Corso “Programmazione 1”
Capitolo 12: Strutture Dati Astratte

Docente: Marco Roveri - marco.roveri@unitn.it
Esercitatori: Giovanni De Toni - giovanni.detoni@unitn.it
Stefano Berlato - stefano.berlato-1@unitn.it

C.D.L.: Informatica (INF)
A.A.: 2021-2022
Luogo: DISI, Università di Trento
URL: https://bit.ly/2VgfYwJ

Ultimo aggiornamento: 17 novembre 2021

Terms of Use and Copyright

The material (text, figures) in these slides is authored mostly by Roberto Sebastiani, with contributions by Marco Roveri, Alessandro Armando, Enrico Giunchiglia e Sabrina Recla.

Outline

Tipo di Dato Astratto/Abstract Data Type

Un tipo di dato astratto (TDA)/abstract data type (ADT) è un un insieme di valori e di operazioni definite su di essi in modo indipendente dalla loro implementazione

- Per definire un tipo di dato astratto occorre specificare:
  - i dati immagazzinati
  - le operazioni supportate
  - le eventuali condizioni di errore associate alle operazioni
- Per lo stesso TDA si possono avere più implementazioni
  - diversa implementazione, diverse caratteristiche computazionali (efficienza, uso di memoria, ecc.)
  - stessa interfaccia (stessi header di funzioni, riportati in un file `.h`)
    $\Longrightarrow$ implementazioni interscambiabili in un programma
- È spesso desiderabile nascondere l'implementazione di un TDA (information hiding): solo i file `.h` e `.o` disponibili

N.B.: La nozione di TDA è la base della programmazione ad oggetti.

Esempio di Tipo di Dato Astratto

- Consideriamo la definizione di un tipo di dato astratto che rappresenta un punto nello spazio cartesiano $X \times Y$.
- Le operazioni che vogliamo effettuare su un punto (indipendentemente da come viene implementato) sono:
  - Crea un nuovo punto.
  - Ritorna la coordinata x (y) rispettivamente come double.
  - Assegna la coordinata x (y) rispettivamente.
  - Confronta due punti per vedere se sono uguali o diversi.
  - Stampa le coordinate di un punto.
  - Calcola la distanza tra due punti.
  - Somma due punti.
  - Verifica se tre punti stanno su una retta.

Esempio di Tipo di Dato Astratto - II

punto.h

```
// versione 1                          // // versione 2
struct Point {                         // struct Point {
  double x;                            //    double coord[2];
  double y;                            // };
};
// Definizione dei metodi dell'ADT Point
Point PointInit(void);
Point PointInit(const double x, const double y);
double Point_GetX(const Point & p);
double Point_GetY(const Point & p);
void Point_SetX(Point & p, double x);
void Point_SetY(Point & p, double y);
bool Point_Equal(const Point & P1, const Point & P2);
void Point_Print(const Point & P, const char * n);
double Point_GetDistance(const Point & P1, const Point & P2);
Point Point_Sum(const Point & P1, const Point & P2);
bool Point_Aligned(const Point & P1, const Point & P2, const Point & P3);
```

Esempio di Tipo di Dato Astratto - III

main.cc

```
#include <iostream>
using namespace std;
#include "point.h"

int main() {
  double t;
  Point P2, P1, P3;
  P1 = PointInit(5.0, 5.0);
  Point_Print(P1, "Coordinate del Punto P1");
  cout << "Inserire coordinate di un Punto P2" << endl << "X = " ;
  cin >> t;
  Point_SetX(P2, t);
  cout << "Y = "; cin >> t;
  Point_SetY(P2, t);
  cout << "La distanza tra P1 e P2 e': "
       << Point_GetDistance(P1, P2) << endl;
```

Esempio di Tipo di Dato Astratto - III

main.cc (cont)

```
  if (Point_Equal(P1, P2)) {
    P3 = Point_Sum(P1, P2);
  }
  else {
    P3 = PointInit(1.0, 1.0);
  }
  Point_Print(P3, "Coordinate del Punto P3");
  if (Point_Aligned(P1, P2, P3)) {
    cout << "I tre punti risiedono su una retta" << endl;
  }
  else {
    cout << "I tre punti non risiedono su una retta" << endl;
  }
}
```

Esempio di Tipo di Dato Astratto - III

point.cc — versione 1

```
Point PointInit() {
  Point r = {0.0, 0.0};
  return r;
}



Point PointInit(const double x,
                const double y) {
  Point r = {x, y};
  return r;
}
```

point.cc — versione 2

```
Point PointInit() {
  Point r;
  r.coord[0] = 0.0;
  r.coord[1] = 0.0;
  return r;
}

Point PointInit(const double x,
                const double y) {
  Point r;
  r.coord[0] = x;
  r.coord[1] = y;
  return r;
}
```

Esempio di Tipo di Dato Astratto - IV

point.cc (cont) — versione 1

```
// Ritorna la coordinate X e Y di P
double Point_GetX(const Point & p) {
  return p.x;
}
double Point_GetY(const Point & p) {
  return p.y;
}

// Assegna le coordinate X e Y di P
void Point_SetX(Point & p,
                const double x) {
  p.x = x;
}
void Point_SetY(Point & p,
                const double y) {
  p.y = y;
}
```

point.cc (cont) — versione 2

```
// Ritorna la coordinate X e Y di P
double Point_GetX(const Point & p) {
  return p.coord[0];
}
double Point_GetY(const Point & p) {
  return p.coord[1];
}

// Assegna le coordinate X e Y di P
void Point_SetX(Point & p,
                const double x) {
  p.coord[0] = x;
}
void Point_SetY(Point & p,
                const double y) {
  p.coord[1] = y;
}
```

Esempio di Tipo di Dato Astratto - V

`point.cc` (cont) — indipendente dalla versione

```
// Predicato per controllare se due Punti sono uguali
bool Point_Equal(const Point & P1, const Point & P2) {
  return ((Point_GetX(P1) == Point_GetX(P2)) &&
          (Point_GetY(P1) == Point_GetY(P2)));
}

// Stampa coordinate di un punto P inserendo
// la stringa n prima della stampa delle coordinate
void Point_Print(const Point & P, const char * n) {
  cout << n << endl;
  cout << ".X = " << Point_GetX(P) << endl;
  cout << ".Y = " << Point_GetY(P) << endl;
}

// calcola la distanza tra due punti
double Point_GetDistance(const Point & P1, const Point & P2) {
  double dx = (Point_GetX(P1) - Point_GetX(P2));
  double dy = (Point_GetY(P1) - Point_GetY(P2));
  return sqrt(dx * dx + dy * dy);
}
```

Esempio di Tipo di Dato Astratto - V

`point.cc` (cont) — indipendente dalla versione

```
// Costruisci il punto risultante dalla somma delle
// rispettive coordinate di due punti P1 e P2
Point Point_Sum(const Point & P1, const Point & P2) {
  return PointInit(Point_GetX(P1) + Point_GetX(P2),
                   Point_GetY(P1) + Point_GetY(P2));
}

bool Point_Aligned(const Point & P1, const Point & P2,
                   const Point & P3) {
  return ((Point_GetY(P1) - Point_GetY(P2)) *
          (Point_GetX(P1) - Point_GetX(P3))) ==
    ((Point_GetY(P1) - Point_GetY(P3)) *
     (Point_GetX(P1) - Point_GetX(P2)));
}
```

- TDA `Point`:
  - `TDA/point.h`
  - `TDA/point.cc`
  - `TDA/point_main.cc`

Esempi Molto Importanti di Tipi di Dato Astratto

- Le Pile (Stack)
- Le Code (Queue)
- Gli Alberi (Tree)

Le Pile (Stack)

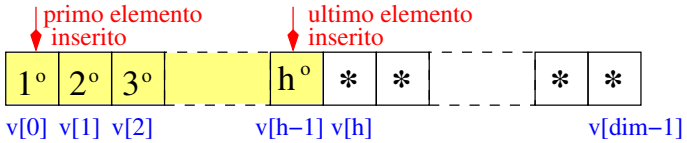
- Una pila è una collezione di dati omogenei (e.g., puntatori a struct) in cui gli elementi sono gestiti in modo LIFO (Last In First Out)
  - Viene visualizzato/estratto l'elemento inserito più recentemente
  - Es: una scatola alta e stretta contenente documenti
- Operazioni tipiche definite su una pila di oggetti di tipo `T`:
  - `init()`/`deinit()`: inizializza/deinizializza la pila
  - `push(T)`: inserisce elemento sulla pila; fallisce se piena
  - `pop()`: estrae l'ultimo elemento inserito (senza visualizzarlo); fallisce se vuota
  - `top(T &)`: ritorna l'ultimo elemento inserito (senza estrarlo); fallisce se vuota
- Varianti:
  - `pop()` e `top(T &)` fuse in un'unica operazione `pop(T &)`
  - talvolta disponibili anche `print()`
  - [`deinit()` non sempre presente]

Le Pile (Stack) II

Nota importante

In tutte le possibili implementazioni di una pila, le operazioni `push(T)`, `pop()`, `top(T&)` devono richiedere un numero costante di passi computazionali, indipendente dal numero di elementi contenuti nella pila!

Implementazione di una pila mediante array



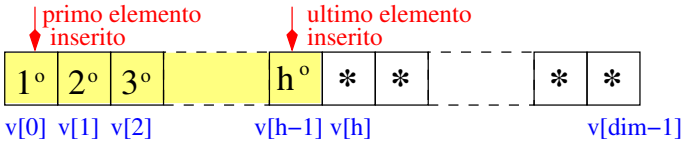
- Dati: un intero `h` e un array `v` di `dim` elementi di tipo `T`
  - `v` allocato staticamente o dinamicamente
  - `h` indice del prossimo elemento da inserire (inizialmente 0)

  $\Longrightarrow$ numero di elementi contenuti nella pila: `h`
  - pila vuota: `h==0`
  - pila piena: `h==dim`

  $\Longrightarrow$ massimo numero di elementi contenuti nella pila: `dim`

N.B.: `dim` elementi sempre allocati.

Implementazione di una pila mediante array II



- Funzionalità:
  - `init()`: pone `h=0` (alloca `v` se allocazione dinamica)
  - `push(T)`: inserisce l'elemento in `v[h]`, incrementa `h`
  - `pop()`: decrementa `h`
  - `top(T &)`: restituisce `v[h-1]`
  - `deinit()`: dealloca `v` se allocazione dinamica

Esempi su pile di interi

- semplice stack di interi come struct:
  - `STACK_QUEUE_ARRAY/struct_stack.h`
  - `STACK_QUEUE_ARRAY/struct_stack.cc`
  - `STACK_QUEUE_ARRAY/struct_stack_main.cc`
- uso di stack per invertire l'ordine:
  - `STACK_QUEUE_ARRAY/struct_stack.h`
  - `STACK_QUEUE_ARRAY/struct_stack.cc`
  - `STACK_QUEUE_ARRAY/struct_reverse_main.cc`

  (`struct_stack.h|.cc` stessi del caso precedente)

Le Code (Queue)

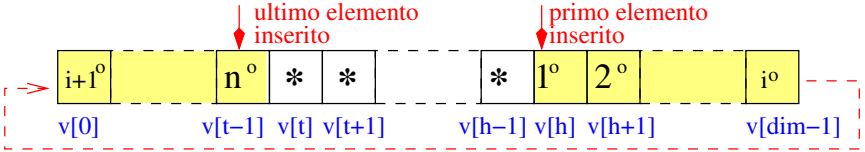
- Una coda è una collezione di dati omogenei in cui gli elementi sono gestiti in modo FIFO (First In First Out)
  - Viene visualizzato/estratto l'elemento inserito meno recentemente
  - Es: una coda ad uno sportello
- Operazioni tipiche definite su una coda di oggetti di tipo `T`:
  - `init()`/`deinit()`: inizializza/deinizializza la coda
  - `enqueue(T)`: inserisce elemento sulla coda; fallisce se piena
  - `dequeue()`: estrae il primo elemento inserito (senza visualizzarlo); fallisce se vuota
  - `first(T &)`: ritorna il primo elemento inserito (senza estrarlo); fallisce se vuota
- Varianti:
  - `dequeue()` e `first(T &)` fuse in un'unica operazione `dequeue(T &)`
  - talvolta disponibili anche `print()`
  - [`deinit()` non sempre presente]

Le Code (Queue) II

Nota importante

In tutte le possibili implementazioni di una coda, le operazioni `enqueue(T)`, `dequeue()`, `first(T &)` devono richiedere un numero costante di passi computazionali, indipendente dal numero di elementi contenuti nella coda!

Implementazione di una coda mediante array



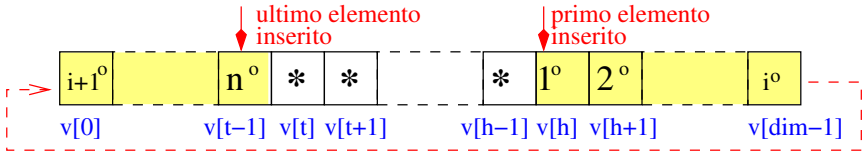
- Idea: buffer circolare: `succ(i)==(i+1)%dim`
- Dati: due interi `h,t` e un array `v` di `dim` elementi di tipo `T`
  - `v` allocato staticamente o dinamicamente
  - `h` indice del più vecchio elemento inserito (inizialmente 0)
  - `t` indice del prossimo elemento da inserire (inizialmente 0)

  $\Longrightarrow$ num. di elementi contenuti nella coda: `n=(t>=h ? t-h : t-h+dim)`
  - coda vuota: `t==h`
  - coda piena: `succ(t)==h`

  $\Longrightarrow$ massimo numero di elementi contenuti nella coda: `dim-1`

N.B.: `dim` elementi sempre allocati.

Implementazione di una coda mediante array II



- Funzionalità:
  - `init()`: pone `h=t=0` (alloca `v` se allocazione dinamica)
  - `enqueue(T)`: inserisce l'elemento in `v[t]`, "incrementa" `t` (`t=succ(t)`)
  - `dequeue()`: "incrementa" `h`
  - `first(T &)`: restituisce `v[h]`
  - `deinit()`: dealloca `v` se allocazione dinamica

Esempi su code di interi

- semplice coda di interi come struct:
  - `STACK_QUEUE_ARRAY/struct_queue.h`
  - `STACK_QUEUE_ARRAY/struct_queue.cc`
  - `STACK_QUEUE_ARRAY/struct_queue_main.cc`

Esercizi proposti

Vedere file `ESERCIZI_PROPOSTI.txt`