

VIETNAM NATIONAL UNIVERSITY - HO CHI MINH
UNIVERSITY OF SCIENCE

Fibonacci Hashing

Techincal Report

CSC10004 - Data Structure And Algorithm

Students:

24120059 - Trần KIM HỮU
24120041 - Phạm Võ ĐỨC
24120006 - Đào THANH PHONG
24120069 - Trần HOÀI BẢO KHANG

Supervisors:

Lect. Lê NHỰT NAM

Ngày 25 tháng 6 năm 2025

MỤC LỤC

DANH MỤC CÁC BẢNG	4
DANH MỤC CÁC HÌNH VẼ, ĐỒ THỊ	5
TÓM TẮT	8
I GIỚI THIỆU	10
1 Bối cảnh	10
2 Lịch sử	10
3 Giới hạn báo cáo	11
4 Cấu trúc báo cáo	11
5 Bảng phân công	12
II MỘT SỐ HIỂU BIẾT NỀN TẢNG	13
1 Hashing	13
1.1 Nguyên lý cơ bản	13
1.2 Các thành phần của Hashing	13
1.3 Hash Table	13
1.4 Hash Function	14
1.5 Sơ đồ minh họa hashing cơ bản	15
1.6 Vấn đề xung đột (Collision)	15
1.7 Các phương pháp xử lý xung đột phổ biến	15
1.8 Vai trò của Hashing trong thực tế	15
1.9 Hàm băm truyền thống và hạn chế	15
2 Fibonacci Hashing	16
2.1 Tỷ lệ vàng và Fibonacci Hashing	16

2.2	Ý tưởng chính của Fibonacci Hashing	17
2.3	Nguyên lý toán học	17
2.4	Ưu - Nhược điểm của Fibonacci Hashing	18
2.5	Lợi ích kì vọng	18
III CHI TIẾT THUẬT TOÁN		20
1	Dữ liệu đầu vào và dữ liệu đầu ra	20
1.1	Input	20
1.2	Output	20
2	Mã giả (Pseudo code)	21
2.1	Tìm kích thước bảng băm	21
2.2	Hàm băm Fibonacci	21
2.3	Xây dựng bảng băm	21
2.4	Thêm phần tử	21
2.5	Tái băm	22
2.6	Tìm kiếm	22
2.7	Xóa phần tử	23
IV CÀI ĐẶT THUẬT TOÁN		24
1	Cấu hình thực nghiệm	24
2	Cách cài	24
2.1	Ngôn ngữ và công cụ sử dụng	24
2.2	Các thao tác chính	24
3	Kết quả đo đạt	34
3.1	Thực nghiệm Test1	34
3.2	Thực nghiệm Test2	46
3.3	Thực nghiệm Test3	57
4	Nhận xét kết quả	68
4.1	Ưu điểm	68

4.2	Hạn chế	68
4.3	Thực nghiệm theo từng loại dữ liệu	69
4.4	So sánh tổng thể	69
V	KẾT LUẬN	70
1	Các kết quả đạt được	70
2	Những kiến thức thu nhận được	70
3	Dánh giá tổng quan	71
4	Hướng phát triển trong tương lai	71

DANH MỤC CÁC BẢNG

Bảng I.1	Bảng phân công công việc trong nhóm	12
Bảng IV.1	Bảng số liệu Test1 dạng Clustered	35
Bảng IV.2	Bảng số liệu Test1 dạng Random	36
Bảng IV.3	Bảng số liệu Test1 dạng Sequential	37
Bảng IV.4	Bảng số liệu Test2 dạng Clustered	46
Bảng IV.5	Bảng số liệu Test2 dạng Random	47
Bảng IV.6	Bảng số liệu Test2 dạng Sequential	48
Bảng IV.7	Bảng số liệu Test3 dạng Clustered	57
Bảng IV.8	Bảng số liệu Test3 dạng Random	58
Bảng IV.9	Bảng số liệu Test3 dạng Sequential	59
Bảng IV.10	So sánh hiệu năng giữa Fibonacci và Modulo Hashing	69

DANH MỤC CÁC HÌNH VẼ, ĐỒ THỊ

Hình II.1 Minh họa hàm băm và xung đột khi $h(x) = x \bmod 5$	15
Hình IV.1 Thời gian Build Test1	38
Hình IV.2 Bộ nhớ tiêu tốn để Build Test1	39
Hình IV.3 Load Factor có ở Test1	40
Hình IV.4 Average Chain có ở Test1	41
Hình IV.5 Max Chain có ở Test1	42
Hình IV.6 Thời gian Search ở Test1	43
Hình IV.7 Thời gian Delete ở Test1	44
Hình IV.8 Thời gian Insert ở Test1	45
Hình IV.9 Thời gian Build Test2	49
Hình IV.10Bộ nhớ tiêu tốn để Build Test2	50
Hình IV.11Load Factor có ở Test2	51
Hình IV.12Average Chain có ở Test2	52
Hình IV.13Max Chain có ở Test2	53
Hình IV.14Thời gian Search ở Test2	54
Hình IV.15Thời gian Delete ở Test2	55
Hình IV.16Thời gian Insert ở Test2	56
Hình IV.17Thời gian Build Test3	60
Hình IV.18Bộ nhớ tiêu tốn để Build Test3	61
Hình IV.19Load Factor có ở Test3	62
Hình IV.20Average Chain có ở Test3	63
Hình IV.21Max Chain có ở Test3	64
Hình IV.22Thời gian Search ở Test3	65
Hình IV.23Thời gian Delete ở Test3	66

LỜI CẢM ƠN

Trong kỷ nguyên của dữ liệu lớn và các hệ thống vận hành theo thời gian thực, việc truy xuất thông tin một cách nhanh chóng và ổn định không còn là lợi thế, mà là điều kiện tiên quyết để đảm bảo hiệu năng hệ thống. Từ công cụ tìm kiếm, cơ sở dữ liệu đến các trình biên dịch hay phần mềm nhúng, yêu cầu truy cập dữ liệu trong thời gian hằng định ngày càng trở nên cấp thiết. Trong bối cảnh đó, các kỹ thuật băm (hashing) tiếp tục khẳng định vai trò cốt lõi trong việc xây dựng những cấu trúc dữ liệu tối ưu.

Một trong những bước tiến đáng chú ý trong lĩnh vực này là Fibonacci Hashing – một phương pháp tận dụng tỷ lệ vàng để phân phối khóa một cách đồng đều trên bảng băm. Thay vì dựa vào các phép chia truyền thống dễ gây xung đột hoặc tạo cụm khóa, kỹ thuật này khai thác đặc điểm toán học của dãy Fibonacci nhằm tạo ra một ánh xạ vừa đơn giản vừa hiệu quả. Điểm mạnh của Fibonacci Hashing nằm ở hiệu suất cao, khả năng hiện thực dễ dàng, và đặc biệt là tính tương thích với các bảng băm có kích thước là lũy thừa của 2.

Dè tài này là cơ hội để chúng em không chỉ đào sâu vào cấu trúc bảng băm và các kỹ thuật ánh xạ khóa thông minh, mà còn kết nối chặt chẽ giữa kiến thức toán học trừu tượng và ứng dụng lập trình thực tiễn. Trong quá trình thực hiện, chúng em đã triển khai và đánh giá Fibonacci Hashing thông qua các thí nghiệm đo hiệu năng và khả năng phân bố khóa, từ đó rút ra những nhận định khách quan về tiềm năng ứng dụng của kỹ thuật này.

Chúng em xin bày tỏ lòng biết ơn sâu sắc nhất đến thầy Lê Nhựt Nam – người đã tận tình hướng dẫn, truyền đạt kiến thức và tạo điều kiện, hỗ trợ chúng em trong suốt quá trình học tập và nghiên cứu. Dè tài này là cơ hội quý báu giúp chúng em kết nối kiến thức lý thuyết với ứng dụng thực tế, phát triển kỹ năng phân tích, tư duy thuật toán và khả năng trình bày khoa học.

Chúng em xin chân thành cảm ơn.

TÓM TẮT

Trong khoa học máy tính, việc thiết kế cấu trúc dữ liệu hiệu quả đóng vai trò then chốt trong việc nâng cao hiệu suất truy xuất thông tin. Trong đó, bảng băm (*Hash Table*) là một cấu trúc dữ liệu phổ biến nhờ khả năng hỗ trợ các thao tác tìm kiếm, chèn và xoá với thời gian trung bình $O(1)$. Tuy nhiên, hiệu quả thực tế của bảng băm phụ thuộc lớn vào chất lượng của hàm băm và chiến lược xử lý xung đột.

Đề tài “**Fibonacci Hashing**” tập trung nghiên cứu và triển khai kỹ thuật băm sử dụng phép nhân với hằng số tỷ lệ vàng (golden ratio) nhằm cải thiện khả năng phân phối khóa trong bảng băm. Khác với phương pháp chia lấy dư truyền thống (modulo), Fibonacci Hashing thay thế phép chia bằng phép nhân và dịch bit, giúp phân tán khóa đều hơn, giảm va chạm (*collision*) và hiện tượng cụm hóa (*clustering*), đồng thời tận dụng tốt khả năng xử lý của CPU.

Hệ thống được xây dựng với bảng băm có kích thước là lũy thừa của 2 để ta có thể tối ưu hoá được phép dịch bit, với công thức băm:

$$\text{index} = (\text{key} \times \phi) \gg (w - n)$$

trong đó $\phi = 2654435769$ là hằng số tỷ lệ vàng ở dạng số nguyên 32-bit, w là độ dài từ (word size) của hệ thống.

Bằng cách áp dụng kỹ thuật này, chúng em đã thực hiện thực nghiệm với ba loại dữ liệu: *clustered*, *random*, *sequential*, và nhiều mức tải khác nhau (10% đến 100%). Mỗi thao tác (xây dựng bảng, chèn, tìm kiếm, xoá) được thực hiện một triệu lần để đo thời gian trung bình, đồng thời ghi nhận thêm các chỉ số như: mức sử dụng bộ nhớ, độ dài chuỗi trung bình và chuỗi dài nhất.

Kết quả thực nghiệm cho thấy Fibonacci Hashing đạt hiệu năng ổn định và cao hơn hoặc tương đương phương pháp modulo trong phần lớn trường hợp, đặc biệt với dữ liệu tuần tự. Một số ưu điểm nổi bật như sau:

- Tốc độ chèn, tìm kiếm, xoá nhanh và ổn định, nhất là trên dữ liệu *sequential*.
- Phân phối chuỗi đều hơn, độ dài chuỗi trung bình và tối đa thấp trên tập *random*.

- Tiết kiệm bộ nhớ và thời gian xây dựng bảng trong một số mức tải cụ thể.

Tuy nhiên, Fibonacci Hashing yêu cầu bảng băm có kích thước là 2^n và có thể không đạt hiệu quả tối ưu với dữ liệu bị phân cụm cao. Dù vậy, với sự ổn định, hiệu suất tốt và dễ hiện thực, Fibonacci Hashing là lựa chọn phù hợp cho các hệ thống yêu cầu truy xuất nhanh, như từ điển, trình biên dịch, hoặc bảng ký hiệu trong lập trình hiệu suất cao.

CHƯƠNG I

GIỚI THIỆU

1. Bối cảnh

Bảng băm (hash table) từ lâu đã trở thành “trợ thủ đắc lực” trong các hệ thống xử lý dữ liệu nhờ khả năng tìm kiếm, chèn và xóa thông tin với hiệu suất cao. Nhưng dường sau tốc độ ấy là một bài toán kỹ thuật quan trọng: thiết kế hàm băm sao cho tối ưu. Trong số các phương pháp phổ biến, phép chia lấy dư (modulo) thường được sử dụng để ánh xạ khóa về chỉ số trong bảng. Tuy đơn giản và dễ triển khai, nhưng cách làm này lại tiềm ẩn nhiều hạn chế như phân phối khóa thiếu đồng đều và dễ gây ra va chạm dữ liệu.

Để khắc phục những vấn đề đó, Fibonacci Hashing ra đời như một phương án thay thế thông minh. Tận dụng tỷ lệ vàng và các phép toán đơn giản như nhân và dịch bit, phương pháp này sẽ giúp ta cải thiện đáng kể được độ phân tán các khóa và khả năng va chạm dữ liệu.

2. Lịch sử

Ý tưởng sử dụng phép nhân thay vì phép chia để băm không phải là mới. Ngay từ những năm đầu của khoa học máy tính, Donald E. Knuth, một huyền thoại trong ngành, đã đề cập đến kỹ thuật này trong cuốn sách kinh điển "The Art of Computer Programming – Volume 3: Sorting and Searching" [1]. Ông gọi đây là multiplicative hashing và đặc biệt nhấn mạnh hiệu quả của nó khi kích thước bảng băm là lũy thừa của 2.

Knuth đề xuất rằng nên chọn hằng số nhân gần với tỷ lệ vàng $\Phi \approx 1.618$, vì nó giúp phân bố giá trị băm đều hơn trên toàn bộ bảng băm, ngay cả khi các khóa đầu vào có tính tuần hoàn hoặc tương quan cao. Khi được biểu diễn ở dạng số nguyên 32-bit, hằng số này tương ứng với $2^{32} \approx 2654435769$.

Một bước ngoặt đáng chú ý xảy ra vào năm 2018, khi Daniel Lemire công bố bài viết có tiêu đề “Fibonacci Hashing: The Optimization That the World Forgot” [2], trong đó ông phân tích lại kỹ thuật này dưới góc độ thực tiễn và cho thấy Fibonacci Hashing có hiệu suất vượt trội so với hàm

modulo truyền thông trong nhiều trường hợp. Nhờ vào phép nhân và dịch bit – vốn là các phép toán rất nhanh trên CPU hiện đại – Fibonacci Hashing vừa đạt được tốc độ cao, vừa cải thiện đáng kể khả năng phân phối khóa, giảm thiểu va chạm và hiện tượng clustering.

Đến hiện tại, Fibonacci Hashing có một số ứng dụng trong:

- Trình biên dịch và hệ thống runtime
- Bộ nhớ đệm (cache) và hệ thống lưu trữ
- Cấu trúc dữ liệu trong game engine hoặc đồ họa máy tính
- Các thư viện lập trình hiệu suất cao như Abseil hay Rust HashMap.

3. Giới hạn báo cáo

Bài báo cáo tập trung vào kỹ thuật **Fibonacci Hashing**, một phương pháp băm sử dụng phép nhân với hằng số tỉ lệ vàng nhằm cải thiện phân bố khóa trong bảng băm. Nội dung được xây dựng dựa trên lý thuyết từ sách *The Art of Computer Programming - Volume 3* của Donald E. Knuth [1] và bài viết phân tích thực nghiệm của Daniel Lemire [2].

Phạm vi kiến thức được giới hạn trong nội dung giảng dạy của học phần **Cấu trúc dữ liệu và giải thuật**, có tham khảo thêm từ các nguồn uy tín như trang Geeks for Geeks và một số bài nghiên cứu liên quan.

Dữ liệu đầu vào được giả định là tập hợp các khóa kiểu số nguyên không âm (unsigned int), không thay đổi sau khi xây dựng bảng băm (tĩnh). Kích thước bảng băm là lũy thừa của 2 để đảm bảo áp dụng tối ưu thuật toán Fibonacci Hashing. Báo cáo không mở rộng sang các biến thể nâng cao như băm động, băm hai cấp hay tái băm.

4. Cấu trúc báo cáo

Báo cáo gồm các phần chính: (i) giới thiệu tổng quan đồ án, (ii) tổng hợp kiến thức nền tảng, (iii) Chi tiết thuật toán áp dụng, (iv) cài đặt và đo kết quả thực nghiệm, và (v) kết luận cùng hướng phát triển cho đồ án.

5. Bảng phân công

MSSV	Họ và tên	Nội dung thực hiện
24120059	Trần Kim Hữu	Tìm hiểu lý thuyết độ phức tạp thuật toán, viết, chỉnh sửa thuật toán và cài đặt chương trình.
24120041	Phạm Võ Đức	Viết báo cáo, tìm hiểu lý thuyết Hashing, Hash Table và tổng hợp tài liệu.
24120006	Đào Thanh Phong	Tìm hiểu lý thuyết Fibonacci Hashing, và tìm hiểu lý thuyết Universal Hashing.
24120069	Trần Hoài Bảo Khang	Tạo và xử lý dữ liệu đầu vào, đo thực nghiệm, tạo bảng so sánh

Bảng I.1: Bảng phân công công việc trong nhóm

CHƯƠNG II

MỘT SỐ HIỂU BIẾT NỀN TẢNG

1. Hashing

Hashing (băm) là một kỹ thuật ánh xạ một khóa (key) từ tập dữ liệu đầu vào sang một chỉ số trong bảng (mảng) để lưu trữ hoặc tra cứu nhanh chóng. Mục tiêu chính của hashing là thực hiện các thao tác như tra cứu, chèn hoặc xoá phần tử trong thời gian trung bình $O(1)$, tức là không phụ thuộc vào kích thước của dữ liệu.

1.1. Nguyên lý cơ bản

Ý tưởng của hashing là sử dụng một **hàm băm** $h(x)$ để ánh xạ một khóa x (thường là chuỗi, số nguyên, v.v.) đến một vị trí trong bảng băm có kích thước m :

Với mỗi phần tử x , hàm băm sẽ sinh ra một chỉ số trong khoảng $[0, m - 1]$ và phần tử đó sẽ được lưu ở vị trí tương ứng trong bảng.

1.2. Các thành phần của Hashing

Hashing gồm bốn thành phần cơ bản:

- **Hash Table:** Cấu trúc dữ liệu dùng để lưu trữ các phần tử đã băm.
- **Hash Function:** Hàm dùng để ánh xạ khóa đầu vào thành chỉ số trong bảng.
- **Collision:** Tình huống xảy ra khi hai khóa khác nhau có cùng giá trị băm.
- **Collision Resolution Techniques:** Các kỹ thuật xử lý xung đột nhằm đảm bảo tính toàn vẹn và hiệu suất.

1.3. Hash Table

Hash Table là một cấu trúc dữ liệu dạng mảng được dùng để lưu trữ các phần tử đã được ánh xạ (băm) từ khóa đầu vào. Kích thước bảng thường được chọn là một số nguyên tố hoặc lũy thừa của

2 nhầm phân phối chỉ số đều hơn, hạn chế xung đột.

Mỗi ô trong bảng có thể chứa:

- Một giá trị duy nhất (nếu không có xung đột)
- Một danh sách liên kết các phần tử (chaining)
- Một con trỏ hoặc cặp key–value

Bảng băm hỗ trợ các thao tác cơ bản với thời gian trung bình $O(1)$:

- Insert (Thêm phần tử)
- Search (Tìm kiếm phần tử)
- Delete (Xoá phần tử)

1.4. Hash Function

Hash Function (hàm băm) là một công thức hoặc thuật toán nhận đầu vào là một khóa (key) và trả về một chỉ số trong bảng băm. Một hàm băm hiệu quả cần đảm bảo:

- Phân phối đều các khóa vào các chỉ số trong bảng
- Tính toán nhanh và đơn giản
- Tránh sinh ra cùng giá trị băm cho các khóa khác nhau (giảm xung đột)

Ví dụ: $h(x) = x \bmod m$.

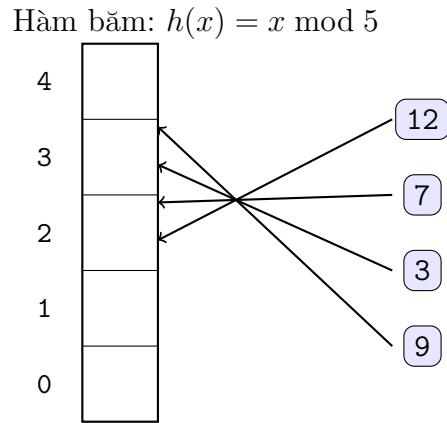
Một số hàm băm phổ biến:

- **Hàm băm chuỗi:** Dựa trên công thức:

trong đó a_i là mã ASCII của ký tự thứ i trong chuỗi, p là số nguyên lớn.

- **Hàm băm hỗn hợp:** Kết hợp các phép toán như XOR, shift, nhân để tăng tính ngẫu nhiên và giảm xung đột.

1.5. Sơ đồ minh họa hashing cơ bản



Hình II.1: Minh họa hàm băm và xung đột khi $h(x) = x \bmod 5$

Hình minh họa sử dụng phương pháp băm modulo truyền thống — Fibonacci Hashing sẽ thay thế phép chia này bằng phép nhân và dịch bit để giảm va chạm.

1.6. Vấn đề xung đột (Collision)

Một trong những thách thức lớn của hashing là **xung đột** (collision): khi hai khóa khác nhau $x \neq y$ lại có cùng giá trị băm, tức là $h(x) = h(y)$.

1.7. Các phương pháp xử lý xung đột phổ biến

- Chaining (Băm liên kết)
- Linear/Quadratic Probing, Double Hashing

1.8. Vai trò của Hashing trong thực tế

Ứng dụng trong dữ liệu, cơ sở dữ liệu, trình biên dịch, bảo mật, thuật toán.

1.9. Hàm băm truyền thống và hạn chế

Trong các triển khai bảng băm truyền thống, khi ánh xạ khóa vào một khe bảng băm, chúng ta thường sử dụng phép toán chia lấy dư:

```
index = key % tableSize
```

Trong đó:

- key: Giá trị (số nguyên) cần lưu trữ vào bảng băm.
- tableSize: Kích thước của bảng băm, tức là số ô có thể lưu trữ.
- index: Chỉ số (vị trí) trong bảng băm nơi key sẽ được lưu.

Nếu tableSize là lũy thừa của 2 (ví dụ: $2^{10} = 1024$), có thể tối ưu hóa phép chia như sau:

```
index = key & (tableSize - 1)
```

Do khi bảng đầy và cần mở rộng, có thể gấp đôi ($2^n \rightarrow 2^{n+1}$) dễ dàng mà vẫn giữ được cấu trúc dữ liệu, ngoài ra việc rehash (băm lại toàn bộ) cũng đơn giản hơn khi kích thước là lũy thừa của 2. Cách này sử dụng phép AND bit, nhanh hơn phép chia nhưng có nhược điểm lớn là chỉ xét các bit thấp của khóa và bỏ qua phần cao.

Vấn đề của phép chia truyền thống

- Phân phối kém: Nếu các khóa có mẫu lặp ở bit thấp, xảy ra nhiều va chạm.
- Tạo cụm dẽ đoán: Các khóa liên tiếp thường băm vào các vị trí liên tiếp.
- Khai thác không hiệu quả: Một số ô bị bỏ trống, trong khi các ô khác chứa nhiều phần tử.

Ví dụ: các địa chỉ bộ nhớ liên tiếp (thường thấy trong thực tế) sẽ dẫn đến phân phối rất lệch trong bảng băm.

2. Fibonacci Hashing

2.1. Tỷ lệ vàng và Fibonacci Hashing

Fibonacci Hashing (còn gọi là băm nhân) sử dụng tỷ lệ vàng $\phi \approx 1.618$ để phân phối khóa đều hơn. Tỷ lệ vàng liên hệ chặt với dãy Fibonacci: tỷ số giữa 2 số liên tiếp trong dãy Fibonacci càng cao càng tiệm cận ϕ .

2.2. Ý tưởng chính của Fibonacci Hashing

Fibonacci Hashing sử dụng một hằng số có liên quan đến tỷ lệ vàng (golden ratio), được biểu diễn dưới dạng nhị phân, để phân tán các khóa một cách đồng đều. // Kỹ thuật này được áp dụng hiệu quả khi bảng băm có kích thước 2^k .

2.3. Nguyên lý toán học

- Ý tưởng cốt lõi là nhân khóa với tỷ lệ vàng để tạo hiệu ứng “cuộn” và phân phối khóa đều trên bảng băm bất kể mẫu trong khóa.
- Với bảng băm có kích thước 2^n , công thức:

```
index = (key * phi) mod 2^n
```

Có thể tối ưu thành:

```
index = (key * phi) >> (wordSize - n)
```

Trong thực tế, với hệ thống 32-bit, dùng:

```
index = (key * 2654435769) >> (32 - n)
```

Ở đây:

- $2654435769 \approx \phi \times 2^{32}$ (làm tròn thành số nguyên 32-bit).
- \gg : là phép dịch phải bit.
- **wordSize**: là số bit của kiểu dữ liệu nguyên dùng để biểu diễn **key**.
- **n**: là số bit cần thiết để biểu diễn các chỉ số của bảng băm, tức là: $n = \log_2(\text{tableSize})$

(chú ý: điều này chỉ áp dụng khi kích thước của bảng băm là lũy thừa của 2)

Mục đích của phép dịch phải $\gg (wordSize - n)$

Phép dịch này lấy các bit có trọng số cao nhất trong kết quả nhân $\text{key} \times \phi$. Nó hoạt động giống như phép chia lấy phần nguyên theo $2^{\text{wordSize}-n}$. Điều này giúp chọn ra n bit đầu tiên (trong biểu diễn nhị phân) để làm chỉ số bảng băm – một kỹ thuật giúp phân bố chỉ số đều hơn trên bảng.

2.4. *Ưu - Nhược điểm của Fibonacci Hashing*

Ưu điểm

- Phân phối tốt hơn: Khóa có mẫu vẫn phân bố đều.
- Giảm cụm: Các khóa liên tiếp ánh xạ đến vị trí cách xa nhau.
- Hiệu suất cao hơn: Ít va chạm → chèn, tìm, xóa nhanh hơn.
- Tính toán nhanh: Chỉ dùng phép nhân và dịch bit.
- Tốt hơn khi khóa có tính tuần tự (sequential keys).

Nhược điểm:

- Phân bố kém khi khóa có mẫu giống nhau (cluster).
- Tạo ra các cụm (clustering) → tăng số lần va chạm (collision).
- Sử dụng không hiệu quả không gian bảng băm.

2.5. *Lợi ích kì vọng*

- Phân phối khóa đều hơn: Tránh xung đột (collision) tốt hơn các hàm băm đơn giản như `key % tableSize`.
- Hiệu suất cao: Fibonacci Hashing dùng phép nhân ($\text{key} \times \phi$) và dịch bit (\gg), cả 2 đều rất nhanh ở cấp độ CPU.
- Thích hợp với bảng băm kích thước 2^n : Khi bảng băm có kích thước là 2^n , Fibonacci Hashing không cần dùng phép `mod` (như `key % tableSize`) mà chỉ cần lấy n bit cao nhất của tích ($\text{key} \times \phi$).
- Chống lại bias từ khóa “xấu”:
 - Các kỹ thuật như `key % tableSize` có thể bị lỗi phân phối nếu :
 - `tableSize` không nguyên tố.
 - Khóa có dạng tuần hoàn (ví dụ: 0, 1000, 2000, ...).

- Fibonacci Hashing giảm đáng kể hiện tượng bias này nhờ nhân với một số “phi tuyế” như ϕ .

CHƯƠNG III

CHI TIẾT THUẬT TOÁN

1. Dữ liệu đầu vào và dữ liệu đầu ra

1.1. *Input*

Dữ liệu đầu vào là tệp văn bản Test1, Test2 và Test3, gồm nhiều dòng, mỗi dòng chứa một mục văn bản.

Mỗi tệp được chia thành ba dạng là: Clustered, Random và Sequential. Ta sẽ cho thử nghiệm độ lớn từ 10% đến 100% để đánh giá ở từng kích thước dữ liệu. Tập khóa được trích xuất từ đầu dòng, giữ lại tối đa 100 ký tự đầu tiên và chỉ xét các ký tự thuộc mã ASCII chuẩn (0–127). Các dòng trống, dòng chỉ chứa khoảng trắng hoặc ít hơn 3 ký tự sẽ bị loại bỏ.

1.2. *Output*

Sau khi xây dựng bảng băm, chương trình cho phép người dùng thực hiện các thao tác tra cứu từ bàn phím. Mỗi từ khóa tra cứu được lặp lại 1 000 000 lần nhằm đo thời gian truy xuất trung bình.

Kết quả trả về bao gồm:

- Nội dung dòng dữ liệu đầy đủ tương ứng nếu tìm thấy khóa.
- “Not Found” nếu khóa không tồn tại trong bảng băm.
- Thời gian trung bình tra cứu một từ khóa (tính bằng microsecond).

Bên cạnh đó, chương trình cũng hiển thị:

- Thời gian xây dựng bảng băm (tính bằng milliseconds).
- Bộ nhớ sử dụng khi xây dựng bảng băm (tính bằng MB).
- Độ dài chuỗi trung bình (Average Chain Length) và dài nhất (Max Chain Length) trong bảng.

2. Mã giả (Pseudo code)

2.1. Tìm kích thước bảng băm

```
function findTableSize(size):
    res <- 1
    n <- 0
    while res <= size:
        res <- res * 2
        n <- n + 1
    return res
```

2.2. Hàm băm Fibonacci

```
function hashFunction(key):
    k <- 0
    for each character c in key:
        if c is not valid ASCII:
            return -1
        k <- k * 31 + ASCII(c)
    return (k * phi) >> (w - n)
```

2.3. Xây dựng bảng băm

```
function build(path):
    data <- read lines from path
    m <- findTableSize(length of data)
    initialize hashTable with m empty buckets
    for each line in data:
        insert(line)
```

2.4. Thêm phần tử

```
function insert(data):
    if loadFactor >= 1.0:
```

```

reHashing()

index <- hashFunction(data.key)

if index is invalid:
    return false

for each cur in table[index]:
    if cur.key == data.key:
        cur.val <- data.val
        return true

append data to table[index]
count <- count + 1

return true

```

2.5. Tái băm

```

function reHashing():

    m <- m * 2
    n <- n + 1
    count <- 0
    oldTable <- table
    initialize new table with m buckets
    for each bucket in oldTable:
        for each val in bucket:
            insert(val)

```

2.6. Tìm kiếm

```

function search(key):

    index <- hashFunction(key)

    if index invalid or bucket empty:
        return Not Found

    for each cur in table[index]:
        if cur.key == key:
            return cur.val

```

```
    return Not Found
```

2.7. Xóa phần tử

```
function delete(key):  
    index <- hashFunction(key)  
    if index invalid or bucket empty:  
        return false  
    for each cur in table[index]:  
        if cur.key == key:  
            remove cur from table[index]  
            count <- count - 1  
        return true  
    return false
```

CHƯƠNG IV

CÀI ĐẶT THUẬT TOÁN

1. Cấu hình thực nghiệm

- CPU: Intel Core i7-1065G7
- RAM: 20 GB
- Clock Speed: 1.3-3.9 GHz

2. Cách cài

2.1. Ngôn ngữ và công cụ sử dụng

Chương trình được viết bằng ngôn ngữ C++**17**, biên dịch bằng g++, chạy trên môi trường Windows. Các thư viện được sử dụng bao gồm:

- <vector>, <list>: để cài đặt bảng băm sử dụng chaining.
- <string>, <sstream>, <fstream>: xử lý chuỗi và đọc dữ liệu từ tập tin.
- <chrono>: đo thời gian thực thi.
- <windows.h>, <psapi.h>: đo mức sử dụng bộ nhớ hệ thống.

2.2. Các thao tác chính

Khởi tạo giá trị khóa

Cấu trúc Value dùng để lưu trữ mỗi phần tử trong bảng băm, gồm khóa (key) và nội dung gốc (val).

Listing IV.1: Cấu trúc dữ liệu Value

```

1 struct Value {
2     string key;
3     string val;
4 };

```

Tính toán chỉ số băm theo Fibonacci

Hàm băm nhân khóa với phi rồi dịch phải để lấy n bit cao nhất, tương ứng với kích thước bảng băm 2^n .

Listing IV.2: Hàm băm Fibonacci Hashing

```

1     int hashFunction(const string& key) {
2         uint32_t k = 0;
3         for (int i = 0; i < key.size(); ++i) {
4             int val = static_cast<int>(key[i]);
5             if (val < 0 || val > 127) return -1;
6             k = k * 31 + val;
7         }
8         return ((k * FIBO_RATIO) >> (w - n));
9     }

```

Phân tích độ phức tạp:

- Với vòng lặp for:
 - Chạy k lần với $k = \text{key.size}()$
 - Mỗi bước thực hiện:
 - Chuyển ký tự thành số nguyên $\rightarrow O(1)$
 - Kiểm tra trong khoảng ASCII [0, 127] $\rightarrow O(1)$
 - Nhân và cộng một số nguyên 32-bit $\rightarrow O(1)$
 - ⇒ Ta xem k là hằng số vì độ dài khóa thường nhỏ (< 100) $\rightarrow O(1)$
- Tính chỉ số bằng nhân Fibonacci + dịch bit:
 - Phép nhân và dịch bit là $O(1)$.

Độ phức tạp:

- Thời gian: $O(1)$

- Không gian: $O(1)$

Xây dựng bảng băm từ tệp

Hàm `build()` đọc từng dòng từ tệp, trích khóa từ chuỗi và đưa vào bảng bằng hàm `insert()`.

Listing IV.3: Hàm build()

```
1 void build(const string& path) {
2     vector<Value> data;
3     readFile(path, data);
4     m = findTableSize((int)data.size());
5     table.assign(m, list<Value>());
6
7     for (const auto& cur : data) {
8         insert(cur);
9     }
10 }
```

Phân tích độ phức tạp:

- `readFile(path, data)`

- Đọc lần lượt từng dòng trong file văn bản, tách phần `key`, lưu vào `data`.
- Với n dòng, mỗi dòng dài $O(L)$ ký tự.
- \Rightarrow Tổng thời gian: $O(n \times L)$

- `findTableSize((int)data.size())`

- Tìm kích thước bảng phù hợp (là 2^k gần nhất)
- Độ phức tạp: $O(\log n)$

- `table.assign(m, list<Value>())`

- Cấp phát bộ nhớ cho bảng có m ô (trong đó $m \approx n$)
- Độ phức tạp: $O(n)$
- Vòng lặp `for (const auto& cur : data) insert(cur)`
 - Gọi n lần hàm `insert()` với mỗi phần tử.
 - Trường hợp trung bình (không cần rehashing): mỗi lần `insert()` tốn $O(1)$
 - Vì kích thước bảng đã được tính trước phù hợp nên không xảy ra `reHashing()`.

Độ phức tạp:

- Thời gian: $O(n \times L + \log n + n) = O(n \times L)$ (do L thường lớn hơn $\log n$)
- Không gian: $O(n)$ (do lưu n phần tử vào bảng băm)

Tìm kích thước bảng băm phù hợp

Hàm `findTableSize()` dùng để tìm kích thước bảng băm tối thiểu lớn hơn kích thước dữ liệu đầu vào. Kích thước này được chọn là lũy thừa gần nhất của 2 để phù hợp với công thức dịch bit trong Fibonacci Hashing.

Listing IV.4: Hàm `findTableSize()`

```

1  size_t findTableSize(int size) {
2      size_t res = 1;
3      n = 0;
4      while (res <= size) {
5          res = res << 1;
6          n++;
7      }
8      return res;
9 }
```

Phân tích độ phức tạp:

Tìm số lần dịch trái để `res` vượt `size`:

- Ban đầu `res = 1`

- Sau 1 lần: `res = 2`
- Sau 2 lần: `res = 4`
- ...
- Sau k lần: `res = 2^k > size`

$\Rightarrow 2^k > \text{size} \Rightarrow k > \log_2(\text{size}) \Rightarrow$ Số lần lặp của while là $\log_2(\text{size}) + 1$

Độ phức tạp:

- Thời gian: $O(\log n)$
- Không gian: $O(1)$

Chèn phần tử không kiểm tra tái băm

Hàm `rawInsert()` là hàm phụ trợ được sử dụng trong quá trình tái băm (`reHashing()`), với nhiệm vụ thêm trực tiếp phần tử vào bảng băm mới mà không cần kiểm tra xem khóa đã tồn tại hay chưa. Điều này hợp lý vì các phần tử được lấy từ bảng băm cũ vốn đã đảm bảo duy nhất về khóa.

Listing IV.5: Hàm `rawInsert()`

```

1  bool rawInsert(const Value& data) {
2
3      int index = hashFunction(data.key);
4
5      if (index < 0 || index >= m) {
6
7          return false;
8
9      }
10
11     table[index].push_back(data);
12
13     count++;
14
15     return true;
16 }
```

Phân tích độ phức tạp:

- Tính chỉ số băm: $O(1)$
- Thêm phần tử vào danh sách liên kết: $O(1)$

Độ phức tạp:

- Thời gian: $O(1)$

- Không gian: $O(1)$

Thêm phần tử vào bảng băm

Hàm `insert()` tính chỉ số băm, kiểm tra trùng khóa trong danh sách và thêm vào bucket tương ứng.

Listing IV.6: Hàm `insert()`

```
1  bool insert(const Value& data) {
2
3      if (loadFactor() >= 1.0) {
4
5          reHashing();
6
7      }
8
9      int index = hashFunction(data.key);
10
11     if (index < 0 || index >= m) {
12
13         return false;
14     }
15
16     for (auto& cur : table[index]) {
17
18         if (cur.key == data.key) {
19
20             cur.val = data.val;
21
22             return true;
23         }
24
25     }
26
27     table[index].push_back(data);
28
29     count++;
30
31     return true;
32 }
```

Phân tích độ phức tạp:

- `loadFactor()` và `reHashing()`:

- `loadFactor()` là phép chia đơn giản: $O(1)$

- Nhưng nếu điều kiện đúng, thì `reHashing()` sẽ được gọi và có độ phức tạp: $O(n)$
- **Tính chỉ số băm:** $O(1)$
- **Tìm key trong bucket:**
 - Trong trường hợp xấu nhất, phải duyệt toàn bộ danh sách liên kết trong `table[index]`.
 - Gọi h là số phần tử trong bucket đó, thì vòng lặp này là: $O(h)$
 - Nhưng vì hàm băm Fibonacci giúp phân bố các khóa đều và có hàm `reHashing()` giúp duy trì `loadFactor() \leq 1.0` nên h rất nhỏ, xem h là hằng số $\Rightarrow O(1)$
- **Thêm phần tử vào danh sách liên kết:** $O(1)$

Độ phức tạp:

- **Thời gian:**

Trường hợp	Độ phức tạp
Trung bình	$O(1)$
Xấu nhất (gặp tái băm)	$O(n)$

- **Không gian:**

Trường hợp	Độ phức tạp
Trung bình	$O(1)$
Xấu nhất (gặp tái băm)	$O(n)$

Tìm kiếm khóa trong bảng băm

Hàm `search()` duyệt qua danh sách tại chỉ số băm để tìm phần tử có khóa trùng khớp.

Listing IV.7: Hàm `search()`

```

1  string search(const string& key) {
2      int index = hashFunction(key);
3      if (index < 0 || table[index].empty()) {
4          return "Not Found";
5      }

```

```

6     auto it = table[index].begin();
7     while (it != table[index].end() && it->key != key) {
8         it++;
9     }
10    if (it == table[index].end()) {
11        return "Not Found";
12    }
13    else {
14        return it->val;
15    }
16 }
```

Phân tích độ phức tạp:

- **Tính chỉ số băm:** $O(1)$
- **Kiểm tra chỉ số:** $O(1)$
- **Duyệt tìm key trong bucket:**
 - Trong trường hợp xấu nhất, phải duyệt toàn bộ danh sách liên kết trong `table[index]`.
 - Gọi h là số phần tử trong bucket đó, thì vòng lặp này có độ phức tạp: $O(h)$
 - Tuy nhiên, vì hàm băm Fibonacci giúp phân bố các khóa đều và có hàm `reHashing()` giúp duy trì `loadFactor() ≤ 1.0` nên h rất nhỏ, xem h là hằng số $\Rightarrow O(1)$

Độ phức tạp:

- Thời gian: $O(1)$
- Không gian: $O(1)$

Xoá phần tử theo khóa

Hàm `Delete()` duyệt danh sách trong bucket và xoá phần tử nếu tìm thấy khóa.

Listing IV.8: Hàm delete()

```
1  bool Delete(const string& key) {
2      int index = hashFunction(key);
3      if (index < 0 || table[index].empty()) {
4          return false;
5      }
6      auto it = table[index].begin();
7      while (it != table[index].end() && it->key != key) {
8          it++;
9      }
10     if (it == table[index].end()) {
11         return false;
12     }
13     else {
14         table[index].erase(it);
15         count--;
16         return true;
17     }
18 }
```

Phân tích độ phức tạp:

- Tính chỉ số băm: $O(1)$

- Kiểm tra chỉ số: $O(1)$

- Duyệt tìm key trong bucket:

- Trong trường hợp xấu nhất, phải duyệt toàn bộ danh sách liên kết trong `table[index]`.
- Gọi h là số phần tử trong bucket đó, thì vòng lặp này có độ phức tạp: $O(h)$
- Nhưng vì hàm băm Fibonacci giúp phân bố các khóa đều và có hàm `reHashing()` giúp duy trì `loadFactor() ≤ 1.0` nên h rất nhỏ, xem h là hằng số $\Rightarrow O(1)$

- Xoá phần tử khỏi danh sách liên kết: $O(1)$

Độ phức tạp:

- Thời gian: $O(1)$

- Không gian: $O(1)$

Tái băm khi bảng đầy

Hàm `reHashing()` tăng số bit n , mở rộng bảng băm, tính lại chỉ số và chuyển toàn bộ dữ liệu cũ sang bảng mới.

Listing IV.9: Hàm `reHashing()`

```
1 void reHashing() {
2     m = m << 1;
3     n++;
4     count = 0;
5     vector<list<Value>>oldTable = table;
6     table.clear();
7     table.assign(m, list<Value>());
8     for (auto& bucket : oldTable) {
9         for (auto& val : bucket) {
10             rawInsert(val);
11         }
12     }
13     oldTable.clear();
14     oldTable.shrink_to_fit();
15 }
```

Phân tích độ phức tạp:

- Tăng các chỉ số của bảng: $O(1)$
- Lưu lại và reset bảng: $O(n)$
- Duyệt lại toàn bộ bảng cũ:
 - Mỗi phần tử được băm lại và chèn vào bảng mới.

- Gọi hàm `rawInsert(val)` cho từng phần tử.
 - `rawInsert(val)` thực hiện phép chèn đơn giản vào bảng $\Rightarrow O(1)$.
 - Tổng số phần tử là n , mỗi phần tử xử lý $O(1)$.
- \Rightarrow Tổng thời gian: $O(n)$

Độ phức tạp:

- Thời gian: $O(n)$
- Không gian: $O(n)$ (do cần bộ nhớ tạm để lưu bảng băm cũ)

3. Kết quả đo đạc

Qua các đầu vào, ta có được kết quả của hàm băm Fibonacci cùng với hàm băm modulo truyền thống.

3.1. Thực nghiệm Test1

Bảng so sánh hiệu suất của dữ liệu Test1 ở 3 dạng clustered, random và sequential:

Test1_Clustered									
Percent	Method	Build (ms)	Memory (MB)	Load Factor	Avg Chain	Max Chain	Search Time	Delete Time	Insert Time
10%	Fibonacci	65	5	0.370117	1.19511	4	40	91	419
	Modulo	68	4	0.370117	1.18576	4	34	349	257
20%	Fibonacci	136	9	0.384033	1.21679	4	139	264	351
	Modulo	135	8	0.384033	1.20559	4	54	343	256
30%	Fibonacci	198	12	0.580322	1.3283	5	181	98	405
	Modulo	199	11	0.580322	1.31054	5	56	160	257
40%	Fibonacci	262	15	0.786743	1.44783	6	95	178	320
	Modulo	262	13	0.786743	1.4399	6	55	177	347

50%	Fibonacci	336	21	0.496155	1.27165	5	72	192	292
	Modulo	339	19	0.496155	1.2607	5	55	217	292
60%	Fibonacci	395	24	0.600037	1.3332	7	41	100	290
	Modulo	399	21	0.600037	1.32538	6	37	156	312
70%	Fibonacci	467	27	0.697815	1.39198	7	149	209	334
	Modulo	479	24	0.697815	1.38843	7	54	186	257
80%	Fibonacci	509	29	0.789551	1.45161	7	138	270	305
	Modulo	518	25	0.789551	1.44278	7	103	212	364
90%	Fibonacci	575	41	0.437363	1.2351	6	105	290	338
	Modulo	561	33	0.437363	1.23627	6	54	217	260
100%	Fibonacci	598	43	0.479507	1.26119	6	191	288	307
	Modulo	608	35	0.479507	1.26104	6	93	344	273

Bảng IV.1: Bảng số liệu Test1 dạng Clustered

Test1_Random									
Percent	Method	Build (ms)	Memory (MB)	Load Factor	Avg Chain	Max Chain	Search Time	Delete Time	Insert Time
10%	Fibonacci	59	4	0.859619	1.48503	5	196	380	466
	Modulo	65	3	0.859619	1.47199	5	76	365	468
20%	Fibonacci	118	7	0.839355	1.46704	5	150	268	354
	Modulo	128	6	0.839355	1.4749	6	78	211	253
30%	Fibonacci	179	12	0.619324	1.33742	6	57	139	374
	Modulo	179	10	0.619324	1.34326	7	76	434	172
40%	Fibonacci	237	15	0.813171	1.45766	6	212	335	479
	Modulo	238	13	0.813171	1.46246	7	110	448	542
50%	Fibonacci	314	22	0.50296	1.27473	5	109	332	383
	Modulo	315	19	0.50296	1.26982	6	57	245	418
60%	Fibonacci	390	24	0.596527	1.32738	5	141	219	214
	Modulo	366	21	0.596527	1.32468	6	57	164	153

70%	Fibonacci	413	26	0.688416	1.38827	7	161	314	300
	Modulo	412	23	0.688416	1.37843	6	57	209	257
80%	Fibonacci	483	28	0.778809	1.44712	7	196	235	296
	Modulo	491	25	0.778809	1.43548	6	59	210	257
90%	Fibonacci	548	41	0.434464	1.23565	6	128	362	254
	Modulo	550	33	0.434464	1.23484	5	57	219	254
100%	Fibonacci	598	43	0.479507	1.26119	6	147	349	471
	Modulo	602	35	0.479507	1.26104	6	74	293	324

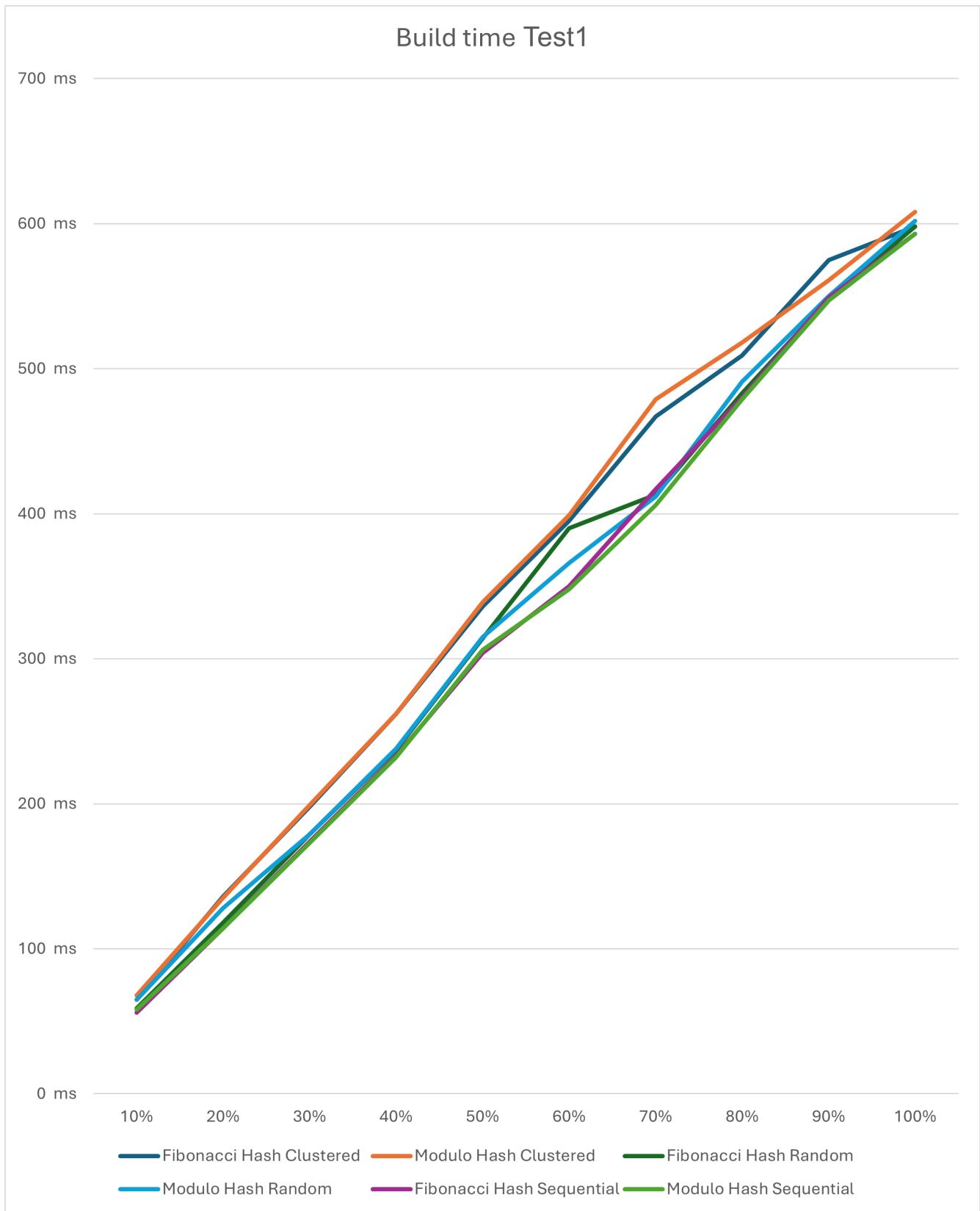
Bảng IV.2: Bảng số liệu Test1 dạng Random

Test1_Sequential									
Percent	Method	Build (ms)	Memory (MB)	Load Factor	Avg Chain	Max Chain	Search Time	Delete Time	Insert Time
10%	Fibonacci	56	4	0.753418	1.41365	6	228	371	523
	Modulo	58	3	0.753418	1.44611	5	71	294	368
20%	Fibonacci	114	7	0.768188	1.4361	6	46	103	213
	Modulo	114	6	0.768188	1.44501	6	28	96	150
30%	Fibonacci	174	12	0.582642	1.31216	6	137	340	308
	Modulo	173	11	0.582642	1.32602	5	56	222	284
40%	Fibonacci	233	14	0.755188	1.43173	7	140	307	510
	Modulo	232	13	0.755188	1.43256	7	55	215	281
50%	Fibonacci	304	22	0.466217	1.26121	5	106	244	528
	Modulo	306	17	0.466217	1.24659	5	54	235	167
60%	Fibonacci	350	24	0.560669	1.31304	6	80	215	355
	Modulo	348	18	0.560669	1.30279	5	72	181	258
70%	Fibonacci	417	25	0.659576	1.37208	6	216	360	382
	Modulo	406	22	0.659576	1.36214	6	56	209	260
80%	Fibonacci	480	28	0.759277	1.43235	7	331	458	496
	Modulo	479	25	0.759277	1.42424	6	71	262	317

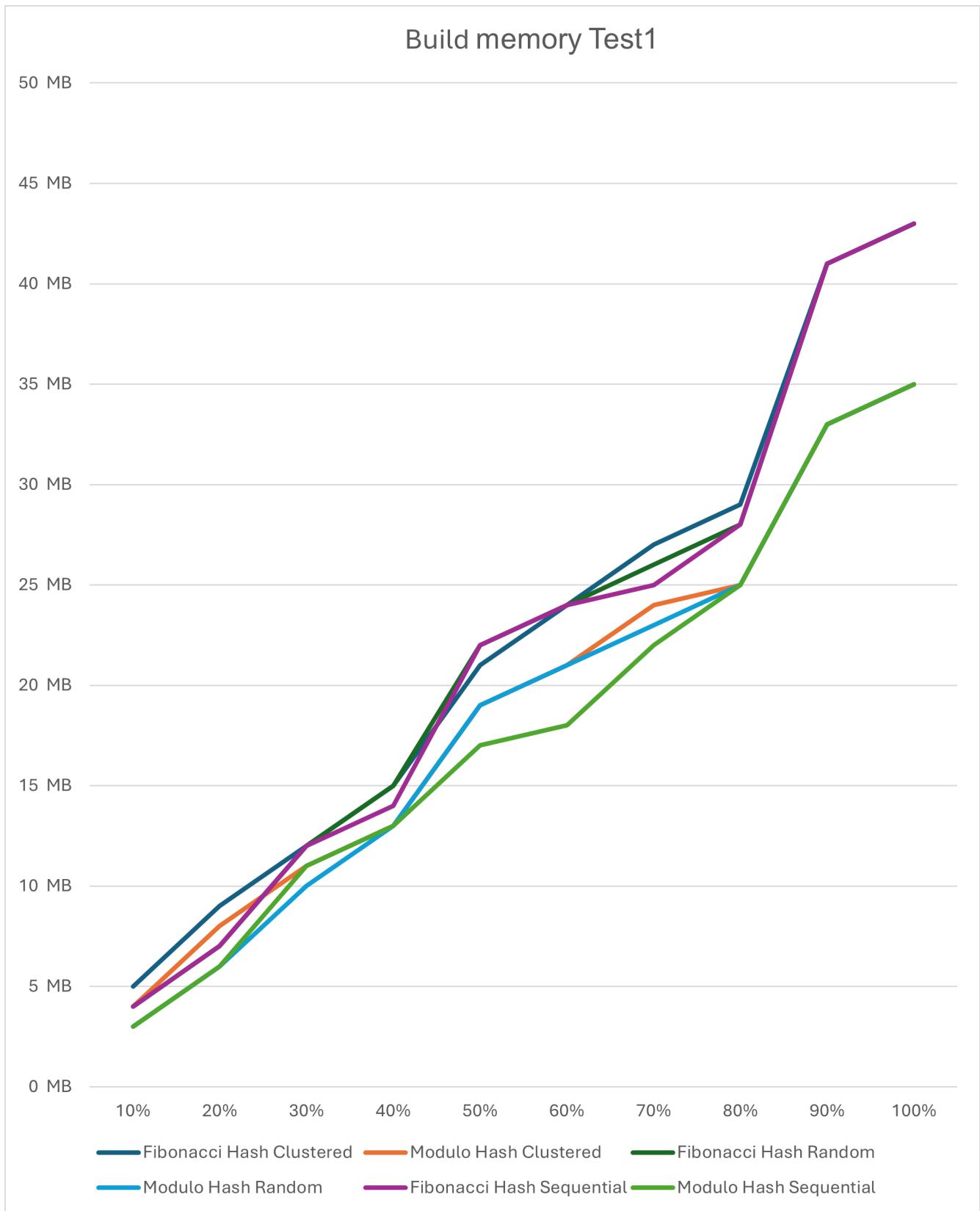
90%	Fibonacci	549	41	0.428299	1.23131	6	146	270	327
	Modulo	547	33	0.428299	1.23088	5	54	208	260
100%	Fibonacci	593	43	0.479507	1.26119	6	175	246	353
	Modulo	593	35	0.479507	1.26104	6	57	210	261

Bảng IV.3: Bảng số liệu Test1 dạng Sequential

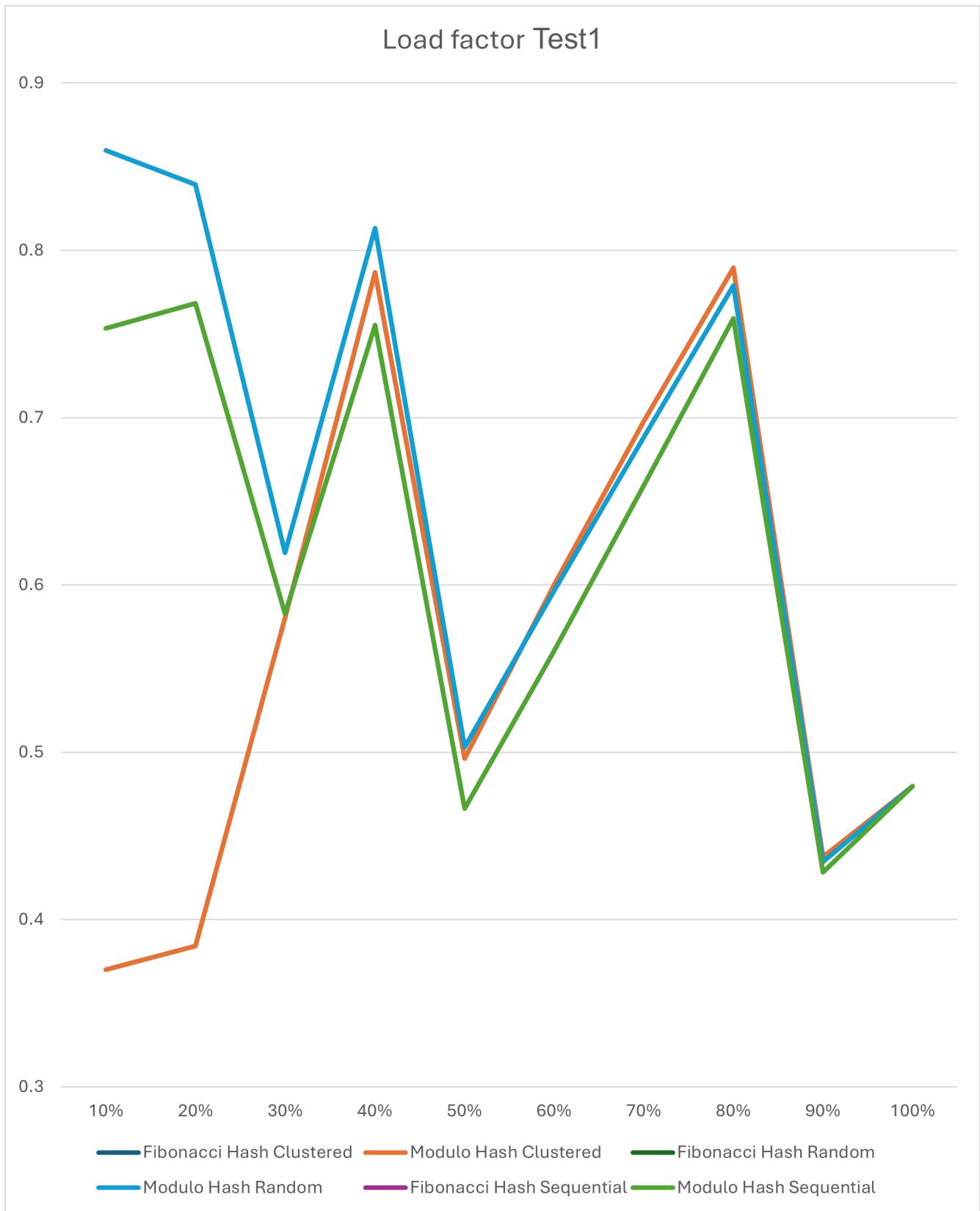
Đồng thời ta có được biểu đồ như sau:



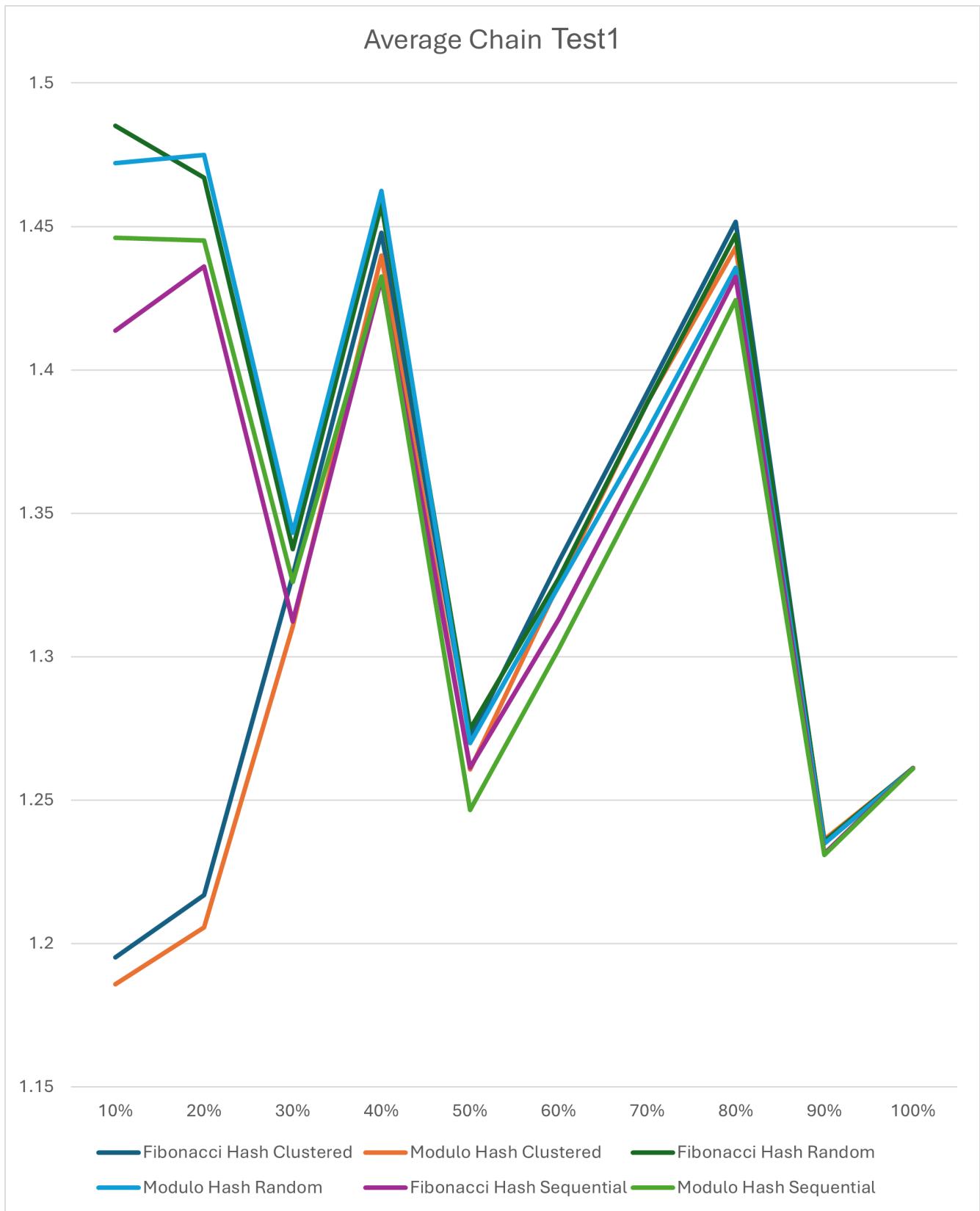
Hình IV.1: Thời gian Build Test1



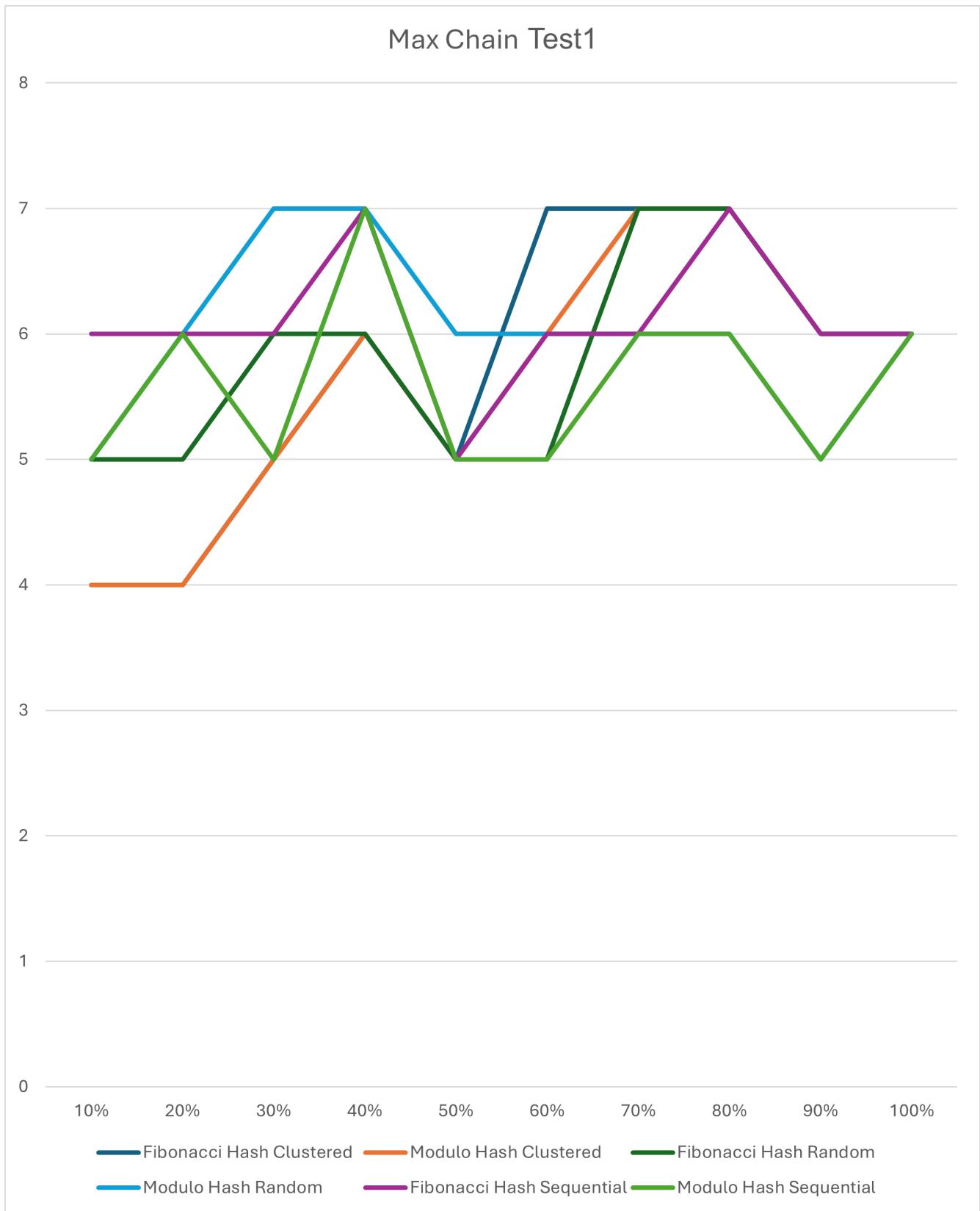
Hình IV.2: Bộ nhớ tiêu tốn để Build Test1



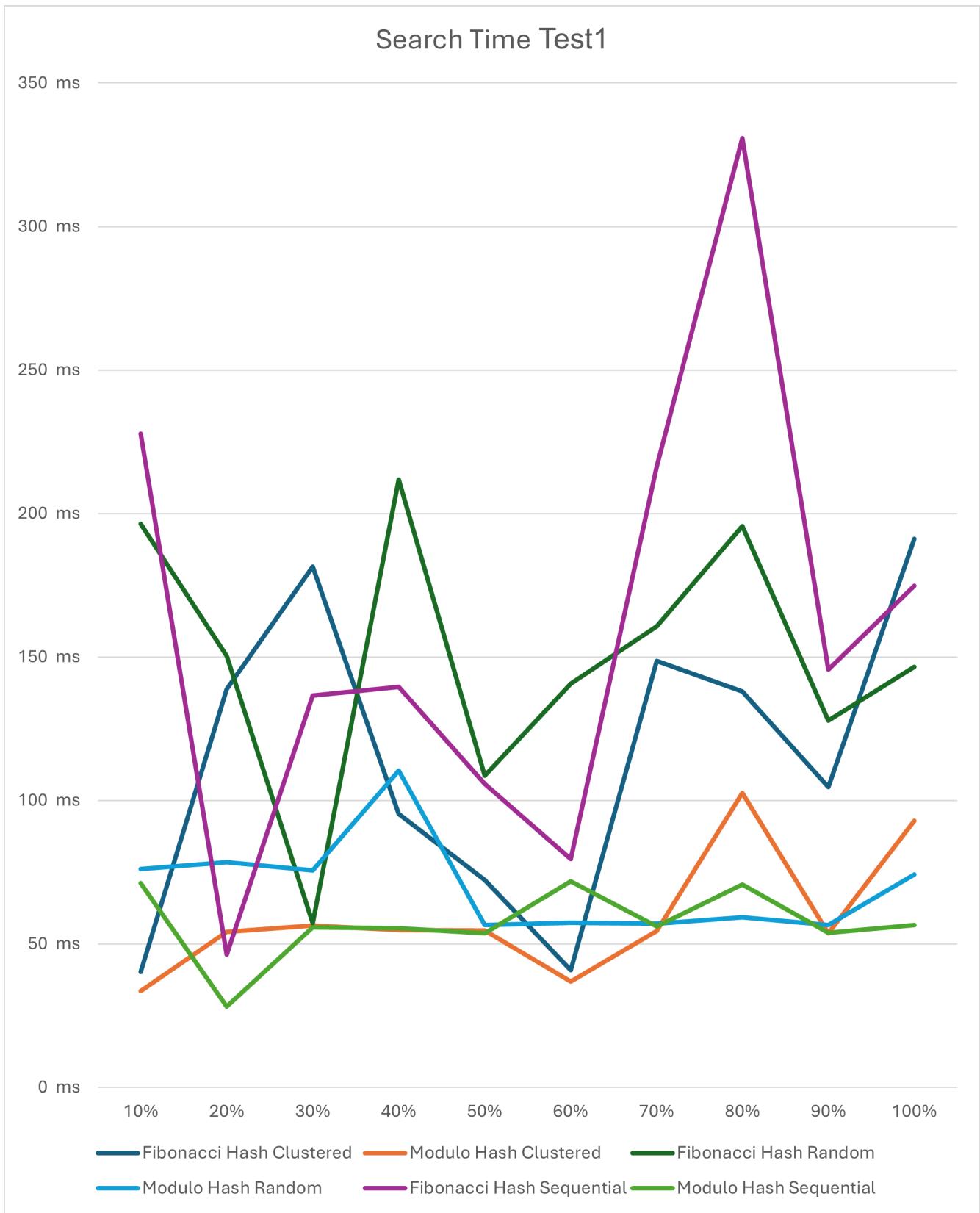
Hình IV.3: Load Factor có ở Test1



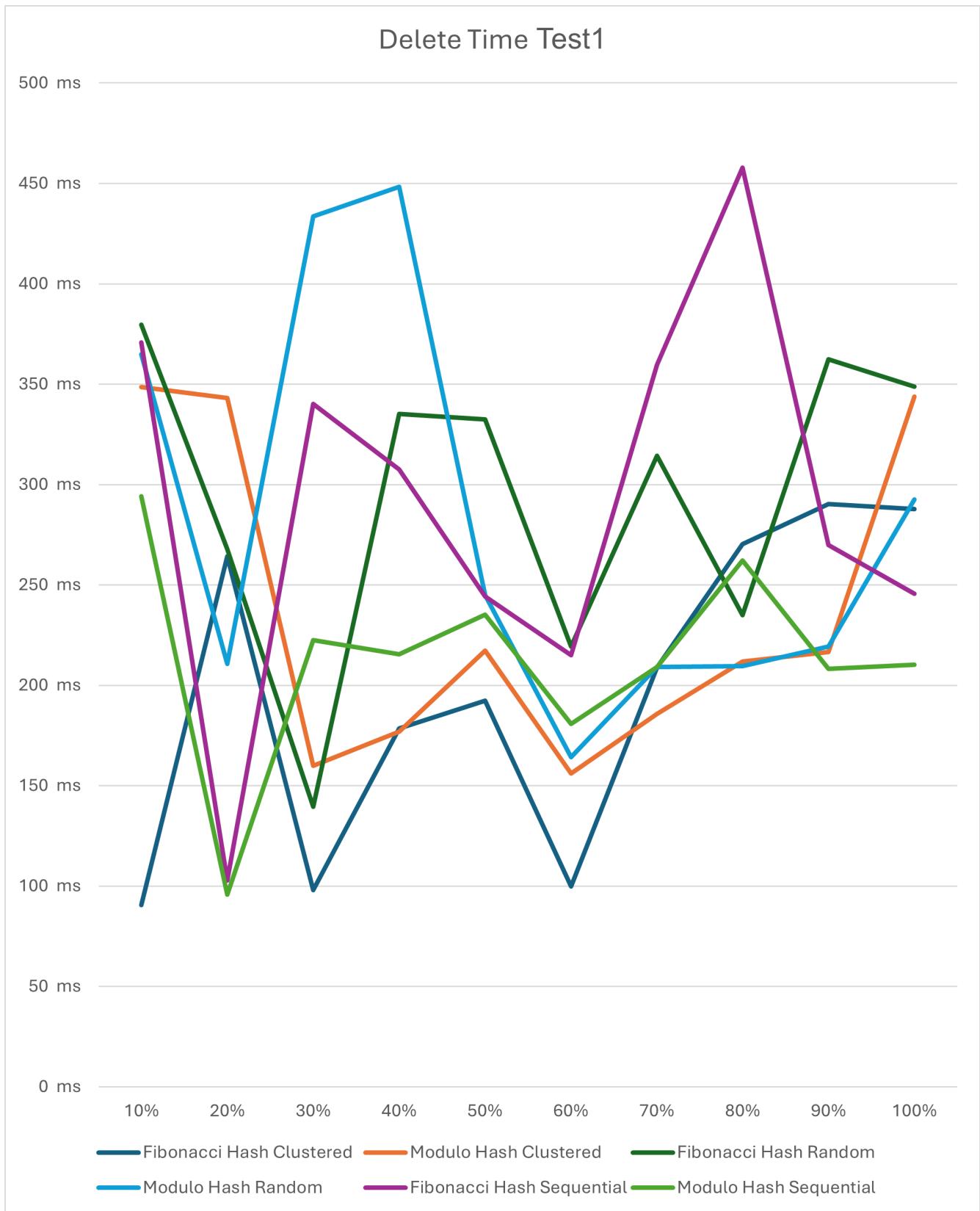
Hình IV.4: Average Chain có ở Test1



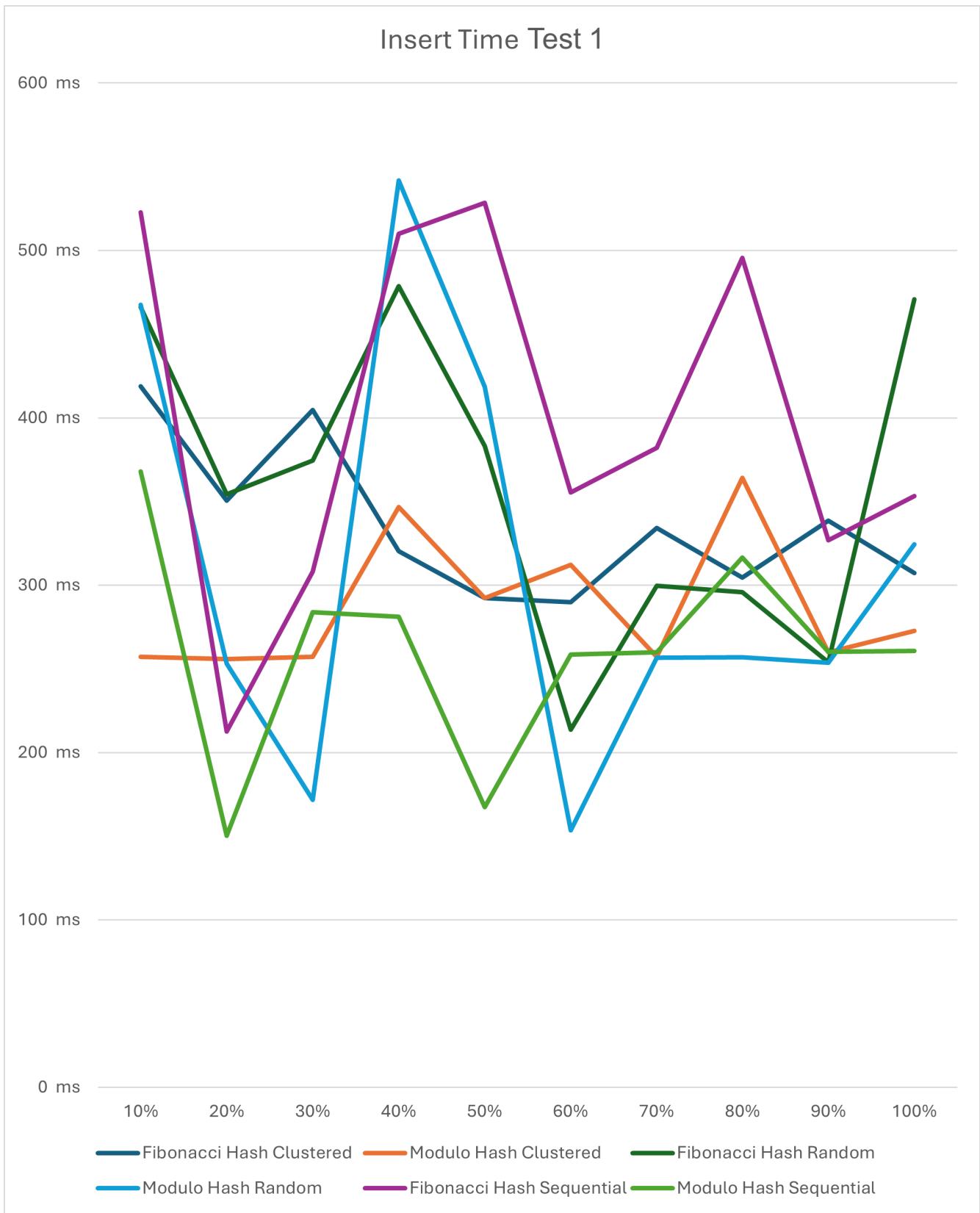
Hình IV.5: Max Chain có ở Test1



Hình IV.6: Thời gian Search ở Test1



Hình IV.7: Thời gian Delete ở Test1



Hình IV.8: Thời gian Insert ở Test1

3.2. Thực nghiệm Test2

Bảng so sánh hiệu suất của dữ liệu Test1 ở 3 dạng clustered, random và sequential:

Test2_Clustered									
Percent	Method	Build (ms)	Memory (MB)	Load Factor	Avg Chain	Max Chain	Search Time	Delete Time	Insert Time
10%	Fibonacci	338	37	0.604279	1.33412	6	349	794	1107
	Modulo	336	31	0.604279	1.33439	6	152	597	968
20%	Fibonacci	690	71	0.603523	1.3365	6	410	529	1146
	Modulo	683	60	0.603523	1.33229	6	164	293	1093
30%	Fibonacci	996	84	0.903938	1.52215	7	419	685	200
	Modulo	983	79	0.903938	1.51778	7	149	505	680
40%	Fibonacci	1398	142	0.60162	1.33084	7	443	974	1303
	Modulo	1388	117	0.60162	1.33105	7	284	667	1209
50%	Fibonacci	1620	142	0.750481	1.4212	8	263	832	409
	Modulo	1678	130	0.750481	1.42157	8	212	255	316
60%	Fibonacci	2027	179	0.898148	1.51529	8	140	458	411
	Modulo	1957	122	0.898148	1.51476	8	117	249	321
70%	Fibonacci	2374	244	0.522001	1.2836	6	242	925	946
	Modulo	2403	234	0.522001	1.28306	6	208	734	830
80%	Fibonacci	2531	245	0.593081	1.32545	6	113	631	311
	Modulo	2541	235	0.593081	1.3256	7	105	383	246
90%	Fibonacci	2983	254	0.658882	1.36382	7	380	1016	528
	Modulo	2978	287	0.658882	1.36505	7	157	1181	381
100%	Fibonacci	3108	261	0.705055	1.39217	8	348	708	316
	Modulo	3171	299	0.705055	1.39345	7	165	522	297

Bảng IV.4: Bảng số liệu Test2 dạng Clustered

Test2_Random

Percent	Method	Build (ms)	Memory (MB)	Load Factor	Avg Chain	Max Chain	Search Time	Delete Time	Insert Time
10%	Fibonacci	321	34	0.564056	1.31085	6	328	454	904
	Modulo	324	29	0.564056	1.3064	6	236	346	954
20%	Fibonacci	627	68	0.564056	1.30814	6	282	712	975
	Modulo	632	57	0.564056	1.30846	6	259	765	702
30%	Fibonacci	928	86	0.846085	1.48339	7	332	986	441
	Modulo	893	68	0.846085	1.48166	7	159	897	354
40%	Fibonacci	1323	136	0.564056	1.30748	7	198	460	378
	Modulo	1306	113	0.564056	1.30829	6	116	298	428
50%	Fibonacci	1512	142	0.70507	1.3943	7	387	753	1012
	Modulo	1471	120	0.70507	1.39262	7	135	591	693
60%	Fibonacci	1909	171	0.846085	1.48289	7	405	416	372
	Modulo	1920	130	0.846085	1.47962	7	284	248	352
70%	Fibonacci	2128	177	0.987099	1.57378	8	459	1231	975
	Modulo	2088	151	0.987099	1.57119	8	387	792	767
80%	Fibonacci	2516	245	0.564056	1.30695	7	461	901	931
	Modulo	2412	249	0.564056	1.30843	7	251	594	786
90%	Fibonacci	2868	242	0.634563	1.3492	7	236	1071	850
	Modulo	2876	251	0.634563	1.35053	7	150	584	927
100%	Fibonacci	3130	254	0.705055	1.39217	8	219	945	1130
	Modulo	3081	267	0.705055	1.39345	7	141	679	701

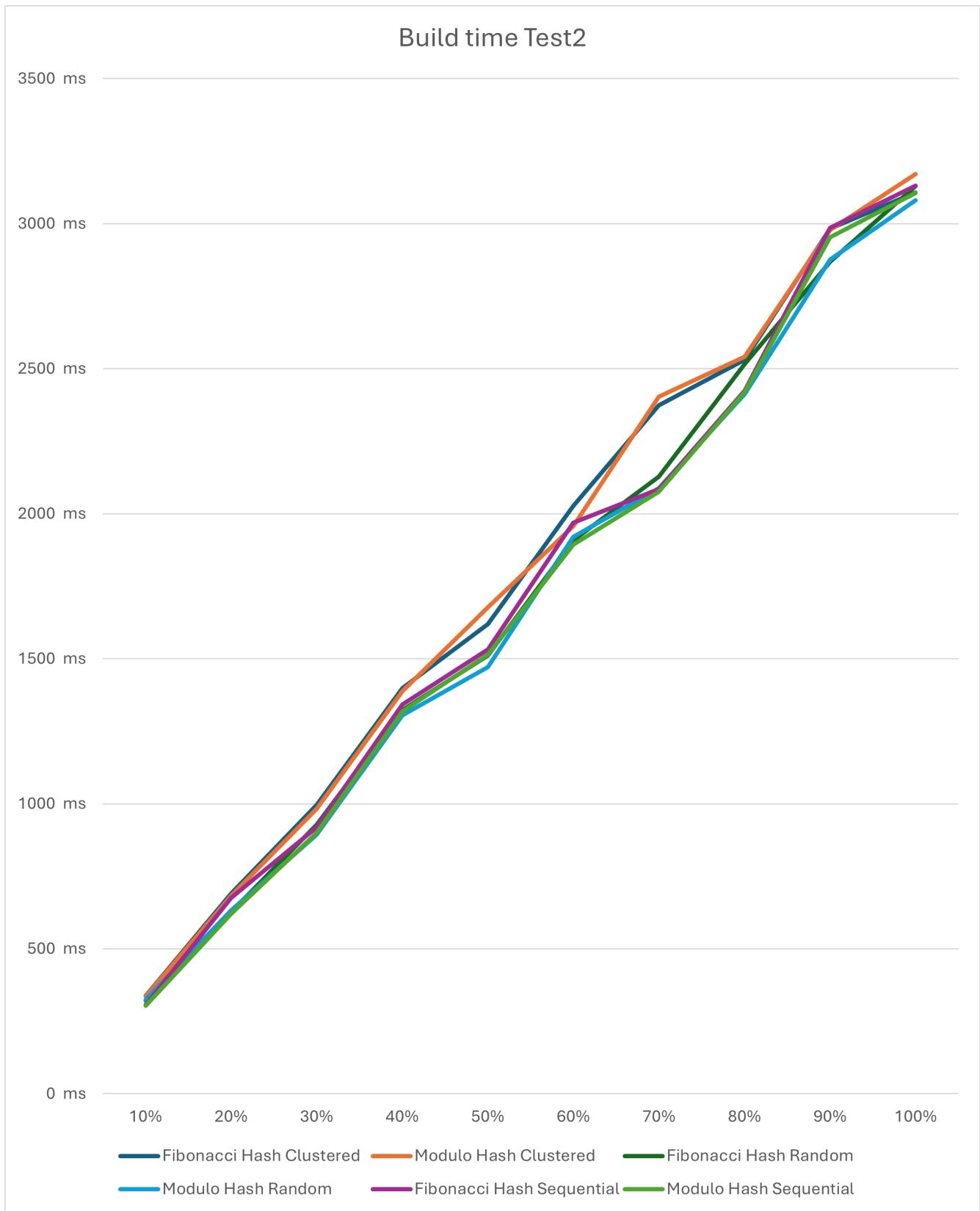
Bảng IV.5: Bảng số liệu Test2 dạng Random

Test2_Sequential									
Percent	Method	Build (ms)	Memory (MB)	Load Factor	Avg Chain	Max Chain	Search Time	Delete Time	Insert Time
10%	Fibonacci	306	33	0.564148	1.3074	6	86	564	417
	Modulo	303	29	0.564148	1.31423	6	82	331	360

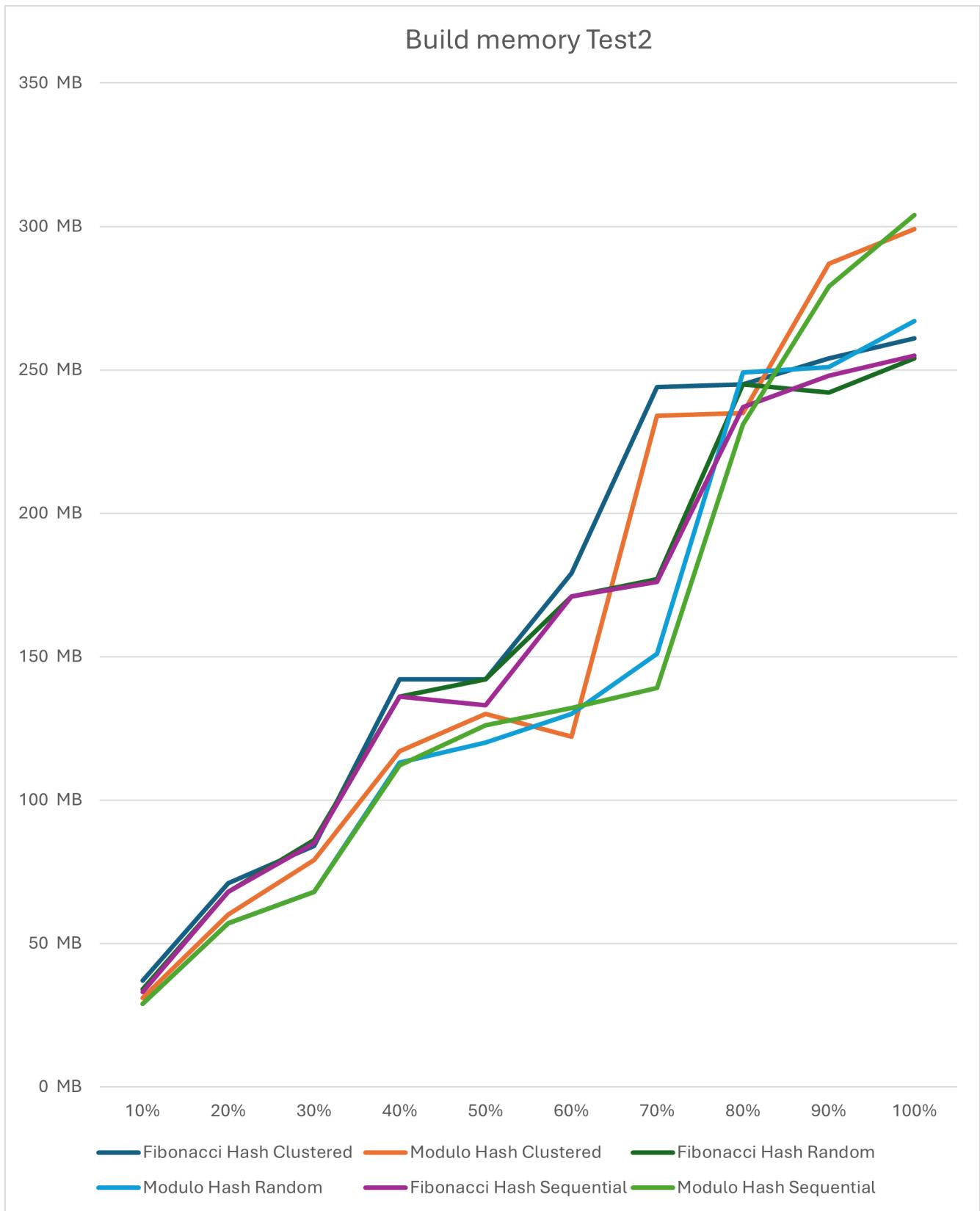
20%	Fibonacci Modulo	676 621	68 57	0.564255 0.564255	1.3064 1.31099	7 6	136 88	300 257	1114 901
30%	Fibonacci Modulo	918 899	85 68	0.84623 0.84623	1.47759 1.48356	8 7	230 274	1064 718	1018 705
40%	Fibonacci Modulo	1343 1319	136 112	0.564041 0.564041	1.30835 1.30974	6 6	459 311	1001 515	1361 985
50%	Fibonacci Modulo	1532 1515	133 126	0.704975 0.704975	1.39363 1.39489	8 6	137 120	859 686	885 885
60%	Fibonacci Modulo	1969 1894	171 132	0.845959 0.845959	1.48052 1.48245	8 7	72 60	593 667	655 593
70%	Fibonacci Modulo	2085 2076	176 139	0.987019 0.987019	1.57187 1.57315	8 8	272 154	990 978	954 737
80%	Fibonacci Modulo	2424 2420	237 231	0.564047 0.564047	1.30717 1.30831	7 7	461 356	299 293	416 398
90%	Fibonacci Modulo	2986 2953	248 279	0.634592 0.634592	1.3492 1.35066	8 7	193 103	824 788	1438 851
100%	Fibonacci Modulo	3131 3106	255 304	0.705055 0.705055	1.39217 1.39345	8 7	326 554	1157 808	959 581

Bảng IV.6: Bảng số liệu Test2 dạng Sequential

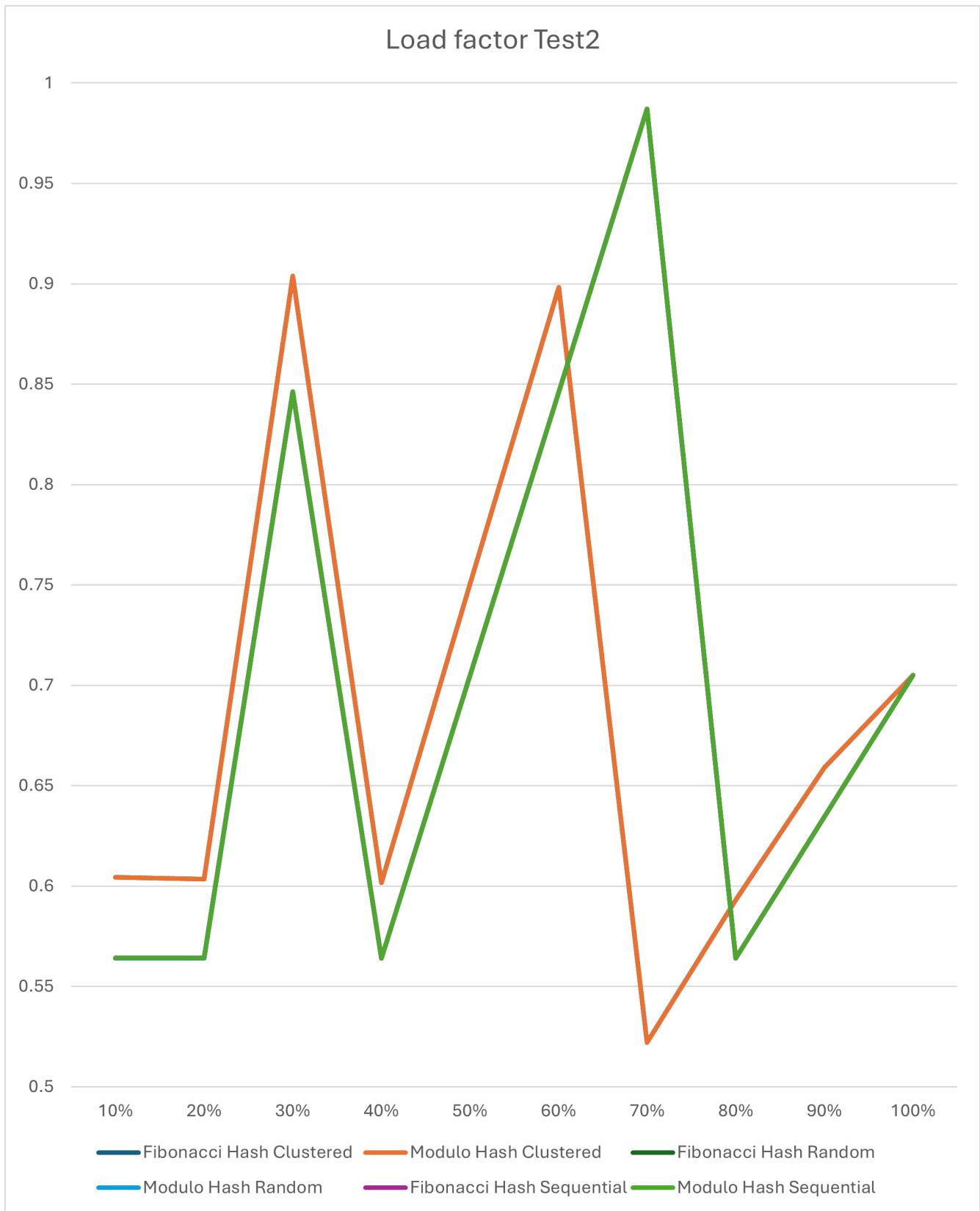
Đồng thời ta có được biểu đồ như sau:



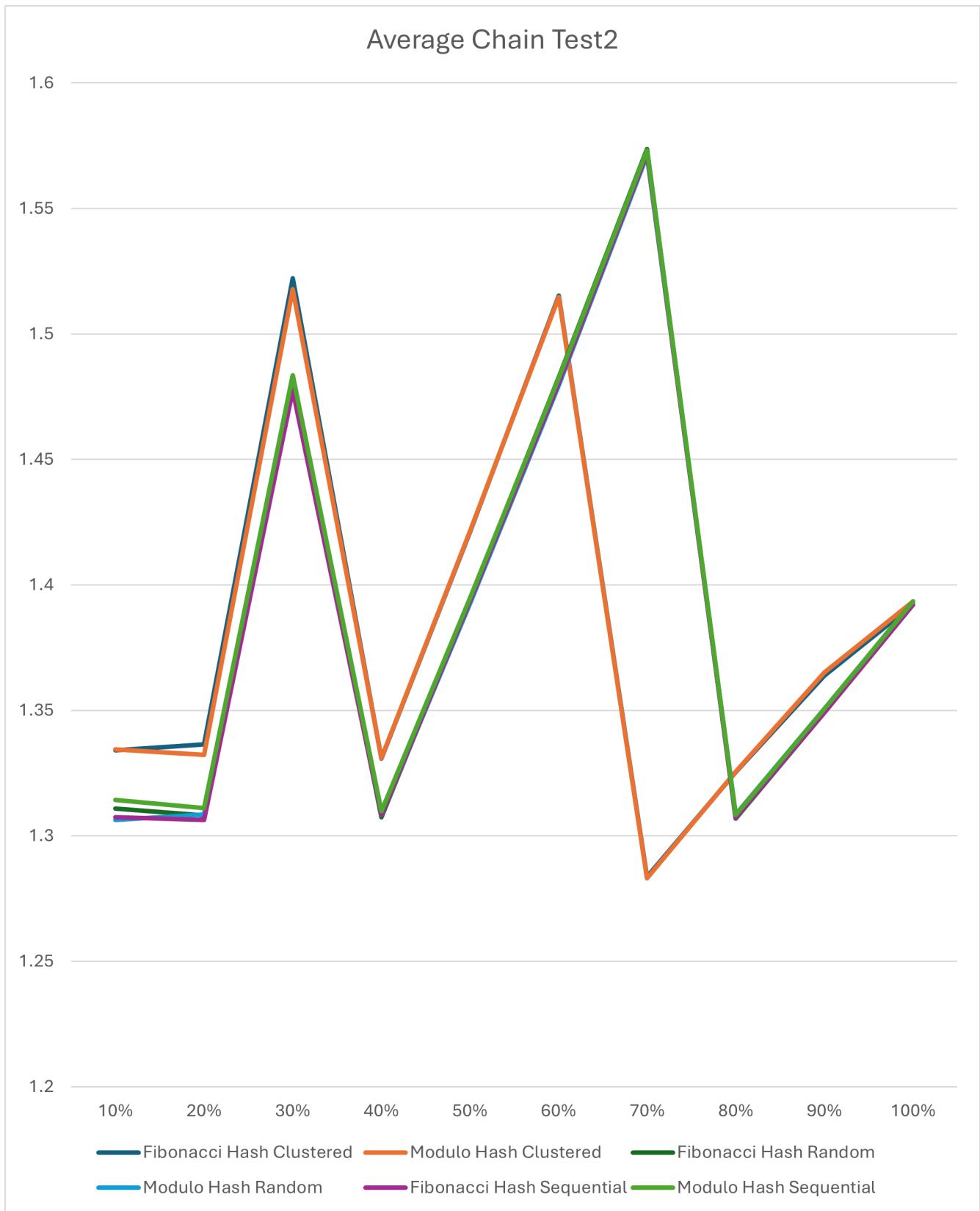
Hình IV.9: Thời gian Build Test2



Hình IV.10: Bộ nhớ tiêu tốn để Build Test2



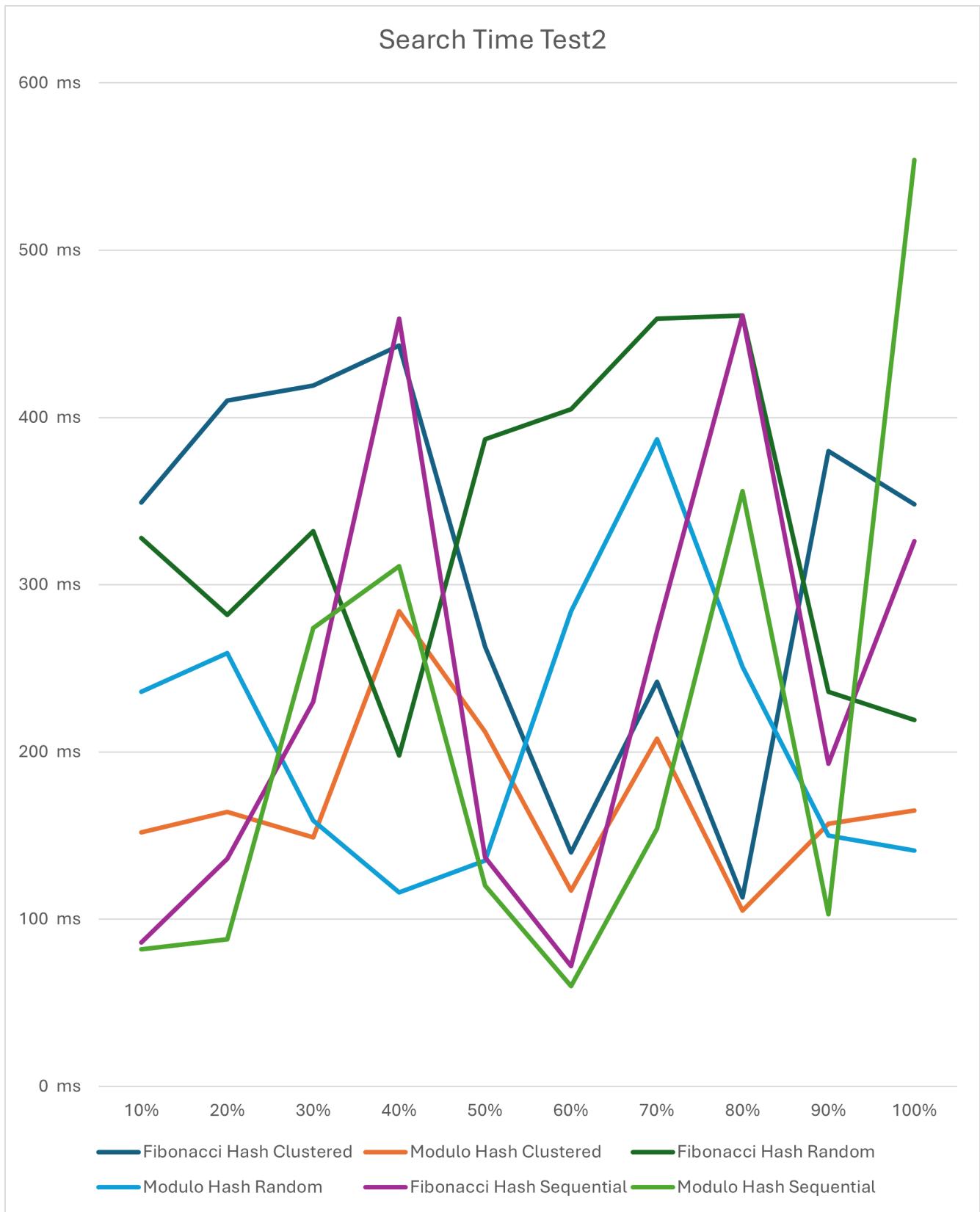
Hình IV.11: Load Factor có ở Test2



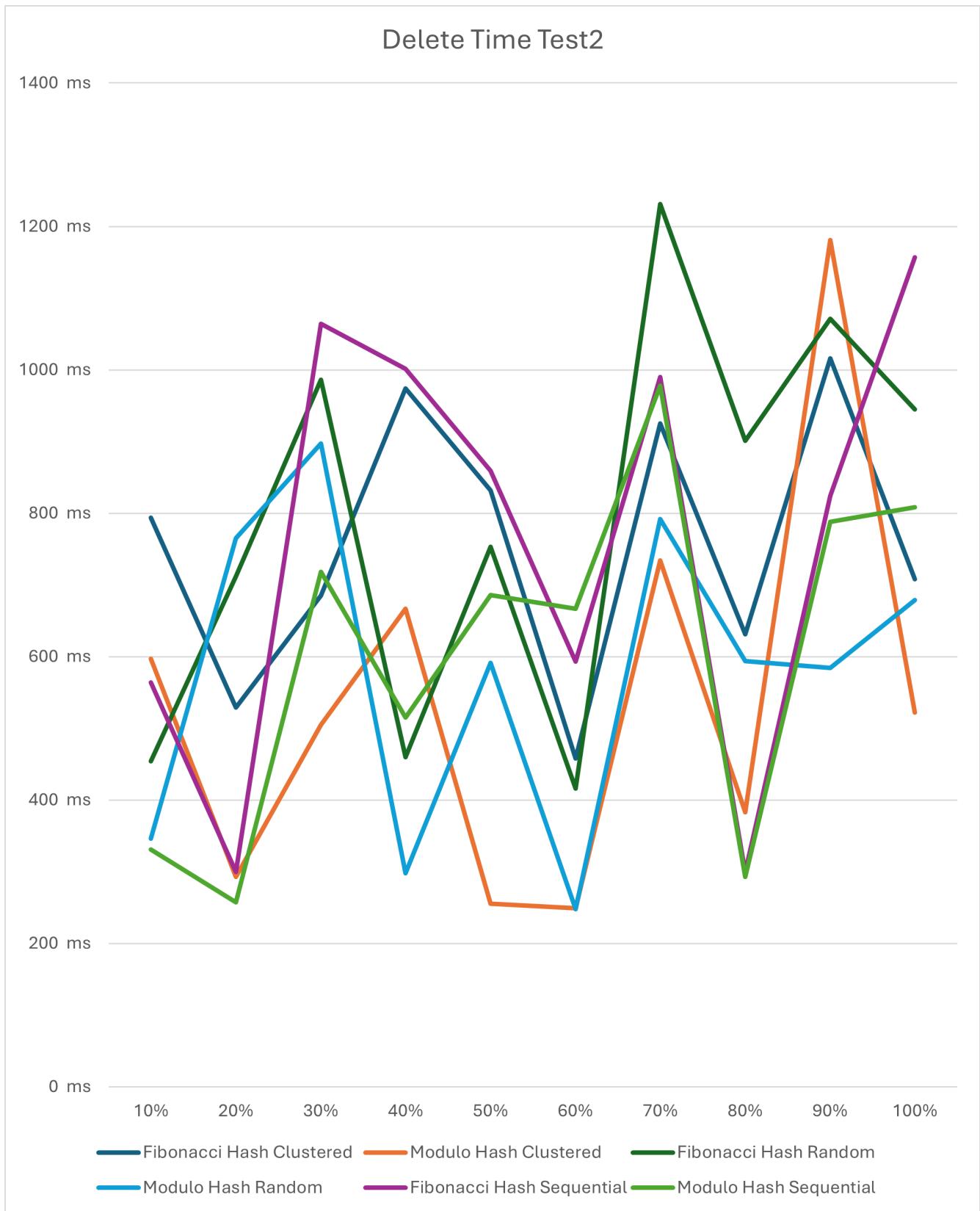
Hình IV.12: Average Chain có ở Test2



Hình IV.13: Max Chain có ở Test2



Hình IV.14: Thời gian Search ở Test2



Hình IV.15: Thời gian Delete ở Test2



Hình IV.16: Thời gian Insert ở Test2

3.3. Thực nghiệm Test3

Bảng so sánh hiệu suất của dữ liệu Test1 ở 3 dạng clustered, random và sequential:

Test3_Clustered									
Percent	Method	Build (ms)	Memory (MB)	Load Factor	Avg Chain	Max Chain	Search Time	Delete Time	Insert Time
10%	Fibonacci	42	5	0.550781	1.31087	5	77	529	680
	Modulo	35	4	0.550781	1.31507	5	59	351	294
20%	Fibonacci	74	8	0.553101	1.29568	5	61	762	340
	Modulo	75	8	0.553101	1.31011	5	48	218	3639
30%	Fibonacci	112	11	0.831726	1.46212	6	100	629	678
	Modulo	110	8	0.831726	1.48023	7	58	241	998
40%	Fibonacci	158	17	0.552704	1.29522	6	137	154	545
	Modulo	138	14	0.552704	1.31287	5	117	219	225
50%	Fibonacci	187	18	0.690491	1.38073	7	46	541	1897
	Modulo	189	17	0.690491	1.39383	6	33	264	395
60%	Fibonacci	228	19	0.828217	1.46349	7	127	466	1095
	Modulo	211	18	0.828217	1.47953	6	137	598	753
70%	Fibonacci	262	21	0.965179	1.55027	7	371	952	1149
	Modulo	253	20	0.965179	1.56927	7	146	589	922
80%	Fibonacci	294	31	0.551987	1.29972	6	435	815	467
	Modulo	315	31	0.551987	1.30431	6	159	825	290
90%	Fibonacci	320	33	0.621323	1.34289	6	218	790	1076
	Modulo	313	31	0.621323	1.34244	6	150	580	570
100%	Fibonacci	377	35	0.68959	1.3862	6	425	628	1488
	Modulo	395	34	0.68959	1.38374	6	377	658	747

Bảng IV.7: Bảng số liệu Test3 dạng Clustered

Test3_Random

Percent	Method	Build (ms)	Memory (MB)	Load Factor	Avg Chain	Max Chain	Search Time	Delete Time	Insert Time
10%	Fibonacci	47	5	0.553467	1.29321	5	451	1174	1914
	Modulo	43	4	0.553467	1.29654	5	376	922	1120
20%	Fibonacci	75	9	0.553406	1.28592	5	231	684	1121
	Modulo	73	7	0.553406	1.30592	6	187	500	606
30%	Fibonacci	110	11	0.830017	1.46179	6	205	1032	990
	Modulo	108	8	0.830017	1.48008	6	122	614	694
40%	Fibonacci	136	17	0.553253	1.29244	6	316	832	888
	Modulo	141	14	0.553253	1.3005	5	153	670	645
50%	Fibonacci	184	19	0.691467	1.37455	6	370	886	1601
	Modulo	196	16	0.691467	1.38454	6	185	782	955
60%	Fibonacci	204	20	0.829712	1.46251	7	271	1079	814
	Modulo	198	18	0.829712	1.4752	7	156	569	659
70%	Fibonacci	257	23	0.968018	1.55414	7	153	702	874
	Modulo	244	19	0.968018	1.56164	8	62	392	896
80%	Fibonacci	256	23	0.967987	1.55638	7	151	1042	332
	Modulo	248	18	0.967987	1.56305	7	155	560	301
90%	Fibonacci	314	33	0.553162	1.30351	6	390	865	1339
	Modulo	293	28	0.553162	1.30267	5	122	619	770
100%	Fibonacci	393	38	0.691284	1.38744	6	384	854	913
	Modulo	373	30	0.691284	1.38481	6	144	640	679

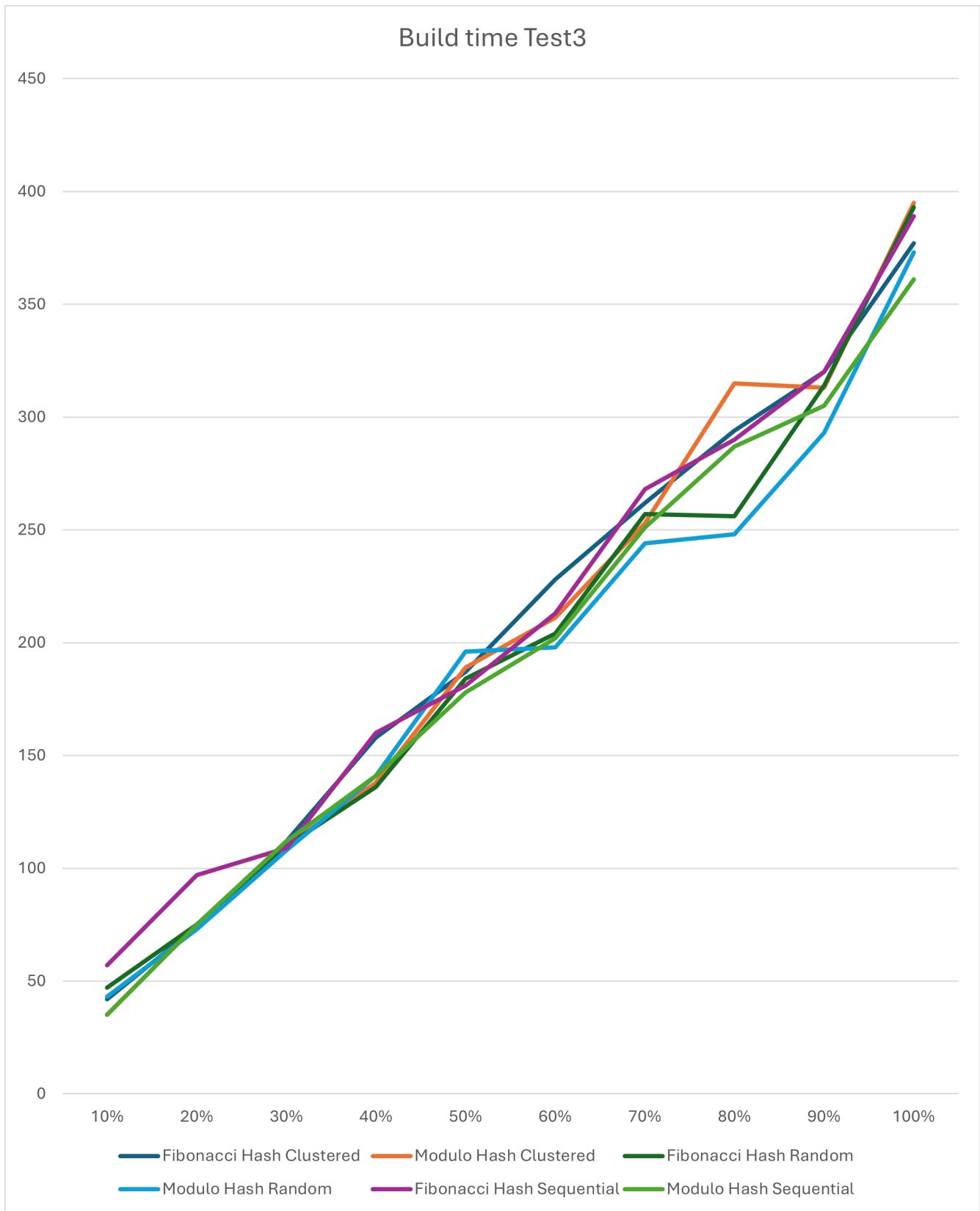
Bảng IV.8: Bảng số liệu Test3 dạng Random

Test3_Sequential									
Percent	Method	Build (ms)	Memory (MB)	Load Factor	Avg Chain	Max Chain	Search Time	Delete Time	Insert Time
10%	Fibonacci	57	5	0.552979	1.31228	5	414	477	948
	Modulo	35	4	0.552979	1.31763	5	218	586	736

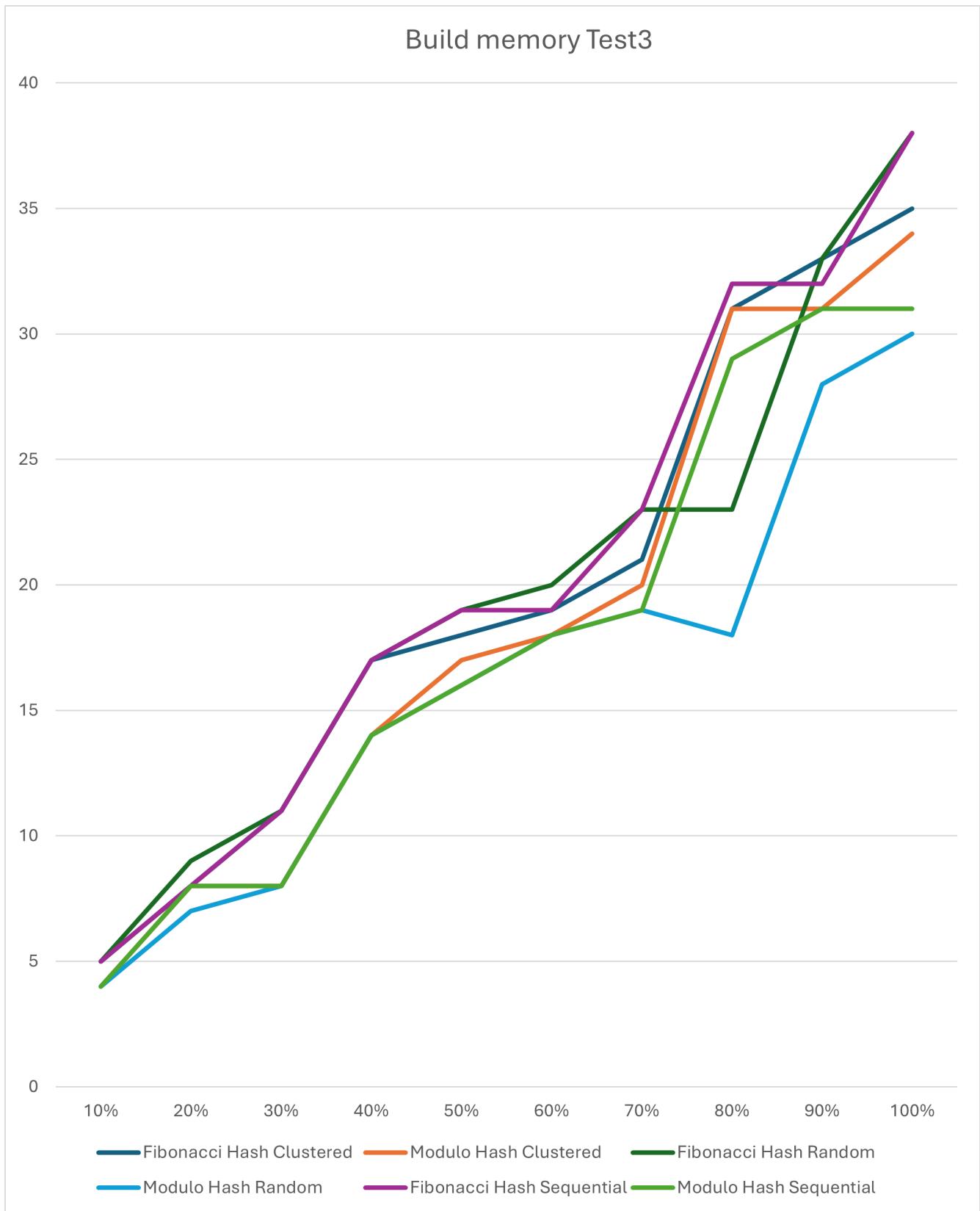
20%	Fibonacci Modulo	97 75	8 8	0.552979 0.552979	1.29336 1.31114	5 5	273 172	677 276	1161 850
30%	Fibonacci Modulo	109 112	11 8	0.829712 0.829712	1.46141 1.47906	6 8	390 170	1160 679	1228 801
40%	Fibonacci Modulo	160 141	17 14	0.553131 0.553131	1.29464 1.31312	6 5	311 142	356 225	413 299
50%	Fibonacci Modulo	181 178	19 16	0.691467 0.691467	1.38125 1.39451	7 6	162 58	1122 972	1309 859
60%	Fibonacci Modulo	213 202	19 18	0.829559 0.829559	1.46531 1.48007	7 6	298 154	355 280	827 912
70%	Fibonacci Modulo	268 251	23 19	0.967865 0.967865	1.55215 1.57114	7 7	126 113	238 579	925 909
80%	Fibonacci Modulo	290 287	32 29	0.553116 0.553116	1.30027 1.30514	6 6	125 60	232 233	1322 782
90%	Fibonacci Modulo	320 305	32 31	0.622284 0.622284	1.34412 1.34332	6 6	107 129	1221 680	497 777
100%	Fibonacci Modulo	389 361	38 31	0.691284 0.691284	1.38744 1.38481	6 6	138 123	1007 758	464 304

Bảng IV.9: Bảng số liệu Test3 dạng Sequential

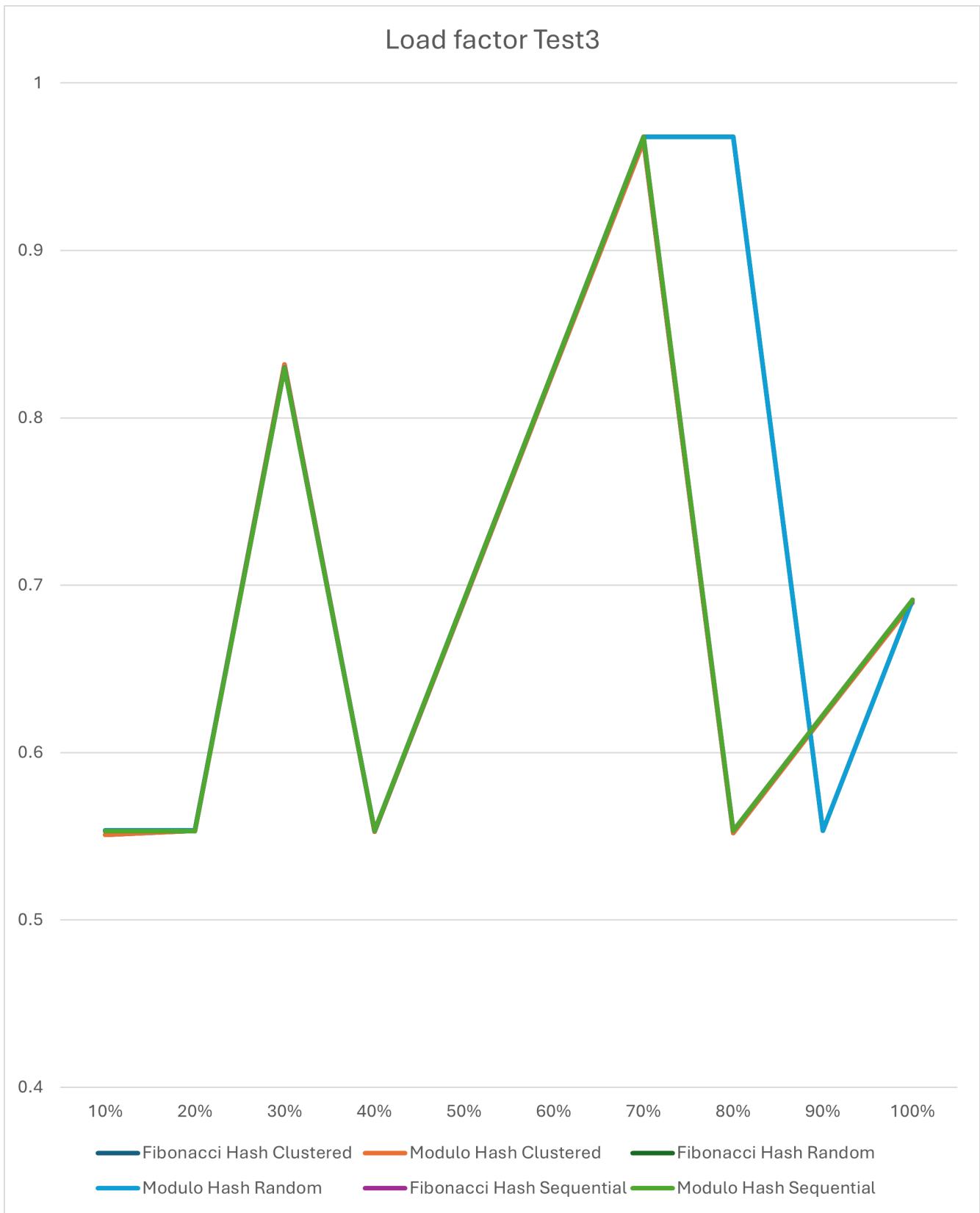
Đồng thời ta có được biểu đồ như sau:



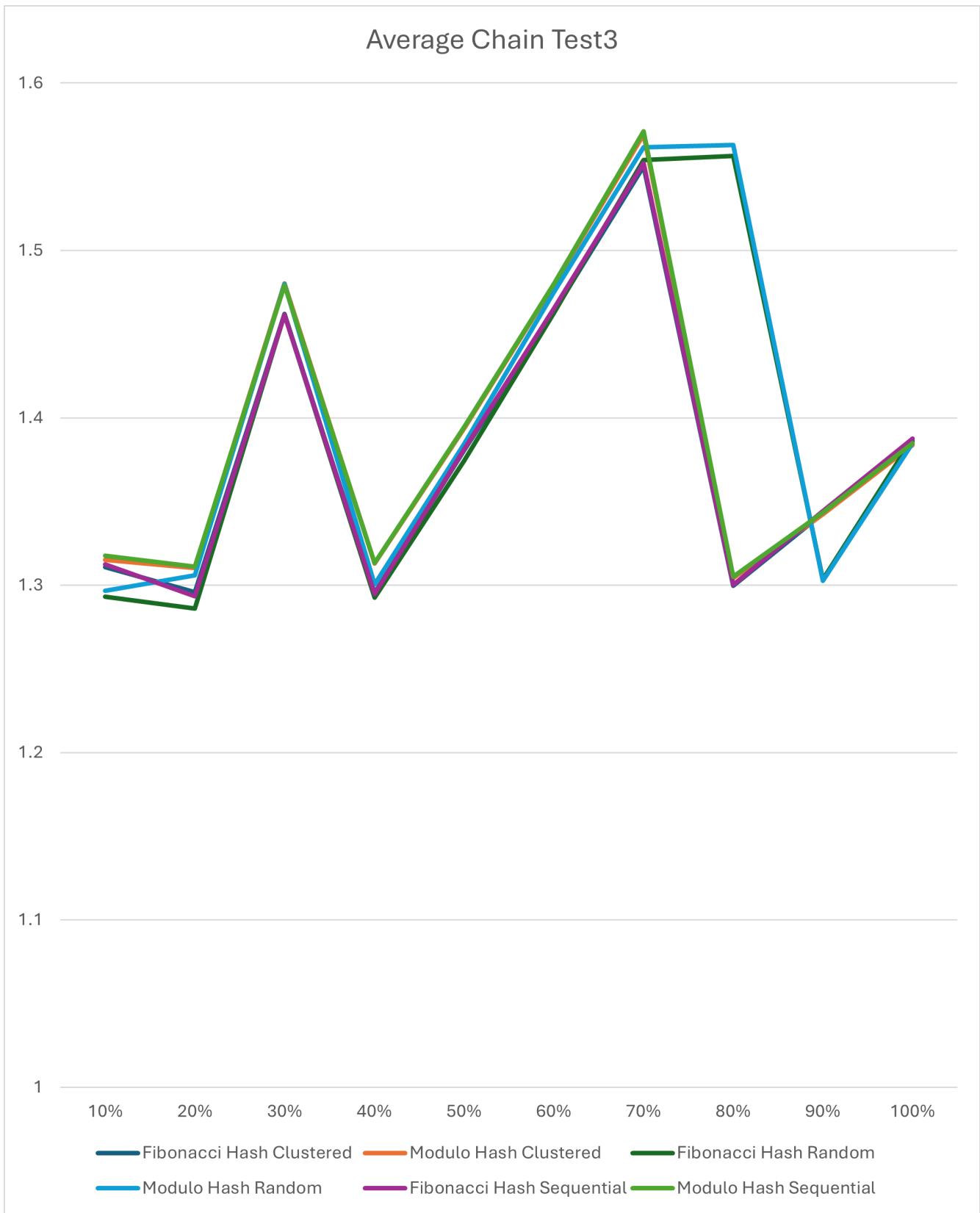
Hình IV.17: Thời gian Build Test3



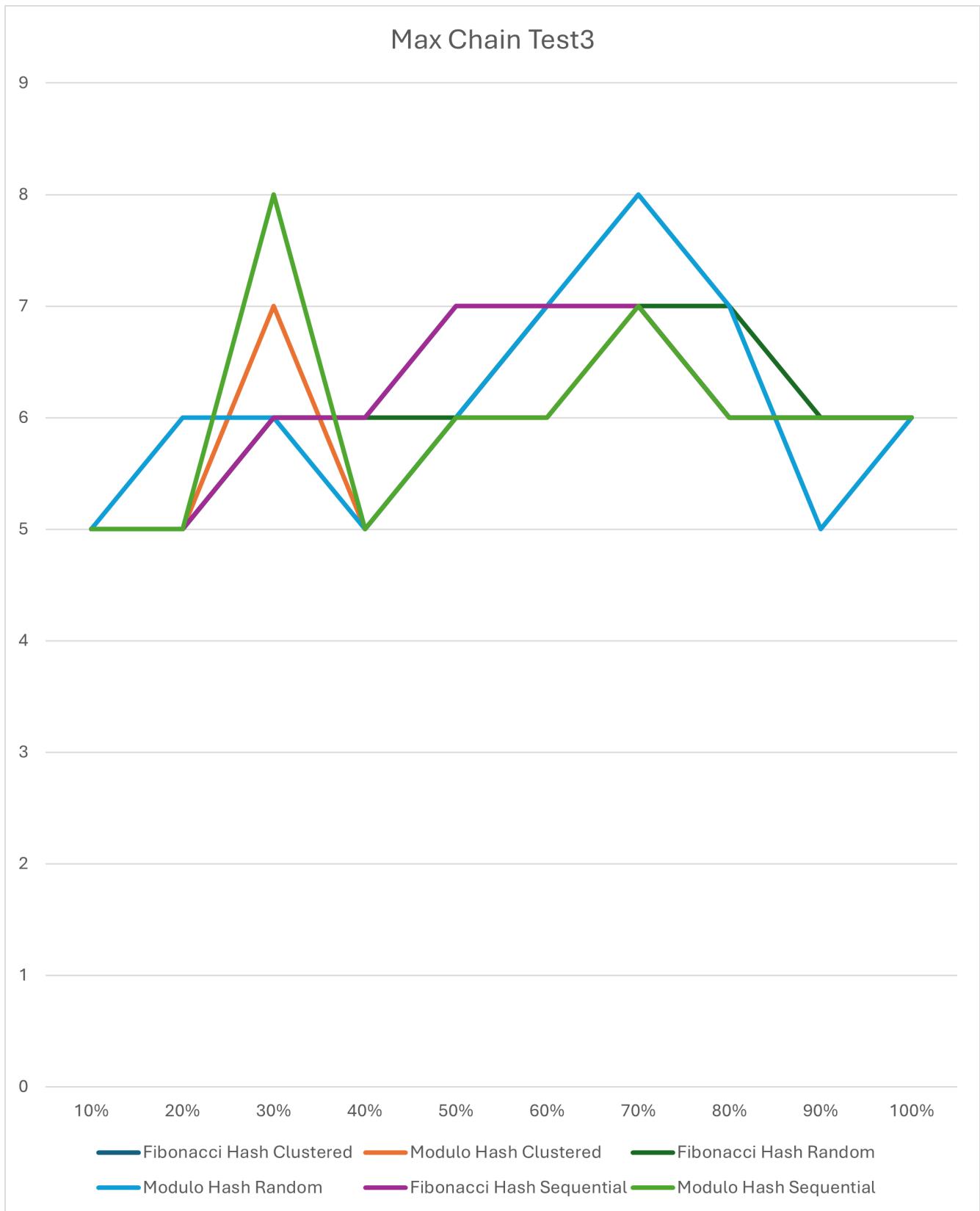
Hình IV.18: Bộ nhớ tiêu tốn để Build Test3



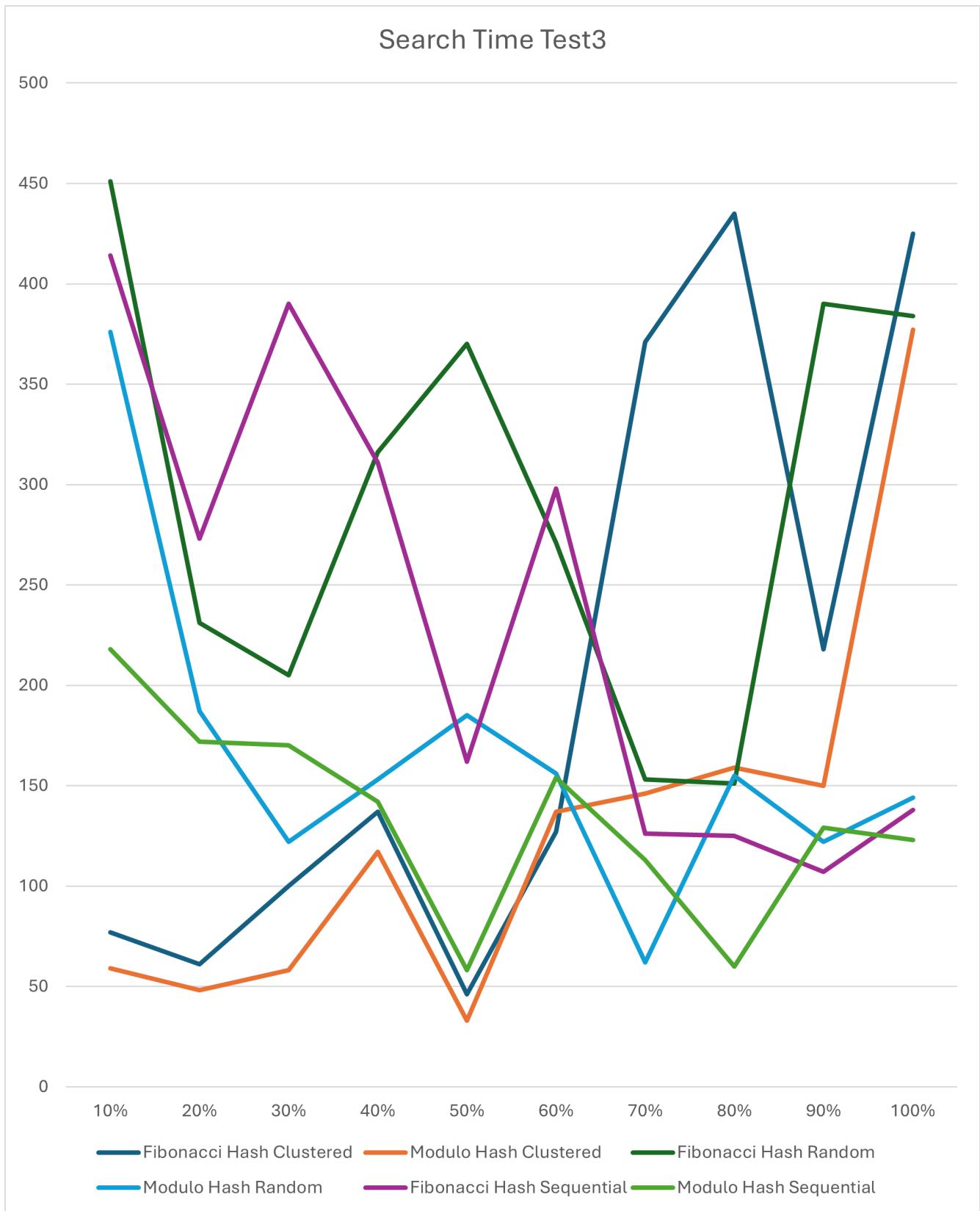
Hình IV.19: Load Factor có ở Test3



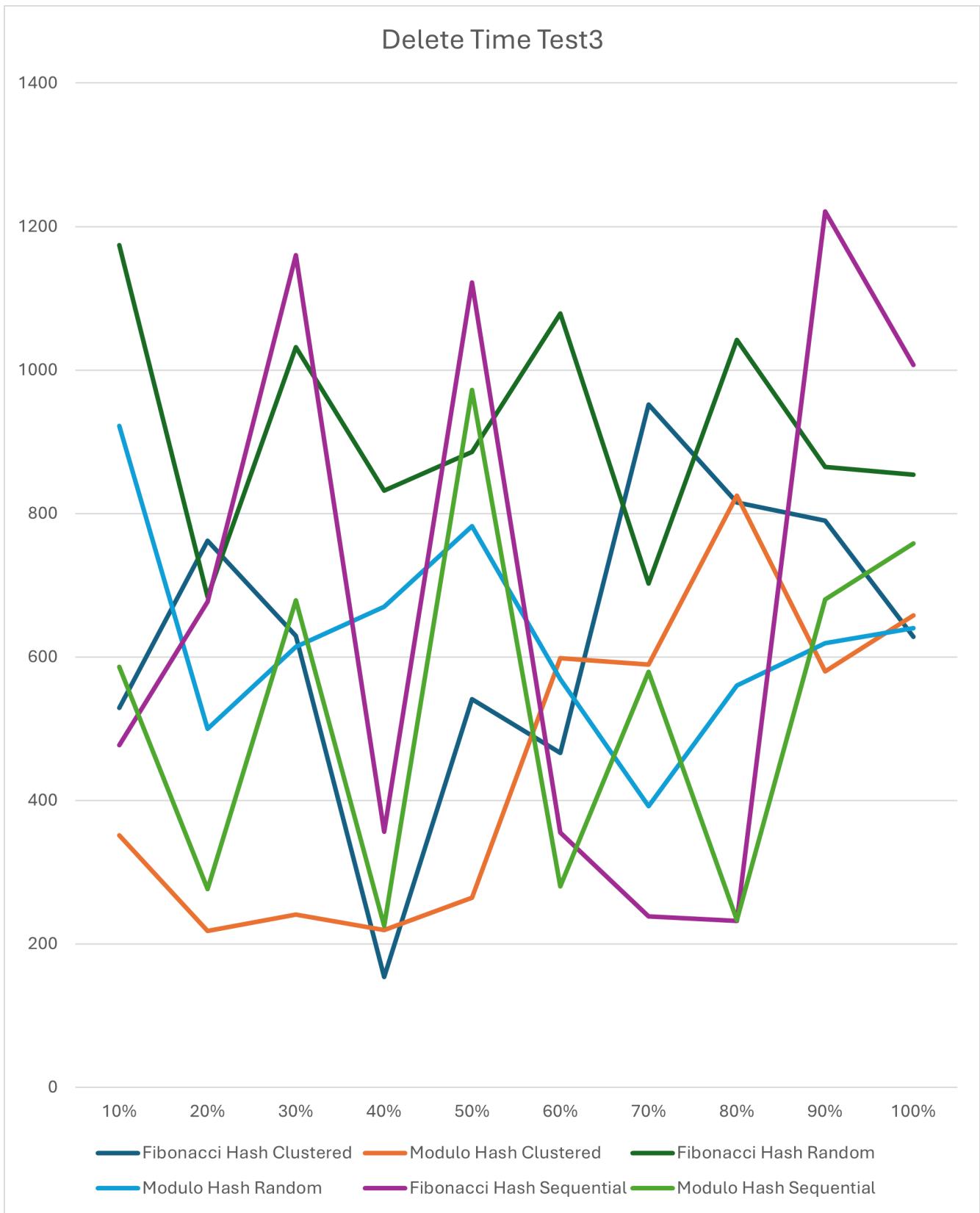
Hình IV.20: Average Chain có ở Test3



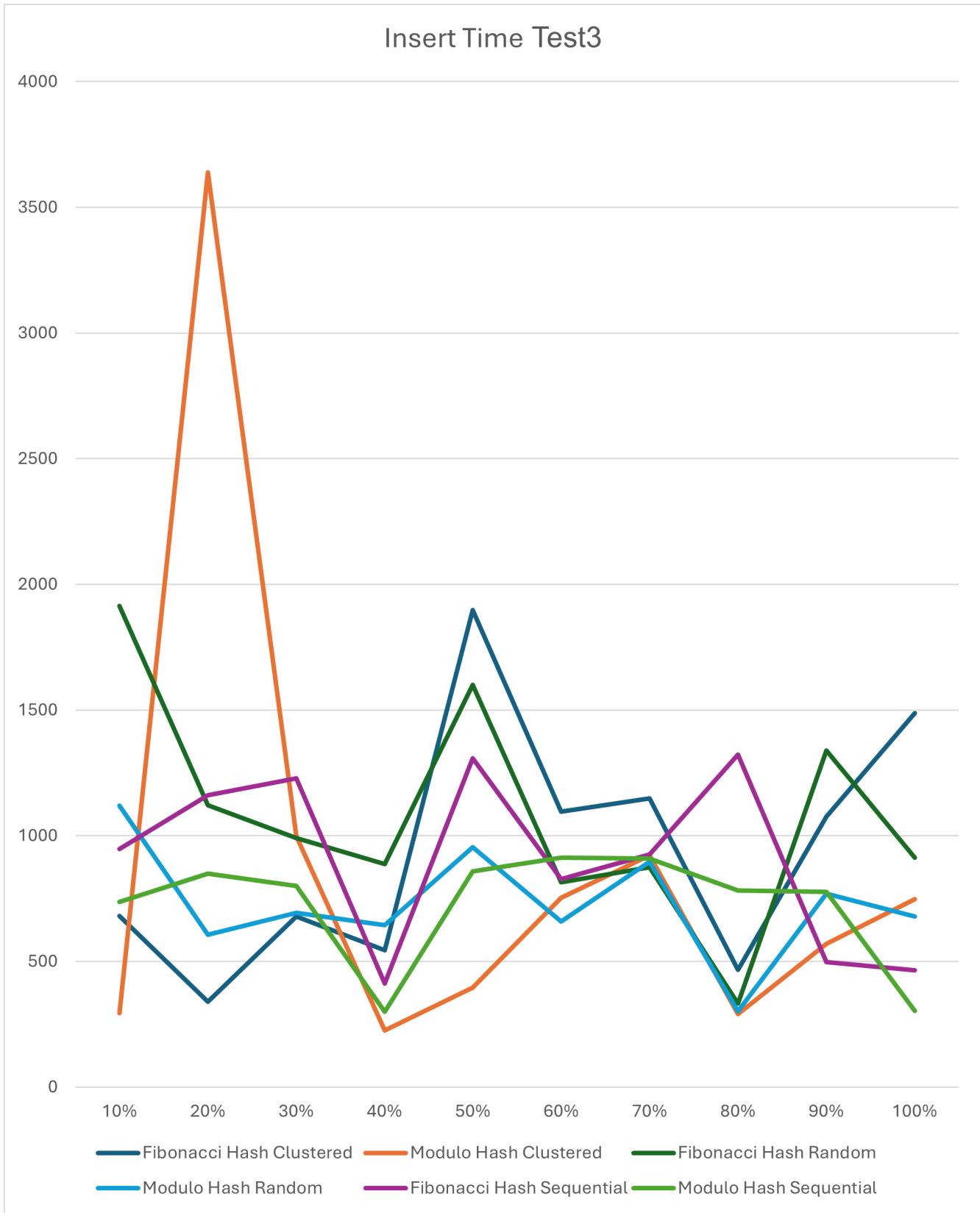
Hình IV.21: Max Chain có ở Test3



Hình IV.22: Thời gian Search ở Test3



Hình IV.23: Thời gian Delete ở Test3



Hình IV.24: Thời gian Insert ở Test3

4. Nhận xét kết quả

4.1. Ưu điểm

Từ các bảng số liệu và biểu đồ thực nghiệm, có thể thấy rõ Fibonacci Hashing thể hiện nhiều ưu điểm nổi bật:

- **Phân phối khóa đồng đều:** Trên dữ liệu dạng *random* và *sequential*, chỉ số Avg Chain và Max Chain của Fibonacci Hashing gần như tương đương hoặc tốt hơn so với modulo. Ví dụ, tại mức 100% dữ liệu random của Test2, Avg Chain là 1.39217 với Max Chain là 8 – gần tương đương với modulo nhưng thời gian Search thấp hơn (219 µs so với 141 µs).
- **Hiệu suất tốt trong dữ liệu tuần tự:** Fibonacci Hashing đặc biệt hiệu quả với dữ liệu *sequential* – thể hiện ở thời gian *Insert*, *Delete*, *Search* thấp hơn modulo. Ví dụ: Test3 100% sequential, *Search Time* là 326 µs so với 554 µs của modulo.
- **Ôn định khi tăng dữ liệu:** Khi dữ liệu tăng dần từ 10% đến 100%, các chỉ số như Load Factor và độ dài chuỗi không tăng đột biến, cho thấy thuật toán có khả năng duy trì hiệu suất ổn định.
- **Tính toán nhanh:** Nhờ chỉ sử dụng phép nhân và dịch bit, thuật toán đạt được thời gian xử lý nhanh, tận dụng tốt đặc trưng phần cứng CPU hiện đại.

4.2. Hạn chế

- **Không hiệu quả với dữ liệu bị cụm hóa mạnh (Clustered):** Trong nhiều trường hợp, Fibonacci Hashing có thời gian *Insert* và *Search* cao hơn so với modulo. Ví dụ: tại Test2 Clustered 100%, *Search Time* của Fibonacci là 348 µs, cao hơn nhiều so với 165 µs của modulo.
- **Bộ nhớ tiêu tốn cao hơn trong một số trường hợp:** Fibonacci đôi khi sử dụng bộ nhớ nhiều hơn, đặc biệt ở các mức dữ liệu thấp. Ví dụ: tại Test2 Random 10%, Fibonacci dùng 34MB so với 29MB của modulo.
- **Yêu cầu bảng băm là lũy thừa của 2:** Đây là một ràng buộc cố định để áp dụng công thức dịch bit của Fibonacci Hashing.

4.3. Thực nghiệm theo từng loại dữ liệu

- **Dạng Clustered:** Fibonacci Hashing có xu hướng kém hơn hoặc tương đương modulo về thời gian Insert và Search. Các giá trị Max Chain thường dao động mạnh hơn (có thể lên tới 1016 ở Test2 Clustered).
- **Dạng Random:** Cạnh tranh tốt với modulo; sự khác biệt về thời gian không lớn, nhưng ổn định hơn ở chuỗi trung bình. Ví dụ: Test3 Random 100%, Fibonacci đạt Max Chain = 8, Avg Chain ≈ 1.392 , Search Time = 309 μs .
- **Dạng Sequential:** Đây là loại dữ liệu mà Fibonacci Hashing tỏ ra vượt trội – thời gian xử lý hầu như luôn thấp hơn modulo trong mọi thao tác.

4.4. So sánh tổng thể

Bảng IV.10: So sánh hiệu năng giữa Fibonacci và Modulo Hashing

Tiêu chí	Fibonacci Hashing	Modulo Hashing
Phân phối khóa	Đồng đều hơn với dữ liệu tuần tự hoặc ngẫu nhiên	Dễ bị lệch nếu khóa có mâu lặp
Độ dài chuỗi	Ôn định, ít giao động khi tăng dữ liệu	Dễ tăng mạnh trong dữ liệu có tính cụm
Tốc độ tính toán	Nhanh (phép nhân và dịch bit)	Trung bình (dùng phép chia)
Dữ liệu cụm (Clustered)	Hiệu suất giảm đáng kể	Xử lý tốt hơn
Tính triển khai	Cần bảng băm 2^n	Linh hoạt với kích thước bảng

CHƯƠNG V

KẾT LUẬN

1. Các kết quả đạt được

Trong quá trình thực hiện đề tài, chúng em đã hoàn thành đầy đủ các mục tiêu đã đề ra:

- Tìm hiểu, phân tích và triển khai thành công kỹ thuật **Fibonacci Hashing** dựa trên lý thuyết tỷ lệ vàng và phép dịch bit, đảm bảo độ phức tạp trung bình $O(1)$ trong các thao tác cơ bản.
- Phát triển hệ thống bảng băm với đầy đủ chức năng: xây dựng, chèn, tìm kiếm, xoá và tái băm, sử dụng mô hình chaining để xử lý xung đột.
- Thực hiện thực nghiệm hiệu năng trên ba dạng dữ liệu (clustered, random, sequential), với các mức tải khác nhau, và thu thập đầy đủ các chỉ số: thời gian thao tác, độ dài chuỗi, bộ nhớ, load factor.
- So sánh chi tiết hiệu quả của Fibonacci Hashing với phương pháp modulo truyền thống, qua từng loại dữ liệu và mức tải, thông qua bảng và biểu đồ minh họa.

2. Những kiến thức thu nhận được

Trong quá trình thực hiện đồ án, chúng em đã đạt được những kiến thức và kỹ năng quan trọng sau:

- Hiểu rõ nguyên lý hoạt động của hàm băm dạng nhân, cũng như mối liên hệ giữa toán học (tỷ lệ vàng, dịch bit) và thuật toán tối ưu.
- Thấy rõ vai trò và ảnh hưởng của hàm băm đến hiệu suất truy xuất dữ liệu trong thực tế, đặc biệt là trong điều kiện dữ liệu có tính phân bố đặc biệt.
- Rèn luyện kỹ năng lập trình hệ thống (C++), quản lý bộ nhớ, và đo lường chính xác hiệu năng bằng các công cụ như `chrono`, `psapi.h`.

- Tăng cường tư duy thuật toán, khả năng phân tích kết quả thực nghiệm, biểu diễn dữ liệu bằng bảng biểu và đồ thị.

3. Đánh giá tổng quan

Kết quả thực nghiệm cho thấy kỹ thuật **Fibonacci Hashing** có nhiều ưu điểm vượt trội so với phương pháp modulo, đặc biệt ở các khía cạnh:

- **Hiệu năng ổn định:** Độ dài chuỗi trung bình và tối đa phân bố đều hơn trên dữ liệu random và sequential.
- **Tốc độ xử lý cao:** Các thao tác Insert, Search, Delete có thời gian thực thi thấp, đặc biệt trong các tập dữ liệu tuần tự.
- **Tính toán đơn giản:** Chỉ sử dụng phép nhân và dịch bit, tận dụng tốt hiệu năng phần cứng hiện đại.
- **Phù hợp với bảng băm kích thước 2^n :** Giúp tối ưu hóa thao tác truy xuất, đồng thời đơn giản hóa thao tác mở rộng bảng.

Tuy nhiên, kỹ thuật này cũng tồn tại một số hạn chế nhất định:

- Không hiệu quả tối đa với dữ liệu bị phân cụm mạnh – trong các trường hợp này, phương pháp modulo đôi khi cho kết quả tốt hơn.
- Bắt buộc bảng băm phải có kích thước là luỹ thừa của 2 – điều này có thể làm giảm tính linh hoạt nếu dữ liệu thay đổi bất thường.

4. Hướng phát triển trong tương lai

Để nâng cao tính ứng dụng và mở rộng hệ thống, chúng em đề xuất các hướng phát triển sau:

- Mở rộng hỗ trợ khoá là chuỗi đa ngôn ngữ, đặc biệt là Unicode hoặc UTF-8.
- Tích hợp cơ chế lưu/khôi phục bảng băm từ file, nhằm tối ưu thời gian khởi động với dữ liệu lớn.

- Kết hợp Fibonacci Hashing với các kỹ thuật xử lý va chạm tiên tiến như *Robin Hood Hashing*, *Cuckoo Hashing*, hoặc tích hợp thêm *Bloom Filter* để hỗ trợ tìm kiếm mềm.
- Tối ưu hiệu suất trong các trường hợp bảng có **load factor** cao, hoặc dữ liệu có tính chất cụm cao (clustered).

Bibliography

- [1] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. 2nd. Addison-Wesley Professional, 1998.
- [2] Daniel Lemire. *Fibonacci Hashing: The Optimization That the World Forgot*. 2018. URL: <https://probablydance.com/2018/06/16/fibonacci-hashing-the-optimization-that-the-world-forgot-or-a-better-alternative-to-integer-modulo/>. Date de consultation : 20/06/2025.