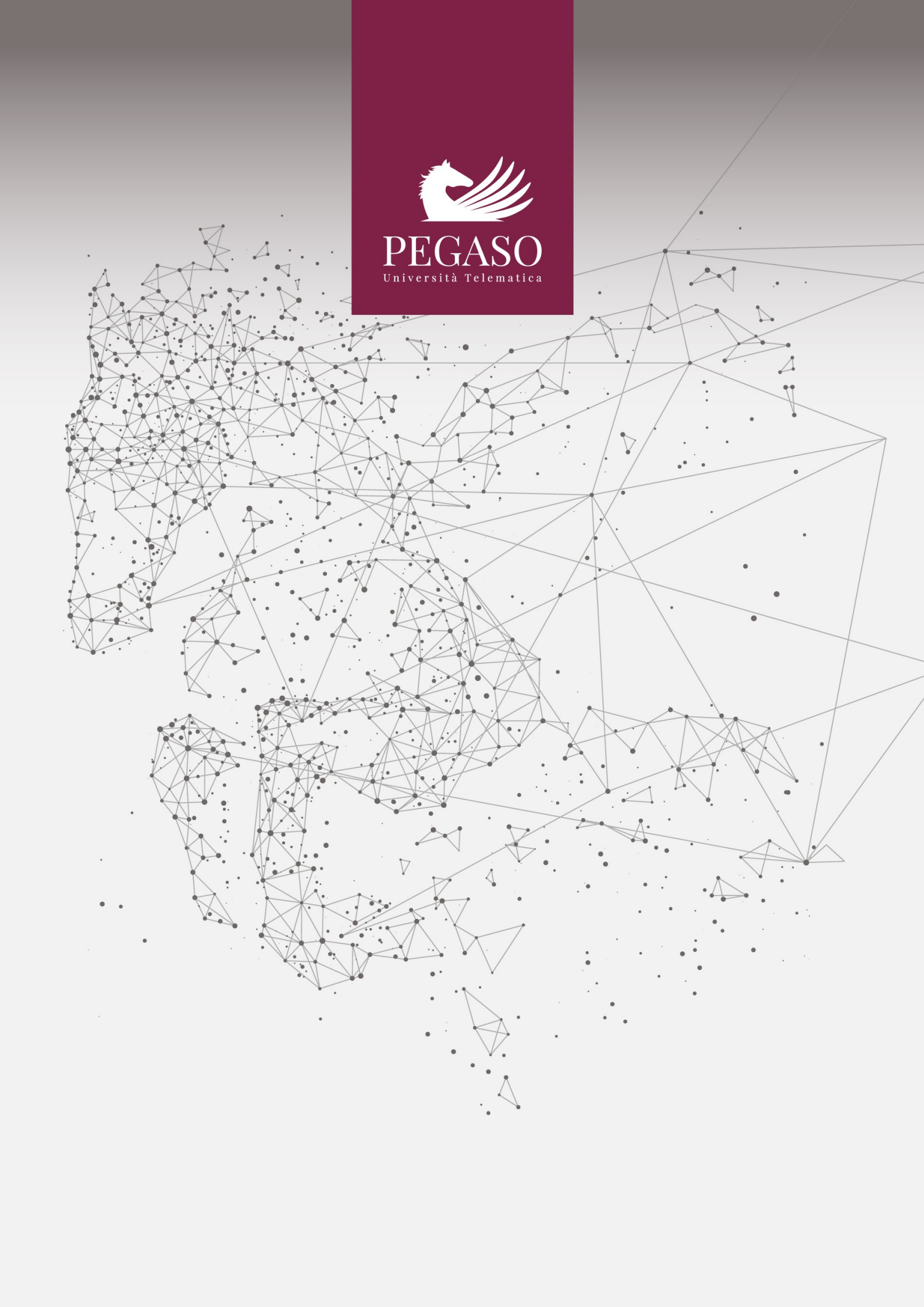
PEGASO
Università Telematica

Indice

1. Hashing

La **tabella hash** è un vettore $T[0 \ldots m-1]$ di dimensione m in cui sono memorizzate le chiavi di un insieme universo U di dimensione u. La modalità di memorizzazione è determinata da una **funzione hash** definita come

$$h : U \rightarrow \{0, 1, \ldots, m-1\}$$

Data dunque una chiave $k \in U$, a questa viene associato un valore hash pari ad $h(k)$ e tale valore viene utilizzato per calcolare l'indice della tabella hash in cui memorizzare la chiave. In altre parole, la funzione hash converte la chiave in un intero che viene utilizzato per determinare l'indice della tabella hash in cui la chiave viene memorizzata.

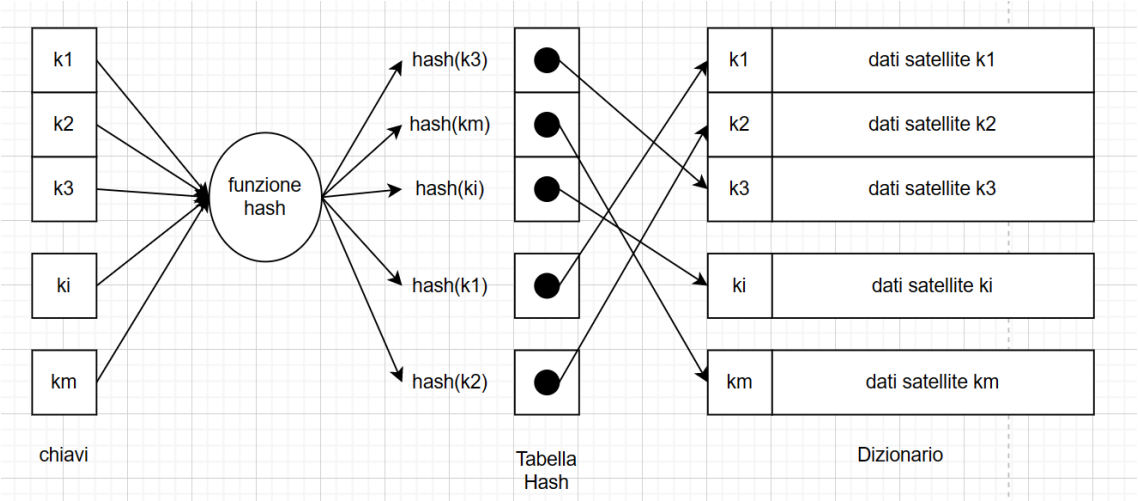

Figura 1: Principio di funzionamento

Supponiamo di avere una chiave key e di volerla memorizzare in una tabella hash di dimensione m. La funzione hash applicata alla chiave restituisce un intero compreso tra 0 e $m-1$: questo valore intero viene quindi utilizzato come indice della tabella hash in cui la chiave key viene memorizzata. In questo modo, quando si cerca la chiave key, la funzione hash viene applicata alla chiave e il valore hash restituito viene utilizzato per accedere direttamente al puntatore all'oggetto identificato dalla chiave.

Quando due o più chiavi nel dizionario hanno lo stesso valore hash, diciamo che è avvenuta una **collisione**; idealmente, vorremmo funzioni hash senza collisioni.

2. Risoluzione delle collisioni

Una tecnica di risoluzione delle collisioni è quella del **concatenamento** (**chaining**): poniamo tutti gli elementi che sono associati allo stesso slot in una lista concatenata: lo slot i contiene un puntatore alla testa della lista di tutti gli elementi memorizzati che corrispondono ad i; se tali elementi non esistono, lo slot i contiene la costante NIL.

Tali liste vengono chiamate anche "liste di trabocco", questo perché le chiavi che causano collisioni vengono "traboccate" dalla posizione originale della tabella hash nella lista concatenata, che viene utilizzata come contenitore per tutte le chiavi che vengono associate a quella posizione della tabella.

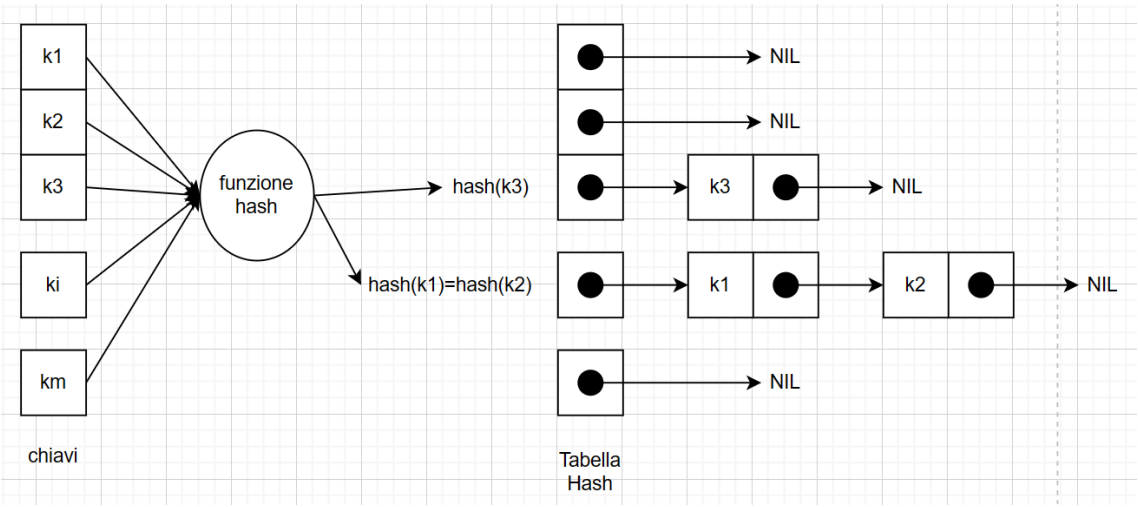

Figura 2: Concatenamento

Come si evince dalla figura, le chiavi $k1$ e $k2$ determinano lo stesso hash e pertanto verrà costruita una lista concatenata a partire dallo stesso slot.

Come sono le prestazioni dell'hashing con chaining?

Definiamo **fattore di carico α** della tabella T il rapporto n/m, ossia il numero medio di elementi memorizzati in una catena, essendo m il numero di slot dove sono memorizzati n elementi (m è anche indicata come la capacità della tabella).

Il comportamento nel caso **peggiore** dell'hashing con chaining è pessimo: tutte le n chiavi sono associate allo stesso slot, creando una lista di lunghezza n:

- il tempo di esecuzione della ricerca è quindi $\Theta(n)$ a cui si somma il tempo per calcolare la funzione hash;
- discorso analogo per la rimozione: nel caso peggiore devo scorrere tutta la lista;

- nel caso dell'inserimento devo invece inserire in testa e dunque la complessità è $\Theta(1)$.

Valutiamo ora il caso medio. Le prestazioni medie dell'hashing dipendono, in media, dal modo in cui la funzione hash distribuisce l'insieme delle chiavi da memorizzare tra gli m slot. Supponiamo che qualsiasi elemento abbia la stessa probabilità di essere associato a uno qualsiasi degli m slot, indipendentemente dallo slot cui sarà associato qualsiasi altro elemento. Questa ipotesi è alla base dell'**hashing uniforme semplice**. Supponiamo inoltre che il valore hash $h(k)$ possa essere calcolato nel tempo $O(1)$.

Abbiamo dunque m slot (da 0 ad $m-1$) nella lista T; se indichiamo con n_i la lunghezza della lista $T[j]$, avremo: $n = n_0 + n_1 + \cdots + n_m - 1$ e il valore medio di n_i sarà:

$$E[n_i] = \alpha = n/m.$$

In altre parole: il valore atteso della lunghezza di una lista è pari ad $\alpha = n/m.$

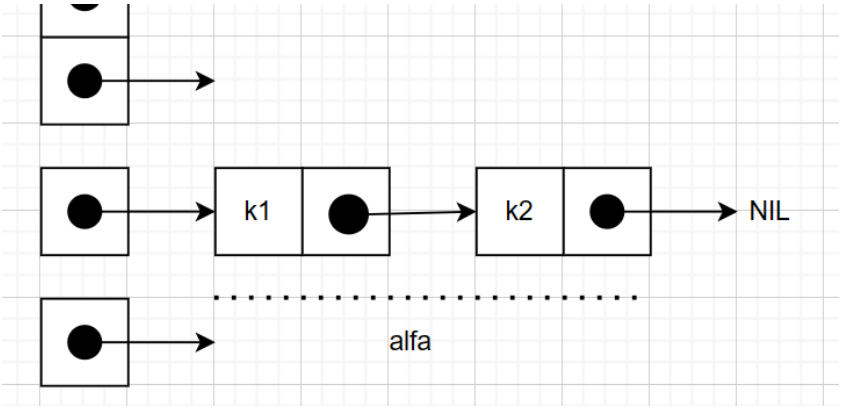

Figura 3: Lunghezza media della lista: alfa

In queste ipotesi, una **ricerca senza successo** richiede un tempo atteso $\Theta(1) + \alpha$.

Nell'ipotesi di hashing uniforme semplice, qualsiasi chiave k non ancora memorizzata nella tabella ha la stessa probabilità di essere associata ad uno qualsiasi degli m slot. Il tempo atteso per ricercare senza successo una chiave k è il tempo atteso per svolgere le ricerche fino alla fine della lista $T[h(k)]$, che ha una lunghezza attesa pari a $E[n_{h(k)}] = \alpha$. Quindi, il numero atteso di elementi esaminato in una ricerca senza successo è α e il tempo totale richiesto (incluso quello per calcolare $h(k)$) è $\Theta(1) + \alpha$.

Nelle stesse ipotesi, una **ricerca con successo** richiede, in media, un tempo $\Theta(1) + \alpha/2$, nell'ipotesi di hashing uniforme semplice; una ricerca con successo tocca infatti in media metà delle chiavi nella lista corrispondente.

Il fattore di carico influenza, dunque, il costo computazionale delle operazioni sulle tabelle hash; se $n = O(m) \rightarrow \alpha = O(1)$. In questa circostanza tutte le operazioni diventano dunque con complessità $O(1)$.

Uno dei problemi nell'utilizzo di liste/vettori di trabocco è ovviamente l'utilizzo di struttura dati complesse, con liste, puntatori, etc. Un'altra tecnica di risoluzione delle collisioni può essere quella dell'**indirizzamento aperto**: l'idea di base è che tutti gli elementi sono memorizzati nella stessa tabella e dunque ogni slot contiene una chiave oppure NIL. In caso di collisione, si cerca di trovare una nuova posizione nella tabella hash in cui memorizzare la chiave utilizzando una tecnica di probing.

Esistono diverse tecniche di probing utilizzate nell'indirizzamento aperto. Le tecniche più comuni sono:

- **Probing lineare**: in questo caso, se la posizione indicata dalla funzione hash è già occupata, si cerca la posizione successiva nella tabella hash, in modo sequenziale, fino a trovare una posizione vuota.
- **Probing quadrato**: in questo caso, se la posizione indicata dalla funzione hash è già occupata, si cerca la posizione successiva nella tabella hash utilizzando una sequenza di incrementi quadrati. Ad esempio, si cerca la posizione $(\text{hash} + 1)^2$, poi $(\text{hash} + 2)^2$, poi $(\text{hash} + 3)^2$, e così via.
- **Probing doppio**: in questo caso, se la posizione indicata dalla funzione hash è già occupata, si cerca la posizione successiva nella tabella hash utilizzando un'altra funzione hash. Ad esempio, si può utilizzare la funzione hash secondaria $hash2(k)$ e cercare la posizione $hash + i * hash2(k)$, dove i è un valore incrementale.

L'indirizzamento aperto può essere più efficiente del chaining in caso di tabelle hash con poche collisioni, poiché evita l'overhead di dover gestire una lista concatenata per ogni posizione della tabella. Tuttavia, la scelta della tecnica di probing e delle funzioni hash deve essere fatta con cura per evitare la formazione di cluster di chiavi nelle posizioni della tabella hash, che potrebbero rallentare l'algoritmo di ricerca e inserimento degli elementi.

Riportiamo di seguito i parametri / indicatori necessari per fare una valutazione della complessità:

- n: numero di chiavi memorizzati in tabella hash
- m: capacità della tabella hash
- $\alpha = n/m$ Fattore di carico

- $I(\alpha)$: numero medio di accessi alla tabella per la ricerca di una chiave non presente nella tabella (ricerca con insuccesso)
- $S(\alpha)$: numero medio di accessi alla tabella per la ricerca di una chiave presente nella tabella (ricerca con successo)

Analizziamo gli scenari:

Metodo	α	$I(\alpha)$	$S(\alpha)$
Indirizzamento diretto semplice	$0 \leq \alpha < 1$	$\frac{(1-\alpha)^2+1}{2(1-\alpha)^2}$	$\frac{1-\alpha/2}{1-\alpha}$
Indirizzamento diretto doppio	$0 \leq \alpha < 1$	$\frac{1}{1-\alpha}$	$-\frac{1}{\alpha}\log_2(1-\alpha)$
Liste di trabocco	$\alpha \geq 0$	$1+\alpha$	$1+\alpha/2$

In termini di fattore di carico, per l'indirizzamento diretto abbiamo il vincolo che questo deve essere compreso tra 0 ed 1 (escluso) mentre per le liste di trabocco non abbiamo restrizioni superiori.

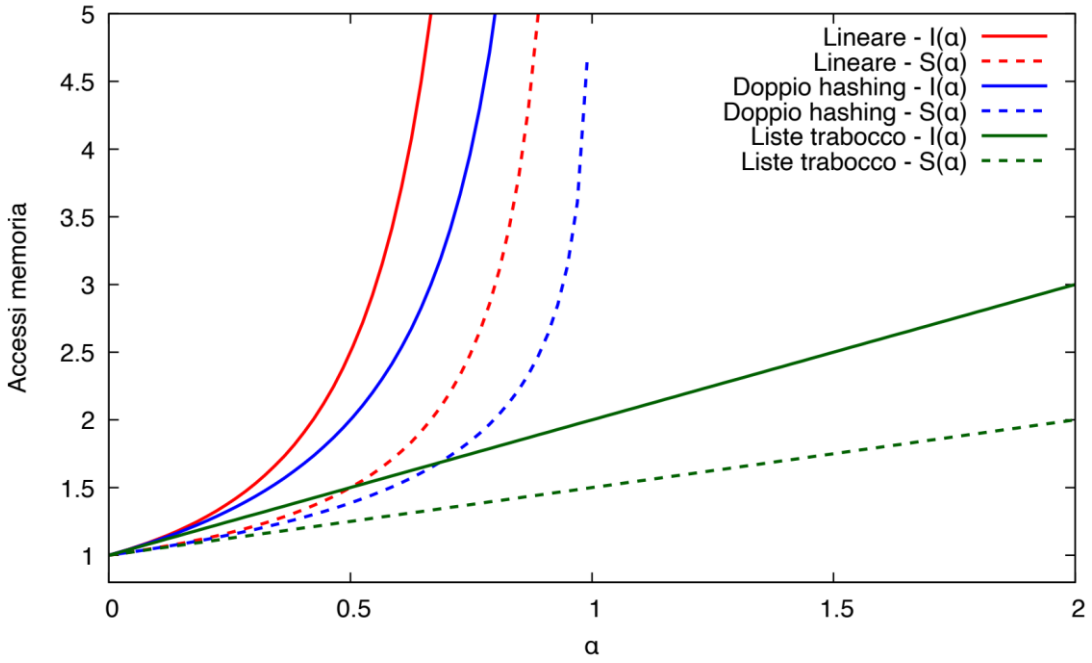

Figura 4: Analisi

Come si evince dal grafico, per valori di fattore di carico inferiori a 0.5, i numeri sono "ragionevoli" ed è per questo che si cerca di restare in quel range. Se $\alpha < 5$ mi trovo in corrispondenza a valori al più di 3 accessi in memoria sia nel caso con successo che con insuccesso e quindi in sostanza i costi sono costanti

3. Hash vs hash crittografico

La differenza principale tra una funzione hash e una funzione hash crittografica è che la seconda è progettata per garantire la sicurezza e l'integrità dei dati, mentre la prima è progettata principalmente per l'efficienza delle operazioni di ricerca e inserimento nella tabella hash.

In particolare, una funzione hash crittografica deve soddisfare alcune proprietà specifiche per garantire la sicurezza dei dati, come ad esempio:

- **Resistenza alla collisione**: la difficoltà di trovare due input differenti che producano lo stesso valore hash;
- **Resistenza alla pre-immagine**: la difficoltà di trovare un input che produca un determinato valore hash; data una funzione hash crittografica $h(x)$, deve risultare difficile trovare un input x' tale che $h(x') = h(x)$;
- **Resistenza alla seconda pre-immagine**: la difficoltà di trovare un input differente che produca lo stesso valore hash di un input noto; data una funzione hash crittografica $h(x)$ e un input x, deve essere difficile trovare un secondo input x' diverso da x che produca lo stesso valore hash. In altre parole, se x è un input noto e $h(x)$ è il suo valore hash, la resistenza alla seconda pre-immagine garantisce che sia difficile trovare un secondo input x' tale che $h(x') = h(x)$ e $x' \neq x$.

Inoltre, le funzioni hash crittografiche devono essere progettate in modo da resistere a diverse tecniche di attacco, come ad esempio la ricerca esaustiva, l'analisi differenziale e l'analisi di codice sorgente.

Le funzioni hash, invece, non devono necessariamente soddisfare le proprietà di sicurezza richieste dalle funzioni hash crittografiche; devono essere invece progettate principalmente per massimizzare l'efficienza delle operazioni di ricerca e inserimento nella tabella hash, minimizzando il numero di collisioni e distribuendo in modo uniforme le chiavi nella tabella.

Ulteriori differenze rilevabili sono:

- **Velocità di calcolo**: le funzioni hash sono progettate per essere molto efficienti dal punto di vista computazionale, in modo da minimizzare il tempo necessario per calcolare i valori hash e gestire le collisioni nella tabella hash. Al contrario, le funzioni hash crittografiche sono progettate per essere molto più lente rispetto alle funzioni hash, in modo da rendere più difficile l'attacco da parte di un aggressore.

- **Dimensione dell'output**: le funzioni hash solitamente producono un output di dimensione fissa, che è specificata al momento della progettazione dell'algoritmo. Ad esempio, una funzione hash potrebbe produrre un output di 32 bit o 64 bit. Le funzioni hash crittografiche, invece, producono spesso un output di dimensione variabile, in modo da poter essere utilizzate per calcolare codici di autenticazione a lunghezza variabile.
- **Standardizzazione**: esistono molte funzioni hash standardizzate, come ad esempio MD5, SHA-1 e SHA-2, che sono utilizzate in molti contesti diversi. Al contrario, le funzioni hash crittografiche sono spesso sviluppate su misura per un'applicazione specifica e non esistono standard universali per queste funzioni.

4. Implementazione

In Python esiste nativamente una struttura dati chiamata **dict** che può essere utilizzata come tabella hash. Il tipo dict di Python implementa una tabella hash basata su una struttura dati nota come "hash table with open addressing", che è una variante dell'indirizzamento aperto.

Per utilizzare una tabella hash in Python, è possibile creare un nuovo oggetto di tipo dict e inserire le coppie chiave-valore utilizzando la sintassi $dizionario[chiave] = valore$, dove "dizionario" è l'oggetto di tipo dict, "chiave" è la chiave da inserire e "valore" è il valore associato alla chiave.

```
# Creazione di una nuova tabella hash
hash_table = {}

# Inserimento di una coppia chiave-valore
hash_table['Alice'] = 25

# Recupero del valore associato a una chiave
age = hash_table['Alice']
print(age)  # Output: 25

# Verifica dell'esistenza di una chiave nella tabella hash
if 'Bob' in hash_table:
    print("Bob's age is", hash_table['Bob'])
else:
    print("Bob is not in the hash table")

# Iterazione attraverso le chiavi della tabella hash
for key in hash_table:
    print(key, hash_table[key])
```

Una possibile implementazione in Python di una tabella hash con liste di trabocco è la seguente:

```
class HashTable:
    def __init__(self, size):
        self.size = size
        self.table = [[] for _ in range(size)]

    def hash_function(self, key):
        return sum(ord(char) for char in key) % self.size

    def insert(self, key, value):
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                pair[1] = value
```

```
                return
        self.table[index].append([key, value])

    def get(self, key):
        index = self.hash_function(key)
        for pair in self.table[index]:
            if pair[0] == key:
                return pair[1]
             raise KeyError(key)
```

Questa implementazione è simile a quella della tabella hash con concatenamento delle collisioni vista in precedenza, ma invece di utilizzare una lista di coppie chiave-valore, utilizza una lista di liste. Ogni lista rappresenta una posizione della tabella hash, e contiene una lista di coppie chiave-valore corrispondenti alle collisioni che si verificano in quella posizione.

La funzione insert viene utilizzata per inserire una nuova coppia chiave-valore nella tabella hash. Se la chiave esiste già nella tabella hash, il valore associato viene aggiornato. Se la chiave non esiste, viene creata una nuova coppia chiave-valore e memorizzata nella lista corrispondente all'indice calcolato dalla funzione hash.

La funzione get viene utilizzata per recuperare il valore associato a una determinata chiave. Se la chiave esiste nella tabella hash, viene restituito il valore associato. Se la chiave non esiste, viene sollevata un'eccezione KeyError.

Ecco un esempio di utilizzo:

```
hash_table = HashTable(10)

hash_table.insert('Alice', 25)
hash_table.insert('Bob', 30)
hash_table.insert('Charlie', 35)

print(hash_table.get('Alice'))       # Output: 25
print(hash_table.get('Bob'))         # Output: 30
print(hash_table.get('Charlie'))     # Output: 35

try:
    hash_table.get('David')
except KeyError as e:
    print("KeyError:", e)  # Output: KeyError: 'David'
```

In questo esempio, viene creata una nuova tabella hash utilizzando la classe HashTable implementata con la tecnica di concatenamento delle collisioni. Vengono inserite tre coppie chiave-valore utilizzando il metodo insert, e viene quindi utilizzato il metodo get per recuperare il valore associato a ciascuna chiave.

Viene quindi utilizzato un blocco try-except per gestire il caso in cui si cerca di recuperare un valore associato a una chiave che non esiste nella tabella hash. In questo caso, viene sollevata un'eccezione KeyError, che viene catturata dal blocco except e gestita stampando un messaggio di errore.

Questa implementazione di una tabella hash con concatenamento delle collisioni può essere utilizzata per memorizzare e recuperare facilmente valori associati a una determinata chiave, anche in presenza di collisioni tra le chiavi.

Di seguito una implementazione in Python di funzioni hash:

```
# Utilizzo della funzione hash "simple_hash"
def simple_hash(key):
    hash_value = 0
    for char in key:
        hash_value += ord(char)
    return hash_value

# Calcolo del valore hash di una stringa
hash_value = simple_hash("Hello World")
print(hash_value)  # Output: 876
```

La funzione simple_hash utilizza una semplice funzione hash che somma i codici ASCII dei caratteri della stringa.

```
# Utilizzo della funzione hash crittografica "md5_hash"
import hashlib

def md5_hash(key):
    hash_object = hashlib.md5(key.encode())
    return hash_object.hexdigest()

# Calcolo del valore hash crittografico di una stringa
hash_value = md5_hash("Hello World")
print(hash_value)  # Output: ed076287532e86365e841e92bfc50d8c

# Utilizzo della funzione hash crittografica "sha256_hash"
def sha256_hash(key):
    hash_object = hashlib.sha256(key.encode())
    return hash_object.hexdigest()

# Calcolo del valore hash crittografico di una stringa
hash_value = sha256_hash("Hello World")
print(hash_value)
“””
Output:
b94d27b9934d3e08a52e52d7
da7dabfac484efe37a5380ee
```

```
9088f7ace2efcde9
“””

# Utilizzo della funzione hash crittografica "sha512_hash"
def sha512_hash(key):
    hash_object = hashlib.sha512(key.encode())
    return hash_object.hexdigest()


# Calcolo del valore hash crittografico di una stringa
hash_value = sha512_hash("Hello World")
print(hash_value)
“””
Output:
cf83e1357eefb8bdf1542850d66d
8007d620e4050b5715dc83f4a921
d36ce9ce47d0d13c5d85f2b0ff83
18d2877eec2f63b931bd47417a81
a538327af927da3e
“””
```

Le funzioni md5_hash, sha256_hash e sha512_hash utilizzano invece rispettivamente le funzioni hash crittografiche MD5, SHA-256 e SHA-512 fornite dalla libreria hashlib di Python.

Per calcolare il valore hash, la stringa di input viene codificata come una sequenza di byte utilizzando il metodo encode(), che converte la stringa in una sequenza di byte utilizzando un determinato encoding (di solito UTF-8). Successivamente, il valore hash viene calcolato chiamando la funzione hash o la funzione hash crittografica corretta e utilizzando il metodo hexdigest() per convertire il risultato in una stringa esadecimale leggibile.

Bibliografia

- Alan Bertossi, Alberto Motresor: Algoritmi e strutture di dati, Città Studi Edizioni, terza edizione;
- C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione;
- Crescenzi, Gambosi, Grossi: Strutture di Dati e Algoritmi, Pearson/Addison-Wesley;
- Sedgewick: Algoritmi in C, Pearson, 2015;
- Cormen Leiserson Rivest Stein-Introduzione Agli Algoritmi E Strutture Dati-Prima Edizione.