Tabelle Hash

ADT Dizionario

- In molte applicazioni è necessario che un insieme dinamico fornisca solamente le seguenti operazioni:
  - INSERT(key,value): Inserisce un elemento nuovo, con un certo valore (unico) di un campo chiave
  - SEARCH(key): Determina se un elemento con un certo valore della chiave esiste; se esiste, lo restituisce
  - DELETE(key): Elimina l'elemento identificato dal campo chiave, se esiste.
- Non è, ad esempio, necessario dover ordinare l'insieme dei dati
- o restituire l'elemento massimo, o il successore
- Queste strutture dati prendono talvolta il nome di *dizionari*

Implementazione Dizionario

- U - Universo di tutte le possibili chiavi
- K - Insieme delle chiavi effettivamente utilizzate
- Possibili implementazioni
  - U corrisponde all'intervallo [0..m-1], |K| ~ |U|
    1. tabelle ad indirizzamento diretto
  - U è un insieme generico, |K| << |U|
    2. tabelle Hash

Esempio: codici fiscali

- U - Universo di tutti i possibili codici fiscali
  - un codice fiscale è formato da 16 caratteri, dei quali 9 sono lettere e 7 sono numeri: RSSMRA78F12F893K
  - ci sono $25^9 \times 10^7$ possibili combinazioni differenti
- K - Insieme dei codici fiscali effettivamente utilizzati
  - circa 6×10^7

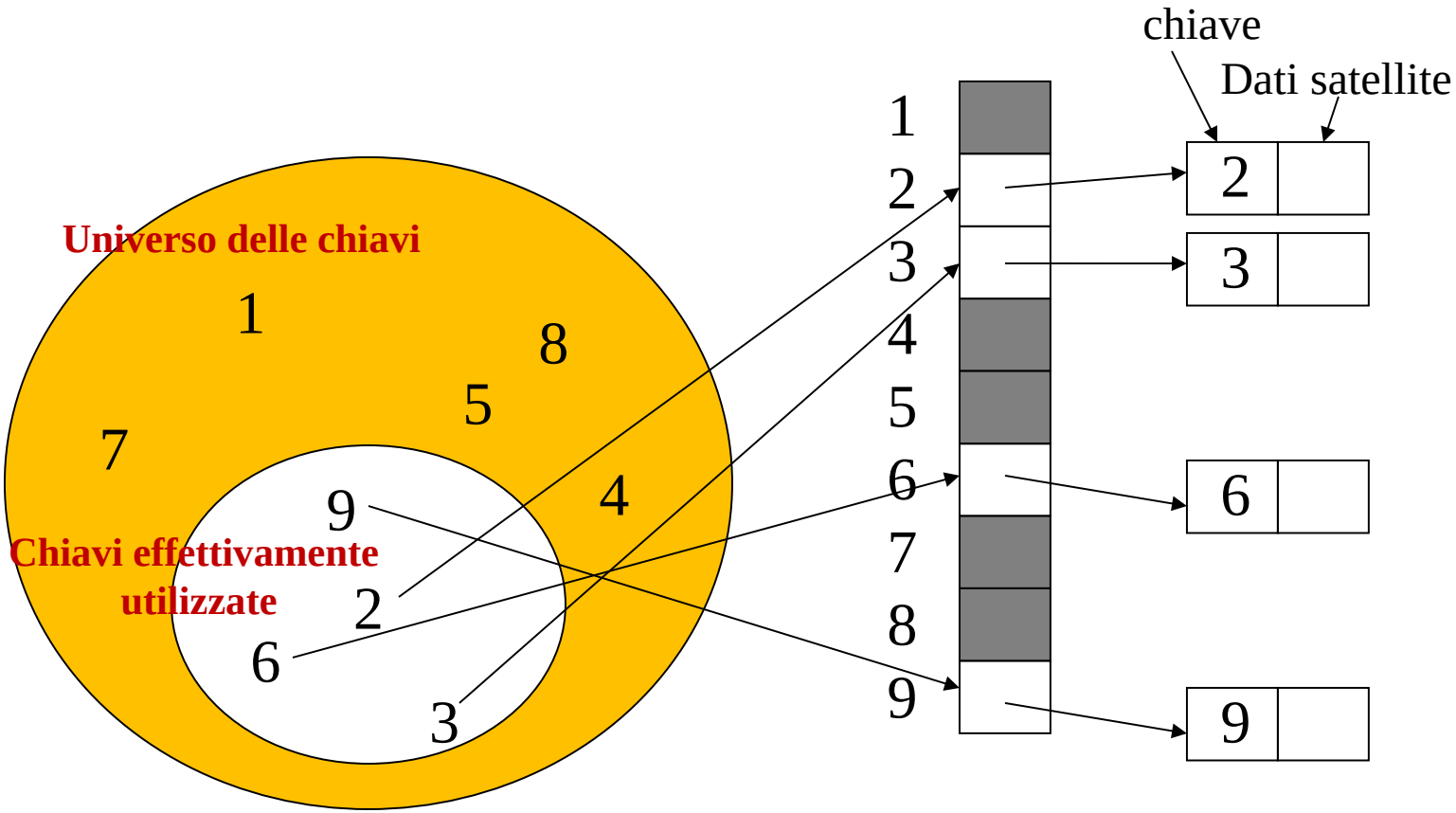
N.B.: $25^9 = 3.814.697.265.625$

Tabelle ad indirizzamento diretto

- Se l'universo delle chiavi è piccolo allora è sufficiente utilizzare una *tabella ad indirizzamento diretto*

- Una tabella ad indirizzamento diretto corrisponde al concetto di array:
  - ad ogni chiave possibile corrisponde una posizione, o slot, nella tabella
  - una tabella restituisce il dato memorizzato nello slot di posizione indicata tramite la chiave in tempo O(1)

Visualizzazione



Dizionario mediante tabella associativa

- T: tabella associativa, k: chiave, x: elemento
- Search(T, k)
  - Return T[k]
- Insert(T, x)
  - T[x->key] ← x
- Delete(T, x)
  - T[x->key] ← NIL

- Complessità O(1), occupazione O(|U|)

Memorizzazione

- E’ possibile memorizzare i dati satellite direttamente nella tabella oppure memorizzare solo puntatori agli oggetti veri e propri

- si deve distinguere l’assenza di un oggetto (oggetto NIL) dal caso particolare di un valore dell’oggetto stesso

Universo grande delle chiavi

- Se l'universo delle possibili chiavi è molto grande non è possibile o conveniente utilizzare il metodo delle tabelle ad indirizzamento diretto
  - Può non essere possibile a causa della limitatezza delle risorse di memoria
  - può non essere conveniente perché se il numero di chiavi effettivamente utilizzato è piccolo si hanno tabelle quasi vuote. Viene allocato spazio inutilizzato
- Le *tabelle hash* sono strutture dati che trattano il problema della ricerca e permettono di mediare i requisiti di memoria ed efficienza nelle operazioni

Compromesso

- Nel caso della ricerca si deve attuare un compromesso fra spazio e tempo:
  - spazio: se le risorse di memoria sono sufficienti si può impiegare la chiave come indice (accesso diretto)
  - tempo: se il tempo di elaborazione non rappresenta un problema si potrebbero memorizzare solo le chiavi effettive in una lista ed utilizzare un metodo di ricerca sequenziale

Compromesso

- Se |K| ~ |U|:
  - Non sprechiamo (troppo) spazio
  - Operazioni in tempo O(1) nel caso peggiore
- Se |K| << |U|: soluzione non praticabile
  - Esempio: studenti PSD con chiave “numero di matricola”
  - Se il numero di matricola ha 6 cifre, l'array deve avere spazio per contenere 10^6 elementi
  - Se gli studenti del corso sono ad esempio 30, lo spazio realmente occupato dalle chiavi memorizzate è $30/10^6 = 0.00003 = 0.003\%$!!!
- L’Hashing permette di impiegare una quantità ragionevole sia di memoria che di tempo operando un compromesso tra i casi precedenti

Importanza

- L’hashing è un problema classico dell’informatica
- gli algoritmi usati sono stati studiati intensivamente da un punto di vista teorico e sperimentale
- gli algoritmi di hashing sono largamente usati in un vasto insieme di applicazioni

- Es: nei compilatori dei linguaggi di programmazione si usano hash che hanno come chiavi le stringhe che corrispondono agli identificatori del linguaggio

Tabelle Hash

- Con il metodo di indirizzamento diretto un elemento con chiave k viene memorizzato nella tabella in posizione k
- Con il metodo hash un elemento con chiave k viene memorizzato nella tabella in posizione h(k)
- La funzione h() è detta *funzione hash*
- Lo scopo della funzione hash è di definire una corrispondenza tra l'universo U delle chiavi e le posizioni di una tabella hash T[0..m-1] con m << |U|

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

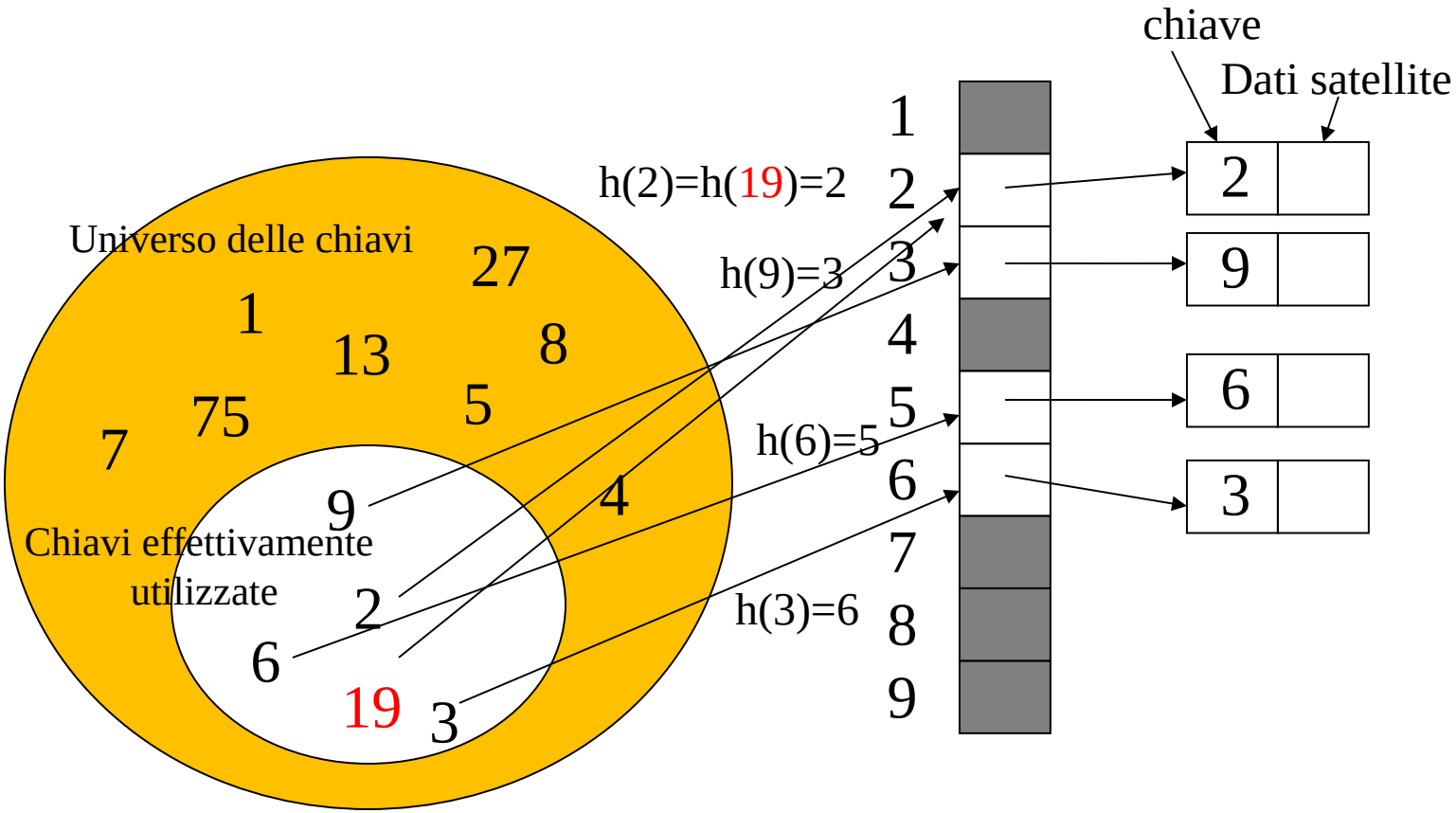
Tabelle Hash

- Un elemento con chiave k ha posizione pari al *valore hash di k denotato con* h(k)
- Tramite l'uso di funzioni hash il range di variabilità degli indici passa da |U| a m
- Utilizzando delle dimensioni *m* comparabili con il numero di dati effettivi da gestire si riduce la dimensione della struttura dati garantendo al contempo tempi di esecuzione di O(1)

Tabelle Hash

- Necessariamente la funzione hash non può essere iniettiva, ovvero due chiavi distinte possono produrre lo stesso valore hash
- Ogniqualvolta $h(k_i)=h(k_j)$ quando $k_i \neq k_j$, si verifica una collisione
- Occorre:
  - Minimizzare il numero di collisioni (ottimizzando la funzione di hash)
  - Gestire opportunamente le collisioni residue, in modo che più elementi associati alla stessa locazione trovino adeguatamente posto nella tabella

Visualizzazione



ADT *HashTable:* Specifica sintattica

- **Tipo di riferimento**: hashtable
- **Tipi usati**: item, key

Usiamo un tipo generico item contenente un attributo chiave di tipo Key

- **Operatori**
  - newHashTable() → hashtable
  - insertHash(item, hashtable) → hashtable
  - searchHash(key, hashtable) → item
  - deleteHash(key, hashtable) → hashtable

ADT *Lista:* Specifica semantica

- **Tipo di riferimento hashtable**
  - Una hashtable è un insieme di elementi *T={a1,a2, …,an}* di tipo ***item***
  - Un item contiene un campo chiave di tipo key e dei dati associati

ADT *Hashtable:* Specifica semantica

- **Operatori**
  - newHashtable() → t
    - Post: t = { }
  - insertHash(e, t) → t'
    - Pre: e deve avere key diversa da quelle presenti
    - Post: t = {a1, a2, … an}, t' = {a1, a2, …, e,..., an}
  - deleteHash(k,t) → t'
    - Pre: t = {a1, a2, …, an} n>0, a_i(key) = k con $1 \leq i \leq n$
    - Post: l' = <a1, a2, …, a_{i-1}, a_{i+1}, ..., an>
  - searchHash(k,t) → e
    - Pre: t = {a1, a2, …, an} n>0
    - Post: e = a_i con $1 \leq i \leq n$ se a_i(key) = k
      e = NULLITEM altrimenti

Funzioni Hash

- Quali sono le caratteristiche di una funzione hash?

- *Criterio di uniformità semplice*:
  - il valore hash di una chiave k è uno dei valori 0..m-1 in modo equiprobabile

- Formalmente: se si estrae in modo indipendente una chiave k dall'universo U con probabilità P(k) allora:

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

Funzioni Hash

- Tuttavia non sempre si conosce la distribuzione di probabilità delle chiavi P

- Esempio:
  - Se si ipotizza che le chiavi siano numeri reali distribuiti in modo indipendente ed uniforme nell'intervallo [0,1] allora la funzione

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

  - soddisfa il criterio di uniformità semplice

Funzioni Hash

- Un altro requisito è che una “buona” funzione hash dovrebbe utilizzare tutte le componenti della chiave per produrre un valore hash
- in questo modo valgono le ipotesi sulla distribuzione dei valori delle chiavi nella loro interezza, altrimenti dovremmo considerare la distribuzione solo della parte di chiave utilizzata

Convertire le chiavi in numeri naturali

- In genere le funzioni hash assumono che l'universo delle chiavi sia un sottoinsieme dei numeri naturali
- quando questo non è verificato si procede convertendo le chiavi in un numero naturale (anche se grande)
- Un metodo molto usato è quello di stabilire la conversione fra sequenze di simboli interpretati come numeri in sistemi di numerazione in base diversa

Conversione stringhe in naturali

- Per convertire una stringa in un numero naturale si considera la stringa come un numero in base 128
- esistono cioè 128 simboli diversi per ogni cifra di una stringa
- è possibile stabilire una conversione fra ogni simbolo e i numeri naturali (codifica ASCII ad esempio)
- la conversione viene poi fatta nel modo tradizionale
- Es: per convertire la stringa “pt” si ha:
  - ‘p’*128^1 + ‘t’*128^0= 112 * 128 + 116 *1= 14452

Metodo di divisione

- La funzione hash è del tipo:

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

- cioè il valore hash è il resto della divisione di k per m
- Caratteristiche:
  - il metodo è veloce
  - si deve fare attenzione ai valori di m

Metodo di divisione

- m deve essere diverso da 2^p per un qualche p
- altrimenti fare il modulo in base m corrisponderebbe a considerare solo i p bit meno significativi della chiave
- in questo caso dovremmo garantire che la distribuzione dei p bit meno significativi sia uniforme
- analoghe considerazioni per m pari a potenze del 10
- buoni valori sono numeri primi non troppo vicini a potenze del due

Chiavi molto grandi

- Spesso capita che le chiavi abbiano dimensione tale da non poter essere rappresentate come numeri interi per una data architettura
- Es: la chiave per la stringa: “averylongkey” è:

  $97*127^{11}+ 118*127^{10}+ 101*127^{9}+ 114*127^{8}+ 121*127^{7}+ 108*127^{6}+ 111*127^{5}+ 110*127^{4}+ 103*127^{3}+ 107*127^{2}+ 101*127^{1}+ 121*127^{0}$
- che è troppo grande per poter essere rappresentata
- un modo alternativo di procedere è di utilizzare una funzione hash modulare, trasformando un pezzo di chiave alla volta

Chiavi molto grandi

- Per fare questo basta sfruttare le proprietà aritmetiche dell'operazione modulo e usare l'algoritmo di Horner per scrivere la conversione
- si ha infatti che il numero precedente può essere scritto come:

  ((((((((((97*128 + 118)*128 + 101)*128 +
  114)*128+121)*128 + 108)*128 + 111)*128 + 110)*128 +
  103)*128 + 107)*128 + 101)*128 + 12)

Chiavi molto grandi

- Possiamo perciò calcolare il valore finale moltiplicando per 128 la prima cifra, aggiungendo la seconda cifra, moltiplicando nuovamente per 128 e così via.
- il procedimento iterativo arriverà a calcolare un intero non rappresentabile
- ma dato che siamo interessati al resto della divisione per m non è necessario arrivare a calcolare numeri molto grandi
- infatti basta eliminare in ogni momento i multipli di m

Chiavi molto grandi

- Tutte le volte che eseguiamo una moltiplicazione ed una somma dobbiamo solo ricordarci il resto modulo m

- Se ogni volta estraiamo il modulo otteniamo lo stesso risultato che avremmo se potessimo calcolare l’intero numero e dividessimo per m alla fine

Codice C per funzione hash per stringhe

```
int hash(char *v, int m){
  int h = 0, a = 128;
  for (; *v != '\0'; v++)
       h = (a*h + *v) % m;
  return h;
}
```

Metodo di Moltiplicazione

- Il metodo di moltiplicazione per definire le funzioni hash opera in due passi:
  - si moltiplica la chiave per una costante A in [0,1] e si estrae la parte frazionaria del risultato
  - si moltiplica questo valore per m e si prende la parte intera
- Analiticamente si ha:

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

- Come scegliere *A*?
  - Knuth suggerisce $A \approx (\sqrt{5} - 1)/2$.

Metodo di moltiplicazione

- Svantaggi: più lento del metodo di divisione
- Un vantaggio del metodo è che non ha valori critici di m
- Es:
  - A=(√5 -1)/2=0.61803…
  - sia k=123456 e m=10000
  - allora:

h(k)= ⌊10000(123456*0.61803.. mod 1)⌋

= ⌊10000(76300.0041151.. mod 1)⌋

= ⌊10000(0.0041151..)⌋

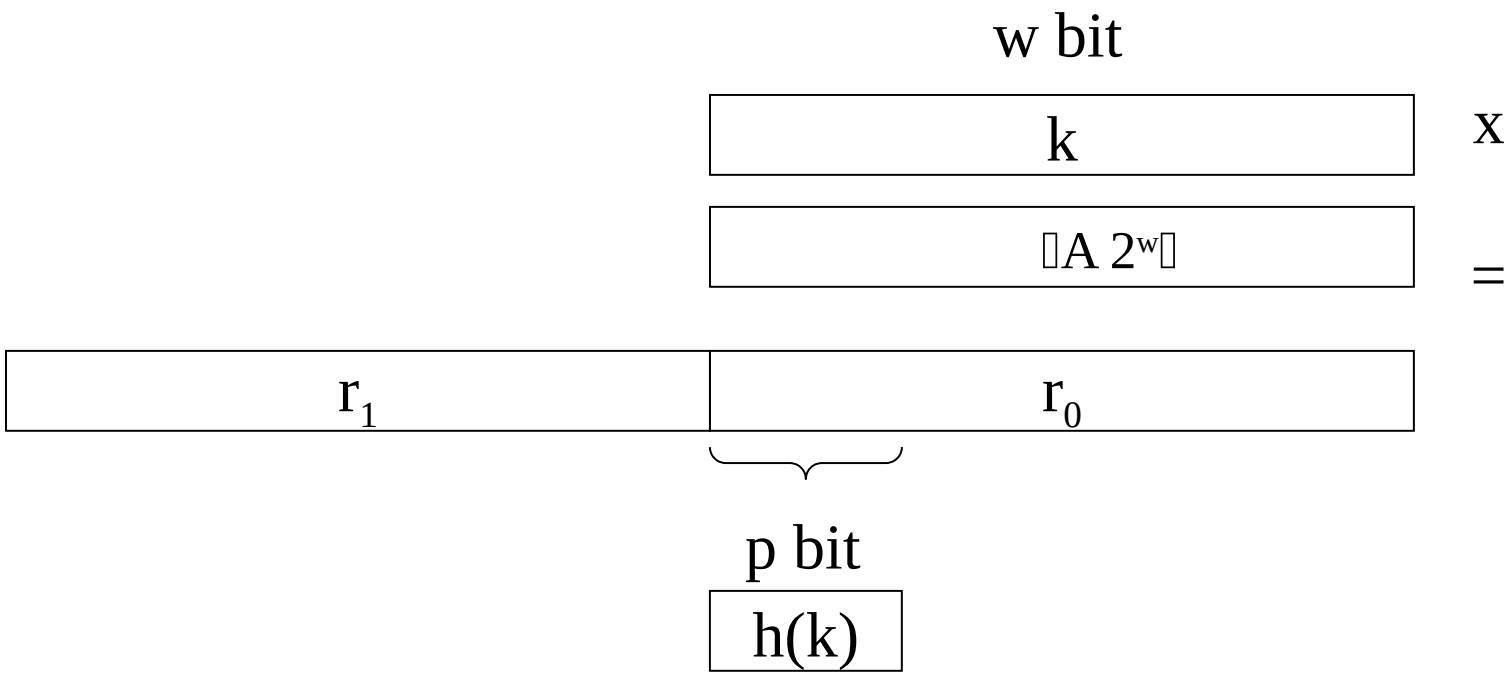
= ⌊41.151..⌋

=41

Metodo di moltiplicazione

- Spesso si sceglie $m=2^p$ per un qualche p in modo da semplificare i calcoli
- Una implementazione veloce di una h per moltiplicazione è il seguente:
  - supponiamo che la chiave k sia un numero codificabile entro w bit dove w è la dimesione di una parola del calcolatore
  - si consideri l'intero anch'esso di w bit $\lfloor A\,2^w \rfloor$
  - il prodotto $k * \lfloor A\,2^w \rfloor$ sarà un numero intero di al più 2w bit
  - consideriamo tale numero come $r_1 2^w + r_0$
  - r_1 è la parola più significativa del risultato e r_0 quella meno significativa
  - il valore hash di p bit consiste nei p bit più significativi di r_0

Visualizzazione



Metodo della Funzione Universale

- E’ possibile che la scelta di chiavi sia tale da avere un elevato numero di collisioni
- Il caso peggiore è che tutte le chiavi collidano in un unico valore hash
- Le prestazioni delle operazioni con tabelle hash dove la maggior parte delle chiavi hanno un unico valore hash peggiorano fino a □(n)
- Si può (come per il quicksort) utilizzare un algoritmo randomizzato di hashing in modo da garantire che non esista nessun insieme di chiavi che porti al caso peggiore

Metodo della Funzione Universale

- L’idea di base è di scegliere per ogni chiave una funzione hash casualmente da una famiglia di funzioni che rispettano specifiche proprietà
- in questo modo per qualsiasi insieme di chiavi si possono avere molte collisioni solo a causa del generatore pseudo casuale
- ma si può rendere piccola la probabilità di questo evento

Metodo della Funzione Universale

- Sia H un insieme finito di funzioni hash che vanno da U in {0,1,…,m-1}
- H è un insieme universale se per ogni coppia di chiavi distinte x,y ∈U, il numero di funzioni hash h∈H per cui h(x)=h(y) è esattamente |H|/m
- questa definizione equivale a dire che con una h scelta a caso la probabilità che h(x)=h(y) è 1/m

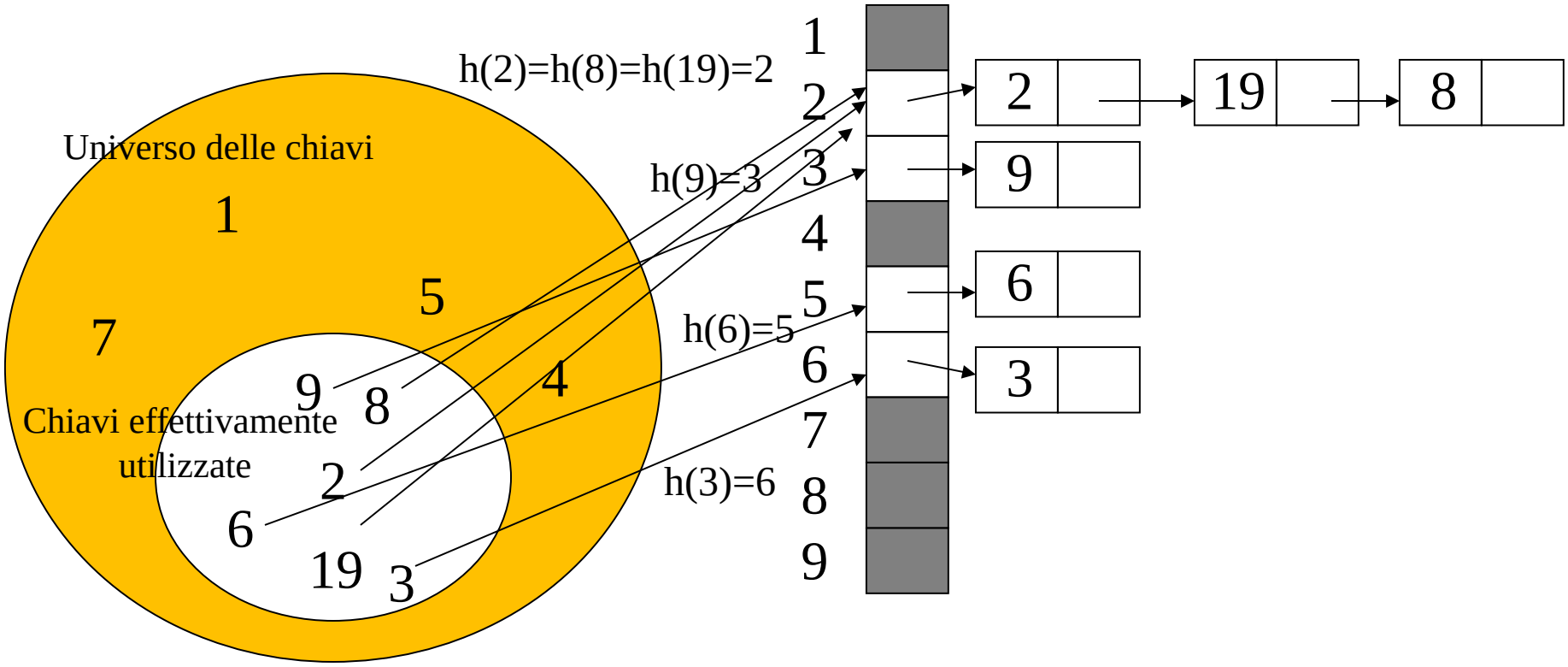
Metodi per la risoluzione delle collisioni

- Per risolvere il problema delle collisioni si impiegano principalmente due strategie:
  - metodo di concatenazione
  - metodo di indirizzamento aperto

Metodo di concatenazione

- L’idea è di mettere tutti gli elementi che collidono in una lista concatenata
- La tabella contiene in posizione j
  - un puntatore alla testa della j-esima lista
  - oppure un puntatore nullo se non ci sono elementi

Visualizzazione



Pseudo codice

- T[i] sono puntatori a liste, inizializzati a NULL.
- **`Chained-InsertHash(T,x)`**
  - inserisci x alla testa della lista T[h(x->key)]
- **`Chained-SearchHash(T,k)`**
  - cerca l'elemento con chiave k nella lista T[h(k)]
- **`Chained-DeleteHash(T,x)`**
  - cancella x dalla lista T[h(x->key)]

Hash: tipi di dato

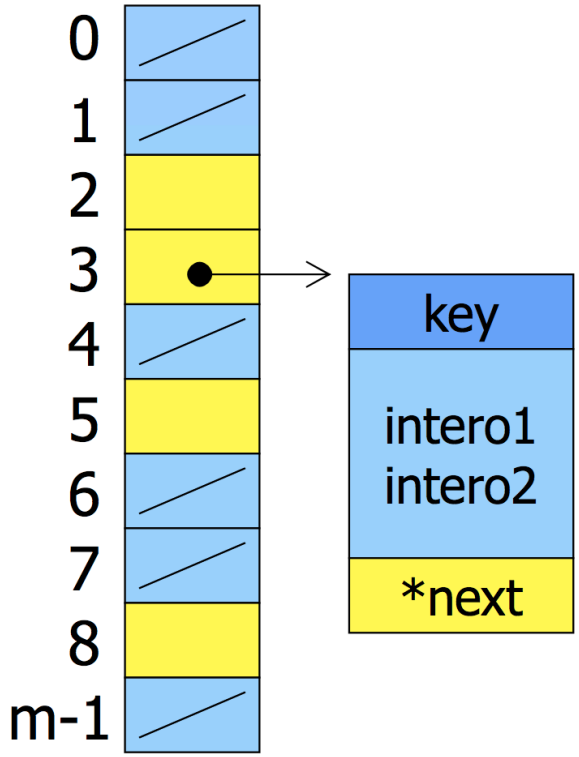
- Vettore di hash T

```
struct hash {
   int size;
   struct item **table;
};
typedef struct hash* hashtable;
```

E' un vettore (*) di puntatori (*) a struct elem.

- Elemento

```
struct item {
   char *key; /* chiave */
   int intero1;
   int intero2;
   struct item *next;
};
```



Hash: prototipi

- **`NEW(T,m)`**
  - hashtable newHashtable(int size);
- **`Chained-InsertHash(T,x)`**
  - int insertHash(hashtable h, item x);
- **`Chained-SearchHash(T,k)`**
  - struct item *hashSearch(hashtable h, char *key);
- **`Chained-DeleteHash(T,k)`**
  - struct item *hashDelete(hashtable h, char *key);
- **`DESTROY(T)`**
  - void DestroyHashtable(hashtable h);

Funzione hash

- Si assuma di disporre di un’opportuna funzione h(k) del tipo:

```
int hashFun(char *k, int m);
```

newHashtable

```
hashtable newHashtable(int size) {
  int i;
  hashtable h = (struct hash *) malloc (sizeof(struct hash));
  h->size = size;
  h->table = (struct item **) calloc (size,sizeof(struct item
*));
  //   for(i=0; i<size; i++) {
  //       h->table[i] = NULL;
  }
  return h;
}
```

InsertHash

```
int InsertHash(hashtable h, struct item *elem) {
   int idx;
   struct item *head, *curr;
   idx = hashFun(elem->key, h->size);
   curr = head = h->table[idx];
   while(curr) {
      if( strcmp(curr->key, elem->key)==0 )
         return (0);
      curr = curr->next;
   }
   h->table[idx] = newItem(elem->key, elem->intero1,
                           elem->intero2);
   h->table[idx]->next = head;
   return (1);
}
```

HashDelete

```
struct item *hashDelete(hashtable h, char *key) {
   int idx;
   struct item *prev, *curr, *head, *temp;
   idx = hashFun(key, h->size);
   prev = curr = head = h->table[idx];
   while(curr) {
         if( !strcmp(curr->key, key) ) {
               if(curr == head) h->table[idx] = curr->next;
               else prev->next = curr->next;
               return(curr);
         }
         prev = curr;
         curr = curr->next;
   }
       return NULL;
}
```

DestroyHash

```
void DestroyHashtable(hashtable h) {
    int i;
    for(i=0; i < h->size; i++) {
        deleteList(h->table[i]);
    }
    free(h->table);
    free(h);
    return;
}

static void deleteList(struct item *p) {
    if(p == NULL) return;
    deleteList(p->next);
    free(p);
    return;
}
```

Fattore di carico

- Data una tabella hash T con
  - **n** il numero di elementi memorizzati
  - **m** la dimensione della tabella di hash
- α =n/m si definisce il *fattore di carico* di T
  - α è il numero medio di elementi memorizzati in ogni lista concatenata
  - in genere vengono fissate delle soglie per α, secondo il metodo di gestione delle collisioni applicato
    - con il metodo del chaining, in genere si tiene α <=1

Tempo di calcolo

- Si fa l'ipotesi che il *tempo di calcolo di h* sia O(1) così che il tempo di calcolo delle varie operazioni dipende solo dalla lunghezza delle liste
- se le liste hanno lunghezza limitata da un valore costante
  - Il tempo di esecuzione per *l'inserimento* è O(1)
  - Il tempo di esecuzione per la *cancellazione* è O(1)

Tempo di calcolo per la ricerca e cancellazione

- Il *caso peggiore* si ha quando tutte le chiavi collidono e la tabella consiste in una unica lista di lunghezza n in questo caso il tempo di calcolo è O(n)

- il *caso medio* dipende da quanto in media la funzione hash distribuisca l'insieme delle chiavi sulle m posizioni

- se la funzione hash soddisfa l'ipotesi di *uniformità semplice* allora ogni lista ha lunghezza media

Tempo di calcolo per la ricerca con insuccesso

- Il tempo medio è il tempo impiegato per generare il valore hash data la chiave e il tempo necessario per scandire una sequenza fino alla fine
- dato che la lunghezza media di una sequenza è si ha O(1+)

Metodo di indirizzamento aperto

- L’idea è di memorizzare tutti gli elementi nella tabella stessa
- in caso di collisione si memorizza l’elemento nella posizione *successiva*
- per l’operazione di ricerca si esaminano tutte le posizioni ammesse per la data chiave in sequenza

- non vi sono liste né elementi memorizzati fuori dalla tabella
- il fattore di carico è sempre 1
  - in genere si tiene sempre <= 0,5

Metodo di indirizzamento aperto

- Per eseguire l’inserzione si genera un valore hash data la chiave e si esamina una successione di posizioni della tabella (*scansione*) a partire dal valore hash fino a trovare una posizione vuota dove inserire l’elemento

Sequenza di scansione

- La sequenza di scansione dipende dalla chiave che deve essere inserita
- per fare questo si estende la funzione hash perché generi non solo un valore hash ma una sequenza di scansione

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

- cioè prenda in ingresso una chiave e un indice di posizione e generi una nuova posizione

Sequenza di scansione

- data una chiave k, si parte dalla posizione 0 e si ottiene h(k,0)
- la seconda posizione da scansionare sarà h(k,1)
- e così via h(k,i)
- ottenendo una sequenza <h(k,0),h(k,1),…,h(k,m-1)>

Pseudocodice per l'inserimento

```
Hash-Insert(T,k)
1  i ← 0
2  repeat j ← h(k,i)
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”
```

Pseudocodice per la ricerca

```
Hash-Search(T,k)
1  i ← 0
2  repeat j ← h(k,i)
3          if T[j]=k
4          then return j
6          i ← i+1
7  until i=m o T[j]=NIL
8  return NIL
```

La cancellazione

- L’operazione di cancellazione è difficile
- non si può marcare la posizione come vuota con NIL perché questo impedirebbe la ricerca degli elementi successivi nella sequenza
- si potrebbe usare uno speciale marcatore DELETED invece di NIL
- la procedura di inserzione dovrebbe poi scrivere in questi elementi
- in questo modo però i tempi non dipendono più solo dal fattore di carico
- In genere se si prevedono molte cancellazioni si utilizzano i metodi con concatenazione

Caratteristiche di h

- Quali sono le caratteristiche di una buona funzione hash per il metodo di indirizzamento aperto?
- Si estende il concetto di uniformità semplice
- la h deve soddisfare la *proprietà di uniformità della funzione hash*:
  - per ogni chiave k la sequenza di scansione generata da h deve essere una qualunque delle m! permutazioni di {0,1, …,m-1}

Funzioni Hash per indirizzamento aperto

- è molto difficile scrivere funzioni h che rispettino la proprietà di uniformità
- si usano generalmente tre approssimazioni:
  - scansione lineare
  - scansione quadratica
  - hashing doppio
- tutte queste classi di funzioni garantiscono di generare una permutazione ma nessuna riesce a generare tutte le m! permutazioni

Scansione Lineare

- Data una funzione hash h':U {0,1,…,m-1} il metodo di scansione lineare costruisce una h(k,i) nel modo seguente:

$$h(k,i)=(h'(k)+i) \bmod m$$

- data la chiave k si genera la posizione h'(k), quindi la posizione h'(k)+1, e così via fino alla posizione m-1.
- Poi si scandisce in modo circolare la posizione 0,1,2
- fino a tornare a h'(k)-1

Agglomerazione primaria

- La scansione lineare è facile da realizzare ma presenta il fenomeno di agglomerazione (*clustering*) primaria:
  - si formano lunghi tratti di posizioni occupate aumentando i tempi di ricerca
- Si pensi ad esempio il caso in cui vi siano n=m/2 chiavi nella tabella
  - se le chiavi sono disposte in modo da alternare una posizione occupata con una vuota, allora la ricerca senza successo richiede 1,5 accessi
  - se le chiavi sono disposte tutte nelle prime m/2 posizioni allora si devono effettuare n/4 accessi in media per la ricerca senza successo

Scansione Quadratica

- Data una funzione hash h':U {0,1,…,m-1} il metodo di scansione quadratica costruisce una h(k,i) nel modo seguente:

$$h(k,i)=(h'(k) + c_1 i + c_2 i^2) \bmod m$$

- dove c_1 e c_2 e m sono costanti vincolate per poter far usa dell'intera tabella
- questo metodo funziona meglio della scansione lineare perché evita l'agglomerazione primaria ma non risolve il problema della agglomerazione (che in questo caso viene detta secondaria) in quanto se si ha una collisione sul primo elemento le due sequenze di scansione sono comunque identiche

Hashing doppio

- L’hashing doppio risolve il problema delle agglomerazioni ed approssima in modo migliore la proprietà di uniformità
- Date due funzione hash

$$h_1 : U \rightarrow \{0,1,\ldots,m-1\}$$

$$h_2 : U \rightarrow \{0,1,\ldots,m'-1\}$$

- il metodo di scansione quadratica costruisce una h(k,i) nel modo seguente:

$$h(k,i) = (h_1(k) + i\, h_2(k)) \bmod m$$

Hashing doppio

- L’idea è di partire da un valore hash e di esaminare le posizioni successive saltando di una quantità pari a multipli di un valore determinato da una altra funzione hash
- in questo modo in prima posizione esaminata è $h_1(k)$ mentre le successive sono distanziate di $h_2(k)$ dalla prima

Esempio

- Si consideri
  - m=13
  - $h_1(k)$= k mod 13
  - $h_2(k)$=1+(k mod 11)
- si consideri l'inserimento della chiave 14 nella seguente tabella:

0	1	2	3	4	5	6	7	8	9	10	11	12
	79			69	98		72				50	

Esempio

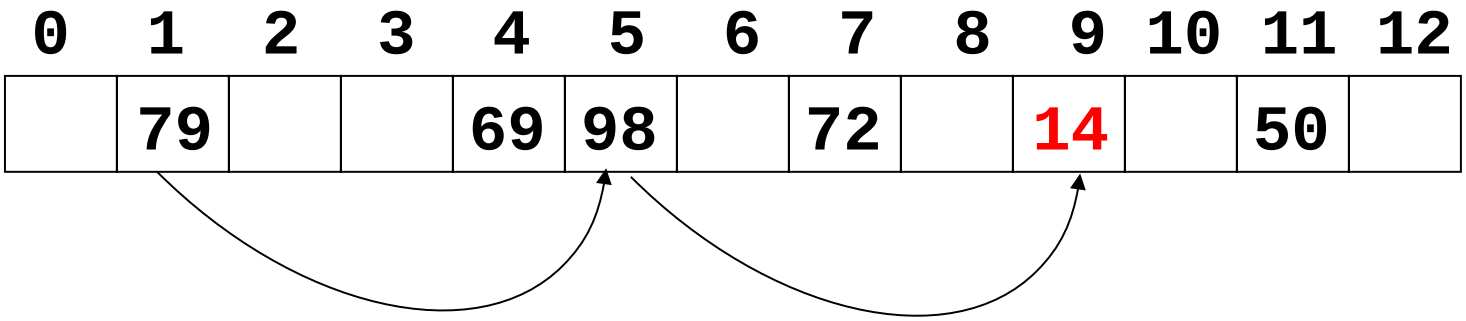
$h_1(14)= 14 \bmod 13= 1$

$h_2(14)=1+(14 \bmod 11) = 1 +3 = 4$

1 posizione esaminata 1

2 posizione esaminata 1 + 4 = 5

3 posizione esaminata 1 + 4*2 = 9



Considerazioni su h1 e h2

- Si deve fare attenzione a far si che $h_2(k)$ sia primo rispetto alla dimensione della tabella m
- infatti, se m e $h_2(k)$ hanno un massimo comun divisore d, allora la sequenza cercata per k esaminerebbe solo m/d elementi
- Esempio: m=10 e $h_2(k)=2$, partendo da 0 si ha la sequenza 0 2 4 6 8 0 2 4 6 8….

Rapporto fra h1 e h2

- Per garantire che $h_2(k)$ sia primo rispetto m si può:
  - prendere $m=2^p$ e scegliere $h_2(k)$ in modo che produca sempre un numero dispari
  - un altro modo è di prendere m primo e scegliere $h_2(k)$ in modo che produca sempre un intero positivo minore di m
  - Ex:

$$h_1(k)=k \bmod m$$

$$h_2(k)=1+(k \bmod m')$$

  - dove m' è di poco minore di m; m'=m-1 o m'=m-2

Considerazioni sull'hashing doppio

- L'hashing doppio approssima in modo migliore la proprietà di uniformità rispetto alla scansione lineare o quadratica
- infatti:
  - la scansione lineare (quadratica) genera solo O(m) sequenze; una per ogni chiave
  - l'hashing doppio genera $O(m^2)$ sequenze; una per ogni coppia (chiave, posizione)
- la posizione iniziale $h_1(k)$ e la distanza $h_2(k)$ possono variare indipendentemente l'una dall'altra

Implementazione C

```
void insert(item item){
    Key v = item->key;
     int i = hash(v, M), k = hashtwo(v, M);
     while (st[i]!=NULL)
          i = (i+k) % M;
     st[i] = item;
}

item search(Key v){
    int i = hash(v, M), k = hashtwo(v, M);
     while (st[i]!=NULL){
       if (v == st[i]->key) return st[i];
       else i = (i+k) % M;
     }
     return NULLITEM;
}
```

Analisi del tempo di calcolo

- Si suppone di lavorare con funzioni hash uniformi
- in questo schema ideale, la sequenza di scansioni <h(k,0),h(k,1),…,h(k,m-1)> è in modo equiprobabile una qualsiasi delle m! permutazioni di <0,1,…,m-1>
- in una ricerca senza successo si accede ogni volta ad una posizione occupata che non contiene la chiave e l'ultimo accesso è poi fatto ad una posizione vuota

Analisi del tempo di calcolo

- Un accesso viene sempre fatto
- un secondo accesso viene fatto con probabilità α
- un terzo accesso con probabilità circa α^2
- e così via
- in media si dimostra che il numero medio di accessi per ricerca senza successo è $1+ \sum_{i=1..\infty} \alpha^i = 1/(1- \alpha)$

Analisi del tempo di calcolo

- Data una tabella hash ad indirizzamento aperto con fattore di carico α, il numero medio di accessi per una ricerca con successo è al più

$$1/\alpha \ln 1/(1-\alpha)$$

- nell'ipotesi di funzione hash uniforme con chiavi equiprobabili

Considerazioni finali

- E’ complicato confrontare i metodi di hashing con concatenazione e ad indirizzamento aperto
- infatti il fattore di carico deve tenere conto del fatto che nel caso di indirizzamento aperto si memorizza direttamente la chiave mentre nel caso di concatenazione si ha la memorizzazione aggiuntiva dei puntatori per gestire le liste
- in generale si preferisce ricorrere al metodo delle concatenazioni quando non è noto il fattore di carico, mentre si ricorre all’indirizzamento aperto quando esiste la possibilità di predirre la dimensione dell’insieme delle chiavi

Alberi binari di ricerca e hash

- La scelta di utilizzare strutture dati di tipo alberi binari di ricerca o hash prende in considerazione i seguenti fattori
- vantaggi per hashing:
  - è di facile e veloce implementazione
  - tempi di ricerca rapidissimi
- vantaggi per alberi binari di ricerca:
  - minori requisiti di memoria
  - dinamici
  - buone prestazioni anche nel caso peggiore
  - supportano un numero maggiore di operazioni (ordinamento)