Algoritmi e Strutture Dati

Analisi ammortizzata

Alberto Montresor

Università di Trento

2020/11/22

This work is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.

Sommario

Introduzione

Analisi ammortizzata

Una tecnica di analisi di complessità che valuta il tempo richiesto per eseguire, nel caso pessimo, una sequenza di operazioni su una struttura dati

- Esistono operazioni più o meno costose.
- Se le operazioni più costose sono poco frequenti, allora il loro costo può essere ammortizzato dalle operazioni meno costose

Importante differenza

- Analisi caso medio: Probabilistica, su singola operazione
- Analisi ammortizzata: Deterministica, su operazioni multiple, caso pessimo

Metodi per l'analisi ammortizzata

Metodo dell'aggregazione

- Si calcola la complessità $T(n)$ per eseguire n operazioni in sequenza nel caso pessimo
- Tecnica derivata dalla matematica

Metodo degli accantonamenti

- Alle operazioni vengono assegnati costi ammortizzati che possono essere maggiori/minori del loro costo effettivo
- Tecnica derivata dall'economia

Metodo del potenziale

- Lo stato del sistema viene descritto con una funzione di potenziale
- Tecnica derivata dalla fisica

Esempio – Contatore binario

Contatore binario

- Contatore binario di k bit con un vettore A di booleani 0/1
- Bit meno significativo in $A[0]$, bit più significativo in $A[k-1]$
- Valore del contatore: $$x = \sum_{i=0}^{k-1} A[i]2^i$$
- Operazione increment() che incrementa il contatore di 1

---

increment(**int**[ ] A, **int** k)

---

int $i = 0$
while $i < k$ **and** $A[i] == 1$ **do**
  $A[i] = 0$
  $i = i + 1$
if $i < k$ **then**
  $A[i] = 1$

Esempio

x	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	1	0
3	0	0	0	0	0	0	1	1
4	0	0	0	0	0	1	0	0
5	0	0	0	0	0	1	0	1
6	0	0	0	0	0	1	1	0
7	0	0	0	0	0	1	1	1
8	0	0	0	0	1	0	0	0
9	0	0	0	0	1	0	0	1
10	0	0	0	0	1	0	1	0
11	0	0	0	0	1	0	1	1
12	0	0	0	0	1	1	0	0
13	0	0	0	0	1	1	0	1
14	0	0	0	0	1	1	1	0
15	0	0	0	0	1	1	1	1
16	0	0	0	1	0	0	0	0

Analisi ammortizzata – Metodo dell'aggregazione

Metodo dell'aggregazione

- Si calcola la complessità $T(n)$ per eseguire n operazioni in sequenza nel caso pessimo
- Si calcola il costo ammortizzato $T(n)/n$ come media su n operazioni

- Sequenza: si considera l'evoluzione della struttura dati data una sequenza di operazioni
- Caso pessimo: si considera la peggior sequenza di operazioni
- Aggregazione: sommatoria delle complessità individuali

Analisi ammortizzata – Metodo dell'aggregazione

- Qual è il costo $T(n)$ per eseguire n operazioni (caso pessimo)?
- Qual è il costo ammortizzato $T(n)/n$?

---

increment(**int**[ ] A, **int** k)

---

int $i = 0$
while $i < k$ **and** $A[i] == 1$ **do**
    $A[i] = 0$
    $i = i + 1$
if $i < k$ **then**
    $A[i] = 1$

Analisi ammortizzata – Metodo dell'aggregazione

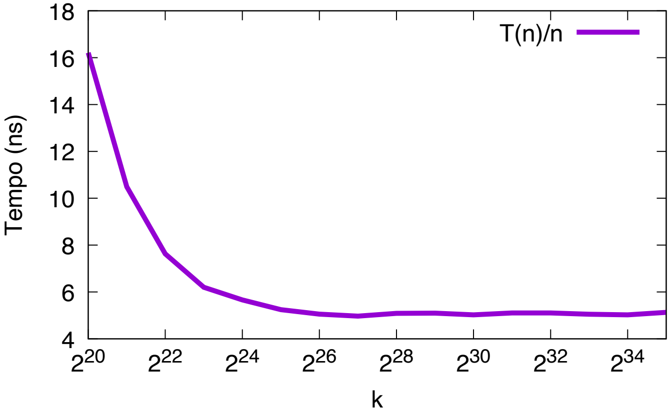
- Qual è il costo $T(n)$ per eseguire n operazioni (caso pessimo)?
- Qual è il costo ammortizzato $T(n)/n$?

Analisi "grossolana"

- Sono necessari $k = \lceil \log(n+1) \rceil$ bit per rappresentare n
- Una chiamata $\mathsf{increment}()$ richiede tempo $O(k)$ nel caso pessimo
- Costo di n operazioni: $T(n) = O(nk)$
- Costo di 1 operazione: $T(n)/n = O(k) = O(\log n)$

Analisi ammortizzata – Metodo dell'aggregazione

- Qual è il costo $T(n)$ per eseguire n operazioni (caso pessimo)?
- Qual è il costo ammortizzato $T(n)/n$?



- $n = 2^k$ operazioni
- $k \in [20, 35]$
- Eseguito in Java

Quanto costano n operazioni di incremento?

Considerazioni per un'analisi più stretta

- Possiamo osservare che il tempo necessario ad eseguire l'intera sequenza è proporzionale al numero di bit che vengono modificati
- Quanti bit vengono modificati?

Esempio

x	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	#bit
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	2
3	0	0	0	0	0	0	1	1	1
4	0	0	0	0	0	1	0	0	3
5	0	0	0	0	0	1	0	1	1
6	0	0	0	0	0	1	1	0	2
7	0	0	0	0	0	1	1	1	1
8	0	0	0	0	1	0	0	0	4
9	0	0	0	0	1	0	0	1	1
10	0	0	0	0	1	0	1	0	2
11	0	0	0	0	1	0	1	1	1
12	0	0	0	0	1	1	0	0	3
13	0	0	0	0	1	1	0	1	1
14	0	0	0	0	1	1	1	0	2
15	0	0	0	0	1	1	1	1	1
16	0	0	0	1	0	0	0	0	5

Esempio

x	$A[7]$	$A[6]$	$A[5]$	$A[4]$	$A[3]$	$A[2]$	$A[1]$	$A[0]$	#bit
0	0	0	0	0	0	0	0	0	
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	2
3	0	0	0	0	0	0	1	1	1
4	0	0	0	0	0	1	0	0	3
5	0	0	0	0	0	1	0	1	1
6	0	0	0	0	0	1	1	0	2
7	0	0	0	0	0	1	1	1	1
8	0	0	0	0	1	0	0	0	4
9	0	0	0	0	1	0	0	1	1
10	0	0	0	0	1	0	1	0	2
11	0	0	0	0	1	0	1	1	1
12	0	0	0	0	1	1	0	0	3
13	0	0	0	0	1	1	0	1	1
14	0	0	0	0	1	1	1	0	2
15	0	0	0	0	1	1	1	1	1
16	0	0	0	1	0	0	0	0	5
#bit	0	0	0	1	2	4	8	16	

Contatore binario – Metodo dell'aggregazione

Dalla simulazione si vede che:

- $A[0]$ viene modificato ogni 1 incremento
- $A[1]$ viene modificato ogni 2 incrementi
- $A[2]$ viene modificato ogni 4 incrementi
- ...
- $A[i]$ viene modificato ogni 2^i incrementi

Analisi ammortizzata

- Costo totale: $T(n) = \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \leq n \sum_{i=0}^{k-1} \frac{1}{2^i} \leq n \sum_{i=0}^{+\infty} \left(\frac{1}{2}\right)^i = 2n$
- Costo ammortizzato: $T(n)/n \leq 2n/n = 2 = O(1)$

Analisi ammortizzata – Metodo degli accantonamenti

- Si assegna un costo ammortizzato *potenzialmente* distinto ad ognuna delle operazioni possibili
- Il costo ammortizzato può essere diverso dal costo effettivo
  - Le operazioni meno costose vengono caricate di un costo aggiuntivo detto credito

$$\mathit{costo\ ammortizzato} = \mathit{costo\ effettivo} + \mathit{credito\ prodotto}$$

  - I crediti accumulati sono usati per pagare le operazioni più costose

$$\mathit{costo\ ammortizzato} = \mathit{costo\ effettivo} - \mathit{credito\ consumato}$$

Analisi ammortizzata – Metodo degli accantonamenti

Obiettivi

- Dimostrare che la somma dei costi ammortizzati a_i è un limite superiore ai costi effettivi c_i:

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} a_i$$

- Dimostrare che il valore così ottenuto è "poco costoso"

Alcuni punti da ricordare:

- La dimostrazione deve valere per tutte le sequenze (caso pessimo)
- Il credito dopo l'operazione t-esima è espresso dalla seguente formula ed è sempre positivo

$$\sum_{i=1}^{t} a_i - \sum_{i=1}^{t} c_i \geq 0$$

Contatore binario – Metodo degli accantonamenti

- Costo effettivo dell'operazione increment(): d
  - Dove d è il numero di bit che cambiano valore
- Costo ammortizzato dell'operazione increment(): 2
  - 1 per cambio di un bit da 0 a 1 (costo effettivo)
  - 1 per il futuro cambio dello stesso bit da 1 a 0
- Ne consegue che:
  - In ogni istante, il credito è pari al numero di bit 1 presenti
  - Costo totale: $O(n)$
  - Costo ammortizzato: $O(1)$

Analisi ammortizzata – Metodo del potenziale

Funzione di potenziale

- Una funzione di potenziale Φ associa ad uno stato S della struttura dati la "quantità di lavoro" $\Phi(S)$ che è stato contabilizzato nell'analisi ammortizzata, ma non ancora eseguito
- In altre parole, $\Phi(S)$ rappresenta la quantità di energia potenziale "immagazzinata" in quello stato

Costo ammortizzato

Costo ammortizzato = Costo effettivo + Variazione di potenziale.

$$a_i = c_i + \Phi(S_i) - \Phi(S_{i-1})$$

dove S_i è lo stato associato alla i-esima operazione.

Analisi ammortizzata – Metodo del potenziale

Costo ammortizzato, sequenza di n operazioni

$$
\begin{aligned}
A &= \sum_{i=1}^{n} a_i \\
&= \sum_{i=1}^{n} (c_i + \Phi(S_i) - \Phi(S_{i-1})) \\
&= \sum_{i=1}^{n} c_i + \sum_{i=1}^{n} (\Phi(S_i) - \Phi(S_{i-1})) \\
&= C + \Phi(S_1) - \Phi(S_0) + \Phi(S_2) - \Phi(S_1) + \ldots + \Phi(S_n) - \Phi(S_{n-1}) \\
&= C + \Phi(S_n) - \Phi(S_0)
\end{aligned}
$$

Se la variazione di potenziale $\Phi(S_n) - \Phi(S_0)$ è non negativa,
il costo ammortizzato A è un limite superiore al costo reale

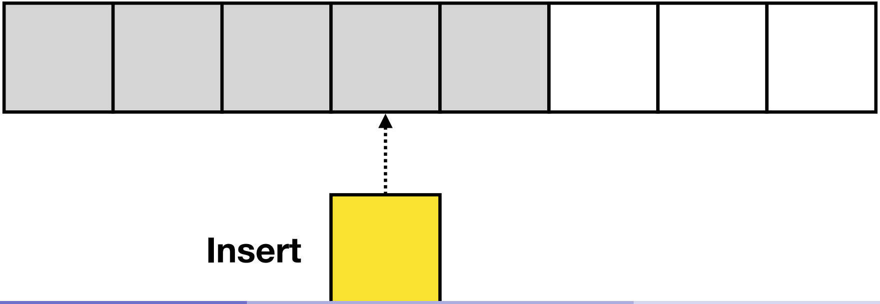
Contatore binario – Metodo del potenziale

- Scegliamo come funzione potenziale Φ il numero bit 1 presenti nel contatore
- Operazione increment():
  - t è il numero di bit 1 incontrati a partire dal meno significativo, prima di incontrare un bit 0
  - Costo effettivo: $1 + t$
  - Variazione di potenziale: $1 - t$
  - Costo ammortizzato: $1 + t + 1 - t = 2$
- All'inizio, zero bit accesi $\Rightarrow \Phi(S_0) = 0$
- Alla fine, $\Phi(S_n) \geq 0 \Rightarrow$ la differenza di potenziale è non negativa

Vettori dinamici – Espansione

Sequenze implementate tramite vettori dinamici

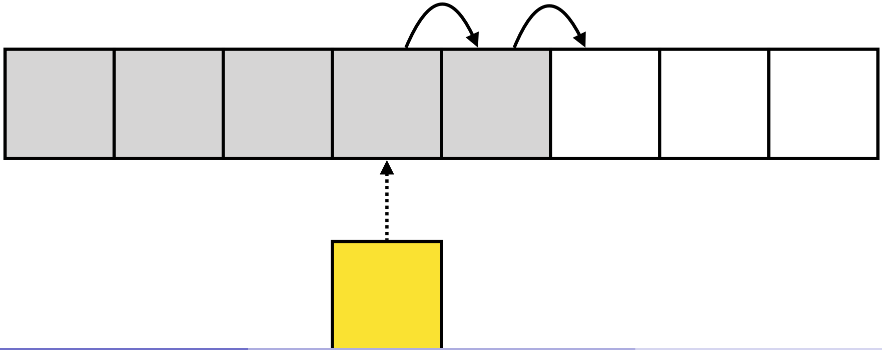
- Si alloca un vettore di una certa dimensione detta capacità
- L'inserimento di un elemento "in mezzo" ha costo $O(n)$
- Inserire un elemento "in fondo" alla sequenza (append) ha costo $O(1)$



Vettori dinamici – Espansione

Sequenze implementate tramite vettori dinamici

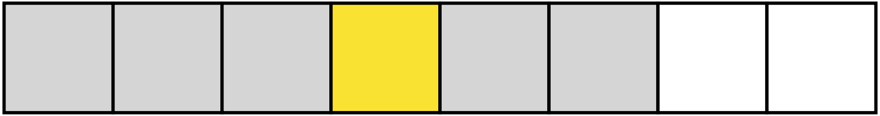
- Si alloca un vettore di una certa dimensione detta capacità
- L'inserimento di un elemento "in mezzo" ha costo $O(n)$
- Inserire un elemento "in fondo" alla sequenza (append) ha costo $O(1)$



Vettori dinamici – Espansione

Sequenze implementate tramite vettori dinamici

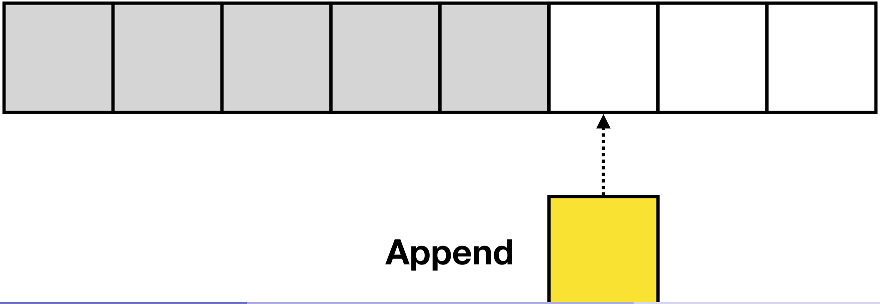
- Si alloca un vettore di una certa dimensione detta capacità
- L'inserimento di un elemento "in mezzo" ha costo $O(n)$
- Inserire un elemento "in fondo" alla sequenza (append) ha costo $O(1)$



Vettori dinamici – Espansione

Sequenze implementate tramite vettori dinamici

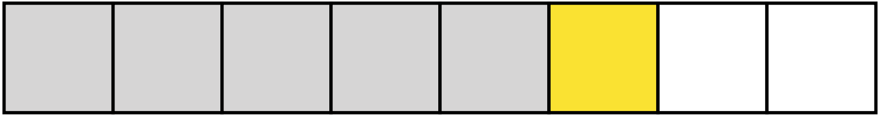
- Si alloca un vettore di una certa dimensione detta capacità
- L'inserimento di un elemento "in mezzo" ha costo $O(n)$
- Inserire un elemento "in fondo" alla sequenza (append) ha costo $O(1)$



Vettori dinamici – Espansione

Sequenze implementate tramite vettori dinamici

- Si alloca un vettore di una certa dimensione detta capacità
- L'inserimento di un elemento "in mezzo" ha costo $O(n)$
- Inserire un elemento "in fondo" alla sequenza (append) ha costo $O(1)$



Vettori dinamici – Espansione

Problema

- Non è noto a priori quanti elementi entreranno nella sequenza
- La capacità selezionata può rivelarsi insufficiente.

Soluzione

- Si alloca un vettore di capacità maggiore, si ricopia il contenuto del vecchio vettore nel nuovo e si rilascia il vecchio vettore
- Esempi: `java.util.Vector`, `java.util.ArrayList`

Vettori dinamici in Java

```java
private Object[] buffer = new Object[INITSIZE];

// Raddoppiamento
private void doubleStorage() {
  Object[] newb = new Object[2*buffer.length];
  System.arraycopy(buffer,0, newb,0, buffer.length);
  buffer = newb;
}

// Incremento fisso
private void incrementStorage() {
  Object[] newb = new Object[buffer.length+INCREMENT];
  System.arraycopy(buffer,0, newb,0, buffer.length);
  buffer = newb;
}
```

Vettori dinamici – Espansione

Domanda

Qual è il migliore fra i due?

- Raddoppiamento
- Incremento costante

Vettori dinamici – Espansione

Domanda

Qual è il migliore fra i due?

- Raddoppiamento
- Incremento costante

Dalla documentazione Java: ArrayList.add():

`As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.`

Vettori dinamici in Java – Quale approccio?

```
private Object[] buffer = new Object[INITSIZE];

// Raddoppiamento - Utilizzato in ArrayList (1.2)
private void doubleStorage() {
  Object[] newb = new Object[2*buffer.length];
  System.arraycopy(buffer,0, newb,0, buffer.length);
  buffer = newb;
}

// Incremento fisso - Utilizzato in Vector (1.0)
private void incrementStorage() {
  Object[] newb = new Object[buffer.length+INCREMENT];
  System.arraycopy(buffer,0, newb,0, buffer.length);
  buffer = newb;
}
```

Analisi ammortizzata, raddoppiamento del vettore

Costo effettivo di un'operazione `add()`:

$$c_i = \begin{cases} i & \exists k \in \mathbb{Z}_0^+ : i = 2^k + 1 \\ 1 & \text{altrimenti} \end{cases}$$

Assunzioni:

- Dimensione iniziale: 1
- Costo di scrittura di un elemento: 1

n	costo
1	1
2	$1 + 2^0 = 2$
3	$1 + 2^1 = 3$
4	1
5	$1 + 2^2 = 5$
6	1
7	1
8	1
9	$1 + 2^3 = 9$
10	1
11	1
12	1
13	1
14	1
15	1
16	1
17	$1 + 2^4 = 17$

Analisi ammortizzata, raddoppiamento del vettore

Costo effettivo di n operazioni add():

$$\begin{aligned}
T(n) &= \sum_{i=1}^{n} c_i \\
&= n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j \\
&= n + 2^{\lfloor \log n \rfloor + 1} - 1 \\
&\leq n + 2^{\log n + 1} - 1 \\
&= n + 2n - 1 = O(n)
\end{aligned}$$

Costo ammortizzato di un'operazione add():

$$T(n)/n = \frac{O(n)}{n} = O(1)$$

Analisi ammortizzata, incremento del vettore

Costo effettivo di un'operazione add():

$$c_i = \begin{cases} i & (i \bmod d) = 1 \\ 1 & \text{altrimenti} \end{cases}$$

Assunzioni:

- Incremento: d
- Dimensione iniziale: d
- Costo di scrittura di un elemento: 1

Nell'esempio:

- $d = 4$

n	costo
1	1
2	1
3	1
4	1
5	$1 + d = 5$
6	1
7	1
8	1
9	$1 + 2d = 9$
10	1
11	1
12	1
13	$1 + 3d = 13$
14	1
15	1
16	1
17	$1 + 4d = 17$

Analisi ammortizzata, incremento del vettore

Costo effettivo di n operazioni add():

$$\begin{aligned}
T(n) &= \sum_{i=1}^{n} c_i \\
&= n + \sum_{j=1}^{\lfloor n/d \rfloor} d \cdot j \\
&= n + d \sum_{j=1}^{\lfloor n/d \rfloor} j \\
&= n + d \frac{(\lfloor n/d \rfloor + 1)\lfloor n/d \rfloor}{2} \\
&\leq n + \frac{(n/d + 1)n}{2} = O(n^2)
\end{aligned}$$

Costo ammortizzato di un'operazione add():

$$T(n)/n = \frac{O(n^2)}{n} = O(n)$$

Reality check

Linguaggio	Struttura dati	Fattore espansione
GNU C++	`std::vector`	2.0
Microsoft VC++ 2003	`vector`	1.5
C#	`List`	2.0
Python	`list`	1.125
Oracle Java	`ArrayList`	2.0
OpenSDK Java	`ArrayList`	1.5

Non ancora convinti? Allocazione memoria

Domande

- Quanto memoria occupa un intero 32 bit se memorizzato in un vettore dinamico, nel caso pessimo/ottimo?
- Quanto memoria occupa un intero 32 bit se memorizzato in un elemento di una lista, nel caso pessimo/ottimo?

Non ancora convinti? Allocazione memoria

Domande

- Quanto memoria occupa un intero 32 bit se memorizzato in un vettore dinamico, nel caso pessimo/ottimo?
- Quanto memoria occupa un intero 32 bit se memorizzato in un elemento di una lista, nel caso pessimo/ottimo?

- Nel caso ottimo (vettore pieno): $4n$ byte
- Nel caso pessimo (vettore appena raddoppiato): $2 \cdot 4n$ byte

Non ancora convinti? Allocazione memoria

Domande

- Quanto memoria occupa un intero 32 bit se memorizzato in un vettore dinamico, nel caso pessimo/ottimo?
- Quanto memoria occupa un intero 32 bit se memorizzato in un elemento di una lista, nel caso pessimo/ottimo?

Alcuni dati interessanti (e inquietanti) relativi a Java:

- un `Object` vuoto richiede 16 byte in un'architettura a 64 bit
- l'allocazione di memoria avviene per multipli di 8 byte
- un oggetto contenente un intero e due reference (lista bidirezionale) richiede 32-40 byte (Java compressed references)

Non ancora convinti? String vs StringBuffer

```
public static void print(int dim)
{
  StringBuffer buffer = new StringBuffer();
  for (int i=0; i < dim; i++) {
    buffer.append('x');
  }

  String string = new String();
  for (int i=0; i < dim; i++) {
    string = string + 'x';
  }
}
```

Non ancora soddisfatti? String vs StringBuffer

dim	StringBuffer (ms)	String (ms)
1024	0	3
2048	1	7
4096	1	13
8192	1	22
16384	3	71
32768	4	285
65536	3	1130
131072	4	4847
262144	6	20853
524288	11	84982
1048576	24	400653

Non ancora soddisfatti? Somme stringhe in Python

Esempio

```
s = ""
for i in range(10000):
    s = s + "1"
```

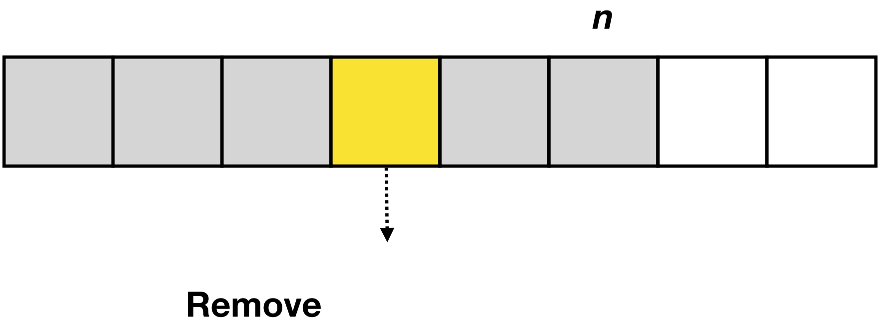
- In CPython, questo codice viene ottimizzato utilizzando `realloc` invece di `malloc` e ha costo lineare - fino ad un certo punto
- Oltre una certa dimensione, il costo torno ad essere quadratico
- Altre implementazioni di Python non seguono questo approccio

https://stackoverflow.com/questions/4435169/how-do-i-append-one-string-to-another-in-python
Credits: Michele Baldo

Vettori dinamici – Cancellazione

Domande

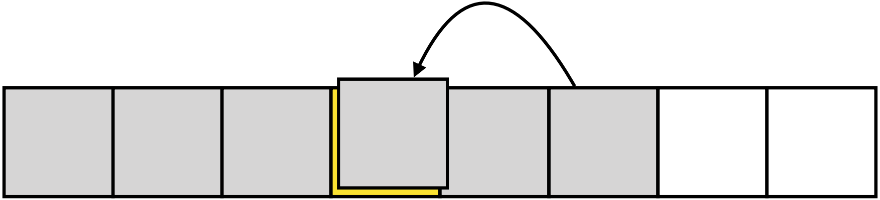
- Quanto costa togliere un elemento da un vettore?
- Quanto costa togliere un elemento da un vettore non ordinato?



Vettori dinamici – Cancellazione

Domande

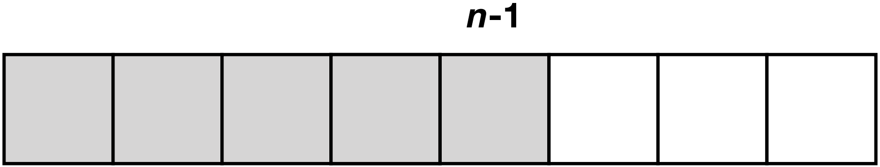
- Quanto costa togliere un elemento da un vettore?
- Quanto costa togliere un elemento da un vettore non ordinato?



Vettori dinamici – Cancellazione

Domande

- Quanto costa togliere un elemento da un vettore?
- Quanto costa togliere un elemento da un vettore non ordinato?



Vettori dinamici – Cancellazione

Contrazione

Per ridurre lo spreco di memoria, è opportuno contrarre il vettore quando il fattore di carico $\alpha = \mathit{dim}/\mathit{capacità}$ diventa troppo piccolo

- dim: numero di elementi attualmente presenti
- Contrazione $\rightarrow$ allocazione, copia, deallocazione

Domanda

Quale soglia per il fattore di carico?

Vettori dinamici – Cancellazione

Strategia naif

Una strategia che sembra ovvia è dimezzare la memoria quando il fattore di carico α diventa $\frac{1}{2}$

Dimostrare che questa strategia può portare ad un costo ammortizzato lineare

Vettori dinamici – Cancellazione

Strategia naif

Una strategia che sembra ovvia è dimezzare la memoria quando il fattore di carico α diventa $\frac{1}{2}$

Dimostrare che questa strategia può portare ad un costo ammortizzato lineare

Considerate la seguente sequenza di Inserimenti / Rimozioni in un vettore di capacità 8:

```
Ops: I I I I I R I R I R I R I
Dim: 1 2 3 4 5 4 5 4 5 4 5 4 5
Cap: 8 8 8 8 8 4 8 4 8 4 8 4 8
```

Vettori dinamici – Cancellazione

Qual è il problema?

- Non abbiamo un numero di inserimenti/rimozioni sufficienti per ripagare le espansioni/contrazioni.

Dobbiamo lasciar decrescere il sistema ad un fattore inferiore a $\alpha = \frac{1}{4}$

- Dopo un'espansione, il fattore di carico diventa $\frac{1}{2}$
- Dopo una contrazione, il fattore di carico diventa $\frac{1}{2}$

Analisi ammortizzata: usiamo una funzione di potenziale che:

- Vale 0 all'inizio e subito dopo una espansione o contrazione
- Cresce fino a raggiungere il numero di elementi presenti nella tavola quando α aumenta fino ad 1 o diminuisce fino ad 1/4

Vettori dinamici: contrazione

Funzione di potenziale

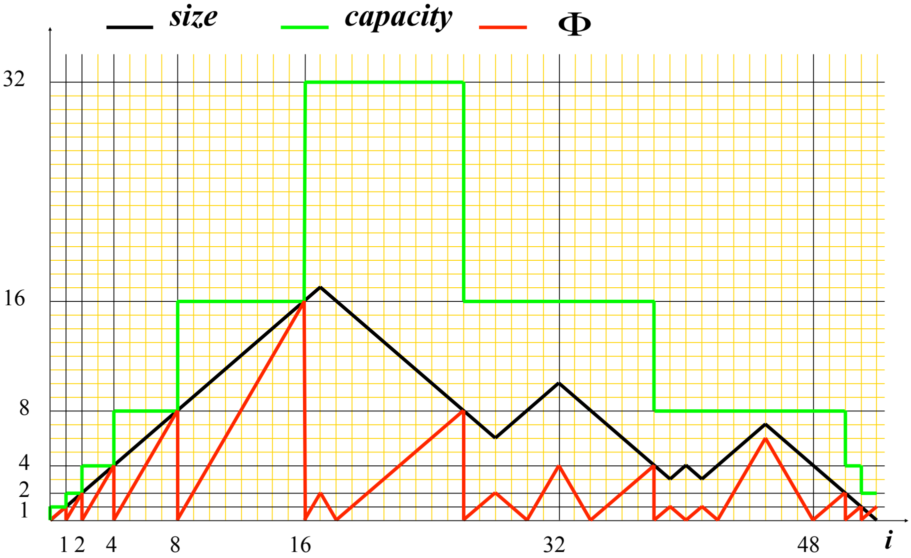
$$\Phi = \begin{cases} 2 \cdot \mathit{dim} - \mathit{capacità} & \alpha \geq \frac{1}{2} \\ \mathit{capacità}/2 - \mathit{dim} & \alpha \leq \frac{1}{2} \end{cases}$$

Alcuni casi esplicativi:

- $\alpha = \frac{1}{2}$ (dopo espansione/contrazione) $\Rightarrow \Phi = 0$
- $\alpha = 1$ (prima di espansione) $\Rightarrow \mathit{dim} = \mathit{capacità} \Rightarrow \Phi = \mathit{dim}$
- $\alpha = \frac{1}{4}$ (prima di contrazione) $\Rightarrow \mathit{capacità} = 4 \cdot \mathit{dim} \Rightarrow \Phi = \mathit{dim}$

In altre parole: immediatamente prima di espansioni e contrazioni il potenziale è sufficiente per “pagare” il loro costo

Vettori dinamici: contrazione



Vettori dinamici: contrazione

Se $\alpha \geq \frac{1}{2}$ il costo ammortizzato di un inserimento senza espansione è:

$$\begin{aligned}
a_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot \mathit{dim}_i - \mathit{capacità}_i) - (2 \cdot \mathit{dim}_{i-1} - \mathit{capacità}_{i-1}) \\
&= 1 + 2 \cdot (\mathit{dim}_{i-1} + 1) - \mathit{capacità}_{i-1} - 2 \cdot \mathit{dim}_{i-1} + \mathit{capacità}_{i-1} \\
&= 3
\end{aligned}$$

Se $\alpha = 1$ il costo ammortizzato di un inserimento con espansione è:

$$\begin{aligned}
a_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + \mathit{dim}_{i-1} + (2 \cdot \mathit{dim}_i - \mathit{capacità}_i) - (2 \cdot \mathit{dim}_{i-1} - \mathit{capacità}_{i-1}) \\
&= 1 + \mathit{dim}_{i-1} + 2 \cdot (\mathit{dim}_{i-1} + 1) - 2 \cdot \mathit{dim}_{i-1} - 2 \cdot \mathit{dim}_{i-1} + \mathit{dim}_{i-1} \\
&= 3
\end{aligned}$$

[*Esercizio: Altri casi per valori differenti di* α *e per contrazione*]

Conclusioni

Esempi di applicazione dell'analisi ammortizzata

- Espansione / contrazione di tabelle hash
- Insiemi disgiunti con euristica sul rango e compressione dei cammini
- Heap di Fibonacci
- ...

Ristrutturazione in tabelle Hash

- Non è conveniente che α cresca troppo
  - In particolare con la scansione interna
  - Ma vale anche per le liste di trabocco
- Sopra un soglia t_α prefissata (tipicamente 0.5-0.75)
  - Si alloca una nuova tabella di dimensione $2m$
  - Si reinseriscono tutte le chiavi presenti nella nuova tabella
- Risultato
  - Fattore di carico dimezzato (tipicamente 0.25)
  - Nessun elemento **deleted**
- Costi
  - Costo $O(m)$ per la ristrutturazione nel caso pessimo
  - Costo ammortizzato costante (vedi dimostrazione per vettori)

Reality check

Linguaggio	Tecnica	t_α	Note
Java 7 `HashMap`	Liste di trabocco basate su `LinkedList`	0.75	$O(n)$ nel caso pessimo Overhead: $16n + 4m$ byte
Java 8 `HashMap`	Liste di trabocco basate su RB Tree	0.75	$O(\log n)$ nel caso pessimo Overhead: $48n + 4m$ byte
C++ `sparse_hash`	Ind. aperto, scansione quadratica	?	Overhead: $2n$ bit
C++ `dense_hash`	Ind. aperto, scansione quadratica	0.5	X byte per chiave-valore $\Rightarrow$ 2-3X overhead
C++ STL `unordered_map`	Liste di trabocco basate su liste	1.00	MurmurHash
Python	Indirizzam. aperto, scansione quadratica	0.66	