52. Strutture dati astratte

Un **tipo di dato astratto (TDA)/abstract data type (ADT)** è 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.) (indipendentemente da come viene implementata l'operazione)
- stessa interfaccia (stessi header di funzioni, riportati in un file .h). Le implementazioni interscambiabili in un programma

È spesso desiderabile nascondere l'implementazione di un TDA (**information hiding**): rendendo solo i file *.h* e *.o* disponibili. Si può implementare facendo delle forward declaration delle struct usate in TUTTI i file *.cc* e definendole bene nei suddetti file.

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

53. Esempio di Tipo di Dato Astratto

Consideriamo la definizione di un tipo di dato astratto che rappresenta un punto nello spazio cartesiano X x Y .

Le operazioni che vogliamo effettuare su un punto (indipendentemente da come viene implementato) sono:
- Crea un nuovo punto.
- Ritorna la coordinata x e/o y rispettivamente come *double*.
- Assegna la coordinata x e/o y ispettivamente.
- 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.

punto.h

//Definizione/dichiarazione struct

```
//versione 1
struct Point{
double x;
double y;
};
```

```
//versione 2
struct Point{
double coord[2];
};
```

```
//versione 3
struct Point;
```

Notiamo come la versione 3 faccia una *forward declaration* hidando i campi, che poi verranno definiti in tutti i file *.cc* in cui si usa la struttura o tutti con la versione 1 o tutti con la versione 2.

//Definizione dei metodi dell'ADT Point

```
Point PointInit(void);
Point PointInit(const double, const double);
```

```
double Point_GetX(const Point &);
double Point_GetY(const Point &);
void Point_SetX(Point &, double);
void Point_SetY(Point &, double);
bool Point_Equal(const Point &, const Point &);
void Point_Print(const Point &, const char *);
double Point_GetDistance(const Point &, const Point &);
Point Point_Sum(const Point &, const Point &);
bool Point_Aligned(const Point &, const Point &, const Point &);
```

punto_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;

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;
}

return 0;
}
```

point.cc (definizione di tutte le funzioni usate nel *main*)

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;
}
```

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;
}
```

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;
}
```

Versione 2

```
double Point_GetX(const Point & p) {
return p.coord[0];
}

double Point_GetY(const Point & p) {
return p.coord[1];
}

void Point_SetX(Point & p, const double x) {
p.coord[0] = x;
}

void Point_SetY(Point & p, const double y) {
p.coord[1] = y;
}
```

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);
}

// 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)));
}
```

53.1 Compilation conditional

Entrambe le versioni possono essere implementate e si può usare una o l'altra grazie alla compilation conditional.

Ad esempio nel file *.h* quando definiamo le *structs* possiamo inserire questo ***pattern:***

…

#define VALUE 0
struct Point{
#if VALUE
double x;
double y;
#else
double coord[2];
#endif
};

Con questo costrutto il compilatore eseguirà:
-le istruzioni nell'*if* (versione uno della *struct*): se *value* avrà valore 1 (=*true*)
-le istruzioni nell'*else* (versione due della *struct*): se *value* avrà valore 0 (= *false*)

Il valore di *value* viene definito da noi andando a cambiarlo manualmente scrivendo sul file *.h* 0 o 1 accanto a *value*.
Oppure si può compilare i file *.cc* usando questo flag: ***g++ -DVALUE=1 <file.cc> …***
(in quest'ultimo caso bisogna OMETTERE l'istruzione *#define value <numero>*)
Poi andando a compilare cambierà, passando da una compilazione all'altra.

…

Nel file *point_main.cc* si può usare la compilation conditional per dire all'utente che tipo di compilazione è stata eseguita

```
#if VALUE
  cout << "Hai scelto la compilazione 1";
#else
  cout << "Hai scelto la compilazione 0";
#endif
```

Nel file *point.cc* si riflette che per le versioni che sopra avevano due versioni, ora dovremmo effettuare un'unica versione usando la compilation conditional:

```
Point PointInit() {
#if VALUE
  Point r = {0.0, 0.0};
#else
  Point r;
  r.coord[0] = 0.0;
  r.coord[1] = 0.0;
#endif
  return r;
}
```

54. Le pile (o stack) implementate con array

Una pila è una collezione di dati omogenei (puntatori a struct) in cui gli elementi sono gestiti in modo LIFO (Last In First Out): viene visualizzato e/o estratto l'elemento inserito più recentemente. Esempio: 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.

Anche qua bisogna prestare attenzione ai casi limite:

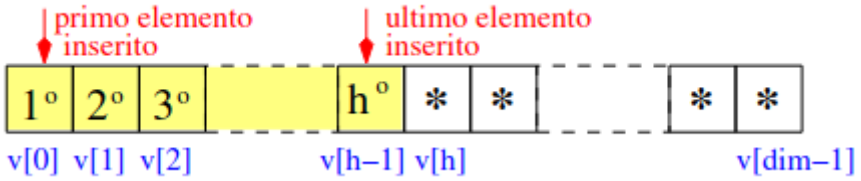
- pila piena (*push*)
- pila vuota (*pop* e *top*)

Varianti:

- *pop()* e *top(T &)* fuse in un'unica operazione: ***pop(T &)*** (che estrae e ritorna l'ultimo elemento inserito)
- talvolta disponibili anche *print()* (stampa tutti gli elementi)
- *deinit()* non sempre presente, magari implementato con *pop()*

N.B.! Essendo le pile dei TDA: 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!

54.1 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). Il numero di elementi contenuti nella pila è *h* (infatti dopo ogni inserimento *h* viene incrementato di 1).
  - pila vuota: *h==0*
  - pila piena: *h==dim*

! Massimo numero di elementi contenuti nella pila: ***dim***.

Funzionalità delle funzioni:

- *init()*: pone *h=0* (e alloca *v* se allocazione dinamica)
- *push(T)*: inserisce l'elemento in *v[h]*, incrementa *h*
- *pop()*: decrementa *h*
- *top(T &)*: restituisce *v[h-1]*
- *pop (T &):* restituisce *v[h-1]* e decrementa *h*
- *print()*: stampa l'array da *v[0]* a *v[h-1]*
- *deinit()*: dealloca v se allocazione dinamica

54.2 Funzioni di una pila

Implementazione statica:
```
struct stack
{
  int indice;
  int elem[dim];
};
```

Implementazione dinamica:
```
struct stack
{
  int indice;
  int * elem;
};
```

Firma delle funzioni:
```
void init(stack & );
void deinit(stack & );
retval push (int, stack &);
retval top (int &, const stack &);
retval pop (stack &);
void print(const stack &);
```

Ricordiamo che *push, top* e *pop* falliscono rispettivamente se la pila è piena e vuota (*top* e *pop*).

Quindi noi definiamo globalmente (insieme a ***const int dim***) una variabile di tipo enumerativo:
enum retval {FAIL,OK};
Le funzioni ritornano *FAIL* se l'operazione non è andata a buon fine, *OK* se invece ha funzionato. Per questo scopo, nelle funzioni viene definita una variabile di tipo *enum ris*, e, dopo aver controllato se la pila è piena o vuota, le viene assegnato il valore *FAIL* o *OK* e viene ritornato *ris*. Nel main, verificando il valore di ritorno alla funzione, si fanno opportuni messaggi all'utente.
La comodità è anche che a *FAIL* è associato il valore 0 e a *OK* il valore 1 (ad altri eventuali messaggi di errore il valore 2, 3, 4, …).

Importante: passare la struttura per riferimento (anteponendo *const* nelle funzioni ove non viene modificata).

(Nel main si implementa un menu' testuale per l'utente ove può scegliere le varie funzioni).

init e *deinit*
```
void init(stack & s){
s.indice=0;
s.elem = new int [dim]; //omesso in caso di implementazione statica
}

void deinit(stack& s){                    //va messo alla fine di tutte le operazioni
delete[] s.elem; //omesso in caso di implementazione dinamica;
}
```

top e *pop*
```
retval top(int & n, const stack& s){   //modifca n con il valore che viene ritornato
retval res;
if(empty(s)){                    //empty(s) verifica se h==0, cioè se la lista è vuota (ritorna un bool)
        res=FAIL;
} else {
        n=s.elem[s.indice-1];
        // s.indice--;    //nel caso di pop e top unito
```

```
        res=OK;
}
return res;
}

retval pop (stack& s){
retval res;
if(empty(s)){
res=FALSE;
} else {
s.indice--;
res=OK;
}
return res;
}
```

<u>push</u>

```
retval push(int n, stack& s){
retval res;
if(full(s)){                    //full(s) verifica se h==dim, cioè se la lista è piena (ritorna un bool)
res = FAIL;
} else{
s.elem[s.indice]=n;
s.indice++;
res=OK;
}
return res;
}
```

<u>print</u>

```
void print(stack & s){
int i;
for(i=0;i<s.indice;i++){
cout << "Elemento " << i+1 << " " << s.elem[i] << endl;
}
}
```

Per stampare dall'ultimo al primo

```
for(int i=s.indice-1;i>=0;i++)
  cout << s.elem[i] << " ";
```

Oppure così se so quanti elementi ci sono:

```
for(int i=0;i<dim;i++){
top(val,s);
pop(s);
cout << val << " ";
}
```

55. Le code (o queue) implementate con array
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.
Esempio: una coda ad uno sportello.

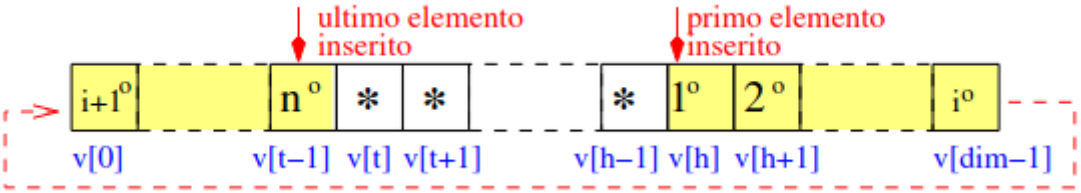
Operazioni tipiche definite su una coda di oggetti di tipo *T*:
- ***init()/deinit()***: inizializza/deinizializza la coda
- ***enqueue(T)***: inserisce elemento sulla coda; fallisce se piena.
- ***dequeue()***: estrae il primo elemento inserito (senza visualizzarlo); fallisce se vuota.
- ***first(T &)***: ritorna il primo elemento inserito (senza estrarlo); fallisce se vuota.

Varianti:
- *dequeue()* e *first(T &)* fuse in un'unica operazione *dequeue(T &)* (ritorna il primo elemento inserito e lo estrae)
- talvolta disponibili anche *print()* (stampa la coda)
- *deinit()* non sempre presente (in caso di allocazione statica)

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

Casi limite: lista vuota nel caso di *dequeue* e *first*, lista piena nel caso di *enqueue*.

55.1 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 inserite (inizialmente 0)

Numero di elementi contenuti nella coda: *num = (t>h ? t-h : t-h+dim)*
- coda vuota *t==h*
- coda piena *succ(t)==h*

Massimo di elementi contenuti in una coda *dim-1*

N.B.! Ci sono *dim* elementi sempre allocati

Funzionalità delle funzioni
- *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* (*h=succ(h)*)
- *first(T &)*: restituisce *v[h]*
- *deinit()*: dealloca *v* se allocazione dinamica

55.2 Implementazione di una coda

Implementazione di una coda con un array statico

```
struct queue
{
  int head, tail;
  int elem[DIM+1];
  int size = DIM+1;    //superfluo
};
```

Implementazione di una coda con un array dinamico

```
struct queue
{
  int head, tail;
  int * elem;
  int size;         //superfluo
};
```

Firma delle funzioni

```
void init (queue &);
void deinit (queue &);
retval enqueue(int,queue &);
retval first(int &,const queue &);
retval dequeue(queue &);
void print (const queue &);
```

Anche qua passiamo la struttura per riferimento e usiamo *retval*. Definiamo globalmente la dimensione della coda con *const int DIM=100;* (nel file *.h*) così sarà visibile a tutti. A questo proposito *size* come campo della struttura non serve.

succ() (trova la posizione successiva dato un indice)

```
(static) int next(int index, const queue & q){
        return (index+1)%q.size;
        //return (index+1)%DIM;
}
```

init() e *deinit()*

```
void init(queue & q){
q.tail=q.head=0;
  //q.size=DIM+1;  superfluo e solo in caso di implementazione dinamica
  q.elem = new int [q.size];   //in caso di implementazione dinamica
}

void deinit(queue & q){
delete [] q.elem;
}
```

enqueue e *dequeue*

```
retval enqueue(int n, queue & q){
retval res;
if(full(q)){                //verifica se q è piena, verificando: next(q.tail,q)==q.head (ritorna bool)
res = FAIL;
```

```
} else {
q.elem[q.tail] = n;
q.tail = next(q.tail,q);
res=OK;
}
return res;
}

retval dequeue(queue & q){
retval res;
if(empty(q)){
res = FAIL;
} else {
q.head = next(q.head,q);
res=OK;
}
return res;
}
```

first()

```
retval first(int & n, const queue & q){          //salva in n il primo elemento inserito
retval res;
if(empty(q)){                    //verifica se q è vuota, verificando q.tail==q.head (ritorna bool)
res = FAIL;
} else {
n = q.elem[q.head];
q.head = next(q.head,q);        //in caso di first e dequeue insieme
res=OK;
}
return res;
}
```

print()

```
void print(queue & q){
int i;
for(i=q.head;i!=q.tail;i=next(i,q)){
cout << "Elemento " << i+1 << " " << q.elem[i] << " ";
}
}
```

55.3 Code a priorità

Potrebbe essere necessario sviluppare un programma che gestisca delle code a priorità, come per gestire ad esempio gli arrivi in ospedale. In questo caso occorre fare più code, una per ogni priorità.

Come funziona l'inserimento con *push()*? Chiedere all'utente la priorità di ciò che si deve inserire e inserirlo nella rispettiva coda.

Come funziona il *pop()* e il *top()*? Sostanzialmente si prende al coda a massima priorità, se non è vuota si stampano/restituiscono/estraggono l'elemento di quella coda. Per ogni operazione bisogna verificare se la coda di priorità maggiore è vuota: se sì, si verifica quella a priorità inferiore. Se anch'essa è vuota, si passa a quella di priorità inferiore e si verifica se è vuota. Così via finché non si arriva ad una coda non vuota, dove si smaltisce la prima persona. Se le code sono tutte vuote *pop()/top()* fallisce.

Come funzione il *print()*? Sostanzialmente come il *top()* e il *pop()* ovvero si stampa all'inizio la coda con priorità maggiore (se non vuota), poi quella subito minore e via via fino all'ultima.

Oppure realizzo una struct coda a priorità in cui oltre al dato da memorizzare si memorizza anche un intero che rappresenti la priorità. Ad ogni push bisogna inserire dato e priorità. Con top e pop bisogna trovare il primo con priorità massima e restituirlo. Se non esiste, priorità inferiore, finché non si arriva alla fine.