code generalizzate, ADT di prima categoria

Lezione 17 di Programmazione 2

Docente: Giuseppe Scollo

Università di Catania, sede di Comiso (RG)
Facoltà di Scienze Matematiche, Fisiche e Naturali
Corso di Studi in Informatica applicata, AA 2006-7

Indice

code generalizzate

dalle definizioni degli ADT pila e coda emerge che essi differiscono solo per la disciplina di accesso, in particolare per la **regola di rimozione** di elementi dalla sequenza

ciò conduce al concetto di **coda generalizzata**:

> sequenza finita, di lunghezza variabile, di dati omogenei, ad accesso sequenziale
>
> > la sequenzialità dell'accesso implica che l'inserimento di elementi avvenga sempre allo stesso estremo della sequenza: i vari tipi di coda generalizzata differiscono dunque solo per la regola di rimozione

coda casuale (ingl. random queue):

> coda generalizzata con scelta casuale equiprobabile dell'elemento da rimuovere

le operazioni delle code LIFO e FIFO sono eseguite in **tempo costante**, sia nelle implementazioni con array che in quelle con liste

> e nelle code casuali? ... ciò è possibile nell'implementazione **con array**, v. esercizio

altri tipi di code generalizzate: coda doppia, code con priorità, tabelle di simboli, ...

elementi duplicati, elementi indice

per un qualsiasi tipo di coda generalizzata è possibile distinguere diversi sottotipi in base alla gestione di **elementi duplicati**

> assumiamo che il tipo degli elementi sia dotato di una **chiave**, sì da considerare due elementi **duplicati se hanno la stessa chiave**

negli ADT visti sinora (code LIFO, FIFO, random), non era specificata alcuna gestione dei duplicati

> l'eventuale onere di prevenire la presenza di duplicati nella coda è lasciato al client dell'ADT

in molti casi è preferibile includere nella specifica dell'ADT una **disciplina di gestione dei duplicati:**

> ignora il nuovo
> dimentica il vecchio
>
> > inserendo al suo posto il nuovo
> > rimuovendolo e accodando il nuovo in ordine di arrivo

per implementare tali discipline occorre in ogni caso la funzionalità di **ricerca** di una chiave fra quelle degli elementi in coda, per il **riconoscimento dei duplicati**

se il tipo della chiave è `int`, una realizzazione rapida e semplice di questa funzionalità si ottiene con un **cast** degli elementi al tipo della chiave, purché il tipo degli elementi ne sia una derivazione

> struttura dati di supporto: un array binario, indicizzato dalle chiavi, dove `p[k] == 1` sse un elemento di chiave `k` è presente in coda

si realizza in tal caso una disciplina di gestione dei duplicati mediante **elementi indice**

pila senza duplicati con elementi indice

assumendo un cast di `Elem` a `int` che restituisca un intero < `maxN`, l'implementazione della pila con array può essere così modificata, per prevenire l'inserimento di duplicati secondo la disciplina "ignora il nuovo"

strutture dati private **ndstack-private.h:**

```
Elem *s; // puntatore alla struttura dati su cui si implementa la pila
bool *d; // puntatore all'array di controllo della presenza di duplicati
int N;   // numero di elementi nella pila + 1
         // (ordinale del prossimo elemento da aggiungere alla pila)
```

implementazione delle operazioni **ndstack.c++** :

```
#include "ndstack.h"
template <typename Elem>
Stack<Elem>::Stack(int maxN)
{ s = new Elem[maxN]; d = new bool[maxN]; N = 0;
  for (int i = 0; i < maxN; i++) d[i] = 0; }
template <typename Elem>
bool Stack<Elem>::empty() const { return N == 0; }
template <typename Elem>
void Stack<Elem>::push(Elem elem)
{ if (d[elem]) return; s[N++] = elem; d[elem] = 1; }
template <typename Elem>
Elem Stack<Elem>::pop() { d[s[--N]] = 0; return s[N]; }
```

ADT di prima categoria

come si è accennato a proposito della ridefinizione di operatori, quando si definisce un nuovo tipo di dato è conveniente poterlo trattare allo stesso modo dei tipi di dati predefiniti

gli ADT che godono di questa proprietà sono detti **ADT di prima categoria**

N.B. questa definizione è necessariamente vaga, perché non è facile determinare quali degli operatori sui tipi predefiniti siano appropriati a un ADT definito dall'utente

ad es., è appropriato ridefinire tutti gli operatori del tipo predefinito `float` a un ADT dei numeri complessi, ma non tutti quegli operatori hanno senso su qualsiasi ADT

seppur vaga, la definizione rende intuitivamente l'idea, lasciando al programmatore il compito di individuare gli operatori predefiniti dei quali è appropriata la ridefinizione per un nuovo ADT

se l'ADT implementato da una classe è accessibile solo per il tramite dell'interfaccia,
e **se non ha puntatori fra i suoi dati membro**, allora è un ADT di prima categoria

ciò perché ogni classe ha operatori di default per l'**assegnamento**, la **copia**, e il rilascio della memoria allocata all'istanza, cioè un **distruttore**: l'adeguatezza di tali operatori di default è però garantita solo se la classe non ha puntatori fra i suoi dati membro

se la classe che implementa un ADT ha puntatori fra i suoi dati membro, perché l'ADT sia di prima categoria occorre ridefinire opportunamente:

l'**operatore di assegnamento** e il **costruttore di copia**, per la completezza della copia
il **distruttore**, per la completezza del rilascio di memoria, cioè per evitare **memory leak**

interfaccia di coda FIFO di I categoria

l'interfaccia dell'ADT coda FIFO di I categoria è un'estensione di quella vista nella lezione precedente con costruttore di copia, operatore di assegnamento e distruttore

naturalmente, costruttore di copia e operatore di assegnamento non modificano il valore del parametro, passato per riferimento

```
template <typename Elem>
class Queue {
  public:
    Queue();                        // costruttore (capacità illimitata)
    Queue(int);                     // costruttore (parametro: capacità)
    Queue(const Queue&);            // costruttore di copia
    Queue& operator=(const Queue&); // overloading dell'assegnamento
    ~Queue();                       // distruttore
    bool empty() const;             // query: true sse la pila è vuota
    void put(Elem);                 // inserimento di un elemento
    Elem get();                     // rimozione di un elemento (restituito)
  private:
    #include "queue-private.h"      // membri dipendenti dall'implementazione
};
```

implementazione di coda FIFO di I categ.

possiamo estendere la già vista implementazione della coda FIFO mediante lista, a un ADT di prima categoria che implementa la precedente interfaccia, se aggiungiamo alle definizioni già date (non replicate qui) le seguenti, rispettivamente nella parte privata e, in quella pubblica, per costruttore di copia, operatore di assegnamento e distruttore:

nel file **queue-private.h**:

```
  void deletelist()
  { for plink t = head; t != 0; head = t) { t = head->next; delete head; } }
```

nel file **queue.c++**:

N.B. il costruttore di copia usa l'operatore di assegnamento ridefinito (appresso) per effettuare la copia dell'argomento

```
template <typename Elem> Queue<Elem>::Queue(const Queue& q) { head = 0; *this = q; }
template <typename Elem> Queue<Elem>& Queue<Elem>::operator=(const Queue<Elem>& q)
{ if (this == &q) return *this;
  deletelist(); plink t = q.head;
  while (t != 0) { put(t->elem); t = t->next; }
  return *this;
}
template <typename Elem> Queue<Elem>::~Queue() { deletelist(); }
```

client di una coda FIFO di I categoria

il programma simula l'accodamento di N richieste di servizio in un sistema casuale di M code FIFO

ciascun accodamento è seguito dalla rimozione di una richiesta da una coda scelta casualmente, se tale coda non è vuota, e quindi dall'output dello stato delle M code

per il funzionamento corretto del programma è necessario che l'ADT sia di prima categoria

SimQueues.c++:

```
#include <iostream>
#include <cstdlib>
#include "queue.h"
using namespace std;
static const int M = 4;
int main(int argc, char* argv[])
{ int N = atoi(argv[1]);
  Queue<int> queues[M];
  for (int i = 0; i < N; i++, cout << endl)
  { int in = rand() % M, out = rand() % M;
    queues[in].put(i); cout << i << " in ";
    if ( !queues[out].empty() ) cout << queues[out].get() << " out";
    cout << endl;
    for (int k = 0; k < M; k++, cout << endl)
      { Queue<int> q = queues[k]; cout << k << ": ";
        while ( !q.empty() ) cout << q.get() << " ";
} }   }
```

istanziazione di template

la compilazione separata di implementazione e client della coda FIFO testè viste non segnala alcun problema, tuttavia il linking riserva una sgradevole sorpresa:

```
$ c++ -o bin/SimQueues queue.o SimQueues.o
SimQueues.o: In function `main':
SimQueues.c++:(.text+0xb3): undefined reference to `Queue<int>::Queue()'
SimQueues.c++:(.text+0x16b): undefined reference to `Queue<int>::put(int)' [...]
```

ciò accade perché la compilazione separata dell'implementazione del template non genera automaticamente le istanze delle definizioni dei template di funzione richieste dal client

soluzione: aggiungere la compilazione (separata) di **dichiarazioni di istanziazione di template**, **SimQueueTFI.c++:**

```
#include "queue.c++"
template Queue<int>::Queue();
template void Queue<int>::put(int);
[...]
```

o, più semplicemente, per tutte le funzioni dell'istanza del template di classe, **SimQueueTCI.c++:**

```
#include "queue.c++"
template class Queue<int>::Queue;
```

procedure di costruzione di programmi

quando si sviluppano più client e/o più implementazioni di uno stesso ADT, è utile predisporre la costruzione di ciascun programma dalla corrispondente combinazione di client e implementazioni

una possibile organizzazione, per lo sviluppo basato su un certo ADT, ne colloca i moduli come segue:

- `clients/` : sorgenti dei client e rispettivi file di dichiarazioni di istanziazione di template
- `include/` : interfacce, con strutture dati private in `include/concrete/`
- `adt_x/` : sorgenti dell'implementazione `x` dell'ADT
- `build/` : procedure di costruzione e moduli oggetto
- `bin/` : programmi eseguibili

interfacce e implementazioni possono essere rese visibili, nelle directory dove servono a tempo di compilazione, mediante link simbolici, magari creati dinamicamente dalle procedure di costruzione

ecco ad es. una shell per la costruzione del programma di simulazione del sistema di code, assumendo l'esistenza di link a `include/queue.h` in `clients/` e in `queue_l/`, qui con link `queue-private.h` a `include/concrete/lista.h`, e della directory `build/simqueues_l/` :

```
cd ../clients
ln -s ../include/concrete/lista.h queue-private.h
ln -s ../queue_l/queue.c++ .
cd ../build/simqueues_l
c++ -c ../../clients/SimQueues.c++ ../../clients/SimQueueTCI.c++ ../../queue_l/queue.c++
c++ -o ../../bin/SimQueues_l *.o
rm ../../clients/queue-private.h ../../clients/queue.c++
```

esercizi

1. esercizio 4.48 del testo (Sedgewick, 2003), a p. 174: realizzare una implementazione con array dell'ADT coda casuale tale che le operazioni di inserimento e rimozione siano eseguite in tempo costante
2. modificare l'implementazione mediante array della pila senza duplicati, per imporre la disciplina "dimentica il vecchio (accodando il nuovo in ordine di arrivo)"
3. modificare l'implementazione mediante array della coda FIFO, vista nella lezione precedente, per implementare una coda FIFO senza duplicati, secondo la disciplina "ignora il nuovo"
4. esercizio 4.64 del testo (Sedgewick, 2003), a p. 191: trasformare l'ADT delle relazioni di equivalenza proposto nella lezione precedente in un ADT di prima categoria
5. estendere l'implementazione mediante array della coda FIFO vista nella lezione precedente, per trasformarla in un ADT di prima categoria