Tabelle Hash

Edoardo Branchi

February 2021

Contents

1 Introduzione

Una tabella hash è una struttura dati che viene utilizzata per mettere in corrispondenza una chiave con un dato valore.

In questo studio tratteremo e vedremo la differenza nella gestione delle collisioni utilizzando due tecniche, l'hashing con indirizzamento aperto (ispezione lineare) e l'hashing con concatenamento.

Sarà proposta una esposizione del codice utilizzato per creare, cercare e cancellare valori da entrambi i tipi di tabelle e poi seguirà un'analisi sulle prestazioni dei vari algoritmi.

2 Algoritmi utilizzati

2.1 Indirizzamento aperto hashing lineare

Utilizzando questo tipo di indirizzamento, tutti gli elementi sono memorizzati direttamente nella tabella hash, ovvero ogni casella della tabella contiene un elemento o la costante NONE. Quando cerchiamo un elemento, esaminiamo le celle della tabella finché non troviamo l'elemento desiderato o finché non ci accorgiamo che l'elemento non è presente.

Uno dei principali svantaggi è che, visto che gli elementi sono memorizzati direttamente sull'array la tabella può riempirsi fino a non avere la possibilità di salvare nuovi elementi, inoltre è sensibile al fenomeno del clustering, ovvero l'inserimento di tanti elementi in celle contigue fra loro.

Per effettuare gli inserimenti calcoliamo la funzione di hash sotto riportata che ci indicherà in che posizione dell'array sarà salvato l'elemento che andremo a inserire.

2.1.1 Generazione

La prima funzione chiamata durante la generazione è insert-generation(), che prende in ingresso la dimensione della tabella e la dimensione dell'inserimento,controlla che sia realizzabile ,ovvero che il numero di inserimenti sia minore o uguale alla dimensione dell'array, in caso positivo genera la tabella con dei placeholder NONE. Nel ciclo for che segue genera un valore casuale da inserire e passa il controllo a linear-insert().

```
1  def insert_generation(hash_dim,insert_dim):
2      global hash_table
3      if insert_dim>hash_dim:
4          print("Il numero di inserimenti eccede la dimensione della
       tabella, riprova")
5          return False
6
7      hash_table = [None] * hash_dim #GENERAZIONE INSERIMENTI
8
9      for i in range (insert_dim):
10         insert_value = random.randrange(0,100,1)
11         linear_insert(hash_dim,insert_value,hash_table) #chiamata
       per inserire
```

```
12      return True
```

2.1.2 Inserimento

linear-insert() è la funzione che si occupa dell' inserimento vero e proprio nella tabella. Calcola h-k alla riga 4 e 5 utilizzando una funzione lineare, controlla alla riga 6 che il valore alla posizione h-k della tabella hash non sia occupato. Nel caso lo sia vuol dire che e' avvenuta una collisione quindi prova l'inserimento nella posizione successiva incrementando il contatore di h-k e delle collisioni di 1, durante questo scorrimento se h-k giunge alla fine della tabella viene riportato a zero e ricomincia a scorrere in cerca di una posizione libera dove inserire il valore. Quando la trova lo inserisce e termina l'esecuzione.

```
1  def linear_insert(hash_dim, insert_value,hash_table):     #
       INSERIMENTO
2      global collision
3      global linear_index
4      h_k = insert_value % hash_dim                    #calcolo h'(k)
5      h_k = (h_k + linear_index) % hash_dim            #calcolo h(k)= (
       h'(k)+i)mod m
6
7      while hash_table[h_k] != None :                  #se la posizione
        h_k e' occupata incrementa il contatore collisione
8          collision += 1                               #scorre alla
       posizione successiva
9          h_k += 1
10
11         if h_k >= hash_dim:                          #se arriva in
       fondo alla tabella ricomincia dalla prima posizione
12             h_k = 0
13
14     hash_table[h_k]= insert_value
```

2.1.3 Cancellazione

La cancellazione qui riportata prende in ingresso il valore da cancellare. Opera una ricerca nella tabella generata precedentemente per trovare il valore da cancellare, se lo trova lo cancella sostituendo il valore con "NONE"[1] altrimenti riporta l'impossibilità di trovare un valore che non è nella tabella.

```
1  def linear_delete(delete_value):                      #CANCELLAZIONE
2      global hash_table
3      i = linear_search(delete_value)        #cerca il valore da
       cancellare
4
5      if i==None:
6          print("impossibile cancellare")
7
8      else:
9          hash_table[i] = None
10         print("valore cancellato in posizione: ", i,)
```

[1] "NONE" viene inserito come cancellazione per rendere possibile senza errori altri inserimenti successivi alla cancellazione

2.1.4 Ricerca

La ricerca prende in ingresso il valore da ricercare. Si scansiona tutta la tabella, se il valore viene trovato restituisce la posizione del primo di essi[2] altrimenti riporta un messaggio di errore.

Questa è la funzione utilizzata per la ricerca del valore della cancellazione al paragrafo 2.1.4.

```
1  def linear_search(search_value):          #RICERCA
2      global hash_table
3      hash_dim = len(hash_table)
4      found = False
5
6      for i in range(hash_dim):
7
8          if hash_table[i] == search_value:                    #
       ritorna il primo valore corrispondente che trova
9              print("trovato in posizione :  " , i)
10             found = True
11             return i
12
13     if found == False:
14         print ("Valore non presente ")
```

2.2 Hashing con concatenamento

2.2.1 Generazione

Fornisco una dimensione di tabella e la alloco per l'utilizzo nelle successive funzioni. La tabella è una Lista di Liste come visibile alla riga 3.

```
1  def list_creation(hash_dim):
2      global HashTable
3      HashTable = [[] for _ in range(hash_dim)]
```

2.2.2 Inserimento

La funzione di inserimento prende in ingresso il numero di inserimenti da operare sulla tabella e per ogni uno di essi genera in valore e una chiave casuale.

```
1  def insert(num_ins):
2      for i in range(num_ins):
3          rand_keyvalue = randint(0, 100)
4          rand_value = randint(0, 300)
5          hash_key = Hashing(rand_keyvalue)
6          HashTable[hash_key].append(rand_value)
```

L'inserimento chiama la funzione Hashing e poi utilizza il suo risultato per appendere il valore da inserire nel giusto slot della tabella. Hashing(), chiamata da Insert() ritorna il modulo del valore da inserire e la dimensione della tabella.

[2] E' riportata la posizione del valore più vicino all' inizio della tabella

```
def Hashing(rand_keyvalue):
    return rand_keyvalue % len(HashTable)
```

2.2.3 Cancellazione

Analogamente al caso Lineare la cancellazione dapprima opera una ricerca sulla tabella, se trova il valore da cancellare effettua un pop sulla sulla lista e rimuove il valore.

Da notare che la ricerca fornisce una tupla contenente la posizione nella lista all' elemento 0 e la posizione nella lista di lista dell'elemento da cancellare alla posizione 1. Se il valore non è presente nella tabella restituisce un errore.

```
def hash_delete(delete_value):
    index=hash_search(delete_value)

    try:
        HashTable[index[0]].pop(index[1])

    except:
        print("Impossibile cancellare")
```

2.2.4 Ricerca

La funzione di ricerca sfrutta due cicli annidati per scorrere la tabella in "verticale" ed in "orizzontale". Cerca nella Lista di lista se non trova il valore cercato passa alla posizione successiva e scorre la sua lista di lista corrispondente. Se trova il valore cercato restituisce i due indici che lo identificano altrimenti restituisce un errore.

```
def hash_search(search_value):
    for index1,list in enumerate(HashTable):
        for index,element in enumerate(list):
            if element == search_value:
                print("valore trovato in : ",index1," alla
    posizione: ",index+1)
                return (index1,index)
    print("valore non trovato")
```

3 Modalità di esecuzione dei test

Per entrambi i tipi di hashing i test sono stati eseguiti utilizzando una tabella con 2000 slot e numero di inserimenti crescente su cui sono state effettuate 200 ricerche. Il numero di confronti è una media dei confronti effettuati nelle 200 ricerche nominate precedentemente.

Entrambi i test sono stati eseguiti utilizzando gli stessi numeri da inserire e tenendo conto delle collisioni che avvengono man mano che il fattore di carico va ad incrementare.

4 Risultati attesi

4.1 Hashing Lineare

Se assumiamo che i valori inseriti siano equamente distribuiti (in parte non corretto per il clustering che si viene a formare) possiamo aspettarci che il numero medio di confronti per una ricerca senza successo sia $\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$ e che sia $\frac{1}{2}(1 + \frac{1}{(1-\alpha)})$ per una ricerca con successo con $\alpha = m/n$ con m dimensione della tabella e con n numero di elementi gia presenti sulla tabella.

Quindi mi aspetto di vedere il numero di confronti necessari per una ricerca che crescono al crescere di α, con la ricerca senza successo che cresce in maniera più rapida.

In pratica, maggiore sarà il fattore di carico e maggiore sarà il degrado nelle prestazioni della tabella, in particolare perchè la ricerca si ferma quando trovauno slot libero, ma con la tabella molto piena è probabile che abbia da operare molti più confronti prima di trovarne uno o trovare il numero che sta cercando. In generale l'hashing lineare ha come caso peggiore $\mathcal{O}(n)$ e come caso migliore $\mathcal{O}(1)$.

4.2 Concatenamento

Per il concatenamento mi aspetto:

Load factor	Performance
$\alpha \gg 1$	$\mathcal{O}(n)$
$\alpha \approx 1$	$\mathcal{O}(1 + \alpha)$
$\alpha \approx 0$	$\mathcal{O}(1)$

dove $\alpha = m/n$ con m dimensione della tabella e con n numero di elementi gia' presenti sulla tabella.

Il caso medio deriva dall'assunzione che una funzione hash ipotetica piazzerà gli elementi in una tabella in maniera uniformemente distribuita. A livello temporale il caso medio per una ricerca senza successo e una con successo è uguale.

Mi aspetto un numero medio di comparazioni di α per una ricerca senza successo e di $1 + \frac{\alpha}{2}$ per quella con successo. Il numero di comparazioni andrà a divergere all'aumentare del fattore di carico, ma, negli esperimenti che seguono il numero di comparazioni è stato considerato per l'intervallo 0,1 perciò mi aspetto un comportamento simile da entrambi i tipi di ricerca.

A differenza dell' hashing lineare dovrei avere meno degrado all'aumentare del fattore di carico.

5 Risultati dei test

5.1 Hashing Lineare

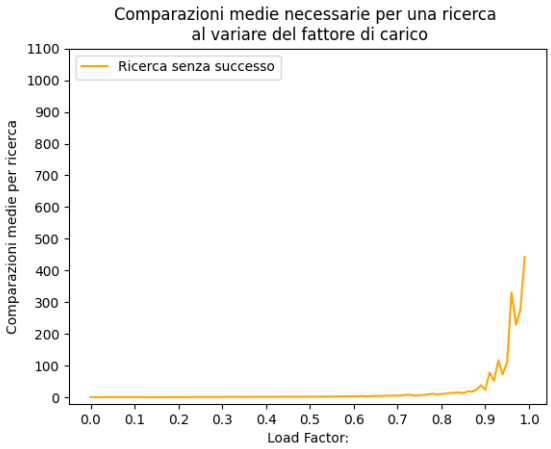

Figure 1: Numero di comparazioni medie necessarie per una ricerca senza successo all'aumentare del fattore di carico con Hashing lineare (2000 slot disponibili)

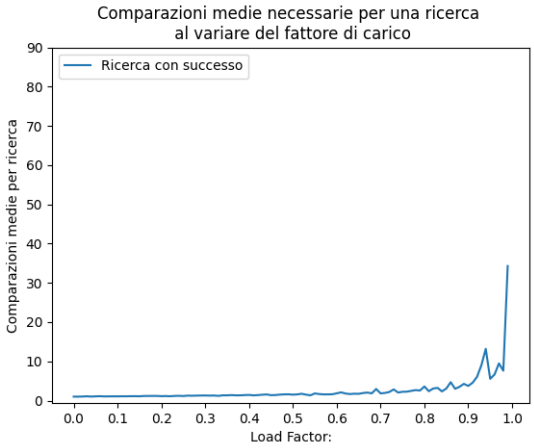

Figure 2: Numero di comparazioni medie necessarie per una ricerca con successo all'aumentare del fattore di carico con Hashing lineare (2000 slot disponibili)

5.2 Concatenamento

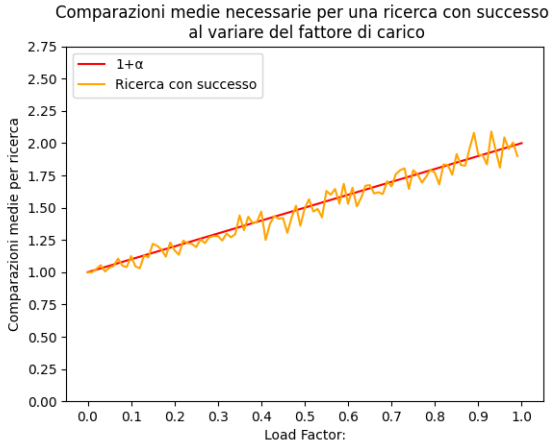

Figure 3: Numero di comparazioni medie necessarie per una ricerca senza successo all'aumentare del fattore di carico con Hashing concatenato (2000 slot disponibili)

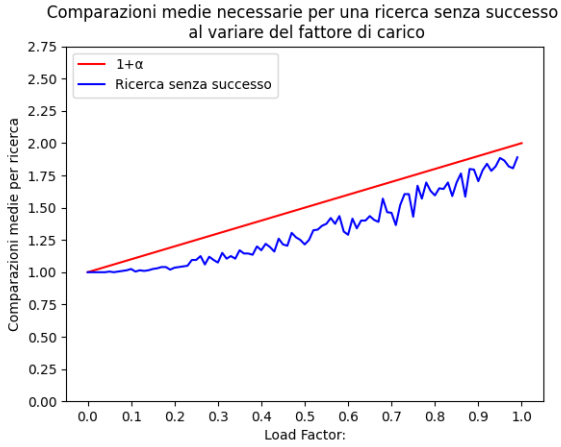

Figure 4: Numero di comparazioni medie necessarie per una ricerca con successo all'aumentare del fattore di carico con Hashing concatenato (2000 slot disponibili)

6 Conclusioni

6.1 Hashing Lineare

Da quello che è stato possibile osservare una tabella Hash con hashing lineare perde molti dei suoi punti di forza all'aumentare del fattore di carico. Una ricerca con la tabella vicina alla saturazione comporterebbe un numero di ispezioni non trascurabile. In particolare con una tabella molto popolata il numero di confronti necessario aumenta in modo particolare se il valore cercato non è presente.

Queste considerazioni ci fanno concludere che i punti di forza nell'utilizzo delle tabelle Lineari si esauriscono velocemente quando la tabella va verso la saturazione e perciò è necessaria una valutazione sui dati prima di scegliere di utilizzare questo tipo di tabelle.

Dal punto di vista della memoria invece, l'hashing lineare non ha bisogno di puntatori per memorizzare i dati perciò con tabelle di dimensioni considerevoli sarebbe possibile in linea teorica risparmiare memoria.

6.2 Concatenamento

A differenza della tabella lineare, nonostante ci sia una degradazione delle performance di inserimento anche in questo caso l'aumento delle collisioni non avviene in maniera esponenziale ma lineare(nell' intervallo preso in considerazione). Probabilmente se il load factor fosse molto maggiore di 1 potrei notare un aumento del numero di confronti anche nell' hashing concatenato, che , comunque risulterebbe più vantaggioso dell' hashing lineare.

Questo di tabella quindi risulta utile nel caso si debbano inserire una grande quantità di dati in una tabella di dimensioni ridotte, [5]. risolvendo anche il problema del clustering che si potrebbe formare in una tabella hash di tipo lineare.

Uno svantaggio di questo metodo di risoluzione delle collisioni risiede nel fatto che per memorizzare i dati è necessario ricorrere alle liste concatenate con un utilizzo maggiore di memoria per memorizzare i puntatori delle liste.

[5] Il fattore di carico in questo tipo di tabelle può abbondantemente superare 1 a differenza dell'hashing lineare