Hashing e tabelle hash

Matteo Ferrara

Dipartimento di Informatica - Scienza e Ingegneria

matteo.ferrara@unibo.it

Funzioni hash (1)

Definizione

Una **funzione hash** è una trasformazione *suriettiva* ma *non iniettiva* tale che:

$h : \mathcal{K} \rightarrow \{1, \ldots, m\}$

Il dominio $\mathcal{K}$ è costituito dall'insieme (potenzialmente infinito) di tutte le possibili chiavi che possono essere fornite come input alla funzione.

Il codominio, $\{1, \ldots, m\}$, è anche detto **spazio di indirizzamento** della funzione hash.

Il valore di codominio restituito da una funzione hash è comunemente noto con diversi termini equivalenti:

- hash value
- hash code
- hash digest

Funzioni hash (2)

Collisioni

Poichè le funzioni hash non sono iniettive, è possibile che a due elementi distinti del dominio corrisponda lo stesso valore di codominio. Si parla in questo caso di collisione.

Definizione

Si consideri una *funzione hash* del tipo:

$h : \mathcal{K} \rightarrow \{1, \ldots, m\}$

Dati $x_i, x_j \in \mathcal{K}$, si verifica una **collisione** se:

$h(x_i) = h(x_j), x_i \neq x_j$

Funzioni hash (3)

Proprietà

Determinismo

Ogni funzione hash deve essere **deterministica**: lo stesso input deve comportare sempre lo stesso output, anche a seguito di più chiamate alla funzione.

Distribution

Un elemento che si presenta con elevata probabilità deve avere un output (hash code) indistinguibile da un elemento meno probabile. In particolare, una funzione hash si dice **ideale** se l'output è uniformemente distribuito nel codominio.

Il metodo della divisione

Una semplice funzione hash

Funzione modulo $O(1)$

INPUT: $k, m \in \mathbb{N}$

OUTPUT: $h \in \{0, \ldots, m-1\}$

1. $h = k \bmod m$
2. **return** h

Cardinalità dello spazio di indirizzamento: m.

```
hash_modulus(997870011,128)     ->      Digest: 59
hash_modulus(941408520,128)     ->      Digest: 8
hash_modulus(107076685,128)     ->      Digest: 77
hash_modulus(587050924,128)     ->      Digest: 44
hash_modulus(984171131,128)     ->      Digest: 123
hash_modulus(710589170,128)     ->      Digest: 114
hash_modulus(316000718,128)     ->      Digest: 78
hash_modulus(938202552,128)     ->      Digest: 56
hash_modulus(437044822,128)     ->      Digest: 86
hash_modulus(911348241,128)     ->      Digest: 17
hash_modulus(305153385,128)     ->      Digest: 105
hash_modulus(454460356,128)     ->      Digest: 68
hash_modulus(258207257,128)     ->      Digest: 25
hash_modulus(4523754,128)       ->      Digest: 106
hash_modulus(106624676,128)     ->      Digest: 36
hash_modulus(468548121,128)     ->      Digest: 25
hash_modulus(170353191,128)     ->      Digest: 39
hash_modulus(280541810,128)     ->      Digest: 114
hash_modulus(48473280,128)      ->      Digest: 64
hash_modulus(154693031,128)     ->      Digest: 39
```

Murmur hash (1)

- Presenta **ottime performance** in termini di velocità di esecuzione;
- Ha buone proprietà di *distribution*;
- Accetta in input stringhe di bit di **lunghezza arbitraria** e restituisce in output una sequenza **apparentemente casuale** di *32 bit*;
- L'input prevede anche un *seed*;
- Le stringhe vengono processate in blocchi da *4 byte* ciascuno.

Cardinalità dello spazio di indirizzamento: 2^{32}.

Murmur hash (2)

MurmurHash - key chunks $O(1)$

INPUT: $x \in \{0,1\}^*$, $s \in \mathbb{N}$ OUTPUT: $h \in \{0,\ldots,2^{32}-1\}$

1. $hash \leftarrow s$
2. **for each** *fourByteBlock* **in** x **do**
3. $k \leftarrow fourByteBlock$
4. $k \leftarrow k \cdot 0xCC9E2D51$
5. $k \leftarrow k$ ROL 15
6. $k \leftarrow k \cdot 0x1B873593$
7. $hash \leftarrow hash$ XOR k
8. $hash \leftarrow hash$ ROL 13
9. $hash \leftarrow hash \cdot 5 + 0xE6546B64$

…

Murmur hash (3)

MurmurHash - trailing chunk $O(1)$

```
...
10.  if x has trailingBytes then
11.      t ← trailingBytes
12.      t ← t ·0xCC9E2D51
13.      t ← t ROL 15
14.      t ← t ·0x1B873593
15.      hash ← hash XOR t
...
```

N.B.: questa porzione di codice deve essere eseguita solo se la chiave data in input occupa una quantità di memoria non divisibile per 4 Byte. In tal caso, infatti, il ciclo di elaborazione dei blocchi di 4 Byte terminerà con una porzione di dati non processata di 1, 2 o 3 Byte (*trailingBytes*).

Murmur hash (4)

MurmurHash -avalanche $O(1)$

```
...
16.  hash ← hash XOR LEN(x)
17.  hash ← hash XOR (hash SHR 16)
18.  hash ← hash · 0x85EBCA6B
19.  hash ← hash XOR (hash SHR 13)
20.  hash ← hash · 0xC2B2AE35
21.  hash ← hash XOR (hash SHR 16)
22.  return hash
```

Obiettivo (1)

1. Implementare la funzione **MurmurHash** seguendo lo pseudo-codice visto nelle slide precedenti.
2. Per semplificare il problema, implementeremo prima la parte di *key chunks*, quindi quella di *trailing chunk* e infine quella di *avalanche*.
3. Testare l'algoritmo implementato con le chiavi contenute nel materiale dell'esercitazione: *data.txt*.

N.B.: per velocizzare lo sviluppo si consiglia di utilizzare il codice sorgente presente nel materiale dell'esercitazione.

```
hashModulus(997870011,128)->    Digest: 59      hashMurmur('997870011',9,128)-> Digest: 849075530
hashModulus(941408520,128)->    Digest: 8       hashMurmur('941408520',9,128)-> Digest: 1297323125
hashModulus(107076685,128)->    Digest: 77      hashMurmur('107076685',9,128)-> Digest: 1395735456
hashModulus(587050924,128)->    Digest: 44      hashMurmur('587050924',9,128)-> Digest: 1857978615
hashModulus(984171131,128)->    Digest: 123     hashMurmur('984171131',9,128)-> Digest: 144354510
hashModulus(710589170,128)->    Digest: 114     hashMurmur('710589170',9,128)-> Digest: 1179177154
hashModulus(316000718,128)->    Digest: 78      hashMurmur('316000718',9,128)-> Digest: 3262798807
hashModulus(938202552,128)->    Digest: 56      hashMurmur('938202552',9,128)-> Digest: 2583955022
hashModulus(437044822,128)->    Digest: 86      hashMurmur('437044822',9,128)-> Digest: 3902410137
hashModulus(911348241,128)->    Digest: 17      hashMurmur('911348241',9,128)-> Digest: 1246146021
hashModulus(305153385,128)->    Digest: 105     hashMurmur('305153385',9,128)-> Digest: 2833491052
hashModulus(454460356,128)->    Digest: 68      hashMurmur('454460356',9,128)-> Digest: 1524134077
hashModulus(258207257,128)->    Digest: 25      hashMurmur('258207257',9,128)-> Digest: 4160012500
hashModulus(4523754,128)->      Digest: 106     hashMurmur('004523754',9,128)-> Digest: 389315228
hashModulus(106624676,128)->    Digest: 36      hashMurmur('106624676',9,128)-> Digest: 1915086570
hashModulus(468548121,128)->    Digest: 25      hashMurmur('468548121',9,128)-> Digest: 325387131
hashModulus(170353191,128)->    Digest: 39      hashMurmur('170353191',9,128)-> Digest: 946407502
hashModulus(280541810,128)->    Digest: 114     hashMurmur('280541810',9,128)-> Digest: 3342662885
hashModulus(48473280,128)->     Digest: 64      hashMurmur('048473280',9,128)-> Digest: 3361886329
hashModulus(154693031,128)->    Digest: 39      hashMurmur('154693031',9,128)-> Digest: 2600151531
```

Obiettivo (2)

Risultato *key chunks*

```
hashMurmur('997870011',9,128)-> Digest: 3289810584
hashMurmur('941408520',9,128)-> Digest: 690353968
hashMurmur('107076685',9,128)-> Digest: 3625695165
hashMurmur('587050924',9,128)-> Digest: 440312040
hashMurmur('984171131',9,128)-> Digest: 2071735226
hashMurmur('710589170',9,128)-> Digest: 2084848371
hashMurmur('316000718',9,128)-> Digest: 1596475164
hashMurmur('938202552',9,128)-> Digest: 4066687315
hashMurmur('437044822',9,128)-> Digest: 917673221
hashMurmur('911348241',9,128)-> Digest: 3385628203
hashMurmur('305153385',9,128)-> Digest: 3963379206
hashMurmur('454460356',9,128)-> Digest: 3998463589
hashMurmur('258207257',9,128)-> Digest: 383563162
hashMurmur('004523754',9,128)-> Digest: 2718475440
hashMurmur('106624676',9,128)-> Digest: 546434302
hashMurmur('468548121',9,128)-> Digest: 1893428967
hashMurmur('170353191',9,128)-> Digest: 1159160771
hashMurmur('280541810',9,128)-> Digest: 3819054561
hashMurmur('048473280',9,128)-> Digest: 469984029
hashMurmur('154693031',9,128)-> Digest: 1370980907
```

Risultato *key chunks* + *trailing chunk*

```
hashMurmur('997870011',9,128)-> Digest: 4168901761
hashMurmur('941408520',9,128)-> Digest: 986001548
hashMurmur('107076685',9,128)-> Digest: 2705816624
hashMurmur('587050924',9,128)-> Digest: 1256985048
hashMurmur('984171131',9,128)-> Digest: 1192675747
hashMurmur('710589170',9,128)-> Digest: 1873053007
hashMurmur('316000718',9,128)-> Digest: 3538738616
hashMurmur('938202552',9,128)-> Digest: 232441125
hashMurmur('437044822',9,128)-> Digest: 3373066611
hashMurmur('911348241',9,128)-> Digest: 4121326642
hashMurmur('305153385',9,128)-> Digest: 2506134923
hashMurmur('454460356',9,128)-> Digest: 1337283471
hashMurmur('258207257',9,128)-> Digest: 1944466909
hashMurmur('004523754',9,128)-> Digest: 4074378624
hashMurmur('106624676',9,128)-> Digest: 2172061972
hashMurmur('468548121',9,128)-> Digest: 1286704382
hashMurmur('170353191',9,128)-> Digest: 2038268378
hashMurmur('280541810',9,128)-> Digest: 4030915165
hashMurmur('048473280',9,128)-> Digest: 266577057
hashMurmur('154693031',9,128)-> Digest: 1843224626
```

Tabelle hash (1)

Definizione

Sia data una funzione hash $h : \mathcal{K} \rightarrow \{1, \ldots, m\}$.
Consideriamo l'elemento $e = (k, v) \in \varepsilon$, identificato dalla chiave $k \in \mathcal{K}$ e contenente il valore v.
Consideriamo poi un array *T* composto da *m* celle (cioè di lunghezza *m*).
T costituisce una **tabella hash** rappresentante l'insieme ε se:

$$\forall e_i \in \varepsilon, T[h(k_i)] = v_i$$

In caso di collisione, la struttura dati preserva la proprietà suddetta mediante un sistema di **gestione delle collisioni** o **degli overflow**.

Informalmente, una **tabella hash** è una struttura dati atta a memorizzare insiemi di elementi tipicamente strutturati secondo il paradigma *chiave-valore* che sfruttano le funzioni hash in modo da garantire:

- un efficiente utilizzo della memoria;
- prestazioni di accesso ai dati ottimali.

Tabelle hash (2)

In fase di costruzione di una tabella hash, si procede tipicamente come segue:

- si valuta la cardinalità dell'insieme $\mathcal{K}$ di tutte le possibili chiavi;
- si valuta la cardinalità dell'insieme ε degli elementi che saranno effettivamente inseriti nella tabella hash;
- si valutano le prestazioni che la struttura deve offrire in termini di velocità di accesso ai dati (numero medio di overflow);
- in base alle valutazioni di cui sopra, si sceglie il numero m di celle, che costituiscono l'area primaria della struttura dati.

Si adotta poi una funzione hash:

- la funzione deve avere uno spazio di indirizzamento di cardinalità maggiore o uguale a m;
- in caso sia maggiore di m, si può utilizzare la funzione modulo per restringere i digest allo spazio di indirizzamento desiderato (cioè $\{1,\ldots,m\}$);

Tabelle hash (3)

Gestione dell'overflow tramite chaining

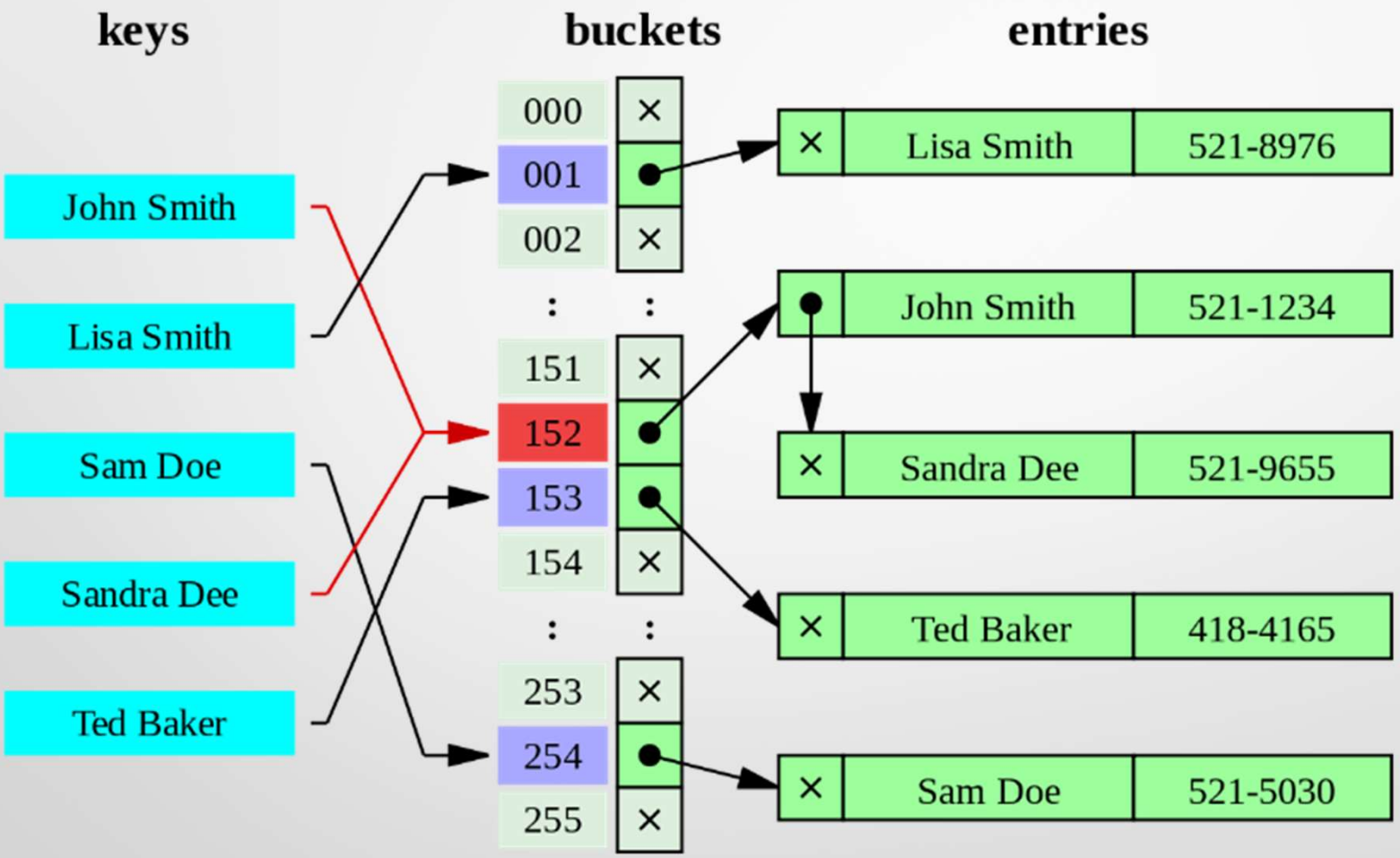

Immagine tratta da Wikipedia: https://en.wikipedia.org/wiki/Hash_table

Tabelle hash (4)

Chained hash table -Insert $O(1)$

INPUT: $T = [L_1,\ldots,L_m]$, $k \in \mathcal{K}$, $v \in \mathbb{N}$

OUTPUT: $T = [L_1,\ldots,L_m]$

1. $h \leftarrow$ HASH(k)
2. $L \leftarrow T[h]$
3. $L_{new}[key] \leftarrow k$
4. $L_{new}[val] \leftarrow v$
5. $L_{new}[next] \leftarrow L$
6. $T[h] \leftarrow L_{new}$

Si noti che T è implementata come un **array di liste concatenate**.

L_i rappresenta l'i-esima lista concatenata il cui nodo di testa è memorizzato nel *bucket* i-esimo della tabella hash.

L'elemento passato in input viene sempre inserito in testa alla lista.

Obiettivo

1. Implementare la procedura **Insert** seguendo lo pseudo-codice visto nelle slide precedenti.
2. Testare l'algoritmo implementato con i dati contenuti nel materiale dell'esercitazione (*data.txt*) utilizzando valori diversi di *m* (numero di bucket) e di *hashSeed* (seme iniziale utilizzato dalla funzione Murmur implementata in precedenza).

N.B.: per velocizzare lo sviluppo si consiglia di utilizzare il codice sorgente presente nel materiale dell'esercitazione.

```
m=15
hashSeed=123
n=20

[Bucket 0]:
[Bucket 1]: (Nodo 1: k='004523754' v=19)-->
[Bucket 2]:
[Bucket 3]:
[Bucket 4]:
[Bucket 5]: (Nodo 1: k='048473280' v=23)-->(Nodo 2: k='280541810' v=25)-->(Nodo 3: k='437044822' v=26)-->(Nodo 4: k='107076685' v=19)-->
[Bucket 6]:
[Bucket 7]: (Nodo 1: k='106624676' v=22)-->
[Bucket 8]: (Nodo 1: k='305153385' v=25)-->
[Bucket 9]: (Nodo 1: k='454460356' v=25)-->
[Bucket 10]: (Nodo 1: k='941408520' v=30)-->
[Bucket 11]: (Nodo 1: k='170353191' v=21)-->(Nodo 2: k='911348241' v=27)-->(Nodo 3: k='938202552' v=26)-->(Nodo 4: k='587050924' v=28)-->
[Bucket 12]: (Nodo 1: k='468548121' v=28)-->(Nodo 2: k='316000718' v=30)-->
[Bucket 13]: (Nodo 1: k='154693031' v=29)-->(Nodo 2: k='984171131' v=24)-->
[Bucket 14]: (Nodo 1: k='258207257' v=20)-->(Nodo 2: k='710589170' v=19)-->(Nodo 3: k='997870011' v=26)-->

Fattore di carico: 1.33
Lunghezza massima delle catene di overflow: 4
Lunghezza media delle catene di overflow: 2.00
```

Tabelle hash (5)

Chained hash table - Search $O(n/m)$

INPUT: T = $[L_1,\ldots,L_m]$, $k \in \mathcal{K}$

OUTPUT: (h, i, v)
$h \in \{1, \ldots, m\}$
$i \in \{-1\} \cup \{1, \ldots, n\}$
$v \in \mathbb{N}$

```
1. h ← HASH(k)
2. i ←1
3. L ← T[h]
4. while L do
5.     if L[key] = k then return (h, i, L[val])
6.     else L ← L[next]
7.     i ← i + 1
8. return (h, -1, -)
```

Il metodo ritorna la tupla *(h,i,v)* che rappresenta:

- il bucket *(h)* in cui è contenuta la chiave *k;*
- la posizione del nodo che contiene l'elemento nella rispettiva lista concatenata *(i). Con i*=-1 se la chiave *k* non è stata trovata;
- il valore corrispondente alla chiave *k* (se presente).

Obiettivo

1. Implementare la funzione **Search** seguendo lo pseudo-codice visto nelle slide precedenti.
2. Testare l'algoritmo implementato con i dati contenuti nel materiale dell'esercitazione (*data.txt*) ricercando vari valori di chiave (anche non presenti).

N.B.: per velocizzare lo sviluppo si consiglia di utilizzare il codice sorgente presente nel materiale dell'esercitazione.

```
Ricerca chiave: 911348241
h=11 i=1 v=27
```

```
Ricerca chiave: 123456789
Chiave non trovata
```

Tabelle hash (6)

Chained hash table - Delete $O(n/m)$

INPUT: T = $[L_1,\ldots,L_m]$, $k \in \mathcal{K}$ OUTPUT: T = $[L_1,\ldots,L_m]$

```
1.  (h,i,v) ← Search(T,k)
2.  if i = -1 then return
3.  L ← T[h]
4.   if L[next] is null then T[h] ← null
5.   else
6.       if i = 1 then T[h] ← L[next], return
7.       while i > 1 do
8.           L2 ← L
9.           L ← L[next]
10.          i ← i - 1
11.      L2[next] ← L[next]
```

Obiettivo

1. Implementare la procedura **Delete** seguendo lo pseudo-codice visto nelle slide precedenti.
2. Testare l'algoritmo implementato con i dati contenuti nel materiale dell'esercitazione (*data.txt*) cancellando uno o più valori di chiave.

N.B.: per velocizzare lo sviluppo si consiglia di utilizzare il codice sorgente presente nel materiale dell'esercitazione.

```
Cancellazione chiave: 911348241
m=15
hashSeed=123
n=19

[Bucket 0]:
[Bucket 1]: (Nodo 1: k='004523754' v=19)-->
[Bucket 2]:
[Bucket 3]:
[Bucket 4]:
[Bucket 5]: (Nodo 1: k='048473280' v=23)-->(Nodo 2: k='280541810' v=25)-->(Nodo 3: k='437044822' v=26)-->(Nodo 4: k='107076685' v=19)-->
[Bucket 6]:
[Bucket 7]: (Nodo 1: k='106624676' v=22)-->
[Bucket 8]: (Nodo 1: k='305153385' v=25)-->
[Bucket 9]: (Nodo 1: k='454460356' v=25)-->
[Bucket 10]: (Nodo 1: k='941408520' v=30)-->
[Bucket 11]: (Nodo 1: k='170353191' v=21)-->(Nodo 2: k='938202552' v=26)-->(Nodo 3: k='587050924' v=28)-->
[Bucket 12]: (Nodo 1: k='468548121' v=28)-->(Nodo 2: k='316000718' v=30)-->
[Bucket 13]: (Nodo 1: k='154693031' v=29)-->(Nodo 2: k='984171131' v=24)-->
[Bucket 14]: (Nodo 1: k='258207257' v=20)-->(Nodo 2: k='710589170' v=19)-->(Nodo 3: k='997870011' v=26)-->

Fattore di carico: 1.27
Lunghezza massima delle catene di overflow: 4
Lunghezza media delle catene di overflow: 1.90
```