tipo di dato astratto o ADT

(acronimo della versione inglese: Abstract Data Type)

Un ADT è un insieme di valori e di operazioni definite su di essi in modo indipendente da una particolare implementazione

Uno dei più semplici ADT è la pila (stack):
una pila è una collezione di oggetti in cui solo l'elemento inserito più recentemente può essere rimosso.
L'ultimo elemento inserito è in cima alla pila.
Le operazioni di base sono push (inserimento) e pop (estrazione).
Spesso si aggiungono pilaVuota, pilaPiena e top (lettura elemento in cima).

ADT: le OPERAZIONI

l'idea chiave di astrazione sui dati è che un tipo è caratterizzato dalle operazioni consentite su di esso:

un numero è qualcosa che si può moltiplicare, sommare, ...

una stringa è qualcosa che si può concatenare, spezzare in più stringhe, ...

ADT

Quando si costruisce un ADT ci si concentra su che *cosa* si vuole ottenere con le operazioni (la specificazione) e non su *come* definirle (l'implementazione)

I dati sono creati e manipolati esclusivamente usando le operazione dell'ADT (astrazione).

Un tipo può essere

- **generico: una collezione, un vettore, un albero**
- **specifico: un archivio di impiegati, una rubrica telefonica**

l'importante è non mescolare caratteristiche generali e specifiche.

La costruzione del tipo deve essere indipendente dalla rappresentazione in memoria dei dati.

ADT: specificazione e implementazione o rappresentazione

Si specifica un tipo di dato astratto specificando le sue operazioni. Queste costituiscono l'interfaccia dell'ADT: I dati sono costruiti, letti, modificati, cancellati solo attraverso l'interfaccia

•L'implementazione è data da un scelta per

• le strutture dati (i tipi) per memorizzare le informazioni

• il codice per le operazioni

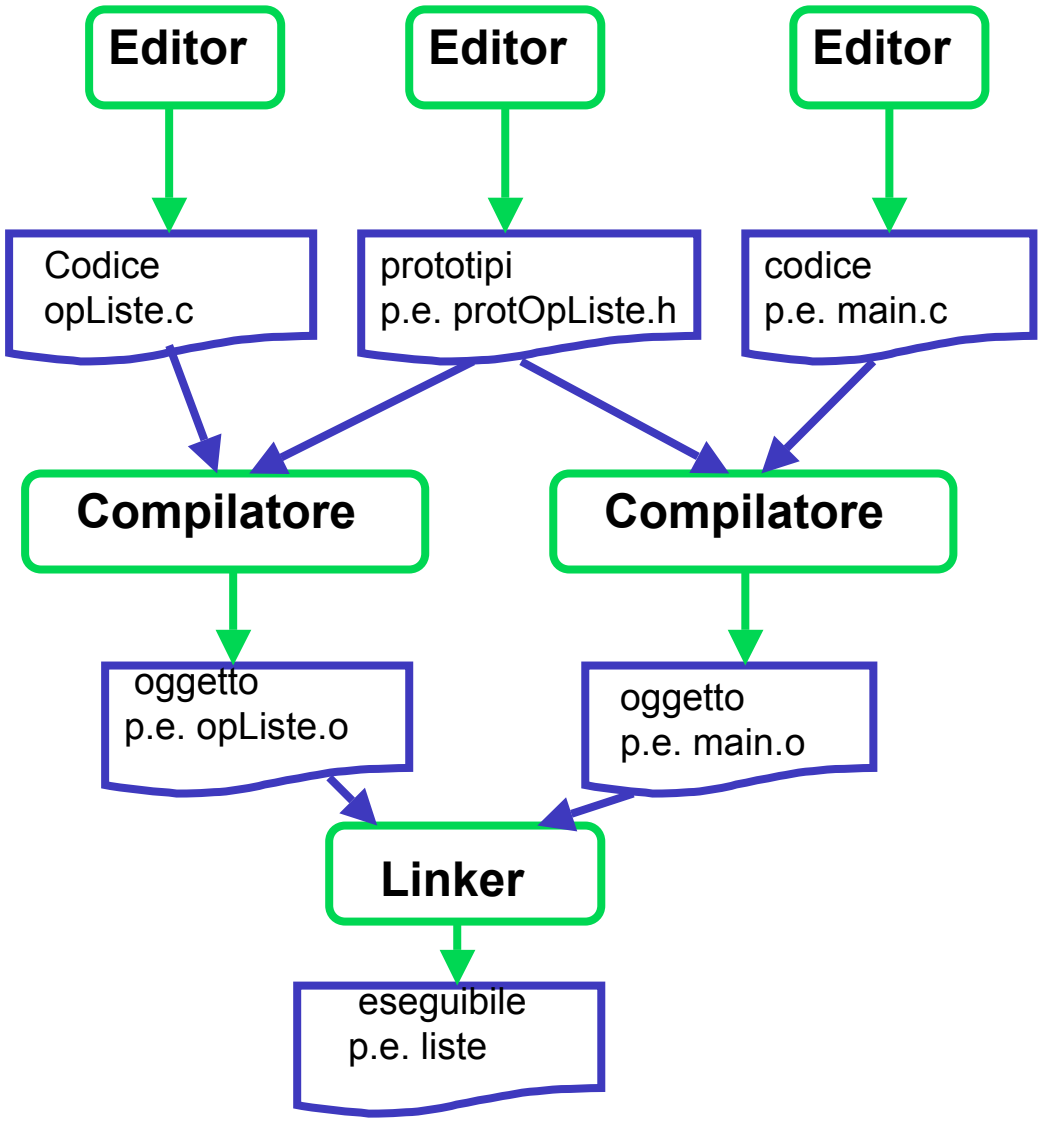
In C l'interfaccia è un insieme di prototipi di funzioni.

Ogni ADT ha un'interfaccia standard, ma può avere diverse implementazioni

ESEMPIO: L' ADT PILA può essere implementata su un lista concatenata o su un vettore

L'utente accede solo alla specificazione, quindi non conosce e non si preoccupa dell'implementazione.

Dai files del programma al codice eseguibile



Sotto Unix o Linux

cc -c mio.c

-c è un opzione che dice al compilatore di fermarsi dopo aver prodotto il codice oggetto corrispondete al codice C nel file di testo mio.c

Se tutto è andato bene il compilatore ha prodotto un file mio.o.

cc -o finale mio.o tuo.o suo.o

l'opzione -o annuncia il nome del file eseguibile

esempio:

```
cc -c funzioni.c
cc -c prova.c
cc -o prova prova.o funzioni.o
```

infine per eseguire il programma si digita semplicemente
prova

In una pila

solo l’elemento in cima è accessibile!!

L’ultimo ad essere inserito è il primo ad essere estratto:
Last In First Out.

specificazione della pila: file Pila.h

```
typedef struct pila* PilaP;   /*nome  del tipo degli elementi*/


PilaP  costrPila(int numMaxEl);
/*alloca la memoria e inizializza una nuova pila  con al più numMaxEl elementi
prec: numMaxel >0
postc: restituisce un puntatore a una nuova pila, NULL se non c'è memoria */

void  distrPila(PilaP p);
/*prec: p è una pila creata con costrPila
postc: libera  la memoria impegnata da p */

int numElPila(PilaP p);
/*prec: p è una pila creata con costrPila
postc: restituisce il numero di elementi nella pila */
```

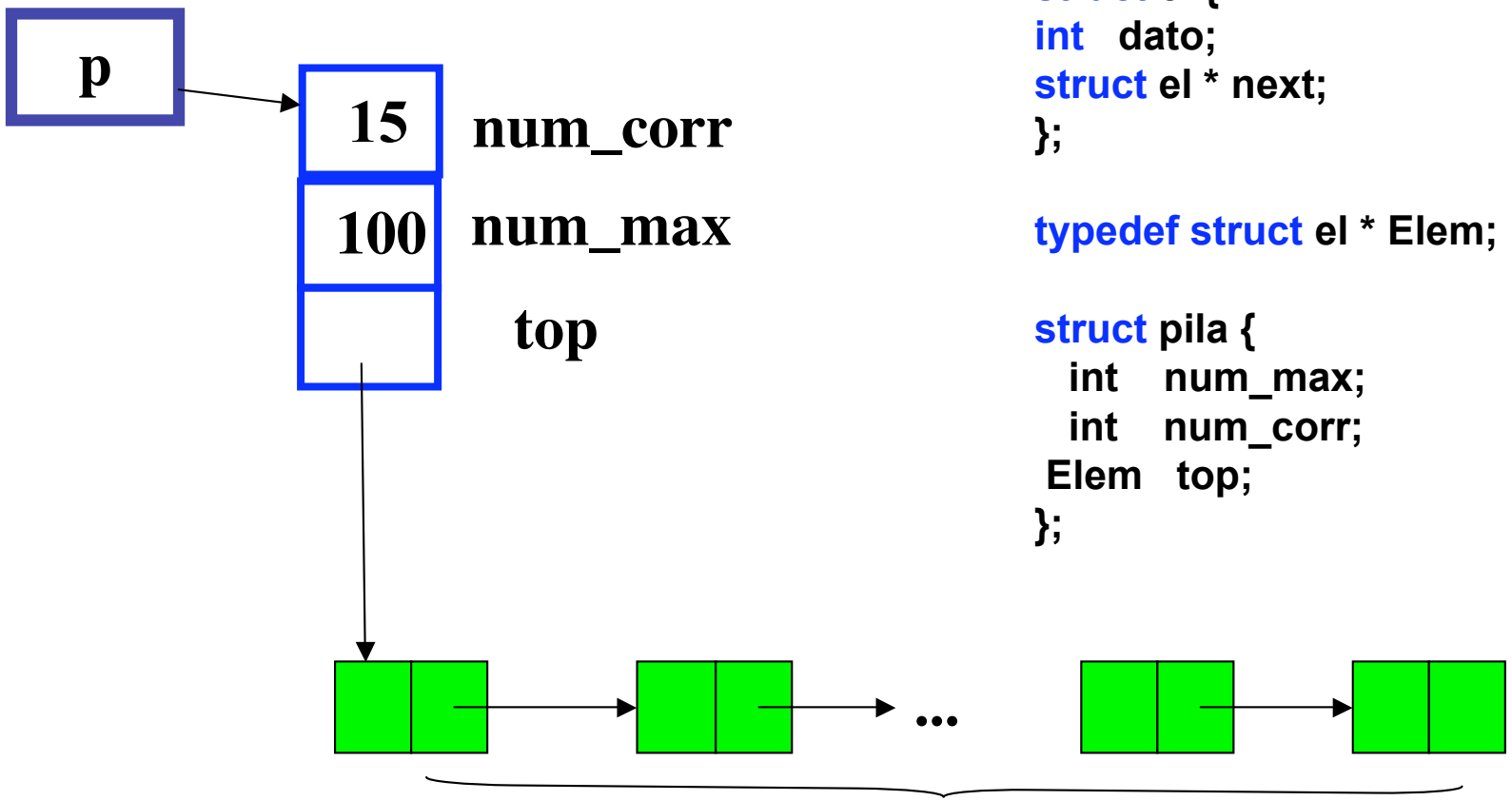
specificazione della pila

```
int   vuota(const PilaP p);
/* da' vero se la pila  p e' vuota, falso altrimenti
*prec: p è una pila creata con costrPila
postc: restituisce un valore !=0  se la pila è vuota, 0 altrimenti*/


int   piena(const PilaP p);
/* da' vero se la  pila p e' piena, falso altrimenti
*prec: p è una pila creata con costrPila
postc: restituisce un valore !=0  se la pila è piena, 0 altrimenti*/


void      push(int el, PilaP p);
/* inserisce el in cima alla pila  p
prec: p è una pila creata con costrPila e non piena
postc: el è in cima alla pila, se c'è abbastanza memoria, esce dal programma
altrimenti*/
```

specificazione della pila

```
int     pop(PilaP p);
/* elimina l'elemento in cima a p, se non e' vuota
*prec: p è una pila creata con costrPila e non vuota
post: restituisce, eliminandolo, l'elemento in cima alla pila*/

int     top(const PilaP p);
/* legge e restituisce l'elemento in cima a p, se non e' vuota
*prec: p è una pila creata con costrPila e non vuota
post: restituisce l'elemento in cima alla pila*/
```

implementazione di una pila su una lista




```
struct el {
int   dato;
struct el * next;
};

typedef struct el * Elem;

struct pila {
   int    num_max;
   int    num_corr;
 Elem   top;
};
```

implementazione di una pila su una lista: file pila.c

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include "pila.h"

struct el {                  /* un elemento dello pila e' un nodo della lista*/
  int   dato;
  struct el * next;
};

typedef struct el * Elem;

 struct pila {
  int    num_max;    /* qui memorizzaimo il numero massimo di elementi nella pila */
  int    num_corr;              /* teniamo conto del numero degli elementi */
  Elem   top;              /* il puntatore all'elemento in cima */
};
```

implementazione di una pila su una lista

```
PilaP costrPila(int numMaxEl)
/*alloca la memoria per una nuova pila  con numMaxEl al più elementi
prec: numMaxel >0
postc: restituisce un puntatore a una nuova pila, NULL se non c'è memoria */
{PilaP p;
assert(numMaxEl >0);
p = (PilaP) calloc(1,sizeof(struct pila));
assert(p);
p->num_max = numMaxEl;
return p;}

void distrPila(PilaP p)
/*prec:  p != NULL
postc: libera la memoria impegnata dalla pila */
{Elem temp;
assert(p);
while (p->top)
{temp = p->top;
p -> top = p-> top-> next;
free(temp);}
free(p);}
```

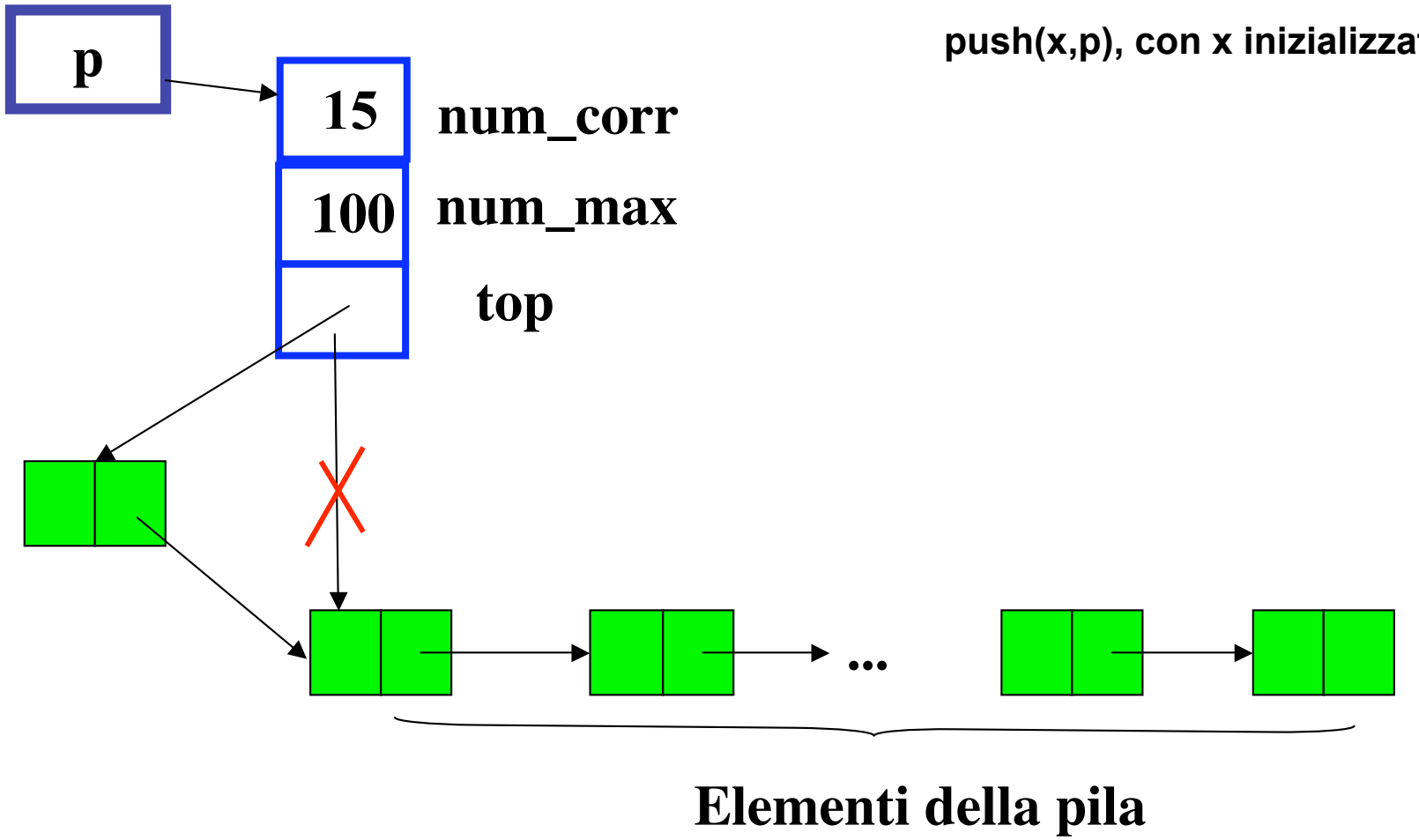
implementazione di una pila su una lista: le funzioni

```
/* selettore  */
int numElPila(PilaP p)
/*prec: p != NULL
postc: restituisce il numero di elementi nella pila */
{return p->num_corr;}

/* proprietà  */

int vuota(const PilaP p)
/* da' vero se la pila  p e' vuota, falso altrimenti
*prec: p != NULL
postc: restituisce un valore !=0  se la pila è vuota, 0 altrimenti*/
{assert(p);
  return p -> num_corr == 0;}

int piena(const PilaP p)
/* da' vero se la  pila p e' piena, falso altrimenti
*prec: p != NULL
postc: restituisce un valore !=0  se la pila è piena, 0 altrimenti*/
{assert(p);
 return p->num_max == p->num_corr;}
```

implementazione di una pila su una lista: le funzioni

```
void push(int el, PilaP p)
/* inserisce el in cima a p
prec: (p != NULL && !piena(p))
postc: el è in cima alla pila, se c'è abbastanza memoria,
altrimenti esce*/
{ Elem x;
  assert(p);
  assert(!piena(p));
  x = (Elem) malloc(sizeof(el));
  assert(x);
   x -> dato = el;
   x -> next = p -> top;
   p -> top = x;
   p -> num_corr++;
}
```

implementazione di una pila su una lista

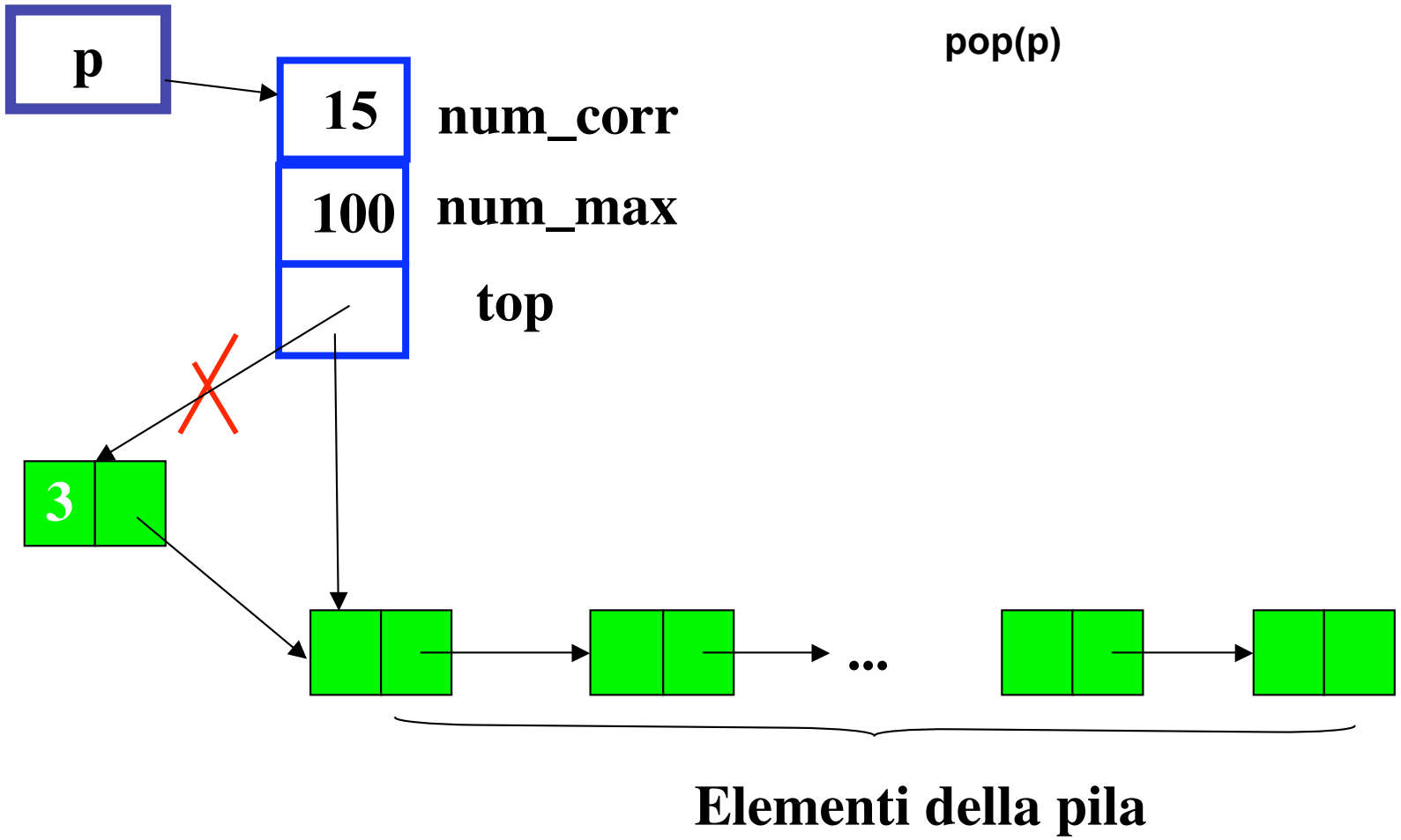


push(x,p), con x inizializzata a 3

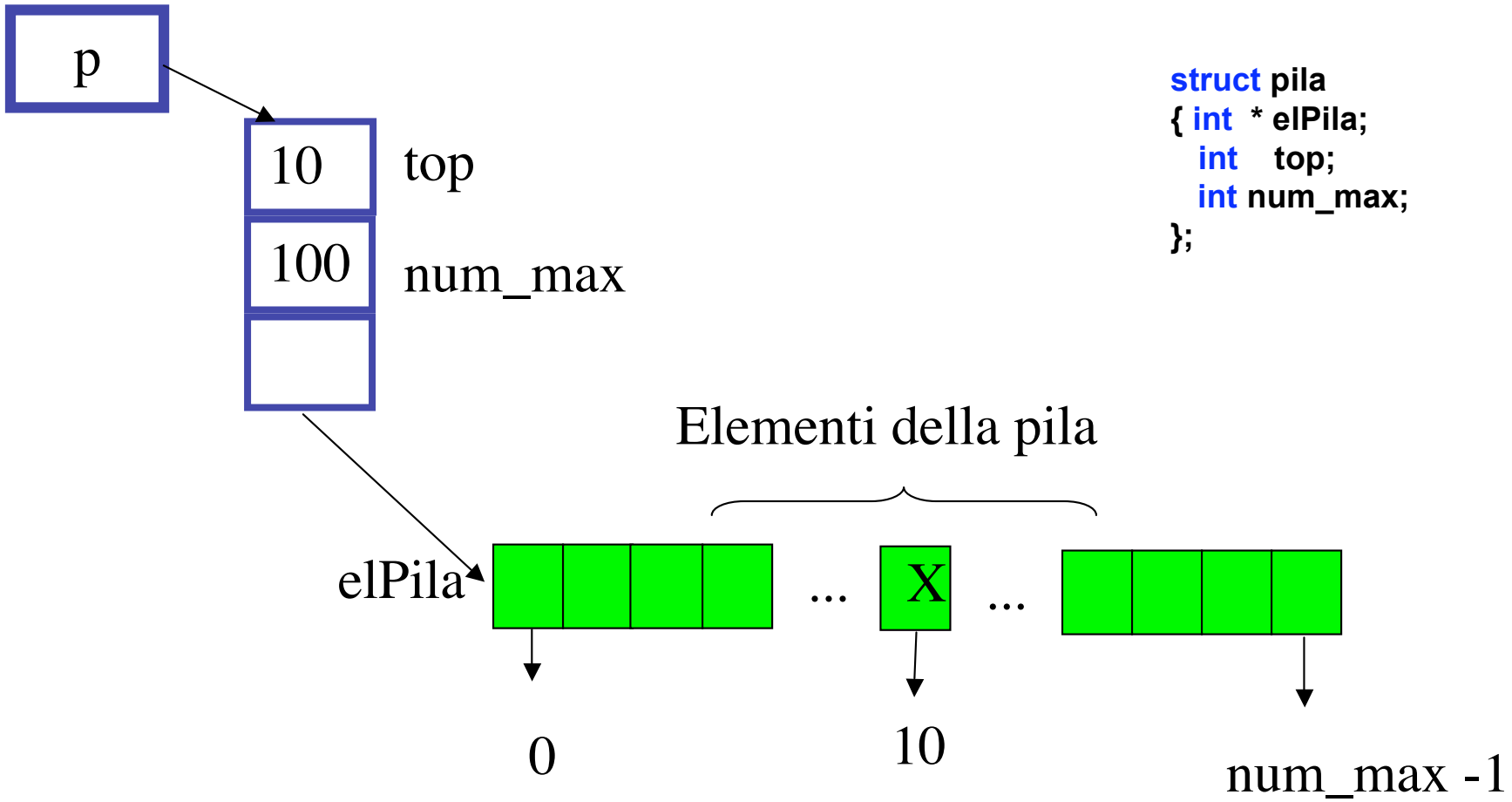
implementazione di una pila su una lista: le funzioni

```
int pop(PilaP p)
/* elimina l'elemento in cima a p, se non e' vuota
*prec: p!= NULL && e !vuota(p)
post: restituisce, eliminandolo, l'elemento in cima alla pila*/
{int  el;
 Elem   x;
 assert(p);assert(!vuota(p));
 el = p -> top -> dato;
  x = p -> top;
  p -> top = p -> top -> next;
  free(x);
  p -> num_corr--;
  return el;}

int top(const PilaP p)
/* legge e restituisce l'elemento in cima allo stack, se non e' vuoto
*prec: p != NULL && !vuota(p) */
{assert(p);
assert(!vuota(p));
return (p -> top -> dato);
}
```

implementazione di una pila su una lista



implementazione di una pila su un vettore




```
struct pila
{ int  * elPila;
  int    top;
  int num_max;
};
```

implementazione di una pila su un vettore

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include "pila.h"

struct pila
{ int  * elPila;
   int    top;              /* è l'indice dell'elemento in cima alla pila, */
   int num_max ;}; /* top dà anche, aumentato di 1, il numero degli elementi nella pila */

PilaP  costrPila(int numMaxEl)
/*alloca la memoria per una nuova pila  con numMaxEl al più elementi
prec: numMaxel >0
postc: restituisce un puntatore a una nuova pila, NULL se non c'è memoria */
{PilaP p;
assert(numMaxEl >0);
p = (PilaP) malloc(sizeof(struct pila));
assert(p);
p->num_max = numMaxEl;
p->top = -1;
p -> elPila = calloc(1,numMaxEl*sizeof(int));
assert(p->elPila);
return p;}
```

implementazione di una pila su un vettore

```
void distrPila(PilaP p)
/*prec: p != NULL
postc: libera la memoria impegnata dalla pila */
{assert(p);
free(p->elPila);
free(p);}

/*selettore */


int numElPila(PilaP p)
/*prec: p != NULL
postc: restituisce il numero di elementi nella pila */
{assert(p);
return p->top+1;}
```

implementazione di una pila su un vettore

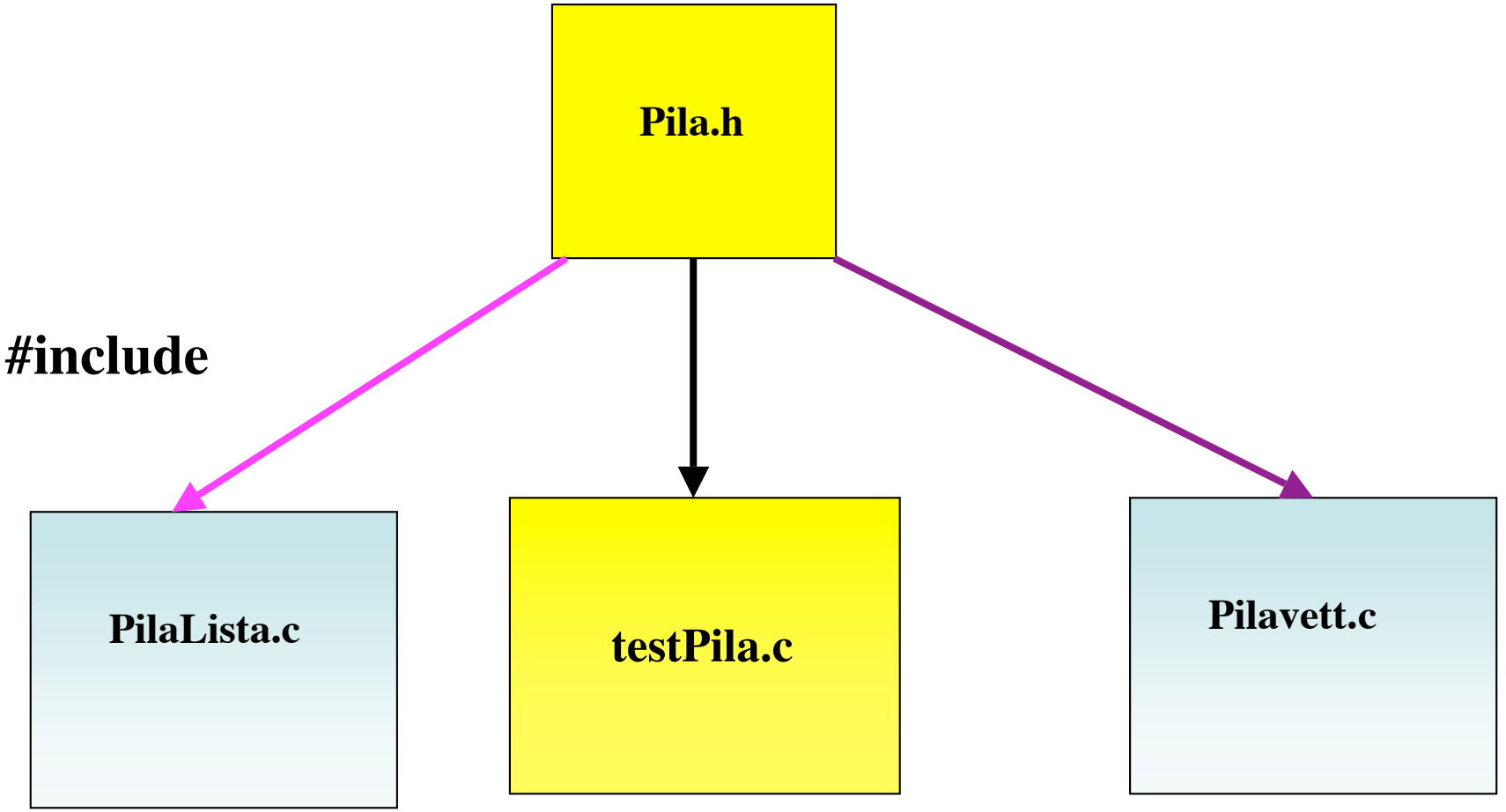
```
int vuota(const PilaP p)
/* da' vero se la PilaP e' vuota, falso altrimenti
*prec: p! NULL;
postc: restituisce un valore != 0 se la pila è vuota, 0 altrimenti*/
{assert(p);
 return p -> top == -1;}

int piena(const PilaP p)
/* da' vero se la PilaP e' piena, falso altrimenti
*prec: p! NULL;
postc: restituisce un valore != 0 se la pila è piena, 0 altrimenti*/
{assert(p);
 return p->num_max == p->top+1;}

void push(int el, PilaP p)
/* inserisce el in cima a p
prec: (p != NULL && !piena(p))
postc: el è in cima alla pila*/
{assert(!piena(p));
p -> top++;
p -> elPila[p -> top] = el;}
```

implementazione di una pila su un vettore

```
int pop(PilaP p)
/* elimina l'elemento in cima a pP, se non e' vuota
*prec: p!= NULL && !vuota(p)
post: restituisce, eliminandolo, l'elemento in cima alla pila*/
{assert(!vuota(p));
return (p-> elPila[p -> top--]);
}

int top(const PilaP p)
/* legge e restituisce l'elemento in cima allo stack, se non e' vuoto
*prec: p != NULL && !vuota(p) */
{assert(!vuota(p));
return  p -> elPila[p -> top];}
```

Breve test della pila

```
#include <stdio.h>
#include <assert.h>
#include <stdlib.h>
#include "pila.h"
#define MAX 100
#define STRINGA "controllo"

int main( )
{char* c = STRINGA;
 PilaP   p;
 p = costrPila(MAX);                    /* costruiamo la pila */
 while (*c != '\0')  /* si inserisce la stringa  nella pila, carattere per carattere */
         {if (!piena(p))
             { push(*c, p);
               printf("il carattere  da inserire e': %c\n",*c);
                printf("il carattere inserito e': %c\n", top(p) );
                c++;}
         else   { printf("la pila è piena!");break;}
         }
 printf("il numero di elementi inseriti è %d \n",numElPila(p));
  while (!vuota(p))   /* si estrae il contenuto dallo stack */
          {putchar(pop(p)); }
 distrPila(p);
 return 0;}
```



Il file del codice “utente” della pila non cambia, cambiando l’implementazione!!

ADT: vantaggi e svantaggi

VANTAGGI:

Programmare pensando in termini di tipi di dato astratto significa che progettiamo le nostre funzioni in modo tale che possano essere usate ignorando i dettagli implementativi, esattamente come accade per i tipi incorporati. Programmare pensando in termini di ADT favorisce quindi la progettazione "orientata agli oggetti" e inoltre di

- **incapsulare dell'informazione**
- **nascondere dell'informazione**

SVANTAGGI

Il codice può risultarne appesantito,
nel caso della Pila per esempio l'accesso agli elementi passa sempre attraverso un puntatore

Nel caso della pila:

Si sono incapsulati tutti gli aspetti implementativi, nascondendoli all'utente dell'ADT.

Incapsulamento è un principio che orienta la progettazione del software verso l'identificazione e, appunto, l' incapsulamento di quelle parti del software che sono rilevanti da un certo punto di vista.

L'incapsulamento facilita il riuso del software, migliora la comprensione, riduce l'impatto delle modifiche e facilita la manutenzione e l'evoluzione.

information hiding è un principio che orienta la progettazione del software verso l'identificazione delle informazioni che si intende nascondere all'utente del programma.

Nel nostro esempio si è secretata completamente l'implementazione.

Nel progettare la suddivisione in moduli di un programma di grandi dimensioni è cruciale stabilire quali informazioni nascondere agli utenti del modulo.