Corso di laurea in Informatica
Introduzione agli Algoritmi
A.A. 2023/2024

Dizionari: Introduzione

Tiziana Calamoneri

SAPIENZA
UNIVERSITÀ DI ROMA

Slides realizzate sulla base di quelle preparate da T. Calamoneri e G. Bongiovanni per il corso di Informatica Generale tenuto a distanza nell'A.A. 2019/20

Dizionari (1)

Un *dizionario* è una struttura dati che permette di gestire un insieme dinamico di dati, che di norma è un *insieme totalmente ordinato*, tramite queste tre sole operazioni:

- *insert*: si inserisce un elemento;
- *search*: si ricerca un elemento;
- *delete*: si elimina un elemento.

Dizionari (2)

Fra le strutture dati già descritte, le uniche che supportano in modo semplice (anche se non efficiente) queste tre operazioni sono gli array, le liste concatenate e le liste doppie. Infatti:

- le code (con o senza priorità, inclusi gli heap) e le pile non consentono né la ricerca né l'eliminazione di un elemento arbitrario,
- negli alberi, l'eliminazione di un elemento comporta la disconnessione di una parte dei nodi dall'altra e quindi è un'operazione che in genere richiede delle successive azioni correttive.

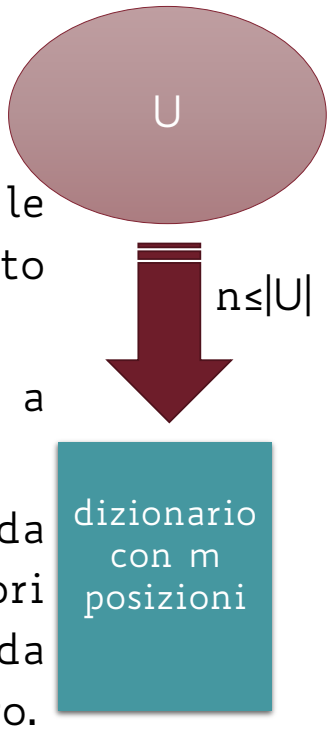
Dizionari (3)

Quando l'esigenza è quella di realizzare un dizionario, ossia una struttura dati che rispetti la definizione data sopra, si ricorre quindi a nuove soluzioni specifiche.

Qui ne illustriamo tre:

- tabelle ad indirizzamento diretto;
- tabelle hash;
- alberi binari di ricerca.

Dizionari (4)



Assunzioni e nomenclatura:

- *U* : insieme universo dei valori che le chiavi possono assumere; U è costituito da valori interi;
- *m* : numero delle posizioni a disposizione nella struttura dati;
- *n* : numero degli elementi da memorizzare nel dizionario; i valori delle chiavi degli elementi da memorizzare sono tutti diversi fra loro.

Tabelle ad indirizzamento diretto (1)

E' un array nel quale ogni indice corrisponde al valore della chiave dell'elemento da memorizzare in tale posizione.

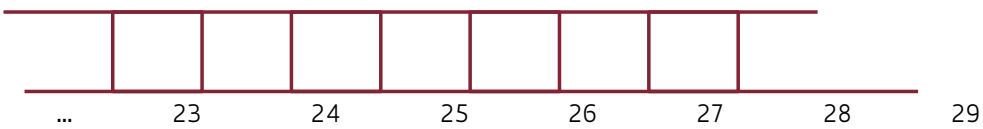


Tabelle ad indirizzamento diretto (2)

Nell'ipotesi $n \leq |U| = m$ un array A di m posizioni assolve al compito di dizionario con grande efficienza:

```
Insert_Indirizz_Diretto (A; k: chiave da ins.)
   A[k] ← dati dell'elemento di chiave k
   return

Search_Indirizz_Diretto (A; k: chiave da cerc.)
   return(dati dell'elemento di A con indice k)

Delete_Indirizz_Diretto (A; k: chiave da canc.)
   A[k] ← None
   return
```

Tutte e tre le operazioni hanno costo $\Theta(1)$.

Tabelle ad indirizzamento diretto (3)

Le cose non sono così semplici nei problemi reali:

- U può essere enorme, tanto da rendere impraticabile l'allocazione di un array A di sufficiente capienza;
- il numero delle chiavi effettivamente utilizzate può essere molto più piccolo di $|U|$, con rilevante spreco di memoria, in quanto la maggioranza delle posizioni dell'array A resta inutilizzata…

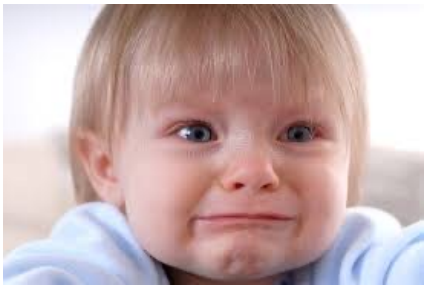


Tabelle ad indirizzamento diretto (4)

Esempio: Codici Fiscali (CF).

Un CF è costituito da 8 lettere (più una lettera di controllo) e 7 cifre. Considerando un alfabeto di 26 lettere, si hanno:

$$26^8 * 10^7 \approx 2*10^{18}$$

possibilità, mentre ci sono circa $6* 10^7$ cittadini. Anche considerando che non tutte le lettere e le cifre vengono usate in ogni posizione, la sproporzione è enorme.

Tabelle ad indirizzamento diretto (5)

E' necessario ricorrere a differenti implementazioni dei dizionari, a meno che non ci si trovi nelle condizioni che permettono l'uso dell'indirizzamento diretto.

Tabelle hash (1)

Si usano quando U (=valori che le chiavi possono assumere) è molto grande mentre l'insieme n delle chiavi da memorizzare effettivamente è molto più piccolo di U.

Idea: utilizzare un array di m posizioni.
Problema: non si può mettere direttamente in relazione la chiave con l'indice corrispondente, poiché le possibili chiavi sono molte di più rispetto agli indici.

Tabelle hash (2)

Sol.: Si definisce una opportuna funzione h, detta ***funzione hash***, che viene utilizzata per calcolare la posizione di un elemento sulla base del valore della sua chiave.

Tabelle hash (3)

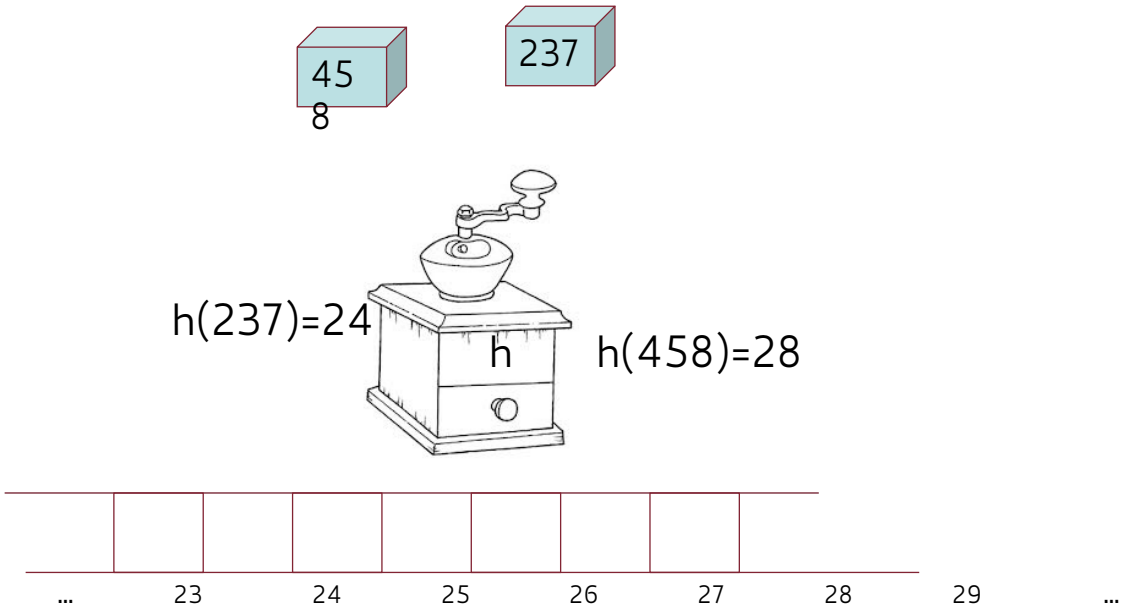


Tabelle hash (4)

Problema: anche se le chiavi da memorizzare sono meno di m, non si può escludere che due chiavi $k_1 \neq k_2$ siano tali che $h(k_1) = h(k_2)$, per cui andrebbero memorizzate nella stessa posizione della tabella.

Tale situazione viene chiamata ***collisione***.

NON c'è modo di evitare le collisioni; possiamo solo evitarle il più possibile e, altrimenti, risolverle.

Tabelle hash (5)

Una buona funzione hash deve rendere il più possibile **equiprobabili** i valori della funzione: tutti gli interi tra *0* ed *(m – 1)* dovrebbero essere prodotti con uguale probabilità.

In altre parole, la funzione dovrebbe far apparire come "casuale" il valore risultante, disgregando qualunque regolarità della chiave.

Inoltre, la funzione deve essere **deterministica**: se applicata più volte alla stessa chiave deve fornire sempre lo stesso risultato.

Tabelle hash (6)

La situazione ideale è quella in cui ciascuna delle *m* posizioni della tabella è scelta deterministicamente con la stessa probabilità: ipotesi di ***uniformità semplice della funzione hash***.

Sorvoliamo su come ottenere una funzione hash con l'ipotesi di uniformità semplice…

Tabelle hash (7)

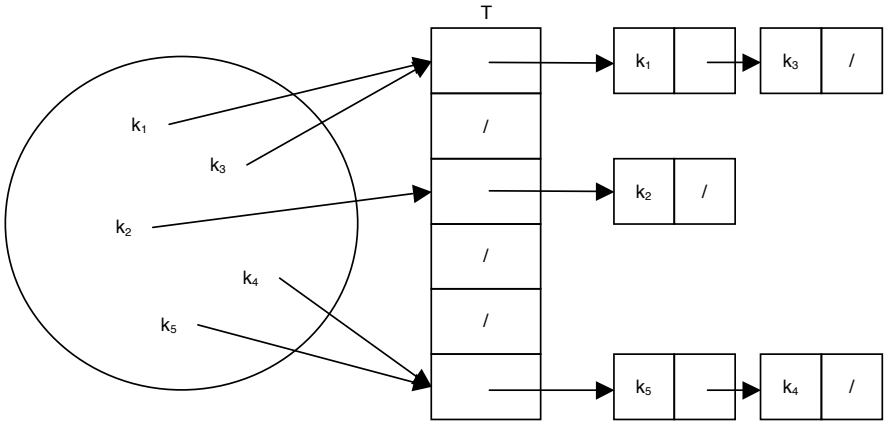
L'ipotesi di uniformità semplice minimizza il numero di collisioni ma queste possono comunque avvenire...

Infatti, per quanto bene sia progettata la funzione hash, è impossibile evitare del tutto le collisioni perché $|U| > m$ ed è inevitabile che esistano chiavi diverse che producono una collisione.

Dobbiamo, quindi, risolverle.

Liste di trabocco (1)

Prevedono di inserire tutti gli elementi le cui chiavi mappano nella stessa posizione in una lista concatenata, detta *lista di trabocco*.



Liste di trabocco (2)

Operazioni elementari:

```
Insert_Liste_Trabocco (A; k: chiave da ins.)
  inserisci i dati dell'elemento di chiave k
  nella lista puntata da A[h(k)]
  return
```

Costo computazionale: $\Theta(1)$, anche nel caso peggiore, con l'inserzione in testa alla lista.

Liste di trabocco (3)

Operazioni elementari (segue):

```
Search_ListeTrabocco (A; k: chiave da cercare)
  ricerca k nella lista puntata da A[h(k)]
  if k è presente
    then: return puntatore all'elemento contenente k
    else: return None
```

Costo computazionale: *O(lunghezza della lista puntata da A[h(k)])* che, nel **caso peggiore**, diviene $O(n)$ - quando tutti gli n elementi memorizzati nella tabella hash mappano nella medesima posizione, ma nel **caso medio** (con l'hp di uniformità semplice) è $O(n/m)$.

$n/m = \alpha$ - *fattore di carico* della tabella.

Liste di trabocco (4)

Operazioni elementari (segue):

```
Delete_ListeTrabocco (A; k: chiave da canc.)
   cancella i dati dell'elem. di chiave k
                dalla lista puntata da A[h(k)]
   return
```

Costo computazionale:
dipende dall'implementazione delle liste di trabocco e valgono, pertanto, tutte le osservazioni fatte per il costo dell'operazione di cancellazione nelle liste concatenate.

Indirizzamento aperto (1)

Questa tecnica prevede di inserire tutti gli elementi direttamente nella tabella, senza far uso di strutture dati aggiuntive.
Essa è applicabile quando:

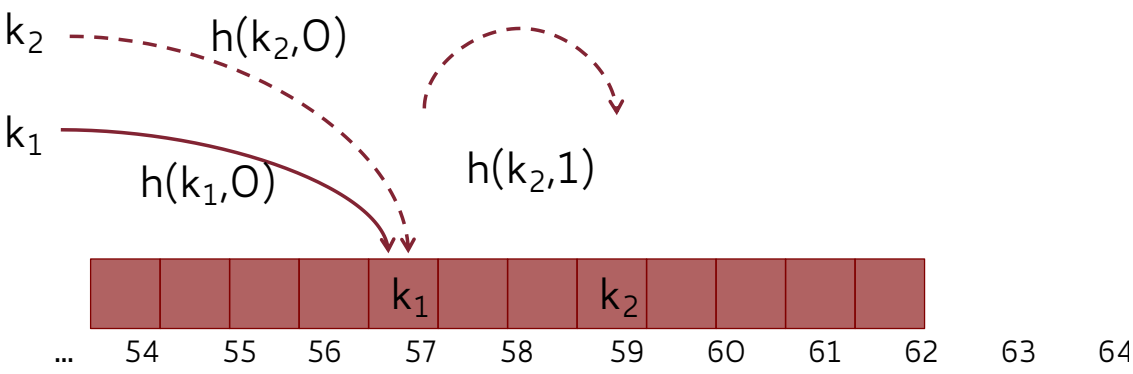
- m è maggiore o uguale al numero n di elementi da memorizzare (quindi il fattore di carico non è mai maggiore di 1);
- $|U| >> m$.

Indirizzamento aperto (2)

Idea: invece di avere una lista per ogni posizione dell'array, inseriamo solo all'interno dell'array stesso, calcolando (...) la sequenza delle posizioni da esaminare.

La funzione hash dipende ora da 2 parametri: la chiave *k* e il numero di collisioni già trovate.
Nota: la sequenza prodotta deve contenere una permutazione di tutti gli indici in modo da garantire che tutte le celle siano considerate.

Indirizzamento aperto (3)



Indirizzamento aperto (4)

Inserimento

Se la posizione iniziale relativa alla chiave *k* è occupata, si scandisce la tabella fino a trovare una posizione libera nella quale l'elemento con chiave *k* può essere memorizzato.

La scansione è guidata da una sequenza di funzioni hash ben determinata: h(k, 0), h(k, 1), …, h(k, m – 1).

Indirizzamento aperto (5)

inserimento (segue)

Costo computazionale: *O(lunghezza della sequenza che è necessario scandire)* che, nel **caso peggiore**, è *O(n)* -quando tutti gli *n* elementi memorizzati nella tabella hash mappano nella medesima posizione, ma nel **caso medio** (in caso di uniformità semplice) è $1/(1-\alpha)$
dove $\alpha = n/m \leq 1$ è il fattore di carico della tabella.

Indirizzamento aperto (6)

Ricerca

Si scandisce la tabella mediante la stessa sequenza di funzioni hash utilizzata per l'inserimento fino ad incontrare l'elemento cercato o una casella vuota (= l'elemento non è presente).

Costo nel caso di ricerca senza successo:
Come per l'inserimento: nel **caso peggiore** $O(n)$, ma nel **caso medio** (quando la funzione hash gode di uniformità semplice) $1/(1-\alpha)$
dove $\alpha = n/m$ è il fattore di carico della tabella.

Indirizzamento aperto (7)

ricerca (segue)

Costo nel caso di ricerca con successo:
nell'ipotesi di uniformità semplice, il numero di accessi **atteso** per una ricerca con successo è:

$$\frac{1}{\alpha}\log_e \frac{1}{1-\alpha} + \frac{1}{\alpha}$$

dove $\alpha = n/m$ è il fattore di carico.

Esempio: con una tabella piena al 50% il numero di accessi atteso è meno di 3,387; con una tabella piena al 90% il numero di accessi atteso è meno di 3,67; tutto questo indipendentemente dalle dimensioni della tabella!

Indirizzamento aperto (8)

ricerca (segue)

ATTENZIONE:

Anche se questa implementazione dei dizionari garantisce grande efficienza nel caso medio, NON si può dire che il costo di ogni operazione sia COSTANTE nel caso peggiore.

Questo deve essere tenuto in conto quando si calcola il costo computazionale di un algoritmo che faccia uso dei dizionari.

Indirizzamento aperto (9)

Cancellazione Operazione critica. Infatti:

- se si lascia la casella vuota, non si possono recuperare gli elementi memorizzati in caselle successive (una ricerca è senza successo quando si incontra una casella vuota...).
- se si marca la casella con un valore *deleted*, il costo computazionale della ricerca non dipende più esclusivamente dal fattore di carico ma anche dal numero delle posizioni precedentemente occupate da elementi e successivamente marcate.

Perciò di solito la cancellazione non è supportata con l'indirizzamento aperto.

Indirizzamento aperto (10)

Hashing doppio

Una tecnica usata nella pratica per generre funzioni adatte all'indirizzamento aperto è l'*hashing doppio*.

IDEA: Usare due diverse funzioni hash, una per determinare l'accesso iniziale alla tabella e l'altra per determinare il passo di scansione:

$$h(k, i) = [h_1(k) + i\ h_2(k)] \bmod m,\ i = 0, 1, \ldots, m-1$$

Funziona perché, se le due funzioni sono ben progettate, è estremamente improbabile che due chiavi $k_1 \neq k_2$ producano una collisione su *entrambe* le funzioni hash (nel qual caso le due chiavi scandirebbero l'intera tabella nello stesso modo, situazione indesiderabile).