

# Tabelle di hash per implementare dizionari



Queste slide integrano parti del materiale a corredo del libro di testo “Algoritmi e Strutture Dati” di Camil Demetrescu, Irene Finocchi, Giuseppe F. Italiano (cap. 7), rilasciato ai docenti dei corsi e protetto da copyright da parte della casa editrice McGraw-Hill. Vengono rese disponibili per ragioni didattiche agli studenti di ASD@Informatica-UniGE, che si impegnano a non rilasciarle ad altri e a non renderle pubbliche.

# Variabili usate in queste slide

$m$  = dimensione della tabella

$n$  = numero di elementi presenti nel dizionario

# Tabelle ad accesso diretto

Sono dizionari basati sulla proprietà di accesso diretto alle celle di un array

Idea:

- dizionario memorizzato in array  $v$  di  $m$  celle
- a ciascun elemento è associata una chiave intera nell'intervallo  $[0, m-1]$
- elemento con chiave  $k$  contenuto in  $v[k]$
- al più  $n \leq m$  elementi nel dizionario

# Tabelle ad accesso diretto: Implementazione

(complessità riferita a caso migliore e peggiore)

**classe** `TavolaAccessoDiretto` **implementa** `Dizionario`:

**dati:**

$$S(m) = \Theta(m)$$

un array  $v$  di dimensione  $m \geq n$  in cui  $v[k] = elem$  se c'è un elemento  $elem$  con chiave  $k$  nel dizionario, e  $v[k] = \text{null}$  altrimenti. Le chiavi  $k$  devono essere interi nell'intervallo  $[0, m - 1]$ .

**operazioni:**

`insert( $elem\ e, chiave\ k$ )`  
 $v[k] \leftarrow e$

$$T(n) = O(1)$$

`delete( $chiave\ k$ )`  
 $v[k] \leftarrow \text{null}$

$$T(n) = O(1)$$

`search( $chiave\ k$ )`  $\rightarrow elem$   
**return**  $v[k]$

$$T(n) = O(1)$$

# Fattore di carico

Misuriamo il grado di riempimento di una tabella usando il fattore di carico  $\alpha$

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

Esempio: tabella con i nomi delle 100 matricole che hanno ASD nel piano di studi, indicizzati da numeri di matricola a 6 cifre

$$n=100 \quad m=10^6 \quad \alpha = 0,0001 = 0,01\%$$

Grande spreco di memoria!

# Tabelle ad accesso diretto: pregi e difetti

## **Pregi:**

Tutte le operazioni, tranne la `createEmpty` che richiede  $\Theta(m)$ , richiedono tempo  $\Theta(1)$ . Questa considerazione vale anche per le tabelle di hash con funzione di hash perfetta.

## **Difetti:**

Le chiavi devono essere necessariamente interi in  $[0, m-1]$

Lo spazio utilizzato è proporzionale a  $m$ , non al numero  $n$  di elementi: può esserci grande spreco di memoria!

# Tabelle di hash

Per ovviare agli inconvenienti delle tabelle ad accesso diretto ne consideriamo un'estensione: le **tabelle di hash** (hash table)

## Idea:

- Chiavi prese da un universo totalmente ordinato  $U$  (possono non essere numeri)
- Funzione hash:  $h: U \rightarrow [0, m-1]$   
(funzione che trasforma chiavi in indici)
- Elemento con chiave  $k$  in posizione  $v[h(k)]$

# Requisito principale su $h$

- Deve essere calcolabile in tempo costante, altrimenti perdo ogni vantaggio nell'usarla!



# Osservazione sull'ordine delle chiavi nella struttura dati

- L'insieme  $U$  delle chiavi solitamente supporta una relazione di ordine totale.
- Tuttavia, il modo in cui immagazziniamo le coppie chiave-valore, **non richiede che sia rispettato alcun vincolo di ordinamento.**
- In alcune strutture dati per memorizzare insiemi e dizionari, si mantengono gli elementi ordinati perché questo può rendere più efficiente la ricerca, ma non perché ci sia qualche “obbligo” in tal senso.

# Tabelle di hash: collisioni

Le tabelle hash possono soffrire del fenomeno delle **collisioni**.

Si ha una collisione quando si deve inserire nella tabella hash un elemento con chiave  $u$ , e nella tabella esiste già un elemento con chiave  $v$  tale che  $h(u)=h(v)$ : il nuovo elemento andrebbe a sovrascrivere il vecchio.

# Funzioni di hash perfette

Un modo per evitare il fenomeno delle collisioni è usare funzioni hash perfette.

Una funzione hash si dice **perfetta** se è iniettiva, cioè per ogni  $u, v \in U$ :

$$u \neq v \Rightarrow h(u) \neq h(v)$$

Deve essere  $|U| \leq m$

# Tabella di hash perfetta: implementazione

(complessità riferita a caso migliore e peggiore)

**classe** TavolaHashPerfetta **implementa** Dizionario:

**dati:**  $S(m) = \Theta(m)$   
un array  $v$  di dimensione  $m \geq n$  in cui  $v[h(k)] = e$  se c'è un elemento  $e$  con chiave  $k \in U$  nel dizionario, e  $v[h(k)] = \text{null}$  altrimenti. La funzione  $h : U \rightarrow \{0, \dots, m - 1\}$  è una funzione hash perfetta calcolabile in tempo  $O(1)$ .

**operazioni:**

insert(*elem*  $e$ , *chiave*  $k$ )  $T(n) = O(1)$

$v[h(k)] \leftarrow e$

delete(*chiave*  $k$ )  $T(n) = O(1)$

$v[h(k)] \leftarrow \text{null}$

search(*chiave*  $k$ )  $\rightarrow$  *elem*  $T(n) = O(1)$

**return**  $v[h(k)]$

# Esempio con funzione di hash perfetta

Tabella hash con i nomi delle 100 matricole che hanno ASD nel piano di studi, aventi come chiavi numeri di matricola nell'insieme  $U=[234717, 235717]$

Funzione hash perfetta:  $h(k) = k - 234717$

$n=100$        $m=1000$        $\alpha = 0,1 = 10\%$

L'assunzione  $|U| \leq m$  necessaria per avere una funzione hash perfetta è raramente conveniente (o possibile)...

# Esempio con funzione di hash **non** perfetta

Tabella hash con elementi aventi come  
chiavi lettere dell'alfabeto  $U=\{A,B,C,\dots\}$

Funzione hash non perfetta (ma buona in  
pratica):  $h(k) = \text{ascii}(k) \bmod m$

# Esempio con funzione di hash non perfetta

Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
0x00	0	NULL null	0x20	32	Space	0x40	64	@	0x60	96	`
0x01	1	SOH Start of heading	0x21	33	!	0x41	65	A	0x61	97	a
0x02	2	STX Start of text	0x22	34	"	0x42	66	B	0x62	98	b
0x03	3	ETX End of text	0x23	35	#	0x43	67	C	0x63	99	c
0x04	4	EOT End of transmission	0x24	36	\$	0x44	68	D	0x64	100	d
0x05	5	ENQ Enquiry	0x25	37	%	0x45	69	E	0x65	101	e
0x06	6	ACK Acknowledge	0x26	38	&	0x46	70	F	0x66	102	f
0x07	7	BELL Bell	0x27	39	'	0x47	71	G	0x67	103	g
0x08	8	BS Backspace	0x28	40	(	0x48	72	H	0x68	104	h
0x09	9	TAB Horizontal tab	0x29	41	)	0x49	73	I	0x69	105	i
0x0A	10	LF New line	0x2A	42	*	0x4A	74	J	0x6A	106	j
0x0B	11	VT Vertical tab	0x2B	43	+	0x4B	75	K	0x6B	107	k
0x0C	12	FF Form Feed	0x2C	44	,	0x4C	76	L	0x6C	108	l
0x0D	13	CR Carriage return	0x2D	45	-	0x4D	77	M	0x6D	109	m
0x0E	14	SO Shift out	0x2E	46	.	0x4E	78	N	0x6E	110	n
0x0F	15	SI Shift in	0x2F	47	/	0x4F	79	O	0x6F	111	o
0x10	16	DLE Data link escape	0x30	48	0	0x50	80	P	0x70	112	p
0x11	17	DC1 Device control 1	0x31	49	1	0x51	81	Q	0x71	113	q
0x12	18	DC2 Device control 2	0x32	50	2	0x52	82	R	0x72	114	r
0x13	19	DC3 Device control 3	0x33	51	3	0x53	83	S	0x73	115	s
0x14	20	DC4 Device control 4	0x34	52	4	0x54	84	T	0x74	116	t
0x15	21	NAK Negative ack	0x35	53	5	0x55	85	U	0x75	117	u
0x16	22	SYN Synchronous idle	0x36	54	6	0x56	86	V	0x76	118	v
0x17	23	ETB End transmission block	0x37	55	7	0x57	87	W	0x77	119	w
0x18	24	CAN Cancel	0x38	56	8	0x58	88	X	0x78	120	x
0x19	25	EM End of medium	0x39	57	9	0x59	89	Y	0x79	121	y
0x1A	26	SUB Substitute	0x3A	58	:	0x5A	90	Z	0x7A	122	z
0x1B	27	FSC Escape	0x3B	59	;	0x5B	91	[	0x7B	123	{
0x1C	28	FS File separator	0x3C	60	<	0x5C	92	\	0x7C	124	
0x1D	29	GS Group separator	0x3D	61	=	0x5D	93	]	0x7D	125	}
0x1E	30	RS Record separator	0x3E	62	>	0x5E	94	^	0x7E	126	~
0x1F	31	US Unit separator	0x3F	63	?	0x5F	95	_	0x7F	127	DEL

# Esempio con funzione di hash **non** perfetta

Ad esempio, per  $m=11$ :  $h('C') = h('N')$

⇒

se volessimo inserire sia 'C' and 'N' nel dizionario avremmo una collisione!



# Uniformità delle funzioni hash

Per ridurre la probabilità di collisioni, una buona funzione hash dovrebbe essere in grado di distribuire in modo uniforme le chiavi nello spazio degli indici della tabella

Questo si ha ad esempio se la funzione hash gode della proprietà di uniformità semplice

Sono necessarie semplici nozioni di calcolo della probabilità per comprendere questa parte. Ci accontentiamo dell'intuizione....

La funzione  $h$  non deve “sovraffollare” alcune celle della tabella, lasciandone vuote altre

# Risoluzione delle collisioni

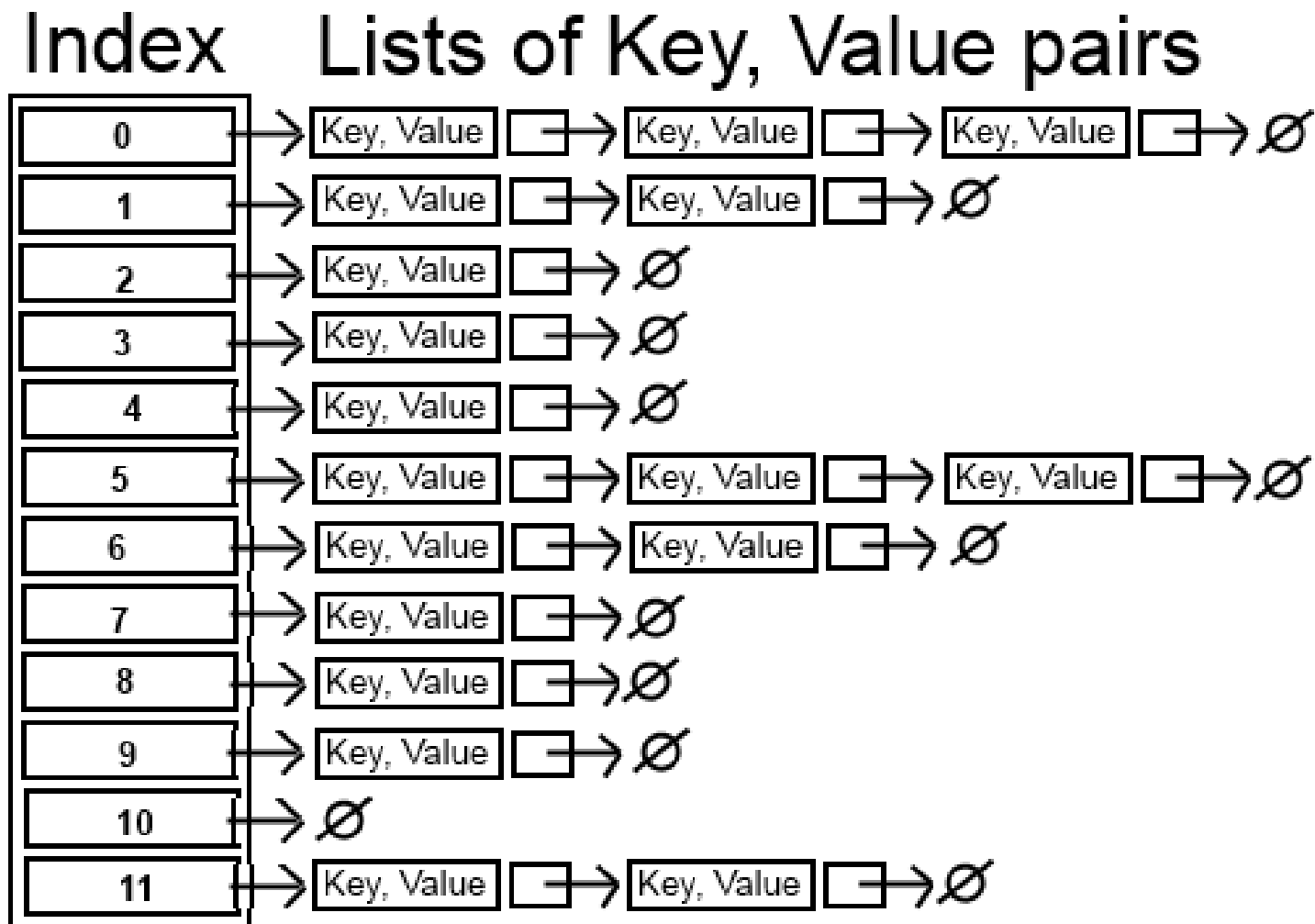
Nel caso in cui non si possano evitare le collisioni, dobbiamo trovare un modo per risolverle. Due metodi classici sono i seguenti:

1. **Liste di collisione.** Gli elementi sono contenuti in liste esterne alla tabella (chiamate bucket):  $v[i]$  punta alla lista degli elementi tali che  $h(k)=i$
2. **Indirizzamento aperto.** Tutti gli elementi sono contenuti nella tabella: se una cella è occupata, se ne cerca un'altra libera

# Terminologia standard...? Mah...

- Useremo la terminologia del libro di testo, ma segnaliamo che esiste un po' di confusione su cosa siano l'hashing (o indirizzamento) aperto e chiuso:
- Per il libro di testo l'**indirizzamento aperto** è quello in cui gli elementi sono contenuti nella tabella: se una cella è occupata, se ne cerca un'altra libera; l'approccio alternativo viene chiamato “con **liste di collisione**”
- Per Aho, Hopcroft, Ullman, “ a **closed hash table** keeps the members of the dictionary in the table itself, rather than using that table to store list headers”; al contrario “the basic data structure for **open hashing** ... is an array called the bucket table, indexed by bucket numbers, containing the headers for the lists....”
- Se parliamo di “liste di collisione” e di “tecnica che tiene tutti gli elementi nella tabella e richiede re-hashing” evitiamo ambiguità

# Liste di collisione



# Tabella di hash con liste di collisione: implementazione (complessità riferita al caso medio)

**classe** TavolaHashListeColl **implementa** Dizionario:

**dati:**  $S(m, n) = \Theta(m + n)$


un array  $v$  di dimensione  $m$  in cui ogni cella contiene un puntatore a una lista di coppie  $(elem, chiave)$ . Un elemento  $e$  con chiave  $k \in U$  è nel dizionario se e solo se  $(e, k)$  è nella lista puntata da  $v[h(k)]$ , con  $h : U \rightarrow \{0, \dots, m-1\}$  funzione hash con uniformità semplice calcolabile in tempo  $O(1)$ .

NOTA: Questa slide è presa dal libro di testo. Qui gli autori non hanno considerato che non si può inserire una coppia  $(e, k)$  se esiste già una coppia  $(e', k)$  nel dizionario: posso inserire  $(e, k)$  solo dopo aver verificato l'assenza di  $(e', k)$  (come facevamo per gli insiemi)

**operazioni:**

`insert(elem e, chiave k)`

aggiungi la coppia  $(e, k)$  alla lista puntata da  $v[h(k)]$ .


$$T(n) = O(1)$$

`delete(chiave k)`

rimuovi la coppia  $(e, k)$  nella lista puntata da  $v[h(k)]$ .

$$T_{avg}(n) = O(1 + n/m)$$

`search(chiave k) → elem`

se  $(e, k)$  è nella lista puntata da  $v[h(k)]$ , allora restituisci  $e$ , altrimenti restituisci null.

$$T_{avg}(n) = O(1 + n/m)$$

# Indirizzamento aperto

(da Aho-Hopcroft-Ullman)

Supponiamo di voler inserire un elemento con chiave  $k$  e la sua posizione “naturale”  $h(k)$  sia già occupata.

L'indirizzamento aperto consiste nell'occupare un'altra cella, anche se potrebbe spettare di diritto a un'altra chiave.

# Indirizzamento aperto

(da Aho-Hopcroft-Ullman)

Chiaramente, possiamo mettere un solo elemento in ogni cella. Associata alla strategia di indirizzamento aperto, ci deve essere una strategia di re-hashing.

Se proviamo a mettere  $x$  in una cella  $h(x)$  che contiene già un elemento, abbiamo una collisione. La strategia di re-hashing individua una sequenza di celle alternative,  $h_1(x)$ ,  $h_2(x)$  ....., in cui potremmo mettere  $x$ .

Analizziamo ciascuna di queste celle in ordine, fino a che non ne troviamo una vuota. Se nessuna cella è vuota, la tabella è piena e non possiamo inserire  $x$ .

# Indirizzamento aperto

(da Aho-Hopcroft-Ullman)

## Esempio:

Supponiamo che  $m = 8$  e le chiavi siano  $a, b, c, d$  (associate a valori che per l'esempio non ci interessano), con hash value  $h(a)=3$ ,  $h(b)=0$ ,  $h(c)=4$  e  $h(d)=3$ .

Usiamo la strategia di rehashing più semplice possibile, il re-hashing lineare, t c.  **$h_i(x) = (h(x) + i) \bmod m$** .

Assumiamo che all'inizio la tabella sia vuota (ogni cella contiene l'elemento EMPTYELEM).



# Indirizzamento aperto

(da Aho-Hopcroft-Ullman)

## Esempio:

Cosa succede se inseriamo a, b, c, d in questo ordine?

0	<i>b</i>
1	
2	
3	<i>a</i>
4	<i>c</i>
5	<i>d</i>
6	
7	

# Indirizzamento aperto

(da Aho-Hopcroft-Ullman)

La verifica che una chiave appartenga alla tabella richiede di ispezionare  $h_1(x)$ ,  $h_2(x)$ ,  $h_3(x)$ , ... finché o troviamo la chiave cercata, o troviamo EMPTYELEM.

Funziona se permettiamo solo inserimenti.... ma se ammettiamo cancellazioni, diventa più complicato!

Dobbiamo aggiungere una costante “DELETEDELEM” per indicare che la cella è vuota, ma perché l'elemento è stato cancellato.

Le celle che contengono DELETEDELEM possono essere riutilizzate in fase di inserimento.

# Note

Una tabella di hash implementata con liste di collisione è una struttura dati adatta a supportare non solo il TDD Dizionario, ma anche il TDD Insieme.