Implementazione di dizionari

Problema del dizionario dinamico

Scegliere una struttura dati in cui memorizzare dei record con un campo ***key*** e alcuni altri campi in cui sono memorizzati i dati associati alla chiave ***key***.

Su tale struttura si devono poter eseguire in modo efficiente le operazioni:

Insert che aggiunge un nuovo record

Search che cerca un record di chiave ***key***

Delete che toglie un record dalla struttura

Tavole ad indirizzamento diretto

Funzionano bene quando le chiavi sono degli interi positivi non troppo grandi.

Ad esempio se le chiavi appartengono all'insieme $U = \{0,1,...,m\text{-}1\}$ (con m non troppo grande) possiamo usare un array $T[0..m\text{-}1]$ in cui ogni posizione (cella) $T[k]$ corrisponde ad una chiave k.

Generalmente $T[k]$ è un puntatore al record con chiave k.

Se la tavola non contiene un record con chiave k allora $T[k] = nil$.

La realizzazione delle operazioni in tempo costante $O(1)$ è semplice:

Search(T,k)
  return $T[k]$

Insert(T,x)
  $T[x.key] = x$

Delete(T,x)
  $T[x.key] = nil$

Inconvenienti

Con l'indirizzamento diretto occorre riservare memoria sufficiente per tante celle quante sono le possibili chiavi.

Se l'insieme U delle possibili chiavi è molto grande l'indirizzamento diretto è inutilizzabile a causa delle limitazioni di memoria.

Anche quando la memoria sia sufficiente se le chiavi memorizzate nel dizionario sono soltanto una piccola frazione di U la maggior parte della memoria riservata risulta inutilizzata.

Tavole hash

Una *tavola hash* richiede memoria proporzionale al numero massimo di chiavi presenti nel dizionario indipendentemente dalla cardinalità dell'insieme U di tutte le possibili chiavi.

In una tavola hash di m celle ogni chiave k viene memorizzata nella cella $h(k)$ usando una funzione

$$h : U \rightarrow \{0..m\text{-}1\}$$

detta *funzione hash*.

Siccome $|U| > m$ esisteranno molte coppie di chiavi distinte $k_1 \neq k_2$ tali che $h(k_1) = h(k_2)$.

Diremo in questo caso che vi è una *collisione* tra le due chiavi k_1 e k_2.

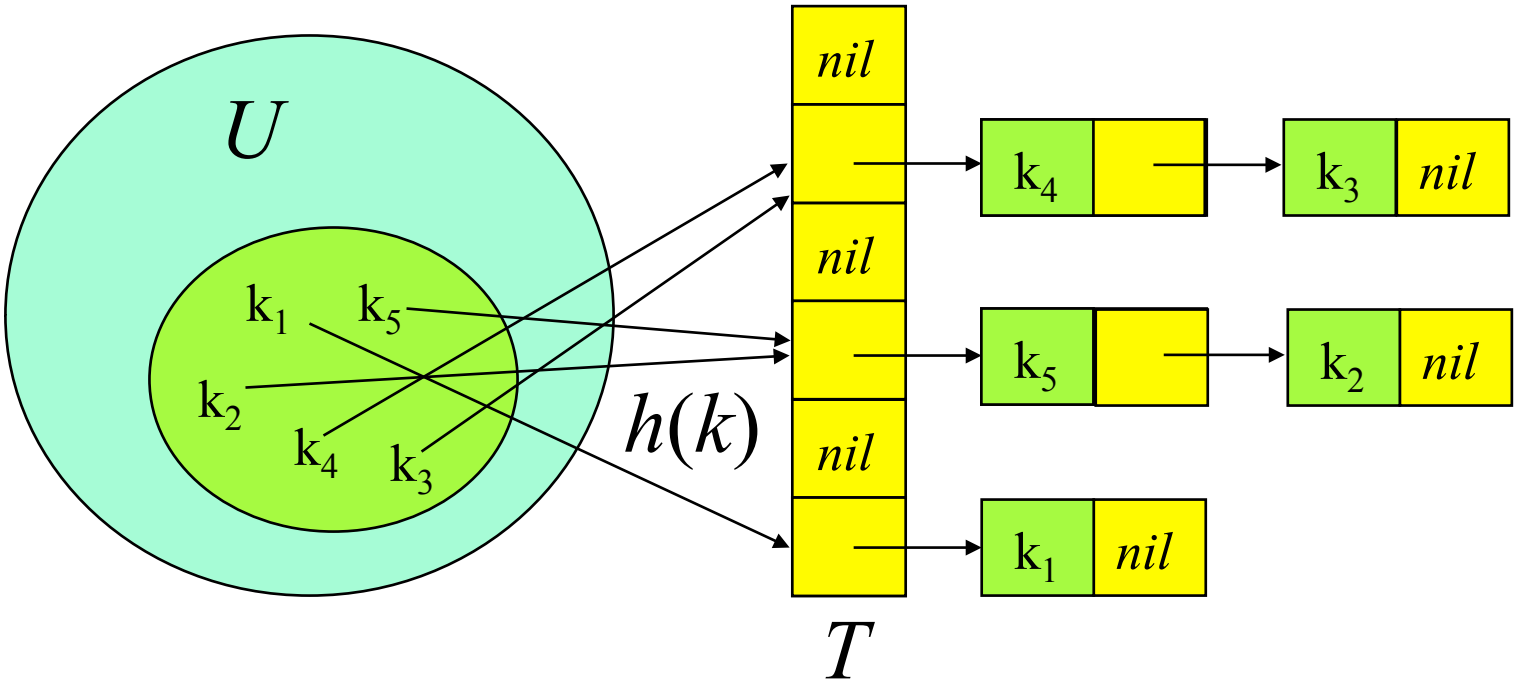
Nessuna funzione hash può evitare le collisioni.

Dovremo quindi accontentarci di funzioni hash che minimizzino la probabilità delle collisioni e, in ogni caso, dovremo prevedere qualche meccanismo per gestire le collisioni.

Risoluzione delle collisioni con liste

Gli elementi che la funzione hash manda nella stessa cella vengono memorizzati in una lista

La tavola hash è un array $T[0..m\text{-}1]$ di m puntatori alle cime delle liste



La realizzazione delle operazioni è facile:

Search(*T*,*k*)

"cerca nella lista *T*[*h*(*k*)] un elemento *x* tale che *x.key* == *k*"

return *x*

Insert(*T*,*x*)

"aggiungi *x* alla lista *T*[*h*(*x.key*)]"

Delete(*T*,*x*)

"togli *x* dalla lista *T*[*h*(*x.key*)]"

Search richiede tempo proporzionale alla lunghezza della lista $T[h(k)]$

Insert si può realizzare in tempo $O(1)$

Delete richiede una ricerca con Search dopo di che l'eliminazione dell'elemento dalla lista si può realizzare in tempo $O(1)$

Analisi di hash con liste

Supponiamo che la tavola hash T abbia m celle e che in essa siano memorizzati n elementi.

Una Search di un elemento con chiave k richiede tempo $O(n)$ nel caso pessimo in cui tutti gli n elementi stanno nella stessa lista $h(k)$ della chiave cercata.

Valutiamo la complessità media di Search in funzione del *fattore di carico* $\alpha = n/m$.

Siccome n è compreso tra 0 e $|U|$ $\quad 0 \leq \alpha \leq |U|/m$.

Supporremo che $h(k)$ distribuisca in modo uniforme le n chiavi tra le m liste.

Più precisamente assumeremo la seguente ipotesi di hash uniforme semplice.

"ogni elemento in input ha la stessa probabilità di essere mandato in una qualsiasi delle m celle"

Siano $n_0, n_1, \ldots, n_{m-1}$ le lunghezze delle m liste.
La lunghezza attesa di una lista è

$$\mathrm{E}[n_j] = \frac{1}{m}\sum_{i=0}^{m-1} n_i = \frac{n}{m} = \alpha$$

Proprietà: Nell'ipotesi di hash uniforme semplice la ricerca di una chiave k *non presente* nella tavola hash richiede tempo $\Theta(1+\alpha)$ in media.

Dimostrazione:

La *Search* calcola $j = h(k)$ (tempo $\Theta(1)$) e poi controlla tutti gli n_j elementi della lista $T[j]$ (tempo $\Theta(n_j)$).

Nell'ipotesi di hash uniforme semplice $E[n_j] = \boldsymbol{\alpha}$ e quindi l'algoritmo richiede tempo medio $\Theta(1+\boldsymbol{\alpha})$.

Proprietà: Nell'ipotesi di hash uniforme semplice la ricerca di una chiave k presente nella tavola hash richiede tempo $\Theta(1+\alpha)$ in media.

Dimostrazione:

Assumiamo che ogni chiave presente nella tavola abbia la stessa probabilità di essere la chiave cercata.

Una ricerca di un una chiave k presente nella tavola richiede il calcolo dell'indice $j = h(k)$, il test sulle chiavi che precedono k nella lista $T[j]$ e infine il test su k (numero operazioni = 2 + numero elementi che precedono k nella lista).

Le chiavi che precedono k nella lista $T[j]$ sono quelle che sono state inserite dopo di k.

Supponiamo che $k = k_i$ sia l'i-esima chiave inserita nella lista.

Per $j = i + 1, \ldots, n$ sia $X_{i,j}$ la variabile casuale che vale 1 se k_j viene inserita nella stessa lista di k e 0 altrimenti

$$X_{i,j} = \begin{cases} 0 & \text{se } h(k_j) \neq h(k_i) \\ 1 & \text{se } h(k_j) = h(k_i) \end{cases}$$

Nell'ipotesi di hash uniforme semplice $\mathrm{E}(X_{i,j}) = 1/m$.

Il valore atteso del numero medio di operazioni eseguite è

$$\mathrm{E}[\frac{1}{n}\sum_{i=1}^{n}(2+\sum_{j=i+1}^{n}X_{i,j})]=\frac{1}{n}\sum_{i=1}^{n}(2+\sum_{j=i+1}^{n}\mathrm{E}[X_{i,j}])$$

$$=\frac{1}{n}\sum_{i=1}^{n}(2+\sum_{j=i+1}^{n}\frac{1}{m})=2+\frac{1}{nm}\sum_{i=1}^{n}(n-i)$$

$$=2+\frac{1}{nm}\sum_{t=0}^{n-1}t=2+\frac{1}{nm}\frac{n(n-1)}{2}$$

$$=2+\frac{n-1}{2m}=2+\frac{\alpha}{2}-\frac{1}{2m}=\Theta(1+\alpha)$$

Se $n \leq cm$ per qualche costante positiva c allora $\alpha = O(1)$ e $\Theta(1+\alpha) = \Theta(1)$.

Funzioni hash

Che proprietà deve avere una una buona funzione hash?

Essa dovrebbe soddisfare l'ipotesi di hash uniforme semplice:

"Ogni chiave ha la stessa probabilità $1/m$ di essere mandata in una qualsiasi delle m celle, indipendentemente dalle chiavi inserite precedentemente"

Ad esempio, se le chiavi sono numeri reali x estratti casualmente e indipendentemente da una distribuzione di probabilità uniforme in $0 \leq x < 1$ allora $h(x) = \lfloor mx \rfloor$ soddisfa tale condizione.

Sfortunatamente l'ipotesi di hash uniforme semplice dipende dalle probabilità con cui vengono estratti gli elementi da inserire; probabilità che in generale non sono note.

Le funzioni hash che descriveremo assumono che le chiavi siano degli interi non negativi.

Questo non è restrittivo in quanto ogni tipo di chiave è rappresentato nel calcolatore con una sequenza di bit e ogni sequenza di bit si può interpretare come un intero non negativo.

Metodo della divisione

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

Difetto: non *"funziona bene"* per ogni m.

Ad esempio se $m = 2^p$ è una potenza di due allora $k \bmod m$ sono gli ultimi p bit di k.

In generale anche valori di m prossimi ad una potenza di 2 non funzionano bene.

Una buona scelta per m è un numero primo non troppo vicino ad una potenza di 2.

Ad esempio $m = 701$.

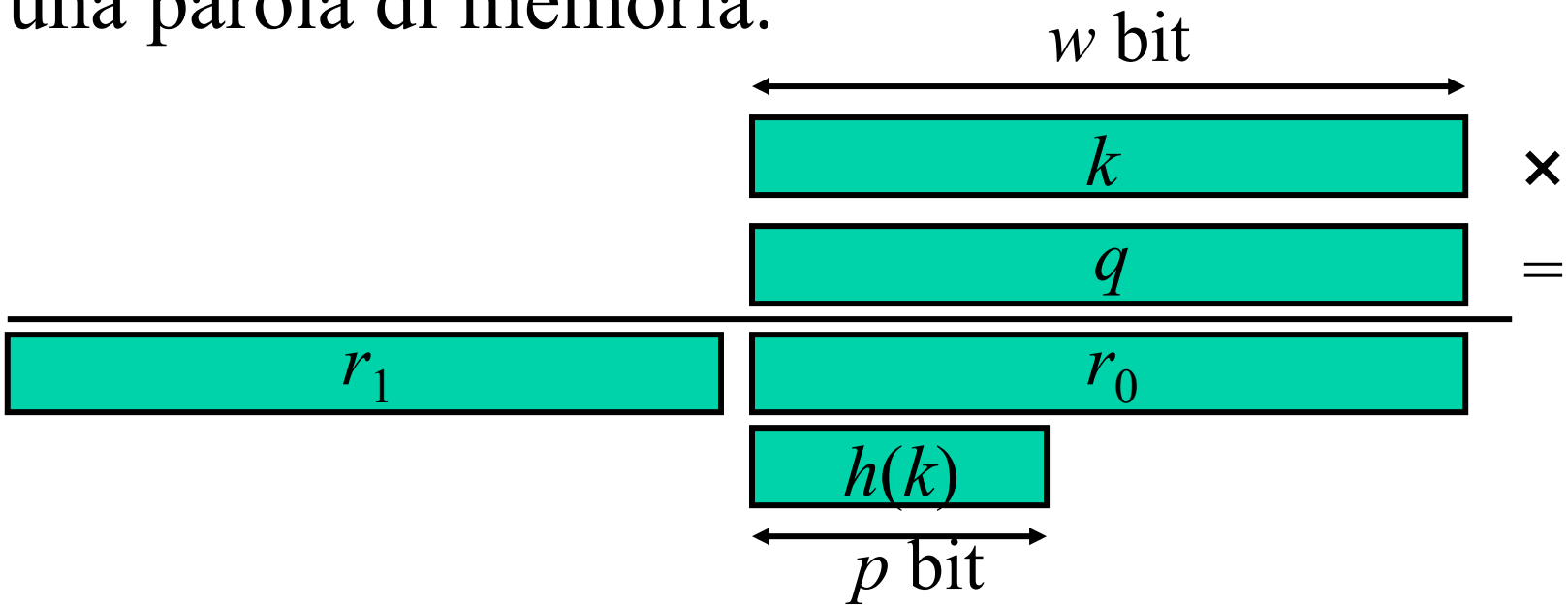
Metodo della moltiplicazione

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

in cui A è una costante reale con $0 < A < 1$ ed $x \bmod 1 = x - \lfloor x \rfloor$ è la parte frazionaria.

Vantaggi : la scelta di m non è critica e nella pratica funziona bene con tutti i valori di A anche se ci sono ragioni teoriche per preferire $A = (\sqrt{5} - 1)/2$ l'inverso del rapporto aureo

$h(k)$ si calcola facilmente se si sceglie $m = 2^p$ e $A = q/2^w$ con $0 < q < 2^w$ dove w è la lunghezza di una parola di memoria.



$$h(k) = \left\lfloor 2^p \frac{r_0}{2^w} \right\rfloor = \left\lfloor 2^p \left(\frac{kq}{2^w} \bmod 1 \right) \right\rfloor$$

$$= \lfloor m(kA \bmod 1) \rfloor$$

Randomizzazione di funzioni hash

Nessuna funzione hash può evitare che un avversario malizioso inserisca nella tavola una sequenza di valori che vadano a finire tutti nella stessa lista.

Più seriamente: per ogni funzione hash si possono trovare delle distribuzioni di probabilità degli input per le quali la funzione non ripartisce bene le chiavi tra le varie liste della tavola hash.

Possiamo usare la randomizzazione per rendere il comportamento della tavola hash indipendente dall'input.

L'idea è quella di usare una funzione hash scelta casualmente in un insieme "*universale*" di funzioni hash.

Questo approccio viene detto *hash universale*.

Un insieme H di funzioni hash che mandano un insieme U di chiavi nell'insieme $\{0,1,\ldots,m\text{-}1\}$ degli indici della tavola hash si dice *universale* se:

"per ogni coppia di chiavi distinte j e k vi sono al più $|H|/m$ funzioni hash in H tali che $h(j) = h(k)$"

Se scegliamo casualmente la funzione hash in un insieme universale H la probabilità che due chiavi qualsiasi j e k collidano è $1/m$, la stessa che si avrebbe scegliendo casualmente le due celle in cui mandare j e k.

Proprietà : Supponiamo che la funzione hash h sia scelta casualmente in un insieme universale H e venga usata per inserire n chiavi in una tavola T di m celle e sia k una chiave qualsiasi.
La lunghezza attesa $E[n_{h(k)}]$ della lista $h(k)$ è $\alpha = n/m$ se k non è presente nella tavola ed è minore di $\alpha+1$ se k è presente.

Quindi, indipendentemente dalla distribuzione degli input, una Search richiede tempo medio $\Theta(1+\alpha)$ che, se $n = O(m)$, è $\Theta(1)$.