UNIVERSITÀ DEGLI STUDI DI FIRENZE

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea in Ingegneria Informatica

Hash Table

Professore:
Ch.mo Prof.
Simone Marinai

Candidato:
Hamidreza Hashemi
Mat. 5996260

Corso:
Algoritmi e Strutture Dati

ANNO ACCADEMICO 2020/2021

Dedicata a coloro che
ci provano ogni giorno.

INDICE

Introduzione **3**

1 Hash con indirizzamento aperto **4**

2 Hash con Concatenamento **7**

3 Conclusione e Confronto **9**

INTRODUZIONE

In questa relazione vediamo insieme due tipi diversi di Tabelle hash che sono hash con concatenamento e indirizzamento aperto. Vedremmo una loro implementazione nel linguaggio python. Queste strutture dati servono a memorizzare diversi valori associati a diverse chiavi. È utile sapere il significato dei termini "Fattore del carico" e "Collisione". Molto spesso le prestazioni di una tabella hash sono fortemente legate anche al cosiddetto fattore di carico calcolato come :

$$\alpha = \frac{n}{m}$$

Questo fattore ci dice quanta probabilità ha un nuovo elemento di collidere con uno già presente nella tabella.

```
# Calcola il fattore di caricamento per ogni inserimento o cancellazione
def __loadCalc(self):
    if self.size == 0:
        self.__loadFactor = 0
    else:
        self.__loadFactor = self.__loadSum / self.size
```

Figura 1: Fattore del carico

CAPITOLO 1

HASH CON INDIRIZZAMENTO APERTO

Questo tipo di hash è basato sulla memorizzazione dei dati nella stessa tabella (senza creare diverse liste aggiunte). Ci sono diverse tecniche comunemente utilizzate per ispezione. Una tecnica comune è la **Ispezione lineare** in cui quando si incontra una collisione non si fa altro che utilizzare l'indice successivo a quello che collide, sino a che non si trovi una casella libera. La funzione hash è basata sulla formula seguente :

$$h(k, i) = h((k) + i) \,\%\, m$$

dove k è la chiave, i è l'indice scorrevole e m è la dimensione della tabella.

```
def insert(self, k, v):
    k = str(k)
    if k in self.keys:
        i = 0
        hashed = (hash(k) + i) % self.size
        while self.bucket[hashed] is not None:
            if self.bucket[hashed].key != k:
                hashed = (hash(k) + i) % self.size
                i += 1
            else:
                self.bucket[hashed].value = v
                self.__collision(True)
                return True
    elif self.__loadFactor == 1.0:
        return False
    else:
        newCell = Cell(k, v)
        i = 0
        hashed = (hash(k) + i) % self.size
        while self.bucket[hashed] is not None:
            hashed = (hash(k) + i) % self.size
            i += 1
        self.bucket[hashed] = newCell
        self.keys.add(k)
        self.__loadSum += 1
        self.__loadCalc()
        return True
```

Figura 1.1: Inserimento in un hash con ispezione lineare

Abbiamo fatto un esperimento in cui viene inizializzato due dimensioni diversi di questa tabella con 100 elementi e 10000 elementi e vengono inseriti diversi dati random con la chiave tra 0 e 10000.

```
D:\Anaconda\python.exe C:/Users/Kaliou/PycharmProjects/Hash/main.py
Linear Hash Table Standard con fattore di carico : 0.6 e n. collisioni : 40
tempo di inserimento : 561900
tempo di ricerca : 4100
Linear Hash Table Grande con fattore di carico : 0.6332 e n. collisioni : 3668
tempo di inserimento : 56052900
tempo di ricerca : 2300
```

Figura 1.2: Esperimento con hash lineare

Come vediamo il numero di collisioni crescono all'aumento della dimensione della tabella. il fattore del carico rimane quasi uguale. In media su 10 esecuzioni abbiamo i seguenti valori :

Hash Lineare (nanosec.)			
T.Inserimento Standard	T.Inserimento Grande	T.Ricerca Standard	T.Ricerca Grande
552000	57934900	3300	4600
555100	61002400	4200	2800
555200	62679400	3500	3300
577300	61602500	3600	2500
549800	60819600	4100	3000
587700	60410400	3900	2600
626300	59611300	3800	1100
599600	61522900	3800	10800
556800	61899600	4200	3300
579500	61154200	4500	2700
avg:573930	avg:60863720	avg:3890	avg:3670

CAPITOLO 2

HASH CON CONCATENAMENTO

In questo tipo di hash, per ogni cella della tabella di hash si fa corrispondere invece di un elemento, una Lista (solitamente una lista concatenata). In questo modo un elemento che collide viene aggiunto alla lista corrispondente all'indice ottenuto.

```
# Se non c'è nessun elemento inizializza un linked list e inserisce il primo elemento
def insert(self, k, value):
    self.__loadSum += 1
    self.__loadCalc()
    hashed = hash(k) % self.size
    if self.table[hashed] is None:
        newLinkedList = LinkedList()
        newLinkedList.insertAtFirst(k, value)
        self.table[hashed] = newLinkedList
    else:
        self.table[hashed].insertAtLast(k, value)
        self.__collision(True)

def get(self, k):
    hashed = hash(k) % self.size
    if self.table[hashed] is not None:
        return self.table[hashed].find(k)

def delete(self, k):
    hashed = hash(k) % self.size
    if self.table[hashed] is not None:
        if self.table[hashed].size > 1:
            self.__collision(False)
        self.table[hashed].remove(k)
```

Figura 2.1: Inserimento in un hash con concatenamento

Anche qui facciamo un esperimento con dimensione 100 e 10000. Abbiamo il seguente output per il numero di collisioni e il fattore del carico:

```
Chained Hash Table Standard con fattore di carico : 0.64 e n. collisioni : 36
tempo di inserimento : 467700
tempo di ricerca : 3400
Chained Hash Table Grande con fattore di carico : 0.6296 e n. collisioni : 3704
tempo di inserimento : 53055300
tempo di ricerca : 1400
```

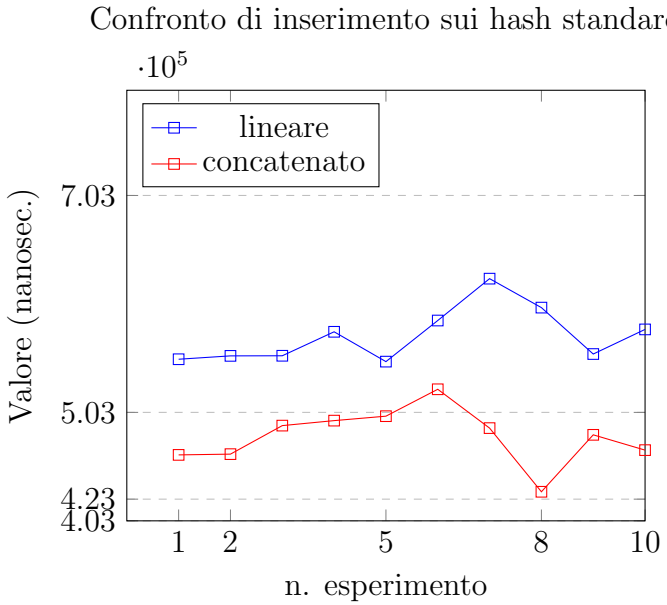
Figura 2.2: Esperimento con hash concatenato

Per i tempi di inserimento e la ricerca abbiamo la seguente tabella :

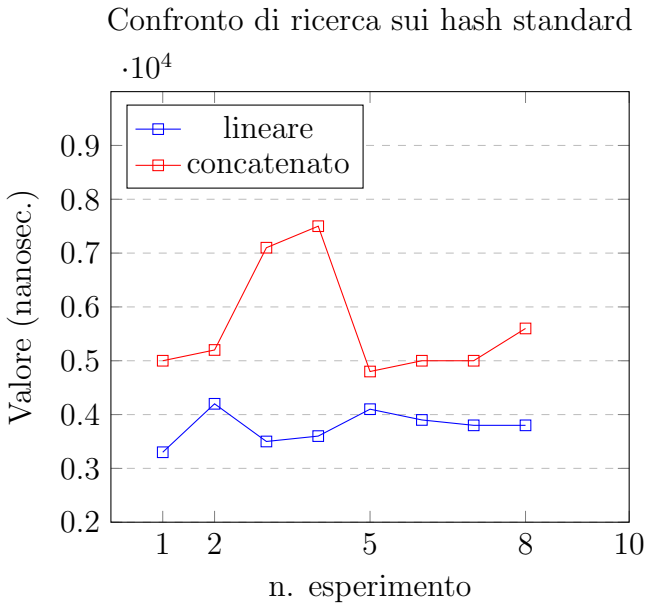
Hash Concatenato (nanosec.)			
T.Inserimento Standard	T.Inserimento Grande	T.Ricerca Standard	T.Ricerca Grande
463800	53483100	5000	1400
695600	53384900	5200	2300
490800	54080300	7100	3400
464500	51659500	7500	1800
495400	53391900	4800	2700
749300	60597300	5000	1400
499500	56169300	5000	2500
524300	53193800	5600	2200
482300	51701300	6400	2700
468200	52239800	3300	3800
533370	53990120	5490	2420

CAPITOLO 3

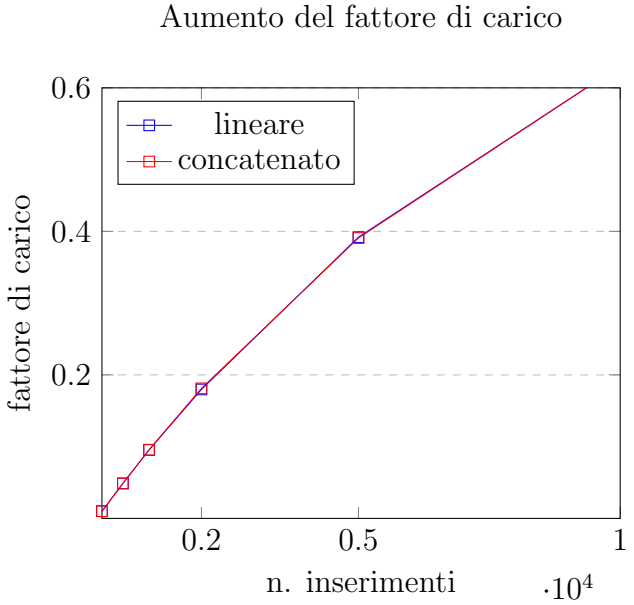
CONCLUSIONE E CONFRONTO

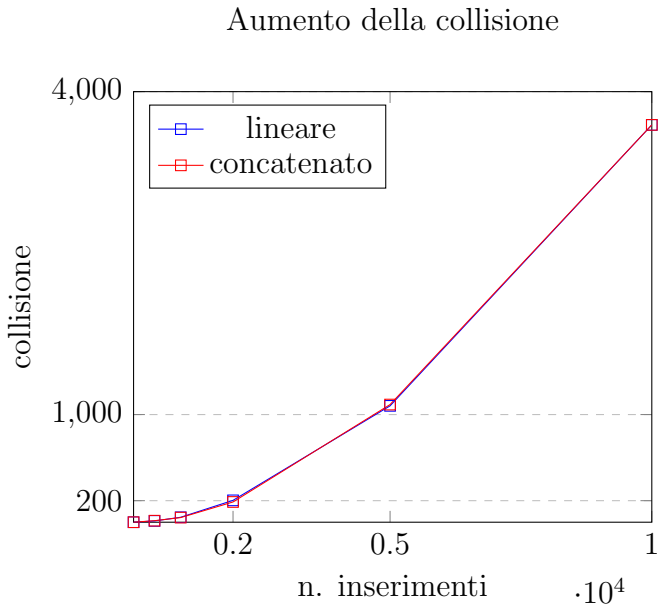


Possiamo concludere che con un fattore di carico intorno a 0.6, il tempo di inserimento in hash lineare è più costoso rispetto a hash con concatenamento.



Invece il tempo di ricerca risulta maggiore in hash concatenato rispetto al hash lineare. Facendo aumentare inserimento, tenendo la dimensione fissa abbiamo un aumento del fattore di carico e la collisione :





In questo esperimento è stato creato un hash di dimensione 10000 elementi. Come vediamo l'aumento del fattore di carico e la collisione sono identici in due tipi diversi di hash e i grafici coincidono.