Array, liste e tabelle hash

April 8, 2019

Obiettivi: capire in che modo la scelta delle strutture dati per rappresentare **insiemi dinamici** influenzino il tempo di accesso ai dati.
Argomenti: array (statico e ridimensionabile), liste, hash, costo ammortizzato.

1 Insiemi dinamici (dizionari)

Studiamo strutture per rappresentare **insiemi dinamici**:
- numero finito di elementi
- gli elementi possono cambiare
- il numero di elementi può cambiare
- si assume che ogni elemento ha un attributo che serve da chiave
- le chiavi sono tutte diverse

Esistono due **tipi di operazioni**:
- interrogazione (query)
- modifiche

Operazione tipiche:
- inserimento (insert)
- ricerca (search)
- cancellazione (delete)

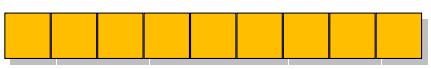
Operazione tipiche in caso di chiavi estratte da **insiemi totalmente ordinati:**
- ricerca del minimo (minimum)
- ricerca del massimo (maximum)
- ricerca del prossimo elemento più grande (successor)
- ricerca del prossimo elemento più piccolo (predecessor)

La complessità delle operazioni
- è misurata in funzione della dimensione dell'insieme,
- **dipende da che tipo di struttura dati si utilizza per rappresentare l'insieme dinamico**

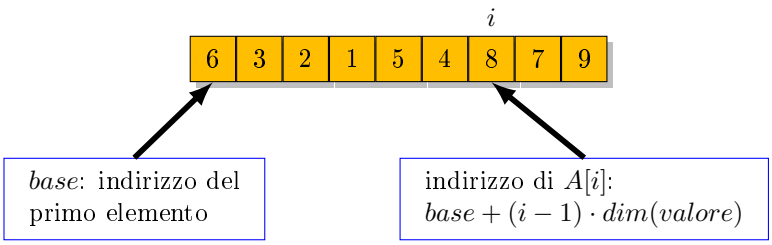
Un operazione che è costosa con una certa struttura dati può costare poco con un'altra. Quali operazioni sono necessarie dipende dall'applicazione.

2 Array

Un **array** è una sequenza di caselle:
- ogni caselle può contenere un elemento dell'insieme
- le caselle sono grandi uguali e sono posizionati in una sequenza nella memoria:



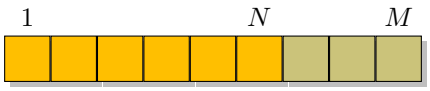
- il **calcolo dell'indirizzo di qualunque casella ha costo costante** (non dipende dal numero di elementi)



- e quindi accedere ad un elemento qualunque ha costo costante
- con l'accesso diretto il tempo per leggere/scrivere in una cella è $O(1)$

Un **array statico** è un array in cui il **numero massimo di elementi è prefissato**:

- M denota il numero massimo di elementi
- N denota il numero attuale di elementi
- gli N elementi occupano sempre le prime N celle del array



Ci interessa studiare

- **quanto costano le varie operazioni**
- quando conviene utilizzare questo tipo di array

Inserimento dell'elemento k nel array A:

```
ArrayInsert(A, k)
if A.N ≠ A.M then
    A.N ← A.N + 1
    A[N] ← k
    return k
else
    return nil
end if
```

Quanto costa un inserimento?

- $O(1)$ (costo costante)
- non dipende ne da N ne da M

Possono esserci delle **ripetizioni** se la stessa chiave viene inserita più volte.

Rimozione dell'elemento k dal array A:

```
ArrayDelete(A, k)
for i ← 1 to A.N do
    if A[i] == k then
        A.N ← A.N − 1
        for j ← i to A.N do
            A[j] ← A[j + 1]
        end for
        return k
    end if
end for
return nil
```

Quanto costa rimuovere un elemento?

- $O(N)$ (costo lineare)
- non dipende da M

Abbiamo assunto che non ci sono ripetizioni. Se conoscessi la posizione? Comunque rimane $O(N)$ perché bisogna spostare elementi.

Ricerca dell'elemento k nel array A:

```
ArraySearch(A, k)
for i ← 1 to A.N do
    if A[i] == k then
        return k
    end if
end for
return nil
```

Quanto costa fare una ricerca? $O(N)$ (costo lineare)

Riassumendo:

- inserimento: $O(1)$ (se non si fa controllo se il dato ci sia già)
- cancellazione: $O(N)$
- ricerca: $O(N)$

E se l'array fosse ordinato?

Eseguire inserimenti tenendo l'array ordinato costa di più. Come sarebbe l'algoritmo ArrayInsertOrd che mantiene l'array ordinato?

1. Si inserisce l'elemento in fondo (se c'è spazio).
2. Si fa scendere l'elemento nella posizione giusta facendo scambi (come fa l'insertion-sort).

Che complessità ha l'algoritmo ArrayInsertOrd? Tempo $O(N)$.

Ulteriori domande:

- come si fa e quanto costa cercare il minimo e il massimo in un array ordinato?
- come si fa e quanto costa cercare il minimo e il massimo in un array non ordinato?
- come si realizzano e che complessità hanno le operazioni successor e predecessor in array ordinati e non ordinati?

2.1 Array ridimensionabile

Cosa si può fare se non si conosce il numero massimo di elementi a priori (oppure se non si vuole sprecare spazio allocando molto più memoria del necessario)? Si può **espandere l'array quando esso diventa troppo piccolo**. Ma **espandere costa tempo** $O(N)$ perché richiede di allocare memoria e copiare gli elementi dell'array:

```
ArrayExtend(A, n)
B ← un array con A.M + n elementi
B.M ← A.M + n
B.N ← A.N
for i ← 1 to A.N do
    B[i] ← A[i]
end for
return B
```

Prima idea:

- allochiamo inizialmente spazio per M elementi (array di lunghezza M)
- quando viene aggiunto un elemento, se l'array è pieno, espandiamo l'array di una cella:

```
DynArrayInsert1(A, k)
if A.N == A.M then
    A ← ArrayExtend(A, 1)
end if
ArrayInsert(A, k)
```

Con questa prima idea:

- quanto costa un inserimento?
- se l'array **non è pieno** il costo è $O(1)$
- se l'array è **pieno** il costo è $O(N)$ perché espandere ha un costo lineare in N
- quindi il **costo** dell'inserimento **dipende dallo stato** dell'array e **quindi dalle operazioni precedenti**

Sempre con la prima idea:

- quanto costano gli inserimenti a lungo andare?
- se M è sufficientemente grande e si sfora poche volte allora il costo di un inserimento è circa $O(1)$ (ma si rischia di sprecare spazio)
- se M è tale che si sfora le maggior parte delle volte allora il costo di un inserimento è circa $O(N)$
- il costo dipende da M e dalle operazioni effettuate
- si può fare meglio?

Seconda idea:

- problema della prima idea: se $N = M$ allora i successivi inserimenti richiedono successivi riallocazioni
- l'idea per evitare questo: se $N = M$ e viene richiesto un inserimento allora **allochiamo spazio per tanti elementi non solo uno**

Seconda idea, in concreto:

- allochiamo inizialmente spazio per M elemento
- quando l'array è pieno raddoppiamo la dimensione potenziale dell'array
- per non sprecare spazio, quando il numero di elementi si riduce ad $1/4$ della dimensione, dimezziamo la dimensione dell'array

Inserimento:

DynArrayInsert2(A, k)
if $A.N == A.M$ **then**
  $A \leftarrow$ ArrayExtend$(A, A.M)$
end if
ArrayInsert(A, k)

Raddoppia il numero di elementi se A è pieno. Si fa **un'investimento pagato in spazio per un guadagno futuro in tempo**.

Cancellazione:

DynArrayDelete2(A, k)
ArrayDelete(A, k)
if $A.N \leq 1/4 \cdot A.M$ **then**
  $B \leftarrow$ un array di dimensione $A.M/2$ ▷ Qui si recupera spazio.
  $B.M \leftarrow A.M/2$
  $B.N \leftarrow A.N$
  for $i \leftarrow 1$ to $A.N$ **do**
    $B[i] \leftarrow A[i]$
  end for
  $A \leftarrow B$
end if

La prima e la seconda idea sono due soluzioni diversi per la **realizzazione di un ADT** (abstract data type). Per confrontarle valutiamo i **tempi di una sequenza di operazioni**. Confrontare i tempi di una singola operazione non avrebbe senso perché essi dipendono dallo stato della struttura dati.

Confrontiamo la prima e la seconda idea per una lunga seria di 2^K inserimenti con $M = 1$ inizialmente. Con la prima idea: ogni inserimento tranne il primo ha costo $O(N)$. Con la seconda idea: ci sono K inserimenti che hanno costo $O(N)$ e gli altri hanno costo $O(1)$. Come si confrontano queste due situazioni?

Costo ammortizzato: quando il costo delle operazione consecutivi hanno costi diversi, conviene considerare **quanto costa un'operazione in media in una sequenza di operazioni**:

$$T_{ammortizzato} = \frac{T_1 + T_2 + ... + T_L}{L}$$

dove T_i è il costo della i-esima operazione e L è il numero di operazioni.

Complessità ammortizzata di un inserimento con la **prima idea** in una lunga seria di $n = 2^K$ inserimenti con $N = 0, M = 1$ inizialmente:

$$T_{amm} = \frac{nd + c + 2c + 3c \cdots + (n-1)c}{n} \in O(n)$$

dove si effettuano n inserimenti "semplici" (con tempo associato uguale a nd) e $n-1$ estensioni (con tempo associato uguale a $c + 2c + ... + (n-1)c$). Quindi, in funzione della dimensione finale del vettore, visto che $N = n$, la a complessità ammortizzata è $O(N)$.

La **complessità ammortizzata** di un inserimento con la **seconda idea** in una lunga seria di 2^K inserimenti con $N = 0, M = 1$ inizialmente può essere scritto come

$$T_{amm} = \frac{\left(c + 2c + 4c + 8c + \cdots + 2^{K-1}c\right) + 2^K d}{2^K} = \frac{(2^K - 1)c + 2^K d}{2^K} \in O(1)$$

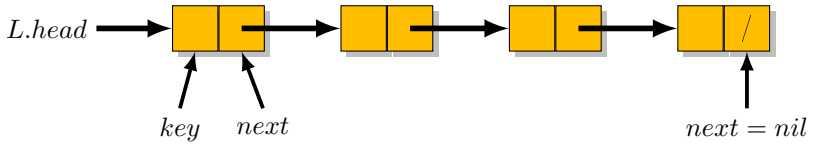
perché si effettua K volta l'estensione (che costa in totale $c + 2c + 4c + 8c + \cdots + 2^{K-1}c$) e 2^K inserimenti semplici (che costano $2^K d$). E quindi la complessità ammortizzata in questo caso è $O(1)$

Ulteriori domande:

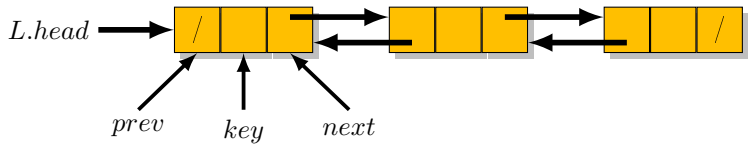
- quanto costa rimuovere gli elementi con la seconda idea?
  - consideriamo una seria di DELETE che rimuove sempre l'ultimo elemento
  - consideriamo una seria di DELETE che rimuove un elemento qualunque
- quanto costa (in senso ammortizzato) un inserimento se espandiamo l'array di un numero costante di elementi invece di raddoppiare?

3 Liste

Una **lista** è una struttura dati lineare in cui **l'ordine degli elementi è determinato dai puntatori** che indicano l'elemento successivo. Data una lista L il primo elemento è indicato dal puntatore $L.head$.

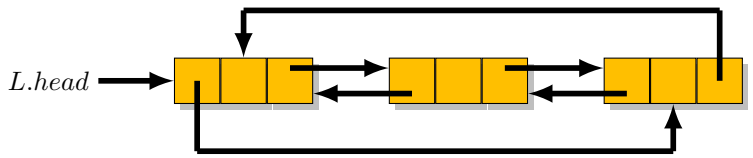


La lista può essere **doppiamente concatenata**:



La lista può essere **ordinata** (gli elementi in ordine secondo la chiave) o **non ordinata**.

La lista può essere **circolare**:



La lista circolare può essere vista come un anello di elementi.

Operazioni di base in una lista doppiamente concatenate e non ordinata:

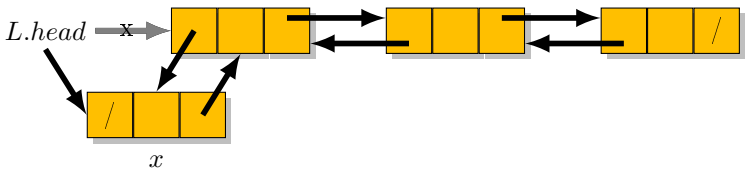
```
LISTSEARCH(L, k)        ▷ Ricerca della chiave k.
x ← L.head
while x ≠ nil and x.key ≠ k do
    x ← x.next
end while
return x
```

Complessità: $O(N)$.

```
LISTINSERT(L, x)        ▷ Inserimento "in testa".
x.next ← L.head
if L.head ≠ nil then
    L.head.prev ← x
end if
L.head ← x
x.prev ← nil
```

Complessità: $O(1)$.

Inserimento in testa graficamente:



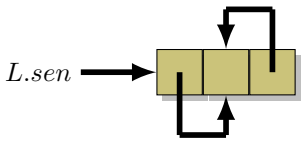

```
LISTDELETE(L, x)        ▷ Rimozione dell'elemento puntato da x (quindi non bisogna cercare l'elemento).
if x.prev ≠ nil then
    x.prev.next ← x.next
else
    L.head ← x.next
end if
if x.next ≠ nil then
    x.next.prev ← x.prev
end if
```

Complessità: $O(1)$.

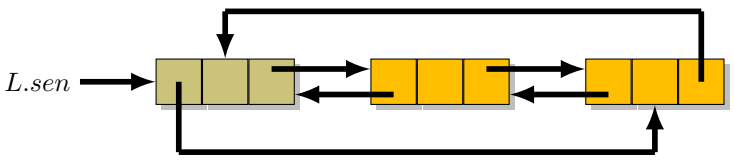
LISTDELETE è macchinoso perché deve controllare le condizioni "in testa" e "in coda" della lista. Aggiungiamo una **sentinella** che c'è sempre:

- un oggetto fittizio che non contiene dati
- serve a rendere più omogenei gli elementi della lista

Lista circolare vuota (solo sentinella):



Lista circolare non vuota:



Operazioni di base su liste doppiamente concatenate e non ordinate **con sentinella**:

ListDeleteSen(L, x) ▷ **Rimozione dell'elemento puntato da** x (quindi non bisogna cercare l'elemento).
$x.prev.next \leftarrow x.next$
$x.next.prev \leftarrow x.prev$

La complessità rimane $O(1)$ ma il codice è più semplice e leggibile. Si risparmia un tempo $O(1)$

ListSearchSen(L, k) ▷ **Ricerca della chiave** k**.**
$x \leftarrow L.sen.next$
while $x \neq L.sen$ **and** $x.key \neq k$ **do**
  $x \leftarrow x.next$
end while
return x

Complessità: $O(N)$.

ListInsertSen(L, x) ▷ **Inserimento "in testa".**
$x.next \leftarrow L.sen.next$
$L.sen.next.prev \leftarrow x$
$L.sen.next \leftarrow x$
$x.prev \leftarrow L.sen$

Complessità: $O(1)$.

Ulteriori domande:

- consideriamo una lista ordinata:
  - come si fa e quanto costa un inserimento?
  - come si fa e quanto costa una ricerca?
  - come si fa e quanto costa una rimozione?
- consideriamo una lista che non è doppiamente concatenata:
  - come si fa la rimozione?
  - come si fa l'inserimento?

4 Hashing

Con array e liste è facile implementare tanti tipi di operazioni ma con ognuna il costo di certi operazioni è $O(N)$.

Le **tabelle hash** forniscono solo le operazioni di base (insert, search e delete) ma ognuna con tempo medio $O(1)$.

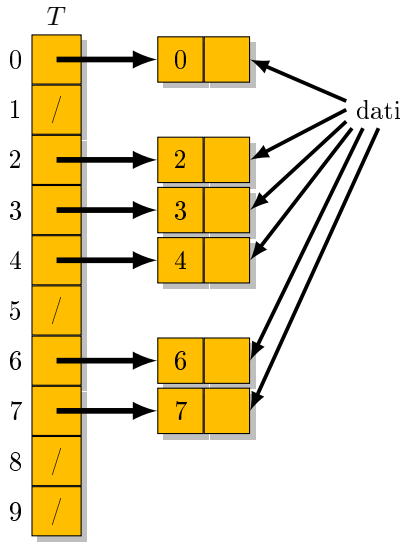
4.1 Tavole a indirizzamento diretto

La tavola a a indirizzamento diretto è un'idea preliminare a quella della tavole hash.

Sia U l'universo delle chiavi: $U = \{0, 1, \ldots, m-1\}$. L'insieme dinamico viene rappresentato con un array T di dimensione m in cui ogni posizione corrisponde ad una chiave. T è la **tavola a indirizzamento diretto** perché ogni sua cella corrisponde direttamente ad una chiave.

Esempio:

- universo delle chiavi: $U = \{0, 1, 2, ..., 9\}$
- insieme delle chiavi: $S = \{0, 2, 3, 4, 6, 7\}$



Le operazioni sono semplicissime:

TableInsert(T, x)
$T[x.key] \leftarrow x$

TableDelete(T, x)
$T[x.key] \leftarrow nil$

TableSearch(k)
return $T[k]$

Tutte le operazioni in tempo $O(1)$.

Sembra una struttura molto efficiente. Da quale punto di viste non lo è? Quanto costa la struttura in termini di spazio? Dipende dal contesto in cui viene utilizzata.

Consideriamo il seguente scenario:

- studenti identificati con matricola composta da 6 cifre: abbiamo 10^6 possibili chiavi
- T occupa $8 \cdot 10^6$ byte di memora (se un puntatore ne occupa 8)
- di ogni studente si memorizza 10^5 byte di dati (100kB)
- ci sono 20000 studenti

Lo spazio occupato ma non utilizzato in assoluto (i *nil*) è $8(10^6 - 20000)$=7840000B=7.84MB. Frazione di spazio occupato ma non utilizzato rispetto al totale:

$$\frac{7.84 \cdot 10^6}{8 \cdot 10^6 + 20000 \cdot 10^5} = 0.0039$$

cioè circa 0.4%. Quindi in questo contesto è ragionevole

Se si memorizza solo 1kB di dati per studente:

$$\frac{7.84 \cdot 10^6}{8 \cdot 10^6 + 20000 \cdot 10^3} = 0.28$$

cioè circa 28% della memoria è occupata "inutilmente". Se si memorizza solo 1kB di dati per studente e ci sono solo 200 studenti (quelli di un corso):

$$\frac{7.84 \cdot 10^6}{8 \cdot 10^6 + 200 \cdot 10^3} = 0.956$$

cioè circa 95.6% della memoria è occupata "inutilmente".

4.2 Tavole hash

L'indirizzamento diretto non è praticabile se l'universo delle chiavi è grande oppure contiene un numero infinito di chiavi. E in ogni caso non è efficiente dal punto di vista della memoria utilizzata.

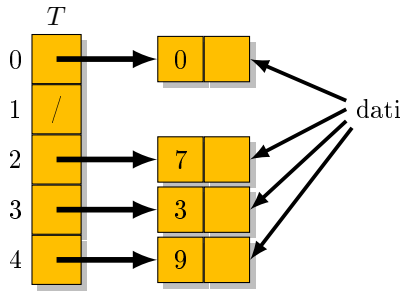
Idea: utilizziamo una tabella T di dimensione m con m molto più piccolo di $|U|$. La posizione della chiave k è determinata utilizzando una funziona

$$h : U \to \{0, 1, \ldots, m-1\}$$

chiamata la **funzione hash**.

Esempio:

- universo delle chiavi: $U = \{0, 1, 2, ..., 9\}$
- insieme delle chiavi: $S = \{0, 3, 7, 9\}$
- funzione hash: $h(k) = k \bmod 5$
- $h(k)$ è il valore hash della chiave k



L'indirizzamento non è più diretto, l'elemento con chiave k si trova nella posizione $h(k)$.

Conseguenze:

- riduciamo lo spazio utilizzato,
- perdiamo la diretta corrispondenza fra chiavi e posizioni,
- $m < |U|$ e quindi inevitabilmente **possono esserci delle collisioni**.

Nel caso dell'esempio precedente le coppie (0,5), (1,6), (2,7), (3,8) e (4,9) sono in collisione.

Una buona funzione hash

- posiziona le chiavi nelle posizioni $0, 1, \ldots, m$ in modo **apparentemente casuale e uniforme**,
- e quindi riduce al minimo il numero di collisioni.

Hash perfetto: una funzione che non crea mai collisione, cioè una **funzione iniettiva**:

$$k_1 \neq k_2 \implies h(k_1) \neq h(k_2)$$

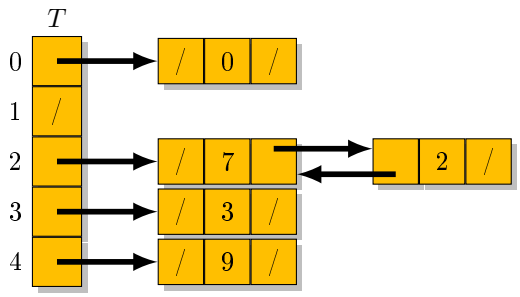
Se $|U| > m$ allora, il **hash perfetto realizzabile solo se l'insieme rappresentato non è dinamico**.

4.3 Tavole hash con concatenamento

Una possibile soluzione per risolvere le collisioni è il **concatenamento degli elementi in collisione in una lista**.

Esempio:

- universo delle chiavi: $U = \{0, 1, 2, ..., 9\}$
- insieme delle chiavi: $S = \{0, 2, 3, 7, 9\}$
- funzione hash: $h(k) = k \bmod 5$



Operazioni in caso di concatenamento:

HashInsert(T, x)
$L \leftarrow T[h(x.key)]$

ListInsert(L, x)

HashSearch(T, k)
$L \leftarrow T[h(k)]$
return ListSearch(L, k)

HashDelete(T, x)
$L \leftarrow T[h(x.key)]$
ListDelete(L, x)

Come sono i tempi di esecuzione delle operazioni?

Il valore hash di una chiave si calcola in tempo costante quindi l'inserimento si fa in tempo $O(1)$. La ricerca di un elemento con la chiave k richiede un tempo proporzionale alla lunghezza della lista $T[h(k)]$. Il costo della ricerca dipende quindi dal numero di elementi e le caratteristiche della funzione hash. La cancellazione (di un elemento già individuato) richiede $O(1)$ perché la lista e doppiamente concatenata.

Analizziamo in dettaglio quanto costa una ricerca. Notazione:

- m: numero di celle in T
- N: numero di elementi memorizzati
- $\alpha = N/m$: fattore di carico

Qual è il **caso peggiore**? Scenario:

- l'universo delle chiavi: matricole con 6 cifre
- $m = 200$
- funzione hash: $h(k) = k \bmod 200$

L'elenco di inserimento che rende pesante la ricerca: 000123,100323,123723,343123,333123,... perché tutte le chiave sono associate con la stessa cella di T. Ricerca costa nel caso peggiore $\Theta(N)$.

Qual è il **caso migliore**? Quando la lista $T[h(k)]$ è vuoto oppure contiene solo un elemento la ricerca costa $O(1)$.

Qual è il costo nel **caso medio**? Dipende dalla funzione hash. Assumiamo di avere una funzione che è facile da calcolare ($O(1)$) e gode della proprietà di uniformità semplice.

Uniformità semplice: la funzione hash **distribuisce in modo uniforme le chiavi fra le celle** (ogni cella è destinazione dello stesso numero di chiavi).

La seguente funzione hash è uniforme semplice?

$$U = \{0, 1, 2, \ldots, 99\}, m = 10, h(k) = k \mod 10$$

Cioè h restituisce l'ultima cifra della chiave. L'ultima cifra c è 0,1,2,...,8 o 9 ($c \in \{0, 1, 2, \ldots, 9\}$) e ognuno di questi numeri appare 10 volte come ultima cifra. Quindi ogni cella è destinazione di 10 chiavi la funzione hash è uniforme semplice.

La seguente funzione hash è uniforme semplice?

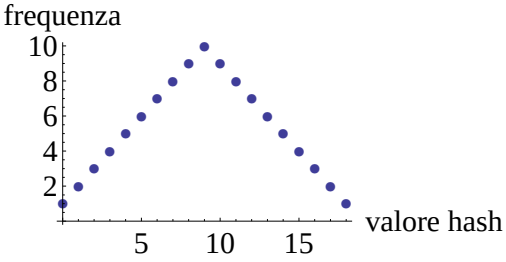
$$U = \{0, 1, 2, \ldots, 99\}, m = 19,$$

$$h(k) = \lfloor k/10 \rfloor + (k \mod 10)$$

cioè h restituisce la somma delle cifre della chiave. Quindi

- $h(k) = 0$ per $k = 0$
- $h(k) = 1$ per $k = 1$ e $k = 10$
- $h(k) = 2$ per $k = 2$ e $k = 11$ e $k = 20$
- ...

E la frequenza dei vari valori hash:



La funzione hash non è uniforme semplice.

Per analizzare il **caso medio** con hashing uniforme semplice dobbiamo capire quanti elementi ci sono in una lista in media. Sia n_i il numero di elementi nella lista $T[i]$ con $i = 0, 1, \ldots, m-1$.

Il numero medio di elementi in una lista è

$$\bar{n} = \frac{n_0 + n_1 + ... + n_{m-1}}{m} = \frac{N}{m} = \alpha$$

Ricerca di un elemento che non c'è. Il tempo di individuare la lista è $\Theta(1)$. Ogni lista ha la stessa probabilità di essere associata con la chiave (grazie all'uniformità semplice). La lista in questione ha in media α elementi e quindi percorrere la lista costa in media $\Theta(\alpha)$. Quindi il tempo richiesto è $\Theta(1)+\Theta(\alpha) = \Theta(1+\alpha)$. Attenzione: α non è costante!

Ricerca di un elemento che c'è. Il tempo di individuare la lista è sempre $\Theta(1)$. Assumiamo che la ricerca riguarda l'i-esimo elemento inserito, denotato con x_i. Per trovare x_i dobbiamo esaminare x_i stesso e tutti gli elementi che

- sono stati inseriti dopo x_i (perché si fa inserimento in testa)
- e hanno una chiave con lo stesso valore hash

Quanti elementi tali ci sono? Dopo x_i vengono inseriti $N - i$ elementi. Quanti di questi finiscono nella lista di x_i? Ogni elemento viene inserito nella lista di x_i con probabilità $\frac{1}{m}$ (uniformità semplice). Quindi **in media** $\frac{N-i}{m}$ elementi precedono x_i nella lista di x_i.

Il tempo per ricercare x_i, calcolo del valore hash a parte, è proporzionale a

$$1 + \frac{N-i}{m}$$

Quindi il tempo per ricercare un elemento scelto a caso, calcolo del valore hash a parte, è proporzionale a

$$\frac{1}{N}\sum_{i=1}^{N}\left(1 + \frac{N-i}{m}\right)$$

Elaboriando la quantità precedente:

$$\frac{1}{N}\sum_{i=1}^{N}\left(1 + \frac{N-i}{m}\right) = \frac{1}{N}\sum_{i=1}^{N}1 + \frac{1}{N}\sum_{i=1}^{N}\frac{N}{m} - \frac{1}{N}\sum_{i=1}^{N}\frac{i}{m} =$$
$$1 + \frac{N}{m} - \frac{N(N+1)}{2Nm} = 1 + \frac{N-1}{2m} =$$
$$1 + \frac{\alpha}{2} - \frac{\alpha}{2N}$$

Quindi il tempo richiesto in totale è

$$\Theta(1) + \Theta\left(1 + \frac{\alpha}{2} - \frac{\alpha}{2N}\right) = \Theta(1+\alpha)$$

Conclusione: in una tabella hash in cui le collisioni sono risolte mediante liste, nell'ipotesi di uniformità semplice, una ricerca richiede in media un tempo $\Theta(1+\alpha)$.

Cosa vuole dire in pratica $\Theta(1+\alpha)$? Se il numero di celle in T è proporzionale a N allora $N = O(m)$ e quindi $\alpha = O(1)$ e quindi la ricerca richiede tempo $O(1)$.

Quindi tutte le tre operazioni richiedono tempo $O(1)$ (se le liste sono doppiamente concatenate).

4.4 Funzioni hash

Significato della parola hash (pl. –es, n):
1. rifrittura, carne rifritta con cipolla, patate o altri vegetali
2. fiasco, pasticcio, guazzabuglio
3. (fig) rifrittume
4. (spec radio) segnali parassiti
5. nella locale slang «to settle sbs hash» mettere in riga qn, zittire o sottomettere qn, sistemare o mettere a posto qn una volta per tutte
6. anche hash sign (tipog) il simbolo tipografico

Una buona funzione hash è **uniforme semplice**. Ma questa è difficile da verificare perché di solito la distribuzione secondo la quale si estraggono le chiavi non è nota. Le chiavi vengono interpretati come numero naturali: ogni chiave è una sequenza di bit. Si cerca di utilizzare ogni bit della chiave. Una buona funzione hash sceglie posizioni in modo tale da **eliminare eventuale regolarità** nei dati.

Il **metodo della divisione** assegna alla chiave k la posizione

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

Un metodo molto veloce, bisogna scegliere m bene.

Stringhe come numeri naturali secondo il codice ASCII:

$$\text{oca} \to 111 \cdot 128^2 + 99 \cdot 128^1 + 97 \cdot 128^0$$

Posizioni di 4 parole con due diverse scelte di m:

parola	$m = 2048$	$m = 1583$
le	1637	695
variabile	1637	1261
molle	1637	217
bolle	1637	680

La scelta $m = 2^p$ è buona solo se si ha certezza che gli ultimi bit hanno distribuzione uniforme. Un numero primo non vicino a una potenza di 2 è spesso una buona scelta.

Il **metodo della moltiplicazione**: con $0 < A < 1$

$$h(k) = \lfloor m(Ak \mod 1) \rfloor$$

dove $x \mod 1$ è la parte frazionaria di x. Il valore di m non è critico, di solito si sceglie una potenza di 2. La scelta ottimale di A dipende dai dati ma $A = (\sqrt{5}-1)/2$ è un valore ragionevole. Esempio:

parola	$m = 2048$
mille	1691
polli	678
molle	242
bolle	1508

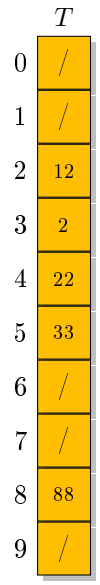
4.5 Indirizzamento aperto

Con l'**indirizzamento aperto** tutti gli elementi sono memorizzati nella tavola T. L'elemento con chiave k viene inserito nella posizione $h(k)$ se essa è libera. Se non è libera allora si cerca una posizione libera secondo

un **schema di ispezione**. La schema più semplice è l'**ispezione lineare**: a partire dalla posizione $h(k)$ l'elemento viene inserito nella prima cella libera.

Esempio:

- universo delle chiavi: $U = \{0, 1, 2, ..., 99\}$
- sequenza di inserimento: $88, 12, 2, 22, 33$
- funzione hash: $h(k) = k \bmod 10$



In generale l'indirizzamento aperto può essere descritto con una funzione hash estesa con l'ordine di ispezione:

$$h : U \times \{0, 1, 2, ..., m-1\} \to \{0, 1, 2, ..., m-1\}$$

Un elemento con la chiave k viene inserita

- nella posizione $h(k, 0)$ se questa è libera
- altrimenti nella posizione $h(k, 1)$ se questa è libera
- altrimenti nella posizione $h(k, 2)$ se questa è libera
- ...

L'ispezione è lineare se

$$h(k, i) = (h'(k) + i) \mod m$$

dove $h'(k)$ è una funzione hash "normale".

Inserimento con indirizzamento aperto:

```
HashInsert(T, x)
i ← 0
while i < m do
    j ← h(x.key, i)
    if T[j] = nil then
        T[j] ← x
        return j
    end if
    i ← i + 1
end while
return nil
```

Ricerca con indirizzamento aperto:

```
HashSearch(T, k)
i ← 0
while i < m do
    j ← h(k, i)
    if T[j] = nil then
        return nil
    end if
```

```
    if T[j].key = k then
        return T[j]
    end if
    i ← i + 1
  end while
  return nil
```

Come si fa **la cancellazione in generale** con indirizzamento aperto? Per cancellare un elemento, non possiamo semplicemente marcare la posizione in cui si trova con *nil*. Si può marcare gli elementi cancellati con *deleted*. Richiede modifiche alla procedura inserimento. Di solito l'indirizzamento aperto si usa quando non c'è necessità di cancellare.

L'ispezione lineare crea file di celle occupate, fenomeno chiamato **addensamento primario**. Questo si può evitare con l' **ispezione quadratica**:

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

Con l'ispezione lineare e l'ispezione quadratica la sequenza dipende solo dal valore di hash, questo crea **addensamento secondario**.

Per evitare addensamento primario e secondario si usa il **doppio hashing**:

$$h(k,i) = (h_1(k) + i h_2(k)) \mod m$$

Con doppio hashing la sequenza dipende dalla chiave e non soltanto dal valore hash della chiave .

Costo della ricerca:

- consideriamo il caso ottimale dal punto di vista della funzione hash e lo schema di ispezione:
  - la posizione di una chiave scelta a caso ha distribuzione uniforme,
  - qualunque sequenza di ispezione ha la stessa probabilità;
- consideriamo la ricerca di un elemento assente.

Denotiamo con X il numero di celle esaminate durante una ricerca senza successo. X è almeno 1:

$$P(X \geq 1) = 1$$

Bisogna esaminare almeno due celle se quella esaminata per primo è occupata:

$$P(X \geq 2) = \frac{N}{m}$$

Bisogna esaminare almeno tre celle se le prime due esaminate per primo sono occupate. Succede con probabilità:

$$P(X \geq 3) = \frac{N}{m}\frac{N-1}{m-1}$$

Bisogna esaminare almeno i celle con probabilità:

$$P(X \geq i) = \frac{N}{m}\frac{N-1}{m-1}\cdots\frac{N-i+2}{m-i+2} = \prod_{j=0}^{i-2}\frac{N-j}{m-j} \leq \alpha^{i-1}$$

Numero medio di celle esaminate:

$$E[X] = \sum_{i=1}^{\infty} P(X \geq i) \leq \sum_{i=1}^{\infty} \alpha^{i-1} = \frac{1}{1-\alpha}$$

Quindi il numero medio di ispezioni è minore di $1/(1-\alpha)$. Quanto fa $1/(1-\alpha)$ con certi valori di α? L'inserimento si analizza con lo stesso approccio. Ricerca con successo richiede esaminare meno celle.