Insiemi dinamici

Sono **strutture** :

- che hanno un **numero finito di elementi**
- che hanno elementi che **possono cambiare**
- dove assumiamo che ogni elemento ha un **attributo** diverso che serve da **chiave**
- le **chiavi** son tutte **diverse**

Possiamo inoltre definire principalmente due tipi di **operazioni:**

- **Interrogazione**(query)
- **Modifiche**

Operazioni tipiche possono essere:

- **Insert**
- **Search**
- **Delete**

Se gli insiemi sono **totalmente ordinati**, è possibile effettuare anche le seguenti operazioni:

- **Ricerca del minimo**
- **Ricerca del massimo**
- **Ricerca** del prossimo elemento più grande(**successor**)
- **Ricerca** del prossimo elemento più piccolo(**predecessor**)

La **complessità** è misurata in funzione della **dimensione dell'insieme**, inoltre **dipende** anche dalla **struttura dati** utilizzata.
Infatti alcune operazioni possono risultare pesanti su alcune strutture dati e leggere su altre.

Array

Un array è una sequenza di caselle **grandi uguali** allocate nella memoria **contiguamente**.
Ogni casella **può contenere** un elemento dell'insieme.

Il **calcolo** dell'indirizzo di qualunque elemento dell'array ha **costo costante.**
Dunque anche **accedere** ad un elemento qualsiasi ha un **costo costante.**

Array statico

E' un array il cui numero di elementi massimo è **prefissato**

Quindi, quand'è che ci conviene **usare** un **array statico** e **quanto costano le varie operazioni**?

Se l'array non è ordinato:

- Per quanto riguarda l'inserimento in un array, il **costo** è **costante** ($\in O(1)$)

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

  Il primo if serve a controllare che ci sia abbastanza spazio per inserire il nuovo elemento

- L'**eliminazione** di un elemento, invece **è lineare** ($\in O(n)$).
  Anche se conoscessimo già la posizione dell'elemento da eliminare, dovremmo spostare tutti gli elementi.

```
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]
          return k
  return nil
```

- Anche la **ricerca** di un elemento **è lineare**.
  Infatti dobbiamo scorrere tutto l'array per trovare l'elemento.
  Stessa cosa si applica per la **ricerca** del **minimo** e del **massimo**

"array_search.png" is not created yet. Click to create.

Ricerca di un generico elemento in un array non ordinato

- Per quanto riguarda la ricerca del **successor** (e del **predecessor**), l'algoritmo è leggermente più **complicato** rispetto alla classica **ricerca** ma il suo tempo computazionale rimane **lineare**

Se l'array **è ordinato** invece:

- L'**inserimento è lineare** visto che bisogna spostare tutti gli elementi
- L'**eliminazione** rimane **lineare**
- La **ricerca** invece, diventa **logaritmica**($\in O(\log n)$, visto che possiamo applicare un algoritmo dicotomico
- La ricerca del **minimo**, **massimo**, del **predecessor** e del **successor** diventano **costanti** ($\in O(1)$)

Array ridimensionabile

Se non conosciamo a priori il **numero massimo** di elementi, possiamo **espandere** l'array quando finisce lo spazio.
Tuttavia espandere costa **tempo lineare**

Una **prima idea** sarebbe quella di **aumentare** la dimensione dell'array **di una cella** ogni volta che viene inserito un elemento nell'array(se è pieno ovviamente).
Tuttavia, così facendo, **ogni inserimento** su un array pieno avrebbe costo **lineare**

Dunque il **costo** dell'inserimento **dipende** dallo **stato** dell'array e dalle **operazioni precedenti**

Una **seconda idea** potrebbe essere quella di **raddoppiare** la dimensione dell'array quando questo è pieno e **dimezzarla** quando il **numero** degli elementi presenti nell'array diventa $\leq \frac{1}{4}$

```
DynArrayInsert2(A, k)
  if A.N == A.M then
      A ← ArrayExtend(A, A.M)
  ArrayInsert(A, k)
```

Il primo if controlla se l'array è pieno, e se lo è aumenta la dimensione dell'array

```
DynArrayDelete2(A, k)
  ArrayDelete(A, k)
  if A.N ≤ 1/4 · A.M then
      B ← un array di dimensione A.M/2
      B.M ← A.M/2
      B.N ← A.N
      for i ← 1 to A.N do
          B[i] ← A[i]
      A ← B
```

Tuttavia, visto che c'è una **dipendenza** tra le varie **operazioni**, bisogna calcolare la **complessità ammortizzata:**

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 $O(N)$

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

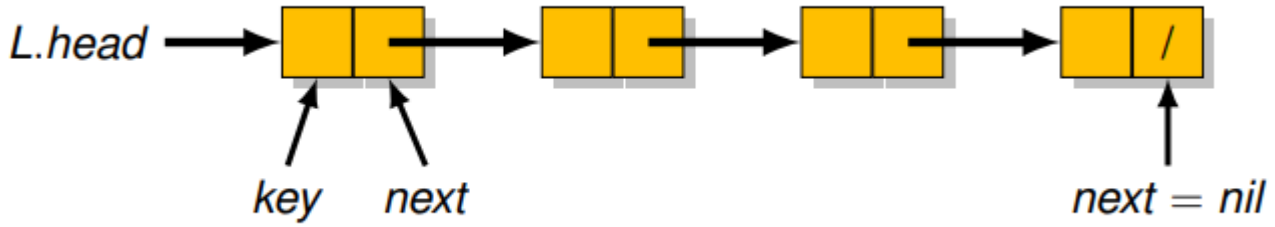
$$\begin{aligned} T_{amm} &= \frac{(c + 2c + 4c + 8c + \cdots + 2^{K-1}c) + 2^K d}{2^K} \\ &= \frac{(2^K - 1)c + 2^K d}{2^K} \in O(1) \end{aligned}$$

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

Liste Concate

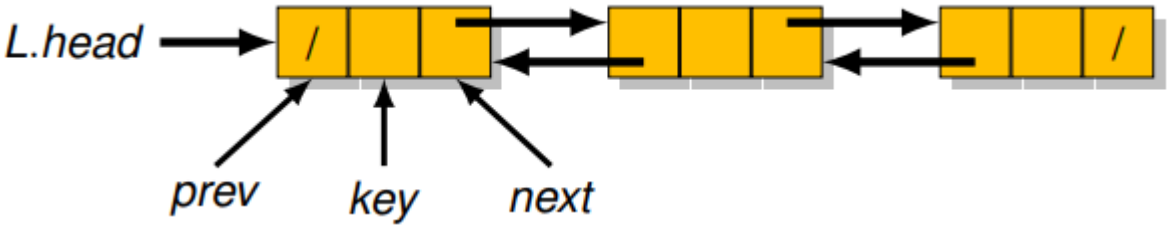
E' una struttura dati **lineare** il cui **ordine** è determinato dai puntatori che indicano l'elemento **successivo**.

Data una lista L, il primo elemento è indicato dal puntatore $L.head$

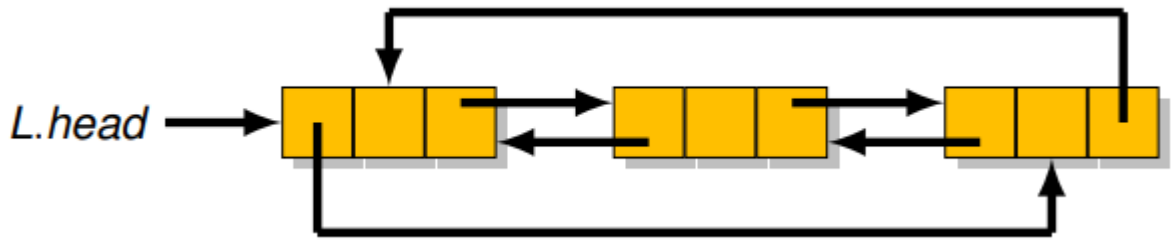


.

Può essere anche **doppiamente concatenata:**
oltre al puntatore all'elemento successivo, abbiamo un puntatore all'**elemento precedente**



Se il **prev** del primo elemento lo facciamo puntare all'ultimo elemento, otteniamo una lista **circolare



Per ognuna di queste versioni ovviamente esiste anche la variante **ordinata** dove gli elementi sono, banalmente, ordinati secondo la **chiave**

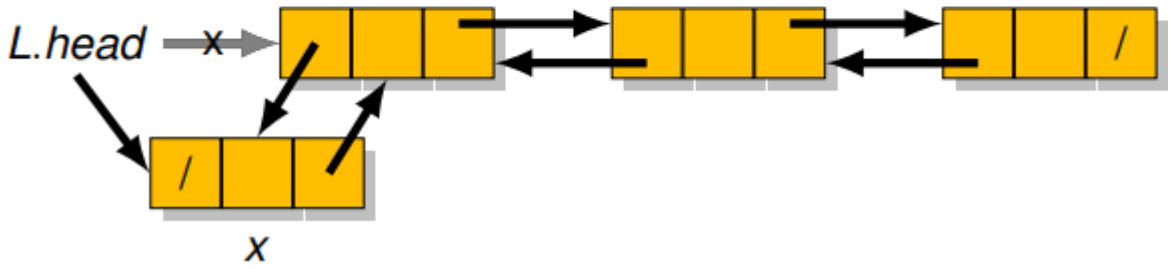
Sulle **liste doppiamente concatenate non ordinate** possiamo ovviamente **ricercare** un elemento:

```
LISTSEARCH(L, k)
  x ← L.head
  while x ≠ nil and x.key ≠ k do
      x ← x.next
  return x
```

dove è facile notare che la sua complessità è $O(n)$

Chiaramente possiamo anche **inserire** un elemento **in testa**

```
LISTINSERT(L, x)
  x.next ← L.head
  if L.head ≠ nil then
      L.head.prev ← x
  L.head ← x
  x.prev ← nil
```



Il prev ora punterà all'elemento in testa, e il nuovo prev sarà nil.
Inoltre il next ora punterà all'elemento che prima era in testa.

La complessità è **costante**($O(1)$), infatti non dipende dal numero degli elementi

Si può inoltre **rimuovere** un elemento **puntato** da x:

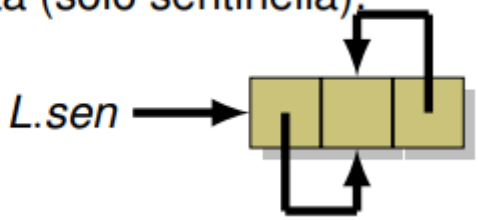
```
LISTDELETE(L, x)
  if x.prev ≠ nil then
      x.prev.next ← x.next
  else
      L.head ← x.next
  if x.next ≠ nil then
      x.next.prev ← x.prev
```

Bisogna controllare se l'elemento da eliminare è il primo(primo if) o l'ultimo(secondo if)
Anche in questo caso la complessità è **costante**

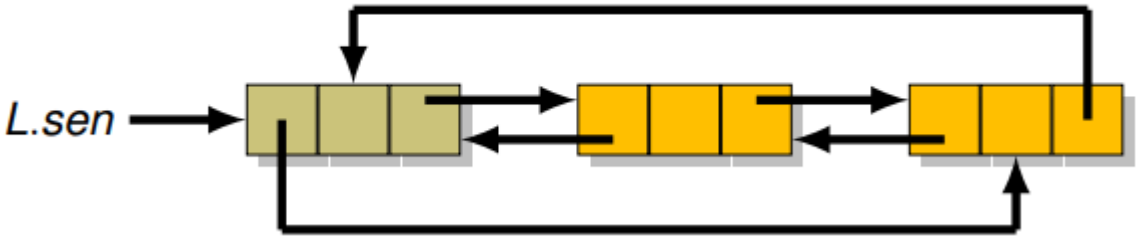
Tuttavia è un po' macchinoso e di difficile comprensione(bisogna effettuare dei controlli in testa e in coda).

Possiamo aggiungere un elemento fittizio che non contiene dati:
una **sentinella** che serve a far sì che la nostra lista non sia mai effettivamente vuota

- lista circolare vuota (solo sentinella):



- lista circolare non vuota:



In questo modo, l'algoritmo di **rimozione** diventa più leggibile:

```
LISTDELETESEN(L, x)
  x.prev.next ← x.next
  x.next.prev ← x.prev
```

Non devo più fare i controlli per vedere se l'elemento che devo rimuovere sia il primo o l'ultimo
La complessità rimane inoltre **costante**

Versione con sentinella della **ricerca di un elemento**:

```
LISTSEARCHSEN(L, k)
  x ← L.sen.next
  while x ≠ L.sen and x.key ≠ k do
      x ← x.next
  return x
```

In modo analogo, anche per l'**inserimento** in testa evitiamo dei controlli da fare:

```
LISTINSERTSEN(L, x)
  x.next ← L.sen.next
  L.sen.next.prev ← x
  L.sen.next ← x
  x.prev ← L.sen
```

Hash Table

Per **array**(e **liste**) molte operazioni hanno costo **lineare**($O(N)$).
Tuttavia per le **Hash Table** vengono fornite le operazioni di base con tempo

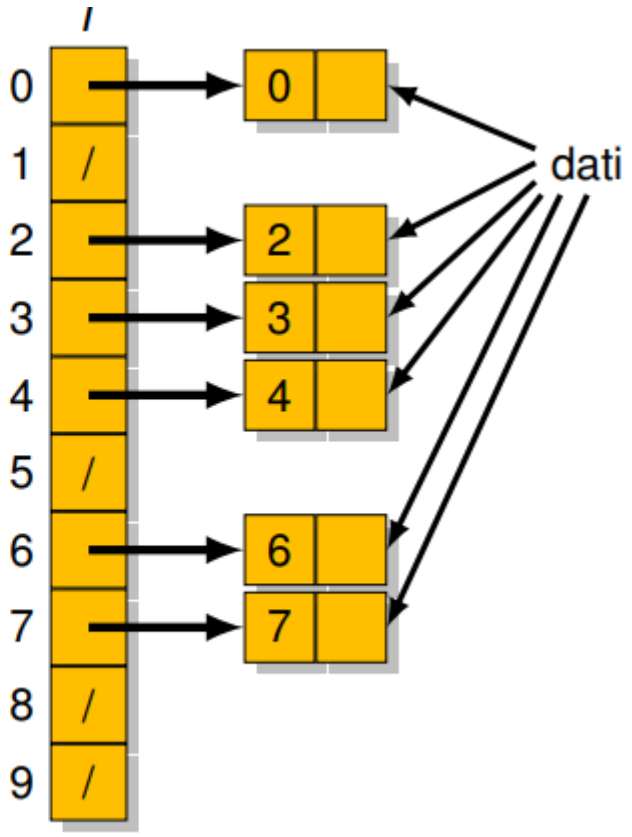
costante

Tavole ad indirizzamento diretto

Una idea di base per le **Hash Table** è quella delle **Tavole ad indirizzamento diretto:**
Sia U l'universo delle chiavi $U = \{0, 1, .., m-1\}$
L'insieme dinamico viene rappresentato con un array T di dimensione $m$in cui ogni posizione corrisponde ad una chiave.

T è ad **indirizzamente diretto** perchè ogni sua cella corrisponde direttamente ad una chiave

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



L'**inserimento/eliminazione/ricerca** però diventa con costo **costante:**

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

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

TABLESEARCH(k)
  return $T[k]$

E' proprio ciò che faremo in un array statico

Dal punto di vista **computazionale** è senza dubbio efficiente.
Tuttavia non è sempre così dal punto di vista dello **spazio occupato**

Analizzando i seguenti casi, possiamo capire in quali contesti conviene usare le **tavole ad 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
- 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”

Hash Table nel dettaglio

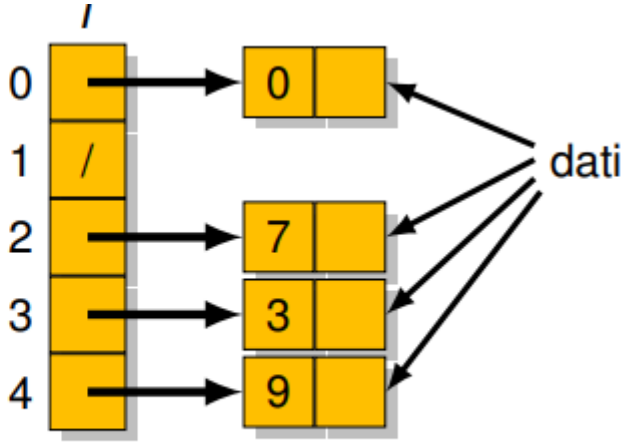
L'indirizzamento diretto non è praticabile se l'universo delle chiavi è grande
e come abbiamo visto **non** è **efficiente** dal punto di vista della memoria utilizzata.

Idea di base: Utilizziamo una tabella T con dimensione m con m molto più piccolo di $|U|$
La posizione della chiave k è determinata utilizzando una funzione:
la cosiddetta **funzione hash**

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

Esempio di **funzione 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



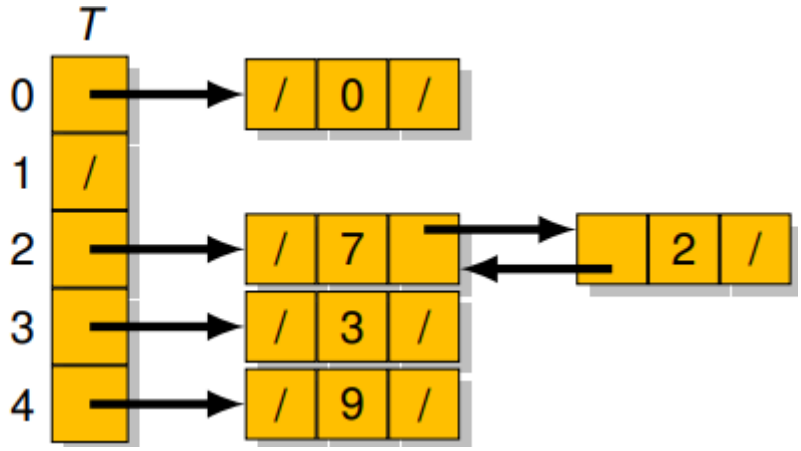
In questo modo perdiamo **l'indirizzamento diretto**.
Infatti ora l'elemento k non si trova più nella posizione k ma in $h(k)$.
Riduciamo però lo spazio utilizzato(dal momento che $m < |U|$).
Tuttavia c'è **rischio di collisione**

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** dunque deve ridurre al **minimo** le collisioni.
Un **hash perfetto** intuitivamente è una funzione che non crea mai collisioni.
E' dunque una **funzione iniettiva:**
$k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2)$
Ma un **hash perfetto** è realizzabile soltanto **se** l'insieme **non è dinamico**

Una possibile soluzione alle collisioni è di **concatenare in una lista gli elementi in collisione**

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



e dunque, in caso di concatenamento, le operazioni diventeranno:

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)

T[h(x.key)] è una lista

- **Inserimento:** E' $O(1)$ visto che il valore hash si calcola in tempo **costante**
- **Cancellazione:** Essendo una lista **doppiamente concatenata**, l'eliminazione di un elemento individuato è **costante**
- **Ricerca:** Dipende dalla lunghezza della lista **T[h(k)]** dunque dipende da:
  - **Numero di elementi**
  - **Caratteristiche della funzione hash**

Analizziamo dunque l'operazione di ricerca ma prima parliamo di

Funzionie hash uniforme semplice

E' una funzione **hash** che distribuisce le chiavi in modo uniforme tra le celle.
Dunque **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 ($c \in \{0, 1, 2, \ldots, 9\}$)
ognuno di questi numeri appare 10 volte come ultima cifra
ogni cella è destinazione di 10 chiavi
è uniforme semplice

Esempio di funzione hash che gode dell'uniformità semplice

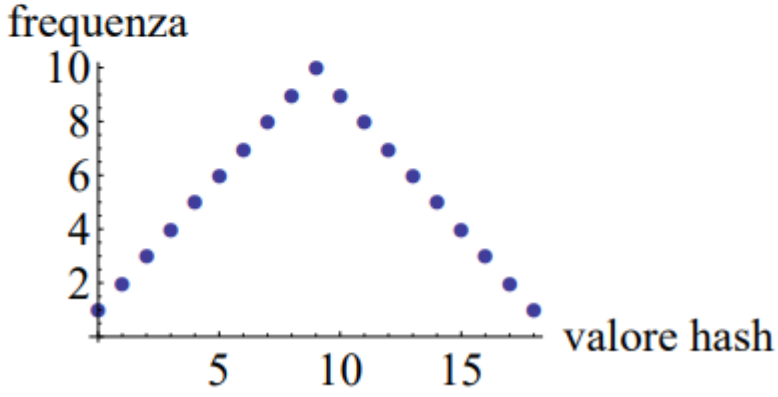
la seguente funzione hash è uniforme semplice?

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

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

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 $k = 20$

frequenza dei vari valori hash:



non è uniforme semplice

Esempio di funzione hash che NON gode dell'uniformità semplice

Worst,Best e Average Case

Worst Case:

Supponendo che:
- **Universo delle chiavi(U):** matricole con 6 cifre

- **Numero di celle nella Hash-Table(m):** 200
- **Funzione hash:** $h(k) = k\ mod\ 200$

Se inseriamo i seguenti numeri
$000123, 100323, 123723, 343123, 333123$
tutte le chiavi saranno associate alla stessa cella di T
dunque in questo caso la ricerca costerà, nel **caso peggiore**, $\Theta(N)$

Best Case:
Se la lista è vuota oppure contiene un solo elemento, la ricerca costerà $O(1)$

Average Case:
Assumiamo di avere una funzione hash che:

- è facile da calcolare, dunque $O(1)$
- gode della proprietà di **uniformità semplice**

Sia n_i il numero di elementi nella lista $T[i]$ con $i = 0, 1, .., m-1$
Il numero medio di elementi in una lista è:
$\overline{n} = \frac{n_0+n_1+...+n_{m-1}}{m} = \frac{N}{m} = \alpha$

Tempo medio per cercare un elemento che non c'è:

- La funzione di hash abbiamo detto che è **costante** dunque il costo per individuare la lista è $\Theta(1)$
- La lista ha in media α elementi e quindi percorerre la lista costa in media $\Theta(\alpha)$
- Dunque in totale il tempo richiesto è $\Theta(1) + \Theta(\alpha) = \Theta(1 + alpha)$

Attenzione: α non è costante

Tempo medio per cercare un elemento che c'è:

- Per individuare la lista il costo è sempre $\Theta(1)$
- Supponendo che vogliamo trovare l'elemento x_i, dobbiamo esaminare x_i stesso e
    - gli elementi che son stati inseriti dopo x_i(inserimento in testa)
    - gli elementi hanno una chiave con lo stesso valore hash

Dopo x_i vengono inseriti $N - i$ elementi (intuitivamente, se ho inserito i elementi, ne rimangono $N - i$ da inserire)

Ogni elemento inserito ha $\frac{1}{m}$ probabilità di finire nella stessa lista di x_i(ovvero $\frac{1}{m}$ di probabilità che l'elemento abbia la chiave con lo stesso valore hash)

Dunque **in media** $\frac{N-i}{m}$ elementi precedono x_i nella lista di x_i

Quindi:

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

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

dove l'ultima quantità la possiamo elaborare in

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

tempo richiesto in totale è

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

Conclusione:
Visto che $\alpha = O(m)$, la ricerca è $O(1)$
Dunque tutte e tre le operazione hanno tempo computazionale costante(sempre sotto l'assunzione che le liste siano doppiamente concatenate)

Metodi per l'hashing

Metodo della divisione: $h(k) = k \ mod \ m$

- E' molto veloce
- Bisogna scegliere m bene tuttavia

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

Esempio di metodo della divisione con m=2048 e m=1583

Un buon valore per m potrebbe essere 2^p se si ha la certezza che gli ultimi bit hanno distribuzione uniforme
Un altro buon valore potrebbe essere un **numero primo** non vicino ad una potenza di 2

Metodo della moltiplicazione: $h(k) = \lfloor m(Ak \ mod \ 1) \rfloor$
con $0 < A < 1$
dove $Ak \ mod \ 1$ è la parte frazionaria di Ak

In queso caso il valore di m non è critico e un valore ragionevole può essere una potenza di 2.

La scelta ottimale di A dipende dai dati ma $A = \frac{\sqrt{5}-1}{2}$ è un valore ragionevole

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.
Altrimenti si cerca una posizione libera secondo uno **schema d'ispezione**.

Quello più semplice è l'**ispezione lineare:** a partire dalla posizione $h(k)$, l'elemento verrà inserito nella prima cella libera.

Insert 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
```

Search 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
```