Programmazione e Strutture Dati (PR&SD)
I° ANNO – Informatica
Prof. V. Fuccella

Tabelle Hash

Tabelle Hash

- Una tabella hash, è una struttura dati usata per mettere in corrispondenza una data chiave con un dato valore.
  - Chiavi e valori possono appartenere a diversi tipi primitivi o strutturati
- Ad esempio, è possibile associare l'età (intero) ad una persona utilizzando il nome (stringa)
  - Alcuni linguaggi (p.e. Perl, PHP, Javascript, …) mettono a disposizione dei tipi di dati appositi chiamati array associativi
  - In Java potremmo utilizzare il tipo HashMap<String,Integer>
  - Esempio **Javascript**

```
» eta = new Object();
  eta["francesco"] = 23;
  eta["filippo"] = 24;
  eta["francesco"]
← 23
```

  - È possibile utilizzare un array, a patto che si associno prima gli indici interi alle persone e poi l'età agli indici

Tabelle Hash
Operatori

- In molte applicazioni è necessario che un insieme dinamico fornisca solamente le seguenti operazioni:
  - INSERT(key,value): Inserisce un elemento nuovo, con un certo valore (unico) di un campo chiave
  - SEARCH(key): Determina se un elemento con un certo valore della chiave esiste; se esiste, lo restituisce
  - DELETE(key): Elimina l'elemento identificato dal campo chiave, se esiste.
- Non è, ad esempio, necessario dover ordinare l'insieme dei dati o restituire l'elemento massimo, o il successore

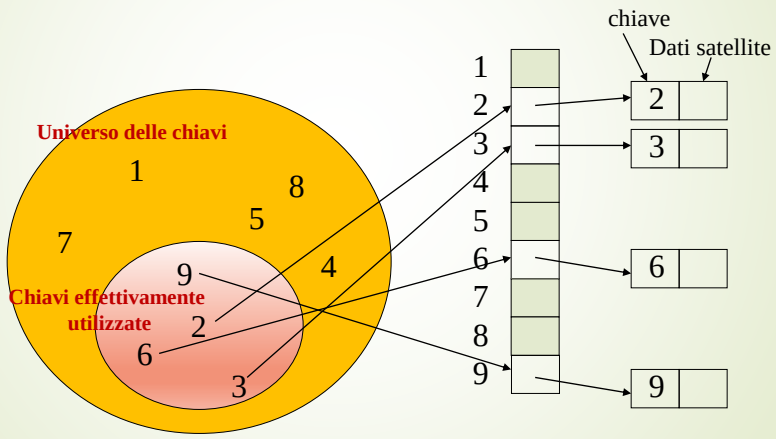
Definizioni

- U - Universo di tutte le possibili chiavi
- K - Insieme delle chiavi effettivamente memorizzate

Chiavi Intere
Indirizzamento diretto

- Se l'universo delle chiavi è piccolo e le chiavi sono intere allora è sufficiente utilizzare una *tabella ad indirizzamento diretto*
- Una tabella ad indirizzamento diretto corrisponde al concetto di array:
  - ad ogni chiave possibile corrisponde una posizione, o slot, nella tabella
  - una tabella restituisce il dato memorizzato nello slot di posizione indicato tramite la chiave in tempo O(1)

Visualizzazione



Universo grande delle chiavi

- Se le chiavi non sono intere e/o l'universo delle possibili chiavi è molto grande non è possibile o conveniente utilizzare il metodo delle tabelle ad indirizzamento diretto
  - Può non essere possibile a causa della limitatezza delle risorse di memoria
  - Può non essere conveniente perché se il numero di chiavi effettivamente utilizzato è piccolo si hanno tabelle quasi vuote. Viene allocato spazio inutilizzato

Compromesso

- Se $|K| \sim |U|$:
  - Non sprechiamo (troppo) spazio
  - Operazioni in tempo O(1) nel caso peggiore
- Se $|K| << |U|$: soluzione non praticabile
  - Esempio: U = {"numero di matricola" degli studenti PSD}
  - Se il numero di matricola ha 6 cifre, l'array deve avere spazio per contenere 10^6 elementi
  - Se gli studenti del corso sono ad esempio 30, lo spazio realmente occupato dalle chiavi memorizzate è $30/10^6 = 0.00003 = 0.003\%$!!!
- Le *tabelle hash* permettono di mediare i requisiti di memoria ed efficienza nelle operazioni
- L'Hashing permette di impiegare una quantità ragionevole sia di memoria che di tempo operando un compromesso tra i casi precedenti

Tabelle Hash

- Con il metodo di indirizzamento diretto un elemento con chiave k viene memorizzato nella tabella in posizione k
- Con il metodo hash un elemento con chiave k viene memorizzato nella tabella in posizione h(k)
- La funzione h() è detta *funzione hash*
- Lo scopo della funzione hash è di definire una corrispondenza tra l'universo U delle chiavi e le posizioni di una tabella hash T[0..m-1] con m << |U|

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

Tabelle Hash
Gestione delle Collisioni

- Necessariamente la funzione hash non può essere iniettiva, ovvero due chiavi distinte possono produrre lo stesso valore hash
- Ogniqualvolta $h(k_i)=h(k_j)$ quando $k_i \neq k_j$, si verifica una collisione
- Occorre:
  - Minimizzare il numero di collisioni (ottimizzando la funzione di hash)
  - Gestire le collisioni residue, quando avvengono (permettendo a più elementi di risiedere nella stessa locazione)

Risoluzione delle Collisioni

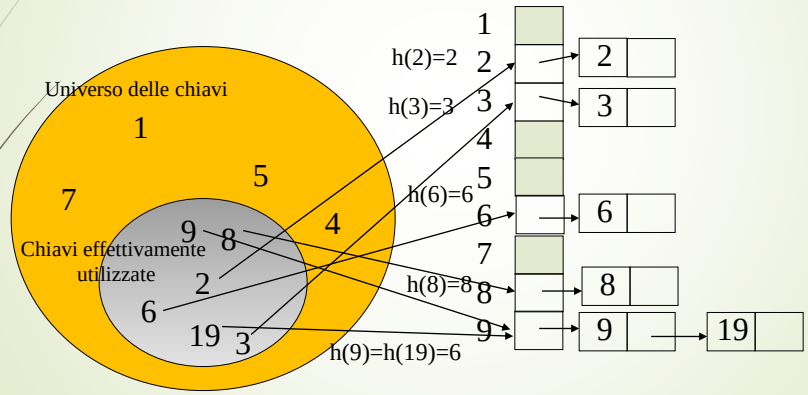
- Per risolvere il problema delle collisioni si impiegano principalmente due strategie:
  - metodo di concatenazione
  - metodo di indirizzamento aperto

Risoluzione delle Collisioni

Metodo di concatenazione

- L'idea è di mettere tutti gli elementi che collidono in una lista concatenata
- La tabella contiene in posizione j
  - un puntatore alla testa della j-esima lista
  - oppure un puntatore nullo se non ci sono elementi

Visualizzazione



ADT: Key

Sintattica	Semantica
Nome del tipo: Key Tipi usati: Int, boolean	Dominio: k ∈ U
equals(Key, Key) → boolean hashValue(Key, int) → int inputKey() → Key outputKey(Key)	equals(k1, k2) → b • Post: b = true se k1=k2; b = false altrimenti hashValue(k, size) → index • Pre: k != nil, size > 0 • Post: 0 <= index < size

ADT: Entry

Sintattica	Semantica
Nome del tipo: Entry Tipi usati: Item, Key	Dominio: Coppia (chiave, valore) chiave è di tipo Key, valore è di tipo Item
newEntry(Key, Item) → Entry	newEntry(key, value) → e • Post: e = (key, value)
getKey(Entry) → Key	getKey(e) → key • Post: e = (key, value)
getValue(Entry) → Item	getValue(e) → value • Post: e = (key, value)

ADT: Hashtable

Sintattica	Semantica
Nome del tipo: Hashtable Tipi usati: Entry, boolean, Key	Dominio: insieme di elem. T={a1,…,an} di tipo Entry
newHashtable() → Hashtable	newHashtable() → t • Post: t = { }
insertHash(Hashtable, Entry) → Hashtable	insertHash(t, e) → t' • Post: t = {a1, a2, … an}, t' = {a1, a2, …, e,..., an}
searchHash(Hashtable, Key) → Entry	searchHash(t, k) → e • Pre: t = <a1, a2, …, an> n>0 • Post: e = ai con 1 ≤ i ≤ n se ai(key) = k
deleteHash(Hashtable, Key) → Hashtable	deleteHash(t, k) → t' • Pre: t = {a1, a2, …, an} n>0, a_i(key) = k, 1 ≤ i ≤ n • Post: t' = <a_1, a_2, …, a_{i-1}, a_{i+1}, …, a_n>

Funzioni Hash

- Quali sono le caratteristiche di una funzione hash?
- *Criterio di uniformità semplice*:
  - il valore hash di una chiave k è uno dei valori 0..m-1 in modo equiprobabile
- Un altro requisito è che una "buona" funzione hash dovrebbe utilizzare tutte le cifre della chiave per produrre un valore hash

Funzioni Hash
Tipo Int

- Metodo di divisione: la funzione hash è del tipo:

$$h(k) = k \bmod m$$

  - Cioè il valore hash è il resto della divisione di k per m
- Pregio: il metodo è veloce

Funzioni Hash
Tipo Stringa

- Per convertire una stringa in un numero naturale si considera la stringa come un numero in base 128
- Esistono cioè 128 simboli diversi per ogni cifra di una stringa
- È possibile stabilire una conversione fra ogni simbolo e i numeri naturali (codifica ASCII ad esempio)
- la conversione viene poi fatta nel modo tradizionale
- Es: per convertire la stringa "pt" si ha:
  - 'p'*128^1 + 't'*128^0= 112 * 128 + 116 *1= 14452

Funzioni Hash
Chiavi molto grandi

- Spesso capita che le chiavi abbiano dimensione tale da non poter essere rappresentate come numeri interi per una data architettura
- Es: la chiave per la stringa: "averylongkey" è:

  $97*128^{11}+ 118*128^{10}+ 101*128^9+ 114*128^8+ 121*128^7+ 108*128^6+ 111*128^5+ 110*128^4+ 103*128^3+ 107*128^2+ 101*128^1+ 121*128^0$
- che è troppo grande per poter essere rappresentata
- un modo alternativo di procedere è di utilizzare una funzione hash modulare, trasformando un pezzo di chiave alla volta

Funzioni Hash
Chiavi molto grandi

- Per fare questo basta sfruttare le proprietà aritmetiche dell'operazione modulo e usare la regola di Horner per scrivere la conversione
- Si ha infatti che il numero precedente può essere scritto come:

  ((((((((((97*128 + 118)*128 + 101)*128 + 114)*128+121)*128 + 108)*128 + 111)*128 + 110)*128 + 103)*128 + 107)*128 + 101)*128 + 12)

Funzioni Hash
Chiavi molto grandi

```
int hash(char *v, int m){
  int h = 0, a = 128;
  for (; *v != '\0'; v++)
     h = (h*a + *v) % m;
  return h;
}
```

Risoluzione delle Collisioni

- Per risolvere il problema delle collisioni si impiegano principalmente due strategie:
  - metodo di concatenazione
  - metodo di indirizzamento aperto

Indirizzamento aperto

- L'idea è di memorizzare tutti gli elementi nella tabella stessa
- in caso di collisione si memorizza l'elemento nella posizione *successiva*
  - si genera un nuovo valore hash fino a trovare una posizione vuota dove inserire l'elemento
- si estende la funzione hash perché generi non solo un valore hash ma una sequenza di scansione
  - $h : U \times \{0,1,\ldots,m-1\} \rightarrow \{0,1,\ldots,m-1\}$
  - cioè prenda in ingresso una chiave e un indice di posizione e generi una nuova posizione

Sequenza di scansione

- data una chiave k, si parte dalla posizione 0 e si ottiene h(k,0)
- la seconda posizione da scansionare sarà h(k,1)
- e così via: h(k,i)
- ottenendo una sequenza <h(k,0),h(k,1),…,h(k,m-1)>

Pseudocodice per l'inserimento

```
Hash-Insert(T,k)
1 i ← 0
2 repeat j ← h(k,i)
3  if T[j]=NIL
4  then  T[j] ← k
5     return j
6  else i ← i+1
7 until i=m
8 error "overflow"
```

Pseudocodice per la ricerca

```
Hash-Search(T,k)
1 i ← 0
2 repeat j ← h(k,i)
3  if T[j]=k
4  then return j
6  i ← i+1
7 until i=m o T[j]=NIL
8 return NIL
```

Caratteristiche di h

- Quali sono le caratteristiche di una buona funzione hash per il metodo di indirizzamento aperto?
- Si estende il concetto di uniformità semplice
- la h deve soddisfare la *proprietà di uniformità della funzione hash*:
  - per ogni chiave k la sequenza di scansione generata da h deve essere una qualunque delle m! permutazioni di {0,1,…,m-1}

Funzioni Hash
indirizzamento aperto

- è molto difficile scrivere funzioni h che rispettino la proprietà di uniformità
- si usano generalmente tre approssimazioni:
  - scansione lineare
  - scansione quadratica
  - hashing doppio
- tutte queste classi di funzioni garantiscono di generare una permutazione ma nessuna riesce a generare tutte le m! permutazioni

Scansione Lineare

- Data una funzione hash h':U → {0,1,…,m-1} il metodo di scansione lineare costruisce una h(k,i) nel modo seguente:

$$h(k,i)=(h'(k)+i) \bmod m$$

- data la chiave k si genera la posizione h'(k), quindi la posizione h'(k)+1, e così via fino alla posizione m-1.
- Poi si scandisce in modo circolare la posizione 0,1,2
- fino a tornare a h'(k)-1