0 |
| **Modulo (va chạm)** | 1303 | 2693 | 10000 | 49999 |
| **Fibonacci (va chạm)** | 1390 | 2678 | 6770 | 78468 |

*Bảng 6.2: Thời gian chạy (ms), bộ nhớ sử dụng (Mb), tổng số va chạm (va chạm) của các thao tác search*

* Clustered Pattern
  + **INITIAL\_SIZE = 210 = 1024**
  + **INPUT\_FILE = “Clustered.txt”**
  + **ChainHashTable(*INITIAL\_SIZE, USE\_FIBONACCI)***
* Với thao tác *input(), insert(): số lần resize() được tính ra tương ứng.*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Số phần tử random** | **30 000 (5 lần)** | **50 000 (6 lần)** | **100 000 (7 lần)** | **500 000 (9 lần)** |
| **Modulo (ms)** | 14.56 | 27.58 | 74.63 | 285.27 |
| **Fibonacci (ms)** | 19.58 | 28.74 | 64.01 | 358.42 |
| **Modulo (Mb)** | 0.54 | 0.16 | 0 | 6.29 |
| **Fibonacci (Mb)** | 0.52 | 0.46 | 0 | 6.3 |
| **Modulo (va chạm)** | 61183 | 111281 | 221143 | 1148578 |
| **Fibonacci (va chạm)** | 85647 | 149470 | 272514 | 1090568 |

*Bảng 7.1: Thời gian chạy (ms), bộ nhớ sử dụng (Mb), tổng số va chạm (va chạm) của các thao tác insert*

* Với thao tác *remove(), search()*: size của table sẽ bằng 10 lần số lượng phần tử tương ứng.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Số phần tử random** | **3 000 (30 000)** | **5 000 (50 000)** | **10 000 (100 000)** | **50 000 (500 000)** |
| **Modulo (ms)** | 0.38 | 0.43 | 2.09 | 14.06 |
| **Fibonacci (ms)** | 0.66 | 0.58 | 1.02 | 16.84 |
| **Modulo (Mb)** | 0 | 0 | 0 | 0 |
| **Fibonacci (Mb)** | 0 | 0 | 0 | 0 |
| **Modulo (va chạm)** | 2694 | 508 | 2099 | 43941 |
| **Fibonacci (va chạm)** | 2597 | 548 | 2237 | 45325 |

*Bảng 7.2: Thời gian chạy (ms), bộ nhớ sử dụng (Mb), tổng số va chạm (va chạm) của các thao tác search*

### **Analysis of results**

* **Open Addressing (Linear Probing)**
* Thời gian chạy

*Biểu đồ 1.1: Thời gian chạy (ms) của 2 phương pháp băm ứng với 3 pattern*

* Khi số lượng phần tử tăng lên, thời gian chạy của tất cả các phương pháp đều tăng.
* Phương pháp modulo hashing khi thử nghiệm với clustered pattern có thời gian chạy lâu hơn hẳn. Tuy nhiên, modulo ở sequential pattern có thời gian chạy nhanh hơn các đường khác.
* Fibonacci hashing cho thời gian ổn định hơn và ít bị ảnh hưởng bởi các pattern của keys.
* Bộ nhớ sử dụng

*Biểu đồ 1.2: Bộ nhớ sử dụng (Mb) của 2 phương pháp băm ứng với 3 pattern*

* Khi số lượng phần tử tăng lên, bộ nhớ sử dụng của tất cả các phương pháp đều tăng.
* Cả hai phương pháp hashing đều sử dụng lượng bộ nhớ gần như giống nhau ở mọi keys pattern.
* Tổng số va chạm

*Biểu đồ 1.3: Tổng số va chạm của 2 phương pháp băm ứng với 3 pattern*

* Số va chạm của Modulo hashing ở clustered pattern cao hơn rất nhiều so với các đường còn lại.
* Ở các pattern còn lại thì số va chạm của Modulo hashing và Fibonacci hashing gần như tương đương và tăng dần theo số lượng phần tử.
* **Closed Addressing (Separate Chaining)**
* Thời gian chạy

*Biểu đồ 2.1: Thời gian chạy (ms) của 2 phương pháp băm ứng với 3 pattern*

* Ở tất cả các keys pattern, thời gian chạy giữa Fibonacci hashing và Modulo hashing gần như tương đương nhau.
* Cả hai hàm băm đều ổn định hơn nhiều so với Linear Probing khi xử lý clustered pattern.
* Bộ nhớ sử dụng

*Biểu đồ 2.2: Bộ nhớ sử dụng (Mb) của 2 phương pháp băm ứng với 3 pattern*

* Mức sử dụng bộ nhớ của tất cả các trường hợp gần như giống nhau tuyệt đối.
* Tổng số va chạm

*Biểu đồ 2.3: Tổng số va chạm của 2 phương pháp băm ứng với 3 pattern*

* Tổng số va chạm của tất cả các trường hợp cũng gần như giống nhau.

## **Conclusion**

* **Modulo hashing**
* Dễ cài đặt, tốc độ xử lý tốt với dữ liệu random hoặc sorted.
* Tuy nhiên, rất dễ bị ảnh hưởng bởi dữ liệu phân cụm (clustered key pattern), đặc biệt khi dùng với Linear Probing gây ra va chạm hàng loạt, làm giảm hiệu năng nghiêm trọng.
* Trong Separate Chaining, số va chạm cao hơn Fibonacci, nhất là với các pattern có quy luật.
* **Fibonacci hashing**
* Phân bố index đều hơn nhờ tỉ lệ vàng, hạn chế tối đa va chạm, đặc biệt hiệu quả khi dữ liệu bị sắp xếp hoặc phân cụm.
* Hiệu năng ổn định hơn Modulo trong mọi trường hợp.
* Khó cài đặt và tương đối phức tạp hơn so với Modulo hashing.

## **Reference**

[1] <https://www.youtube.com/watch?v=2_3fR-k-LzI&t=381s>

[2] <https://www.youtube.com/watch?v=y11XNXi9dgs&t=137s>

[3] <https://www.youtube.com/watch?v=KyUTuwz_b7Q>

[4] <https://en-m-wikipedia-org.translate.goog/wiki/Golden_ratio?_x_tr_sl=en&_x_tr_tl=vi&_x_tr_hl=vi&_x_tr_pto=tc>

[5] <https://www.geeksforgeeks.org/bitwise-operators-in-c-cpp/>