# Hash con Metodo della Divisione vs Metodo della Moltiplicazione

Laboratorio di Algoritmi - Esercizio C

Samuel Bruno

10 Ottobre 2022

**Abstract**

L'obiettivo di questa relazione è quello di analizzare le differenze tra Hash con Metodo della Divisione e Hash con Metodo della Moltiplicazione.

## 1 Introduzione

Un **Hash con metodo della Divisione** è un metodo che serve a generare una funzione hash; consiste nell' inserire una **chiave k** in uno degli **m slot** calcolato come resto della divisione di k per m.
Ovvero **h(k) = (k mod m)**. Questo risulta essere un metodo molto veloce in quanto si effettua solo un' operazione di divisione. Il valore m dovrebbe, preferibilmente, essere diverso da una potenza di 2, in particolare sarebbe meglio utilizzare un numero primo lontano da una potenza di 2, in modo da limitarne le collisioni.

Un **Hash con metodo della Moltiplicazione**, invece, consiste in 2 passi:

i.) Si moltiplica la chiave k per una costante A nell'intervallo $0 < A < 1$ e si estrae la parte frazionaria (ossia $kA - \lfloor(kA)\rfloor$);

ii.) Si moltiplica il valore ottenuto per m e si prende la parte intera inferiore del risultato, ovvero $h(k) = m * \lfloor(kA - \lfloor(kA)\rfloor)\rfloor$.

In questo caso m non è critico e può assumere un valore 2p (potenza di 2). In particolare si ha la probabilità che questo metodo funzioni bene quando il valore di A si avvicina al valore: $(\sqrt{5} - 1)/2$.

Nel programma verranno implementate le **tabelle hash** con **gestione delle collisioni basate su concatenamento**.
Il confronto verrà effettuato proprio sull'applicazione dei suddetti metodi e, più dettagliatamente, verrà analizzato attraverso i relativi tempi di esecuzione. Date le basi, si può procedere a descrivere la struttura dati della tabella hash.

## 2 Struttura dati

Nelle tabelle hash con risoluzione delle collisioni per concatenazione (chaining), gli elementi collidenti vengono inseriti nella stessa posizione della tabella in una lista concatenata. In questo modo, una tabella hash *T[0,..,m-1]* contiene, in posizione *i*, un puntatore alla testa di una lista di oggetti con valore di hash *i*.

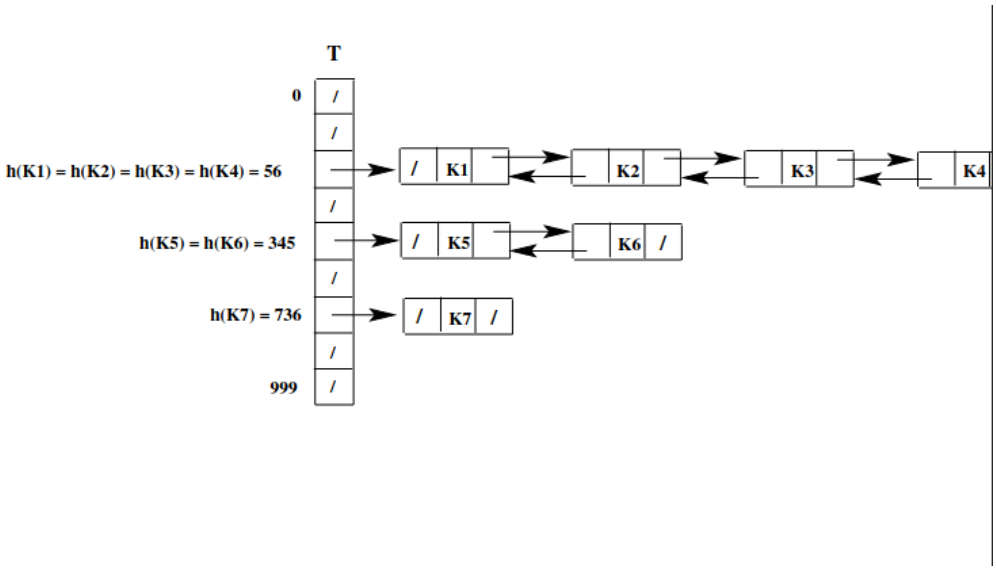


Figure 1: Hash con concatenamento

Inoltre, implementa i seguenti metodi:

i.) **Chained-Hash-Search(T,k):** Ricerca un elemento con chiave k nella lista T[h(k)].

ii.) **Chained-Hash-Insert(T,x):** Inserisce x in testa alla lista T[h(x.key)].

iii.) **Chained-Hash-Delete(T,x):** Cancella x dalla lista T[h(x.key)].

# 3 Prestazioni attese

Le prestazioni dell'hashing sono valutate sulla base del fatto che ogni chiave ha la **stessa probabilità** di essere sottoposta ad hashing in qualsiasi slot della tabella. Per ottenere prestazioni migliori, si dovrebbe implementare un'adeguata funzione hash; questa ottimizza la prestazione distribuendo le chiavi in modo uniforme sulle posizioni della tabella, minimizzando le collisioni.

## 3.1 Caso peggiore

Supponiamo di usare una **buona funzione** di hash:

- **Operazione di inserimento:** Ha costo costante $O(1)$.
- **Operazione di ricerca:** Nel caso peggiore, tutti gli inserimenti hanno generato una collisione e quindi sono stati concatenati nella stessa lista. La tabella hash, quindi, si riduce a una struttura lineare, come una lista collegata. La complessità pessima della ricerca è dunque $O(n)$, dove n è il numero di elementi nella tabella.
- **Operazione di cancellazione:** Se si usano liste bidirezionali, il tempo nel caso peggiore è $O(1)$. Con liste semplici, invece, richiede lo stesso tempo della ricerca.

## 3.2 Caso medio della ricerca

### 3.2.1 Fattore di carico

Si definisce il **fattore di carico** della tabella di hash come $\alpha = n/m$, dove $\alpha \geq 0$ è un numero razionale, n è il numero degli elementi memorizzati e m è la dimensione della tabella. Se la funzione hash $h$ soddisfa l'ipotesi di uniformità semplice, $\alpha$ corrisponde alla lunghezza media di una qualsiasi lista di concatenazione.

Si distinguono i casi di ricerca con insuccesso (ricerca di una chiave non presente in tabella) e di ricerca con successo (ricerca di una chiave presente in tabella). In generale, si presuppone che il costo medio di una ricerca con successo sia minore del costo di una ricerca con insuccesso.

### 3.2.2 Ricerca con insuccesso

La ricerca con insuccesso consiste nel calcolo di un valore della funzione hash, per individuare la lista di concatenazione in cui cercare e nella scansione esaustiva di tale lista. Essendo $\alpha$ la lunghezza media di una lista di concatenazione, si può affermare che, il costo medio della ricerca con insuccesso è $\theta(1 + \alpha)$.

### 3.2.3 Ricerca con successo

La ricerca con successo consiste nel calcolo di un valore della funzione hash, per individuare la lista di concatenazione in cui cercare e nella scansione di tale lista, fino a trovare l'elemento cercato. Essendo $\alpha$ la lunghezza media di una lista di concatenazione e ritenendo che, mediamente, si debba scandire metà della lista prima di trovare l'elemento cercato, si può affermare che il costo medio della ricerca con successo è $\theta(1 + \alpha/2) = \theta(1 + \alpha)$.

### 3.2.4 Complessità media della ricerca

Combinando entrambe le analisi possiamo infine affermare che:

Se

$$n = O(m)$$

allora

div.delete(0) mul.delete(0)
div.delete(0) mul.delete(0)

$$\alpha = n/m = O(m)/m = O(1)$$

e il costo medio della ricerca risulta

$$\theta(1 + \alpha) = \theta(1 + 1) = \theta(1)$$

Di seguito una tabella riassuntiva contenente tutte le complessità:

| Operazione | Medio | Peggiore |
|---|---|---|
| Inserimento | $O(1)$ | $O(n)$ |
| Ricerca | $O(1)$ | $O(n)$ |
| Cancellazione | $O(1)$ | $O(n)$ |

Table 1: Complessità dell'hashing con concatenamento

# 4 Documentazione del progetto

Il programma è formato in totale da 1 file contenente tutte le classi necessarie:

i.) **main.py:** Nel file sono presenti le classi Node, LinkedList, HashTableDivision ed HashTableMultiplication contenenti l'implementazione delle tabelle hash con metodo del concatenamento e i relativi metodi della divisione e della moltitplicazione. Infine è presente anche il confronto tra i due metodi.

Il progetto presenta quindi la seguente struttura:

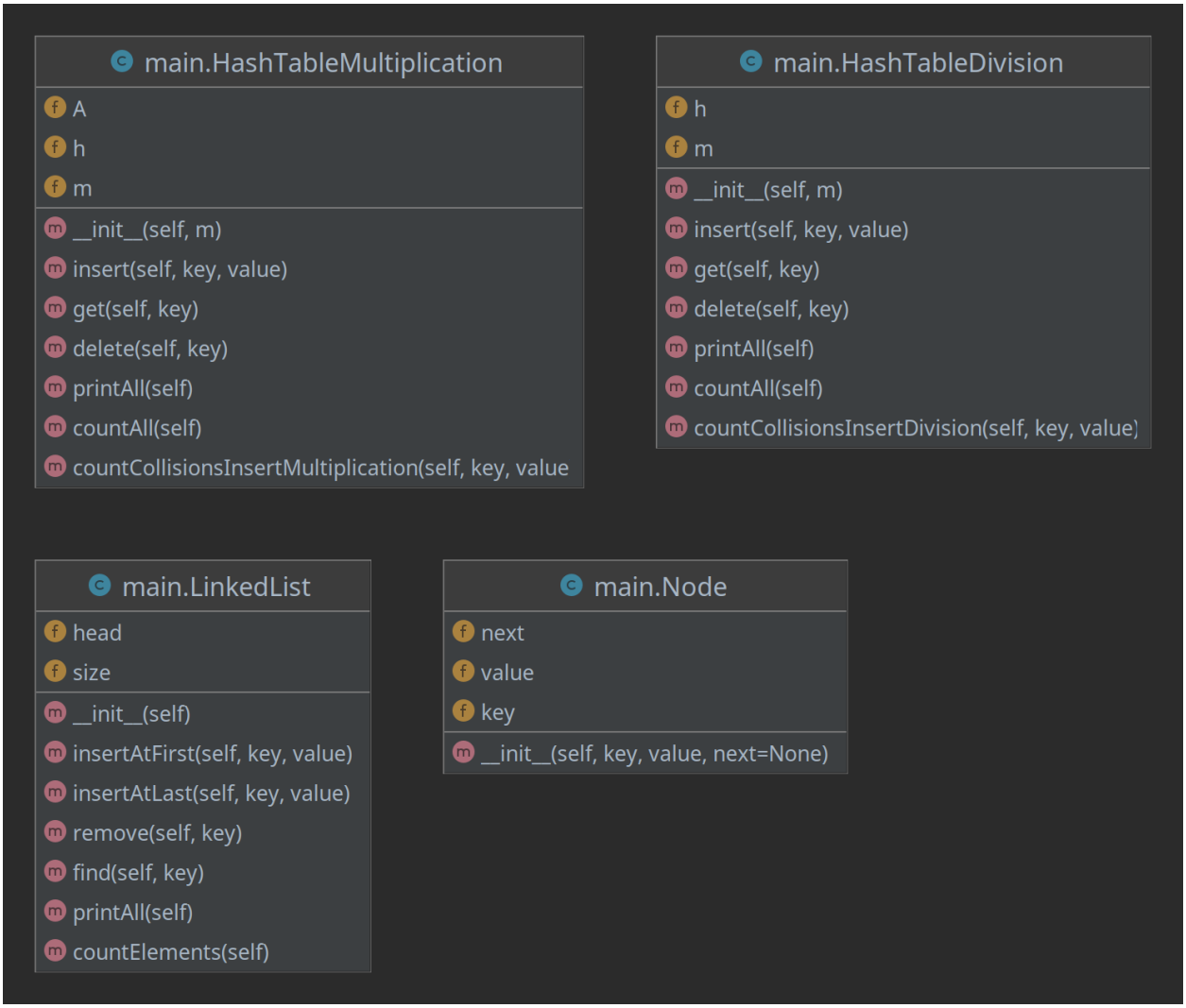


Figure 2: Diagramma delle classi

## 4.1 Classi e metodi

### 4.1.1 Classe Node

E' la classe che implementa il singolo nodo della lista. I suoi attributi sono: key, value, next.

### 4.1.2 Classe LinkedList

E' la classe che implementa la linked list e possiede i seguenti attributi: head, size. Questa classe fornisce i seguenti metodi:

- **insertAtFirst:** Inserisce un elemento in testa alla lista.
- **insertAtLast:** Inserisce un elemento in fondo alla lista.
- **remove:** Rimuove una chiave dalla lista.
- **find:** Ricerca una chiave all'interno della lista.
- **countElements:** Conta gli elementi all'interno della lista, scorrendola interamente.

#### 4.1.3 Classe HashTableDivision

E' la classe che implementa la tabella hash usando il metodo della divisione. Presenta gli attributi m (dimensione della tabella), h (funzione hash) e fornisce questi metodi:

- **insert:** Inserisce una chiave attraverso un indice, calcolato col metodo della divisione, un elemento all'interno della tabella hash.
- **get:** Ricerca una chiave richiesta in input.
- **delete:** Cancella una chiave richiesta in input.
- **countAll:** Conta, scorrendo ogni indice e la sua relativa lista concatenata, tutti gli elementi presenti nella tabella hash.
- **countCollisionsInsertDivision:** Conta le collisioni che avvengono quando si inserisce una chiave, al crescere del fattore di caricamento $\alpha$.

#### 4.1.4 Classe HashTableMultiplication

E' la stessa classe descritta sopra, solo che il metodo di calcolo della funzione hash viene effettuato attraverso il metodo della moltiplicazione.

#### 4.1.5 Classe Main

In questa classe avviene il confronto tra i due metodi usati per calcolare la funzione hash. Vedremo nella prossima sezione i risultati di tali esperimenti.

# 5 Risultati Sperimentali

## 5.1 Primo test

Nel primo test, gli esperimenti sono stati effettuati, per il **metodo della divisione**, su una sequenza di 127 interi positivi mentre per il **metodo della moltiplicazione** su una sequenza di 128 interi positivi. Nel primo metodo, la scelta ottimale per $m$ è quella di prendere un numero primo che non sia una potenza di 2, da qui il 127. Invece per il secondo metodo, la scelta ottimale per $m$ è, viceversa, quella di avere un numero potenza di 2, da qui il 128.
I test sono stati eseguiti su un HP Pavilion DV6 le cui specifiche sono:
*Processore Intel(R) Core(TM) i5-2450M CPU @ 2.50GHz 3.10 GHz Dual-core Quad-thread*
*RAM 8 GB 1333 Mhz*
*Sistema Operativo EndeavourOS - kernel Linux*

Le misurazioni effettuate riguardano il tempo di esecuzione dei processi nella CPU e i vari test sono stati eseguiti per più di cinque volte, in modo da verificare la costanza dei risultati.
Nella seguente tabella elenco gli ultimi 5 test eseguiti (con i risultati espressi in $\mu$**s**):

| DIV vs MUL | | Div (µs) | Mul (µs) | Migliore |
|---|---|---|---|---|
| TEST N° 1 | INSERT (I) | 3.079 | 3.707 | DIV |
| | SUCCESSFUL FIND (F) | 11 | 5 | MUL |
| | NON SUCCESSFUL FIND (NF) | 4 | 27 | DIV |
| | DELETE (D) | 7 | 6 | MUL |
| TEST N° 2 | I | 2.106 | 2.434 | DIV |
| | F | 10 | 4 | MUL |
| | NF | 4 | 19 | DIV |
| | D | 6 | 4 | MUL |
| TEST N° 3 | I | 3.036 | 3.815 | DIV |
| | F | 17 | 7 | MUL |
| | NF | 5 | 27.428 | DIV |
| | D | 7.613 | 6 | MUL |
| TEST N° 4 | I | 3.098 | 3.419 | DIV |
| | F | 18 | 6 | MUL |
| | NF | 4 | 33 | DIV |
| | D | 8 | 7 | MUL |
| TEST N° 5 | I | 3.088 | 3.697 | DIV |
| | F | 21 | 6 | MUL |
| | NF | 3 | 32 | DIV |
| | D | 8 | 6 | MUL |

Figure 3: Risultati sul confronto tra metodo della divisione e moltiplicazione

## 5.2 Secondo test

Nel secondo test, invece, l'esperimento consiste nel contare quante collisioni si hanno eseguendo un numero variabile di inserimenti in una tabella hash con entrambi i metodi; in pratica constatare cosa accade al crescere del fattore di caricamento $\alpha = n/m$. Per semplicità e per rendere equivalenti le due misurazioni, non ho tenuto considerazione di nessun $m$ ottimo, prendendo dunque come riferimento 1000 interi positivi (come massima dimensione della tabella) ed inserendo, ad ogni ciclo di test, rispettivamente, un massimo di elementi ($n$) di: 10, 25, 50, 75, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000.
Di seguito una tabella riassuntiva dell'esperimento:

| N = | α = n/m | N° collisioni divisione | % | N° collisioni moltiplicazione | % |
|---|---|---|---|---|---|
| 10 | 0,01 | 0 | 0 | 0 | 0 |
| 25 | 0,025 | 0 | 0 | 1 | 4 |
| 50 | 0,05 | 1 | 2 | 1 | 2 |
| 75 | 0,075 | 4 | 5,3333333 | 8 | 10,6667 |
| 100 | 0,1 | 2 | 2 | 3 | 3 |
| 200 | 0,2 | 20 | 10 | 24 | 12 |
| 300 | 0,3 | 35 | 11,666667 | 43 | 14,3333 |
| 400 | 0,4 | 73 | 18,25 | 71 | 17,75 |
| 500 | 0,5 | 102 | 20,4 | 129 | 25,8 |
| 600 | 0,6 | 139 | 23,166667 | 162 | 27 |
| 700 | 0,7 | 198 | 28,285714 | 239 | 34,1429 |
| 800 | 0,8 | 233 | 29,125 | 287 | 35,875 |
| 900 | 0,9 | 306 | 34 | 351 | 39 |
| 1000 | 1 | 366 | 36,6 | 407 | 40,7 |
| Totale | | 1479 | 15,77% | 1726 | 19,02% |

Figure 4: Risultati sul confronto delle collisioni al crescere del fattore di caricamento

**N.B.:** Da notare che il campo percentuale viene inteso come la percentuale di collisioni avvenute nell'inserimento di N elementi, quindi matematicamente n° di collisioni diviso n elementi.

## 5.3 Analisi dei risultati

Come possiamo vedere dalla *Figura 3* i risultati sono costanti e coerenti dopo ogni test, che vede il metodo della **divisione** più vantaggioso durante **l'inserimento** e la **ricerca senza successo**, mentre il metodo della **moltiplicazione** migliore durante la **ricerca con successo** e la **cancellazione** di una chiave. L'unica differenza tra questi due risultati è che le tempistiche di computazione della moltiplicazione, rispetto a quelle della divisione, sono solo marginalmente migliori, ma nel confronto dei suoi processi,la divisione risulta nettamente più veloce. Invece nella tabella più in basso (*Figura 4*), possiamo notare come il numero delle collisioni della divisione sia alquanto inferiore rispetto alla controparte, con una percentuale media di collisioni minore. Entrambe comunque tendono ad incrementare il numero delle collisioni in maniera direttamente proporzionale all'aumento del numero degli elementi da inserire nella tabella.

## 5.4 Analisi delle complessità

Analizzando le complessità dei due metodi nell'operazione dell'inserimento (*Figura 5*), notiamo come questi tendano sempre a crescere leggermente con l'avanzare degli elementi inseriti in tabella. In questo caso, il numero degli elementi inseriti corrisponde esattamente alla dimensione totale della tabella, questo fa si che il costo computazionale non rientri precisamente nel caso medio $O(1)$, ma invece, tenda ad essere un **approssimazione** del caso peggiore $O(n)$ (improbabile però che avvenga precisamente, data la randomicità dell'inserimento degli elementi). Tutti i tempi nei grafici sono espressi in *secondi*:

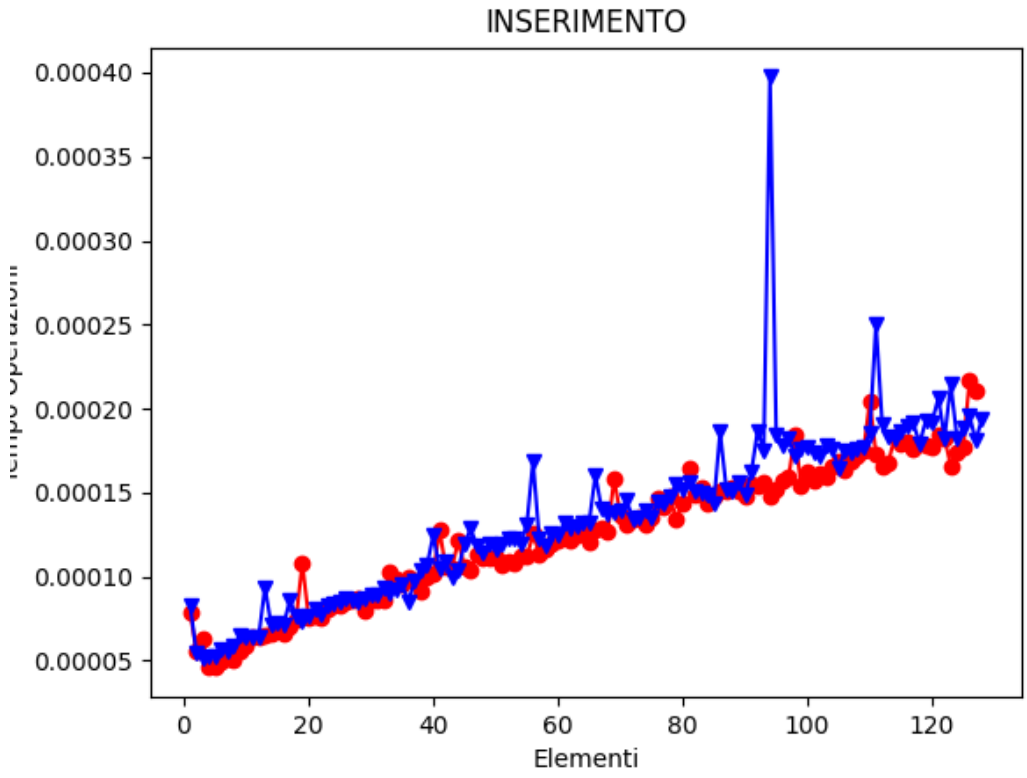


Figure 5: In **rosso** Hash con metodo della divisione ed in **blu** Hash con metodo della moltiplicazione

Per quanto riguarda la ricerca, invece, la situazione risulta essere conforme a quanto detto nella *Sezione 3*. Le operazioni di ricerca con successo e ricerca senza successo, vengono effettuate con un tempo quasi costante $O(1)$, sia per il metodo della divisione (*Figura 6*) che per il metodo della moltiplicazione (*Figura 7*).

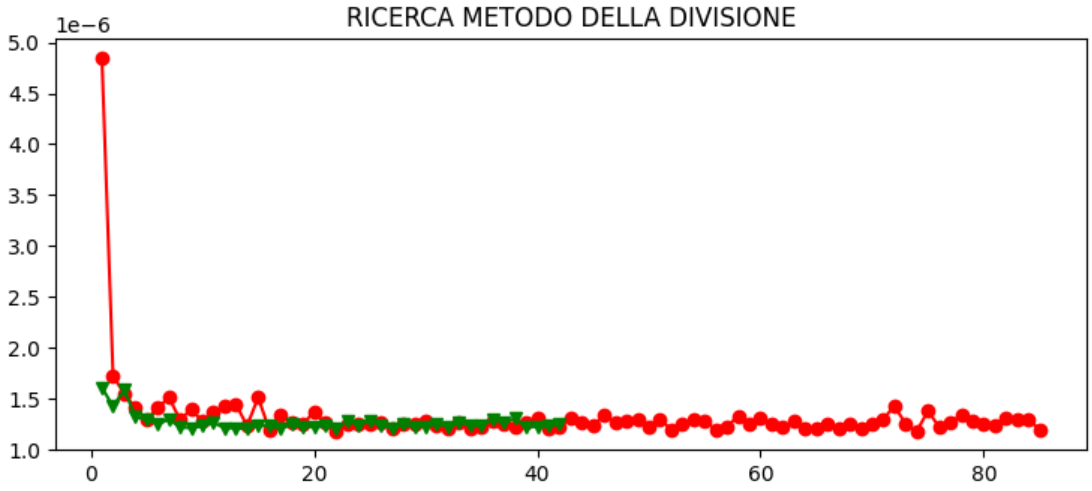


Figure 6: Confronto Ricerca **con successo** e **senza successo** del metodo della divisione

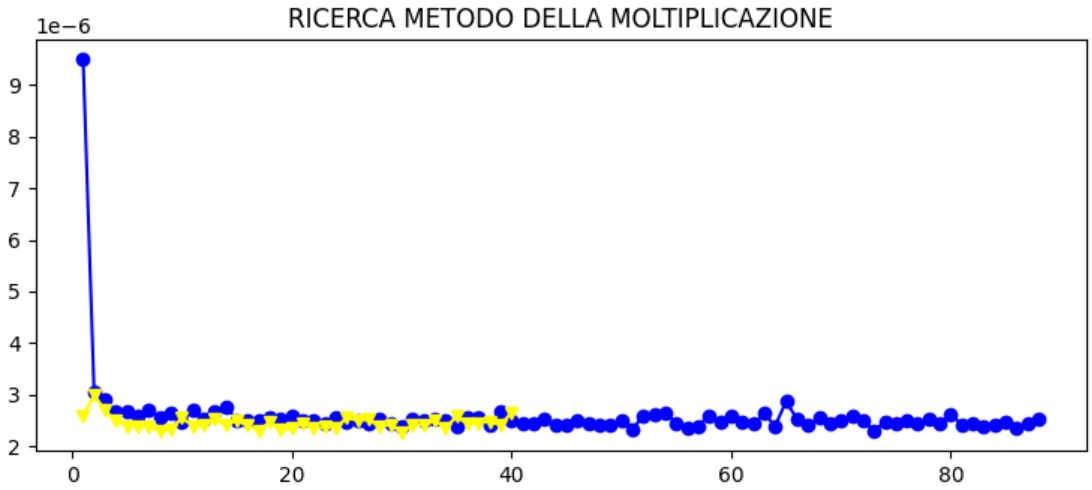


Figure 7: Confronto Ricerca **con successo** e **senza successo** del metodo della moltiplicazione

Infine, non ho incluso l'operazione di cancellazione, poichè sostanzialmente molto simile alla ricerca, evitando quindi di inserire risultati ripetitivi.

# 6 Conclusione

In conclusione, come si evince dagli esperimenti effettuati, nella maggior parte dei casi, è conveniente optare per l'utilizzo del **metodo della divisione** in quanto, non solo il calcolo della sua funzione hash risulta più semplice (essendo una sola operazione in modulo), ma si rivela anche leggermente meno laborioso nei calcoli (in CPU). Risulta quindi più efficace, riducendo la quantità di collisioni di elementi con la stessa chiave e diminuendo così il principale problema degli inserimenti in una tabella hash.