Algoritmi e Strutture di Dati

Tabelle Hash

m.patrignani

Nota di copyright

- queste slides sono protette dalle leggi sul copyright
- il titolo ed il copyright relativi alle slides (inclusi, ma non limitatamente, immagini, foto, animazioni, video, audio, musica e testo) sono di proprietà degli autori indicati sulla prima pagina
- le slides possono essere riprodotte ed utilizzate liberamente, non a fini di lucro, da università e scuole pubbliche e da istituti pubblici di ricerca
- ogni altro uso o riproduzione è vietata, se non esplicitamente autorizzata per iscritto, a priori, da parte degli autori
- gli autori non si assumono nessuna responsabilità per il contenuto delle slides, che sono comunque soggette a cambiamento
- questa nota di copyright non deve essere mai rimossa e deve essere riportata anche in casi di uso parziale

Sommario

- Richiami sui tipi astratti di dato
  - array associativo
  - insieme
- Tabelle hash
  - collisioni e liste di trabocco
  - uso per la realizzazione di tipi astratti di dato
- Funzioni hash
  - per interi, per stringhe, per oggetti

Array associativi

- Gli array associativi sono tipi astratti di dato costituiti da coppie <chiave,valore>
  - le operazioni che vogliamo fare sono l'inserimento di una nuova coppia, la cancellazione di una specifica chiave e la ricerca del valore corrispondente ad una chiave
- Possono essere realizzati con coppie di array, liste, alberi binari di ricerca, alberi rosso-neri
- Le realizzazioni di array associativi si prestano anche a realizzare insiemi generici
  - basta omettere il valore e tenere solo la chiave
  - le operazioni supportate sono inserisci elemento, cancella elemento e verifica esistenza elemento

Realizzazioni efficienti di array associativi

- La realizzazione più efficiente che conosciamo degli array associativi utilizza alberi rosso-neri
  - garantisce un tempo $\Theta(\log n)$ per la ricerca, l'inserimento e la cancellazione nel caso peggiore e nel caso medio
  - occorre che sulle chiavi siano definite le funzioni MINORE e UGUALE
- Tutti gli algoritmi di ricerca basati su confronti hanno complessità $\Omega(\log n)$ nel caso peggiore
  - la ricerca è l'operazione più frequente e critica

Tabelle hash

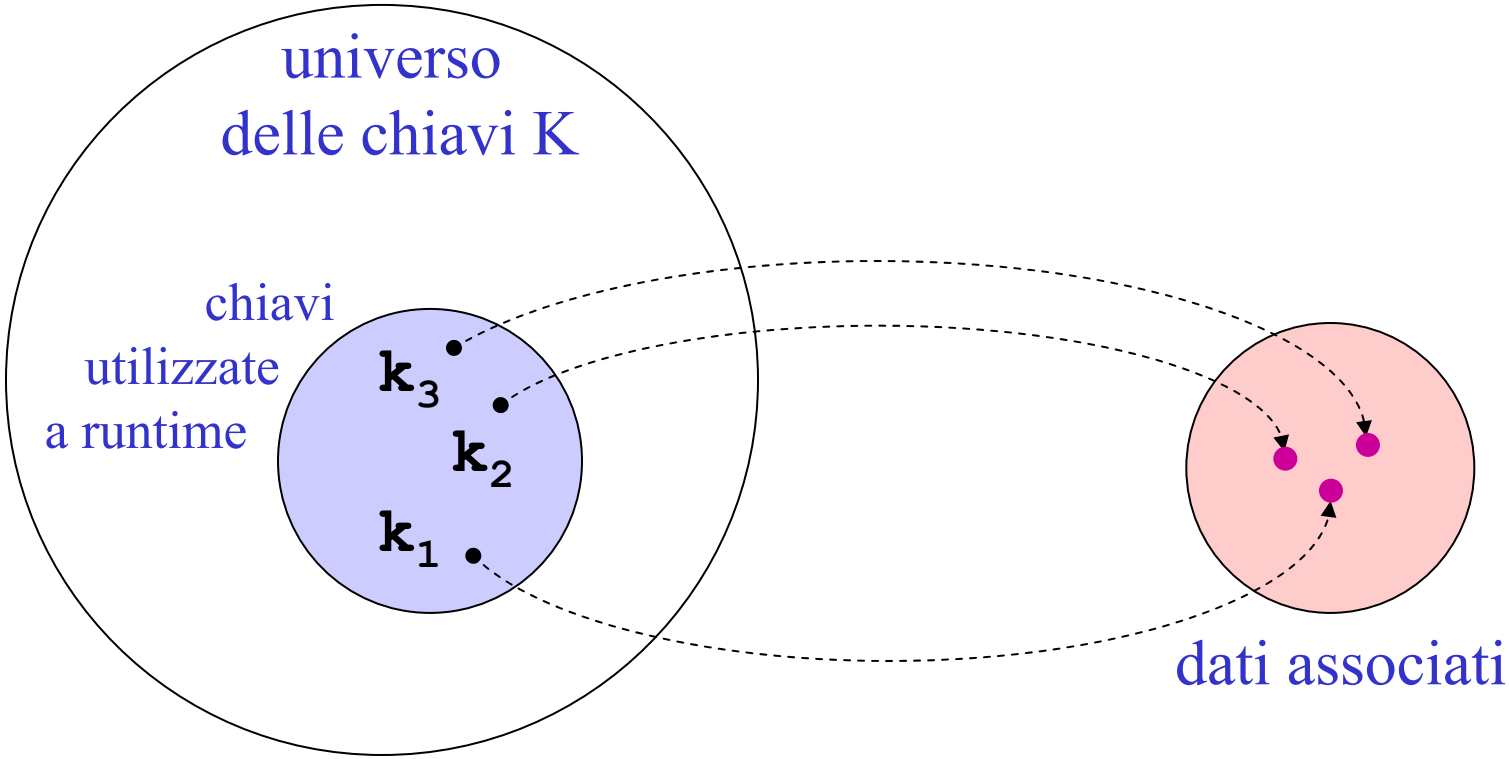
- Le tabelle hash realizzano array associativi con le seguenti caratteristiche
  - gli algoritmi di ricerca, inserimento e cancellazione non sono basati su confronti
  - le ricerche, gli inserimenti e le cancellazioni avvengono con un tempo $\Theta(n)$ nel caso peggiore
    - peggiorativo rispetto agli alberi rosso-neri
  - le ricerche, gli inserimenti e le cancellazioni avvengono con un tempo $\Theta(1)$ nel caso medio
    - migliorativo rispetto agli alberi rosso-neri
  - la struttura di dati utilizzata è effettivamente un singolo array

Difficoltà da superare nella realizzazione

- Due difficoltà principali
  1. la tipologia delle chiavi
     - le chiavi non sono necessariamente degli interi
     - non possiamo confidare nelle chiavi per indicizzare direttamente un array
  2. la numerosità delle possibili chiavi
     - anche se le chiavi fossero degli interi, un array in grado di contenere tutte le chiavi sarebbe troppo grande e troppo sparso
       - per esempio se la chiave fosse un numero di matricola di sei cifre dovrei allocare un array con un milione di posizioni anche se gli studenti del corso che voglio considerare sono solo qualche centinaio

Array associativi: osservazione

- Il numero di chiavi effettivamente utilizzate dal programma in esecuzione è molto minore del numero delle chiavi possibili
  - quest'ultimo è chiamato "universo" delle chiavi K

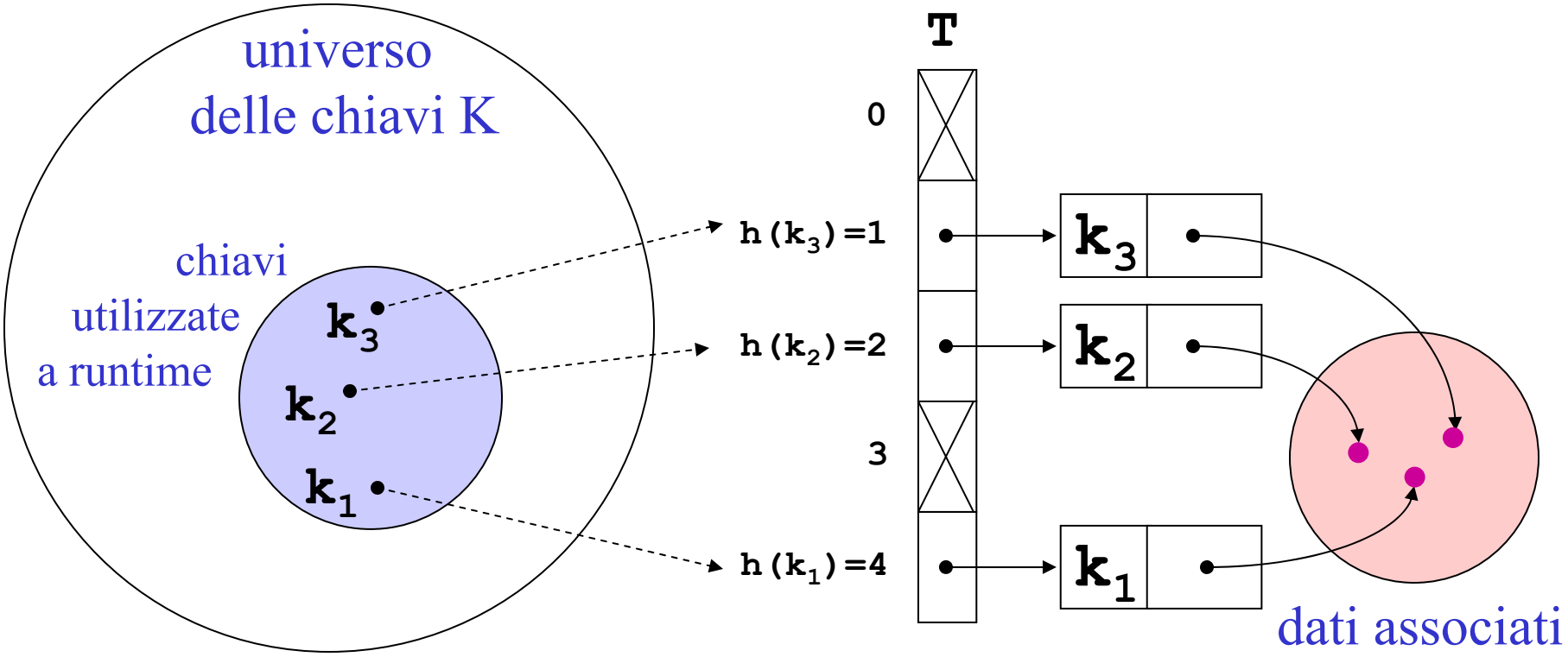


Realizzazione tramite tabelle hash

1. Utilizzo un array **`T`** per memorizzare i dati associati
   - la dimensione *m* dell'array **`T`**
     - è molto minore della dimensione dell'universo K
     - è molto vicina al numero delle chiavi effettivamente utilizzate dal programma in esecuzione
   - l'array **`T`**, come tutti gli array, può essere indicizzato solo da un intero
2. Definisco una funzione hash *h* che trasforma le chiavi di K negli interi nel range [0…*m*-1]

Tabella hash

- L'array **T** indicizzato tramite la funzione hash *h* è chiamato *tabella hash* (oppure *hashtable*)



Funzione hash

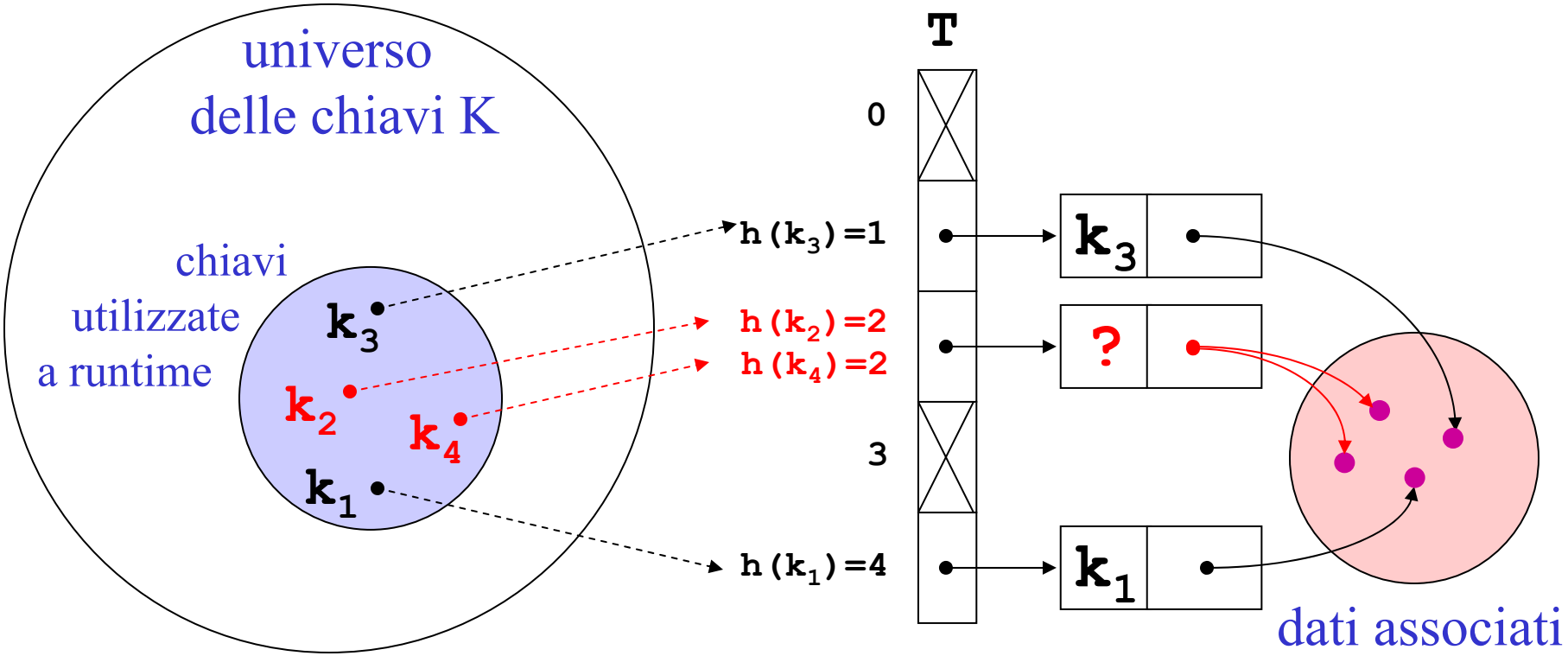
- La funzione hash *h*
  - definisce una corrispondenza tra l'universo K delle chiavi e gli indici della tabella hash **T** [0…*m*-1]

$$h: \mathrm{K} \rightarrow \{0, 1, \ldots, m\text{-}1\}$$

  - deve essere deterministica
    - altrimenti dopo aver messo i valori nell'array non riesco più a ritrovare la loro posizione
  - si richiede che sia calcolabile in tempo costante
    - per contenere i tempi di calcolo
- L'elemento con chiave $k \in \mathrm{K}$ si troverà nella posizione $h(k)$ nella tabella **T**

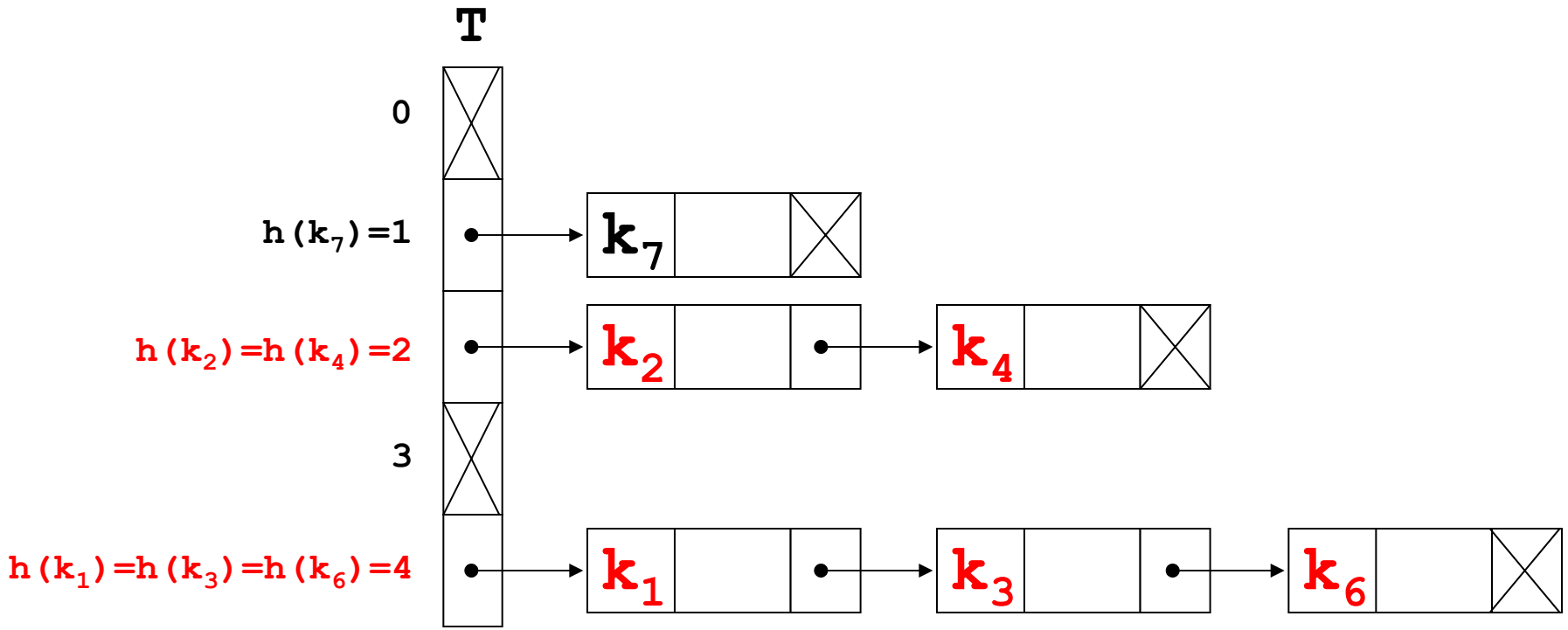
Il problema della collisione

- La funzione *h* ha un codominio (gli *m* indici di **T**) molto più piccolo del dominio (tutti gli elementi di K)
  - è inevitabile che si generino collisioni



Gestione delle collisioni con liste di trabocco

- Ogni posizione di **T** è un riferimento al primo elemento di una lista detta "di trabocco"



Liste di trabocco

- La lista di trabocco è una lista semplicemente concatenata
- Ogni nodo della lista è un oggetto con tre campi
  - `key`: valore della chiave
    - può essere un riferimento ad oggetto
  - `info`: valore associato alla chiave
    - può essere un riferimento ad oggetto
  - `next`: riferimento al prossimo nodo
    - è `NULL` per l'ultimo nodo della lista

Ricerca di un elemento in base alla chiave

- Per ricercare un elemento devo scorrere la lista di trabocco opportuna

```
GET(T,k) // ritorna il valore associato alla chiave (o NULL)
1.  i = HASH(k)     // devo guardare la lista i-esima
2.  x = T[i]        // iteratore per elementi della lista T[i]
3.  while x != NULL
4.      if EQUAL(k,x.key)
5.          return x.info     // l'ho trovato!
6.      x = x.next
7.  return NULL               // non l'ho trovato
```

Inserimento di una coppia ⟨chiave, valore⟩

- Anche l'inserimento prevede una ricerca

```
PUT(T,k,v) // inserisce <k,v> (eventualmente sovrascrive)
1.  i = HASH(k)     // devo cercare nella lista i-esima di T
2.  x = T[i]        // iteratore per elementi della lista T[i]
3.  while x != NULL
4.      if EQUAL(x.key,k)
5.          x.info = v    // sovrascrivo il vecchio valore
6.          return        // ho finito ed esco
7.      x = x.next
8.  y.key = k             // y nuovo elemento della lista
9.  y.info = v
10. y.next = T[i]
11. T[i] = y              // inserimento in testa
```

Cancellazione di un elemento

- L'elemento viene rimosso in base alla chiave

```
DELETE(T,k) // rimuove l'elemento (se esistente)
1.  i = HASH(k)     // devo cercare nella lista i-esima di T
2.  x = T[i]        // iteratore per elementi della lista T[i]
3.  prev = NULL     // punterà all'elemento che precede x
4.  while x != NULL
5.      if EQUAL(x.key,k)          // l'ho trovato
6.          if prev == NULL        // x è il primo della lista
7.              T[i] = x.next
8.          else                   // non è il primo della lista
9.              prev.next = x.next      // lo saltiamo
10.         return                 // ho finito ed esco
11.     prev = x      // non trovato, provo il prossimo
12.     x = x.next
```

Funzione EQUAL e funzione HASH

- Una precondizione perché si possa realizzare un array associativo con hashtable è che siano definite
  - una funzione EQUAL
  - una funzione HASH
- Entrambe le funzioni devono essere definite in base al contesto applicativo
- Per motivi di efficienza si richiede generalmene che entrambe le funzioni siano calcolabili in tempo costante
  - rispetto al numero *n* degli elementi nell'hashtable

Una funzione EQUAL errata

- Supponi che le chiavi siano stringhe
  - per esempio realizzate tramite array di caratteri
- La funzione EQUAL seguente è errata:

```
EQUAL_WRONG(A,B)     // A e B sono due array di caratteri
1.  return A == B
```

  - in questo modo non vengono confrontati i valori contenuti negli array A e B, ma i loro riferimenti
    - cioè gli indirizzi, che sono necessariamente diversi anche quando le due stringhe sono uguali

Una funzione EQUAL corretta

- La funzione seguente è una funzione corretta per questo contesto applicativo

```
EQUAL(A,B) // A e B sono due array di caratteri
1.  if A.length != B.length
2.      return FALSE          // lunghezza diversa
3.  for i=0 to A.length-1
4.      if A[i] != B[i]
5.          return FALSE      // almeno un carattere diverso
6.  return TRUE
```

  - questa volta vengono confrontati tutti i caratteri delle due stringhe
    - la complessità della procedura è ancora $\Theta(1)$ se le stringhe hanno una dimensione massima nota e indipendente da n

HASH: requisiti

- Requisiti funzionali
  - è deterministica
    - data una chiave k, dà sempre lo stesso risultato HASH(k)
- Requisiti prestazionali
  - è calcolabile in tempo costante
  - distribuisce le chiavi utilizzate in esecuzione in maniera pseudocasuale nell'intervallo [0…*m*-1]
    - questo requisito potrà solo essere soddisfatto solo in modo probabilistico
      - le chiavi che saranno utilizzate dall'utente in esecuzione non sono note a priori
      - comunque si scelga la funzione `HASH` esisterà sempre un insieme di chiavi che corrispondono alla stessa casella di T

HASH: ipotesi di distribuzione uniforme

- E' soddisfatta dalla funzione `HASH` quando, data una chiave $k \in$ K, la probabilità che `HASH`(k)=c sia la stessa per ogni casella $c \in [0 \ldots m\text{-}1]$ di `T`
  - indipendentemente da quali altre chiavi siano state già inserite in `T`
- Implicazioni dell'ipotesi di distribuzione uniforme
  - per chiavi simili vengono generati hash diversi
    - spesso le chiavi utilizzate sono molto simili
  - quali che siano le chiavi utilizzate, queste vengono con alta probabilità distribuite uniformemente nell'intervallo $[0 \ldots m\text{-}1]$

Fattore di carico

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

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

- α è il numero medio di elementi memorizzati in ogni lista concatenata
- A seconda del valore di α abbiamo
  - $\alpha < 1$ molte posizioni disponibili rispetto agli elementi memorizzati
  - $\alpha = 1$ il numero di elementi corrisponde al numero delle posizioni disponibili
  - $\alpha > 1$ molti elementi da memorizzare rispetto al numero delle posizioni disponibili

Complessità delle operazioni

- Caso migliore
  - l'operazione è eseguita su una lista di trabocco vuota o con un solo elemento
  - la complessità dell'operazione è data dalla complessità di `HASH` oppure di `HASH` + `EQUAL`
    - complessità $\Theta(1)$
- Caso peggiore
  - tutte le chiavi utilizzate corrispondono alla stessa posizione
  - la complessità coincide con quella che si ha per il calcolo di `HASH`(k) + la ricerca in una lista con n posizioni + il calcolo di `EQUAL` per n volte
    - complessità $\Theta(1) + \Theta(n) + \Theta(n) = \Theta(n)$

Complessità nel caso medio

- Caso medio
  - se la funzione `HASH` distribuisce le chiavi in modo uniforme nell'intervallo [0…*m*-1]
    - la lunghezza attesa delle liste di trabocco coincide con la lunghezza media α
    - le operazioni hanno complessità $\Theta(\alpha)$
    - se α non supera mai una soglia fissata α_{max} la complessità di ogni operazione è $\Theta(1)$
  - se la funzione `HASH` non dà garanzie rispetto alla distribuzione delle chiavi
    - la complessità è la stessa del caso peggiore

Note sulla complessità nel caso medio

- Abbiamo stabilito che nel caso medio la complessità delle operazioni sulle tabelle hash è $\Theta(\alpha)$
  - dove α è per definizione $\alpha = n/m$
- Dunque sembrerebbe che $\alpha \in \Theta(n)$
- Questo vorrebbe dire che, anche nel caso medio, la complessità delle operazioni è $\Theta(\alpha) = \Theta(n)$
  - cioè lineare come nel caso delle liste
  - non si avrebbe nessun vantaggio dall'adozione delle tabelle hash
- In che cosa sono errate le considerazioni qui sopra?

Note sulla complessità nel caso medio

- E' errato considerare $\alpha \in \Theta(n)$ a partire dalla definizione $\alpha = n/m$
- Anche m, infatti, cambia al crescere di n
  - quando il fattore di carico α supera una determinata soglia la dimensione m della tabella viene raddoppiata
    - il fattore di carico α viene dimezzato
  - il costo medio delle operazioni è ancora costante in virtù del fatto che il costo (lineare) del raddoppio della tabella viene assorbito dai precedenti inserimenti eseguiti in tempo costante
    - vedi slides sulla gestione telescopica delle pile e sulla complessità ammortizzata
- Una realizzazione delle tabelle hash che non preveda la gestione telescopica della tabella non può garantire tempi costanti di accesso

Funzioni hash

Funzioni hash

- Considereremo delle funzioni hash per le seguenti tipologie di chiavi
  - funzioni hash per interi
    - metodo della divisione
      - veloce ma raramente adottato
    - metodo della moltiplicazione
  - funzioni hash per stringhe
  - funzioni hash per oggetti arbitrari

Metodo della divisione: MOD_HASH

- Utilizza il resto di una divisione intera

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

- In pseudocodice:

```
MOD_HASH(k,m)          // k ed m sono interi
1.  return k mod m
```

- E' un metodo molto veloce
- Se le chiavi sono già degli interi pseudocasuali la funzione MOD_HASH viene utilizzata per riportare le chiavi nell'intervallo [0…*m*-1]

Metodo della divisione: MOD_HASH

- Se le chiavi non sono pseudocasuali
  - la praticità del metodo è compromessa
  - MOD_HASH ha delle forti proprietà di località
    - con altissima probabilità MOD_HASH(k+1) = MOD_HASH(k)+1
    - se m è una potenza di 2 o di 10, MOD_HASH(k) produce la parte meno significativa del numero k espresso in quella base
  - se si vuole usare questo metodo, è raccomandabile adottare come m un numero primo lontano da una potenza di due per limitare le collisioni
    - per esempio 701

Metodo della moltiplicazione: osservazione 1

- Supponiamo che le chiavi siano numeri reali pseudocasuali nell'intervallo (0,1)
- La funzione

$$h(k) = \lfloor m \cdot k \rfloor$$

è una buona funzione hash

  - $h(k)$ è deterministica
  - $h(k)$ può essere calcolata in $\Theta(1)$
  - $h(k)$ distribuisce uniformemente le chiavi nell'intervallo $[0 \ldots m\text{-}1]$
    - in quanto le chiavi erano già uniformemente distribuite!

Metodo della moltiplicazione: osservazione 2

- Sia *irr* un numero irrazionale
  - *irr* ha infinite cifre dopo la virgola, ma non è periodico
- Date delle chiavi intere qualsiasi, le cifre decimali dopo la virgola del prodotto $k \cdot irr$ si possono assumere uniformemente distribuite nell'intervallo (0,1)
  - questo valore coincide con $k \cdot irr - \lfloor k \cdot irr \rfloor$
  - Knuth propone

$$irr = \frac{\sqrt{5}-1}{2} = 0.6180339...$$

    - è la parte dopo la virgola della sezione aurea

$$\varphi = \frac{\sqrt{5}+1}{2} = 1.6180339...$$

    - è un numero irrazionale

Metodo della moltiplicazione

- Si utilizza come hash la parte intera di un prodotto

$$h(k) = \lfloor\, m \cdot (k \cdot irr - \lfloor k \cdot irr \rfloor) \,\rfloor$$

- Dove
  - *irr* è un numero irrazionale in (0,1)
    - per esempio la parte dopo la virgola della sezione aurea φ definita nella slide precedente
  - *m* può essere scelto arbitrariamente
    - di solito si usa una potenza di due: $m=2^p$, dove p è un intero

Metodo della moltiplicazione: MUL_HASH

- La costante *irr* viene calcolata una volta sola

```
1.  a.irr = (SQRT(5)-1)*0.5    // a.irr costante irrazionale
```

- La funzione MUL_HASH riceve in input l'array associativo a (per avere *m* ed *irr*) e la chiave *k*

```
MUL_HASH(a,k)                  // a array associativo, k intero
1.  m = a.T.length             // m è un numero intero
2.  prod = k * a.irr           // prod è un numero reale
3.  prod = m * (prod - INT(prod))     // reale in (0,m)
4.  out = INT(prod)                   // intero in [0,m-1]
5.  return out
```

  - la funzione INT tronca un reale all'intero inferiore

HASH per stringhe: SIMPLE_HASH

```
SIMPLE_HASH(S)        // S è una stringa (array di caratteri)
1.  hash = 0
2.  for i = 0 to S.length-1
3.      hash = hash + ASCII(S[i])
4.  return hash
```

- Ritorna un numero intero che poi deve essere ridotto nell'intervallo [0…*m*-1] con la funzione MOD_HASH
- Introdotta nella prima edizione del Kernigham-Ritchie
- Veloce ma generalmente considerata poco efficace nel distribuire i valori in modo pseudocausuale
  - permutazioni di caratteri hanno lo stesso hash!

HASH per stringhe: DJB2_HASH

```
DJB2_HASH(S)  // S è una stringa (array di caratteri)
1.  hash = 5381
2.  for i = 0 to S.length-1
3.      hash = hash*33 + ASCII(S[i])
4.  return hash
```

- Introdotta da Daniel J. Bernstein (donde il nome)
- Il numero 5381 è un numero primo
- Il "magic numer" 33 non è giustificato teoricamente
  - ma dà ottimi risultati nella pratica
  - corrisponde al prodotto * 32 (traslazione di cinque caselle della rappresentazione) + un incremento di uno
    - può essere realizzato velocemente

Funzioni hash per oggetti

- Supponiamo che
  - l'oggetto abbia come chiave h campi $c_1, c_2, \ldots, c_h$
  - siano già definite opportune funzioni hash $\texttt{HASH}_1(c_1), \texttt{HASH}_2(c_2), \ldots \texttt{HASH}_h(c_h)$
- Una funzione hash si può ottenere facilmente con la loro somma

$$\texttt{HASH}(o) = \texttt{HASH}_1(o.c_1)+\texttt{HASH}_2(o.c_2)+\ldots+\texttt{HASH}_h(o.c_h)$$

- Il risultato può essere riportato nell'intervallo opportuno tramite `MOD_HASH`

Insiemi

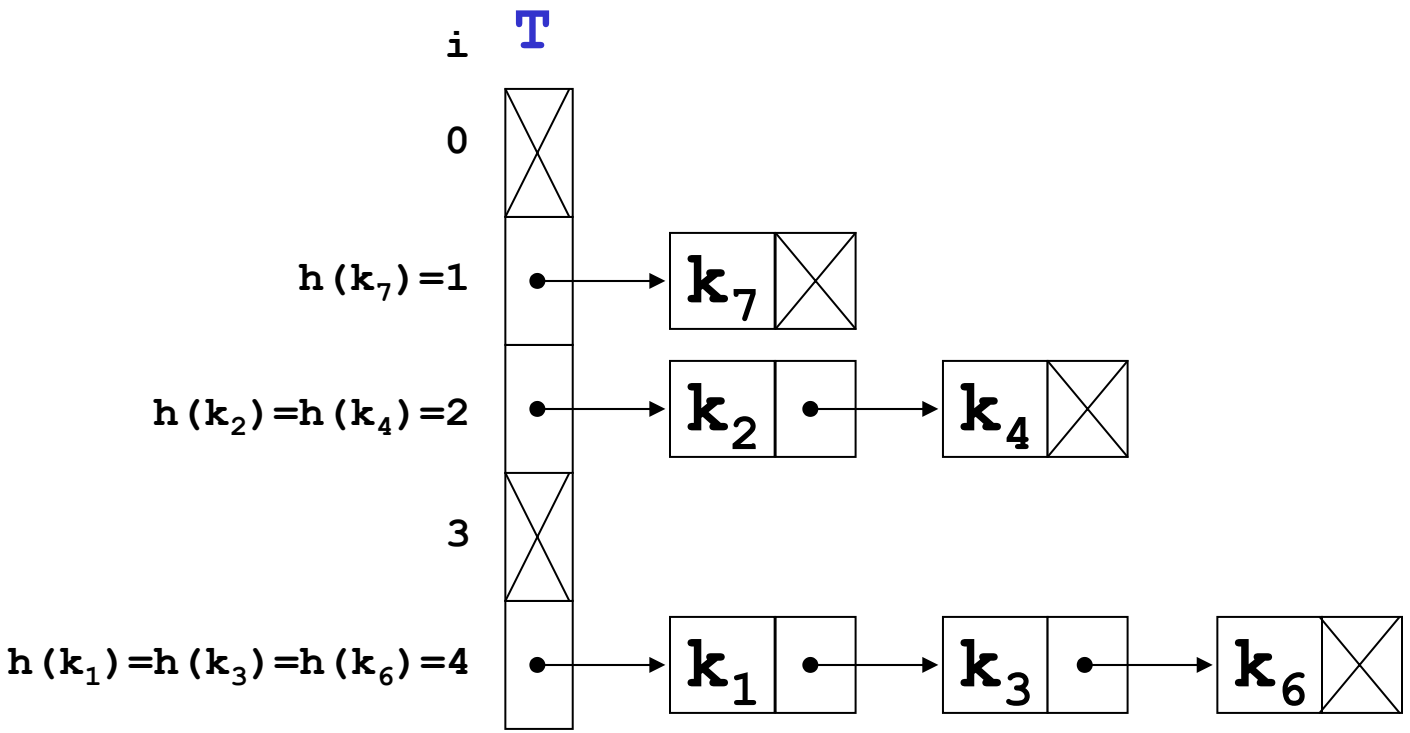
- Un insieme è una collezione di elementi omogenei
- Esempi di insieme
  - l'insieme degli studenti
  - l'insieme degli oggetti creati da un programma
  - l'insieme delle variabili utilizzate da un programma

Il tipo astratto di dato insieme

- Domini
  - il dominio di interesse è l'insieme I degli insiemi
  - dominio di supporto: gli elementi E dell'insieme
  - dominio di supporto: i booleani B = {true, false}
- Costanti
  - l'insieme vuoto
- Operazioni
  - aggiunge un elemento: $\text{ADD: } I \times E \rightarrow I$
  - elimina un elemento dall'insieme: $\text{REMOVE: } I \times E \rightarrow I$
  - verifica l'appartenenza: $\text{CONTAINS: } I \times E \rightarrow B$
  - …

Realizzazione di un insieme

- Si può usare una hashtable in cui lo stesso elemento funge da valore e da chiave



Problemi

1. Illustra l'inserimento in una tabella hash di dimensione m=10 gestita con liste di trabocco delle chiavi 32, 17, 19, 31, 33, 15, 38, 46, utilizzando la funzione hash `MOD-H`
2. Scrivi lo pseudocodice delle funzioni `ADD`(I,e), `REMOVE`(I,e) e `CONTAINS`(I,e) dove I è un insieme realizzato tramite una hashtable ed e è un elemento dell'insieme
   - assumi che siano definite opportune funzioni `HASH`(e) e `EQUAL`(e_1,e_2)

Problemi

1. Quali sono le prestazioni delle tabelle hash se al posto di una lista semplicemente concatenata usiamo come trabocco un albero rosso-nero?

