7° lezione algoritmi

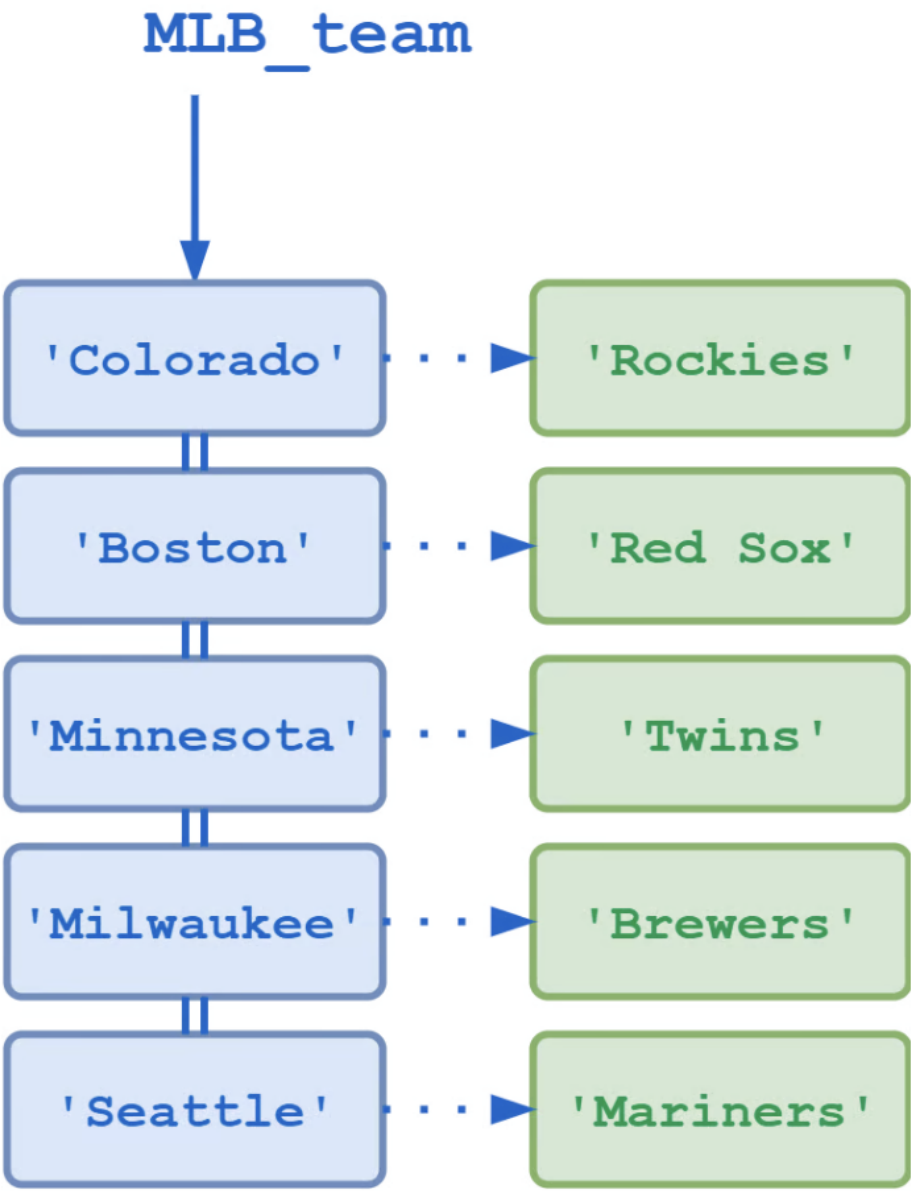
Due Date	@October 29, 2024
Materia	Algoritmi e Laboratorio
Status	Done

Hash e tabelle hash

Tabelle hash

Le tabelle hash servono per implementare una struttura astratta, chiamata **dizionario,** una struttura dati formata da due parametri **chiave valore**.

```
MLB_team = dict([
    ('Colorado', 'Rockies'),
    ('Boston', 'Red Sox'),
    ('Minnesota', 'Twins'),
    ('Milwaukee', 'Brewers'),
    ('Seattle', 'Mariners')
])
```



La chiave è colorata in blu, invece, il valore è colorato in verde .

l'operazione di ricerca può dare due risultati:

1. **Ricerca con successo**, si sta cercando un elemento che sicuramente esiste. Si ha una complessità $O(\frac{n}{2})$
2. **Ricerca senza successo** che non troviamo l'elemento che cerchiamo, quindi sicuramente non esiste. In questo caso, però è necessario scorrere tutta la struttura. Con complessita $O(n)$

La principale operazione che si esegue in questa struttura dati è la **ricerca**, sono messa anche operazioni di **inserimento e cancellazione**, ma queste sono in genere più rare rispetto alla ricerca.

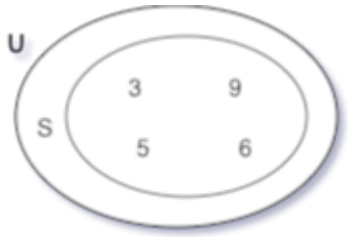
Si può implementare un dizionario nel seguente modo :

- ▼ **Array ordinato**
  - **Inserimento:** $O(n)$ perché si deve trovare lo spazio, ed anche spostare gli elementi a destra
  - **Cancellazione:** $O(n)$ perché quando si elimina un elemento tutti gli elementi restanti vanno spostati da destra verso sinistra, mantenendo l'ordine.
  - **Ricerca:** $O(log\ n)$ con l'ausilio della ricerca dicotomica
- ▼ **Array non ordinato**
  - **Inserimento:** $O(1)$ in quanto l'elemento da inserire va in qualsiasi posizione tanto non ha importanza l' ordine
  - **Cancellazione:** $O(1)$ in quanto non si ha la necessità di spostare tutti gli elementi a sinistra, perché il vettore non è ordinato
  - **Ricerca:** $O(n)$, in quanto il vettore non è ordinato
- ▼ **Liste ordinate**
  - **Inserimento:** $O(n)$
  - **Cancellazione:** $O(1)$
  - **Ricerca:** $O(n)$ in quanto la ricerca binaria non è utilizzabile perché non si può accedere in tempo costante ad ogni elemento della lista
- ▼ **Liste non ordinate**
  - **Inserimento:** $O(1)$
  - **Cancellazione:** $O(1)$ **non è necessario spostare tutti gli elementi a sinistra, in quanto il vettore non è ordinato**
  - **Ricerca:** $O(n)$
- ▼ **BST bilanciato**
  - **Inserimento:** $O(log\ n)$
  - **Cancellazione:** $O(log\ n)$
  - **Ricerca:** $O(log\ n)$ Grazie alle proprie proprietà di bilanciamento.

Da quanto detto fino adesso si può affermare che la soluzione migliore è **l'albero binario di ricerca bilanciato**. Si potrebbe anche utilizzare la t**abella a indirizzamento diretto.**

Tabella ad indirizzamento diretto

Supponendo di avere U l'insegna tutte le chiave rappresentabili dizionario ed S le chiavi effettivamente presenti. Le operazioni relative ad un elemento k. $U \subseteq S$




```
insert(T, k):
    T[k] = 1

delete(T, k):
    T[k] = 0

search(T, k):
    if T[k] = 1 then
        return true
    return false
```

Nella tabella ad indirizzamento diretto l'inserimento, la cancellazione e la ricerca sono operazioni in tempo costante.

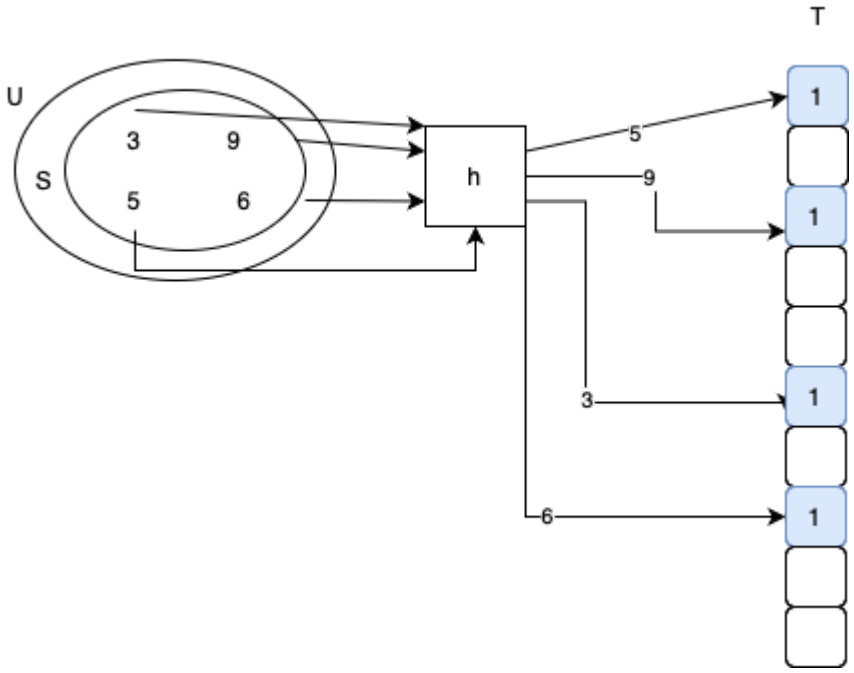
T a cardinalità m (elemento più grande di S)

- Se m è molto grande ed S è **sparso(**elementi non distribuiti adeguatamente ma con molte celle di memoria saltate)**,** allora T È uno spreco di memoria;
- L'insieme U non può essere un insieme grande, pertanto risulta impraticabile in quanto non si ha abbastanza spazio di memoria.

Funzioni Hash

Una soluzione alla resezione del problema citato prima può essere l'introduzione di una **black box** che ha lo scopo di indirizzare gli elementi nella tabella T. Questo utilizza una funzione di hashing.

$$h : U \to \{0, 1, ..., n-1\}$$



Il vantaggio di avere adottato questa soluzione è che l'insieme H può essere composto da quanti elementi si vuole così da ridurre notevolmente il numero di zeri presenti nella tabella.

```
insert(T,k):
    T[h(k)] = 1
```

```
delete(T, k):
    T[h(k)] = 0

search(T, k):
    if T[h(k)] = 1 then
        return true
    return false
```

La funzione di hash viene calcolata in tempo costante

Può capitare che la funzione hash restituisce lo stesso indice per due elementi?

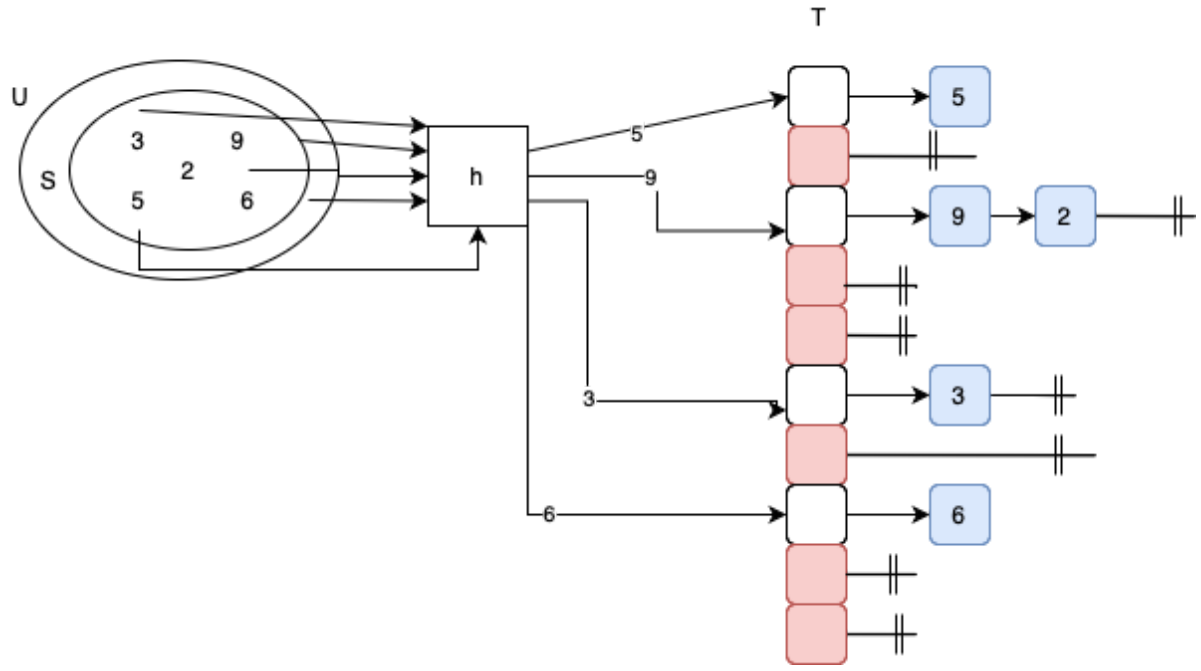
Supponiamo di avere due chiavi $k_1 \ e \ k_2 \in U$ tali che $h(k_1) = h(k_2)$. In questo caso, si ha una **collisione di due chiavi,** non è possibile evitarla. Tale problema è dovuto al fatto che la funzione di hashing passa da un dominio grande ad un condominio piccolo.

si hanno due modi per risolvere le collisioni:

1. **Risoluzione per concatenazione**
2. **Risoluzione per indirizzamento aperto**

Risoluzione per concatenazione

Se si ha una collezione cioè $k_1 \ e \ k_2$ hanno la stessa immagine per h, gli elementi coinvolti vengono inseriti all'interno di una lista concatenata che mantiene tutti gli elementi che hanno la stessa immagine per h.




```
insert(T,k):
    list-insert(T[h(k)], k)

delete(T, k):
    list-delete(T[h(k)], k)

search(T, k):
    return list-search(T[h(k)], k)
```

La funzione hash **deve essere deterministica** ciò vuol dire che quando si va a calcolare una funzione di hash per un determinato elemento deve essere la stessa se si va a ricalcolare questa nello stesso elemento considerato.

La funzione `insert(T,k)` e la `delete(T,k)` hanno complessità $O(1)$ considerando per la funzione `delete(T,k)` una lista doppiamente concatenata.

La funzione `search(T,k)` prende tempo proporzionale alla lunghezza della vista stessa. Nel caso pessimo è che gli $|S|$ elementi sono tutti indirizzati nella stessa posizione. Nel caso peggiore, quindi la complessità è $O(n)$. La tabella hash ha un comportamento migliore nel caso medio.

> **Una tabella HASH mantiene costante la ricerca nel caso medio.**

Hashing uniforme semplice

Per il caso medio è necessario che la funzione di hashing sia un **hashing uniforme semplice.**

> Un hashing si dice uniforme semplice se soddisfa la seguente proprietà:
>
> $$P_i\{h(x) = i\} = \frac{1}{m}$$
>
> Comunque presa una chiave $k \in U$. La probabilità che $h(k) = i$ con $0 < i < m$ questa sia indirizzata ad una cella i è la stessa per ogni cella con m celle. Da ciò si deduce che si ha una distribuzione uniforme delle chiavi.
>
> In altri termini
>
> $$\sum_{0 \leq i \leq m} P_i\{h(k) = i\} = 1$$

Introduciamo, inoltre, un fattore α che prende il nome di **fattore di carico**.

$$\alpha = \frac{n}{m}$$
$$\text{in generale } \alpha \geq 0$$

$m \rightarrow$ **numero di celle** ed $n \rightarrow$ **numero di elementi**. Esprime un percentuale quanto la tabella è carica. α **non deve superare di molto** 1

La dimensione media di una lista concatenata di una cella di T in media è uguale a α, poichè se $\frac{1}{m}$ è la probabilità che un elemento venga inserito nella cella i, allora, per ogni cella i la sua è la somma:

Quanto è lunga in media ogni lista concatenata?

Supponiamo di avere n elementi è questi vengono distribuiti maniera uniforme nelle m caselle. Ogni casella contiene $\frac{n}{m}$ caselle.

$$\sum_{k \in S} P(h(k) = i) = \frac{n}{m}$$

La complessità nel caso medio della ricerca è $O(\alpha)$.

Implementazioni di funzioni di hashing

I principali metodi di implementazione di funzioni di hashing sono due:

1. Metodo della **divisione**
2. metodo della **moltiplicazione**

Metodo della divisone

Dato $k \in U$. k è rappresentabile come un numero intero(assunzione più che valida, dopo che anche i reali sono rappresentati come interi con codifiche diversa nei PC). Ogni elemento viene rappresentato in memoria mediante una sequenza di bit è questa può essere sempre comunque interpretata come un intero.

$$h(k) = k \ mod \ m$$
$$k \ mod \ m \leq m$$
$$0 < k \ mod \ m < m - 1$$

Esempio

Supponiamo che $m = 10$. Per eseguire l'operazione $h(k) = k \ mod \ 10$ basta guardare l'utima cifra e si determina la posizione. Se i k dovessero essere multipli di 6, ad esempio ci sarebbero tante collisioni, quindi non si avrebbe una funzione ottima,

Quando si stabilisce come sarà composta la funzione di hash non si conosce la composizione dell' insieme S quindi si va un pò a fortuna, ma se si riesce a fare in modo che data una chiave k la funzione hash dipenda da. tutta la chiave e non da una porzione di essa allora si riesce ad andare nella direzione di un **hashing uniforme**. P**iù è grande la porzione di informazioni su cui si calcola la funzione di hash e più si riduce la possibiltà che ci siano delle collisioni.**

Si dovrebbe evitare di avere $m = 2^p$ in quanto se dovessimo eseguire questa operazione si avrebbe come risultato lo spostamento di p bit a destra. I p bit spostati sono il resto della divisione e quindi l'hashing dipenderebbe dagli ultimi p bit dei valori. Supponendo di avere due chiavi $k_1 \ e \ k_2$ diverse si potrebbe

avere il caso in cui la parte decimale sia la stessa tra le due chiavi e per tale ragione si andrebbe in contro ad una **collisione**

Dopo queste considerazioni si deduce che converrebbe prendere un **numero primo**.

Metodo della moltiplicazione

Si considera un parametro A con $0 < A < 1$. si moltiplica $k \cdot A$ ottenendo un numero $0 < A < k$ si vra una parte intera e d una decimale, si procede facedo il $mod\ 1$ di $k \cdot A$ ottenendo solo la parte decimale. $0 < kA\ mod\ 1 < 1$ con 1 escluso, si procede moltiplicando $(k \cdot A \bmod 1) \cdot m$. Si avrà come risultato finale:

$$h(k) = (kA \bmod 1) \cdot m$$
$$0 < (kA \bmod 1) \cdot m < m$$

Siccome l'insieme continuo è dobbiamo prendere i numeri interi, quindi si considera il **floor** pertanto si avrà:

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

Questa funzione così com'è fatta può essere dispendioso, ma la si può migliorare con le operazioni di **bitwise**.

Sopponiamo di avere una **word** di dimensione w. A è un numero $0 < A < 1$. È possibile rappresentare A come numero intero, se avviene uno spostamento a sinistra dei bit di A di w posizioni, si sta trasformado A in un numero intero in simboli:

$$S = A \cdot 2^w$$

S è un intero rappresentabile in w bit, ed anche un numero la cui dimensione è $2 \cdot w$ bit. Per calcolare $kA \mod 1$ si vanno a considerare i bit appartenenti alla word di sinistra

Supponiamo di avere $m = 2^p$ facendo il prodotto $k \cdot A$ si vanno a spostare i bit a sinistra di p posti. Si vanno a considerare in generale i primi p bit a sinistra della prima word a partire da destra del numero $S \cdot k$ quando si va a considerare $m = 2^p$