Strutture dati - Parte 1

Dipartimento di Elettronica, Informazione e Bioingegneria
Politecnico di Milano

8 maggio 2024

Strutture dati

Organizzare i dati processati

- Algoritmi efficienti necessitano di poter accedere, modificare, cancellare i dati da una collezione in modo efficiente
- 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 $\rightarrow$ strutture dati più evolute
- Queste strutture possono usare etichette per identificare un oggetto: le etichette sono dette *chiavi*
- Analizziamo l' efficienza delle strutture dati quantificando quella delle operazioni che effettuiamo su di esse

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
- 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 copia degli elementi esistenti)

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 alla posizione di inserimento
- 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: $\mathcal{O}(n)$

Operazioni tipiche su strutture dati

Interrogare la struttura

- `Search(S,k)`: restituisce il riferimento a e con `e.key`=k in S, NIL se k non è contenuto in S
- `Minimum(S)`: se le chiavi sono ordinate, restituisce e con la chiave minima
- `Maximum(S)`: come sopra, ma la chiave ha valore massimo
- `Successor(S,k)`: restituisce e t.c. `e.key` è la più piccola tra le chiavi $>$k
- `Predecessor(S,k)`: come sopra, ma considerando il predecessore

Modificare la struttura

- `Insert(S,e)`: Inserisce un oggetto e nella struttura
- `Delete(S,e)`: Cancella un oggetto e dalla struttura

Pile (Stack)

Una struttura dati familiare

- Una pila è una struttura dati con le seguenti operazioni:
  - `Push(S,e)`: aggiunge l'elemento in cima alla pila
  - `Pop(S)`: restituisce l'elemento in cima alla pila cancellandolo
  - `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:
  - `Push(S,e)`: inserisci in testa alla lista $\mathcal{O}(1)$
  - `Pop(S)`: restituisci il primo elemento della lista, cancellandolo dalla stessa $\mathcal{O}(1)$
  - `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)
  - `Push(S,e)`: se c'è spazio, Incrementa ToS, salva e in `A[ToS]`: $\mathcal{O}(1)$; se manca spazio rifiuta $\mathcal{O}(1)$ o rialloca $\mathcal{O}(n)$
  - `Pop(S)`: restituisci `A[ToS]` corrente, decrementa `ToS`: $\mathcal{O}(1)$
  - `Empty(S)`: Restituisci `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
  - $\rightarrow$ 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 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`$\stackrel{?}{=}$`tail`: $\mathcal{O}(1)$

Code (Queues)

Realizzazione con vettore

- Salviamo i dati in un vettore A, lungo l, 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 $\bmod l$
  - `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)`: $\Theta(n)$

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 testa, 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[tail]` corrente, decrementa n, decrementa `tail`: $\mathcal{O}(1)$
  - `PushFront(Q,e)`: se $n < l$, decrementa e `head`, inserisci l'elemento in $A[\texttt{head}]$, 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`$\stackrel{?}{=}$`tail` in $\mathcal{O}(1)$
- 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 fornito alla `Delete(L,e)` per riferimento è $O(1)$: $e.prev.next \leftarrow e.next$; $e.next.prev \leftarrow e.prev$

Dizionari

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 (Python)
- Nei nostri esempi, assumiamo che le chiavi siano numeri naturali
  - Se non lo sono, è sufficiente considerare la loro rappresentazione binaria come il corrispettivo numero
- Le operazioni supportate da un dizionario sono `Insert`, `Delete` e `Search`
- E'possibile implementare un dizionario con diverse strutture dati concrete

Dizionari

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:
  - `Insert(D,e)`: `D[e.key]` $\leftarrow$ `e`
  - `Delete(D,e)`: `D[e.key]` $\leftarrow$ `NIL`
  - `Search(D,e.key)`: **return** `D[e.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

Tabelle Hash

Maggior efficienza in spazio

- Una tabella hash implementa un dizionario con una complessità in memoria pari al numero di chiavi m per cui è *effettivamente presente* un valore
  - Il dominio delle chiavi $\mathbf{D}$ può essere arbitrariamente grande/infinito
- Approccio tipico: prealloco spazio per m chiavi
  - Rialloco solo quando devo inserire più di m chiavi
- Idea principale: uso come indice (indirizzo) della tabella il risultato del calcolo di una funzione della chiave detta funzione di hash $h(k)$
  - $h(\cdot) : \mathbf{D} \to \{0, \ldots, m-1\}$ serve a trasformare una chiave arbitraria in un indirizzo ammissibile (tra 0 e $m-1$)

Tabelle Hash

Efficienza

- Se il calcolo di $h(\cdot)$ è $\mathcal{O}(1)$ la tabella di hash ideale ha la stessa efficienza temporale del dizionario realizzato con un vettore di $|\mathbf{D}|$ puntatori

Il problema delle collisioni

- Idealmente, h mappa ogni chiave su di un distinto elemento del suo codominio
  - Impossibile! Per costruzione $|\mathbf{D}| \gg m$ (specie se $|\mathbf{D}| = \infty$)
- Chiamiamo *collisione* i casi in cui, date due chiavi k_1, k_2; con $k_1 \neq k_2$ abbiamo che $h(k_1) = h(k_2)$
  - Dobbiamo gestire le collisioni, oppure due elementi con chiavi distinte finiranno allo stesso indirizzo

Gestione delle collisioni

Indirizzamento chiuso (a.k.a. open hashing, chaining)

- 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 $h(k)$

Efficienza computazionale

Stime di costo

- Caso pessimo: tutti gli elementi collidono dando origine ad una lista (open hashing) lunga m elementi: `Insert`/`Delete`/`Search` in $\mathcal{O}(m)$
- Chiamiamo fattore di carico $\alpha = \frac{n}{m}$, $0 \leq \alpha \leq \frac{|\mathbf{D}|}{m}$
  - α può essere più grande di $1 \rightarrow$ tutti i bucket sono pieni

Ipotesi di Hashing Uniforme Semplice (IHUS)

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

Efficienza computazionale

Stime di costo

- Assumiamo 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 $1+\frac{\alpha}{2}-\frac{\alpha}{2n}$
- Se il fattore di carico non è eccessivo tutte le operazioni sono $\mathcal{O}(1)$ in media
  - In pratica: si decide un fattore di carico e si rialloca quando la tabella eccede

Gestione delle collisioni - 2

Indirizzamento aperto (a.k.a., open addressing, closed hashing)

- In caso di collisione si seleziona secondo una regola deterministica l'indirizzo di un altro bucket di destinazione (procedimento di ispezione)
- Nel caso non si trovino bucket vuoti:
  - L'inserimento semplicemente fallisce $\Theta(m)$
  - Si rialloca una tabella più grande, vuota, e si ri-inseriscono tutti gli elementi della vecchia nella nuova (ricalcolando la loro hash, re-hashing), incluso il nuovo $\Theta(n)$
- Si modifica la procedura di ricerca, affinchè, 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 $h(k, i) = a + c \cdot i \bmod m$ 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 primario* 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: $h(k, i) = a + c_1 i + c_2 i^2 \bmod m$
  - 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: potrei dover fare rehashing a tabella non piena
- Rimuove il clustering primario
- Chiavi con la stessa posizione iniziale generano ancora clustering: hanno la stessa sequenza di ispezione (clustering secondario)

Una sequenza ideale

Per tabelle di dimensione $m = 2^x$

- Lemma: $h(k,i) = a + \frac{1}{2}i + \frac{1}{2}i^2$ genera tutti i valori in $[0, m-1]$
- Dimostrazione: (Per assurdo) Esistono $0 < p < q < m-1$ tali che $\frac{1}{2}p + \frac{1}{2}p^2 = \frac{1}{2}q + \frac{1}{2}q^2 \bmod m \Rightarrow p + p^2 = q + q^2 \bmod 2m$
- Fattorizzando abbiamo $(q-p)(p+q+1) = 0 \bmod 2m$
  - Se $(q-p) = 0 \bmod 2m \Rightarrow q = p$ ↯
  - $(p+q+1) = 0 \bmod 2m$: dati i range possibili $0<p<q<m-1$ la somma è $\in [1, 2m-1]$ ↯
  - $(q-p)(p+q+1) = 0 \bmod 2m$, ma $(q-p) \neq 0 \bmod 2m$ e $(p+q+1) \neq 0 \bmod 2m$. Osserviamo che almeno uno tra $(q-p)$ e $(p+q+1)$ è dispari
  - Essendo $m = 2^x$ solo il fattore pari può essere multiplo di $2m$, ma $(q-p) \leq m-1$ e $(p+q+1) \leq 2m-2$ ↯ ■

Doppio Hashing

Un'ispezione dipendente dalla chiave

- Definiamo $h(k, i) = h_1(k) + h_2(k)i \bmod m$: il passo di ispezione dipende dalla chiave
- Per essere sicuro di ispezionare tutti i bucket, $h_2(k)$ deve essere coprimo con m :
  - Per $m = 2^x$ 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

Stime di costo

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 $\Pr(\mathcal{X} \geq i) \leq \alpha^{i-1}$ il cui valor medio su i è $\frac{1}{1-\alpha}$: cercare un valore *non presente* costa $\frac{1}{1-\alpha}$ (in media)
- 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 $\frac{1}{\alpha}\log(\frac{1}{1-\alpha})$

Possibili funzioni di hash

Metodo della divisione

- Un metodo semplice è $h(k) = k \bmod m$
  - Distribuzione di $h(k)$ non uniforme in $\{0, \ldots, m-1\}$
  - Va evitato se $m = 2^i$: $h(k)$ dipende solo dai bit meno significativi
- Distrib. (quasi) uniforme: m 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.,$127, 8191, 131071$

Possibili funzioni di hash

Metodo della moltiplicazione

- Un metodo semplice è $h(k) = \lfloor m(\alpha k - \lfloor \alpha k \rfloor) \rfloor$ con α scelto costante $\in \mathbb{R}$
  - n.b.: $\alpha k - \lfloor \alpha k \rfloor$ è la parte frazionaria di αk
- In questo caso, la dimensione della tabella m non è critica
  - Spesso $m = 2^x$ in modo da effettuare le moltiplicazioni con un semplice shift
- Una scelta possibile per α è $\frac{\sqrt{5}-1}{2}$, rappresentato a virgola fissa (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 $2^{32}\alpha k$ su 64 bit, moltiplicare per k e troncare a 32 bit

```
(uint32_t)( k * (double)ALPHA * ((uint64_t)1 << 32))
```

- La porzione in rosso è costante e può essere precalcolata

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 certi 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 $h_{a,b}(k) = ((ak + b) \bmod p) \bmod m$ con p primo, $p > m$, 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

Hashing crittografico

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
- È importante che non si possano trovare collisioni o preimmagini (dato $h(k)$, trovare k) in tempo utile
- Caso pratico: `SHA-2-256`: $\mathbf{D} = \{0,1\}^{2^{64}}$, codominio $\{0,1\}^{256}$