Algoritmica 07/08
16. TABELLE HASH

Queste tabelle sono strutture di dati che consentono la ricerca, l'inserzione e in certi casi la cancellazione di n chiavi in tempo medio costante, e tempo pessimo $\Theta(n)$. Poiché tutte le chiavi sono sequenze binarie, esse possono essere interpretate come numeri interi.

Una *funzione hash* h ha come dominio l'insieme C di tutte le possibili chiavi, e come codominio l'insieme {0, ..., m-1} per un dato intero m. Essa trasforma quindi ogni possibile chiave K di C in un *indirizzo hash* h(K) tra 0 e m-1, inteso come posizione per K in un array A[0..m-1]. Idealmente la chiave K sarà memorizzata in A[h(K)].

Le chiavi possibili sono in genere moltissime e non note a priori (per esempio i codici di login in un sistema, o le variabili che un utente inserirà in un programma, o i cognomi dei clienti di una ditta), quindi si ha |C| >> m. Dunque è inevitabile che nell'applicazione si presentino due o più chiavi K1, K2 ... con lo stesso indirizzo hash, cioè h(K1)=h(K2)=... Nasce quindi un problema di *collisioni*: solo una delle chiavi (tipicamente la prima che si è presentata, diciamo K1) potrà essere allocata in A[h(K1)], e le altre saranno poste altrove.

Dobbiamo quindi affrontare tre problemi:
1) come si sceglie la dimensione m;
2) come si calcola la funzione h;
3) come si risolvono le collisioni.

1. **Scelta di m e definizione del fattore di carico**

In ogni istante si indica con n il numero di chiave presenti nella tabella: dunque n varia durante l'applicazione mentre la dimensione m del vettore A è fissata all'inizio. Si definisce il *fattore di carico* α = n/m e si sceglie m in modo che presumibilmente α non superi 0.9: se ciò dovesse accadere si raddoppia la dimensione di A e si riallocano tutte le chiavi nel nuovo vettore.

In genere m si sceglie come potenza di due o come numero primo (vedi oltre).

2. **Calcolo della funzione hash**

Tra tanti metodi ne indichiamo due, usati rispettivamente per m potenza di 2 o per m primo. In entrambi i casi, se si ammette che tutte le sequenze di bit delle chiavi siano equiprobabili, per ogni chiave K l'indirizzo hash h(K) ha valore tra 0 e m-1 con pari probabilità 1/m. In questo caso la funzione hash si dice **semplicemente uniforme.**

a) Poniamo che sia m=2^s, quindi le posizioni di A sono indirizzate con s bit. La sequenza binaria che rappresenta la

chiave K è divisa in parti di s bit ciascuna, e tra esse si calcola lo XOR bit a bit per ottenere una sequenza di s bit che rappresenta h(K). Si noti che tutti i bit della chiave contribuiscono a formare h(K).

b) Poniamo che m sia un numero primo. Per ogni chiave K si pone h(K)=K mod m (cioè h(K) è il resto della divisione tra K e m).

3. Risoluzione delle collisioni con indirizzamento aperto

Vi sono due tecniche di base dette *indirizzamento aperto* e *concatenazione.* Trattiamo qui la prima e rimandiamo la seconda al prossimo paragrafo.

Affrontiamo il problema di una tabella inizialmente vuota che si riempie per inserzioni successive. La cancellazione di elementi è più complicata e sarà discussa a parte. Il principio è quello di allocare ogni chiave K in A[h(K)] se tale posizione è libera, altrimenti percorrere la tabella (array A) con un'opportuna *legge di scansione* fino a trovare una posizione libera per K. Come vedremo si possono usare diverse leggi.

Notiamo subito che una successiva ricerca di K ci farà percorrere esattamente le stesse posizioni incontrate nell'inserzione. Dunque, come accade negli alberi di ricerca, gli algoritmi di inserzione e ricerca sono sostanzialmente uguali, salvo che il primo procede sempre fino a trovare una posizione libera, il secondo si ferma se incontra una posizione occupata da K. Diversamente dagli alberi, perché ciò non dia errori è necessario che nessuna chiave K' inserita prima di K sia successivamente cancellata, perché l'inserzione di K sarebbe potuta passare attraverso la posizione occupata da K' per procedere in cerca di una posizione libera: togliendo K' una successiva ricerca di K incontrerebbe tale posizione ora vuota e concluderebbe che K non è presente. Il problema della cancellazione sarà ripreso in seguito.

Per tutte le leggi di scansione, posto n<m l'algoritmo di ricerca o inserzione è del tipo seguente, ove il contatore c indica il numero di posizioni toccate, e la nuova posizione INCR(c) calcolata a ogni passo dipende dalla legge scelta.

```
HASHRICINS(A,K,OP) \\OP=0 per ricerca, OP=1 per inserzione
  p = h(K); c = 1;
  while (A[p]≠K && A[p]≠null) {p = INCR(c); c=c+1;}
  if (A[p]=K) return <p,c>;
  \\ A[p]=null
  if (OP=0) return <null,c>
    else {A[p]=K; n=n+1; return <p,c>;}
```

3.1 Legge di scansione lineare.

Calcolato h(K) si procede in A attraverso le posizioni $h_0(K)$, $h_1(K), h_2(K), \ldots$ ove $\mathbf{h_i(K)=[h(K)+qi]_{mod\ m}}$, i=0,1,2...; ovvero:

```
INCR(c) = (h(K)+q*c)%m; o meglio l'equivalente
INCR(c) = (p+q)%m;
```

Il *passo di scansione* q è un intero costante primo con m. L'addizione in modulo assicura che superata la posizione m-1 si ritorni in testa al vettore A. La primalità tra q e m garantisce che si possano raggiungere tutte le posizioni di A. Normalmente questa legge si esegue con passo q=1, ovvero INCR(c) = (p+1)%m.

Studiare il seguente esempio, in cui la chiave 15 collide con 4; 17 collide con 28; 86 collide con 31, 43 e 22; 60 collide con 15, 28 e 17. Alla fine si ha α = 9/11 = 0.82.

Esempio 1

m=11; h(K)=K%11 (metodo b) del par. 2);
chiavi da inserire: 43,22,31,4,15,28,17,86,60

```
posizione:   0  1  2  3  4  5  6  7  8  9 10
       A:  22           4 15          31 43   dopo 5 inserzioni
       A:  22 86        4 15 28 17 60 31 43   dopo 9 inserzioni
```

Nella scansione lineare il **numero medio** S di posizioni toccate da una ricerca con successo (cioè la media dei valori del contatore c restituiti dalla procedura HASHRICINS per tutte le chiavi inserite), può essere calcolato con la formula approssimata: $\mathbf{S = (1-\alpha/2)/(1-\alpha)}$, con approssimazione molto buona per $\alpha \leq 0.9$. La formula si riferisce all'impiego di una funzione hash semplicemente uniforme (paragrafo 2), quindi all'atto pratico i valori di S possono risultare leggermente superiori. Si noti comunque che S è funzione di n e m solo attraverso il loro rapporto.

Impiegando la scansione lineare si formano in A, al crescere di α, gruppi di chiavi consecutive mod m (o a distanza q, se q>1) detti **agglomerati**. Nell'esempio, dopo cinque inserzioni si hanno gli agglomerati 4-15 e 31-43-22; dopo nove inserzioni tutte le chiavi formano un unico agglomerato. Se una nuova chiave cade in esso, per esempio la chiave 18 per cui h(18)=7, essa deve percorrere l'agglomerato fino in fondo prima di trovare la posizione libera 2. Questo fenomeno mostra che le prestazioni degradano fortemente all'aumentare di α. Dalla formula sopra riportata si hanno i valori in tabella 1 per la scansione lineare:

Tabella 1. Numero medio S di passi in una ricerca con successo

α:	0,10	0,50	0,75	0,90	
S:	1,06	1,50	2.50	5.50	scansione lineare
S:	1,05	1,44	1.99	2.79	scansione quadratica
S:	1,05	1,38	1.83	2.55	scansione doppio hash

Ovviamente per α = 1 una chiave può toccare al massimo n posizioni e il valore medio non può quindi superare n, mentre la formula darebbe S=∞: infatti per α > 0.9 la formula diviene progressivamente più imprecisa. In effetti la ricerca di una chiave è in media velocissima come si vede dalla tabella, ma il **caso pessimo**, che qui si verifica se tutte le chiavi hanno lo stesso indirizzo hash, richiede ordine di n passi. Torneremo su questo punto nel paragrafo 5.

3.2 Legge di scansione quadratica.

Si procede in A attraverso le posizioni $h_0(K), h_1(K), h_2(K), ...$ ove $\mathbf{h_i(K)=[h(K)+ai+bi^2]_{mod\ m}}$, i=0,1,2...; ovvero:

INCR(c) = (h(K)+a*c+b*c^2)%m;

Il passo di scansione è ora variabile e una tale legge in genere non raggiunge tutte le posizioni della tabella (cioè dopo un numero di passi <m le posizioni toccate possono ripetersi all'infinito). Vediamo dunque il motivo di considerare questa legge.

L'**agglomerazione** delle chiavi discussa per la scansione lineare, responsabile del degrado delle prestazioni al crescere di α, è detta **primaria**. In quel caso tutte le nuove chiavi il cui indirizzo hash cadesse nello stesso agglomerato farebbero in esso un identico percorso competendo per la stessa posizione libera alla fine dell'agglomerato stesso. In sostanza, col crescere di α e quindi degli agglomerati, il vantaggio sulla distribuzione delle chiavi dovuto al fatto che chiavi diverse hanno indirizzi hash diversi va progressivamente scemando. La scansione quadratica procede invece, per ogni chiave, con passi diversi tra loro: due chiavi con indirizzi hash diversi possono collidere in una posizione p solo se ciò avviene in passi diversi della loro scansione (cioè per diversi valori di c) e di lì seguiranno percorsi differenti. Resta comunque un'agglomerazione **secondaria** che riguarda le chiavi con uguale indirizzo hash, le quali anche con la legge quadratica seguono lo stesso percorso sulla tabella e competono per la stessa cella libera.

Il fatto che non tutte le celle possono essere toccate può non essere un danno molto grave se comunque se ne toccano molte. Infatti nelle tabelle hash le chiavi si raggiungono in media con poche prove: solo chiavi particolarmente sfortunate potrebbero non trovare posto. Vi sono due scelte per le costanti a,b nella legge di scansione, particolarmente interessanti e in genere impiegate:

L1: $h_i(K)=[h(K)+ i^2]_{mod\ m}$ con m primo (cioè a=0, b=1)
L2: $h_i(K)=[h(K)+ i/2+ i^2/2]_{mod\ m}$ con $m=2^s$ (cioè a=b=1/2)

Si può dimostrare che con L1 si toccano esattamente (m+1)/2 posizioni, con L2 si toccano tutte le posizioni. La prima dimostrazione non è facile, la seconda può essere eseguita come esercizio personale di media difficoltà (vedi soluzione in appendice A). Si può inoltre dimostrare come facile esercizio che:

per L1: INCR(c) = (p+2c-1)%m;
per L2: INCR(c) = (p+c)%m;

che sono ovviamente due forme semplicissime per calcolare l'incremento: cioè, a partire da h(K), con la L1 si procede per passi lunghi 1, 3, 5, 7...., con la L2 per passi lunghi 1, 2, 3, 4... (vedi appendice A).

Con le chiavi 43,22,31,4,15,28,17,86,60 del precedente esempio 1, ponendo ancora h(K)=K%11 e la legge L1 perché m=11 è primo, abbiamo l'allocazione seguente. Si noti che tutte le chiavi, tranne 86 e 60, sono andate nelle stesse posizioni della scansione lineare perché hanno fatto al massimo due passi; 86 collide ora con 31 e 43; 60 collide con 15, 28 e 31.

```
posizione:  0  1  2  3  4  5  6  7  8  9 10
A:          22    86 60  4 15 28 17    31 43   dopo 9 inserzioni
```

Come detto questa legge permette di raggiungere (m+1)/2 = 6 posizioni diverse. Per esempio la nuova chiave 13 toccherebbe le sei posizioni 2, 3, 6, 0, 7, 5 tutte occupate, e ritornerebbe poi su 5, 7, 0, 6, 3, 2, 3 .... senza mai raggiungere le posizioni libere 1 e 8.

Nella scansione quadratica il numero medio S di posizioni toccate da una ricerca con successo può essere calcolato con una formula molto complicata i cui valori approssimati sono riportati nella tabella 1. Si noti che S è ancora funzione solo di α e che la scomparsa dell'agglomerazione primaria riduce i tempi di accesso al crescere di α. Come per la scansione lineare il caso pessimo si verifica se tutte le chiavi hanno lo stesso indirizzo hash, e la ricerca per tutte le chiavi richiede in media ordine di n passi.

3.3 Legge di scansione doppio hash.

Le posizioni $h_0(K), h_1(K), h_2(K), ...$ si determinano ora con un passo variabile *in funzione di K*. Si impiegano a tale scopo due diverse funzioni hash h'(K), h"(K), la prima per il consueto accesso iniziale alla tabella, la seconda per stabilire il passo. Il modo più semplice è quello di impiegare una scansione lineare con passo h"(K), cioè $\mathbf{h_i(K)=[h'(K)+h''(K)i]_{mod\ m}}$, i=0,1,2..., ovvero:

```
INCR(c) = (h'(K)+h"(K)*c)%m; o meglio l'equivalente
INCR(c) = (p+h"(K))%m;
```

Si noti che il valore h"(K) deve essere primo con m per ogni K, onde raggiungere tutte le posizioni della tabella (vedi paragrafo 3.1). Vi sono due scelte per h' e h" particolarmente interessanti e in genere impiegate, rispettivamente per m primo e $m=2^s$:

L3 (m primo): $h'(K) = K_{\text{mod } m}$, $h''(K) = 1+K_{\text{mod}(m-1)}$
il passo h"(K) è compreso tra 1 e m-1
L4 ($m=2^s$): h'(K) = s bit estratti da K (paragrafo 2)
h"(K) = 2b+1, con b=s-1 bit estratti da K
il passo h"(K) è dispari, quindi primo con m

Con questo metodo si contrastano le agglomerazioni primaria e secondaria, perché due chiavi con lo stesso valore di h' accedono ad A nella stessa posizione, ma hanno in genere diversi valori di h" e seguono percorsi differenti sulla tabella sin dal secondo passo.

Con i dati del precedente esempio 1, applicando legge L3 abbiamo l'allocazione seguente. Le chiavi 43,22,31,4 entrano senza collisioni e raggiungono le solite posizioni iniziali. Per le successive 15,28,17,86,60 abbiamo: h'(15)=4 (collisione con 4), h"(15)=6 (collisione con 43 e inserzione in posizione 5); h'(28)=6 (inserzione); h'(17)=6 (collisione con 28), h"(17)=8 (inserzione in posizione 3); h'(86)=9 (collisione con 31), h"(86)=7 (collisione con 15 e inserzione in posizione 3); h'(60)=5 (collisione con 15), h"(60)=1 (collisione con 28 e inserzione in posizione 7).

posizione:	0	1	2	3	4	5	6	7	8	9	10	
A:		22	86		17	4	15	28	60	31	43	dopo 9 inserzioni

Nella scansione con doppio hash il valore di S è praticamente uguale a quello, minimo in linea teorica, che si otterrebbe facendo seguire a ogni chiave un proprio percorso di inserzione costituito da una permutazione casuale delle posizioni {0,...,m-1}. Per tale situazione vale la formula: $\mathbf{S = -\ln(1-\alpha)/\alpha}$ con approssimazione molto buona per $\alpha \leq 0.9$ (ln indica il logaritmo naturale: si ha $1-\alpha < 1$, quindi il logaritmo è < 0 e S è positiva). I valori di S sono riportati nella tabella 1. Si noti che la scomparsa delle agglomerazioni primaria e secondaria riduce i tempi di accesso al crescere di α. Come per le scansioni precedenti il caso pessimo richiede in media ordine di n passi, ma ora tutte le chiavi devono avere lo stesso valore di h' e h".

4. Risoluzione delle collisioni per concatenazione

La concatenazione è un'organizzazione delle tabelle hash alternativa all'indirizzamento aperto. In essa l'array A contiene puntatori anziché chiavi, e le chiavi che collidono per avere lo stesso indirizzo hash h vengono poste in una lista il cui puntatore è contenuto in A[h]. Un vantaggio del metodo è che non è necessario stimare quante chiavi entreranno nella tabella perché l'allocazione a liste consente di inserirne un numero arbitrario: e in effetti questo metodo è più efficiente per $\alpha>1$, cioè se le liste contengono più elementi di A, cosa impossibile con gli altri metodi (come vedremo, non si può però eccedere nel valore di α).

Per le chiavi 43,22,31,4,15,28,17,86,60 dell'esempio 1, ponendo m=5 quindi α =1.8, e h(K)=$K_{mod\ 5}$, avremmo:

posizione in A:	0	1	2	3	4
lista (in verticale):	15	31	22	43	4
	60	86	17	28	

Il numero medio S di chiavi ispezionate in una ricerca con successo, secondo l'usuale ipotesi di uniformità della funzione hash, è chiaramente legato alla lunghezza media delle liste. Poniamo che sia α>1. Nel caso ottimo n è multiplo di m, e per ogni valore h tra 0,...,m-1 vi sono esattamente n/m chiavi con indirizzo hash h, cioè le liste hanno tutte la stessa lunghezza n/m=α. In questo caso si ispezionano banalmente (1+α)/2 chiavi in media (per esempio se tutte le liste contengono α =3 chiavi queste si raggiungono in media in due passi). L'analisi generale è più difficile perché le liste hanno lunghezze diverse, e alcune di esse possono essere vuote perché non è mai stato generato il corrispondente indirizzo hash. Si può dimostrare che il numero atteso di chiavi che entrano in collisione con chiavi già memorizzate è $n-m(1-e^{-\alpha})$ (nell'esempio qui sopra tale espressione vale $9-5(1-e^{-1.8})=4.85$, mentre le chiavi entrate in collisione sono 4). Tale numero sarebbe n-m se non vi fossero liste vuote, dunque il numero atteso di queste liste è m/e^{α}: al crescere di α il numero di liste vuote diminuisce, ma per α =2 più del 13% delle posizioni di A sono ancora inutilizzate. Comunque il valore atteso di S può essere valutato come **$S = 1+\alpha/2$**, poco superiore al caso ottimo, con approssimazione che migliora al crescere di α.

In sostanza il metodo di concatenazione è in media molto efficiente, ma come i precedenti è inutilizzabile al caso pessimo. Questo si verifica anche qui quando tutte le chiavi hanno lo stesso indirizzo hash e si accodano quindi in un'unica lista, richiedendo ordine di n passi in media.

Contrariamente all'indirizzamento aperto questo metodo richiede l'impiego dei campi puntatore, ma consente di eseguire semplicemente l'eliminazione di elementi (paragrafo 6).

5. Complessità della ricerca

Tutti i metodi di ricerca hash che abbiamo considerato si rivelano estremamente efficienti, richiedendo in media pochissimi passi per il reperimento di ogni chiave. In particolare è notevole il fatto che i tempi di ricerca sono funzione di n e m solo attraverso il loro rapporto α: una tabella di 100 posizioni contenente 75 chiavi, o una di 100.000 posizioni contenente 75.000, richiedono entrambe identico (e bassissimo) tempo di ricerca: naturalmente nel secondo caso si "sprecano" però 25.000 posizioni di A.

È bene sottolineare che i valori indicati per S riguardano la media dei tempi calcolata su tutta le chiavi inserite nella tabella e ricercate con pari probabilità. Nei casi reali i valori potranno spostarsi da quelli indicati, ma ciò avviene di regola solo marginalmente. È anche bene notare che l'inserzione di nuove chiavi, o la ricerca di chiavi non presenti nella tabella, devono raggiungere la fine di un agglomerato o di una lista e richiedono tempi maggiori. Anche queste funzioni sono state studiate in media e mostrano valori circa doppi di quelli qui visti.

In sostanza possiamo affermare che le tabelle hash sono le strutture di dati più efficienti in media per la ricerca e l'inserimento di chiavi. Purtroppo, rispetto per esempio agli alberi binari, hanno un comportamento disastroso al caso pessimo. In ordine di grandezza i tempi medi diventano infatti $\Theta(n)$, e ciò non si verifica solo nei casi patologici illustrati sopra. Per l'indirizzamento aperto, per esempio, $\Omega(n)$ chiavi potrebbero incontrare $\Omega(n)$ posizioni occupate nella scansione (n/10 chiavi potrebbero incontrare n/10 posizioni occupate nella scansione contribuendo al totale con $n^2/100$ passi, ovvero $\Theta(n)$ in media). Oppure, nel metodo di concatenazione, $\Omega(n)$ chiavi potrebbero scorrere liste lunghe $\Omega(n)$.

Infine, sempre nel confronto con gli alberi binari, le tabelle hash non consentono in genere l'eliminazione di elementi, come vedremo nel prossimo paragrafo.

6. Eliminazione di elementi

Se si organizza una tabella con il metodo di concatenazione, l'eliminazione di elementi può essere spontaneamente realizzata con i metodi tipici delle liste. Dopo la ricerca di un elemento da eliminare si procede a estrarlo aggiornando i relativi puntatori: non v'è praticamente nulla da spiegare in proposito.

Nell'indirizzamento aperto l'eliminazione è invece assai problematica. Riprendiamo l'esempio 1 con m=11 e scansione lineare a passo 1. Le chiavi 43,22,31,4,15,28,17,86,60 sono inserite in successione dando luogo all'allocazione già vista:

posizione:	0	1	2	3	4	5	6	7	8	9	10
A:	22	86			4	15	28	17	60	31	43

Immaginiamo di eliminare la chiave 43. Se essa venisse semplicemente tolta dalla tabella, una successiva ricerca della chiave 86, che prima passava attraverso le posizioni 9,10,0,1, si arresterebbe ora sulla posizione 10 vuota inducendo a concludere che 86 non è presente. In sostanza togliendo una chiave K dalla sua posizione p si bloccano tutte le ricerche di chiavi inserite successivamente a K, i cui percorsi sono passati per p.

Si può risolvere questo problema associando una marca T[p] di un bit a ogni posizione p dell'array A. La marca, inizialmente posta a 0, viene posta a 1 la prima volta che A[p] è occupata da una chiave, ed è lasciata a 1 nel seguito anche se tale chiave contenuta viene eliminata. L'algoritmo di ricerca includerà un test su T[p]: se la marca ha valore 1 la ricerca prosegue anche se A[p]=null. Se la posizione p si incontra successivamente nella scansione per l'inserzione di una nuova chiave K (notoriamente non contenuta in A) si porrà A[p]=K recuperando la posizione. Il difetto di questo metodo è che, col procedere di inserzioni e estrazioni, il numero di posizioni p vuote (A[p]=null) ma con T[p]=1 tende a crescere: la ricerca passa attraverso tali posizioni con un apparente aumento del valore di α rispetto al numero di chiavi presenti, e in breve tale valore può divenire critico (α prossimo a 1) con completo degrado delle prestazioni.

L'unica alternativa è che, tolta una chiave contenuta in A[p], si sposti in tale posizione un'altra chiave in A[q] la cui ricerca passava attraverso p: ciò crea la nuova posizione libera q ove deve essere spostata un'altra chiave, e così via. A tale scopo è necessario che sia ragionevolmente semplice individuare la posizione q, e che il procedimento termini con pochi spostamenti di chiavi. Ciò in effetti è possibile solo con la scansione lineare. Nell'esempio visto sopra, eliminata la chiave 43 si devono esaminare ordinatamente le chiavi 22,86 contenute nella porzione di agglomerato che segue la posizione 10 liberata, uniche che possono aver toccato tale posizione. Per ciascuna di tali chiavi si ricalcola l'indirizzo hash e si confronta con 10: si ha h(22)=0, quindi la ricerca di 22 non è passata attraverso la posizione 10 e la chiave viene lasciata in 0; si ha poi h(86)=9, quindi la ricerca di 86 è passata attraverso la posizione 10 e tale chiave è spostata in 10 liberando la posizione 1. In questo caso l'agglomerato è terminato e l'eliminazione è completata; altrimenti si continua la scansione dell'agglomerato riferendosi alla nuova posizione libera. Questo metodo è praticabile per valori di α non troppo alti, perché in questo caso gli agglomerati sono in media corti e non è necessario ricalcolare l'indirizzo hash e rilocare molte chiavi. L'algoritmo di eliminazione basato su questo metodo è riportato nell'appendice B. Si noti che con leggi di scansione diverse dalla lineare il metodo non è utilizzabile perché non è praticamente possibile determinare per quali chiavi la ricerca è passata attraverso la posizione liberata.

Appendice A

La legge di scansione hash: $h_i(K)=[h(K)+ i/2+ i^2/2]_{\text{mod } m}$ con $m=2^s$:
a) equivale alla: $h_i(K)=[h_{i-1}(K)+ i]_{\text{mod } m}$;
b) tocca tutte le m posizioni della tabella.

Dimostrazione.

Punto a)
Al passo i si ha un incremento pari a:
$H(k)+i/2+i^2/2 - (H(k)+(i-1)/2+(i-1)^2/2) = 1/2+(2i-1)/2 = i.$

Punto b)
1) Per i=m, si salta di m e si torna sulla posizione del passo precedente j=i-1.

2) Dimostriamo che per $i<m=2^s$ non si può tornare sulla posizione di un passo precedente $j\leq i-1$. Per assurdo dovremmo avere:

$H(k)+i/2+i^2/2 - (H(k)+j/2 + (j^2)/2) = cm = c2^s$
poiché, se la differenza è multipla di m, si torna sulla stessa posizione mod m. Quindi, posto i=j+d, con $1\leq d\leq m-1$ avrei:

$H(k)+ (j+d)/2+ (j+d)^2/2 - (H(k)+ j/2+ j^2/2) = d(2j+d+1)/2 = c2^s$,
ovvero $d(2j+d+1) = c2^{s+1}$. Vi sono due casi:

i) d è pari, quindi 2j+d+1 è dispari. Si avrebbe allora $d=c'2^{s+1}$, contro l'ipotesi $d<m=2^s$;

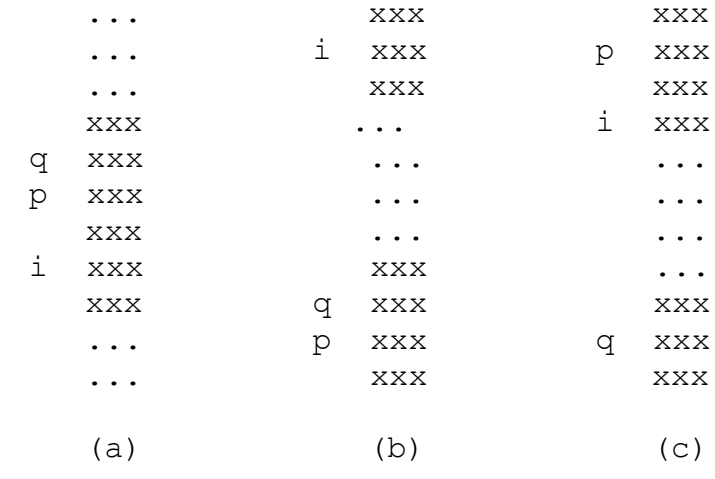
ii) d è dispari, quindi 2j+d+1 è pari. Si avrebbe allora $2j+d+1 = j+i+1 = c'2^{s+1}$ contro l'ipotesi $j<i<2^s$.

Appendice B

Estrazione di un elemento K da una tabella hash A[0..m-1] contenente n chiavi, riempita con scansione lineare a passo 1

```
HASHESTR(A,K)
  p=h(K);
  while (A[p]≠K && A[p]≠null) p=(p+1)%m;
  if (A[p]=null) return "K non è presente";
    \\ A[p]=K: si scandisce l'agglomerato
  i=(p+1)%m;
    \\ p posizione liberata; i posizione corrente
  while (A[i]≠null) {
    q=h(A[i]);
    if ((q≤p<i)||(i<q≤p)||(p<i<q)) {A[p]=A[i]; p=i;}
      \\ trasposizione della chiave da i a p
      \\ nelle situazioni (a),(b),(c) illustrate sotto
    i=(i+1)%m; }
  A[p]=null; n=n-1; return "K estratto";
```

Situazioni di trasposizione di chiave da i a p (le posizioni con XXX indicano l'agglomerato)



Bibliografia per approfondimenti

T.H.Cormen, C.E.Leiserson, R.L.Rivest, C.Stein. Introduzione agli algoritmi e strutture dati. McGraw-Hill 2005.