Indirizzamento Aperto

Sommario

- Metodo di indirizzamento aperto
  - Scansione lineare
  - Scansione quadratica
  - Hashing doppio

Metodo di indirizzamento aperto

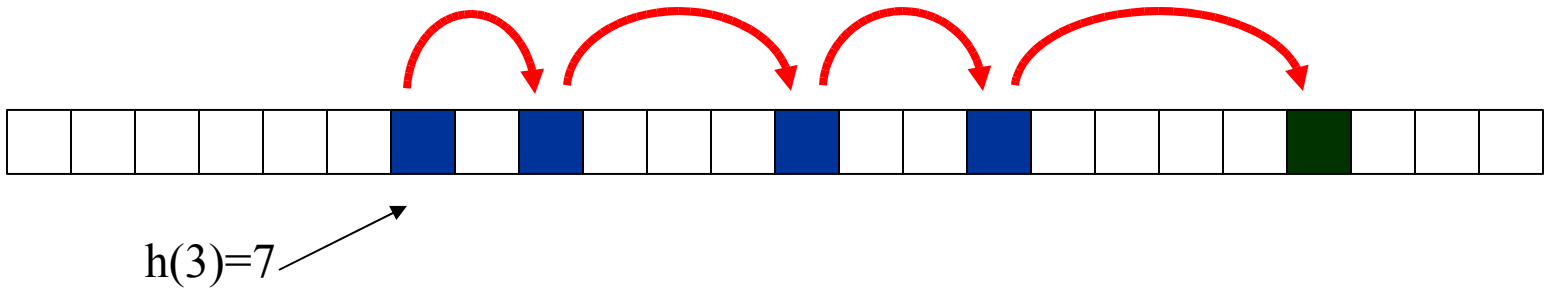
- L'idea è di memorizzare tutti gli elementi nella tabella stessa
- In caso di collisione si memorizza l'elemento nella posizione *successiva*
- Per l'operazione di ricerca si esaminano tutte le posizioni ammesse per la data chiave in sequenza

- Non vi sono liste né elementi memorizzati fuori dalla tabella
- Il fattore di carico α è sempre necessariamente ≤ 1

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

Inserimento chiave k = 3
primo valore h(3) = 7
sequenza di scansione = <7,9,13,16,21...>



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
- **NOTA:** Si passa da una funzione di tipo h(k) a una di tipo h(k,i)

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 or T[j]=NIL
8  return NIL
```

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

- E' molto difficile scrivere funzioni h che rispettino la proprietà di uniformità
- Si usano tre approssimazioni:
  - scansione lineare
  - scansione quadratica
  - hashing doppio
- Tutte queste classi di funzioni garantiscono di generare un certo numero di permutazioni ma nessuna riesce a generare tutte le m! permutazioni

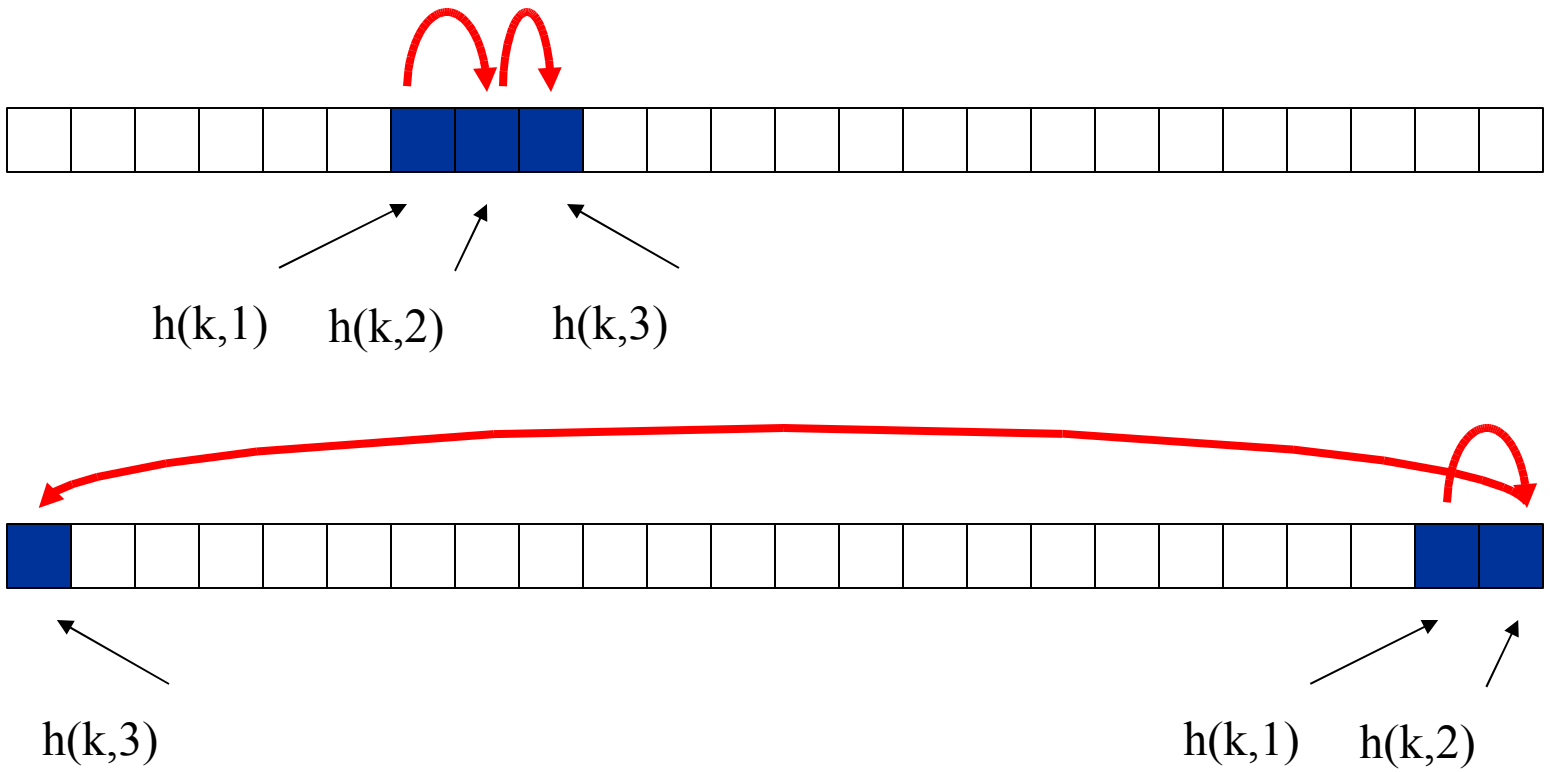
Scansione Lineare

- Data una funzione hash h':U $\rightarrow$ {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
- Si possono generare cioè sequenze con lunghezza massima pari a m

Scansione Lineare

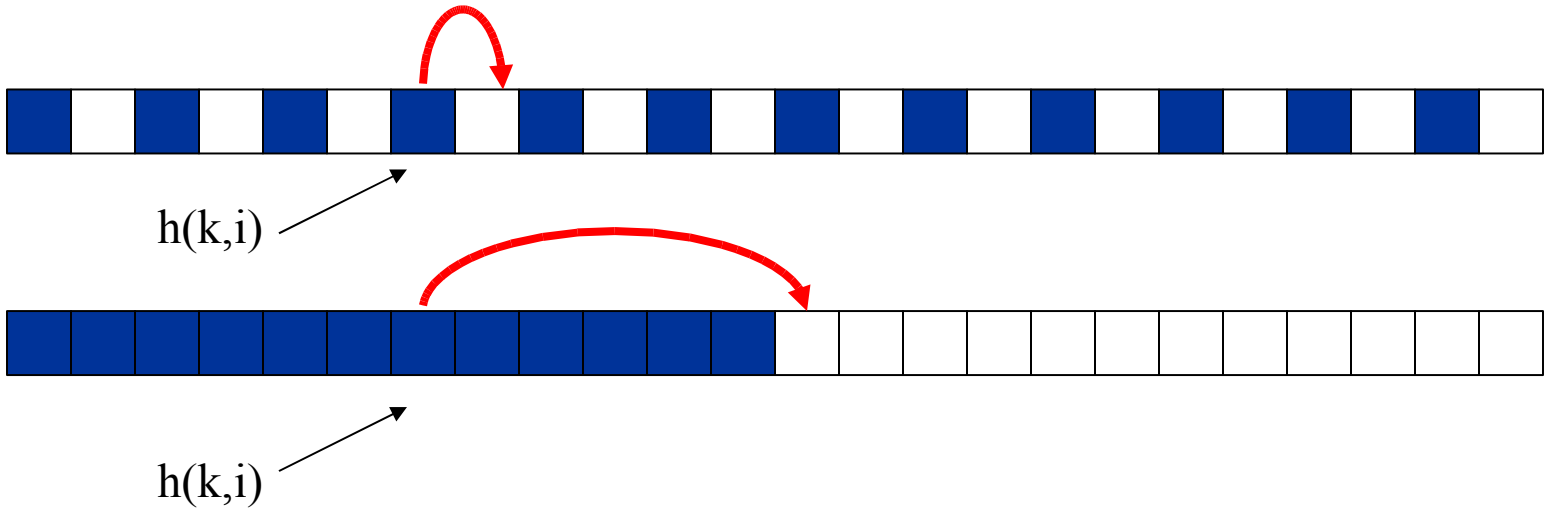


Agglomerazione primaria

- 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

Agglomerazione primaria

- 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 $\rightarrow$ {0,1,…,m-1} il metodo di scansione quadratica costruisce una h(k,i) nel modo seguente:

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

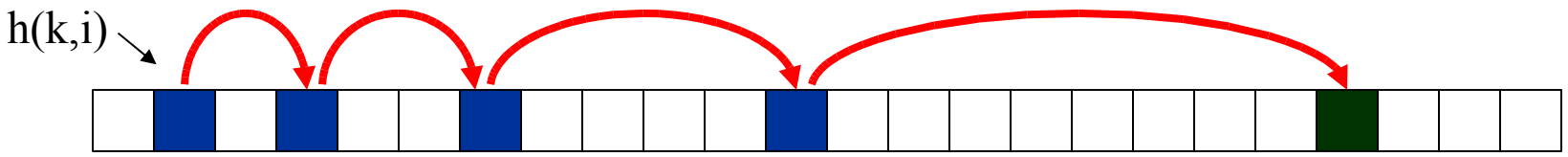
- Dove c_1 e c_2 e m sono costanti vincolate per poter far uso dell'intera tabella

Agglomerazione secondaria

- Questo metodo funziona meglio della scansione lineare perché evita l'agglomerazione primaria
- ..ma non risolve del tutto il problema della agglomerazione, si ha infatti una

  agglomerazione secondaria

  se si ha una collisione sul primo elemento, le due sequenze di scansione risultano comunque identiche
- L'agglomerazione secondaria è meno dannosa di quella primaria
- ..ma le "code" di collisione diventano comunque sempre più lunghe e questo peggiora i tempi di inserzione e ricerca



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 con hashing doppio 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

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

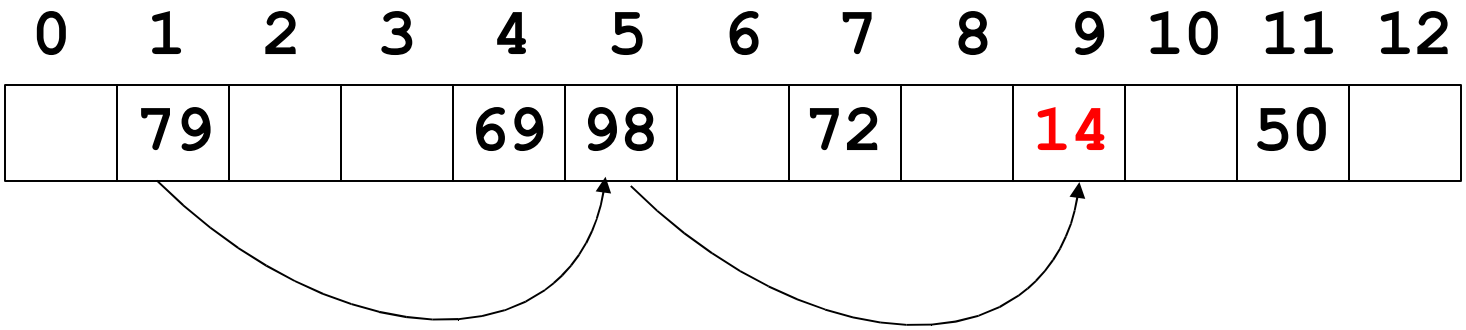
$h_2(14)=1+(14 \bmod 11) = 1 +3 = 4$

$h(14,i)=(h_1(k)+i\, h_2(k)) \bmod m = (1 + i * 4) \bmod m$

$i=0 \rightarrow$ 1 posizione esaminata $h(14,0)=1$

$i=1 \rightarrow$ 2 posizione esaminata $h(14,1)=1 + 4 = 5$

$i=2 \rightarrow$ 3 posizione esaminata $h(14,2)=1 + 4*2 = 9$



Considerazioni su h_1 e h_2

- 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 comune divisore d, allora la sequenza cercata per k esaminerebbe solo m/d elementi e non tutti e m

- 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 h_1 e h_2

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

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

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

  - dove m' è di poco minore di m; m'=m-1 o m'=m-2
  - Nota: $h_2(k)=1+\ldots$ aggiungere 1 serve a non avere la possibilità di ottenere $h_2(k)=0$

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 e quadratica generano solo O(m) sequenze; una per ogni chiave
  - l'hashing doppio genera $O(m^2)$ sequenze; una per ogni coppia (chiave, posizione)
  - ..dato che la posizione iniziale $h_1(k)$ e la distanza $h_2(k)$ possono variare indipendentemente l'una dall'altra

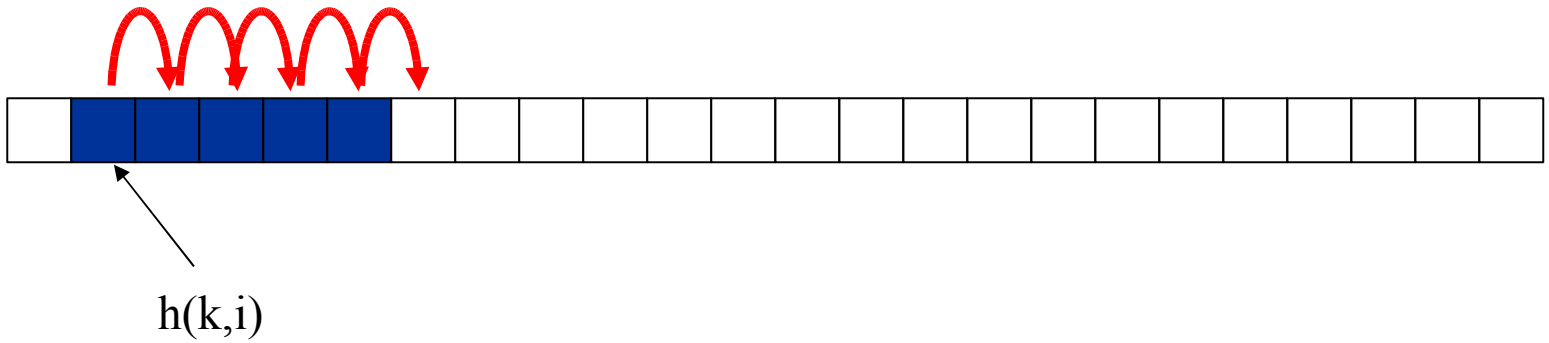
Implementazione C/C++

```
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
- Se la tabella immagazzina n elementi in m posizioni allora la probabilità che una cella sia piena è proprio

$$n/m = \alpha$$

- Pertanto un secondo accesso ad una casella piena viene fatto con probabilità α^2
- Un terzo accesso con probabilità circa α^3 ... e così via
- In media si dimostra che il numero medio di accessi per ricerca senza successo è

$$1 + \sum_{i=1..\infty} \alpha^i = \sum_{i=0..\infty} \alpha^i = 1/(1-\alpha)$$

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 si dimostra nell'ipotesi di funzione hash uniforme con chiavi equiprobabili (noi non lo faremo) essere al più

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

Numero di accessi

α	½	**1/3**	¼	**1/8**	**1/10**
hit	1.4	1.2	1.1	1	1
miss	2	1.5	1.3	1.1	1.1

α		**2/3**	¾	**7/8**	**9/10**
hit		1.6	1.8	2.3	2.5
miss		3	4	8	10

Confronto fra metodi

- E' complicato confrontare i costi/benefici per occupazione di memoria e velocità dei metodi di hashing con concatenazione e ad indirizzamento aperto
- Infatti il fattore di carico α 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 predire la dimensione dell'insieme delle chiavi

La cancellazione

- L'operazione di cancellazione per tabelle con indirizzamento aperto è difficile
- Non basta marcare la posizione come vuota con NIL perché questo impedirebbe la ricerca degli elementi successivi nella sequenza
- Due soluzioni:
  - marcatori
  - reinserimento

Marcatori

- Si usa uno speciale marcatore DELETD invece di NIL
- La ricerca non si ferma quando si trovano celle marcate con deleted, ma prosegue
- La procedura di inserzione sovrascrive il contenuto delle celle marcate con deleted
- *NOTA: in questo modo però i tempi non dipendono più solo dal fattore di carico*
- In alternativa si puo' tenere il conto di quanti elementi sono stati cancellati e quando il conteggio supera una soglia critica si puo' cambiare la dimensione del dizionario (vedi dopo)

Reinserimento

- Si reinseriscono tutti gli elementi che la cancellazione renderebbe irraggiungibili
  - cioe' tutti gli elementi nella sequenza di scansione a partire dall'elemento cancellato fino alla prima cella libera (marcata con nil)
- Se la tabella e' sparsa (fattore di carico basso) questa operazione richiede il reinserimento solo di pochi valori

Tabelle Hash Dinamiche

- Quando il fattore di carico tende a uno (cresce) nelle tabelle hash che utilizzano l'indirizzamento aperto (liste di concatenazione) le prestazioni decrescono
- Per mantenere prestazioni accettabili si ricorre al ridimensionamento della tabella:
  - quando si oltrepassa un certo fattore di carico si raddoppia la dimensione della tabella
  - quando si scende sotto un certo fattore di carico si dimezza la dimensione della tabella
- Ogni volta che si ridimensiona una tabella hash cambia anche la funzione hash e l'associazione chiave-valore hash calcolata precedentemente non e' piu' valida pertanto si devono reinserire tutti gli elementi!

Strategia

- Si deve fare attenzione a non avere soglie per innescare l'espansione e la riduzione troppo vicine ..
- .. altrimenti per frequenti operazioni di inserzione/cancellazione si consuma molto tempo a ridimensionare la tabella
- Una buona strategia e':
  - raddoppiare la dimensione quando il fattore di carico supera ½
  - dimezzare la dimensione quando il fattore di carico scende sotto 1/8
- In questo modo si garantisce la proprieta' per la quale:
  - *una sequenza di t operazioni fra ricerche, inserimenti e cancellazioni puo' essere eseguita in tempo proporzionale a t*
  - *inoltre si usa sempre una quantita' di memoria al piu' pari a una costante moltiplicata per il numero di chiavi nella tabella*

Prestazioni

- Sebbene l'operazione di espansione o riduzione della dimensione di una tabella sia molto costosa, questa viene fatta solo raramente
- Inoltre ogni volta che raddoppiamo o dimezziamo la dimensione il fattore di carico della nuova tabella e' 1/4 quindi l'inserimento sara' efficiente (ogni inserimento provochera' <2 collisioni)
- Si dimostra che raddoppiando/dimezzando e inserendo/cancellando elementi al massimo si raddoppia il numero di operazioni (inserimenti/cancellazioni) pertanto la complessita' rimane lineare rispetto alla complessita' del metodo hash non dinamico

Spiegazione dettagliata

- vogliamo determinare il numero di operazioni aggiuntivo (in proporzione agli inserimenti) da fare nel caso in cui raddoppiamo la dimensione della tabella ogni volta che il fattore di carico supera ½
- per fare questo si considera che dopo il raddoppio gli elementi da inserire sono N/4 rispetto alla nuova dimensione N della tabella
- inoltre prima di innescare una nuova espansione si devono inserire altri N/4 elementi
- quindi per inserire N/2 elementi ho inserito la prima volta N/4 elementi, poi ho raddoppiato (altri N/4 inserimenti) e poi ho aggiunto i nuovi N/4 elementi,
- quindi il numero di operazioni per inserire 2/4N e' 3/4N ovvero non ho nemmeno raddoppiato le operazioni

Spiegazione dettagliata

- vogliamo determinare il numero di operazioni aggiuntivo (in proporzione alle cancellazioni) da fare nel caso in cui dimezziamo la dimensione della tabella ogni volta che il fattore di carico diventa inferiore a 1/8
- per fare questo si considera che dopo il dimezzamento gli elementi da inserire sono N/4 rispetto alla nuova dimensione N della tabella
- inoltre prima di innescare un nuovo dimezzamento si devono cancellare altri N/8 elementi
- quindi per cancellare N/8 elementi ho cancellato N/8 elementi e poi ho dimezzato la tabella dovendo conseguentemente effettuare N/8 inserimenti
- quindi il numero di operazioni per cancellare N/8 el

Varianti

- Sono state proposte numerose varianti e miglioramenti alle soluzioni presentate
- hashing ordinato: si mantengono liste ordinate o sequenze ordinate. La ricerca senza successo si può fermare non appena si incontra una chiave maggiore senza dover terminare la scansione
- hashing di Brent: si spostano le chiavi durante le ricerche senza successo in modo da migliorare le prestazioni delle ricerche successive
- coalescing hash: si usa un metodo ibrido con liste concatenate i cui nodi sono memorizzati direttamente nella tabella hash

Alberi binari di ricerca e hash

- La scelta di utilizzare strutture dati di tipo alberi binari di ricerca o hash per implementare ADT per la ricerca 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 (cancellazione)
  - buone prestazioni anche nel caso peggiore
  - supportano un numero maggiore di operazioni (ordinamento)