UNIVERSITA' DEGLI STUDI DI PADOVA
DIPARTIMENTO DI INGEGNERIA
DELL'INFORMAZIONE

CUCKOO HASHING - TEORIA E PRATICA

Relatore: **Geppino Prof. Pucci**

Autore: **Grigolo Fabio**

DEPARTMENT OF
INFORMATION
ENGINEERING
UNIVERSITY OF PADOVA

Sommario

Questo lavoro intende esporre una tecnica di hashing di recente invenzione, il cuckoo hashing, che riesce ad avere le stesse performance teoriche del dizionario illustrato da Dietzfelbinger, ma unisce a questa efficienza un migliore utilizzo dello spazio di memoria e una implementazione più semplice.

Il lavoro sarà articolato in due sezioni principali: una prima parte che riguarderà i principi teorici del funzionamento e l'analisi della sua complessità computazionale, e una seconda parte che riguarderà l'implementazione dell'algoritmo in codice Java e test comparativi con algoritmi di hashing classici.

Indice

1 Introduzione **5**

2 Cuckoo hashing: implementazione **6**
- 2.1 *Cuckoo hashing design* 6
- 2.2 *Varianti dello schema originale* 9
- 2.3 *Analisi dell'algoritmo* 10
  - 2.3.1 Premessa 10
  - 2.3.2 Analisi 11

3 Test paralleli con altri algoritmi di hashing **16**
- 3.1 Traduzione delle procedure in Java da pseudocodice 16
  - 3.1.1 procedura di Lookup 16
  - 3.1.2 procedura di Insert 16
  - 3.1.3 procedura di rehashing 18
- 3.2 Richiami teorici su LINEAR PROBING e CHAINED HASHING 19
- 3.3 Test inserimento di n chiavi (eventualmente ripetute) 19
  - 3.3.1 tempi di inserimento di n chiavi con fattore di carico massimo di 0.5 (in μs) 20
  - 3.3.2 tempi di inserimento di n chiavi con fattore di carico massimo pari a 1/3 (in μs) 20
- 3.4 Test *succesfull lookup* (ricerca di chiave con esito positivo) . . 21
  - 3.4.1 tempi di lookup in ns, con load factor massimo di 0.5 . 21
  - 3.4.2 tempi di lookup in ns, con load factor massimo di 1/3 . 21
- 3.5 Test *unsuccesfull lookup* (ricerca di chiave con esito negativo) . 22
  - 3.5.1 tempi di lookup in ns, con load factor massimo di 0.5 . 22
  - 3.5.2 tempi di lookup in ns, con load factor massimo di 1/3 . 22
- 3.6 Test di rimozione di un elemento 23
  - 3.6.1 tempi di delete in ns, con load factor massimo pari a 0.5 23
  - 3.6.2 tempi di delete in ns, con load factor massimo pari a 1/3 23

4 Funzioni di hashing universale **24**
- 4.1 costruzione di una funzione di hashing universale 24
  - 4.1.1 hashing di interi 24
  - 4.1.2 hashing di vettori 24

5 bibliografia **25**

A Implementazione in java **I**

1 Introduzione

Definizione: Un dizionario su un universo U = {0,1,..,N-1} è una funzione $S : U \rightarrow I$ dove gli elementi di I sono chiamati *chiavi*. Il dizionario è inoltre fornito delle seguenti operazioni:

- *Lookup(x)*: cerca un elemento nel dizionario corrispondente ad un valore x appartenente a U;
- *Insert(x,i)*: aggiunge x al dominio di S e imposta *S(x)* al valore i;
- *Delete(x)*: rimuove x dal dominio di *S*.

Il dizionario viene solitamente implementato mediante alberi di ricerca o tabelle hash. Le tecniche più efficienti sia dal punto di vista teorico che pratico sono quelle basate su tabella hash. Gli aspetti cruciali nella creazione di un dizionario sono la possibilità di accedere ai dati in tempo costante (almeno sotto il punto di vista dell'analisi ammortizzata) e limitare l'occupazione di memoria. Mediante il CUCKOO HASHING è possibile ottenere una occupazione di spazio *O(n)*.

Come anticipato nel sommario, l'algoritmo cuckoo hashing ha le stesse proprietà teoriche della struttura dati di Dietzfelbinger [2], ma la comprensione del funzionamento e la sua successiva implementazione sono più semplici. Lo schema ha un tempo di lookup che è nel *worst case O(1)* e un tempo ammortizzato (cioè il tempo medio) costante per gli *Insert*. Inoltre, l'utilizzo di spazio è di *2n* parole, contro le *35n* parole dell'implementazione di Dietzfelbinger. Altra caratteristica distintiva del cuckoo hashing è che l'operazione di *Lookup* richiede sempre al più due accessi alla struttura dati, che possono essere effettuati anche *in parallelo* se la struttura hardware lo consente (oppure con un multithreading java sulla VM).

L'algoritmo usa un insieme di funzioni di hash scelte *casualmente con probabilità uniforme* da una famiglia universale. La definizione di una famiglia universale è data da Carter e Wegman[3].

Definizione: *Una famiglia*$\{h_i\}_{i \in I}$*, :* h_i*:* $U \rightarrow R$*, è* (c,k)-universale *se, per ogni k distinti elementi* x_1,..,$x_k \in U$*, ogni* y_1,...,$y_k \in R$*, ed un* $i \in I$ *scelto con probabilità uniforme,*

$$Pr[h_i(x_i) = y_i, ..., h_i(x_k) = y_k] \leq c/|R|^k$$

2 Cuckoo hashing: implementazione

2.1 *Cuckoo hashing design*

Nel cuckoo hashing, il dizionario è composto da due tabelle hash, che in seguito indicheremo con T_1 e T_2, ognuna di dimensione r. Vengono inoltre definite due funzioni di hash $h_1 : U \rightarrow \{T_1[0], ..., T_1[r-1]\}$,
e $h_2 : U \rightarrow \{T_2[0], ..., T_2[r-1]\}$, dove con U indichiamo l'insieme di tutti i valori inseribili nel dizionario. Ogni chiave $x \in S$ è immagazzinata nella cella $h_1(x)$ di T_1 oppure nella cella $h_2(x)$ di T_2.

A seguire l'implementazione in pseudocodice della procedura di lookup:

function lookup(x)
return $T_1[h_1(x)] = x \vee T_2[h_2(x)] = x$
end

Pagh, in [4], dimostra che, definendo con r la dimensione di T_1 e T_2 e con n il numero delle chiavi contenute in T_1 o in T_2, se $r \geq (1+\epsilon)n$ per una certa costante $\epsilon > 0$ (cioè la tabella T_1 e la tabella T_2 hanno al più $r/(1+\epsilon)$ celle occupate da chiavi), con h_1,h_2 scelte casualmente con distribuzione di probabilità uniforme da una famiglia $(O(1), O(\log n))$-universale, la probabilità che non ci sia modo di organizzare le chiavi di S secondo h_1 e h_2 è $O(1/n)$.

Delete:
La procedura di cancellazione è sostanzialmente un lookup con successiva eliminazione della chiave. Questa operazione si effettua in un tempo costante, se non si considera l'eventualità di ridimensionare le tabelle quando il fattore di carico diventa troppo basso e si ha una utilizzazione dello spazio di memoria poco efficiente.

Insert:
La procedura di inserimento di una nuova chiave è sicuramente la parte più innovativa di questo algoritmo. Infatti in fase di inserimento si utilizza una tecnica chiamata "cuckoo approach". Il funzionamento è il seguente:

- La posizione $T_1[h_1(x)]$ è occupata da una chiave y (cioè $h_1(x) = h_1(y)$)

- x viene posizionata in $T_1[h_1(x)]$, mentre y viene posizionata in $T_2[h_2(y)]$, se la posizione $T_2[h_2(y)]$ risulta occupata da una terza chiave z, la chiave z viene posizionata in $T_1[h_1(z)]$ ...

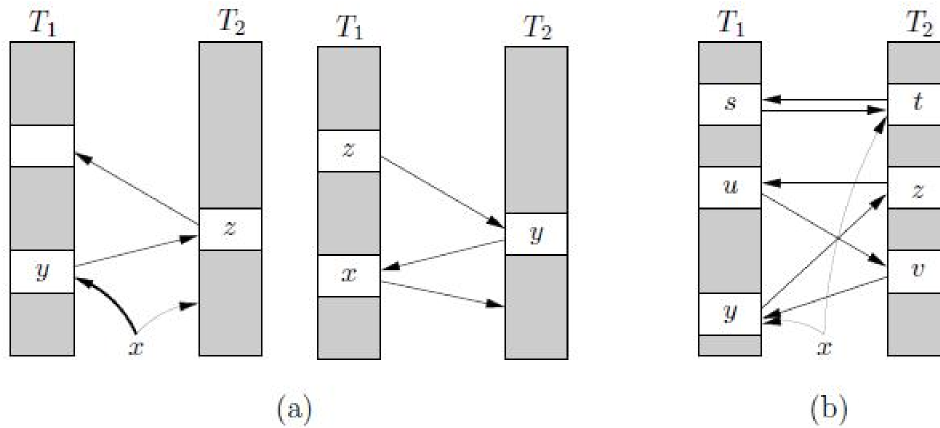

Figura 1: *Procedura di inserimento: le frecce mostrano le varie possibilità di movimento delle chiavi. (a) La chiave x è inserita con successo spostando le chiavi y e z in un'altra posizione. (b) La chiave x non si può inserire per l'insorgere di un ciclo ed il rehash è necessario.*

Dalla spiegazione precedente è evidente che questo procedimento può portare a cicli infiniti, per cui è necessario impostare un valore *MaxLoop* che indichi il numero massimo di iterazioni consentite. Nel caso di raggiungimento del limite superiore di iterazioni consentite da *MaxLoop*, si opera un rehashing delle chiavi, usando due nuove funzioni di hash h_3 ed h_4, e si cerca di reinserire nuovamente tutte le chiavi. Non c'è bisogno di allocare due nuove tabelle per effettuare l'operazione di rehash: basta semplicemente accedere a tutti gli indici di T_1 e T_2, cancellare dalle celle ogni chiave che si incontra e reinserirla mediante una procedura di insert che usi h_3 ed h_4 quali funzioni di hash.

Usando la simbologia $x \leftrightarrow y$ per indicare che i valori di x e y vengono scambiati, e il simbolo $\perp$ per indicare la cella vuota, ecco lo pseudocodice della procedura insert:

procedure insert(x):

if lookup(x) **then return**
loop MaxLoop **times**
$x \leftrightarrow T_1[h_1(x)]$
if $x = \perp$ **then return**
$x \leftrightarrow T_2[h_2(x)]$
if $x = \perp$ **then return**
end loop
rehash();
insert(x);
end

La procedura di insert così strutturata si utilizza solo se è verificata la condizione $r \geq (1 + \epsilon)n$ sia su T_1 che su T_2. Quando il fattore di carico non è conosciuto a priori si deve effettuare un test per capire se la condizione precedente è ancora valida: in caso contrario si dovrà utilizzare, prima dell'inserimento della chiave, una procedura di ridimensionamento delle tabelle. Per eseguire il ridimensionamento si dovranno allocare due nuove tabelle T_3 e T_4 di dimensione $2r$, e creare due nuove funzioni di hash $h_3 : U \to \{T_3[0], ..., T_3[(2r)-1]\}$, e $h_4 : U \to \{T_4[0], ..., T_4[(2r)-1]\}$. Tutte le chiavi in T_1 e T_2 vengono inserite in T_3 o T_4 con l'ausilio delle nuove funzioni di hash. A questo punto si utilizzano T_3 e T_4 come tabelle del dizionario, mentre T_1 e T_2 vengono cancellate.

Dalla configurazione della struttura dati (2 tabelle, entrambe formate da r celle), si nota che le chiavi possono essere inserite in r^2 possibili combinazioni differenti. Una buona norma di implementazione della procedura di insert dovrebbe provvedere ad un *rehash forzato* se si giunge a r^2 inserimenti consecutivi senza un rehash.

2.2 *Varianti dello schema originale*

- Una prima variante, creata allo scopo di rendere l'inserimento leggermente più rapido, si ottiene rimuovendo il lookup iniziale nella procedura *Insert*. Ovviamente così facendo il codice perde di robustezza, e questa scelta è auspicabile se e solo se il dominio di S è costituito da elementi distinti.

- Si nota che la procedura di lookup effettua prima la ricerca di x in T_1 ed in seguito su T_2. Dunque un elemento giacente in T_1 viene trovato più rapidamente di uno giacente in T_2. Questo fenomeno può essere sfruttato per incrementare le prestazioni della procedura di lookup sequenziale costruendo una tabella T_1 *più grande* di T_2, anche se questa configurazione aumenta sensibilmente i tempi di inserimento.

2.3 *Analisi dell'algoritmo*

2.3.1 Premessa

Si chiama **algoritmo online***(online algorithm)* un algoritmo che analizza l'input "*pezzo per pezzo*" in modo seriale, cioè analizza l'input nell'ordine in cui gli viene fornito, senza avere a disposizione la totalità dello stesso dall'istante iniziale della sua esecuzione. Viceversa, se un algoritmo ha a disposizione l'intero input fin dall'inizio e può processarlo in un qualsiasi ordine, viene detto **algoritmo offline***(offline algorithm)*.

Ad esempio, un algoritmo offline può essere il *quicksort*, che è un algoritmo di ordinamento di n chiavi.

Nell'analisi degli algoritmi online e probabilistici si opera un procedimento chiamato **analisi** *(competitive analysis)*, con il quale le performance di un *algoritmo online* vengono comparate con le performance di un *algoritmo offline* ottimizzato che conosca dall'istante iniziale la completa sequenza di input.
A differenza dell'*analisi del caso peggiore*, nell'analisi vengono considerati sia i casi computazionalmente più onerosi *(worst cases)*, sia i casi che richiedono minor tempo; per decidere se un caso è worst case o no ci si riferisce all'analisi dell'algoritmo offline considerato per l'analisi.
In questo tipo di analisi si immagina un "avversario" *(adversary)*, che sceglie a piacere i dati da inserire, in modo da massimizzare tempo necessario alle elaborazioni da parte dell'algoritmo online.

Gli "avversari" possono essere di diversa abilità:

- **oblivious adversary:** l'avversario più debole, conosce l'algoritmo ma non gli esiti dei calcoli probabilistici;
- **online adversary:** l'avversario prende la sua decisione prima di conoscere la decisione presa dall' algoritmo;
- **offline adversary:** l'avversario conosce tutto, anche i valori delle variabili aleatorie ottenute da calcoli di tipo probabilistico.

2.3.2 Analisi

Nel nostro caso l'analisi verrà effettuata mediante l'utilizzo dell'*oblivious adversary model.* L'analisi degli inserimenti si svilupperà in tre parti:

1. verranno mostrate le peculiarità del comportamento della procedura di inserimento.
2. verrà derivato un limite sulla probabilità che la procedura di inserimento usi al più t inserimenti.
3. alla fine verrà dimostrato che la procedura impiega un tempo costante ammortizzato.

Comportamento della procedura di inserimento

Il caso più semplice si ha quando nella procedura di inserimento viene coinvolta al più una cella. In questo caso si ha una sequenza di chiavi $x_1, x_2, ..., x_N$ tutte diverse le une dalle altre, si inserisce dunque x_1 in T_1 e si spostano le rimanenti chiavi della sequenza da una tabella all'altra.

Altrimenti può accadere che ad un certo punto la procedura di inserimento ritorna in una cella della tabella hash precedentemente visitata. La chiave x_i viene a trovarsi in corrispondenza con la chiave x_j (naturalmente con $i < j$). A questo punto la chiave x_i viene a trovarsi senza cella per la seconda volta, e viene rimessa nella posizione occupata inizialmente. Conseguentemente anche tutte le altri chiavi della sequenza contenute nel ciclo, e cioè quelle di indice $x_{i-1}, ..., x_2$ tornano nelle posizioni iniziali.

Se ad un certo punto una chiave senza cella x_l trova un collocamento in una cella vuota la procedura di inserimento si conclude, altrimenti si arriva a *MaxLoop* iterazioni e si procede al rehashing.

Lemma 1: *Supponiamo che la procedura di inserimento non entri in un ciclo infinito. Allora per ogni prefisso della sequenza di chiavi $x_1, x_2, ..., x_p$, deve esserci una sottosequenza di almeno $p/3$ chiavi consecutive senza ripetizioni, con primo termine della sequenza dato dalla chiave che si sta per inserire.*

Dimostrazione. Nel caso in cui la procedura di inserimento non ritorni mai in una cella precedentemente visitata, si ha una sequenza di p chiavi

distinte che cominciano da x_1.
Nel caso in cui $p \geq i + j$, almeno una delle due sequenze $x_1, ..., x_{j-1}$ e $x_{i+j-1}, ..., x_p$ deve avere lunghezza di almeno $p/3$.
Infine, nel caso in cui $p < i + j$, abbiamo che le prime $j - 1 \geq \frac{i+j-1}{2} \geq p/2$ chiavi formano la sequenza cercata.

Valore atteso del numero di iterazioni della procedura di insert

Ora analizzeremo, nei vari casi, la probabilità che la procedura di inserimento compia almeno t iterazioni. Per $t > MaxLoop$ la probabilità è banalmente nulla.

Per $t < MaxLoop$ consideriamo il caso in cui le funzioni di hashing siano $(1, MaxLoop)$-universali.

Definizione: Data una variabile aleatoria x discreta si definisce *valore atteso* o *media* la seguente quantità:

$$E[x] = \sum_x xp(x)$$

dove la somma (finita o non) è estesa a tutti i possibili valori della variabile aleatoria.

Inserimento con ciclo

La prima situazione che si deve considerare è quella in cui l'inserimento entra in un ciclo infinito. In questa situazione sia $v \leq l$ il numero di chiavi distinte senza cella. Il ciclo infinito può essere formato in al più $v^3 r^{v-1} n^{v-1}$ combinazioni possibili, dove questo numero è dato da:

- un coefficiente v^2 che indica *tutti* i possibili valori assunti dalla coppia *(i,j)*;
- un coefficiente v per i possibili valori del termine x_l (ricordiamo che x_l è l'ultimo termine della serie, quello cioè che *non* viene inserito per il raggiungimento delle t iterazioni);
- r^{v-1} possibili combinazioni di celle utilizzabili (infatti sono $v - 1$ elementi collocabili in r celle);
- n^{v-1} scelte di combinazioni di valori di chiavi possibili ($v - 1$ perché la v-esima è la chiave x_l, già stabilita).

Dato che $v \leq MaxLoop$, la funzione è(1,v)-universale.

Per la definizione di funzione (1,v)-universale, la probabilità che si verifichi ognuna delle combinazioni precedenti è r^{-2v} (eventi *equiprobabili*). Sommando i possibili valori di v (non è possibile che si crei un ciclo di meno di 3 elementi), e ricordando che $r > (1+\epsilon)n$, abbiamo che la probabilità che durante l'inserimento si crei un ciclo è data da:

$$\sum_{v=3}^{l} v^3 r^{v-1} n^{v-1} r^{-2v} \geq \frac{1}{rn} \sum_{v=3}^{\infty} v^3 \left(\frac{n}{r}\right)^v = O(1/n^2)$$

Questo perché la serie a secondo termine è una serie a termini positivi (si può il criterio della radice), considerando la serie con v da zero a infinito (se converge questa serie converge anche la serie da noi considerata, che è un suo resto).

$$\sum_{v=0}^{\infty} v^3 \left(\frac{n}{r}\right)^v \leq \sum_{v=0}^{\infty} v^3 \frac{1}{(1+\epsilon)^v}$$

Applico ora il criterio della radice:

$$\lim_{v \to \infty} \sqrt[v]{\frac{v^3}{(1+\epsilon)^v}} = \frac{1}{(1+\epsilon)} < 1$$

La serie converge, abbiamo dimostrato che la complessità è $O(1/n^2)$.

inserimento con numero di iterazioni minore di MaxLoop (nessun ciclo)

Il secondo caso è quello in cui una si ha sequenza di $v \geq \lceil (2t-1)/3 \rceil$ chiavi, chiamate $b_1, ..., b_v$, con b_1 chiave da inserire, tale che per una delle coppie (β_1,β_2) = (1,2) , (β_1, β_2) = (2,1):

$h_{\beta_1}(b_1) = h_{\beta_1}(b_2)$, $h_{\beta_2}(b_2) = h_{\beta_2}(b_3)$,
$h_{\beta_1}(b_3) = h_{\beta_1}(b_4)$, $h_{\beta_2}(b_4) = h_{\beta_2}(b_5)$, ... (*)

Cioè ogni elemento b_i andrà ad occupare la posizione occupata dall'elemento b_{i+1}, per $i \in [2, ..., v]$, mentre la chiave b_v occuperà una cella vuota.

Una volta fissato b_1 ci sono al più n^{v-1} possibili modi di creare una sequenza di v chiavi distinte scelte tra un gruppo di n elementi. Per ognuna delle n^{v-1} possibili sequenze di chiavi e per ognuna delle due possibili scelte di (β_1, β_2), la probabilità che si verifichi una delle situazioni descritte dall'equazione (*) è superiormente limitata da $r^{-(v-1)}$ (questo perché le funzioni

di hashing sono (1,MaxLoop)-universali e quindi anche (1,v-1)-universali). La probabilità che si verifichi una situazione tale è data da :

$$2(n/r)^{v-1} \leq 2(1+\epsilon)^{-(2t-1)/3+1}$$

Conclusione dell'analisi:
Il numero di iterazioni attese per ogni inserimento è dato dunque da:

- una iterazione di base (inserimento con cella vuota);
- la somma del numero di tentativi aggiuntivi dovuti ai due casi particolari precedenti

$$E[x] = 1 + \sum_{t=2}^{MaxLoop} \left(2(1+\epsilon)^{-(2t-1)/3+1} + O(1/n^2)\right) \tag{2}$$

$$\leq 1 + O(\frac{MaxLoop}{n^2}) + 2\sum_{t=0}^{\infty} \left(\frac{1}{1+\epsilon}\right)^{(\frac{2t-1}{3})+1}$$

$$= 1 + O(\frac{MaxLoop}{n^2}) + 2(1+\epsilon)^2 \sum_{t=0}^{\infty} \frac{1}{(1+\epsilon)^{\frac{2t}{3}}}$$

$$= O\left(1 + \frac{1}{1-(1+\epsilon)^{-2/3}}\right) = O\left(1 + 1/\epsilon\right)$$

Costo dei rehashing forzati

Questo è un costo da considerarsi nel caso in cui la procedura di inserimento compia un numero di iterazioni superiore a *MaxLoop*. Dall'analisi precedente si sa che questo avviene con probabilità $O(1/n^2)$. Impostando $MaxLoop$ al valore di $\lceil 3\log_{1+e} r \rceil$ si ottiene che la probabilità di dover effettuare un rehashing è data da

$$2(1+\epsilon)^{-(2MaxLoop-1)/3+1} = O(1/n^2)$$

In particolare, la probabilità che durante una procedura di rehash l'inserimento di n numeri vada a buon fine (cioè non sia necessario chiamare per ricorsione un ulteriore rehash), è $1 - O(1/n)$. Il tempo che si ci aspetta per ogni singolo inserimento è $O(1)$, quindi il tempo necessario a cercare di inserire tutte le chiavi è $O(n)$. Se un inserimento fallisce durante la procedura di rehash, un ulteriore rehash viene lanciato ricorsivamente. Dato che la probabilità di dover effettuare una procedura di rehash è sicuramente (per definizione di probabilità) un numero minore di 1, il costo totale di un rehashing è $O(n)$. Quindi, per ogni inserimento, il tempo aggiuntivo dovuto al rehashing è $O(1/n)$.

Bisogna inoltre considerare l'ulteriore rehash che viene lanciato quando si hanno r^2 inserimenti consecutivi senza rehashing. Dato che anche il costo di questo ulteriore rehash è $O(n)$, per ogni inserimento il tempo ulteriore dovuto a questo rehash è $O(1/n)$.

A conti fatti, una singola procedura di inserimento ha un costo di

$$O\left(1 + 1/\epsilon\right) + O(1/n)$$

Eseguendo la somma su n termini, si ottiene che il tempo di n inserimenti è $O(n)$. Il tempo *ammortizzato* della procedura di inserimento è quindi superiormente limitato da una costante.

3 Test paralleli con altri algoritmi di hashing

In questa sezione verranno presentati i risultati ottenuti confrontando le prestazioni dell'algoritmo del cuckoo hashing con altre tecniche di hashing: la tecnica del LINEAR PROBING e quella del CHAINED HASHING.

3.1 Traduzione delle procedure in Java da pseudocodice

L'implementazione adottata per i test è la versione standard del Cuckoo Hashing, con Lookup sequenziale. Un limite di questa implementazione rispetto a quella teorica analizzata al punto 2 è l'impossibilità del calcolatore di generare numeri casuali con probabilità uniforme. Nel nostro caso su è utilizzata la classe *java.util.Random* per la generazione dei numeri, che però sono pseudo-casuali.

3.1.1 procedura di Lookup

```
1      public boolean lookup(int o)
2      {
3          int k1 = integerHashing(o,H1);
4          int k2 = integerHashing(o, H2);
5          return ( T1[k1] == o)  ||  (T2[k2] == o);
6      }
```

La procedura di Lookup, dopo aver ricavato gli indici mediante le due funzioni di hash, esegue un controllo sequenziale sulla prima e poi sulla seconda tabella.

3.1.2 procedura di Insert

```
1      public void insert(int o)
2      {
3          if(o == 0)
4              return;
5          int swap;
6          boolean done = false;
7         if ( ((double) size1/T1.length) >= MAXIMUM_LOAD_FACTOR ||
8              ((double) size2/T2.length) >= MAXIMUM_LOAD_FACTOR)
9              resize();
10        if(getInsertions() > (double) Math.pow(T1.length,2))
11         rehash();
12         if (!lookup(o))
```

```
13          {
14                  int x = o;
15              for(int j = 0; (j < MaxLoop) && (!done); j++)
16              {
17                  swap = T1[integerHashing(x, H1)];
18                  T1[integerHashing(x, H1)] = x;
19                  x = swap;
20                  if (x == 0)
21                  {
22                      size1++;
23                      done = true;
24                  }
25                   if(!done)
26                   {
27                     swap = T2[integerHashing(x, H2)];
28                     T2[integerHashing(x, H2)] = x;
29                     x = swap;
30                  }
31                  if((x == 0) && (!done))
32                  {
33                      size2++;
34                      done = true;
35                  }
36              }
37              if(!done)
38              {
39                  rehash();
40                  insert(x);
41              }
42              insertions++;
43          }
44      }
```

Alle righe 7 e 8 viene effettuato un controllo del fattore di carico, con eventuale ridimensionamento delle tabelle T1 e T2. Deve essere usata una variabile ausiliaria (definita col nome di *swap*), necessaria per effettuare appunto il cambio di posto tra la variabile e la cella dell'array. A riga 10 si vuole controllare se si sono effettuati r^2 inserimenti senza *rehasing*. Da riga 12 a 44 invece viene sviluppata la procedura analizzata in precedenza mediante pseudocodice.

La variabile *insertions* viene incrementata a seguito di un inserimento senza rehashing successivo.

3.1.3 procedura di rehashing

```
1     private void rehash()
2     {
3      H1 = new HashValues(random.nextInt(2048)+ 1,
4         random.nextInt(2048)+ 1, p, T1.length);
5      H2 = new HashValues(random.nextInt(2048)+1,
6         random.nextInt(2048)+ 1, p, T2.length);
7      size1 = 0;
8      size2 = 0;
9      insertions = 0;
10     int[] t1 = T1;
11     int[] t2 = T2;
12     T1 = new int[t1.length];
13     T2 = new int[t2.length];
14     for(int i = 0; i < t1.length; i++)
15     {
16        if(t1[i] != 0)
17        {
18            insert(t1[i]);
19        }
20
21    }
22      for(int i = 0; i < t2.length; i++){
23        if(t2[i] != 0)
24        {
25            insert(t2[i]);
26        }
27     }
28    }
```

Vengono istanziati due oggetti *HashValues*, che in questa implementazione raccolgono i parametri necessari per generare due funzioni di hashing universali. Le funzioni di hashing universali richiedono due numeri scelti casualmente (qui scelti mediante un metodo statico della classe *Random*), un numero primo di valore maggiore alla cardinalità delle chiavi (in questo caso è stato scelto il 2003), e la dimensione della tabella su cui effettuare l'hash. Per dettagli riguardo al funzionamento di queste funzioni rimandiamo alla sezione 4, dove è possibile visionarne una implementazione completa.

3.2 Richiami teorici su LINEAR PROBING e CHAINED HASHING

Analizziamo ora le caratteristiche di funzionamento di questi due algoritmi.

- CHAINED HASHING: non è altro che un array di liste concatenate a cui viene aggiunto un elemento in caso di collisione.
- LINEAR PROBING: questa tecnica funziona mediante una ricerca lineare di una posizione libera all'interno dell'array. Quando si verifica una collisione in corrispondenza della cella i-esima, si cerca di posizionare la chiave nella cella *i+1*-esima, se anche questa risulta occupata si cercherà di posizionare l'elemento nella cella *i+2*-esima e così via.

I test sono stati effettuati su macchina virtuale Java, e abbiamo utilizzato solo numeri interi senza segno (questo per garantire un tempo di hashing costante). Sono stati testati i tempi di inserimento di un set di numeri da 32 a 2048 elementi (eventualmente con valori ripetuti), il tempo di lookup (sia nel caso di HIT che di MISS), e il tempo di rimozione. Inoltre sono state fatte svariate prove per ogni set di valori per verificare come la scelta casuale dei numeri utilizzati per generare la funzione di hash potesse influire sulle prestazioni.
In ogni test effettuato i tre algoritmi hanno lo stesso fattore di carico massimo. Le funzioni di hashing utilizzate per gli algoritmi CHAINED HASHING e LINEAR PROBING dipendono da un solo parametro scelto casualmente al momento dell'inizializzazione della classe.

funzione di hashing di LINEAR PROBING e CHAINED HASHING

$$\mathbf{h}(\mathbf{x}) = ((\mathbf{ax}) \mod \mathbf{p}) \mod \mathbf{l}$$

Dove con x indichiamo il valore, con a un intero casuale, con p un numero primo maggiore della cardinalità dell'universo delle possibili chiavi, con l la dimensione della tabella.

3.3 Test inserimento di n chiavi (eventualmente ripetute)

In questa sezione verrà testato il funzionamento dei tre algoritmi nel caso di inserimento di set di chiavi (con ripetizioni), generate casualmente con algoritmo pseudorandom.

3.3.1 tempi di inserimento di n chiavi con fattore di carico massimo di 0.5 (in μs)

Chiavi	CUCKOO	L.PROBING	C.HASHING
2048	52406	16678	19035
1024	8186	5665	8228
512	5577	2167	2513
256	1261	3457	1084
128	542	472	536
64	281	263	238
32	171	134	135

3.3.2 tempi di inserimento di n chiavi con fattore di carico massimo pari a $1/3$ (in μs)

Chiavi	CUCKOO	L.PROBING	C.HASHING
2048	56689	15385	19239
1024	9183	4153	7573
512	5795	2328	2430
256	1304	3462	1291
128	582	457	477
64	275	258	237
32	157	125	134

Si vede che il tempo di inserimento nel dizionario con che utilizza CUCKOO HASHING è mediamente superiore a quello necessario per gli altri due algoritmi, anche se solo per una costante moltiplicativa. Questo può essere dovuto a diversi fattori:

- I numeri usati per comporre le funzioni di hashing del CUCKOO non sono casuali con probabilità uniforme, ma pseudorandom, e quindi la probabilità di dover effettuare un rehashing diventa maggiore di $O(1/n)$;

- La generazione dei numeri casuali da parte della classe *java.util.Random* richiede un tempo che può arrivare anche a 1 microsecondo; questo tempo va ad aumentare il tempo necessario al rehashing (che richiede ogni volta la generazione di 4 numeri casuali).

3.4 Test *succesfull lookup* (ricerca di chiave con esito positivo)

In questa sezione verrà verificata la velocità i accesso ad una chiave già presente nella struttura dati.

3.4.1 tempi di lookup in ns, con load factor massimo di 0.5

Chiavi	CUCKOO	L.PROBING	C.HASHING
2048	6844	7752	8730
1024	2514	2864	4121
512	2515	2305	2723
256	2165	2235	2794
128	2165	2095	2444
64	2025	2095	2375
32	2025	2165	2305

3.4.2 tempi di lookup in ns, con load factor massimo di $1/3$

Chiavi	CUCKOO	L.PROBING	C.HASHING
2048	6915	7054	7613
1024	2235	2025	2864
512	2375	2375	3212
256	2304	2235	3213
128	2095	1956	2934
64	1955	2025	2584
32	1956	1956	2654

I dati sperimentali verificano le considerazioni teoriche: il tempo di accesso ad una chiave con CUCKOO HASHING è costante, mentre nel LINEAR PROBING e nel CHAINED HASHING è direttamente proporzionale al fattore di carico. Infatti in entrambi i casi analizzati il dizionario basato su CUCKOO HASHING è mediamente più veloce. Il fatto di aver utilizzato una procedura di lookup *sequenziale* e non *parallelo* però può portare a delle eccezioni: infatti un elemento potrebbe trovarsi sulla seconda tabella e sarebbero necessari due accessi, mentre si sa che, quando non si sono verificate collisioni, sia il LINEAR PROBING che il CHAINED HASHING necessitano di un solo accesso.

3.5 Test *unsuccesfull lookup* (ricerca di chiave con esito negativo)

In questa sezione verrà verificata la velocità di ricerca di una chiave mancante nel dizionario.

3.5.1 tempi di lookup in ns, con load factor massimo di 0.5

Chiavi	CUCKOO	L.PROBING	C.HASHING
2048	2025	2165	2235
1024	2025	2235	2235
512	1886	1956	1956
256	1956	1955	2025
128	2026	2096	2025
64	1955	1956	1956
32	2025	2235	2095

3.5.2 tempi di lookup in ns, con load factor massimo di $1/3$

Chiavi	CUCKOO	L.PROBING	C.HASHING
2048	2165	2026	1955
1024	1956	1956	2165
512	2095	1886	1816
256	1956	1886	1956
128	1956	1886	2933
64	1955	2025	2584
32	2025	1955	1886

Anche in questo caso CUCKOO HASHING ha un tempo di rimozione costante (al più due accessi). CUCKOO HASHING è sicuramente migliore nel caso di *load factor* che si attesta a valori intorno al 50% o superiori, cioè quando iniziano ad aumentare le collisioni in rapporto agli inserimenti totali, mentre con un fattore del 33% non si riscontrano grandi differenze (dell'ordine dei 100 *ns*). Infatti quando le collisioni aumentano l'algoritmo di ricerca del CHAINED HASHING deve scorrere per intero liste la cui lunghezza aumenta proporzionalmente con il fattore di carico. Allo stesso modo un aumento delle collisioni sotto LINEAR PROBING genera sequenze di dati adiacenti sempre più lunghe.

3.6 Test di rimozione di un elemento

In questa sezione verrà verificata la velocità di rimozione di una chiave presente nel dizionario. I tre dizionari non hanno un fattore di carico minimo (cioè non è necessario ridimensionare le tabelle in seguito alla rimozione di elementi).

3.6.1 tempi di delete in ns, con load factor massimo pari a 0.5

Chiavi	CUCKOO	L.PROBING	C.HASHING
2048	7753	8521	11105
1024	3003	3841	7962
512	3353	3213	7054
256	3003	3212	7124
128	2794	3073	6146
64	2793	1955	4759
32	2864	2025	5308

3.6.2 tempi di delete in ns, con load factor massimo pari a $1/3$

Chiavi	CUCKOO	L.PROBING	C.HASHING
2048	7263	8241	9848
1024	3212	3562	5937
512	3492	3492	7123
256	3143	3213	6635
128	2933	2794	5448
64	2495	1886	5378
32	2584	1886	5308

Nel caso della rimozione si possono fare considerazioni analoghe a quelle fatte per l'*unsuccessfull lookup*.

4 Funzioni di hashing universale

L'*hashing universale* è una tecnica di hashing che consiste nello scegliere la funzione di hash da utilizzare in modo casuale mediante un algoritmo di tipo probabilistico. Questo metodo garantisce un basso numero di collisioni, anche se i dati sono scelti mediante l'*adversary model*. Si conoscono diverse funzioni universali di hashing (applicabili a settori diversi, cioè una per le stringhe, una per gli interi, una per i vettori...). Le funzioni universali di hashing vengono utilizzate in vari campi delle scienze informatiche quali implementazione di tabelle hash e crittografia.

4.1 costruzione di una funzione di hashing universale

In tutti calcoli seguenti si deve considerare r la dimensione della tabella che ospita le varie chiavi.

4.1.1 hashing di interi

La proposta originaria è di Carter e Wegman [3]. Sia l'universo dei valori da considerare $U = \{0, .., u-1\}$, $|U| = u$, sia p un numero intero primo tale che $p > u$, allora una funzione di hash universale è data da

$$h(x) = ((ax + b) \mod p) \mod r$$

dove a,b sono due numeri positivi interi scelti casualmente entrambi minori di p con $a \neq 0$.

4.1.2 hashing di vettori

Con vettore si intende un oggetto di lunghezza prefissata di istruzioni macchina (ad esempio un array di bit), ma questo metodo può anche essere applicato ad oggetti formati da un numero costante di caratteri. Sia ora $\overline{x} = \{x_0, ..., x_{k-1}\}$ formato da k elementi. Allora una funzione di hash universale è data dal seguente:

$$h(\overline{x}) = \left(\sum_{i=0}^{k-1} h_i(x_i) \right) \mod r$$

5 bibliografia

[1] Rasmus Pagh and Flemming Friche Rodler, *Cuckoo Hashing*, Journal of Algorithms Volume 51, Issue 2, May 2004,

[2] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer auf der Heide, Hans Rohnert and Robert E. Tarjan. *Dynamic perfect hashing: Upper and lower bounds*. SIAM J.Comput., 23(4):738-761, 1994.

[3] Carter, Larry; Wegman, Mark N., *Universal Classes of Hash Functions. Journal of Computer and System Sciences,1979*.

[4] Rasmus Pagh. *On the Cell Probe Complexity of Membership and Perfect Hashing*. In Proceedings of the 33rd Annual ACM Symposium on Theory of Computing (STOC '01), pages 425–432. ACM Press, 2001.

A Implementazione in java

Ecco l'implementazione completa della classe *CuckooTable.java*, usata nei test:

```
import java.util.Random;
import java.lang.Math;

/**
 * This is Cuckoo Hashing simple implementation described by Rasmus Pagh
 * and Flemming Richie Rodler.
 *
 * @author Fabio Grigolo
 * @version 1.0
 */
public class CuckooTable
{
    private int insertions; //procedure di insert effettuate
    private int size1; //elementi contenuti
    private int size2;
    private double MAXIMUM_LOAD_FACTOR;  //max allowed load factor
    private int[] T1;//primo array
    private int[] T2;//secondo array
    private int MaxLoop;
    private HashValues H1;
    private HashValues H2;
    private int p = 2003;
    private Random random;
    private int epsilon;

    /**
     * Constructor; creates an empty table
     *with maximum load factor = 0,5 (epsilon)
     */
    public CuckooTable()
    {
        T1 = new  int[1];
        T2 = new  int[1];
        size1 = 0;
        size2 = 0;
        insertions = 0;
        random = new Random();
        epsilon = 1;
        MAXIMUM_LOAD_FACTOR = 0.5;
        H1 = new HashValues(random.nextInt(2048)+ 1,
            random.nextInt(2048)+ 1, p, T1.length);
        H2 = new HashValues(random.nextInt(2048)+ 1,
            random.nextInt(2048)+ 1, p, T2.length);
            MaxLoop = setMaxLoopValue();
    }

    /**
     * Constructor; creates an empty table with maximum load factor = 1/(1+epsilon);
     *
     * @param epsilon the value of epsilon
     */
    public CuckooTable(int e)
    {
        T1 = new  int[1];
        T2 = new  int[1];
```

```
        size1 = 0;
        size2 = 0;
        insertions = 0;
        random = new Random();
        epsilon = e;
        MAXIMUM_LOAD_FACTOR =(double) 1/(1+e);
        H1 = new HashValues(random.nextInt(2048)+ 1,
            random.nextInt(2048)+ 1, p, T1.length);
        H2 = new HashValues(random.nextInt(2048)+ 1,
            random.nextInt(2048)+ 1, p, T2.length);
            MaxLoop = setMaxLoopValue();
    }

    /**
     *Search a value in the table
     *
     *@param o the value to search
     *
     *@return true if the value is been found, false otherwise
     */
    public boolean lookup(int o)
    {
        int k1 = integerHashing(o,H1);
        int k2 = integerHashing(o, H2);
        return ( T1[k1] == o)  ||  (T2[k2] == o);
    }


    /**
     * inserts new values into the table
     *
     * @param o the value to insert
     */
    public void insert(int o)
    {
        if(o == 0)
            return;
        int swap;
        boolean done = false;
       if ( ((double) size1/T1.length) >= MAXIMUM_LOAD_FACTOR ||
            ((double) size2/T2.length) >= MAXIMUM_LOAD_FACTOR)
            resize();
       if(getInsertions() > (double) Math.pow(T1.length,2))
        rehash();
        if (!lookup(o))
        {
                int x = o;
            for(int j = 0; (j < MaxLoop) && (!done); j++)
            {
                swap = T1[integerHashing(x, H1)];
                T1[integerHashing(x, H1)] = x;
                x = swap;
                if (x == 0)
                {
                    size1++;
                    done = true;
                }
                 if(!done)
                 {
                   swap = T2[integerHashing(x, H2)];
                   T2[integerHashing(x, H2)] = x;
                   x = swap;
```

```
                }
                if((x == 0) && (!done))
120             {
                    size2++;
                    done = true;
                }
            }
125         if(!done)
            {
                rehash();
                insert(x);
            }
130         insertions++;
        }
    }
    //ridimensionamento della tabella in caso di massimo carico
    private void resize()
135 {

        int[] a = T1;
        int[] b = T2;
        T1 = new int[2 * a.length];
140     T2 = new int[2 * b.length];
        size1 = 0;
        size2 = 0;
        insertions = 0;

145     H1 =  new HashValues(random.nextInt(2048)+ 1,
            random.nextInt(2048)+1, p, T1.length);
        H2 =  new HashValues(random.nextInt(2048)+ 1,
            random.nextInt(2048)+ 1, p, T2.length);

150     for(int i = 0; i < a.length; i++ )
        {
            if(a[i] != 0)
            insert(a[i]);
        }
155
                for(int i = 0; i < b.length; i++ )
        {
            if(b[i] != 0)
            insert(b[i]);
160     }

        MaxLoop = setMaxLoopValue();
    }

165 /**
     * removes a value from the table
     *
     * @return the value removed if it's present, otherwise -1
     */
170 public boolean remove(int o)
    {
            if(T1[integerHashing(o,H1)] == o)
            {
                size1--;
175             T1[integerHashing(o,H1)] = 0;
                return true;
            }
            if(T1[integerHashing(o,H2)] == o)
            {
```

```
180                 size2--;
                    T1[integerHashing(o,H2)] = 0;
                    return true;
                }
            return false;
185     }

        /**
         * Computes the universal hash
         *
190      * @param value the value to hash
         *
         * @param h the hash function parameters
         *
         * @return the hashed value
195      */
        public static int integerHashing(int value, HashValues h)
        {

           return ((h.getA() * value + h.getB()) % h.getP()) % h.getR();
200     }

        /**
         * gives the actual load factor of the first table
         *
205      * @return the load factor calculated by the formula size/TableSize
         */
       public double getLoadFactor1()
       {
           return (double) size1/T1.length;
210     }

           /**
         * gives the actual load factor of the second table
         *
215      * @return the load factor calculated by the formula size/TableSize
         */
       public double getLoadFactor2()
       {
           return (double) size2/T1.length;
220     }

        //rehash the table with 2 new hash function chosen randomly
        private void rehash()
        {
225      H1 = new HashValues(random.nextInt(2048)+ 1,
            random.nextInt(2048)+ 1, p, T1.length);
         H2 = new HashValues(random.nextInt(2048)+1,
            random.nextInt(2048)+ 1, p, T2.length);
         size1 = 0;
230      size2 = 0;
         insertions = 0;
         int[] t1 = T1;
         int[] t2 = T2;
         T1 = new int[t1.length];
235      T2 = new int[t2.length];
         for(int i = 0; i < t1.length; i++)
         {
            if(t1[i] != 0)
            {
240             insert(t1[i]);
            }
```

```
        }
          for(int i = 0; i < t2.length; i++){
            if(t2[i] != 0)
            {
                insert(t2[i]);
            }
         }
        }

        /**
         * return a string message list with the occuped cells list,
         * the table size and the getInsertions() value
         *
         * @return string message
         */
        public String toString()
        {

            String s ="";
            s = s+ "Table␣1:\n";
            for(int i = 0; i <T1.length; i++)
            {
                if(T1[i] != 0)
                s = s + "riga␣" + i + '\t' + "valore␣" + T1[i]+'\n';

            }
             s = s + '\n';
             s = s + "Table␣2:\n";
            for(int i = 0; i <T2.length; i++)
            {
                if(T2[i] != 0)
                s = s + "riga␣" + i + '\t' + "valore␣" + T2[i]+'\n';

            }
            s = s+ '\n';
            s = s + "dimensione(numero␣elementi):␣"+getSize()+'\n';
            s = s + "inserimenti:"+getInsertions()+'\n';
            return s;
        }

        /**
         * Returns the amount of elements in the table
         *
         * @return the amount of elements in the table
         */
        public int getSize()
        {
            return size1+size2;
        }

        /**
         * returns the amount of insertion procedures after the last rehash
         *
         * @return an integer value
         *
         */
        public int getInsertions()
        {
            return insertions;
        }
```

```
305     private int setMaxLoopValue()
        {

            double logvalue = 3 * Math.log(T1.length) / Math.log(1+epsilon) ;
            return Math.max(1, (int)logvalue);
310     }

        public int getMaxLoopValue()
        {
            return MaxLoop;
315     }

        private class HashValues
        {
            private int a;
320         private int b;
            private int p;
            private int r;

           public HashValues()
325         {
                a = 0;
                b = 0;
                p = 0;
                r = 0;
330         }
            public HashValues(int aa, int bb, int pp, int rr)
            {
                a = aa;
                b = bb;
335             p = pp;
                r = rr;
            }
            public int getA()
            {return a;}
340         public int getB()
            {return b;}
            public int getP()
            {return p;}
            public int getR()
345         {return r;}
        }
    }
```