Array, liste e tabelle hash

Corso di **Algoritmi e strutture dati**
Corso di Laurea in **Informatica**
Docenti: Ugo de'Liguoro, András Horváth

Indice

Sommario

- **obiettivo**: capire in che modo la scelta delle strutture dati per rappresentare **insiemi dinamici** influenzino il tempo di accesso ai dati
- strutture dati:
  - array (statico e ridimensionabile)
  - liste
  - hash

1. Insiemi dinamici

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

1. Insiemi dinamici, operazioni

Esistono due **tipi di operazioni**:

- interrogazione (query)
- modifiche

Operazione tipiche:

- inserimento (insert)
- ricerca (search)
- cancellazione (delete)

1. Insiemi dinamici, operazioni

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)

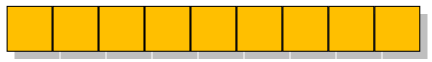
1. Complessità delle operazioni

- la **complessità**
  - 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
- quale operazione sono necessarie dipende dall'applicazione

2. Array

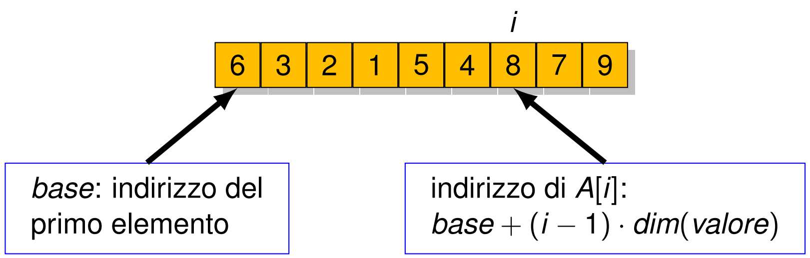
Un **array** è una sequenza di caselle:

- 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

2. Array, indirizzo di una cella



- con l'accesso diretto il tempo per leggere/scrivere in una cella è $O(1)$

2.1 Array statico

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

2.1 Array statico

- inserimento dell'elemento k nel array A:

ArrayInsert(A, k)
  if $A.N \neq A.M$ **then**
    $A.N \leftarrow A.N + 1$
    $A[N] \leftarrow k$
    return k
  else
    return *nil*

- 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

2.1 Array statico

- rimozione dell'elemento k dal array A:

ArrayDelete(A, k)
  for $i \leftarrow 1$ to $A.N$ **do**
    if $A[i] == k$ **then**
      $A.N \leftarrow A.N - 1$
      for $j \leftarrow i$ to $A.N$ **do**
        $A[j] \leftarrow A[j+1]$
      return k
  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

2.1 Array statico

- ricerca dell'elemento k nel array A:

ArraySearch(A, k)
  for $i \leftarrow 1$ to $A.N$ **do**
    if $A[i] == k$ **then**
      return k
  return *nil*

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

2.1 Array statico

- 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?

2.1 Array statico

- ricerca nell'array ordinato:

ArrayBinarySearch(A, k)
  $l, h \leftarrow 1, A.N$
  while $l \leq h$ **do**
    $m \leftarrow \lfloor l + h)/2 \rfloor$
    if $A[m] == k$ **then**
      return m
    if $A[m] > k$ **then**
      $h \leftarrow m - 1$
    if $A[m] < k$ **then**
      $l \leftarrow m + 1$
  return *nil*

- quanto costa fare una ricerca se l'array è ordinato?
  $O(\log N)$, costo logaritmico

2.1 Array statico

- eseguire inserimenti tenendo l'array ordinato costa di più
- come sarebbe l'algoritmo ArrayInsertOrd che mantiene l'array ordinato?:
  - si inserisce l'elemento in fondo (se c'è spazio)
  - si fa scendere l'elemento nella posizione giusta facendo scambi (come fa l'insertion-sort)
- che complessità ha l'algoritmo ArrayInsertOrd?
  - tempo $O(N)$

2.1 Array statico

- 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?

2.2 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**
- **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]
  return B
```


2.2 Array ridimensionabile

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)
  ARRAYINSERT(A, k)
```


2.2 Array ridimensionabile

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**

2.2 Array ridimensionabile

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?

2.2 Array ridimensionabile

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**

2.2 Array ridimensionabile

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

2.2 Array ridimensionabile

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

- raddoppia il numero di elementi se A è pieno
- un'investimento pagato in spazio per un guadagno futuro in tempo

2.2 Array ridimensionabile

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

- ... qui si recupera spazio

2.2 Array ridimensionabile

- 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

2.2 Array ridimensionabile

- 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?

2.2 Costo ammortizzato

Quando il costo delle operazione consecutive 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

2.2 Array ridimensionabile

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

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

cioè la a complessità ammortizzata è $O(N)$

2.2 Array ridimensionabile

complessità ammortizzata di un inserimento con la **seconda idea** in una lunga seria di 2^K inserimenti con $M = 1$ inizialmente:

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

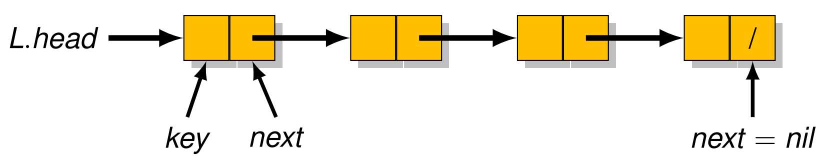
cioè la a complessità ammortizzata è $O(1)$

2.2 Array ridimensionabile

- 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?

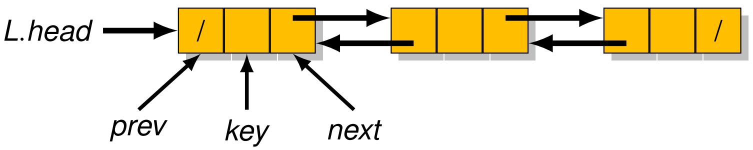
3. Liste concatenate

- una struttura dati lineare
- **l'ordine è determinato dai puntatori** che indicano l'elemento successivo
- data una lista L il primo elemento è indicato dal puntatore $L.head$



3. Liste concatenate

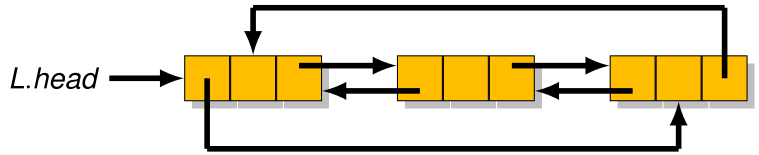
- la lista può essere **doppiamente concatenata**:



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

3. Liste concatenate

- la lista può essere **circolare**:



- la lista circolare può essere vista come un anello di elementi

3. Liste concatenate

- liste doppiamente concatenate e non ordinate
- **ricerca**:

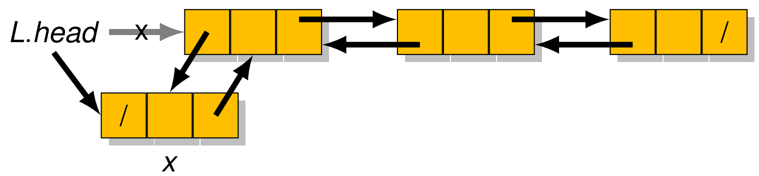
LISTSEARCH(L, k)
  $x \leftarrow L.head$
  while $x \neq nil$ and $x.key \neq k$ **do**
    $x \leftarrow x.next$
  return x

- complessità?
- $O(N)$

3. Liste concatenate

- liste doppiamente concatenate e non ordinate
- **inserimento "in testa"**:

LISTINSERT(L, x)
  $x.next \leftarrow L.head$
  if $L.head \neq nil$ **then**
    $L.head.prev \leftarrow x$
  $L.head \leftarrow x$
  $x.prev \leftarrow nil$



- complessità? $O(1)$

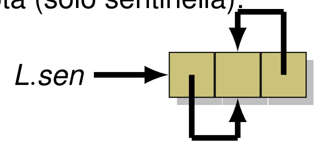
3. Liste concatenate

- liste doppiamente concatenate e non ordinate
- **rimozione** di un elemento puntato da x:

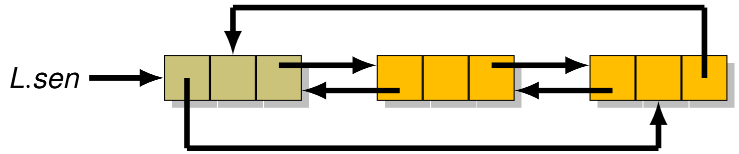
  LISTDELETE(L, x)
  if $x.prev \neq nil$ **then**
  $x.prev.next \leftarrow x.next$
  else
  $L.head \leftarrow x.next$
  if $x.next \neq nil$ **then**
  $x.next.prev \leftarrow x.prev$
- complessità? $O(1)$

3. Liste concatenate circolare con sentinella

- 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:



3. Liste concatenate circolare con sentinella

- operazioni su liste doppiamente concatenate e non ordinate **con sentinella**
- **rimozione** di un elemento puntato da x:

  LISTDELETESEN(L, x)
  $x.prev.next \leftarrow x.next$
  $x.next.prev \leftarrow x.prev$
- complessità? rimane $O(1)$ ma il codice è più semplice e leggibile
- si risparmia un tempo $O(1)$

3. Liste concatenate circolare con sentinella

- operazioni su liste doppiamente concatenate e non ordinate **con sentinella**
- **ricerca** (codice analogo con qualche sostituzione):

  LISTSEARCHSEN(L, k)
  $x \leftarrow L.sen.next$
  while $x \neq L.sen$ and $x.key \neq k$ **do**
  $x \leftarrow x.next$
  return x
- complessità?
- $O(N)$

3. Liste concatenate circolare con sentinella

- operazioni su liste doppiamente concatenate e non ordinate **con sentinella**
- **inserimento "in testa"** (si risparmia un controllo):

  LISTINSERTSEN(L, x)
  $x.next \leftarrow L.sen.next$
  $L.sen.next.prev \leftarrow x$
  $L.sen.next \leftarrow x$
  $x.prev \leftarrow L.sen$
- complessità?
- $O(1)$

3. Liste concatenate

- 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. Tavole hash, introduzione

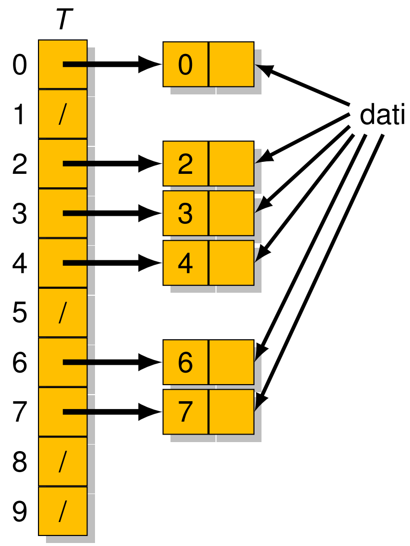
- con array e liste è facile implementare tanti tipi di operazioni
- ma con ognuna il costo di certi operazioni è $O(N)$
- le **tabelle hash** forniscono solo le operazioni di base (insert, search e delete) ma ognuna con tempo medio $O(1)$

4.1 Tavole a indirizzamento diretto

- 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

4.1 Tavole a indirizzamento diretto

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



4.1 Tavole a indirizzamento diretto

- le operazioni sono semplicissime:

  TABLEINSERT(T, x)
  $T[x.key] \leftarrow x$
  TABLEDELETE(T, x)
  $T[x.key] \leftarrow nil$
  TABLESEARCH(k)
  return $T[k]$
- operazioni in tempo $O(1)$

4.1 Tavole a indirizzamento diretto

- 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

4.1 Tavole a indirizzamento diretto

- 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
- spazio occupato ma non utilizzato in assoluto (i *nil*): $8(10^6 - 20000)$=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

4.1 Tavole a indirizzamento diretto

- 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

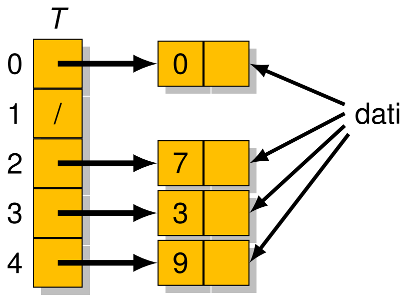
- l'indirizzamento diretto non è praticabile se l'universo delle chiavi è grande
- e in ogni caso non è efficiente dal punto di vista della memoria utilizzata
- idea: utilizziamo una tabella T di dimensione m con m molto più piccolo di $|U|$
- la posizione della chiave k è determinata utilizzando una funziona

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

chiamata la funzione hash

4.2 Tavole hash

- 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



4.2 Tavole hash

- 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**

4.2 Tavole hash

- 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 $0, 1, \ldots, m-1$ 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**:

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

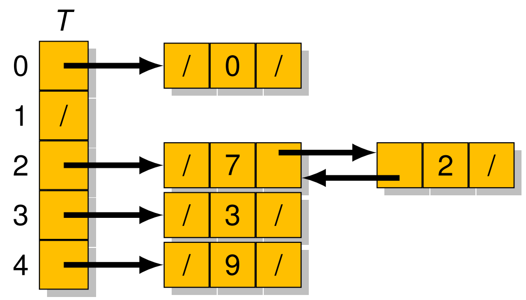
- se $|U| > m$ allora, il **hash perfetto realizzabile solo se l'insieme rappresentato non è dinamico**

4.3 Tavole hash con concatenamento

- come si fa a risolvere le collisioni che comunque possono capitare?
- una possibile soluzione: **concatenando gli elementi in collisione in una lista**

4.3 Tavole hash con concatenamento

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



4.3 Tavole hash con concatenamento

- 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?

4.3 Tavole hash con concatenamento

- 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 $T[h(k)]$
- 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

4.3 Tavole hash con concatenamento

- 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

4.3 Tavole hash con concatenamento

- qual è il **caso peggiore**?
- scenario:
  - l'universo delle chiavi: matricole con 6 cifre
  - $m = 200$
  - funzione hash: $h(k) = k \bmod 200$
- elenco di inserimento che rende pesante la ricerca: 000123,100323,123723,343123,333123,...
- 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
- ricerca costa nel caso migliore $O(1)$

4.3 Tavole hash, uniformità semplice

- qual è il costo nel **caso medio**?
- dipende dalla funzione hash
- assumiamo di avere una funzione che
  - è facile da calcolare ($O(1)$)
  - 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)

4.3 Tavole hash, uniformità semplice

- 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 ($c \in \{0, 1, 2, \ldots, 9\}$)
- ognuno di questi numeri appare 10 volte come ultima cifra
- ogni cella è destinazione di 10 chiavi
- è uniforme semplice

4.3 Tavole hash, uniformità semplice

- la seguente funzione hash è uniforme semplice?

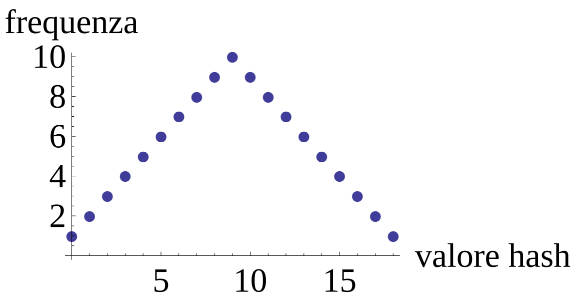
$$U = \{0, 1, 2, \ldots, 99\}, m = 19,$$

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

- cioè h restituisce la somma delle cifre della chiave
- $h(k) = 0$ per $k = 0$
- $h(k) = 1$ per $k = 1$ e $k = 10$
- $h(k) = 2$ per $k = 2$ e $k = 11$ e $k = 20$
- ...

4.3 Tavole hash, uniformità semplice

- frequenza dei vari valori hash:



- non è uniforme semplice

4.3 Tavole hash con concatenamento

- **caso medio** con hashing uniforme semplice
- quanti elementi ci sono in una lista in media?
- sia n_i il numero di elementi nella lista $T[i]$ con $i = 0, 1, \ldots, m-1$
- numero medio di elementi in una lista:

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

4.3 Tavole hash con concatenamento

- tempo medio di **cercare un elemento che non c'è**:
  - 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 ha in media α elementi e quindi percorrere la lista costa in media $\Theta(\alpha)$
- il tempo richiesto è $\Theta(1) + \Theta(\alpha) = \Theta(1 + \alpha)$
- attenzione: α non è costante

4.3 Tavole hash con concatenamento

- tempo medio di **cercare un elemento che c'è**
- cerchiamo di capire quanto costa la ricerca di un elemento scelto a caso fra quelli presenti
- 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 (inserimento in testa)
  - e hanno una chiave con lo stesso valore hash

4.3 Tavole hash con concatenamento

- 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

4.3 Tavole hash con concatenamento

- tempo per ricercare x_i, calcolo del valore hash a parte, è proporzionale a
$$1 + \frac{N-i}{m}$$
- tempo per ricercare un elemento scelto a caso, calcolo del valore hash a parte, è proporzionale a
$$\frac{1}{N}\sum_{i=1}^{N}\left(1 + \frac{N-i}{m}\right)$$

4.3 Tavole hash con concatenamento

- elaboriamo la quantità precedente:

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

- tempo richiesto in totale è

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

4.3 Tavole hash con concatenamento

- **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

4.4 Funzioni hash

- 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

4.4 Metodo della divisione

- il **metodo della divisione** assegna alla chiave k la posizione
$$h(k) = k \mod m$$
- molto veloce
- bisogna scegliere m bene

4.4 Metodo della divisione

- stringhe come numeri naturali secondo il codice ASCII
$$\text{oca} \rightarrow 111 \cdot 128^2 + 99 \cdot 128^1 + 97 \cdot 128^0$$
- posizioni con diverse scelte di m

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

- $m = 2^p$ è una buona scelta 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

4.4 Metodo della moltiplicazione

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

4.5 Indirizzamento aperto, ispezione lineare

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

	T
0	/
1	/
2	12
3	2
4	22
5	33
6	/
7	/
8	88
9	/

4.5 Indirizzamento aperto

- 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 la funzione hash "normale"

4.5 Indirizzamento aperto

- **inserimento in generale** 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
    i ← i + 1
  return nil
```


4.5 Indirizzamento aperto

- **ricerca in generale** con indirizzamento aperto

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

4.5 Indirizzamento aperto

- **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

4.5 Indirizzamento aperto, schemi di ispezione

- l'ispezione lineare crea file di celle occupate, fenomeno chiamato **addensamento primario**
- **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**
- **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

4.5 Indirizzamento aperto, 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

4.5 Indirizzamento aperto, costo della ricerca

- 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 la prima è occupata:

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

- bisogna esaminare almeno tre celle con probabilità:

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

- bisogna esaminare almeno i celle con probabilità:

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

4.5 Indirizzamento aperto, costo della ricerca

- numero medio di celle esaminate:

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

- numero medio di ispezioni è minore di $1/(1-\alpha)$
- come viene $1/(1-\alpha)$ con certi valori di α?
- l'inserimento si analizza con lo stesso approccio
- ricerca con successo richiede esaminare meno celle