Tabelle

Servono per implementare dizioniari in modo efficiente.

DIZIONARI insiemi di coppie (*chiave, valore*).

ESEMPIO: in un compilatore di un linguaggio di programmazione viene creata una tabella dei simboli, in cui le chiavi sono tutti gli identificatori del linguaggio e i valori sono i valori associati a ciascun identificatore.

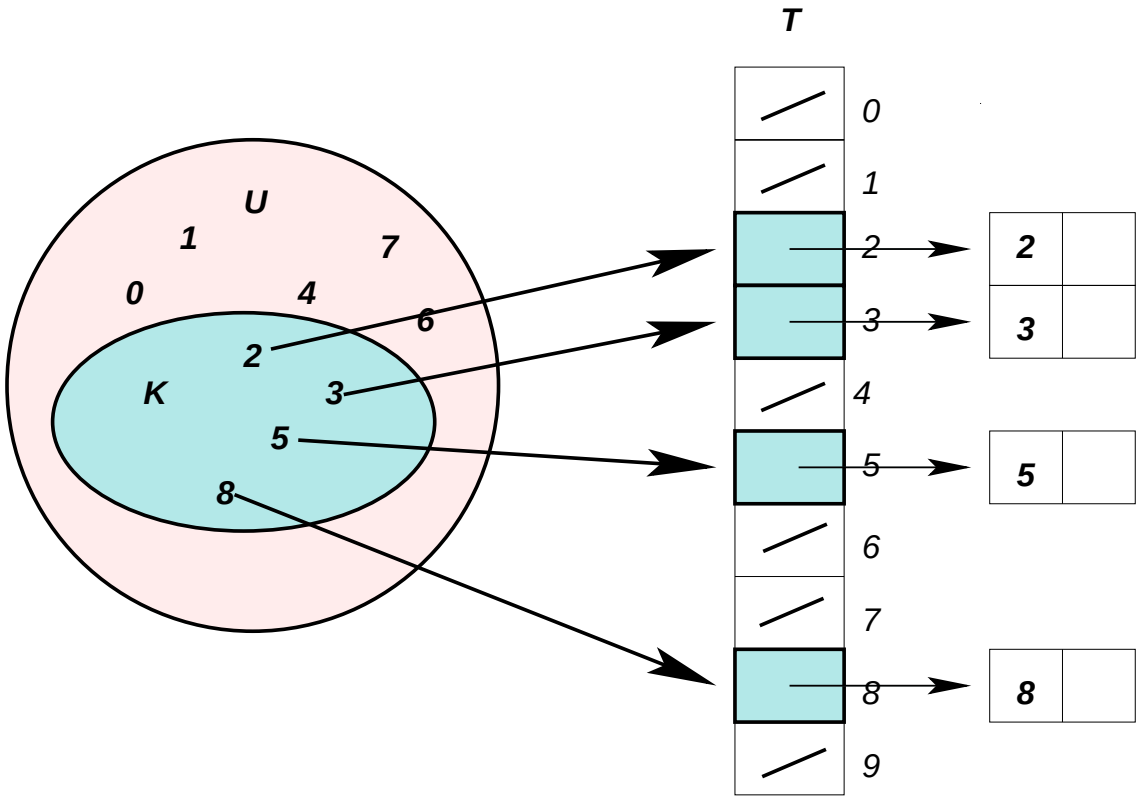
EFFICIENTE le operazioni di INSERT, SEARCH, DELETE possono essere realizzate, nel caso medio, in tempo $O(1)$.

TABELLE AD INDIRIZZAMENTO DIRETTO

Sia:

- U universo delle possibili chiavi, $U = \{0, 1, \ldots, m-1\}$,
- $T[0, 1, \ldots, m-1]$ una *tabella ad indirizzamento diretto* (array),
- K insieme delle chiavi effettive $K \subseteq U$.

L'insieme dinamico tramite una tabella T:



Se l'insieme non contiene la chiave k $T[k]=$ NIL.

OPERAZIONI DI BASE

k è una chiave, x è un puntatore

- DIRECT-ADDRESS-SEARCH(T, k) restituisce la componente di T di indice k: $T[k]$
- DIRECT-ADDRESS-INSERT(T, x) inserisce l'elemento x nella componente $key[x]$ dell'array T: $T[key[x]] \longleftarrow x$
- DIRECT-ADDRESS-DELETE(T, x) elimina l'elemento in posizione $key[x]$ sostituendo NIL: $T[key[x]] \longleftarrow$NIL

PROBLEMI NELLA RAPPRESENTAZIONE Se l'universo U è troppo grande la dimensione dell'array è improponbile.

ESEMPIO:
Consideriamo una tavola per la memorizzazione dei dati relativi agli studenti di una Facoltà. Supponiamo che gli studenti siano al massimo 3000 e che la chiave sia il cognome degli studenti (non più di 15 caratteri).

Il numero delle chiavi possibili sarebbe

$$26^{15}$$

quindi sarebbe necessario un array di

$$1.67725934228573e+21 \sim 2^{70}$$

componenti.

1 byte per componente $\longrightarrow 2^{70}$ *byte: 1Kb* $= 2^{10}$, *1Mb* $= 2^{20}$, *1Gb*$= 2^{30}$, *1Tb*$= 2^{40}$, ...

TABELLE HASH

Una funzione hash definisce una corrispondenza tra l'universo U delle chiavi e gli indici della **tabella hash** $T[0 \ldots m-1]$*:*

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

L'elemento $k \in U$ si trova nella posizione $h(k)$ nella tabella T.

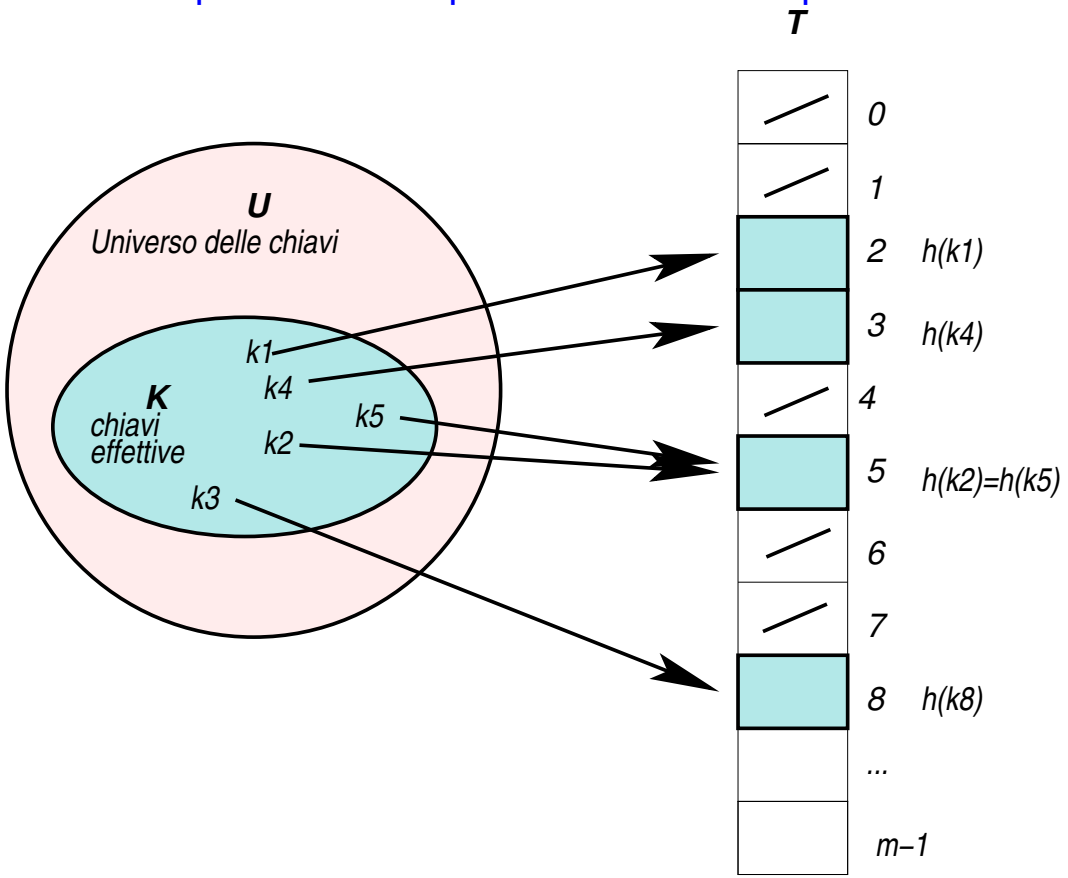
A quale funzione corrisponde h nel caso delle tabelle ad indirizzamento diretto?

Caratteristiche della funzione h

Ridurre l'intervallo degli indici che devono essere gestiti: invece di $\mid U \mid$ indici si vogliono avere $m << \mid U \mid$ indici.

Problema della collisione

Più chiavi possono corrispondere alla stessa posizione nell'array.

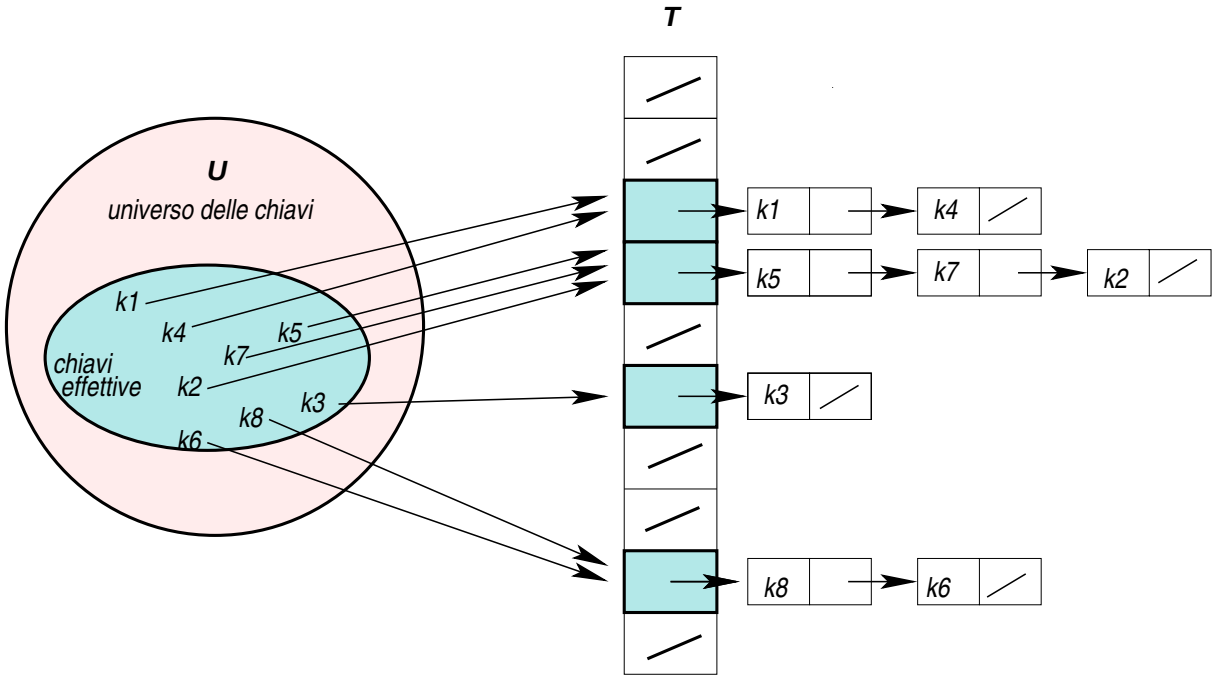


• h deve essere scelta in modo che le chiavi vengano **distiribuite in modo uniforme** sugli m indici a disposizione.

• h deve essere **deterministica**.

RISOLUZIONE DELLE COLLISIONI PER CONCATENAZIONE

- L'array T è un array di puntatori;
- $T[j]$ è un puntatore alla testa della lista che contiene tutti gli elementi con chiave k tale che $h(k) = j$



OPERAZIONI DI BASE

k è una chiave, x è un puntatore

- CHAINED-HASH-SEARCH(T, k) cerca un elemento con chiave k nella lista $T[h(k)]$ (riporta un puntatore);
- CHAINED-HASH-INSERT(T, x) inserisce l'elemento x in testa alla lista $T[h(key[x])]$ dell'array T;
- CHAINED-HASH-DELETE(T, x) cancella x dalla lista $T[h(key[x])]$.

COMPLESSITÀ DELLE OPERAZIONI DI BASE

- CHAINED-HASH-SEARCH(T, k) è proporzionale alla lunghezza della lista;
- CHAINED-HASH-INSERT(T, x) è $O(1)$ nel caso peggiore;
- CHAINED-HASH-DELETE(T, x) è $O(1)$ se le liste sono doppiamente collegate.

DELL'ORGANIZZAZIONE HASH CON COLLISIONI

Bisogna considerare

su quale

insieme di indici la funzione hash distribuisce le chiavi e

in quale

modo

(ipotesi: uniformità semplice della funzione hash).

Si definisce *fattore di carico* il rapporto tra il numero n di *elementi memorizzati* e il numero m di *posizioni disponibili*.

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

$\alpha =$ *numero medio di elementi memorizzati in ogni lista concatenata*

$\alpha < 1$ molte posizioni disponibili rispetto agli elementi memorizzati;

$\alpha = 1$ il numero di elementi corrisponde al numero delle componenti dell'array;

$\alpha > 1$ situazione attesa: molti elementi da memorizzare rispetto al numero delle posizioni disponibili.

Analisi del caso peggiore

Tutte le n chiavi corrispondono alla stessa posizione:

$$\forall k \in U, \quad h(k) = c$$

La complessità coincide con quella della lista concatenata $\Theta(n)$, più il tempo per il calcolo della funzione h.

... ma una funzione hash $h(k) = c$ non si definisce mai ...

Analisi del caso medio

Ipotesi:

- uniformità semplice della funzione hash: $h(k)$ distribuisce le chiavi in modo equiprobabile nell'array
- $h(k)$ è calcolata in un tempo $O(1)$.

In una tabella hash in cui le collisioni sono risolte per concatenazione e nell'ipotesi di uniformità semplice della funzione hash, sia una ricerca *senza* successo che una ricerca con successo richiedono in media $\Theta(1+\alpha)$.

... infatti nell'ipotesi di uniformità semplice
la lunghezza media di una lista coincide con il fattore di carico α

Fissato un universo di chiavi U e una relativa tabella hash, se il numero di posizioni è proporzionale al numero di elementi nella tabella ($n = O(m)$), si ha

$$\alpha = n/m = O(n/m) = O(1)$$

La ricerca richiede in media tempo costante.

Come definire buone funzioni hash

Ogni chiave corrisponde ad una delle m posizioni in modo equamente probabile

$P(k)$: probabilità che sia estratta una chiave k tale che $h(k) = j$.

$$\sum_{k:h(k)=j} P(k) = \frac{1}{m} \qquad \text{per} \quad j = 0, 1, \ldots m-1.$$

... perché ogni chiave ha la stessa probabilità di finire in posizione j

Una buona funzione hash è tale quando, applicata ad una chiave, genera dei numeri pseudocasuali uniformemente distribuiti nell'intervallo $1, \ldots, m$.

Esempio

Se le chiavi sono numeri reali pseudocasuali nell'intervallo $(0, 1)$ la funzione hash

$$h(k) = \lfloor km \rfloor$$

è una buona funzione hash:

per $m = 1000$, $\quad h(0.5217) = 521, h(0.1428) = 142, h(0.1121) = 112, h(0.1131) = 113, \ldots$

Qualità di una Buona Funzione Hash

Una buona funzione hash è tale quando per chiavi simili vengono generati indici diversi

Esempio

Supponiamo di dover creare la tavola dei simboli corrispondenti agli identificatori (solo car. alfabetici) presenti in un programma.

Sia l la max. lunghezza che può avere l'identificatore:
possono essere generati 26^l identificatori.

È usuale che in un programma ci siano identificatori simili (e.g. `pt`, `pts`, `app`, `app1`, `app2`): una buona funzione hash deve generare indici differenti per questi nomi.

Il valore riportato da una buona funzione hash deve essere indipendente dalla configurazione delle chiavi.

Metodi per generare buone funzioni hash (I)

Metodo di divisione $h(k) = k \bmod m$

DA EVITARE:

m **potenza di** 2 quando le chiavi sono numeri binari

m **potenza di** 10 quando le chiavi sono numeri decimali

Esempio: sia $m = 100$.
Per $k = 187622, 93822, 22, 3422, \quad h(k) = 22$

DA AUSPICARE:
m numero primo lontano da potenze di 2 o di 10.

Esempio: sia $m = 127$.

$$h(187622) = 43, \quad h(93822) = 96, \quad h(22) = 22, \quad h(3422) = 120$$

Metodi per generare buone funzioni hash (II)

Metodo di moltiplicazione $h(k) = \lfloor m(kA - \lfloor kA \rfloor) \rfloor$ (parte frazionaria), $A \in (0, 1)$, costante:

- un buon valore per A è $A = \frac{\sqrt{5}-1}{2} = 0.6180\ldots$ (Knuth)
- di solito $m = 2^p$

Hashing universale Si definisce un insieme di funzioni hash e a run-time se ne sceglie una. L'algoritmo si comporta diversamente per ogni esecuzione.

Se c'è una cattiva prestazione, non si verifica sempre (cfr. quick sort).

Sia $\mathcal{H}$ un insieme di funzioni da $U \longrightarrow \{0, 1, \ldots, m-1\}$. $\mathcal{H}$ è universale se $\forall (x, y) \in U$ il numero di funzioni hash $h \in \mathcal{H}$: $h(x) = h(y)$ è $\mid \mathcal{H} \mid /m$.

La probabilità di una collisione tra x e y è $1/m$ per $x \neq y$.

Sotto questa ipotesi si dimostra che:

Se una tabella hash ha più posizioni di quante siano le chiavi ($n \leq m$) e se la funzione fa parte di un insieme universale allora non ci sono collisioni tra le chiavi.

TABELLE HASH CON INDIRIZZAMENTO APERTO

- La rappresentazione non fa uso di puntatori.
- Il fattore di carico α non può mai superare 1.
- Tutto il nodo è contenuto nel'array T.
- Invece delle liste le collisioni vengono gestite memorizzando gli elementi *in altri punti* della hash stessa.
- Invece di seguire le collisioni si *calcola* la sequenza di posizioni da esaminare.
- Si utilizza meno memoria rispetto alla rappresentazione con le liste concatenate perché non ci sono puntatori.

Esempio

Per memorizzare 1000 elementi in un array di 500 componenti ($\alpha = 2$ servono 500 celle di memoria per le componenti dell'array + 1000 record per gli elementi. Ogni record è composto almeno da due celle: 1 pt alla chiave e 1 pt al record successivo. Quindi servono 2000 celle per gli elementi da memorizzare.

complessivamente servono 2500 celle

... che possono essere utilizzate per un array con 2500 celle adiacenti che puntano direttamente alle chiavi

Tabelle Hash con Indirizzamento Aperto - Inserzione

- La sequenza di posizioni esaminate dipende dalla chiave che deve essere inserita:
  $h : U \times \{0, 1, \ldots, m-1\} \longrightarrow \{0, 1, \ldots, m-1\}$

Hash-Insert(T, k) Ipotesi:

- non ci sono dati satellite,
- la tabella ha solo la chiave k.

```
Hash-Insert(T,k)

1. i <- 0
2. repeat j <- h(k,i)  ;; i=0,1,...,m-1
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 sulla tabella hash'
```

Hash-Search(t,k)

- La sequenza di posizioni esaminata quando la chiave è stata inserita è esattamente la stessa sequenza esaminata quando la chiave viene ricercata.
- Il fallimento della ricerca avviene quando una posizione viene individuata vuota. Infatti se la chiave ci fosse stata (essendo la posizione vuota, sarebbe stata inserita proprio in quella posizione).

```
Hash-Search(T,k)

1. i <- 0
2. repeat j <- h(k,i)
3.    if T[j] = k
4.       then  return j
5.    i <- i+1
6. until T[j] = Nil  o  i=m
7. return Nil
```

Tabelle Hash con Indirizzamento Aperto - Cancellazione

- Quando si cancella una chiave la componente si deve marcare con `Deleted` e non con `Nil`, altrimenti si può interrompere erroneamente la scansione nella ricerca.
- Si dovrebbe anche modificare l'inserzione per riutilizzare questa componente.
- Le tabelle hash con indirizzamento aperto, di solito, si usano quando **non** sono previste cancellazioni.

Scansione lineare: $h(k,i) = (h'(k) + i) \bmod m$ per $i = 0, 1, \ldots, m-1$.

Prima si scandiscono tutte le posizioni generate da $T[(h'(k)], T[(h'(k)+1]$, fino a $T[m-1]$, poi si riparte da $T[0]$ finché non si raggiunge $T[h'(k)-1]$.

Scansione quadratica: $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$ dove $c_1, c_2 \neq 0$, $i = 0, 1, \ldots, m-1$.

Hashing doppio: $h(k,i) = (h_1(k) + i h_2(k)) \bmod m$ dove h_1 e h_2 sono due funzioni hash. Il fatto che la posizione della chiave e la distanza dalla chiave successiva varino secondo due distinte funzioni, rende il comportamento di queste funzioni, vicino allo schema ideale della funzione hash uniforme.

Analisi

- Nel caso dell'indirizzamento aperto il fattore di carico è $\alpha \leq 1$.
- Nell'ipotesi di una funzione hash uniforme ogni possibile permutazione delle sequenze di scansione, per ogni chiave k, è equiprobabile per un'inserzione o una ricerca.
- Nell'ipotesi precedente si dimostra che *se la tabella ha un fattore di carico $\alpha < 1$ allora il numero medio di accessi di una* **ricerca senza successo** *è al più* $1/(1-\alpha)$.
- e che *se la tabella ha un fattore di carico $\alpha < 1$ il numero medio di accessi di una* **ricerca con successo** *è*

$$\frac{1}{\alpha} ln \frac{1}{1-\alpha}$$

Esercizi

VI-1 Si consideri l'inserzione delle chiavi $10, 22, 31, 4, 15, 28, 17, 88, 59$ in una tabela hash di lunghezza $m = 11$ usando l'indirizzamento aperto con funzione hash $h'(k) = k \bmod m$. Si illustri il risultato dell'inserimento di queste chiavi usando la scansione lineare, usando la scansione quadratica con $c_1 = 1$ e $c_2 = 3$ e usando l'hashing doppio con $h_2(k) = 1 + (k \bmod (m - 1))$.

VI-2 Scrivere in pseudocodice la procedura Hash-Delete, Hash-insert e Hash-Search perché prevedano il valore speciale DELETED.

Esercizio d'esame del 21 aprile 2004 - rosa Nel caso della gestione delle tabelle hash con concatenazione, si illustri l'inserimento in una tabella hash di $m = 10$ componenti, le chiavi $32, 17, 19, 31, 33, 15, 38, 46$, utilizzando la seguente funzione:

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

La bontà di una funzione hash dipende dallo specifico insieme di chiavi oppure è indipendente da esso?

Dire se la funzione dell'esercizio in questione è o meno una buona funzone hash, giustificando la risposta.

Esercizio d'esame del 21 aprile 2004 - giallo Relativamente alla gestione delle tabelle hash con indirizzamento aperto, si illustri l'inserimento in una tabella hash di $m = 7$ componenti, le chiavi $34, 12, 15, 26, 2, 32, 40$, utilizzando l'hashing doppio con le seguenti funzioni:

$$h'(k) = k \ mod \ 7 = h_1(k), \quad h_2(k) = 1 + (k \ mod \ 6)$$

Si ricorda che $h(k, i) = (h_1(k) + i \cdot h_2(k)) \ mod \ m$.

Quando si verifica la coincidenza tra l'hashing doppio e la scansione lineare?