

Algoritmi e Strutture Dati

Schifano S. Fabio `schifano@fe.infn.it`

I Dizionari

I **dizionari** sono un caso particolare di insieme in cui sono possibili le seguenti operazioni:

- ▷ **verificare** l'appartenenza di un elemento
- ▷ **cancellare** un elemento
- ▷ **inserire** un elemento

Gli elementi del dizionario sono detti **chiavi**.

La specifica della struttura dati **dizionario** é identica a quella dell'insieme in cui le uniche operazioni ammesse sono:

- ▷ **CREADIZIONARIO**: crea un dizionario vuoto
- ▷ **APPARTIENE**: verifica l'appartenenza di una chiave al dizionario
- ▷ **CANCELLA**: elimina una chiave dal dizionario
- ▷ **INSERISCI**: inserisce una nuova chiave nel dizionario

Ogni realizzazione adottata per gli insiemi può essere usata per realizzare un dizionario, ma esistono realizzazione più efficienti tipo **vettori ordinati** e **tabelle hash**.

I Dizionari: Realizzazione tramite vettore ordinato

Supponiamo che tra gli elementi dell'insieme delle chiavi sia definita una relazione di ordinamento totale \leq .

La realizzazione con vettore ordinato utilizza:

- ▷ un vettore per memorizzare, in posizioni **contigue** e in ordine **crescente**, le chiavi del dizionario
- ▷ un cursore all'ultima posizione occupata per sapere quanto é lungo il vettore

Con tale realizzazione le operazioni sul dizionario possono essere implementate nel seguente modo:

- ▷ **CREADIZIONARIO**: crea una struttura con due componenti, un vettore di chiavi ed un cursore
- ▷ **INSERISCI**: scandisce il vettore e colloca la chiave in una posizione opportuna in modo da mantenere il vettore ordinato
- ▷ **CANCELLA**: scandisce il vettore e quando trova la chiave la elimina
- ▷ **APPARTIENE**: l'appartenenza di una chiave k al dizionario può essere realizzata tramite una ricerca **binaria** (o **dicotomica** o **logaritimica**) secondo il seguente algoritmo:
 1. si accede all'elemento di mezzo m
 2. se $m = k$ allora abbiamo trovato la chiave
 3. altrimenti, se $k < m$ si ripete la ricerca sulla metà sinistra altrimenti sulla metà destra.

I Dizionari: Realizzazione tramite vettore ordinato

La realizzazione C della struttura dizionario é la seguente:

```
typedef int tipoelem; // chiavi di tipo intero

struct dizionario_struct {

    tipoelem chiavi[MAXLUNG];
    int      ultimo;

};

typedef struct dizionario_struct * dizionario;
```

Utilizzando la suddetta realizzazione per il tipo di dato dizionario, l'operazione di **APPARTIENE** é implementata nel seguente modo:

- ▶ la funzione **APPARTIENE** prende in input una chiave k ed un dizionario D e ritorna **True** o **False** a seconda che la chiave esista o meno nel dizionario
- ▶ per effettuare ricerca la funzione **APPARTIENE** chiama un'altra funzione ricorsiva **RICBIN** che effettua la ricerca binaria sul vettore

```
int RICBIN ( int[] V, int k, int i, int j ) {  
    int m;  
  
    if ( i > j )  
        return 0;           // return False  
    else {  
        m = (i+j) / 2;       // calcolo il punto di mezzo  
        if ( k == V[m] )  
            return 1;        // return True  
        else  
            // richiamo ricorsivamente RICBIN  
            if ( k < V[m] )  
                RICBIN ( V, k, i, m-1 );  
            else  
                RICBIN ( V, k, m+1, j );  
    }  
}
```

Qual'è la complessità computazionale della funzione **RICBIN** ?

Indichiamo con n il numero degli elementi del vettore e $T(n)$ la relazione che esprime il tempo di calcolo in funzione della dimensione del vettore su cui la funzione effettua la ricerca.

- ▷ per $n = 1$ il tempo di calcolo è costante, basta verificare se l'elemento è quello cercato
- ▷ per $n > 1$ il tempo di calcolo è, nel caso pessimo, una costante più il tempo di calcolo per cercare un elemento in vettore di dimensione $n/2$

$$T(n) = \begin{cases} \alpha & \text{se } n = 0 \\ T(n/2) + \beta & \text{se } n > 0 \end{cases}$$

Supponiamo per semplicità che $n = 2^h$ per qualche $h \geq 0$, sviluppando manualmente la relazione $T(n)$ si ottiene:

$$\begin{aligned} T(n) &= \\ &= T(n/2) + \beta = T(n/2^1) + \beta \\ &= T(n/4) + \beta + \beta = T(n/2^2) + 2\beta \\ &= T(n/8) + \beta + \beta + \beta = T(n/2^3) + 3\beta \\ &\vdots \text{ dopo } h \text{ passi} \\ &= T(n/(2^h)) + h\beta \\ &= T(1) + \log_2 n \cdot \beta \\ &= \alpha + \log_2 n \cdot \beta \end{aligned}$$

Quindi, $T(n) \in \mathcal{O}(\log n)$, ovvero algoritmi basati su uno schema di ricerca dicotomica hanno una complessità logaritmica.

La funzione **APPARTIENE** può quindi essere definita come segue:

```
int APPARTIENE ( int k, dizionario D ) {  
    return RICBIN ( D->chiavi, k, 1, D->ultimo );  
}
```

Poiché l'operazione di **APPARTIENE** si basa su uno schema di ricerca binaria, la complessità computazionale é $\mathcal{O}(\log n)$ ove n sono il numero delle chiavi contenute nel vettore.

Ricerca di Interpolazione

La complessità dell'operazione di **APPARTIENE** può essere ulteriormente abbassata adottando un metodo di ricerca detto di **interpolazione**.

Supponiamo che le n chiavi siano numeriche ed **uniformemente distribuite** nell'intervallo $[k_0 \dots k_1]$ ove k_0 è il più piccolo valore delle chiavi e k_1 è il più grande valore.

Volendo cercare la chiave k anziché provare a cercarla nella posizione mediana potremmo provare a cercarla laddove è più probabile trovarla, cioè vicino alla chiave di posizione $\alpha \cdot n$, ove $\alpha = \frac{k - k_0}{k_1 - k_0}$.

- ▷ $k - k_0$ ci dice di quanto la chiave k si discosta dall'estremo sinistro dell'intervallo delle chiavi
- ▷ $k_1 - k_0$ ci dice quanto è grande l'intervallo delle chiavi
- ▷ il rapporto $\alpha \in [0 \dots 1]$ ci dice di quanto si discosta, in percentuale, il valore k dai due estremi:
 - ▷ se $\alpha = 0$ allora $k = k_0$, quindi conviene cercare la chiave in prima posizione
 - ▷ se $\alpha = 1$ allora $k = k_1$, quindi conviene cercare la chiave in ultima posizione
 - ▷ se $\alpha = 0.5$ allora $k = \frac{k_1 + k_0}{2}$, quindi conviene cercare la chiave in posizione centrale
 - ▷ in generale quindi, sotto l'ipotesi che le chiavi siano uniformemente distribuite, conviene cercare la chiave in posizione $\alpha * n$

Il metodo di ricerca derivante è detto di **interpolazione**.

Ricerca di Interpolazione: Realizzazione

La realizzazione C della funzione **RICBIN** può essere trasformata in una ricerca di **interpolazione** sostituendo il calcolo di

$$m = (i + j) / 2 = i + ((j - i)/2)$$

con l'istruzione, ove l'ampiezza dell'intervallo delle chiavi è $j - i$:

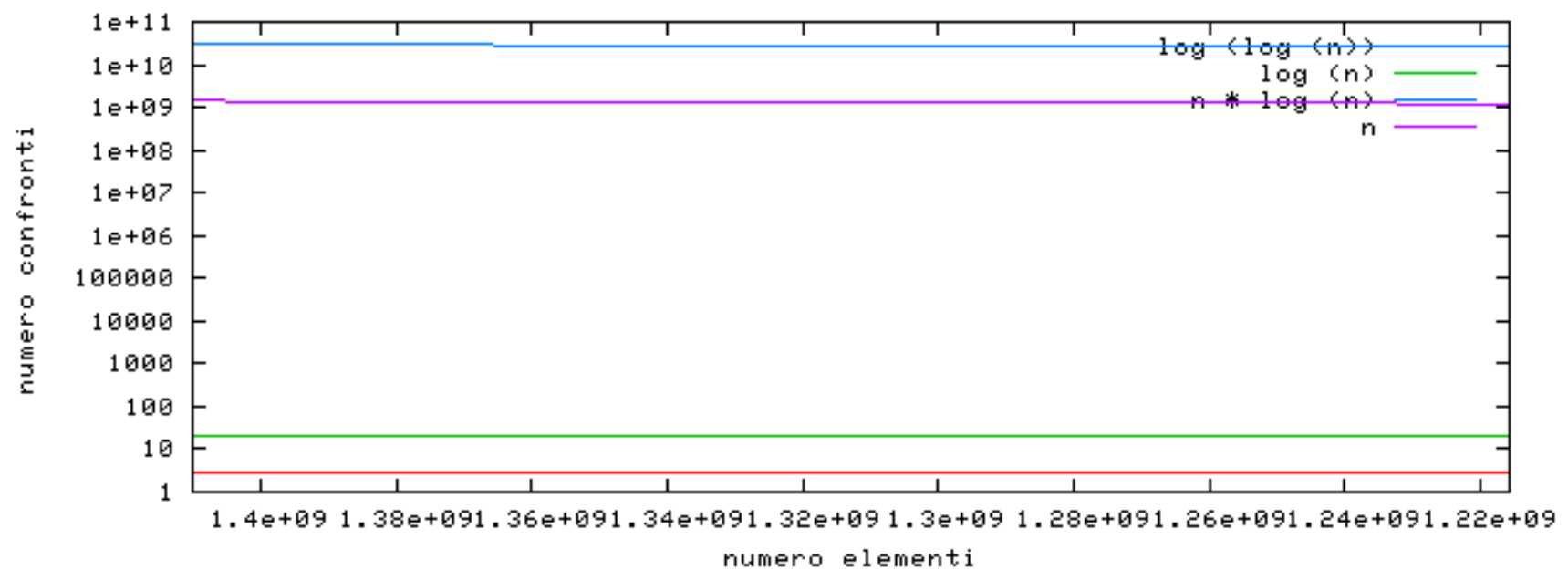
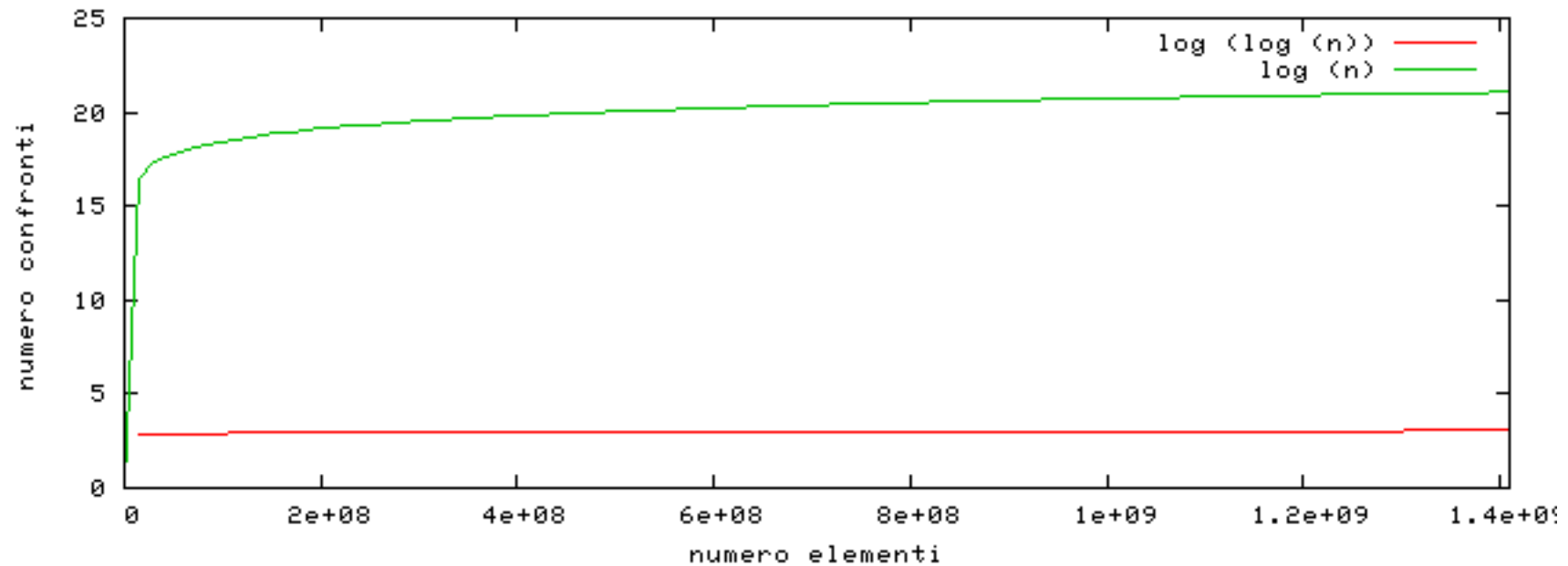
$$m = i + ((j - i) * (k - V[i])) / (V[j] - V[i])$$

Infatti, in generale, la ricerca viene effettuata sull'intervallo $[i \dots j]$ e $i > 0$.

Quindi, quando $\alpha = 0$ devo ottenere $k_0 = i$ mentre quando $\alpha = 1$ devo ottenere $k_1 = j$.

É possibile dimostrare che sotto l'ipotesi di **distribuzione uniforme** delle chiavi, la complessità dell'operazione **APPARTIENE** con ricerca di **interpolazione** é $\mathcal{O}(\log \log n)$.

La funzione $\log \log n$ é paragonabile a $\log n$ per valori di n piccoli, ma mantiene valori molto bassi anche per valori di n molto grandi.



Considerazioni

Adottando una realizzazione dei dizionari tramite vettori ordinati:

- ▶ la complessità dell'operazione **APPARTIENE** con metodo di ricerca binaria é $\mathcal{O}(\log n)$
- ▶ la complessità dell'operazione **APPARTIENE** con metodo di ricerca di interpolazione é $\mathcal{O}(\log \log n)$
- ▶ la complessità delle operazioni di **INSERISCI** é $\mathcal{O}(n)$
- ▶ la complessità delle operazioni di **CANCELLA** é $\mathcal{O}(n)$.

La complessità delle operazioni di **INSERISCI** e **CANCELLA** sono $\mathcal{O}(n)$, poiché devono preoccuparsi di mantenere gli elementi in posizioni contigue del vettore. Solo in questo modo é possibile realizzare una operazione di **APPARTIENE** in tempo $\mathcal{O}(\log n)$ o $\mathcal{O}(\log \log n)$.

Pertanto, la realizzazione dei dizionari tramite vettori ordinati risulta conveniente quando sul dizionario si effettuano principalmente operazioni di verifica di appartenenza delle chiavi.

Se le operazioni più frequenti sono invece quelle di inserimento e/o cancellazione, allora la realizzazione della struttura dati per n molto grandi risulta inefficiente.

I Dizionari: Realizzazione tramite tabella Hash

La realizzazione della struttura dati **dizionario** tramite tabella **hash** si basa sull'idea di ricavare la posizione che la chiave dovrebbe occupare nel vettore direttamente dal valore della chiave k stessa. Con questa realizzazione, se fattibile, ogni operazione richiederebbe tempo $\mathcal{O}(1)$.

Sia K l'insieme di tutte le possibili chiavi distinte e si memorizzi il dizionario in un vettore V di dimensione m .

La soluzione ideale sarebbe quella di avere una funzione

$$H : K \longrightarrow \{1 \dots m\}$$

che permetta di ricavare la posizione $H(k)$ di una chiave k nel vettore V , e tale che valga la seguente relazione di biunivocità:

$$\forall k_1, k_2 \in K, k_1 \neq k_2, H(k_1) \neq H(k_2)$$

Per garantire che la funzione H sia biunivoca occorre che $m \geq |K|$.

Ad esempio, se utilizzo un vettore V di lunghezza $m = |K|$, possiamo accedere direttamente alla posizione di una chiave k in $\mathcal{O}(1)$.

Purtroppo questa realizzazione é **improponibile** per un numero di chiavi molto grande perché richiede una quantità enorme di memoria.

Infatti bisognerebbe allocare un vettore V molto grande anche se non contiene tutte le chiavi, causando uno spreco di memoria.

Per evitare inutili sprechi di memoria, la dimensione m del vettore deve essere scelta in modo opportuno in base al numero delle chiavi **attese** e non in base al numero delle chiavi **possibili**.

Bisogna quindi rinunciare alla biunivocità della funzione H accettando il fatto che due chiavi diverse k_1 e k_2 possano occupare la medesima posizione nel vettore V .

La soluzione estrema che minimizza l'uso dello spazio di memoria é quella che utilizza un vettore V di una sola posizione contenente un riferimento ad una lista. In questo caso l'occupazione della memoria sarebbe lineare nel numero degli elementi, ma anche il costo delle operazioni sarebbe $\mathcal{O}(n)$.

Abbiamo quindi bisogno di una soluzione di compromesso avente le seguenti caratteristiche:

- ▷ un vettore V ad m posizioni con $m > 1$ e $m \ll |K|$
- ▷ mantenga basso il tempo di esecuzione delle operazioni, ad esempio $\mathcal{O}(1)$ come nel caso in cui $m = |K|$
- ▷ mantenga bassa l'occupazione di memoria, ad esempio $\mathcal{O}(n)$ come nel caso di $m = 1$

I Dizionari: Realizzazione tramite tabella Hash

Se rinunciamo alla biunivocità della funzione H bisogna risolvere il problema di dove posizionare due chiavi k_1, k_2 per cui $H(k_1) = H(k_2)$.

Esempio: sia K l'insieme di nomi dei persona.

Si consideri la funzione $H(k) = h$, $1 \leq h \leq 21$, che per ogni nome calcola la posizione corrispondente alla posizione della prima lettera del nome nell'alfabeto.

Quindi ad esempio $H(\text{FABIO}) = 6$.

Poiché la funzione H può generare 21 valori diversi, consideriamo un vettore V di $m = 21$ posizioni.

Ovviamente la funzione H non è biunivoca poiché $H(\text{FABIO}) = H(\text{FABRIZIO})$.

Inizialmente il vettore V è vuoto, quindi quando inserisco la chiave FABIO non ho alcun problema. Quando provo ad inserire la chiave FABRIZIO, trovo la posizione 6 occupata quindi devo decidere cosa fare.

Una soluzione potrebbe essere quella di posizionare la chiave FABRIZIO in posizione 7.

Se adesso inserisco la chiave GIANNI, poiché $H(\text{GIANNI}) = 7$ è occupata devo posizionarla in posizione 8, mentre, ad esempio, la chiave successiva ALBERTO viene posizionata in posizione 1.

Se voglio verificare se una chiave appartiene al dizionario devo procedere allo stesso modo, rischiando di dover scorrere tutto il vettore per trovare una chiave la cui lettera iniziale é F, mentre per trovare una chiave con lettera iniziale A mi occorre un tempo di accesso costante.

La non uniformità dei tempi di accesso é dovuta al fatto che si stanno formando degli agglomerati dovuti alla scarsa bontá della funzione H che non distribuisce uniformemente le chiavi sul vettore.

Inoltre, il metodo di ricerca adottato per trovare una posizione libera per una chiave che ha trovato la propria posizione occupata favorisce la formazione di agglomerati.

Infatti, se una posizione vuota é preceduta da j posizione occupate, allora la probabilità che sia la prossima ad essere occupata é data dalla probabilità che la prossima chiave voglia occupare una delle j posizioni, ovvero $\frac{1}{m}(j + 1)$, contro $\frac{1}{m}$ nel caso in cui la posizione precedente sia libera.

Ricapitolando ci serve:

- ▷ una funzione hash H calcolabile in $\mathcal{O}(1)$ che distribuisca uniformemente le chiavi sul vettore
- ▷ ci serve un metodo di ricerca di una posizione libera che non favorisca il formarsi di agglomerati

La dimensione del vettore V deve essere sovrastimata, ad esempio il doppio delle chiavi attese, in modo da evitare di riempire completamente V .

Funzioni Hash

Una funzione **hash** é una funzione che dato un valore di input restituisce un valore di output il piú possibile **scorrelato** dal valore di input. La funzione hash **non** deve essere una funzione casuale poiché calcolando piú volte la funzione sulla medesima chiave deve restituire sempre lo stesso valore.

Sia V un vettore ad m posizioni, ed utilizziamo, per comoditá, la seguente notazione:

- ▷ $\text{bin}(k)$ é la rappresentazione binaria di una chiave k
- ▷ $\text{int}(b)$ é il valore intero corrispondente ad una rappresentazione binaria b

Se la chiave é una stringa, allora la rappresentazione binaria é data dalla concatenazione della rappresentazione binaria del valore ordinale di ogni carattere.

Le seguenti sono quattro buoni esempi di funzioni hash:

- ▷ $H(k) = \text{bin}(b)$, ove b é una porzione di p bits della rappresentazione binaria di k . Solitamente la porzione é estratta dal centro.
- ▷ $H(k) = \text{int}(b)$, ove b é dato dalla somma bit a bit di diversi sottoparti di p bit della rappresentazione binaria di k
- ▷ $H(k) = \text{int}(b)$, ove b é un sottoinsieme di p bit della stringa binaria ottenuta come $\text{bin}(\text{int}(\text{bin}(k)))^2$
- ▷ $H(k)$ é uguale al resto della divisione intera tra $\text{int}(\text{bin}(k))$ e m

Le prime 3 definizioni assumono $m = 2^p$, l'ultimo pressuppone che m sia un numero primo.

Funzioni Hash Esempio

Data una chiave k di lunghezza **MAXLENCHIAVE**, per calcolare la funzione hash

$$H(k) = \text{int}(\text{bin}(k)) \bmod m$$

bisogna calcolare la rappresentazione binaria di k e successivamente l'intero ad esso corrispondente.

Se supponiamo che ogni carattere sia rappresentato con 8 bit (rappresentazione ascii), $\text{int}(\text{bin}(k))$ può essere interpretato come un numero in base 256.

Quindi, se indichiamo con k_0, k_1, \dots, k_n rispettivamente i valori ordinali dei caratteri che compaiono nella stringa da destra verso sinistra, bisogna calcolare il seguente valore:

$$\text{int}(\text{bin}(k)) = k_0 + k_1 \cdot 256 + k_2 \cdot 256^2 + \dots + k_n \cdot 256^n$$

Per calcolare tale valore è conveniente applicare la **regola di Horner** per la valutazione di un polinomio in un punto.

Dato un polinomio $p(x) = a_q x^q + a_{q-1} x^{q-1} + \dots + a_1 x + a_0$ può essere scritto come

$$p(x) = x(\dots(x(a_q x) + a_{q-1})\dots) + a_1) + a_0$$

che permette di valutare $p(x)$ utilizzando q prodotti e q addizioni.

Il seguente programma C calcola la funzione hash $H(k)$ applicando la regola di Horner ed effettuando l'operazione di modulo ad ogni passo per evitare che b divenga troppo grande.

La dimensione massima delle chiavi é supposta di 6 caratteri ed ogni carattere è rappresentato su 8 bit.

```
#define MAXLENCIAVE 6
#define m          383 // m numero primo, stima per eccesso del numero di chiavi attese

int H ( char * k ) {
    int b, j;

    b = k[0];

    for ( j = 1; j < MAXLENCIAVE; j++ )
        b = ((b * 256) + k[j]) mod m // b = ((b << 8) + k[j]) mod m

    return b;
}
```

Esempio:

$$H(\text{fabio}) = (256^5 * "f" + 256^4 * "a" + 256^3 * "b" + 256^2 * "i" + 256^1 * "o" + 256^0 * "") \% m = 86$$

Metodi di Scansione

Se per una chiave k si trova la propria posizione occupata bisogna ricercare una posizione libera. I metodi di ricerca detti di scansione si distinguono in **esterni** ed **interni** a seconda che eseguano o meno la ricerca all'esterno o all'interno del vettore.

Il metodo di scansione esterno é quello detto a **liste di trabocco**. In pratica ogni posizione del vettore contiene il riferimento ad una lista ove vengono memorizzate tutte le chiavi che hanno il medesimo indirizzo hash.

Il metodo di scansione con liste di trabocco ha i seguenti vantaggi:

- ▷ non impone alcuna limitazione alla dimensione del dizionario
- ▷ evita il formarsi di agglomerati
- ▷ le operazioni di **APPARTIENE**, **INSERISCI** e **CANCELLA** si realizzano rispettivamente con procedure di:
 - ▷ ricerca, e quindi ha un costo lineare nella dimensione della lista
 - ▷ inserzione in testa, quindi costo costante
 - ▷ cancellazione dell'elemento k , quindi costo lineare

Tutte operano sull'elemento del vettore $V[H(k)]$

I costi delle operazioni sono veri se assumiamo una implementazione efficiente delle liste.

La dimensione m del vettore V deve essere scelta in modo da evitare di avere troppe liste vuote o poche liste piene.

Metodi di Scansione Interni

Se non possiamo usare le liste allora si può ricorrere a metodi di scansione interna, cioè che operano all'interno del vettore.

Sia f_i la funzione da utilizzare in un metodo di scansione interna, quando si trova la posizione $H(k)$ di una chiave k occupata per l' i -esima volta.

Diverse definizioni di f_i danno origine a metodi di scansione interna diversi di cui i più utilizzati sono:

- ▷ scansione lineare: $f_i = (H(k) + h * i) \bmod m$, con h numero primo (se $h = 1$ si ottiene una scansione a passo unitario)
- ▷ scansione quadratica: $f_i = (H(k) + h * i + \frac{i * (i - 1)}{2}) \bmod m$, con m numero primo.
- ▷ scansione pseudocasuale: $f_i = (H(k) + r_i) \bmod m$, dove r_i è l' i -esimo numero random generato in modo pseudo-casuale
- ▷ scansione ad hashing doppio: $f_i = (H(k) + i * F(k)) \bmod m$, ove F è un'altra funzione hash.

Se si utilizza un metodo di scansione interna e si permette l'operazione di cancellazione delle chiavi sorge un problema quando, durante la ricerca di una chiave k si incontra una posizione vuota. Infatti in questo caso non si è sicuri se la chiave non esista oppure è stata memorizzata altrove poiché al momento della sua inserzione la posizione risultava occupata.

Pertanto bisogna procedere nella ricerca finché o si incontra una posizione mai riempita o si ritorna su una posizione già visitata.

Metodi di Scansione Interni

Definiamo una procedura C **SCANDISCI** che effettua una scansione lineare a passo unitario. La procedura termina :

- ▷ quando trova la chiave k
- ▷ quando incontra la prima posizione vuota, cioè mai scritta
- ▷ dopo aver visitato tutto il vettore

La procedura accetta in input una chiave k ed un dizionario V e calcola due valori, j e c :

- ▷ j corrisponde alla posizione su cui termina la procedura
- ▷ c corrisponde alla prima cella **cancellata** o **vuota** incontrata durante la scansione

Si suppone quindi, che le celle del vettore possano contenere i seguenti valori:

- ▷ il valore di una chiave k
- ▷ un particolare valore **vuoto** che denota una cella mai scritta
- ▷ un particolare valore **cancellato** che denota una cella da cui é stata cancellata una chiave

Utilizziamo come dizionario la seguente struttura dati:

```
#define vuoto      = const1;  
#define cancellato = const2;  
  
typedef tipoelem[m] dizionario;
```

Procedura SCANDISCI

```
// k = chiave , V = dizionario , *j = posizione su cui termina SCANDISCI,
//                               *c = posizione cella vuota o cancellata
void SCANDISCI ( chiave k, dizionario V, int * j, int * c ) {
    int i;
    boolean libero    = 0; // flag che vale True se ho incontrato la prima cella vuota o cancellata
    boolean trovato   = 0; // flag che vale True se ho trovato la chiave
    boolean esaurito  = 0; // flag che vale True se ho visitato tutto il vettore

    i = H(k); // posizione calcolata dalla funzione hash
    *j = i;    // inizialmente *j ha lo stesso valore di i
    c = i;    // inizialmente c ha lo stesso valore di i

    while ( ( V[*j] != vuoto ) && ( ! (trovato or esaurito) ) ) {

        if ( V[*j] == k )
            trovato = 1;           // trovata la chiave cercata
        else {
            *j = (*j+1) mod m;    // incremento unitario in modulo il valore di *j

            if ( ! libero && ((V[*j] == cancellato) || (V[*j] == vuoto)) ) {
                *c = *j; // inizializzo *c con il valore della prima posizione non piena incontrata
                libero = 1; // flag libero vale True
            }

            esaurito = (i == *j) ? 1 : 0; // ho visitato tutto il vettore ?
        }
    }
}
```

Utilizzando la procedura **SCANDISCI** possiamo realizzare le procedure **APPARTIENE**, **INSERISCI** e **CANCELLA**.

Infatti, dopo aver chiamato la procedura **SCANDISCI**, valgono le seguenti considerazioni:

- ▷ se $V[j] == k$ allora la chiave appartiene al dizionario. In questo caso può essere eventualmente cancellata
- ▷ se $V[j] \neq k$ allora la chiave non appartiene al dizionario. In questo caso, se
 $V[c] == \text{vuoto}$ oppure $V[c] == \text{cancellato}$
 possiamo inserire una nuova chiave in posizione c
- ▷ se
 - ▷ $V[j] \neq k$ e
 - ▷ $V[c] \neq \text{vuoto}$ e $V[c] \neq \text{cancellato}$

allora il vettore è stato completamente scandito, non abbiamo trovato la chiave e non abbiamo individuato alcuna posizione libera, quindi non è possibile alcuna altra inserzione.

```
boolean APPARTIENE ( chiave k, dizionario V ) {  
    int j, c;  
  
    SCANDISCI(k, V, &j, &c);  
    return ( V[j] == k ) ? 1 : 0;  
}
```

```
void INSERISCI ( chiave k, dizionario V ) {  
    int j, c;  
  
    SCANDISCI(k, V, &j, &c);  
    if ( (V[j] <> k) && ( (V[j] == vuoto) || (V[c] == cancellato)) )  
        V[c] = k;  
}
```

```
void CANCELLA ( chiave k, dizionario V ) {  
    int j, c;  
  
    SCANDISCI(k, V, &j, &c);  
    if (V[j] == k)  
        V[j] = cancellato;  
}
```


Conclusioni

La realizzazione di dizionari tramite tabelle Hash dipende fortemente dalla qualità della funzione Hash.

Sebbene nel caso pessimo la complessità delle operazioni rimane lineare nel numero degli elementi del dizionario, si può dimostrare che nel caso medio la complessità è drasticamente inferiore.

Riassumendo, l'implementazione di dizionari per mezzo di tabelle hash ha il seguente costo:

▷ caso pessimo:

▷ APPARTIENE $\in \mathcal{O}(n)$

▷ INSERISCI $\in \mathcal{O}(n)$

▷ CANCELLA $\in \mathcal{O}(n)$

ove n è il numero degli elementi del dizionario. Infatti nel caso pessimo ogni operazione richiede la scansione di tutto il vettore.

▷ caso medio, supponendo di avere una buona funzione hash ed un buon metodo di scansione:

▷ APPARTIENE $\in \mathcal{O}(1)$

▷ INSERISCI $\in \mathcal{O}(1)$

▷ CANCELLA $\in \mathcal{O}(1)$