

# Strutture dati - Parte 1

Dipartimento di Elettronica, Informazione e Bioingegneria  
Politecnico di Milano

19 maggio 2020

# Strutture dati

## Organizzare i dati processati

- Spesso algoritmi efficienti necessitano di poter accedere, modificare, cancellare i dati su cui agiscono con opportune complessità asintotiche
- L'unico modo che abbiamo visto finora di organizzare elementi su cui effettuiamo un calcolo (nello pseudocodice) sono i *vettori*
- Analizziamo come è possibile rappresentare collezioni di elementi in modo più organizzato → strutture dati più evolute
- Queste strutture possono usare etichette opache (chiavi) per identificare un oggetto
- Analizziamo la loro efficienza in termini di quella delle operazioni che effettuiamo su di esse

# Operazioni tipiche su strutture dati

## Interrogare la struttura

- $\text{Search}(S, k)$ : restituisce il riferimento a  $k$  in  $S$ , NIL se  $k$  non è contenuto in  $S$
- $\text{Minimum}(S)$ : se gli elementi sono ordinati, restituisce il più piccolo (o quello con la chiave più piccola)
- $\text{Maximum}(S)$ : come sopra, ma il più grande
- $\text{Successor}(S, x.k)$ : restituisce l'elemento che segue  $x$  (se ordinati) nella struttura (o la cui chiave segue quella di  $x.k$ )
- $\text{Predecessor}(S, x.k)$ : come sopra, ma considerando il precedente

## Modificare la struttura

- $\text{Insert}(S, x)$ : Inserisce un oggetto nella struttura
- $\text{Delete}(S, x)$ : Cancella un oggetto dalla struttura

# Analisi critica di strutture note

## Vettori

- Un vettore è una struttura dati compatta in memoria in cui si accede direttamente ad ogni elemento, data la sua posizione
- L'indice del vettore agisce come chiave a tutti gli effetti
- Se il vettore di lunghezza  $n$  non è ordinato:
  - ricerca, minimo, massimo, successore sono  $\mathcal{O}(n)$
  - inserimento e cancellazione costano  $\mathcal{O}(n)$  (la cancellazione può essere ridotta a  $\mathcal{O}(1)$  usando dei simboli di “cella vuota”)
- Se il vettore di lunghezza  $n$  è ordinato:
  - minimo e massimo:  $\Theta(1)$ , ricerca e successore  $\Theta(\log(n))$
  - inserimento e cancellazione costano  $\mathcal{O}(n)$
- Inserimenti in vettore pieno:
  - sono rifiutati (tenendo un conteggio degli elementi):  $\mathcal{O}(1)$
  - causano una riallocazione  $\mathcal{O}(n)$ : (causa copie)

# Analisi critica di strutture note

## Liste semplicemente connesse

- Una lista semplice stocca gli elementi sparsi in memoria: ogni elemento ha un riferimento al successivo (i.e., puntatore)
- Se la lista di lunghezza  $n$  non è ordinata:
  - ricerca, minimo, massimo, successore sono  $\mathcal{O}(n)$
  - inserimento:  $\mathcal{O}(1)$ , cancellazione:  $\mathcal{O}(n)$  se l'elemento va trovato,  $\mathcal{O}(1)$  se si ha un riferimento
- Se la lista di lunghezza  $n$  è ordinata:
  - uno dei due tra minimo e massimo è  $\Theta(1)$  l'altro  $\Theta(n)$ 
    - Con puntatore accessorio all'ultimo elemento: entrambi  $\Theta(1)$
  - ricerca e successore sono  $\mathcal{O}(n)$
  - inserimento:  $\mathcal{O}(n)$ , cancellazione:  $\mathcal{O}(n)$

# Pila (Stack)

## Una struttura dati familiare

- Una pila è una struttura dati con le seguenti operazioni:
  - $\text{Push}(S, e)$ : aggiunge l'elemento in cima alla pila
  - $\text{Pop}(S, e)$ : restituisce l'elemento in cima alla pila cancellandolo
  - $\text{Empty}(S)$ : restituisce true se la pila è vuota
- Questa struttura dati astratta può essere realizzata usando una lista semplicemente connessa o un vettore

## Realizzazione con lista

- Lo stoccaggio dati è nella lista, le operazioni diventano:
  - $\text{Push}(S, e)$ : inserisci in testa alla lista  $\mathcal{O}(1)$
  - $\text{Pop}(S, e)$ : restituisci il primo elemento della lista, cancellandolo dalla stessa  $\mathcal{O}(1)$
  - $\text{Empty}(S)$ : controlla se il successore della testa è NIL:  $\mathcal{O}(1)$

# Pile (Stack)

## Realizzazione con vettore

- Lo stoccaggio dati è nella celle del vettore, viene mantenuto l'indice della cima della pila (Top of Stack, ToS)
  - $\text{Push}(S, e)$ : se c'è spazio, Incrementa ToS, salva  $e$  in  $A[\text{ToS}]$ :  $\mathcal{O}(1)$ ; se manca spazio rifiuta  $\mathcal{O}(1)$  o rialloca  $\mathcal{O}(n)$
  - $\text{Pop}(S)$ : restituisci  $A[\text{ToS}]$  corrente, decrementa ToS:  $\mathcal{O}(1)$
  - $\text{Empty}(S)$ : Restituisci  $\text{ToS} \stackrel{?}{=} 0$ :  $\mathcal{O}(1)$
- Nessun vantaggio (dal punto di vista astratto) rispetto all'implementazione a pila, (uno svantaggio se si rialloca)
  - In pratica, avere dati non coesi in memoria penalizza le caches
  - → può valer la pena di usare un vettore se non ci sono troppe riallocazioni (e.g., con una preallocazione ragionata)

# Code (Queues)

## Struttura ed operazioni

- Una coda è una struttura dati con le seguenti operazioni:
  - Enqueue(Q,e): aggiunge e alla fine della coda
  - Dequeue(Q): restituisce l'elemento all'inizio della coda, cancellandolo dalla stessa
  - Empty(Q): restituisce true se la coda è vuota
- Come nel caso della pila, è possibile realizzare una coda sia con una lista che con un vettore



# Code (Queues)

## Realizzazione con vettore

- Lo stoccaggio dei dati è effettuato in un vettore  $A$ , lungo  $l$ , con indice del primo elemento 0
- Teniamo traccia della posizione dove va inserito un nuovo elemento e di quella dell'elemento più vecchio con due indici  $tail$  e  $head$  e del numero di elementi contenuti  $n$
- Gli indici vengono incrementati mod  $l$ 
  - Enqueue( $Q, e$ ): se  $n < l$ , inserisci l'elemento in  $A[tail]$ , incrementa  $n$  e  $tail$ :  $\mathcal{O}(1)$ , altrimenti segnala l'errore,  $\mathcal{O}(1)$
  - Dequeue( $Q$ ): se  $n > 0$ , restituisci  $A[head]$  corrente, decrementa  $n$ , incrementa  $head$ :  $\mathcal{O}(1)$
  - Empty( $S$ ): Restituisci  $n \stackrel{?}{=} 0$ :  $\mathcal{O}(1)$
- Per ampliare lo stoccaggio: allocazione fresca e copia degli elementi estraendoli con Dequeue( $Q$ ):  $\Theta(n)$

## Realizzazione con lista

- Lo stoccaggio dei dati è effettuato negli elementi della lista
- Teniamo traccia dell'ultimo elemento della lista (oltre al primo) con un puntatore `tail`
  - `Enqueue(Q,e)`: inserisci l'elemento `e` in coda alla lista, aggiornando `tail`:  $\mathcal{O}(1)$
  - `Dequeue(Q)`: restituisci l'elemento in testa se diverso da `NIL`, cancellandolo e aggiornando `head`:  $\mathcal{O}(1)$
  - `Empty(S)`: Restituisci `head`<sup>?</sup>`tail`:  $\mathcal{O}(1)$

# Mazzo o coda a due fini (Deque)

## Struttura dati

- La struttura dati si comporta come un mazzo di carte, di cui ognuna contiene un elemento
- E' possibile aggiungere sia in testa che in coda alla struttura:
  - `PushFront(Q, e)`: inserisci l'elemento e in testa al mazzo
  - `PushBack(Q, e)`: inserisci l'elemento e in coda al mazzo
  - `PopFront(Q)`: restituisci l'elemento in testa, cancellandolo
  - `PopBack(Q)`: restituisci l'elemento in coda, cancellandolo
  - `Empty(S)`: Restituisci true se il mazzo è vuoto

# Mazzo o coda a due fini (Deque)

## Realizzazione con vettore

- Lo stoccaggio dei dati è effettuato in modo analogo alla coda semplice
- PushBack e PopFront si comportano come Enqueue e Dequeue della coda realizzata con un vettore
  - PopBack(Q): se  $n > 0$ , restituisci  $A[\text{tail}]$  corrente, decrementa  $n$ , decrementa  $\text{tail}$ :  $\mathcal{O}(1)$
  - PushFront(Q, e): se  $n < l$ , decrementa  $\text{head}$ , inserisci l'elemento in  $A[\text{head}]$ , incrementa  $n$ :  $\mathcal{O}(1)$ , altrimenti segnala l'errore,  $\mathcal{O}(1)$
  - Empty(S): Restituisci  $n \stackrel{?}{=} 0$ :  $\mathcal{O}(1)$
- Ampliamento dello stoccaggio: come per la coda

# Mazzo o coda a due fini (Deque)

## Realizzazione con lista *doppiamente* concatenata

- Un elemento di una lista doppiamente concatenata ha puntatori sia al precedente che al seguente.
- Un modo comune di rappresentarne una vuota è una coppia di elementi, `head` e `tail` che puntano l'uno all'altro
- `PushBack` e `PopFront` si comportano come `Enqueue` e `Dequeue` di una coda realizzata con una lista
  - `PopBack(Q)`: restituisci `tail.prev` corrente se diverso da `head`, rimuovendolo dalla lista:  $\mathcal{O}(1)$
  - `PushFront(Q, e)`: aggiungi l'elemento in testa, aggiornando `head` e il suo successore  $\mathcal{O}(1)$
  - `Empty(S)`: Restituisci `head.next?tail.prev`:  $\mathcal{O}(1)$
- Ampliamento dello stoccaggio: come per la coda

# Riassumendo

## Strutture dati lineari

- Tutte le operazioni viste su pile, code e mazzi sono  $\mathcal{O}(1)$  nel caso delle implementazioni basate su lista, o su vettore con stoccaggio finito
- Le implementazioni che utilizzano vettori come stoccaggio e consentono di ampliarli nel caso vi sia necessità pagano un costo lineare per l'ampliamento

## Liste doppiamente concatenate

- Si comportano come le liste semplici, tranne la cancellazione
- Cancellare un elemento arbitrario, che viene fornito alla `Delete(L, e)` è  $\mathcal{O}(1)$  :  $e.prev.next \leftarrow e.next$ ;  
 $e.next.prev \leftarrow e.prev$

## Rappresentare collezioni di oggetti

- Un dizionario è una struttura dati astratta che contiene elementi accessibili direttamente, data la loro chiave
- Offerto da alcuni linguaggi di programmazione come tipo base
- Assumiamo che le chiavi siano numeri naturali
  - Nel caso non lo siano, è sufficiente considerare la loro rappresentazione binaria il corrispettivo numero
- Le operazioni supportate sono Insert, Delete e Search
- E' possibile implementare un dizionario con diverse strutture dati concrete

## Un primo approccio

- Nel caso in cui le *possibili* chiavi siano un numero limitato un'implementazione di un dizionario è un vettore di puntatori
- Le chiavi vengono usate come indice del vettore
- Le operazioni sul dizionario sono implementate come:
  - $\text{Insert}(D, e): D[e.\text{key}] \leftarrow e$
  - $\text{Delete}(D, e): D[e.\text{key}] \leftarrow \text{NIL}$
  - $\text{Search}(D, e.\text{key}): \text{return } D[e.\text{key}]$
- Complessità computazionale:  $\Theta(1)$  per tutte le azioni
- Complessità spaziale:  $\mathcal{O}(|\mathbf{D}|)$ , con  $\mathbf{D}$  il dominio delle chiavi
  - Estremamente oneroso se il dominio è molto ampio



## Maggior efficienza in spazio

- Una tabella hash implementa un dizionario con una complessità in memoria pari al numero di chiavi *effettivamente presenti*
  - Il dominio delle chiavi  $\mathbf{D}$  essere arbitrariamente grande/infinito
- Approccio tipico: prealloco spazio per  $m$  chiavi
  - Rialloco solo quando devo inserire  $n > m$  chiavi
- Idea principale: uso come indice della tabella il risultato del calcolo di una funzione della chiave  $h(k)$ 
  - $h(\cdot) : \mathbf{D} \rightarrow \{0, \dots, m - 1\}$  è detta funzione di hash

# Tabelle Hash

## Efficienza

- Se il calcolo di  $h$  è  $\mathcal{O}(k)$  la tabella di hash ideale ha la stessa efficienza temporale del dizionario fatto con il vettore di  $|\mathbf{D}|$  puntatori

## Il problema delle collisioni

- Idealmente,  $h$  dovrebbe mappare ogni chiave su di un distinto elemento del suo codominio
  - Impossibile! Per costruzione  $|\mathbf{D}| \gg m$  (specie se  $|\mathbf{D}| = \infty$ )
- Chiamiamo *collisione* ogniqualvolta per  $k_1, k_2; k_1 \neq k_2$  abbiamo che  $h(k_1) = h(k_2)$

## Indirizzamento chiuso (open hashing)

- Ogni riga della tabella (bucket) contiene la testa di una lista al posto del puntatore ad un singolo elemento
- Nel caso di collisione, l'elemento nuovo viene aggiunto in testa alla lista ( $\Theta(1)$ )
- Per cercare/cancellare un elemento di chiave  $k$ , è necessario cercare nell'intera lista di quelli del bucket  $h(k)$

# Gestione delle collisioni

## Indirizzamento aperto (closed hashing)

- In caso di collisione si seleziona secondo una regola deterministica un altro bucket (procedimento di ispezione)
- Nel caso non si trovino bucket vuoti:
  - L'inserimento semplicemente fallisce  $\Theta(1)$
  - Si rialloca una tabella più grande e si ri-inseriscono tutti gli elementi della vecchia nella nuova (re-hashing), incluso il nuovo  $\Theta(n)$
- Si modifica la procedura di ricerca, affinché, se l'elemento non viene trovato nel suo bucket, essa effettui la stessa ispezione
- La cancellazione è effettuata inserendo un opportuno valore (tombstone) che non corrisponde ad alcuna chiave

# Procedure di ispezione

## Ispezione lineare (Linear probing) e clustering

- Il metodo di ispezione più semplice è l'*ispezione lineare*
  - Dato  $h(k, 0) = a$  il bucket dove avviene la collisione al primo ( $i = 0$ ) tentativo di inserimento, si sceglie  $h(k, i) = a + c \cdot i$  come bucket candidato per l' $i$ -esimo inserimento
- Problema: se ci sono molte collisioni su un dato bucket, peggiorerà la probabilità di collisione in tutte le vicinanze
  - Il fenomeno è detto di *clustering* delle collisioni
  - Per alcune scelte di  $h$ , il peggiorare delle prestazioni dovuto al clustering dell'ispezione lineare è molto forte
  - É possibile avere clustering di dimensione logaritmica nella dimensione della tabella, effettuando rehashing molto prima che sia piena

# Procedure di ispezione

## Ispezione quadratica (Quadratic probing)

- Per mitigare il fenomeno del clustering è possibile utilizzare il criterio di ispezione quadratica:  
$$h(k, i) = a + c_1i + c_2i^2 \pmod n$$
  - Viene evitato il clustering banale nell'intorno di alcuni elementi
  - Non è più garantito a priori che la sequenza di ispezione tocchi tutte le celle: potresti dover fare rehashing a tabella non piena
- Chiavi con la stessa posizione iniziale generano ancora clustering: hanno la stessa sequenza di ispezione!

# Una sequenza ideale

Per tabelle di dimensione  $n = 2^m$

- Lemma:  $h(k, i) = a + \frac{1}{2}i + \frac{1}{2}i^2$  genera tutti i valori in  $[0, n - 1]$
- Dimostrazione: (Per assurdo) Esistono  $0 < p < q < n - 1$  tali che  $\frac{1}{2}p + \frac{1}{2}p^2 = \frac{1}{2}q + \frac{1}{2}q^2 \pmod n \Rightarrow p + p^2 = q + q^2 \pmod{2n}$
- Fattorizzando abbiamo  $(q - p)(p + q + 1) = 0 \pmod{2n}$ 
  - Se  $(q - p) = 0 \pmod{2n} \Rightarrow q = p \not\equiv$
  - $(p + q + 1) = 0 \pmod{2n}$ : dati i range possibili  $0 < p < q < n - 1$  la somma è  $\in [1, 2n - 2] \not\equiv$
  - $(q - p)(p + q + 1) = 0 \pmod{2n}$ , ma  $(q - p) \not\equiv 0 \pmod{2n}$  e  $(p + q + 1) \not\equiv 0 \pmod{2n}$ :  $(q - p) - (p + q + 1) = 2p + 1$ , quindi almeno uno tra  $(q - p)$  e  $(p + q + 1)$  è dispari
  - Essendo  $n = 2^m$  il fattore pari è multiplo di  $2n$ , ma  $(q - p) \leq n - 1$  e  $(p + q + 1) \leq 2n - 2 \not\equiv$



## Un'ispezione dipendente dalla chiave

- Definiamo  $h(k, i) = h_1(k) + h_2(k)i \bmod n$ : il passo di ispezione dipende dalla chiave
- Per essere sicuro di ispezionare tutti i bucket,  $h_2(k)$  deve essere coprimo con  $n$  :
  - Per  $n = 2^m$  basta fare sì che  $h_2$  generi solo numeri dispari
  - Se  $m$  è primo, basta fare sì che  $h_2$  generi un numero  $< m$
  - N.B.:  $h_2$  non deve mai dare zero, altrimenti la sequenza di ispezione degenera



## Ipotesi di Hashing Uniforme Semplice (IHUS)

- Una opportuna scelta di  $h$  fa sì che ogni chiave abbia la stessa probabilità  $\frac{1}{n}$  di finire in una qualsiasi delle  $n$  celle
- Come fare “la scelta opportuna”? Dipende dalla distribuzione delle chiavi da inserire...

## Metodo della divisione

- Un metodo semplice è  $h(k) = k \bmod n$ 
  - Va evitato  $m = 2^i$ :  $h(k)$  dipende solo dai bit meno significativi
- Un'idea ragionevole è  $n$  primo, vicino ad una potenza di 2
  - Primi di Fermat: hanno forma  $2^i + 1$ , e.g., 17, 257, 65537
  - Primi di Mersenne: hanno forma  $2^i - 1$ , e.g., 127, 8191, 131071

# Efficienza computazionale

## Metodo della moltiplicazione

- Un metodo semplice è  $h(k) = \lfloor n(\alpha k - \lfloor \alpha k \rfloor) \rfloor$  con  $\alpha$  scelto come una costante  $\in \mathbb{R}$
- In questo caso, la dimensione della tabella  $n$  non è critica
  - Spesso si prende  $n = 2^m$  in modo da effettuare le moltiplicazioni con un semplice shift
- Una scelta possibile per  $A$  è  $\frac{\sqrt{5}-1}{2}$  (proposto da Knuth): dà buoni risultati in pratica
- Un modo pratico di calcolare  $h(k)$  in C, nota la larghezza di parola del calcolatore (e.g. 32b) è calcolare  
`k * (uint32_t)((double)A * ((uint64_t)1 << 32))`
- La porzione in rosso è costante e può essere precalcolata

## Stime di costo

- Caso pessimo: tutti gli elementi collidono dando origine ad una lista (open hashing) o sequenza di ispezione (closed hashing) lunga  $n$  elementi: Insert/Delete/Search in  $\mathcal{O}(n)$
- Chiamiamo fattore di carico  $\alpha = \frac{n}{m}$ ,  $0 \leq \alpha \leq \frac{|D|}{m}$
- Sotto l'IHUS, per l'open hashing abbiamo che:
  - La lunghezza media di una lista è il fattore di carico
  - Il tempo medio per cercare una chiave non presente è  $\Theta(1 + \alpha)$
  - Il tempo medio per cercare una chiave presente è sempre  $\Theta(1 + \alpha)$  (risultato del calcolo del valor medio del numero di oggetti aggiunti al bucket di  $x$  dopo che  $x$  è stato inserito)
- In pratica, se il fattore di carico non è eccessivo tutte le operazioni sono  $\mathcal{O}(1)$  in media

## Closed hashing

- Il tempo per trovare un elemento dipende anche dalla sequenza di ispezione
- Ipotesi di hashing uniforme: generalizziamo la IHUS dicendo che tutte le sequenze di ispezione sono equiprobabili
- Il fattore di carico è sempre:  $0 \leq \alpha \leq 1$  (al massimo un oggetto per bucket)
- Se consideriamo la v.a.  $\mathcal{X}$  che modella il numero di passi di ispezione fatti senza trovare il valore desiderato, abbiamo  $\Pr(\mathcal{X} \geq i) = \alpha^{i-1}$  il cui valor medio su  $i$  è  $\frac{1}{1-\alpha}$
- Il numero medio di tentativi prima di trovare un elemento desiderato è ricavato assumendo di trovarlo al  $j$ -esimo tentativo e mediando il numero di insuccessi su tutte le  $n$  chiavi presenti in tabella: si ottiene  $\frac{1}{\alpha} \log\left(\frac{1}{1-\alpha}\right)$

# Hashing universale

## Prevenire casi pessimi indotti

- Abbiamo finora presunto che le collisioni fossero accidentali
- Se vengono indotte da un utente che conosce la funzione di hash ed inserisce una sequenza di elementi ad-hoc? Accessi in  $\mathcal{O}(n)$ ! Casi pratici: PHP, filesystem ext3/ext4
- Come evitarlo? Scegliendo una funzione di hash casualmente all'interno di una **famiglia** di buone funzioni
- Si può dimostrare che  $h_{a,b}(k) = ((ak + b) \bmod p) \bmod n$  con  $p$  primo,  $p > n$ , per qualunque  $a, b \in \mathbb{Z} \setminus 0$  distribuisce uniformemente le chiavi nella tabella
- É sufficiente scegliere casualmente  $a$  e  $b$  all'interno del programma, per ogni istanza della tabella

## Una tabella di hash, senza tabella

- Consideriamo il caso in cui il codominio di  $h(\cdot)$  sia enorme (e.g. stringhe da 256 bit  $\rightarrow$  codominio da  $2^{256}$  elem.)
- Con una buona  $h$  (che rispetti l'IHUS), servono moltissime chiavi prima di vedere una collisione
- Non potrò mai materializzare la tabella di hash, ma l' hash di un valore funge da “etichetta unica” del valore stesso
- E' importante che non si possano trovare collisioni o preimmagini (dato  $h(k)$ , trovare  $k$ ) in tempo utile
- Caso pratico: SHA-2-256:  $D = \{0, 1\}^{256}$ , codominio  $\{0, 1\}^{256}$