9 - Dizionari

Abbiamo compreso come con l'utilizzo degli insiemi possiamo di fatto utilizzare l'algebra degli insiemi per operare su strutture di dati che necessitano le operazioni di unione, intersezione, ecc...
Ma esistono in informatica molte applicazioni che necessitano una struttura come gli **insiemi** ma che non richiedono l'uso delle loro funzioni tipiche, per questo nascono i **dizionari**, definiti come **sottotipi del tipo di dato astratto del padre** (ovvero la classe degli insiemi).

Struttura dei Dati e la Coppia Chiave-Valore

Nei dizionari gli elementi non sono valori semplici ma sono **tipi strutturati**, per questo l'accesso a questi elementi deve avvenire tramite un riferimento ad uno specifico campo, definito come **chiave**. Di conseguenza ogni elemento assume la forma di una coppia costituita da:

- **Chiave**: Il significato della chiave non è fisso, ma è strettamente legato all'applicazione specifica che si sta realizzando.
- **Valore**: rappresenta l'informazione associata a quella chiave specifica, ove essa sia presente.

***Esempi di varie coppie di chiave-valore*:**

- **Trattamento tesi**: in questo caso il dizionario viene utilizzato per verificare la correttezza ortografica di uno scritto, in cui la **chiave** individua la parola stessa ed il **valore** è assente.
- **Gestione magazzino**: in questo esempio la **chiave** viene rappresentata dal **codice a barre** ed il **valore** contiene le informazioni relative al prodotto, come la quantità, la descrizione, ecc...
- **Compilatori e Tabelle dei simboli**: i compilatori utilizzano i dizionari per gestire le loro operazioni (LDP) chiamati **tabelle dei simboli**. Qui la **chiave** è l'identificatore della funzione e/o variabile, mentre il **valore** memorizza le informazioni di gestione necessarie al sistema, come la locazione di memoria, la dimensione del dato e la posizione in cui l'identificatore viene usato all'interno del codice.

Operazioni

I dizionari hanno delle operazioni molto simili a quelle degli insiemi, essendo essi dei casi particolari di insiemi; la loro specifica è molto simile a quella del tipo di dato astratto insieme. Le operazioni presenti nei dizionari devono consentire di gestire il ciclo di vita delle coppie *chiave-valore*.

Le funzionalità principali richieste sono:

- **Verifica**: la verifica controlla se è stata definita una chiave
- **Inserimento**: l'inserimento permette di aggiungere nuove coppie di chiave-valore
- **Cancellazione**: la cancellazione permette la rimozione invece delle coppie di chiave-valore
- **Accesso**: permette di entrare nelle informazioni del valore associato ad una chiave per aggiornarle o estrapolarle.

Specifica sintattica

Le specifiche semantiche dei dizionari indicano i domini di ingresso e i codomini di uscita. I tipi di dati coinvolti sono:

Tipo	Descrizione
`dizionario`	La struttura dati principale.
`boolean`	Valore di verità (vero/falso).
`chiave`	L'identificatore univoco per accedere ai dati.
`valore`	L'informazione associata alla chiave.

Gli operatori coinvolti sono:

```
<tr>
  <td rowspan="2"><strong>Modifica</strong></td>
  <td><code>inserisci</code></td>
  <td><code>(chiave, valore, dizionario)</code></td>
  <td><code>dizionario</code></td>
  <td><strong>Inserimento/Accesso:</strong> Aggiunge una nuova coppia (chiave-valore) o aggiorna il valore se la chiave esiste già.</td>
</tr>
<tr>
  <td><code>cancella</code></td>
  <td><code>(chiave, dizionario)</code></td>
  <td><code>dizionario</code></td>
  <td><strong>Cancellazione:</strong> Rimuove la coppia identificata dalla chiave specificata.</td>
</tr>
```

Categoria	Operatore	Input (Dominio)	Output (Codominio)	Descrizione Semantica
Costruttore	`creadizionario`	`()`	`dizionario`	Crea una nuova istanza vuota del dizionario.
Ispezione	`dizionariovuoto`	`(dizionario)`	`boolean`	Verifica se il dizionario è privo di elementi.
	`appartiene`	`(chiave, dizionario)`	`boolean`	**Verifica:** Controlla l'esistenza di una chiave specifica nel dizionario.
	`recupera`	`(chiave, dizionario)`	`valore`	**Accesso:** Recupera le informazioni presenti nel valore associato a una chiave data.

Operazioni di creazione:

- `creadizionario`, crea un dizionario vuoto, $D = \varnothing$.
- `dizionariovuoto(D)=b`, nella post-condizione `b=true` se è $D = creadizionario$, altrimenti è `b=false`.
- `appartiene(k,D)=b`, se esiste all'interno di D una coppia $< k', v | k' = k > \in D$, altrimenti `b=false`, dove k' è la chiave memorizzata e k è la chiave cercata.

Operazioni di verifica:

- `inserisci(<k,v>,D)=D'`, l'operazione di inserimento gestisce due casi distinti, l'inserimento **puro** o l'inserimento di **aggiornamento**.
  - **Caso di una nuova chiave**: se la coppia non esiste, viene creata una nuova coppia all'insieme tramite funzione di unione.

$$D' = D \bigcup < k, v >$$

  - **Caso di una chiave esistente**: se esiste già una coppia $< k', v' | k' = k >$, questa viene rimossa e sostituita con la nuova coppia.

$$D' = D \setminus < k', v' > \bigcup < k, v >$$

- `cancella(k,D)=D'`, per eseguire questa operazione deve almeno esistere una coppia $< k', v' | k' = k >$, la chiave deve esistere. Se la chiave esiste si può cancellare il

dizionario tramite differenza insiemistica rimuovendo la coppia identificata

$$D' = D \setminus < k', v' >$$

Operazione di accesso:

- `recupera(k,D)=v`, il pre-requisito è analogo alla cancellazione, ovvero $k' = k$. Successivamente viene restituito il valore $v | v = v'$ il valore viene associato alla chiave trovata.

Sebbene la specifica semantica sia simile tra l'operazione di `inserisci` e di `cancella` e `recupera`.
L'operazione `inserisci` funge sia da inserimento che da aggiornamento (sovrascrittura del valore se la chiave esiste).
Le operazioni `cancella` e `recupera` richiedono una pre-condizione **forte**: la chiave deve esistere affinché l'operazione sia definita.

Rappresentazioni

Oltre alle realizzazione classiche ereditate dalla struttura degli insiemi, i dizionari adottano realizzazione più efficienti volte ad ottimizzare i tempi di accesso e ricerca tramite l'uso o di **vettori ordinati** o **tabelle hash**.

Rappresentazione di un vettore ordinato

Si utilizza un vettore con un cursore all'ultima posizione occupata, dove è necessario definire una **relazione di ordinamento totale** sulle chiavi. Le coppie vengono memorizzate nel vettore in posizioni contigue tenendo traccia dell'ordine crescente delle chiavi a partire dalla prima cella.

In base a questa informazione risulta chiaro come sia facile trovare una chiave di appartenenza k senza dover scorrere tutto il vettore, viene utilizzata la ricerca **dicotomica**, già vista precedentemente in laboratorio di programmazione.
In breve: si confronta k con l'elemento in posizione centrale, in base all'esito, si stabilisce in quale metà del vettore proseguire la ricerca, ottenendo così una complessità computazionale di tipo **logaritmica**.

Rappresentazione in metodologia Hash

Le tipologie della forma hash si dividono in dinamica e statica, noi affronteremo lo studio di quella **statica**, concentrandoci quindi sulla dimensione fissa.
L'hash statico si divide in due varianti principali, entrambe basate su un array allocato sequenzialmente:

1. **Hash chiuso**: la variante chiusa permette la memorizzazione limitata di valori all'interno del suo spazio fisico; la sua struttura è divisa in contenitori definiti **bucket**, costituiti fino ad un certo numero di bucket, detto **maxbucket**. Ogni bucket ha una capienza massima

predefinita di nb elementi. Spesso si usa la notazione $nb = 1$ in modo tale che ogni cella ospiti una solo coppia di chiave-valore.

2. **Hash aperto**: in questo caso a differenza di quello precedente non abbiamo (quasi) il limite dei valori inseribili ed i bucket possono contenere un numero indeterminato di elementi, solitamente gestiti tramite liste di trabocco.

Basandosi sulle nozioni precedenti, non esiste un "migliore" assoluto, ma la scelta dipende dalle esigenze specifiche del programma (memoria vs flessibilità).
Tuttavia, **l'Hash Aperto è generalmente considerato più robusto e flessibile** per la maggior parte delle applicazioni generaliste. In entrambi i casi il cuore del sistema è una **funzione aritmetica** in cui la chiave diviene una posizione della tabella.

La Funzione di Accesso e di Dimensionamento

L'utilizzo della metodologia hash è un ottimo sostituto all'indirizzamento diretto, poiché è utile quando lo spazio delle chiavi possibili è proporzionale alle chiavi effettivamente attese.

Def: Sia K l'insieme di tutte le possibili chiavi distinte e sia v il vettore di dimensione m in cui si memorizza il dizionario, la **soluzione ideale** è la funzione di accesso $h : K \to \{1, \dots, m\}$ che permette di ricavare la posizione $h(k)$ della chiave cercata nel vettore v così che, se $k_1 \in K \land k_2 \in K, k_1 \neq k_2$ si ha che $h(k_1) \neq h(k_2)$ per il principio delle funzioni.

Per ottenere una garanzia di biunivocità e di accesso diretto alla posizione contente la chiave si pone $m = |K|$ e se K è grande si ottiene uno spreco enorme di memoria. Per questo la dimensione m va scelta in base al **numero di chiavi attese**. Una soluzione di compromesso in questo caso quindi è scegliere $1 < m < |K|$ di cui m dev'essere strettamente e di molto minore del valore assoluto di K.

Esempio:
L'obiettivo è inserire una lista di cognomi di musicisti in una tabella di dimensione fissa ($m = 26$). La funzione hash scelta è molto semplice:
$h(k) =$ posizione della prima lettera nell'alfabeto definiti con j, avendo per cui:
$(A = 1, B = 2, \dots, P = 16, \dots, R = 18, \dots, Z = 26)$
Una possibile funzione è quindi $h(k) = j, 1 \leq j \leq 26$

ALBINONI	1
:	
OFFENBACH	15
PALESTRINA	16
PUCCINI	17
PROKOFEV	18
ROSSINI	19
:	
	26

I primi inserimenti avvengono senza problemi come si può notare, perché le posizioni calcolate sono libere; la tabella funziona perfettamente, ogni chiave va al suo indirizzo naturale:

- Albinoni (inizia per A) → Va alla posizione 1. La cella 1 è vuota, quindi si inserisce.
- Offenbach (inizia per O) → Va alla posizione 15.
- Palestrina (inizia per P) → Va alla posizione 16.
  Il problema sorge quando proviamo a inserire altre chiavi che iniziano con la stessa lettera già utilizzata e quindi risultante uguale per la funzione, dato che la funzione hash guarda solo l'iniziale, avremmo quindi lo stesso indirizzo per chiavi diverse, avendo di fatto un **conflitto**. Questa è la definizione di **collisione** o **funzione non biunivoca**.

Una collisione si verifica quando chiavi diverse producono lo stesso risultato della funzione. Esistono funzioni hash più o meno buone, ma le collisioni non si potranno mai evitare del tutto. Nel nostro esempio la collisione viene risolta adottando una strategia semplice, la **scansione lineare**: se $h(k)$ per qualunque chiave k indica una posizione già occupata, si ispeziona la posizione successiva nel vettore. Se la posizione è piena si prova con la seguente e così via, fino a trovare una posizione libera o fino a quando si “capisce” che la tabella è completamente piena (lo notiamo dai nomi in rosso nella foto).

Una posizione **libera** può essere segnalata facilmente in fase di realizzazione da chiavi fittizie definite con *libero*. Stessa cosa avviene con la **cancellazione**, in cui le chiavi cancellate vengono sostituite con una chiave fittizia definita con *cancellato*, in modo tale che possa essere distinguibile da quelle reali e quelle *libero*.
La strategia lineare in alcuni casi, può produrre nel tempo il casuale addensamento di informazioni ,i così detti **agglomerati**, in certe zone della tabella, piuttosto che una loro dispersione:

- **Rossini** che inizia per R viene allocato all'indirizzo 18
  il sistema quindi va nella cella 18, quello segnato per la lettera R, tuttavia troverà la cella occupata dal nome **Prokofev** che è finito nella cella destinata ad R poiché è stato fatto "scalare" dalle celle precedenti, seppur iniziando con P.
  In questo modo Rossini subisce una collisione **indiretta**, dove è obbligato a spostarsi, per questo vediamo nell'esempio della foto che Rossini finisce nella cella 19, data dallo spostamento del valore R + 1 successivo che da appunto 19.

Le collisioni nelle tabelle hash

Tornando alla definizione precedente della soluzione ideale, comprendiamo come la collisione avviene quando il fattore di disuguaglianza tra $h(k_1)$ e $h(k_2)$ non viene rispettato e si ha che $h(k_1) = h(k_2)$. Statisticamente è impossibile avere un dizionario privo di collisioni poiché il calcolo statistico dato dalla dimensione m del vettore e molto inferiore al numero totale di chiavi possibili $|K|$, bisogna avere un ottima gestione del dizionario quindi, basata su due componenti:

1. Una **funzione hash** robusta che distribuisca le chiavi uniformemente per minimizzare le collisioni.
2. Un **metodo di scansione** per gestire le chiavi sottoposte a coincidenze.

Design della funzione hash

Il design della funzione hash ci permette di minimizzare le collisioni, poiché si definiscono delle buone funzioni lavorando direttamente sulla rappresentazione binaria della chiave, denotata con `bin(k)`, attraverso l'uso dei suoi metodi principali per generare gli indirizzi:

- **Estrazione**:
  si estraggono p bit dalla stringa binaria `bin(k)`, solitamente dalle posizioni centrali. L'indirizzo ottenuto è il numero intero rappresentato da questi bit.
  Contro: se le chiavi hanno pattern simili nelle posizioni scelte si andrà incontro alle collisioni.
- **Folding (XOR)**:
  si suddivide la rappresentazione binaria in diversi gruppi di bit, questi gruppi vengono sommati in **modulo 2** tramite l'operazione **XOR** (come visto in AESO). L'indirizzo ottenuto è il valore intero del risultato.
  Esempio: Più lunga è la chiave più influenti saranno i caratteri del risultato finale, aumentando la variabilità.
- **Metodo della divisione**:
  Questo è definito come metodo migliore dal punto di vista probabilistico, garantendo un ottima distribuzione nell'intervallo $[0, m-1]$. L'indirizzo è dato dal resto della divisione intera del valore della chiave per la dimensione della tabella:

  $$h(k) = int(bin(k))(mod\ m)$$

  Regola per m: La dimensione della tabella (m) deve essere dispari e preferibilmente

non una potenza di 2, se $m = 2^p$ l'hash dipenderebbe solo dagli ultimi p bit, ignorando il resto della chiave.

Esempi per comprendere meglio:

Esempio

- **Funzione hash**: si supponga che le chiavi siano di 6 caratteri alfanumerici (chiavi più lunghe sono troncate, mentre chiavi più corte sono espanse a destra con spazi). Si assuma ord(a)=1, ord(b)=2, ... , ord(z)=26, e ord($)=32 (con $ che rappresenta lo spazio), con rappresentazione di ogni ordinale su 6 bit.
- Per le chiavi weber e webern si ottiene:
  - bin(weber$) = 010111 000101 00**0010 0001**01 010010 100000
  - bin(webern) = 010111 000101 00**0010 0001**01 010010 001110
- dove si sono evidenziati per chiarezza i gruppi di 6 bit che rappresentano ciascun carattere. Calcoliamo gli indirizzi hash ottenuti con le diverse funzioni hash:
  - **Funzione 1)** sia m = 256 = 2^8. Estraendo gli 8 bit dalla posizione 15 alla 22 di bin(k), h(weber) = h(webern) = int(00100001) = 33. Le due chiavi danno pertanto luogo ad una collisione

17

- **Funzione 2)** sia ancora m = 256 = 2^8 e si calcoli b raggruppando bin(k) in cinque gruppi di 8 bit (dopo aver espanso a destra bin(k) con quattro zeri).
  h(weber) = int(11000011) = 195, poiché
  - 11000011 = 01011100 + 01010000 + 10000101 + 01001010 + 00000000

  dove + indica la somma bit a bit modulo 2. Analogamente, h(webern) = int(00100001) = 33, poiché
  - 00100001 = 01011100 + 01010000 + 10000101 + 01001000 + 11100000

  In questo caso, la collisione non é presente.
- **Funzione 3)** sia m = 383. Utilizzando 6 bit, int(bin(k)) è un numero in base $2^6 = 64$.

  int(bin(webern))= $23 * 64^5 + 5 * 64^4 + 2 * 64^3 + 5 * 64^2 + 18 * 64^1 + 14 * 64^0$=
  64(64(64(64(23*64)+5)+2)+5)+18)+14

  h(webern) e h(weber) sono ottenuti prendendo il resto della divisione int(bin(.....))/383

18

Scansione Interna per la risoluzione della collisioni con l'hash chiuso

Come abbiamo precedentemente visto nell'hash chiuso tutti gli elementi sono presenti in unico vettore principale e se una posizione è occupata bisogna cercarne un'altra secondo una sequenza deterministica f_i. Le strategie di scansione per determinare questa sequenza sono:

- **Scansione Lineare**: le celle vengono controllate sequenzialmente:
  $f_i = (h(k) + h \cdot i)(mod\, m)$, dove $h = 1$ principalmente.
  Il **problema** di questa strategia è che se si crea un blocco di celle piene, qualsiasi chiave che impatta in quel blocco deve percorrerlo tutto, allungando il blocco stesso e peggiorando le prestazioni future, avviene la generazione di **agglomerati primari**.
- **Scansione Quadratica**: la distanza tra i tentativi cresce quadraticamente:
  $f_i = (h(k) + h \cdot i^2)(mod\, m)$.
  Qui il **vantaggio** avviene poiché si riduce la formazione di agglomerati poiché le chiavi che coincidono sulla stessa cella *saltano* via con passi sempre più lunghi.
  Lo **svantaggio** come si può intuire è che non visualizzerà mai tutte le celle del vettore se si sposta sempre di passi in più sempre più lunghi, a meno che non si rispettino certe condizioni con m e che sia un numero primo.
- **Scansione Pseudocasuale**: si usa un generatore di numeri casuali, dato da un `seed`, per determinare la sequenza di celle da visitare: $f_i = (h(k) + r^i)(mod\, m)$.
- **Hashing Doppio**: qui infine, si utilizza una seconda funzione $h'(k)$ per poter calcolare il passo di spostamento: $f_i = (h(k) + i \cdot h'(k))(mod\, m)$. Il suo **vantaggio** sta nella sua alta efficienza per la dispersione delle chiavi.

La scansione interna soffre gravemente l'operazione di cancellazione, questo perché se si cancella fisicamente un elemento, rendendo la cella **libera**, si interromperà la catena di ricerca per gli elementi che erano già inseriti successivamente ad una collisione con l'elemento appena rimosso.
La soluzione studiata prende il nome di **Tampone**, si utilizza una chiave fittizia chiamata **cancellato**, distinta da **libero**, così nella fase di **ricerca** il *cancellato* permette di proseguire e nella fase di **inserimento** il *cancellato* può essere sovrascritto.
La conseguenza data da questa chiave fittizia è la **degradazione del tempo di ricerca**. Nel caso la tabella fosse sottoposta a molte cancellazioni, la C.C. diventa lineare ($O(m)$), rendendo necessario ristrutturare totalmente la tabella.

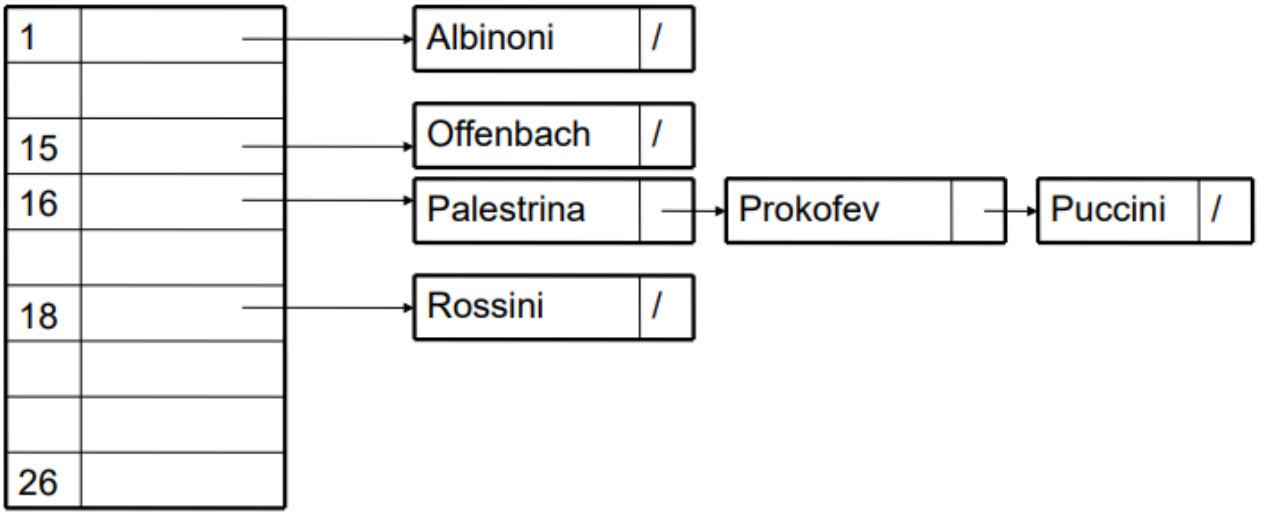
Liste di Trabocco e Hash aperto

Rispetto agli esempi precedenti il più utile (e richiesto all'esame) è il trabocco.
Le liste di trabocco sono la struttura fondamentale utilizzata nella tecnica dell'hash aperto. A differenza di quello chiuso, dove è presente un vettore principale con dimensione fissa, quello aperto permette una gestione di più insiemi di valori di qualsiasi dimensione in uno spazio quasi illimitato.

La tabella hash non memorizza più i dati nelle celle di un vettore, ma la tabella stessa viene trasformata in un array di liste di bucket.

Ogni posizione del vettore contiene un puntatore alla testa di una lista collegata, questa lista contiene tutti gli elementi chiavi, definite tramite la funzione hash, in modo da poter attivare una successiva operazione di ricerca e da restituire la posizione del bucket che contiene la chiave.

Il vettore v contiene in ogni posizione un puntatore ad una lista



Nell'esempio si nota che la cella 16 contiene il puntatore alla testa della lista in cui sono memorizzati tutti gli elementi che, tramite hash, hanno restituito il valore 16.
Altre celle, come la 1 (Albinoni) o la 15 (Offenbach), hanno liste che contengono un solo elemento perché non ci sono state collisioni per quelle lettere.

L'uso delle liste di trabocco risolve due grandi problemi della scansione interna, evita gli **agglomerati** e risolve il problema legato alle **cancellazioni**.

Discorso sulla complessità

Analizziamo tutte le strutture argomentate in precedenza, confrontandole tra loro e capiamo perché una è vantaggiosa rispetto ad altre.

Il **vettore ordinato** rispetto ad un insieme generico ha la possibilità di eseguire una ricerca **binaria** su di esso, grazie all'ordinamento delle chiavi contigue, questo ci fornisce una C.C. di **ricerca** pari a $O(\log\ n)$, ovvero al raddoppiare dei dati, il tempo cresce di una sola unità. Tuttavia, in un vettore compatto la cancellazione e l'inserimento richiedono lo spostamento fisico degli elementi successivi per mantenere un determinato ordine: questo porta la complessità delle modifiche a $O(n)$, rendendole molto più **costose** rispetto alla tabella hash, la quale tende a $O(1)$.

Perché l'hash

La tabella hash nasce dalla fortissima necessità di ottenere operazioni di ricerca e modifica in tempo costanti.

L'obiettivo madre è che il tempo di accesso sia indipendente alla dimensione del dizionario e dallo spazio delle chiavi. Si calcola la funzione interessata e si accede direttamente.
Per garantire una C.C. pari a $O(1)$ senza collisioni, la dimensione del vettore dovrebbe essere uguale a tutte le chiavi possibili. Se il massimo valore delle chiavi possibili è grande, si avrà sicuramente uno spreco di memoria esorbitante; la risoluzione sarebbe quella di accettare un **compromesso**, si accetta $m < |K|$ ma con il rischio di collisioni, che ovviamente avranno un impatto sulla C.C.

Hash chiuso

Nell'hash chiuso la C.C. dipende dal fattore di riempimento e dalla gestione delle collisioni. Finché la tabella è piena e si limita alla ricerca e all'inserimento regolare, la sua C.C. si avvicina quasi a $O(1)$, il problema nasce con gli **agglomerati**.
Utilizzando la scansione lineare, si formano gruppi di celle occupate consecutive.
Questo degrada la complessità, poiché per inserire o trovare una chiave che finisce in un agglomerato, bisogna scorrere tutta la tabella hash. La **scansione quadratica** migliora questo aspetto ma non lo risolve del tutto.

Il **caso pessimo** avviene quando per non poter lasciare *buchi* vuoti durante la **cancellazione** (se no si presenterebbe il problema trattato precedentemente per gli elementi successivi al buco), si usano i marcatori logici come *cancellato* e continuare a scansionare le celle vuote marcate. La ricerca a livello computazionale si comporterà come se quelle celle non sono effettivamente vuote e quindi impiegherà un tempo di ricerca **maggiore** e un aumento di C.C.

Immaginando una tabella dove hai inserito 1000 elementi e ne hai poi cancellati 990. La tabella è quasi vuota di dati reali, ma è piena di marcatori *cancellato*.
Se cerchi un dato, l'algoritmo deve scavalcare tutti i 990 marcatori, uno per uno prima di capire se il dato esiste o meno.

In sintesi, "diventa lento quanto una scansione sequenziale ($O(n)$)", cioè, invece di saltare direttamente al punto giusto, il computer è costretto a camminare passo dopo passo attraverso tutte le celle, rendendo l'uso della tabella Hash **inutile** (tanto valeva usare un semplice array **non ordinato**).

Hash aperto

L'hash aperto viene presentato da molti come la soluzione a tutti i problemi computazionali precedenti, poiché:

- Per lo **spazio**: consente di gestire le dimensioni in modo illimitato in uno spazio fisso. La **C.C. spaziale è data dal vettore più la memoria dinamica** per i nodi delle liste, nel momento della creazione del nodo in più.
- Per il **tempo**: l'accesso alla cella è istantaneo avendo una C.C. di $O(1)$. Il costo aggiuntivo è dato dai bucket presenti. Nell'hash aperto le collisioni si verificano in modo verticale e non orizzontalmente e non creano **agglomerati** che rallentano tutto il

sistema, dato che se le liste vengono costruite in modo intelligente, la lunghezza media di essa sarà molto **piccola** e contenuta.

Esempio:
Nell'Hash Chiuso lo scaffale 42 è pieno. Il libro deve vagare nello scaffale 43, poi 44, finché non trova un buco vuoto. *Crea confusione negli scaffali vicini.*

Nell'Hash Aperto Lo scaffale 42 è pieno. Il libro **rimane allo scaffale 42**, ma viene "appeso" in una catena sotto quello scaffale, la lista di trabocco. Non tocca assolutamente lo scaffale 43, che rimane libero per i suoi libri legittimi.

Anche in caso di cancellazione, a differenza dell'hash chiuso, qui la **cancellazione** sarà effettiva e concreta, si sgancia il nodo dalla lista e non **degrada** ne rallenta le ricerche future; concludendo, non si avrà più una C.C. lineare causata dai residui delle cancellazioni **fantasma**.

Struttura	Ricerca	Modifica	Note sulla complessià
Vettore ordinato	Logaritmica $O(\log\ n)$	Lineare	Efficiente in lettura, lento in scrittura
Hash Chiuso	Costante tendente a Lineare	Costante tendente a Lineare	Degrado verso $0(n)$ se ci sono agglomerati o molte cancellazioni.
Hash Aperto	Costante $O(1)$	Costante $O(1)$	La struttura più stabile. Il tempo dipende solo dalla lunghezza della singola lista di trabocco, non dalla pienezza della tabella.