UNIVERSITÀ DEGLI STUDI DI TRIESTE

MODULO 2: Tabelle hash

Prof.ssa Giulia Cisotto

giulia.cisotto@units.it

Trieste, 30 aprile 2026

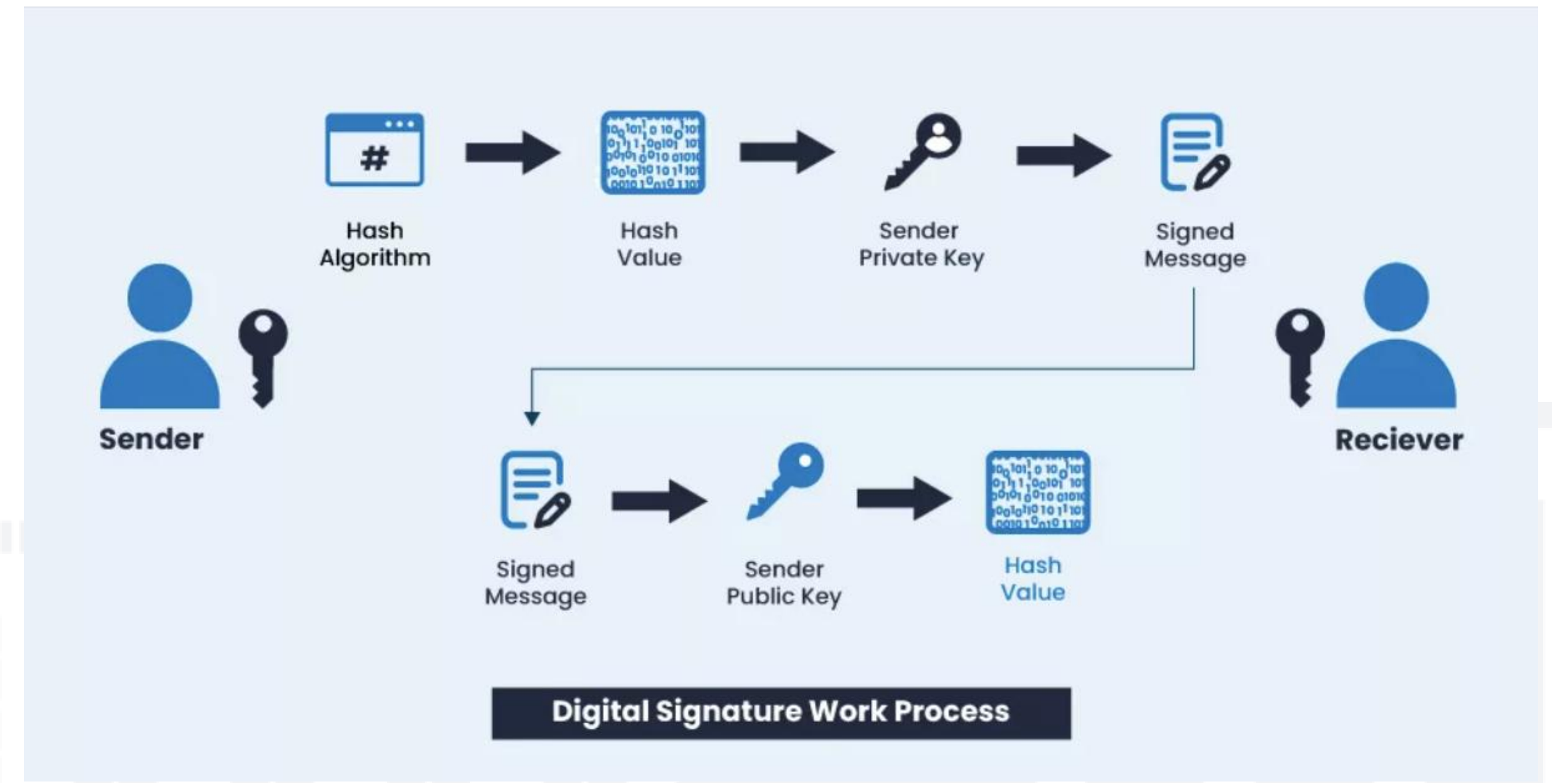
APPLICAZIONI

Tabelle di simboli nei compilatori (MODULO 1)

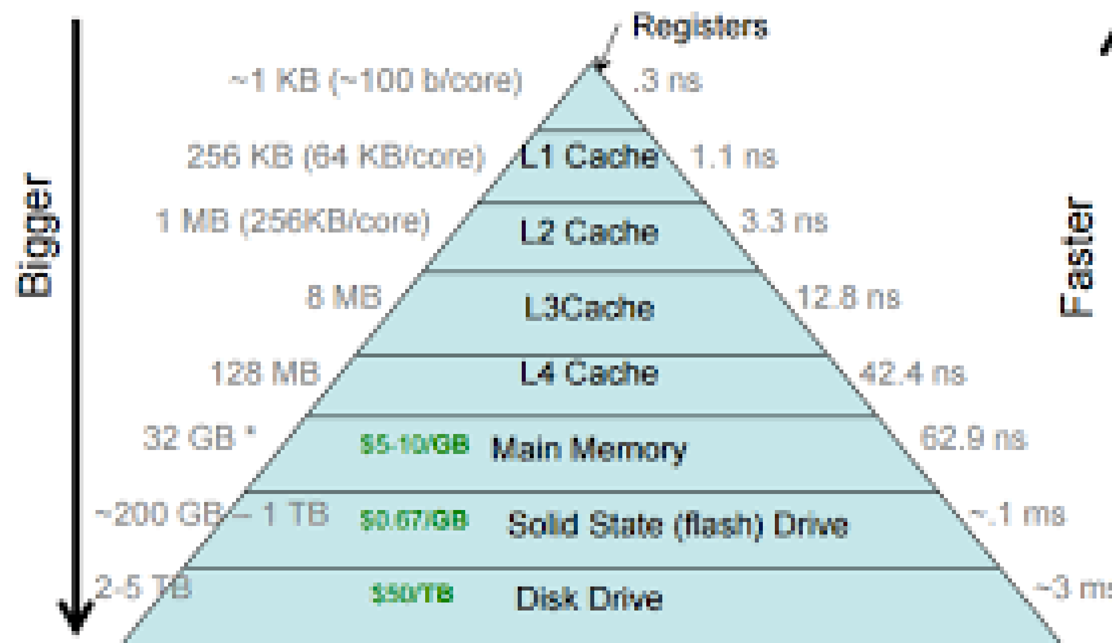
Symbol Table

SIMBOLO	Valore ILC	altro…
….	….	….
is_sorted	0x420	….
.L18	0x432	…..
.L20	0x462	….
…..	….	….

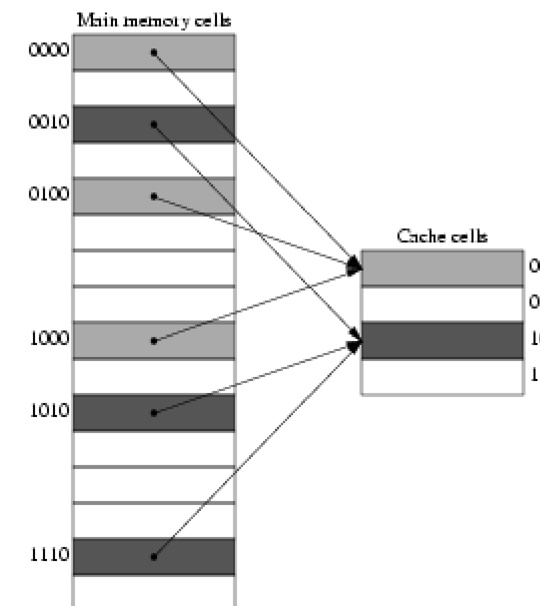
Certificati di firma digitale



https://certera.com/blog/digital-certificate-vs-digital-signature/

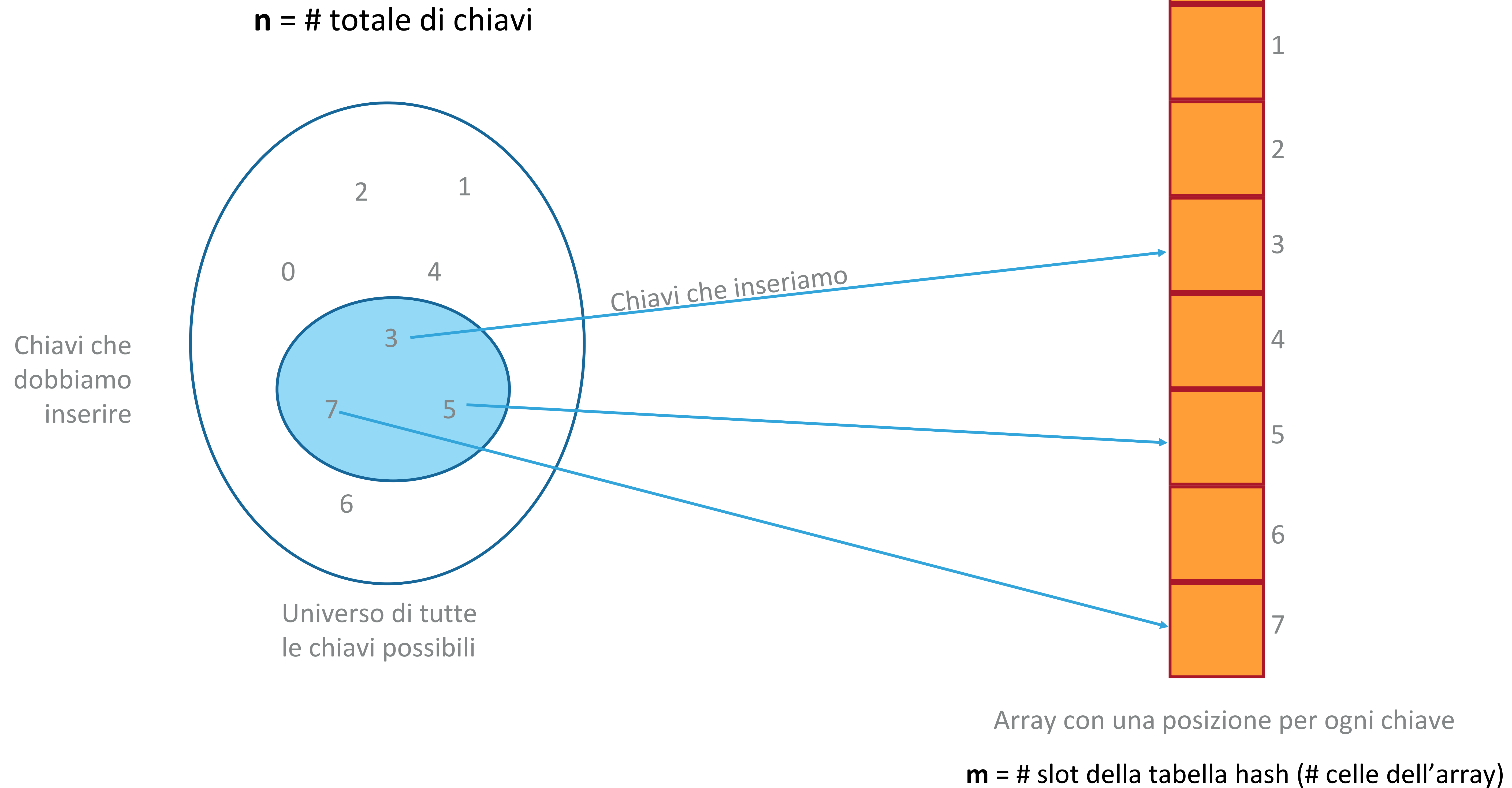


Memoria cache



UNIVERSITÀ DEGLI STUDI DI TRIESTE

IDEA DI BASE



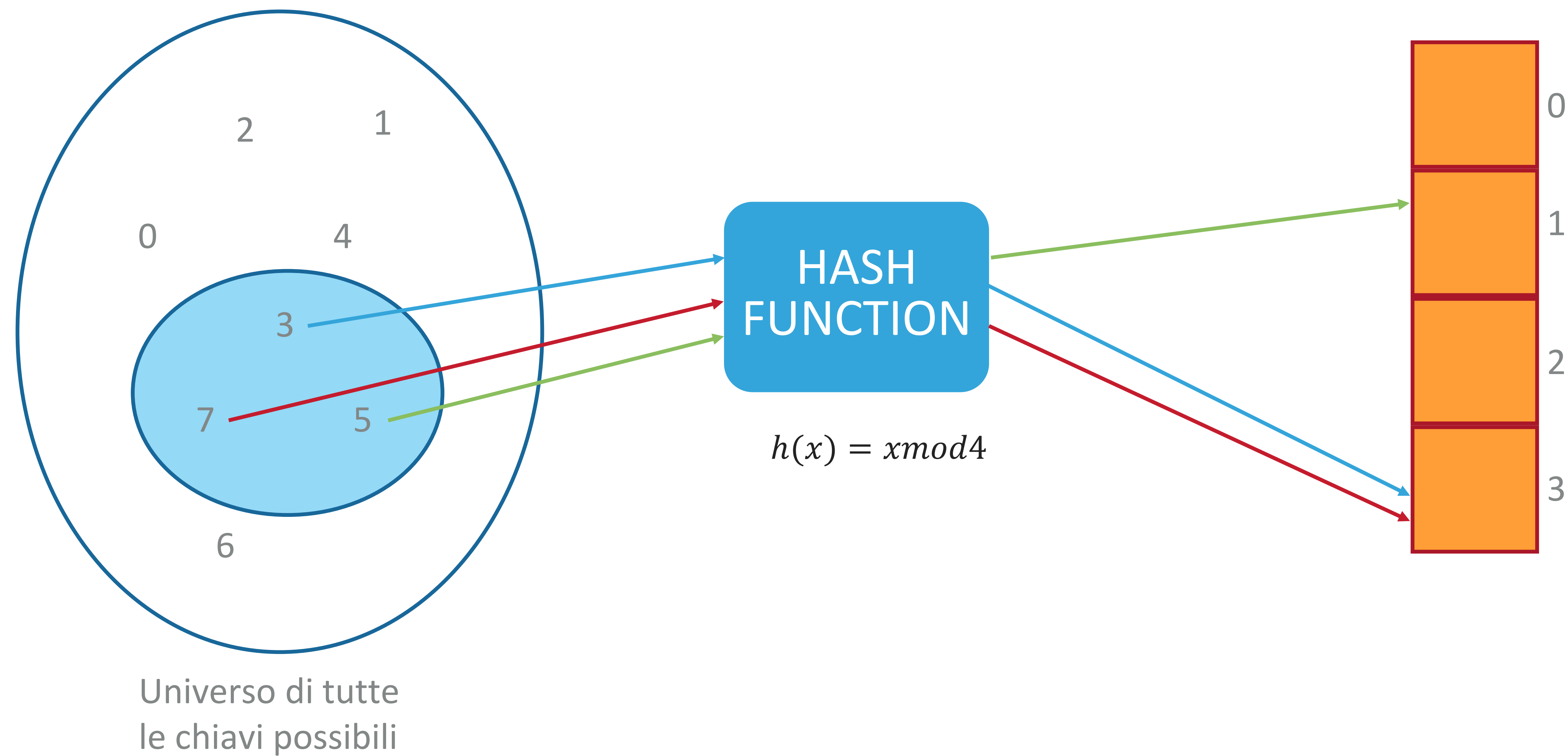
CI SONO DEI PROBLEMI?

- Questo approccio ci permette di fare ricerca, inserimento e rimozione in tempo costante!
- **Problema**: l’insieme di tutte le chiavi potrebbe non essere abbastanza piccolo da permettere questo approccio.
  - Esempio: con numeri di 32 bit si avrebbero circa quattro miliardi di slot, anche salvando solo un byte per ogni slot avremmo 4GB di memoria occupati!
- *Possiamo riformulare l’idea in modo che possa funzionare?*

TABELLE HASH

- **Invece di utilizzare direttamente le chiavi come indici, usiamo una funzione (detta *funzione di hash*)** che, data una chiave, ci dice dove trovarla all'interno di una tabella
- Questo ci permette di definire quelle che sono chiamate le *tabelle hash:*
  - Dato un array, una chiave k, ed una funzione di hash h, **la posizione in cui inserire/trovare $\boldsymbol{k}$ nell'array è $\boldsymbol{h(k)}$.**
  - Il codominio di h può essere molto ridotto!

UNA PRIMA TABELLA HASH



**mod = funzione che calcola il resto della divisione tra x e 4*

TABELLE HASH

- Finché non abbiamo due chiavi distinte con lo stesso hash (i.e., $k_1 \neq k_2$ ma $h(k_1) = h(k_2)$) tutte le operazioni continuano ad essere effettuabili in tempo costante (assumendo che h richieda tempo costante)
- Però dobbiamo gestire questo caso (le **collisioni**).
- A seconda di come decidiamo di gestirlo abbiamo diverse varianti di tabelle hash.

GESTIRE LE COLLISIONI

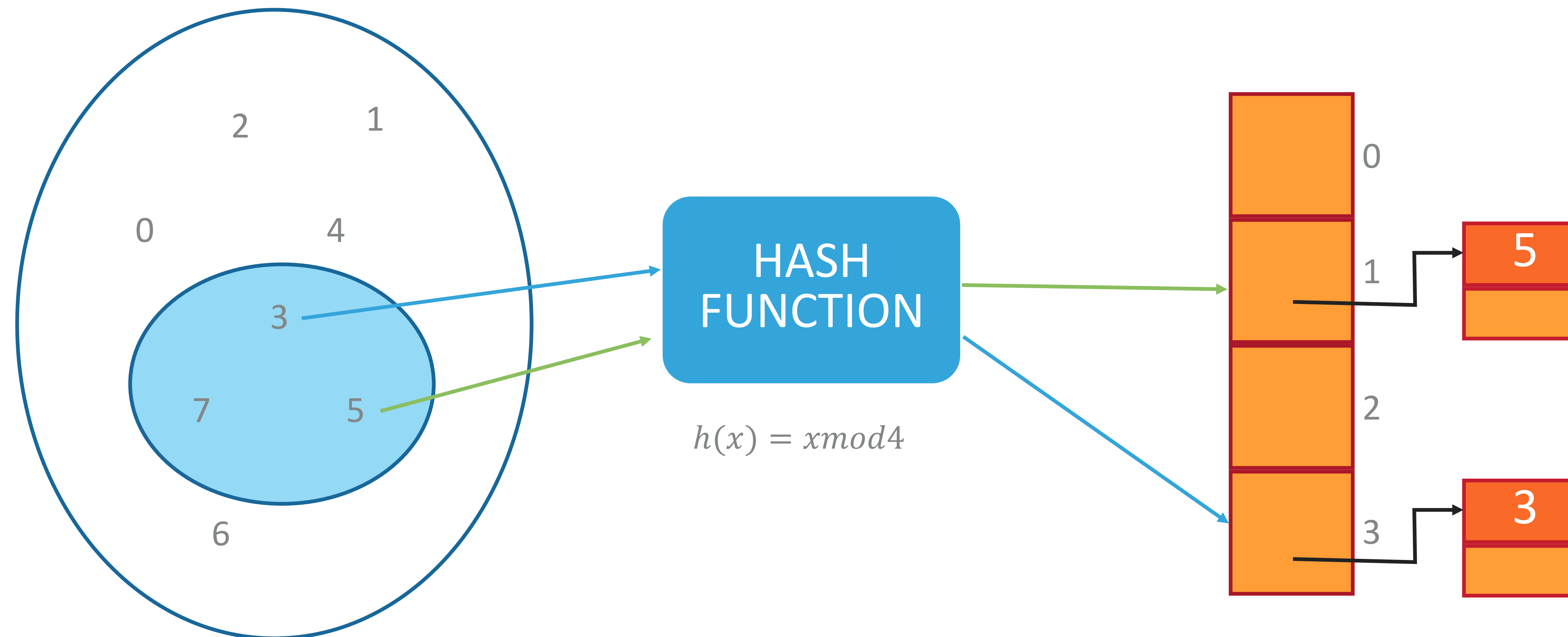
1. Possiamo tenere per ogni slot una lista concatenata di valori che hanno lo stesso hash: **chaining** o "**hash con concatenazione**"

2. Possiamo invece cercare un altro posto libero nella tabella: **open addressing** o **indirizzamento aperto o *probing***. Ne esistono diverse varianti, tra cui ispezione lineare o quadratica, oppure doppio hashing.

CHAINING / CONCATENAZIONE

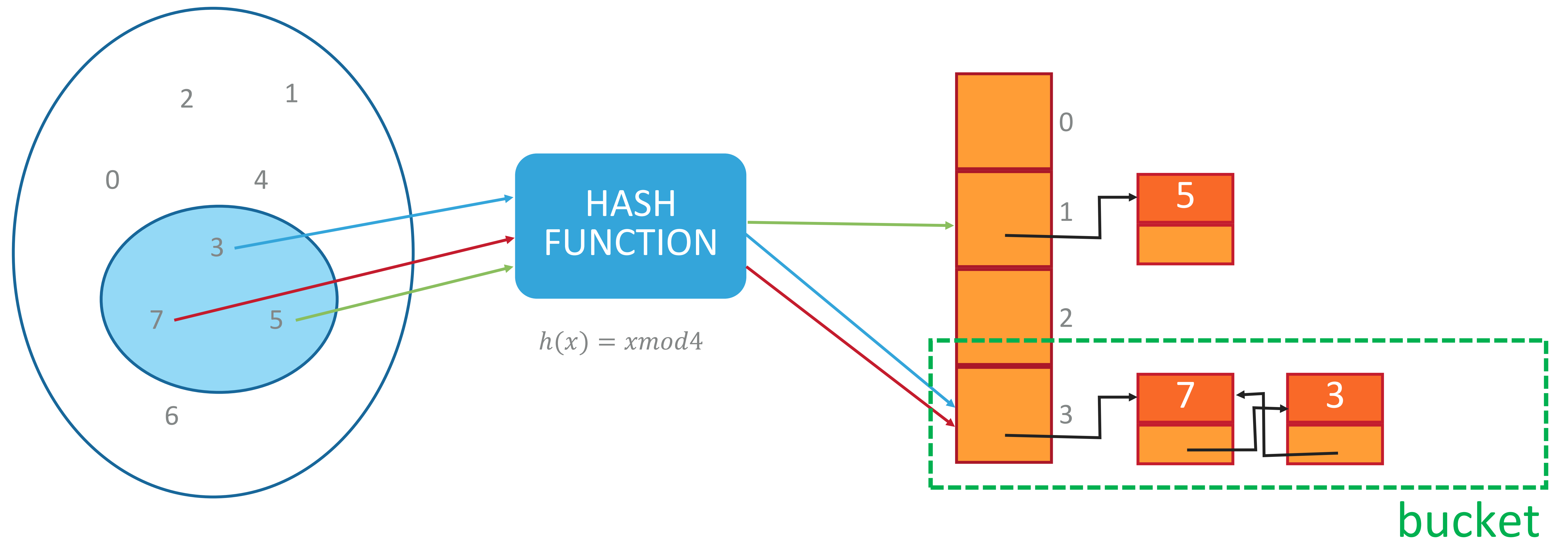
- Abbiamo un array di m elementi, ognuno punta ad una lista concatenata (di solito doppia)
- **L'inserimento** di un elemento x di chiave k si riconduce a un inserimento in testa alla lista di indice $h(k)$
- La **rimozione** di un elemento x di chiave k si riconduce a una rimozione dalla lista di indice $h(k)$
- La **ricerca** data una chiave k si riconduce alla ricerca di k nella lista di indice $h(k)$

HASH CON CHAINING



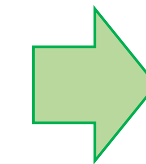
Il prossimo inserimento provocherà una collisione. Come lo gestiamo?

HASH CON CHAINING



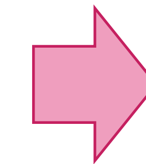
CHAINING / CONCATENAZIONE E COMPLESSITÀ

```
Inserimento
Parametri: x (l'oggetto da inserire) e la tabella T
inserisci x in testa a T[h(x.key)]
```



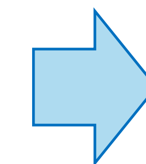
$O(1)$

```
Ricerca
Parametri: k (la chiave) e la tabella T
ricerca lineare nella lista T[h(k)]
```



$O(n)$

```
Rimozione
Parametri: x (l'oggetto da rimuovere) e la tabella T
rimuovi x da T[h(x.key)]
```



$O(1)$ $O(n)$

*Nota: **rimozione** nel caso in cui si abbia già puntatore ad elemento da rimuovere.

CHAINING / CONCATENAZIONE

- Nel caso peggiore abbiamo tutti gli n elementi nella stessa lista concatenata
- Quindi, anche se inserimento e rimozione richiedono tempo $O(1)$, la ricerca richiede invece tempo $O(n)$
- *L'analisi è corretta per il caso peggiore, ma possiamo ottenere qualcosa di meglio guardando il* ***tempo medio?***

TABELLE HASH: ALCUNE DEFINIZIONI

- Come al solito, indichiamo con $\boldsymbol{n}$ **il numero di elementi contenuti nella tabella**
- Indichiamo con $\boldsymbol{m}$ **la dimensione della tabella**
- $\alpha = n/m$ è il **load factor** o **fattore di carico** della tabella.
- Il fattore di carico indica quanto “piena” è la tabella:
  - $\alpha < 1$ abbiamo più posti nella tabella che elementi inseriti
  - $\alpha > 1$ abbiamo più elementi inseriti che posti nella tabella

dimostrazione disponibile in fondo al set di slide della lezione

CHAINING / CONCATENAZIONE

Assumiamo che la funzione di hash distribuisca uniformemente le chiavi negli m slot.

Sotto queste assunzioni, si dimostra che **il tempo medio per cercare un elemento in una tabella hash è $\boldsymbol{\Theta(1+\alpha)}$.**

La funzione di hash è una funzione deterministica. Questo serve per poter trovare le chiavi: *la chiave k finirà sempre nello stesso bucket h(k).*

Le prestazioni medie però valgono sotto l'assunzione che la funzione di hash sia stata scelta in modo che le chiavi si distribuiscano in modo uniforme tra gli slot.
Es. $f(k) = k mod 10$ non è una «buona» scelta se le chiavi che arrivano sono spesso dei multipli di 10.

Serve inoltre che $\boldsymbol{n = O(m)}$, quindi se la tabella si riempie troppo servirà sostituirla con una più grande (ad esempio raddoppiando il numero di slot) e reinserire tutti i valori contenuti nella tabella vecchia.

COMPLESSITÀ PER TABELLE HASH CON HASHING UNIFORME

Operazione	Caso medio	Caso peggiore
Ricerca	$O(1 + \alpha)$	$O(n)$
Inserimento in testa	$O(1)$	$O(1)$
Inserimento con controllo duplicati	$O(1 + \alpha)$	$O(n)$
Cancellazione	$O(1 + \alpha)$ se bisogna cercare la chiave	$O(n)$
Cancellazione con riferimento diretto al nodo	$O(1)$ con lista doppia	$O(1)$

GESTIRE LE COLLISIONI

1. Possiamo tenere per ogni slot una lista concatenata di valori che hanno lo stesso hash: **chaining** o “**hash con concatenazione**”

2. Possiamo invece cercare un altro posto libero nella tabella: **open addressing** o **indirizzamento aperto o *probing***. Ne esistono diverse varianti, tra cui ispezione lineare o quadratica, oppure doppio hashing.

Si veda il file

«Lezione 27b – Tabelle Hash – PARTE 2.pdf»

COMPLESSITÀ PER TABELLE HASH CON PROBING LINEARE

Operazione	Caso medio	Caso peggiore(**)
Ricerca	O(1) se α è costante(*) e << 1	O(m)
Inserimento	O(1) se α è costante e << 1	O(m)
Cancellazione	O(1) se α è costante e << 1	O(m)

(*) Si può mantenere α circa costante facendo resize/rehashing della tabella (es. raddoppiando il # di bucket). *Nota: se la tabella si riempie troppo, le sequenze di probing diventano lunghe.*

(**) Nel caso peggiore → O(m), perchè è probabile dover scandire tutta la tabella

Complessità in dettaglio (per curiosità, senza dimostrazione)		
Operazione	Caso medio/atteso, hashing uniforme	Caso peggiore
Ricerca con successo	$O(1 / (1 - \alpha))$	O(n) o O(m)
Ricerca senza successo	$O(1 / (1 - \alpha)^2)$ per probing lineare	O(m)
Inserimento	$O(1 / (1 - \alpha)^2)$	O(m)
Cancellazione semplice, cercando la chiave	costo della ricerca + O(1)	O(m)
Cancellazione con lazy deletion	costo della ricerca + O(1)	O(m)
Cancellazione con riferimento diretto allo slot	O(1) per marcare DELETED	O(1)

CHAINING VS LINEAR PROBING

Conviene chaining quando serve fare *cancellazioni più semplici*, avere *carichi anche maggiori di 1*, *gestione più robusta delle collisioni*, e non si vuole che la tabella degradi troppo quando è quasi piena.

- Svantaggio: servono puntatori/lista, quindi più memoria e peggiore località in cache.

Conviene probing lineare quando vuoi una *struttura compatta, senza puntatori*, con buona località in cache e accessi molto veloci se α è basso.

- Svantaggio: soffre il clustering, richiede $\alpha < 1$, e le cancellazioni sono più delicate per via dei segnaposto.

Materiale per la lezione

- Cormen et al. **CAP. 11**

Prossima lezione: 5 ***maggio, h.14:00, aula 4C***

Dimostrazione complessità ricerca di un elemento con hashing uniforme

CHAINING / CONCATENAZIONE

Assumiamo che la funzione di hash distribuisca uniformemente le chiavi negli m slot.

Sotto queste assunzioni, mostriamo che **il <u>tempo medio</u> per cercare un elemento in una tabella hash è $\boldsymbol{\Theta(1+\alpha)}$**

Dividiamo la dimostrazione in due parti:

- Il caso in cui ***l'elemento cercato non sia nella tabella***
- Il caso in cui l'elemento cercato sia nella tabella

CHAINING / CONCATENAZIONE

Se l'elemento cercato non è presente:

1. Calcolare l'hash della chiave → si ottiene l'indice di un bucket → **tempo O(1)**
2. Andare nel bucket corrispondente → contiene una **lista concatenata**
3. Scorrere tutta la lista per cercare la chiave → alla fine ci accorgiamo che **non c'è**

Ipotesi: la chiave k con cui lo cerchiamo ha uguale probabilità di finire in uno qualsiasi degli m slot.

Il tempo medio per **scoprire che la chiave non è presente** è dato dalla lunghezza media della lista di indice $h(k)$, ovvero il fattore di carico $\alpha = n/m$

Quindi il tempo medio richiesto per una ricerca senza successo è: $\Theta(1+\alpha)$

CHAINING / CONCATENAZIONE

Se assumiamo che l'elemento x di chiave k sia presente nella lista, allora è egualmente probabile che sia uno qualsiasi degli n elementi presenti nella lista.

Gli elementi sono inseriti in testa alla lista, quindi il tempo richiesto per trovare x dipende da quanti elementi con lo stesso hash sono stati inseriti dopo di lui

Siano $\boldsymbol{x_1, \dots, x_n}$ **gli elementi inseriti** di chiavi $k_1, \dots, k_n$

Indichiamo con $X_{i,j} = 1$ il caso $h(k_i) = h(k_j)$ e $X_{i,j} = 0$ altrimenti

CHAINING / CONCATENAZIONE

Il numero di elementi da visitare prima di trovare x_i sarà quindi:

$$1 + \sum_{j=i+1}^{n} X_{i,j}$$

ovvero uno più tutti gli elementi inseriti dopo di lui

Dato che $\boldsymbol{x}$ **potrebbe essere uno qualsiasi degli** $\boldsymbol{x_i}$,facciamo la media su tutte le possibili posizioni in cui potrebbe essere x, ottenendo il **seguente tempo medio**:

$$E\left[\frac{1}{n}\sum_{i=1}^{n}\left(1 + \sum_{j=i+1}^{n} X_{i,j}\right)\right]$$

CHAINING / CONCATENAZIONE

Possiamo portare dentro il valore atteso per linearità:

$$\frac{1}{n}\sum_{i=1}^{n}\left(1+\sum_{j=i+1}^{n}E[X_{i,j}]\right)$$

Dato che assumiamo che **la funzione di hash distribuisca in modo uniforme le chiavi**, **la probabilità che $X_{i,j}$ sia 1 è $\frac{1}{m}$**, quindi $E[X_{i,j}]=\frac{1}{m}$

Nota: la probabilità di avere una collisione è 1/m*1/m. Ma la collisione può avvenire in una qualsiasi delle m celle della tabella. Quindi la probabilità di collisione nell'intera tabella è 1/m.

CHAINING / CONCATENAZIONE

Otteniamo quindi

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

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

$$=1+\frac{1}{nm}\left(n^2-\frac{n(n+1)}{2}\right)=1+\frac{n}{2m}-\frac{1}{2m}$$

CHAINING / CONCATENAZIONE

Sostituiamo quindi $\alpha = n/m$ ovunque ottenendo

$$1 + \frac{n}{2m} - \frac{1}{2m} = 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} = \Theta(1+\alpha)$$

Sommato al tempo di calcolo per la funzione di hash (che è costante), otteniamo $\boldsymbol{\Theta(1+\alpha)}$.

CONCLUSIONE: Finché $\boldsymbol{n = O(m)}$, abbiamo $\alpha = \frac{O(m)}{m} = O(1)$ e **quindi il tempo medio per la ricerca è $\boldsymbol{O(1)}$**.