Informatica per Statistica
Riassunto della lezione del 07/11/2012

Igor Melatti

Strutture dati e operazioni per pile e code

- Una *pila* (*stack*) è una *struttura dati* che rappresenta un insieme *dinamico*
  - ovvero, a questo insieme possono essere aggiunti o tolti elementi in qualsiasi momento
- La caratteristica della pila è che l'aggiunta e la rimozione di elementi avvengono secondo la politica *LIFO* (*last in first out*, l'ultimo ad essere stato aggiunto è il primo ad essere rimosso)
  - per questo si chiama pila: può essere immaginata come una sequenza di oggetti impilati uno sull'altro
  - per aggiungere un elemento, lo si aggiunge in cima
  - per rimouvere un elemento, lo si rimuove dalla cima
  - dato che nella cima della pila c'è l'elemento inserito più di recente, una rimozione rimuove l'ultimo elemento arrivato (da qui *LIFO*)
  - inoltre, il primo arrivato, che si trova in fondo alla pila, sarà l'ultimo ad essere preso (infatti si può dire anche *FILO*, *first in last out*)
- Assumendo di essere a conoscenza del numero massimo di elementi da memorizzare, si può realizzare una coda con un array S (da Stack...)
  - negli esempi che seguono, assumeremo che gli elementi da memorizzare siano interi
- Tuttavia, il solo array non basta: occorre un indice che dica qual è l'elemento attualmente in cima
  - infatti, un "normale" array S ha sempre come elementi validi $S[1] \dots S[\text{length}[S]]$
  - quindi sarebbe impossibile dire in quale delle posizioni di S si trovi l'elemento che è attualmente sulla testa dello stack (così da sapere dove effettuare una rimozione o un'aggiunta)

- Allora, oltre ad S c'è anche top[S] ("cima")
  - contiene l'indice di S in cui è memorizzato l'ultimo elemento inserito
  - per esempio, se in S sono stati inseriti (in quest'ordine) 3, 6, 9, 2, e length[S] = 6, allora $S = \langle 3, 6, 9, 2, x, y \rangle$, dove x e y potrebbero essere qualsiasi
  - se ora si volesse effettuare una rimozione, senza top[S] non si saprebbe quale elemento prendere ($S[1]$? $S[6]$? $S[3]$?)
  - invece, c'è anche top[S] = 4, quindi se la prossima operazione è una rimozione allora $S[4]$ è l'elemento da rimuovere, mentre se è un'ulteriore aggiunta allora $S[4+1]$ è la posizione dove inserire
- Nelle pile, l'operazione di aggiunta viene chiamata *Push* ("premi", sottinteso: sulla cima della pila), mentre l'operazione di rimozione viene chiamata *Pop* ("togli")
- In Figura 1 vengono riportati gli pseudocodici per le operazioni sulla pila

```
1  InitStack(S)
2    top[S] ← 0
3
4  Push(S, x)
5    top[S] ← top[S] + 1
6    S[top[S]] ← x
7
8  Pop(S)
9    top[S] ← top[S] - 1
10   return S[top[S] + 1]
11
12 StackEmpty(S)
13   return top[S] = 0
14
15 StackFull(S)
16   return top[S] = length[S]
```

Figure 1: Operazioni su una pila

  - *InitStack* è la prima operazione da chiamare quando si vuole cominciare ad usare lo stack
  - *StackEmpty* ritorna vero se e solo se lo stack è vuoto, ovvero se non contiene alcun elemento
  - *StackFull* ritorna vero se e solo se lo stack è pieno, ovvero se contiene già length[S] elementi
  - è sbagliato sia effettuare una push su uno stack pieno (*overflow*), che effettuare una pop su uno stack vuoto (*underflow*)

- la Figura 2 mostra come *Push* e *Pop* possano tenere conto di questi casi

```
1  Push(S, x)
2    if StackFull(S)
3      return OVERFLOW_ERROR
4    top[S] ← top[S] + 1
5    S[top[S]] ← x
6
7  Pop(S)
8    if StackEmpty(S)
9      return UNDERFLOW_ERROR
10   top[S] ← top[S] - 1
11   return S[top[S] + 1]
```

Figure 2: *Push* e *Pop* con condizioni di errore

- Tutte le operazioni di Figura 1 (e Figura 2) eseguono un numero costante di operazioni
  - non c'è nessun ciclo che scorra tutto l'array S
  - quindi, se n =length[S], tutte queste operazioni sono *indipendenti* da n
- Quindi, la complessità delle operazioni sulla pila è $O(1)$
- Una *coda* (*queue*) è anch'essa una struttura dati che rappresenta un insieme dinamico
- La caratteristica della coda è che l'aggiunta e la rimozione di elementi avvengono secondo la politica *FIFO* (*first in first out*, il primo ad essere stato aggiunto è il primo ad essere rimosso)
  - per questo si chiama coda: può essere immaginata come una sequenza di oggetti infilati l'uno dopo l'altro
  - per aggiungere un elemento, lo si aggiunge in fondo alla fila
  - per rimouvere un elemento, lo si rimuove dall'inizio della fila
  - dato che all'inizio della fila c'è l'elemento inserito meno di recente (ovvero il primo arrivato a suo tempo), una rimozione rimuove il primo elemento arrivato (da qui *FIFO*)
  - inoltre, l'ultimo arrivato, che si trova in fondo alla fila, sarà l'ultimo ad essere preso (infatti si può dire anche *LILO*, *last in last out*)
- Assumendo di essere a conoscenza del numero massimo di elementi da memorizzare, si può realizzare una coda con un array Q (da queue...)

- Tuttavia, il solo array non basta: occorre un indice che dica qual è l'elemento attualmente in fondo alla fila, e un altro che dica qual è l'elemento attualmente all'inizio della fila
  - infatti, un "normale" array Q ha sempre come elementi validi $Q[1] \dots Q[\text{length}[Q]]$
  - quindi sarebbe impossibile dire in quale delle posizioni di Q si trovi l'elemento che è attualmente all'inizio della coda (così da sapere dove effettuare una rimozione), oppure in quale delle posizioni di Q si trovi l'elemento che è attualmente in fondo alla coda (così da sapere dove effettuare un'ulteriore aggiunta)
- Allora, oltre ad Q c'è anche head$[Q]$ ("testa") e tail$[Q]$ ("coda")
  - head$[Q]$ contiene l'indice di Q in cui è memorizzato il primo elemento inserito
  - tail$[Q]$ contiene l'indice di Q in cui occorre inserire il prossimo elemento
  - per esempio, se in Q sono stati inseriti (in quest'ordine) 3, 6, 9, 2, e length$[Q] = 6$, allora $Q = \langle 3, 6, 9, 2, x, y \rangle$, dove x e y potrebbero essere qualsiasi
  - se ora si volesse effettuare una rimozione, senza head$[Q]$ non si saprebbe quale elemento prendere ($Q[1]$? $Q[6]$? $Q[3]$?)
  - invece, c'è anche head$[Q] = 1$, quindi se la prossima operazione è una rimozione allora $Q[1]$ è l'elemento da rimuovere
  - inoltre c'è anche tail$[Q] = 5$, quindi se la prossima operazione è un'ulteriore aggiunta allora $Q[5]$ è la posizione dove inserire
- Nelle code, l'operazione di aggiunta viene chiamata *Enqueue* ("metti in coda"), mentre l'operazione di rimozione viene chiamata *Dequeue* ("togli dalla coda")
- Differentemente dalla pila, nella coda l'array Q viene usato in modo *circolare*
  - l'idea è che tail$[Q]$ viene incrementato ogni volta che c'è da fare una *Enqueue*
  - ma così facendo, si potrebbero fare al massimo $n =$length$[Q]$ operazioni di *Enqueue*
  - quando invece, la presenza di eventuali *Dequeue* libererebbero dello spazio in Q per fare ulteriori *Enqueue*
  - questo spazio si libera all'inizio di Q
  - quindi, per trarre vantaggio da ciò, occorre che tail$[Q]$ possa "sfondare" oltre length$[Q]$ per tornare all'indice 1 (nel caso ovviamente che qualche *Dequeue* abbia liberato dei posti)

  - lo stesso farà ovviamente head[Q]
  - questo ha anche l'effetto di dover "sprecare" una posizione all'interno di Q
  - ovvero, in Q si potranno memorizzare non length[Q] elementi, ma solo length[Q] - 1
  - infatti, nel caso della pila, dato che top[S] può essere sia incrementato che decrementato, per sapere se la pila è vuota basta vedere se è 0, e per vedere se la pila è piena basta vedere se è andato oltre length[S]
  - nella coda no, perché è circolare, quindi a 0 non ci si va mai, e non è un errore superare length[Q]
  - per stabilire se la coda è piena o vuota occorre quindi confrontare tra loro head[Q] e tail[Q]
  - concettualmente, se sono uguali vuol dire che sono state fatte tanti inserimenti quante cancellazioni, quindi la coda è vuota
  - non c'è invece modo di dire quando la coda è piena, dato che qualsiasi posizione reciproca tra head[Q] e tail[Q] sarebbe legittima
    * potrebbe succedere che head[Q] sia più grande di tail[Q]: ad esempio, se si fanno length[Q] + 1 *Enqueue*, con in mezzo una *Dequeue*
    * potrebbe succedere che head[Q] sia più piccola di tail[Q]: ad esempio, dopo la prima *Enqueue*...
  - allora si dice che se head[Q] è uguale a tail[Q] + 1 la coda è piena: vuol dire che pur potendo inserire un elemento in tail[Q] (che attualmente è libero e disponibile), si rinuncia a farlo
  - quindi, pur avendo length[Q] posizioni, si memorizzano solo length[Q] - 1 elementi

- In Figura 3 vengono riportati gli pseudocodici per le operazioni sulla coda
  - *InitQueue* è la prima operazione da chiamare quando si vuole cominciare ad usare la coda
  - *QueueEmpty* ritorna vero se e solo se la coda è vuoto, ovvero se non contiene alcun elemento
  - *QueueFull* ritorna vero se e solo se la coda è pieno, ovvero se contiene già length[Q] elementi
    * scritta così non considera però tutti i casi: se head[Q] è uguale a length[Q], e tail[Q] è 1, la coda sarebbe piena, ma *QueueFull* ritorna falso
    * **esercizio**: correggere *QueueFull* in modo da considerare il caso di cui sopra
  - è sbagliato sia effettuare una Enqueue su una coda piena (*overflow*), che effettuare una Dequeue su una coda vuota (*underflow*)

```
1  InitQueue(Q)
2    head[Q] ← 1
3    tail[Q] ← 1
4
5  Enqueue(Q, x)
6    Q[tail[Q]] ← x
7    if tail[Q] = length[Q]
8    then tail[Q] ← 1
9    else tail[Q] ← tail[Q] + 1
10
11 Dequeue(Q)
12   x ← Q[head[Q]]
13   if head[Q] = length[Q]
14   then head[Q] ← 1
15   else head[Q] ← head[Q] + 1
16   return x
17
18 QueueEmpty(Q)
19   return head[Q] = tail[Q]
20
21 QueueFull(Q)
22   return head[Q] = tail[Q] + 1
```

Figure 3: Operazioni su una coda

- **esercizio**: correggere la *Enqueue* e la *Dequeue* di Figura 3 per tener conto di overflow ed underflow

- Tutte le operazioni di Figura 3 eseguono un numero costante di operazioni
  - non c'è nessun ciclo che scorra tutto l'array Q
  - quindi, se n =length[Q], tutte queste operazioni sono *indipendenti* da n

- Quindi, la complessità delle operazioni sulla coda è $O(1)$

Implementazione delle pile in C

- La Figura 4 mostra un modo di realizzare in C le pile

```
1  #define LENGTH_S 50
2
3  /* variabili globali */
4  int S[LENGTH_S];
5  int top;
6
7  void InitStack()
8  {
9    top = -1;
10 }
11
12 void Push(int x)
13 {
14   top++;
15   S[top] = x;
16 }
17
18 int Pop()
19 {
20   top--;
21   return S[top + 1];
22 }
```

Figure 4: Operazioni su una pila

  - realizzati con variabili globali: se si passassero `S` e `top` come parametri a `Push` e `Pop`, le modifiche a `top` non avrebbero alcun effetto
  - questo perché verrebbero eseguite sulla copia locale di `top`, e non sul `top` della funzione che chiama `Push` e `Pop`

- in realtà sarebbe possibile evitare di avere variabili globali (che su programmi grandi possono generare confusione), ma va al di là degli attuali scopi di questo corso
- dato che ovviamente l'array `S` ora va da 0 a `LENGTH_S - 1`, `top` va opportunamente scalato di 1 (vedere `InitStack`)
- **esercizio**: aggiungere le funzioni mancanti e il controllo di errore su `Push` e `Pop`
- **esercizio**: aggiungere una funzione che scriva su schermo tutti gli elementi validi di uno stack
- **esercizio**: realizzare una pila di `double` anziché di `int`
- **esercizio**: scrivere un programma che legge da tastiera dapprima un numero k (che dev'essere minore della lunghezza dell'array usato per la pila), poi legge k interi, e infine, sfruttando una pila, scrive i k interi nell'ordine inverso a quello di immissione
- **esercizio**: riscrivere le funzioni della pila in modo tale che lo stack `S` sia non una variabile globale, ma un parametro delle funzioni. Lasciare invece `top` come variabile globale. Provare un programma di esempio: funziona? E se anche `top` viene passata come parametro delle funzioni, il tutto funziona ancora?

- **Esercizio**: scrivere l'implementazione in C delle operazioni sulla coda