Pile e Code

Algoritmi 1 – Lezione 11

A. Monti
Sapienza Università di Roma

Corso: Algoritmi 1

Indice

1 Pile e Code

Introduzione alle Strutture Dati: Pile e Code

Le **pile** e le **code** sono due delle strutture dati fondamentali in informatica, utilizzate per gestire collezioni di oggetti in ordine.

Pile:

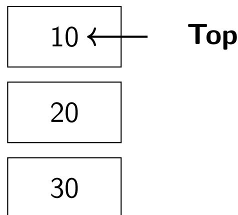
- La pila segue la strategia *Last In, First Out* (**LIFO**), ovvero l'ultimo elemento inserito è il primo ad essere estratto.
- Operazioni principali: **push** (inserimento) e **pop** (estrazione).

Code:

- La coda segue la strategia *First In, First Out* (**FIFO**), ovvero il primo elemento inserito è il primo ad essere estratto.
- Operazioni principali: **enqueue** (inserimento) e **dequeue** (estrazione).

Queste strutture sono essenziali in una vasta gamma di applicazioni, come la gestione di processi ricorsivi o la navigazione di alberi.

Esempio: Pila (LIFO)



La pila cresce verso l'alto: l'ultimo elemento inserito è il primo a uscire (Last In, First Out).

Implementazione di una Pila con un Array Dinamico in Python

In Python, per implementare una pila possiamo usare un **array dinamico** (ovvero una lista):

Programma per l'inserimento (push):

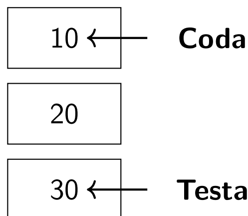
```
def push(pila, elemento):
    # Aggiunge l'elemento alla fine della lista
    pila.append(elemento)
```

Programma per l'estrazione (pop):

```
def pop(pila):
    if len(pila) == 0:
        # La pila è vuota
        return None
    # Estrae l'ultimo elemento
    return pila.pop()
```

Le operazioni di inserimento (`push`) e di estrazione (`pop`) sono molto efficienti richiedendo tempo $O(1)$.

Esempio: Coda (FIFO)



Gli elementi entrano dall'alto ed escono dal basso: (First In, First Out).

- **Inserimento (`enqueue`) in coda**: aggiunge un elemento alla fine dell'array.
- **Estrazione (`dequeue`) dalla testa**: rimuove l'elemento all'inizio dell'array.

Implementazione di una Coda con un Array Dinamico in Python

In una **coda** implementata con un array, possiamo inserire nuovi elementi alla fine (in coda) e rimuovere elementi dalla testa:

Programma per l'inserimento (enqueue):

```
def enqueue(coda, elemento):
    # Aggiunge l'elemento alla fine dell'array
    coda.append(elemento)
```

Programma per l'estrazione (dequeue):

```
def dequeue(coda):
    if len(coda) == 0:
        # La coda è vuota
        return None
    # Rimuove il primo elemento (testa della coda)
    return coda.pop(0)
```

- L'operazione di **inserimento** richiede tempo $O(1)$ poiché aggiungiamo l'elemento alla fine dell'array.
- L'operazione di **estrazione** richiede tempo $O(n)$, dove n è il numero degli elementi nella coda, poiché la rimozione del primo elemento comporta lo spostamento di tutti gli altri di una posizione verso sinistra.

La Coda con Array Dinamico e Cancellazione Logica

Implementiamo una **coda** utilizzando un **array dinamico** in Python, dove teniamo traccia della **testa** tramite un indice che si aggiorna ad ogni operazione di cancellazione.

Funzione di inserimento (enqueue):

```
def enqueue(coda, elemento):
    # Aggiunge l'elemento alla fine dell'array
    coda.append(elemento)
```

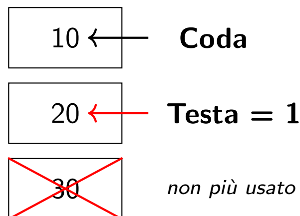
Funzione di cancellazione logica (dequeue):

```
def dequeue(coda, testa):
    if testa >= len(coda):
        # La coda è vuota
        return None, testa
   # Otteniamo il valore da "rimuovere"
    valore = coda[testa]
    testa += 1  # Aggiorniamo l'indice della testa
    # Restituiamo l'elemento rimosso e il nuovo indice della testa
    return valore, testa
```

- L'operazione di **inserimento** richiede tempo $O(1)$, poiché aggiungiamo l'elemento alla fine dell'array.
- L'operazione di **cancellazione logica** richiede tempo $O(1)$, poiché si limita ad aggiornare l'indice della testa senza spostare gli altri elementi.

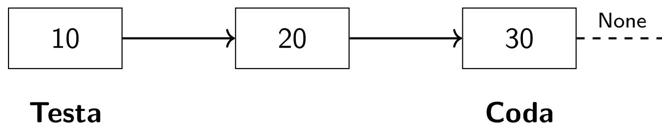
Conclusione: Con la cancellazione logica, la gestione temporale della coda diventa molto più efficiente, tuttavia c'è un problema di spazio: la cancellazione logica non libera effettivamente la memoria degli elementi "cancellati".

Cancellazione Logica nella Coda



L'indice 'testa' avanza, ma gli elementi precedenti restano in memoria.

Lista a Puntatori per una Coda



Ogni nodo punta al successivo. Il primo nodo è la testa della coda, l'ultimo è la coda (il prossimo è ‘None‘).

La Coda con Lista a Puntatori

Quando implementiamo una **coda** tramite una lista a puntatori, possiamo inserire e cancellare gli elementi in modo dinamico, utilizzando solo due puntatori: uno alla **testa** e uno alla **coda** della lista.

```
def enqueue(testa, coda, x):
    q = Nodo(x)  # Creiamo un nuovo nodo con valore x
    if testa == None:
        # Se la coda era  vuota, ora la testa e la coda sono lo stesso nodo
        return q, q
    # Colleghiamo l'elemento alla coda
    coda.next = q
    # La testa rimane invariata, aggiorniamo la coda
    return testa, q

def dequeue(testa, coda):
    if testa == None:
        # La coda è vuota
        return None, None, None
    if testa == coda:
        # se la coda ha un solo elemento lo restituiamo e testa=coda=None
        return testa.key, None, None
    # Rimuoviamo l'elemento dalla testa e aggiorniamo la testa
    return testa.key, testa.next, coda
```

- L'**inserimento** richiede tempo $O(1)$, poiché inseriamo l'elemento alla fine della lista.
- La **cancellazione** richiede tempo $O(1)$, poiché rimuoviamo un elemento dalla testa della lista.

Conclusioni: La gestione della memoria è dinamica, quindi non c'è un utilizzo fisso di spazio e la dimensione della coda coincide con il numero di elementi in essa contenuti.

La Coda con deque in Python

La struttura *deque* (Double Ended Queue) di Python consente di effettuare operazioni di inserimento e cancellazione in tempo $O(1)$ sia alla **testa** che alla **coda**.
In particolare, la funzione `popleft()` permette di rimuovere un elemento dalla testa della coda con un costo di $O(1)$, senza dover spostare gli altri elementi, cosa che invece accadrebbe con una lista normale.

Funzioni principali:

- `append(x)`: Inserisce l'elemento x in coda ($O(1)$).
- `popleft()`: Estrae l'elemento dalla testa della coda ($O(1)$).

Esempio di utilizzo di `deque` per implementare una coda:

```
from collections import deque

# Creazione di una coda vuota
coda = deque()

# Inserimento in coda (O(1))
coda.append(x)  # Inserisce x in coda

# Cancellazione dalla testa (O(1))
x = coda.popleft()  # Rimuove e restituisce l'elemento in testa alla coda
```

Conclusione: La struttura deque è ideale per implementare una coda dinamica con operazioni di inserimento e cancellazione efficienti alle estremità.