Dizionari

Dizionario = collezione di elementi

Ricerca — Inserimento — Cancellazione

Ciascun elemento e contiene una chiave e dei dati

e.chiave → chiave di ricerca

e.sat → dati satellite

Dipendono dalle applicazioni

Sia $\mathcal{U}$ il dominio o universo delle chiavi di ricerca.

Un dizionario memorizza un insieme $S = \{e_0, e_1, \dots, e_{n-1}\}$ di n elementi, dove n è la dimensione di S.

Fornisce le seguenti operazioni, per ogni chiave $K \in \mathcal{U}$

- Ricerca (K): restituisce l'elemento e se K = e.chiave, oppure NULL se nessun $e \in S$ ha chiave K.
- Inserisci (e): supponendo che e.chiave sia distinta da tutte le altre chiavi presenti nel dizionario, estende $S = S \cup \{e\}$.

  Se e è già presente nel dizionario l'insieme non cambia (e.sat vengono aggiornati)

- Cancella (K): elimina dal dizionario l'elemento e tale che K = e.chiave, ponendo $S = S - \{e\}$.
  Se nessun elemento ha chiave K, S non cambia.

Nelle applicazioni in cui i dati satellite sono assenti e il dizionario memorizza soltanto le chiavi (distinte), l'operazione di ricerca diventa una operazione di appartenenza all'insieme:

- Appartiene (K): restituisce TRUE se e solo se K appartiene all'insieme S (cioè Ricerca (K) ≠ NULL).

Un dizionario che fornisce solo l'operazione di Ricerca è detto STATICO. Se fornisce anche Inserimento e Cancellazione è detto DINAMICO.

Quando sull'universo delle chiavi U è definita una relazione d'ordine totale (cioè, per ogni coppia $K, K' \in U$ risulta $K < K'$ o $K' < K$), il dizionario è detto <u>ordinato</u>.

Scriveremo $e_0 < e_1 < e_2 < \ldots < e_{n-1}$ per dire che gli elementi di S, in base ai valori di chiave, soddisfano la relazione.

Un dizionario ordinato fornisce le seguenti ulteriori operazioni:

- Successore (K): restituisce l'elemento e_i tale che i è il minimo intero per cui $K \leq e_i$ e $0 \leq i < n$, oppure NULL se K è maggiore di tutte le chiavi in S.

- Predecessore (K): restituisce l'elemento e_i tale che i è il massimo intero per cui $e_i \leq K$ e $0 \leq i < n$, oppure NULL se K è minore di tutte le chiavi in S.

- Intervallo (K, K'): restituisce tutti gli elementi $e \in S$ tali che $K \leq e \leq K'$, dove assumiamo $K \leq K'$, oppure NULL se non ci sono elementi in S in tale intervallo.

- Rango (K): restituisce l'intero r che rappresenta il numero di chiavi in S che sono minori oppure uguali a K, dove $0 \leq r \leq n$.

Osservazioni: Successore (K) e Predecessore (K) restituiscono lo stesso valore di Ricerca (K) quando esiste $e \in S$ tale che e.chiave = K.

Intervallo (K, K') restituisce lo stesso valore di Ricerca (k) quando $K = K'$.

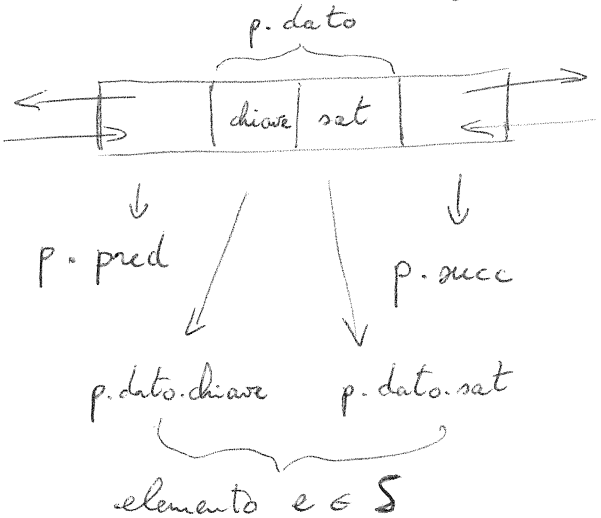
Quando sono presenti più dizionari in uno pseudo-codice, indicheremo esplicitamente il dizionario su cui effettuiamo l'operazione usando la notazione S. Ricerca (K), per esempio.

Implementazioni : con lista doppia

L lista doppia. Ha tre campi:

L. inizio	L. fine	L. lunghezza
↓	↓	↓
punta all'inizio della lista.	punta alla fine della lista.	tiene conto del numero di nodi.
↓	↓	↓
INIZIALIZZATI A NULL	NULL	0

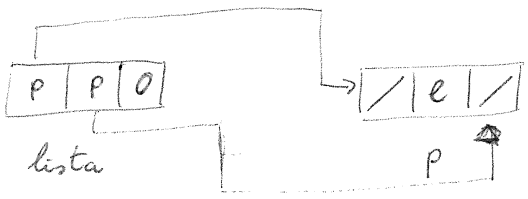
Ogni nodo ha la seguente struttura

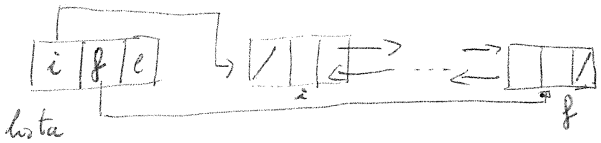


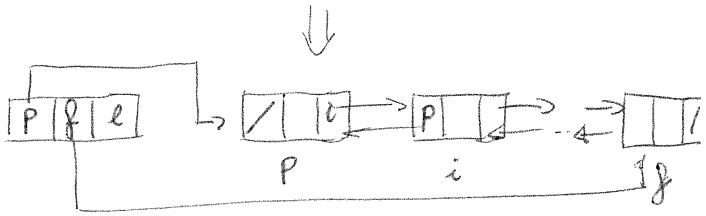
La funzione NuovoNodo crea un nuovo nodo

Per l'inserimento, implementiamo due funzioni: una inserisce in cima, l'altra in fondo.

```
InserisciCima (e):
p = NuovoNodo();                    /* alloca il nuovo nodo
p.dato = e;                         /* inserisce e
lun = lista.lunghezza;              /* memorizza in lun la lunghezza
                                       per distinguere i casi
IF (lun == 0) {                     /* dizionario vuoto
    p.succ = p.pred = NULL;
    lista.inizio = p;
    lista.fine = p;
} ELSE {                            /* dizionario non vuoto
    p.succ = lista.inizio;
    p.pred = NULL;
    lista.inizio.pred = p;
    lista.inizio = p;
}
lista.lunghezza = lun + 1;          /* aggiorna la lunghezza della lista
RETURN lista;
```

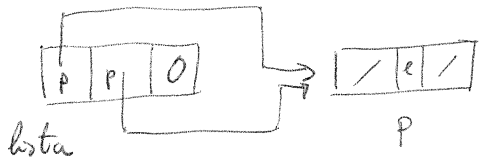


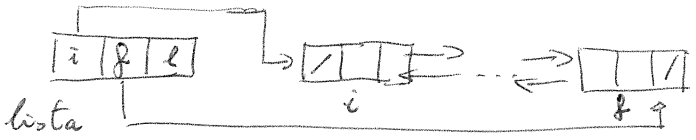


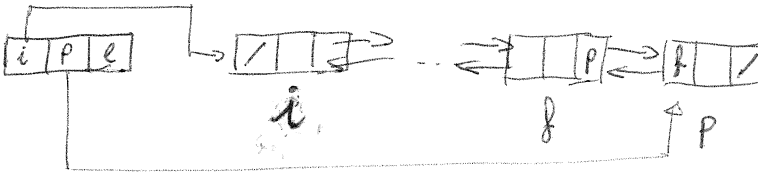


Il costo dell'operazione è $O(1)$.

```
InserisciFondo (e):
  p = NuovoNodo ();          /* alloca nuovo nodo
  p.dato = e;                /* inserisce e
  lun = lista.lunghezza;     /* usa variabile d'appoggio per
                                la lunghezza
  IF (lun == 0){             /* dizionario vuoto
    p.succ = p.pred = NULL;
    lista.inizio = p;
    lista.fine = p;
  } ELSE {                   /* dizionario non vuoto
    p.succ = NULL;
    p.pred = lista.fine;
    lista.fine.succ = p;
    lista.fine = p;
  }
  lista.lunghezza = lun + 1; /* aggiorna la lunghezza della lista
  RETURN lista;
```







Il costo dell'operazione è $O(1)$.

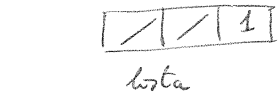
```
Ricerca (K):
  p = lista.inizio;
  WHILE ((p != NULL) && (p.dato.chiave != K))
    p = p.succ;
  RETURN p;
```

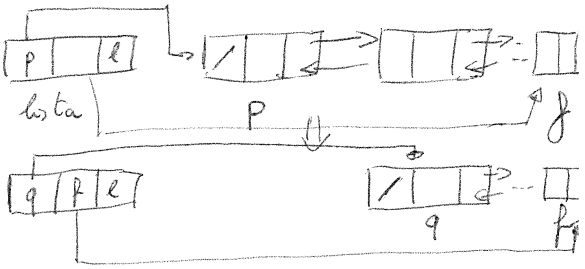
/* scorre la lista e restituisce o il nodo che contiene K o NULL quando giunge a fine lista senza successo

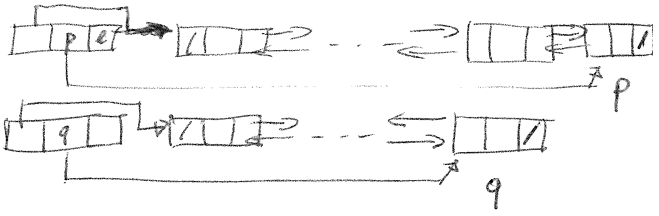
Il costo di Ricerca(K) è $O(n)$.

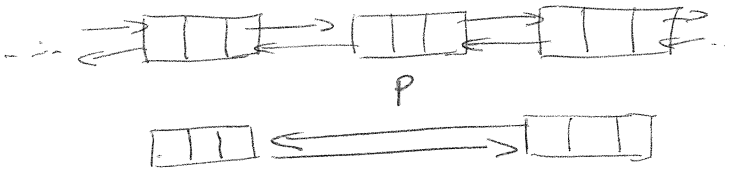
Cancella(K):

```
p = Ricerca(K);                          /* individua il nodo con chiave K
IF (p != NULL){
  IF (lista.lunghezza == 1){             /* ... x c'è ...
    lista.inizio = lista.fine = NULL;    /* x è l'unico
  } ELSE IF (p.pred == NULL){            /* x è il primo ...
    p.succ.pred = NULL;
    lista.inizio = p.succ;
  } ELSE IF (p.succ == NULL){            /* x è l'ultimo ...
    p.pred.succ = NULL;
    lista.fine = p.pred;
  } ELSE {                               /* x sta al centro
    p.succ.pred = p.pred;
    p.pred.succ = p.succ;
  }
  lista.lunghezza = lista.lunghezza - 1;
}
RETURN lista;
```











Il costo di Cancellazione è $O(n)$.

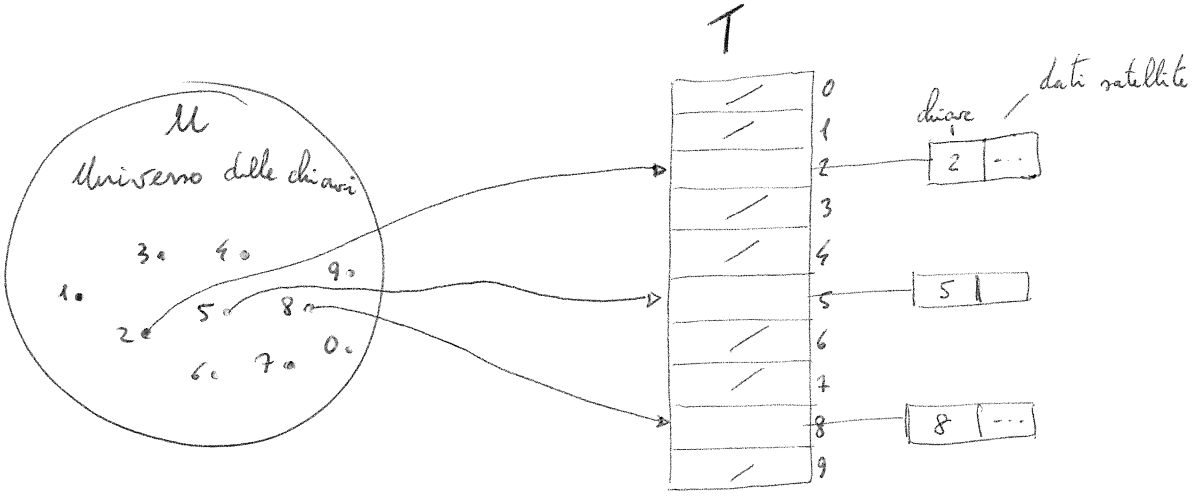
Tabelle Hash: Motivazioni

Le operazioni di inserimento e cancellazione hanno costo $O(n)$ nell'implementazione di un dizionario attraverso liste doppie.

Vogliamo fare meglio...

Se l'universo U delle chiavi è ragionevolmente piccolo possiamo usare una tavola ad indirizzamento diretto.

Precisamente, se $U = \{0, 1, \dots, m-1\}$, possiamo usare un array o tavola ad indirizzamento diretto dove ogni cella corrisponde ad una chiave dell'universo



Ricerca, Inserimento e Cancellazione sono immediate

Ricerca (k)	Inserimento (e)	Cancellazione (k)
RETURN T(k)	T[e.chiave] = e	T[k] = NULL;

Costo $O(1)$.

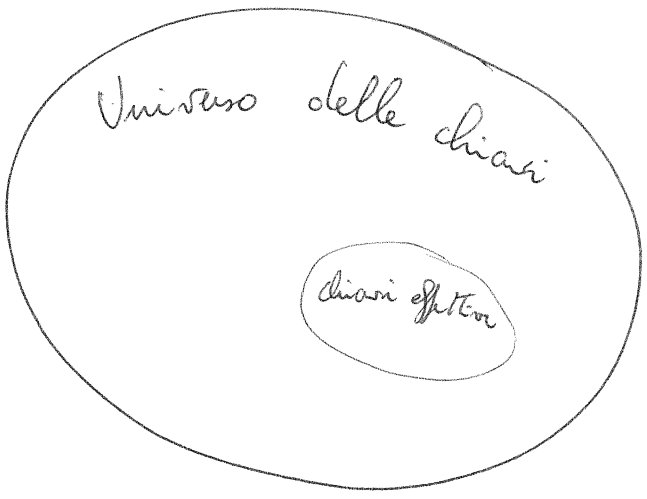
Nell'esempio le chiavi sono numeri. Se fossero elementi di un altro insieme potremmo sempre usare una funzione che ad ogni chiave associa un indice e che risulti one-to-one.

Tabella ad accesso diretto è una generalizzazione dell'array.

Se invece U è molto grande l'approccio non è ~~pra~~ praticabile. In particolare, se le chiavi di U effettivamente usate fossero soltanto una piccola parte di U, butteremmo moltissimo spazio di memoria allocato inutilmente.

Cosa possiamo fare allora quando

- U è un universo grande
- la quantità di chiavi effettive è ragionevolmente piccola ?



Le tabelle hash (hash map) risolvono il problema e permettono di ottenere le operazioni di inserimento e cancellazione con costo nel caso medio $O(1)$.

Hash = in Inglese indica un polpettone ottenuto tritando carne e verdura, di volume ridotto, i cui ingredienti sono mescolati e amalgamati.

Funzione Hash = $h : U \to [0, \dots, m-1]$

(dove $m << |U|$)

h associa ad ogni chiave di U un indice in $[0, \dots, m-1]$

Non preserva necessariamente l'ordine delle chiavi in U.

Diverse funzioni hash utilizzano la codifica binaria delle chiavi. Alcuni esempi:

a) $h(K) = K \,\%\, m$ { calcola il resto della divisione per m, dove m è un numero primo.

b) $h(K) = K_0 \oplus K_1 \oplus K_2 \oplus \dots \oplus K_{S-1}$

dove $\oplus$ è l'operatore di or esclusivo e $K_0 K_1 K_2 \dots K_{S-1} = K$

↓

"sottostringa di bit di K"

b) è una funzione iterativa

(ripete l'operazione di $\oplus$ più volte, a partire dai primi due sottoblocchi, K_0 e K_1, fino a considerare l'ultimo blocco, K_{S-1}.

Nota. Attenzione. Il termine hash viene usato anche per indicare le "funzioni hash crittografiche", quali MD4, MD5 e la famiglia SHA (SHA-1, SHA-2, SHA-3).

Queste funzioni associano a stringhe di bit di qualsiasi lunghezza, stringhe finite di pochi bits, tipo

MD5 (File) = 128 bit SHA-1 (File) = 160 bit

ma soddisfano proprietà di sicurezza aggiuntive e richiedono un costo computazionale non banale.

Il libro di testo illustra alcuni usi nelle applicazioni di queste funzioni. Leggeteli. Tuttavia, le funzioni hash di nostro interesse in questo contesto sono più semplici, non soddisfano requisiti di sicurezza, e sono più veloci.

Precisamente, ci interessano funzioni che:

- "distribuiscono" uniformemente le chiavi in U sul codominio $[0, \ldots, m-1]$

Se ciò accade le "collisioni", chiavi $k_1 \neq k_2$ tali che $h(k_1) = h(k_2)$, sono ridotte.

Infatti, essendo $|U| >> m$, esistono sempre due chiavi K_1 e K_2 in U tali che $h(K_1) = h(K_2)$. Occorre, pertanto, capire come gestire le collisioni.

Per ogni chiave $u \in U$, indicheremo con $h(u)$ il valore hash della chiave u. Vedremo nel dettaglio due metodi per far fronte al problema delle collisioni:

- Chaining (liste di trabocco)
- Open addressing (indirizzamento aperto)