Università Roma La Sapienza
Corsi di Laurea
Informatica/Tecnologie Informatiche

Abstract Data Types

Prof. Stefano Guerrini
guerrini@di.uniroma1.it

Programmazione II (can. P-Z)
A.A. 2005-06

Tipo di Dato

- Un ***tipo di dato*** è caratterizzato da
  1. un insieme di valori
  2. un insieme di operazioni elementari
- Esempi.
  - *Interi*: valori interi compresi in un range finito con le usuali operazioni aritmetiche.
  - *Stringhe*: sequenze finite di caratteri con le operazioni che permettono di leggere/cambiare l'*i*-esimo carattere della stringa, di troncare una stringa, di concatenare due stringhe, etc.

Tipi di Dato Astratti /1

Un tipo di dato astratto o ADT (abstract data type) è un tipo di dato (valori e operazioni) completamente specificato indipendentemente da una sua particolare implementazione.

- Il comportamento delle operazioni del tipo di dato e l'interpretazione dei suoi valori è indipendente dalla particolare rappresentazione dei valori in un programma e da come sono implementate le operazioni.

Tipi di Dato Astratti /2

- I valori e le operazioni del tipo di dato sono definiti mediante equazioni o assiomi o altre descrizioni (semi-)formali.
- Dato che un ADT non è descritto da una implementazione ma in modo (semi-)formale, gli ADT permettono di
  - ragionare sulle proprietà delle operazioni
  - sulle relazioni tra ADT
  - sulla correttezza di una implementazione
  - ...

Tipi di Dato Astratti /3

- Un ADT è caratterizzato dalle seguenti proprietà:
  - definisce/esporta un **tipo**;
  - definisce/esporta un insieme di **operazioni**, la cosiddetta **interfaccia**;
  - le operazioni dell'interfaccia sono l'unico meccanismo di accesso alle strutture dati (l'effettiva **implementazione**) dell'ADT;
  - il dominio di applicazione del tipo e il comportamento delle sue operazioni sono definiti mediate assiomi e precondizioni.

Information hiding /1

- Il principio di separazione tra interfaccia e implementazione è normalmente chiamato **information hiding** o **encapsulation**.
- L'obiettivo dell'information hiding è quello di nascondere tutte quelle scelte di progetto che possono essere soggette a cambiamenti, proteggendo così le altre parti del programma da cambiamenti di tali scelte.
- Bisogna porre molta cura nella scelta e nella definizione dell'interfaccia. In particolare, l'interfaccia deve essere particolarmente stabile.

Information hiding /2

- Relativamente all'accesso ai dati della struttura, il corretto funzionamento delle parti di programma che usano un ADT è garantito dal
  - rispetto delle specifiche dell'interfaccia,
  - ovvero, dal corretto utilizzo delle operazioni.
- Il programmatore deve preoccuparsi solo della correttezza logica delle operazioni che vuole eseguire, ovvero, se corrispondono effettivamente all'algoritmo che vuole implementare.

Information hiding /3

- In un ADT, i dettagli sul modo in cui i dati sono memorizzati sono nascosti all'utilizzatore.
- L'accesso e la modifica dei valori memorizzati deve essere garantito da opportune funzioni dell'interfaccia.
- Questo è anche l'unico mezzo di accesso alla rappresentazione interna dei dati.

Modularità

- L'information hiding è un principio fondamentale per la progettazione e la realizzazione di software modulare.
- La corretta suddivisione in **moduli** e la corretta specifica delle loro interfacce permette di svilupparli e verificarli in modo indipendente.
- Ogni errore è ristretto ad un modulo (a meno che non si tratti di un errore di decomposizione).
- La correzione dell'errore implica la modifica/ riscrittura solo del modulo che contiene l'errore e non ha effetti sugli altri moduli.

Numeri complessi /1

- I numeri complessi $c = a+ib$ sono coppie di reali:
  - a è la parte reale
  - b è la parte immaginaria
- Le operazioni di somma (e sottrazione) sui complessi sono definite applicando le corrispondenti operazioni sui reali alle componenti omologhe:

$$(a_1+ib_1) + (a_2+ib_2) = (a_1+a_2)+i(b_1+b_2)$$

Numeri complessi /2

- Il prodotto tra due numeri complessi è definito da

$$(a_1+ib_1)\,(a_2+ib_2) = (a_1a_2-b_1b_2)+i(a_1b_2+a_2b_1)$$

- Il complemento di un numero complesso è

$$(a+ib)^* = (a-ib)$$

- Il quadrato del modulo di un numero complesso è

$$|a+ib|^2 = a^2+b^2$$

- posso ottenerlo per mezzo delle altre operazioni

$$|c|^2 = c\,c^*$$

Numeri complessi /3

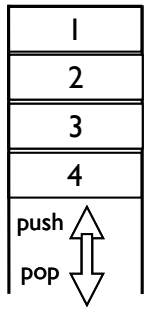
- *newComplex(a,b)* crea e ritorna un nuovo numero complesso pari a *Complex*(*a,b)*
- *Re(Complex(a, b)) = a* e *Im(Complex(a, b)) = b*
- Se c_1 = *Complex*(a_1, b_1) e c_2 = *Complex*(a_2, b_2)
  - $s(c_1, c_2)$ = *Complex*(a_1+a_2, b_1+b_2)
  - $p(c_1, c_2)$ = *Complex*($a_1*a_2+b_1*b_2$, $a_1*b_2+a_2*b_1$)
  - $ast(c_1)$ = *Complex*(a_1, $-b_1$)

Numeri complessi /4

- Come rappresentare i numeri complessi?
  a. Rappresento le coppie con un vettore:
     Re(c) = c[0] e *Im(c) = c[1]*
  b. Rappresento le coppie con una struct:
     Re(c) = c.re e *Im(c) = c.im*
- Per usare i numeri complessi così definiti non è necessario sapere come sono implementati. Ad esempio, definisco semplicemente
  mod(c) = sqrt(p(c,ast(c)))

Pile

- Una **pila** o **stack** è una struttura dati in cui vengono memorizzati e prelevati dati secondo una disciplina *Last-In-First-Out* (LIFO).
  - L'ultimo elemento inserito è il primo ad essere prelevato.
- Gli elementi vengono prelevati in ordine inverso a quello di inserimento.



Pile: funzioni interfaccia

1. *newStack()* *newStack: Stack*
   crea una nuova pila vuota *Empty*
2. *push(v,P)* *push: Data × Stack → Stack*
   inserisce l'elemento *v* nella pila *P*
3. *pop(P)* *pop: Stack → Stack*
   elimina dalla pila *P* l'ultimo elemento che vi è stato inserito
4. *top(P)* *top: Stack → Data*
   ritorna l'ultimo elemento che è stato inserito nella pila *P*
5. *isEmpty(P)* *push: Stack → Boolean*
   verifica se la pila *P* non contiene elementi (è vuota)

Pile: specifiche interfaccia

1. *newPila() = Emp* pila vuota
2. *pop(push(v, P)) = P*
3. *top(push(v, P)) = v*
4. *isEmpty(Emp) = true*
5. *isEmpty(push(v, P)) = false*

Code

- Una **coda** o **queue** è una struttura dati in cui vengono memorizzati e prelevati dati secondo una disciplina *First-In-First-Out* (FIFO).
  - Il primo elemento inserito è il primo ad essere prelevato.
- Gli elementi vengono prelevati rispettando l'ordine in cui sono inseriti.

Code: funzioni interfaccia

1. *newQ()* *newQ: Queue*
   ritorna una coda vuota
2. *enq(v, Q)* *enq: Data × Queue → Queue*
   inserisce l'elemento *v* nella coda *Q*
3. *deq(Q)* *deq: Queue → Queue*
   rimuove il primo elemento della coda *Q*
4. *first(Q)* *first: Queue → Data*
   ritorna il primo elemento della coda *Q*
5. *isEmpty(Q)* *isEmpty: Queue → Boolean*
   verifica se la coda *Q* non contiene elementi

Code: specifiche interfaccia

1. *newQueue() = Emp* coda vuota
2. *deq(enq(v, Emp)) = Emp*
3. *deq(enq(v, enq(w,Q))) = enq(v,deq(enq(w,Q)))*
4. *first(enq(v, Emp)) = v*
5. *first(enq(v, enq(w,Q))) = first(enq(w,Q))*
6. *isEmpty(Emp) = true*
7. *isEmpty(enq(v, Q)) = false*