Pile e code

Corso di **Algoritmi e strutture dati**
Corso di Laurea in **Informatica**
Docenti: Ugo de'Liguoro, András Horváth

Indice

1. Tipo di dato astratto (abstract data type, ADT)
2. Pile (stack)
3. Code

Sommario

Obiettivo:

- introdurre il concetto del **tipo di dato astratto (abtrasct data type, ADT)**
- specificare **pile e code come ADT** e definire **due diverse implementazioni**

1. Tipi di dati

- i linguaggi di programmazione tipati forniscono **tipi predefiniti**
- ogni tipo di dato è associato con un **insieme di valori** e **operatori**:
  - unsigned int: $0, 1, 2, \ldots$ con operazioni $+, -, \ldots$
  - boolean: T, F con operazioni $\neg, \wedge, \ldots$
- ogni operatore funzione secondo certe **regole**
- quando usiamo i tipi forniti dal linguaggio non ci chiediamo come vengono effettuate le operazioni
- si possono introdurre nuovi tipi di dati e implementare operazione per i nuovi tipi

1. Tipo di dato astratto (abstract data type, ADT)

- un tipo di dato è **astratto** se è descritto prescindendo dalla sua concreta implementazione
- tale descrizione riguarda
  - **la collezione di dati**: a partire da quali tipi di dati si costruisce una struttura del nuovo tipo (ma non come la si costruisce!)
  - **le operazioni**: che cosa devono fare le operazioni definite sul nuovo tipo (ma non come lo devono fare!)
  - **complessità**: eventualmente dei vincoli di complessità su tali operazioni
- la descrizione delle operazioni con le pre- e postcondizioni è una sorta di **assiomatizzazione** del tipo

1. Tipo di dato astratto (abstract data type, ADT)

- un'**implementazione** concreta di un ADT è
  - una **struttura dati** con cui memorizzare la collezione di dati
  - ed una collezione di **procedure** con cui realizzare le operazioni
- la relazione fra tipo astratto e struttura concreta è analoga a quella fra problema algoritmico e algoritmo.

2. Pile (stack)

In una pila i dati vengono estratti in ordine inverso rispetto a quello in cui sono stati inseriti.

Terminologia:

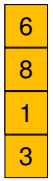
- **push**: inserire un elemento nella pila
- **pop**: estrarre un elemento dalla pila
- **top**: restituisce l'elemento in cima

2. Pile (stack)

Operazioni:

- push(3)
- push(5)
- push(9)
- pop()
- pop()
- push(1)
- push(8)
- push(2)
- pop()
- push(6)

Pila finale:



2. Pila come ADT

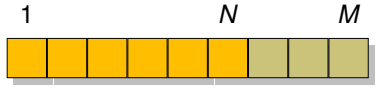
- **collezione dati**: elementi di qualunque tipo T di dati
- **operazioni**:
  - *void* PUSH(*Stack* S, T t)
  - T POP(*Stack* S)
  - T TOP(*Stack* S)
  - *bool* EMPTY(*Stack* S)
  - *int* SIZE(*Stack* S)

2. Pila come ADT

- **assiomi**:
  - SIZE(S), EMPTY(S) e PUSH(S, t) sono sempre definiti
  - POP(S) e TOP(S) sono definiti se e solo se EMPTY(S) restituisce falso
  - EMPTY(S), SIZE(S) e TOP(S) non modificano la pila S
  - EMPTY(S) restituisce vero se e solo se SIZE(S) restituisce 0
  - la sequenza PUSH(S, t);POP(S) restituisce t e non modifica la pila S
  - la sequenza PUSH(S, t);TOP(S) restituisce t
  - PUSH(S, t) incrementa SIZE(S) di 1
  - POP(S) decrementa SIZE(S) di 1

2. Implementazione concreta con array

- usiamo un array statico di M celle per definire un'implementazione concreta del ADT pila



- grazie al meccanismo LIFO (Last In First Out) conviene fare così:
  - gli elementi presenti nella pila occupano sempre le prime posizioni dell'array
  - quando ci sono N elementi, il prossimo elemento da estrarre è nella posizione N
- la scelta della struttura dati concreta "aggiunge un assioma":
  - PUSH(S, t) è definito se solo se SIZE(S)$< M$

2. Implementazione concreta con array

PUSH(S, t)
  if $S.N \neq S.M$ **then**
    $S.N \leftarrow S.N + 1$
    $S[N] \leftarrow t$
  else
    error*overflow*

SIZE(S)
  return $S.N$

EMPTY(S)
  if $S.N == 0$ **then**
    return *true*
  return *false*

TOP(S)
  if $S.N == 0$ **then**
    error*underflow*
  else
    return $S[S.N]$

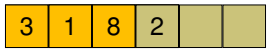
POP(S)
  if $S.N == 0$ **then**
    error*underflow*
  else
    $S.N \leftarrow S.N - 1$
    return $S[S.N + 1]$

2. Implementazione concreta con array

Operazioni:

- push(3), push(5), push(9), pop(), pop(), push(1), push(8), push(2), pop()

Pila finale:



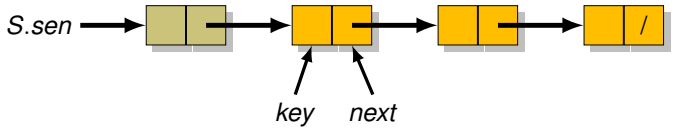
2. Implementazione concreta con array

- l'array darebbe la possibilità di inserimenti e estrazioni in una posizione qualsiasi
- ma se il programmatore ha deciso di utilizzare l'ADT pila allora può interagire con la pila solo seconda la specifica del ADT
- l'implementazione concreta e la struttura dati
  - sono **nascosti dietro una interfaccia**
  - e **possono essere modificate** senza fare modifiche a programmi che usano l'ADT

2. Implementazione concreta con lista

Utilizziamo una lista per realizzare la pila:

- quale tipo di lista conviene utilizzare: doppiamente concatenate, circolari?
- conviene una lista semplice ma con sentinella per non dover fare controlli



- conviene tener conto del numero di elementi che sarà denotato con $S.N$

2. Implementazione concreta con lista

PUSH(S, t)
  $S.N \leftarrow S.N + 1$
  $t.next \leftarrow S.sen.next$
  $S.sen.next \leftarrow t$

SIZE(S)
  return $S.N$

EMPTY(S)
  if $S.N == 0$ **then**
    return *true*
  return *false*

TOP(S)
  if $S.N == 0$ **then**
    error*underflow*
  else
    return $S.sen.next$

POP(S)
  if $S.N == 0$ **then**
    error*underflow*
  else
    $S.N \leftarrow S.N - 1$
    $t \leftarrow S.sen.next$
    $S.sen.next \leftarrow S.sen.next.next$
    return t

2. Confronto delle due implementazioni

- complessità temporale delle operazioni?
  sono tutte $O(1)$
- complessità spaziale delle strutture?
  con l'array $O(M)$ (proporzionale al numero massimo di elementi), con le liste $O(N)$ (ma c'è l'overhead dovuto ai puntatori)
- con l'array bisogna stabilire a priori il numero massimo di elementi, con le liste no

2. Utilizzo della struttura dati pila

- chiamate ricorsive di funzioni
- visita in profondità di grafi
- valutazione di un'espressione in notazione postfissa

3. Code (queue)

In una coda i dati vengono estratti nell'ordine in cui sono stati inseriti.

Terminologia:

- **enqueue**: inserire un elemento nella coda
- **dequeue**: estrarre un elemento dalla coda
- **front**: restituisce il primo elemento nella coda

3. Code (queue)

Operazioni:

- queue(3), queue(5), queue(9), dequeue(), dequeue(), queue(1), queue(8), queue(2), dequeue()

Coda finale:

3. Coda come ADT

- **collezione dati**: elementi di qualunque tipo T di dati
- **operazioni**:
  - *void* ENQUEUE(*Queue* Q, T t)
  - T DEQUEUE(*Queue* Q)
  - T FRONT(*Queue* Q)
  - *bool* EMPTY(*Queue* Q)
  - *int* SIZE(*Queue* Q)

3. Coda come ADT

- **assiomi**:
  - SIZE(Q), EMPTY(Q) e ENQUEUE(Q, t) sono sempre definiti
  - DEQUEUE(Q) e FRONT(Q) sono definiti se e solo se EMPTY(Q) restituisce falso
  - EMPTY(Q), SIZE(Q) e FRONT(Q) non modificano la coda Q
  - EMPTY(Q) restituisce vero se e solo se SIZE(Q) restituisce 0
  - se SIZE(Q)$= N$ e viene effettuata ENQUEUE(Q, t), allora dopo N esecuzione di DEQUEUE(Q) abbiamo FRONT(Q)$= t$
  - se FRONT(Q)$= t$ allora DEQUEUE(Q) estrae t dalla coda
  - ENQUEUE(Q, t) incrementa SIZE(Q) di 1
  - DEQUEUE(Q) decrementa SIZE(Q) di 1

3. Implementazione concreta con array

Usiamo un array statico di M celle per definire un'implementazione:

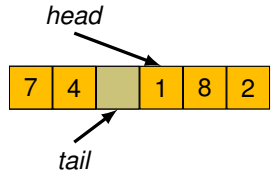
- anche in questo caso conviene tenere gli elementi nelle prime posizioni dell'array?
  no! perché?
  - se l'elemento da estrarre è nella prima posizione allora DEQUEUE(Q) richiede spostare gli elementi rimanenti
  - se l'elemento da estrarre è nell'ultima posizione allora ENQUEUE(Q) richiede spostare gli elementi presenti
  - sarebbero operazioni da $O(N)$

3. Implementazione concreta con array

Useremo l'array in maniera **"circolare"** tenendo conto di dove si trova l'inizio (head) e la fine (tail) della coda.

Operazioni: queue(3), queue(5), queue(9), dequeue(), dequeue(), queue(1), queue(8), queue(2), dequeue(), queue(7), queue(4).

Coda finale:



3. Implementazione concreta con array

- $Q.head$ indica la posizione da dove estrarre l'elemento successivo
- $Q.tail$ indica la posizione dove inserire l'elemento successivo
- come si controlla se la coda sia vuota?

$$Q.head == Q.tail \iff \text{la coda è vuota}$$

- di conseguenza possiamo gestire $M - 1$ elementi al massimo con un array di M celle

3. Implementazione concreta con array

```
SIZE(Q)
  if Q.tail ≥ Q.head then
      return Q.tail − Q.head
  return Q.M − (Q.head − Q.tail)
EMPTY(Q)
  if Q.tail == Q.head then
      return true
  return false
NEXTCELL(Q, c)
  if c ≠ Q.M then
      return c + 1
  return 1
```


3. Implementazione concreta con array

```
ENQUEUE(Q, t)
  if SIZE(Q) ≠ Q.M − 1 then
      Q[Q.tail] ← t
      Q.tail ← NEXTCELL(Q,Q.tail)
  else
      erroroverflow
FRONT(Q)
  if SIZE(Q)== 0 then
      errorunderflow
  else
      return Q[Q.head]
```

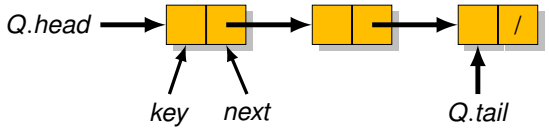

3. Implementazione concreta con array

```
DEQUEUE(Q)
  if SIZE(Q)== 0 then
      errorunderflow
  else
      t ← Q[Q.head]
      Q.head ← NEXTCELL(Q,Q.head)
      return t
```

3. Implementazione concreta con lista

Utilizziamo una lista per realizzare la coda:

- quale tipo di lista conviene utilizzare: doppiamente concatenate, circolari?
- inserimenti vengono fatti in testa, estrazioni in coda
- usiamo una lista semplice ma aggiungiamo un puntatore all'ultimo elemento della coda



3. Implementazione concreta con lista

- *Q.head* indica l'elemento da estrarre
- *Q.tail* indica l'ultimo elemento inserito
- come si controlla se la coda sia vuota?

$Q.head == nil \iff$ la coda è vuota

- ma comunque teniamo conto del numero di elementi in $Q.N$

3. Implementazione concreta con lista

ENQUEUE(Q, t)
  if $Q.N == 0$ **then**
    $Q.head \leftarrow t$
    $Q.tail \leftarrow t$
  else
    $Q.tail.next \leftarrow t$
    $Q.tail \leftarrow t$
  $Q.N \leftarrow Q.N + 1$

SIZE(Q)
  return $Q.N$

EMPTY(Q)
  if $Q.N == 0$ **then**
    return *true*
  return *false*

FRONT(Q)
  if $Q.N == 0$ **then**
    error*underflow*
  else
    return $Q.head$

DEQUEUE(Q)
  if $Q.N == 0$ **then**
    error*underflow*
  else
    $t \leftarrow Q.head$
    $Q.head \leftarrow Q.head.next$
    $Q.N \leftarrow Q.N - 1$
    return t

(...sentinella aiuterebbe?)

3. Confronto delle due implementazioni

- complessità temporale delle operazioni?
  sono tutte $O(1)$
- complessità spaziale delle strutture?
  con l'array $O(M)$ (proporzionale al numero massimo di elementi), con le liste $O(N)$ (ma c'è l'overhead dovuto ai puntatori)
- con l'array bisogna stabilire a priori il numero massimo di elementi, con le liste no

3. Utilizzo della struttura dati coda

- buffer
- visita in ampiezza di grafi
- simulazione