Corso Python per docenti

Strutture dati

Alberto Montresor

Università di Trento

2021/02/12

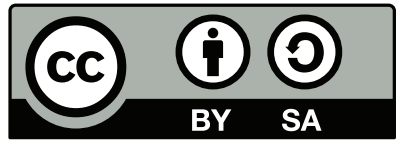


Table of contents

Vettori dinamici

Le liste Python sono implementate come vettori dinamici

- Un vettore di una certa dimensione (capacità iniziale) viene allocato
- Quando si inserisce un elemento all'inizio o in mezzo, tutti gli elementi devono essere mossi (costo $O(n)$)
- Quando si inserisce un elemento alla fine (append), il costo è $O(1)$ (scrivere l'elemento nel primo slot disponibile)

Problema:

- Non è chiaro quanti elementi verranno inseriti
- La capacità iniziale potrebbe non essere sufficiente

Soluzione:

- Viene allocato in memoria un nuovo vettore, il contenuto del vecchio è copiato nel nuovo, il nuovo valore viene scritto, il vecchio valore viene deallocato dalla memoria

Vettori dinamici

Domanda

Qual è l'approccio migliore?

Approccio 1

Se il vecchio vettore ha dimensione n, allocate il nuovo vettore di dimensione dn. Per esempio, $d = 2$.

Approccio 2

Se il vecchio vettore ha dimensione n, allocate un nuovo vettore di dimensione $n + d$, dove d è una costante. Per esempio, $d = 16$.

Analisi ammortizzata - raddoppiamento

Costo effettivo di un'operazione append():

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

Assunzioni:

- Capacità iniziale: 1
- Costo di scrittura: $\Theta(1)$

n	cost
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

Costo effettivo di n operazioni append():

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

Costo ammortizzato di una singola append():

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

Analisi ammortizzata - incremento

Costo effettivo di un'operazione append():

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

Assunzioni

- Incremento: d
- Dimensione attuale: d
- Costo di scrittura: $\Theta(1)$

Esempio

- $d = 4$

n	cost
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

Costo effettivo di n operazioni append():

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

Costo ammortizzato di una singola append():

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

Reality check

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

Python – List

Operatore		Caso pessimo	Caso pessimo ammortizzato
`L.copy()`	Copy	$O(n)$	$O(n)$
`L.append(x)`	Append	$O(n)$	$O(1)$
`L.insert(i,x)`	Insert	$O(n)$	$O(n)$
`L.remove(x)`	Remove	$O(n)$	$O(n)$
`L[i]`	Index	$O(1)$	$O(1)$
`for x in L`	Iterator	$O(n)$	$O(n)$
`L[i:i+k]`	Slicing	$O(k)$	$O(k)$
`L.extend(s)`	Extend	$O(n+k)$	$O(n)$
`x in L`	Contains	$O(n)$	$O(n)$
`min(L)`, `max(L)`	Min, Max	$O(n)$	$O(n)$
`len(L)`	Get length	$O(1)$	$O(1)$

Funzioni/tabelle hash in Python

Python `sets` e `dict`

- Sono implementati tramite tabelle hash
- Gli insiemi sono forme degenerate di dizionari, dove non ci sono valori, solo chiavi

Strutture dati senza ordine

- L'ordine fra le chiavi non è preservato da una funzione hash; questo è il motivo per cui si può ottenere un ordinamento diverso quando si stampa o si iterano

Python – `set`

Operazione		Caso medio	Caso pessimo
`set()`	Constructor	$O(1)$	$O(1)$
`x in S`	Contains	$O(1)$	$O(n)$
`S.add(x)`	Insert	$O(1)$	$O(n)$
`S.remove(x)`	Remove	$O(1)$	$O(n)$
`S|T`	Union	$O(n+m)$	$O(n \cdot m)$
`S&T`	Intersection	$O(\min(n,m))$	$O(n \cdot m)$
`S-T`	Difference	$O(n)$	$O(n \cdot m)$
`for x in S`	Iterator	$O(n)$	$O(n)$
`len(S)`	Get length	$O(1)$	$O(1)$
`min(S)`, `max(S)`	Min, Max	$O(n)$	$O(n)$

$n = \texttt{len(S)}, m = \texttt{len(T)}$

`https://docs.python.org/2/library/stdtypes.html#set`

Python – `dict`

Operazione		Caso medio	Caso pessimo
`x in D`	Contains	$O(1)$	$O(n)$
`D[] =`	Insert	$O(1)$	$O(n)$
`= D[]`	Lookup	$O(1)$	$O(n)$
`del D[]`	Remove	$O(1)$	$O(n)$
`for x in S`	Iterator	$O(n)$	$O(n)$
`len(S)`	Get length	$O(1)$	$O(1)$

$$n = \texttt{len(S)}, m = \texttt{len(T)}$$

Implementare funzioni hash [1]

Regola: Se due oggetti sono uguali, i lor hash devono essere uguali

- Se implementate `__eq__()`, dovete implementare anche `__hash__()`

Regola: Se due oggetti hanno lo stesso valore hash, allora probabilmente sono uguali

- Dovreste evitare di restituire valori che generano collisioni nella vostra funzione hash

Regola: Affinchè un oggetto sia hashable, deve essere immutabile

- Il valore hash di un oggetto non deve cambiare nel tempo

[1] `http://www.asmeurer.com/blog/posts/what-happens-when-you-mess-with-hashing-in-python/`

Esempio

```
class Point:

  def __init__(self, x, y):
    self.x = x
    self.y = y

  def __eq__(self, other):
    return self.x == other.x and self.y == other.y

  def __hash__(self):
    return hash( (self.x,self.y) )
```

Table of contents

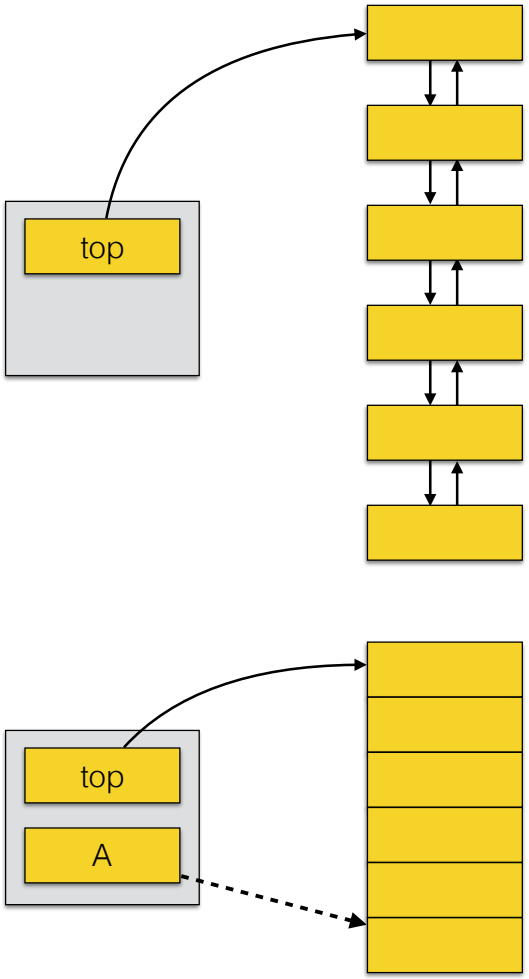
Esempio Stack

Stack Operazione	Stack Contents	Return Value
`s.isEmpty()`	`[]`	`True`
`s.push(4)`	`[4]`	
`s.push('dog')`	`[4,'dog']`	
`s.peek()`	`[4,'dog']`	`'dog'`
`s.push(True)`	`[4,'dog',True]`	
`s.size()`	`[4,'dog',True]`	`3`
`s.isEmpty()`	`[4,'dog',True]`	`False`
`s.push(8.4)`	`[4,'dog',True,8.4]`	
`s.pop()`	`[4,'dog',True]`	`8.4`
`s.pop()`	`[4,'dog']`	`True`
`s.size()`	`[4,'dog']`	`2`

Stack

Possibili usi

- In linguaggi come Python:
  - Compilatore: Per bilanciare parentesi
  - Nell'interprete: Un nuovo record di attivazione viene creato ad ogni chiamata di funzione
- Nell'analisi dei grafi:
  - Per eseguire visite dell'intero grafo



Possibili implementazioni

- Attraverso liste bidirezionali
  - riferimento all'elemento top
- Attraverso vettori (statici o no)
  - Basso overhead

Stack in Python

```
class Stack:
    def __init__(self):
        self.items = []

    def size(self):
        return len(self.items)

    def isEmpty(self):
        return len(self.items) == 0

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[-1]

    def push(self, item):
        self.items.append(item)
```

Esempio di applicazioni: parentesi bilanciate

- Controllate se le seguenti stringhe di parentesi sono bilanciate
  - `{ { ( [ ] [ ] ) } ( ) }`
  - `[ [ { { ( ( ) ) } } ] ]`
  - `[ ] [ ] [ ] ( ) { }`
  - `( [ ) ]`
  - `( ( ( ) ] ) )`
  - `[ { ( ) ]`

Esempio di applicazioni: parentesi bilanciate

```
def parChecker(parString):
  s = Stack()
  index = 0
  for symbol in parString:
    if symbol in "([{":
      s.push(symbol)
    else:
      if s.isEmpty():
        return False
      else:
        top = s.pop()
        if not matches(top,symbol):
            return False
  return s.isEmpty()
```

Esempio di applicazioni: parentesi bilanciate

```
def matches(openpar,closepar):
  opens = "([{"
  closers = ")]}"
  return opens.index(openpar) == closers.index(closepar)


print(parChecker('{{([][])}()}'))
print(parChecker('[{()]'))
```

Queue in Python (errato)

```
class Queue:
  def __init__(self):
    self.items = []

  def isEmpty(self):
    return self.items == []

  def enqueue(self, item):
    self.items.insert(0,item)

  def dequeue(self):
    return self.items.pop()

  def size(self):
    return len(self.items)
```

Python data structure - Dequeue

```
>>> from collections import deque
>>> Q = deque(["Eric", "John", "Michael"])
>>> Q.append("Terry")          # Terry arrives
>>> Q.append("Graham")         # Graham arrives
>>> Q.popleft()          # The first to arrive now leaves
'Eric'
>>> Q.popleft()          # The second to arrive now leaves
'John'
>>> Q                  # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

Table of contents

Reverse string

```
def reverse(s):
  n = len(s)-1
  res = ""
  while n >= 0:
    res = res + s[n]
    n -= 1
  return res
```

Reverse string

```
def reverse(s):
  n = len(s)-1
  res = ""
  while n >= 0:
    res = res + s[n]
    n -= 1
  return res
```

Complessità: $\Theta(n^2)$

- n somme di stringhe
- Ogni somma copia la stringa precedente in una nuova stringa

Reverse string

```
def reverse(s):
  res = []
  for c in s:
    res.insert(0, c)
  return "".join(res)
```

Reverse string

```
def reverse(s):
  res = []
  for c in s:
    res.insert(0, c)
  return "".join(res)
```

Complessità: $\Theta(n^2)$

- n inserimenti di stringhe
- Ogni inserimento muove tutti i caratteri di una posizione in avanti nella lista

Reverse string

```
def reverse(s):
  n = len(s)-1
  res = []
  while n >= 0:
    res.append(s[n])
    n -= 1
  return "".join(res)
```

Reverse string

```
def reverse(s):
  n = len(s)-1
  res = []
  while n >= 0:
    res.append(s[n])
    n -= 1
  return "".join(res)
```

Complessità: $\Theta(n)$

- n operazioni append
- Each append has an amortized cost of $O(1)$

Reverse string

```
def reverse(s):
  n = len(s)-1
  res = []
  while n >= 0:
    res.append(s[n])
    n -= 1
  return "".join(res)
```

Complessità: $\Theta(n)$

- n operazioni append
- Each append has an amortized cost of $O(1)$

Soluzione pythonica

```
def reverse(s):
  return s[::-1]
```

Rimozione dei duplicati

```
def deduplicate(L):
  res=[]
  for item in L:
    if item not in res:
      res.append(item)
  return res
```

Rimozione dei duplicati

```
def deduplicate(L):
  res=[]
  for item in L:
    if item not in res:
      res.append(item)
  return res
```

Complessità: $\Theta(n^2)$

- n operazioni append
- n controlli se un elemento è già presente
- Ogni controllo ha costo lineare $O(n)$

Rimozioni dei duplicati

```
def deduplicate(L):
  res=[]
  present=set()
  for item in L:
    if item not in present:
      res.append(item)
      present.add(item)
  return res
```

Rimozioni dei duplicati

```
def deduplicate(L):
  res=[]
  present=set()
  for item in L:
    if item not in present:
      res.append(item)
      present.add(item)
  return res
```

Complessità: $\Theta(n)$

- n operazioni append
- n controlli se un elemento è già presente
- Ogni controllo ha costo lineare $O(1)$

Rimozioni dei duplicati

```
def deduplicate(L):
  res=[]
  present=set()
  for item in L:
    if item not in present:
      res.append(item)
      present.add(item)
  return res
```

Complessità: $\Theta(n)$

- n operazioni append
- n controlli se un elemento è già presente
- Ogni controllo ha costo lineare $O(1)$

Un'altra possibilità – distrugge l'ordine originale

```
def deduplicate(L):
  return list(set(L))
```

Exercise

Queues and Priority Queues are data structures which are known to most computer scientists. The Italian Queue, however, is not so well known, though it occurs often in everyday life. At lunch time the queue in front of the cafeteria is an italian queue, for example.

In an italian queue each element belongs to a group. If an element enters the queue, it first searches the queue from head to tail to check if some of its group members (elements of the same group) are already in the queue.

- If yes, it enters the queue right behind them.
- If not, it enters the queue at the tail and becomes the new last element (bad luck).

Dequeuing is done like in normal queues: elements are processed from head to tail in the order they appear in the italian queue.