PROGRAMMAZIONE 2
2. Verso OOP & Java

Problel solving & Paradigmi di programmazione

- Consideriamo il problema di determinare l'area di una figura geometrica (ad esempio un rettangolo)
- La costruzione del modello computazionale eseguibile, ovvero il programma, può essere realizzata mediante differenti paradigmi di programmazione (e usando i relativi linguaggi)

Paradigma procedurale

```
#include<stdio.h>
#include<math.h>

main( ){
    double base, height, area;
    printf("Enter the sides of rectangle\n");
    scanf("%lf%lf", &base, &height);
    area = base * height;
    printf("Area of rectangle=%lf\n", area);
    return 0;
}
```

[Codice scritto in C]

Paradigma funzionale

```
let area b h =  b *. h;;
val area : float -> float -> float = <fun>
```

[Codice scritto in OCaml]

Alan J. Perlis [1922 – 1990]

- A good programming language is a conceptual universe for thinking about programming.
- A language that doesn't affect the way you think about programming, is not worth knowing.
- There will always be things we wish to say in our programs that in **all known languages can only be said poorly.**

Passo di astrazione

- Generalizziamo il problema
  - Determinare il valore dell'area di figure geometriche

```
double size( ) {
  double total = 0;
  for (Shape shape: shapes) {
    switch (shape.kind( )) {
      case SQUARE:
        Square square = (Square) shape;
        total += square.width * square.width;
        break;
      case RECTANGLE:
        Rectangle rectangle = (Rectangle) shape;
        total += rectangle.width * rectangle.height;
        break;
      case CIRCLE:
        :
    }
  }
  return total;
}
```

Astrazioni sui dati

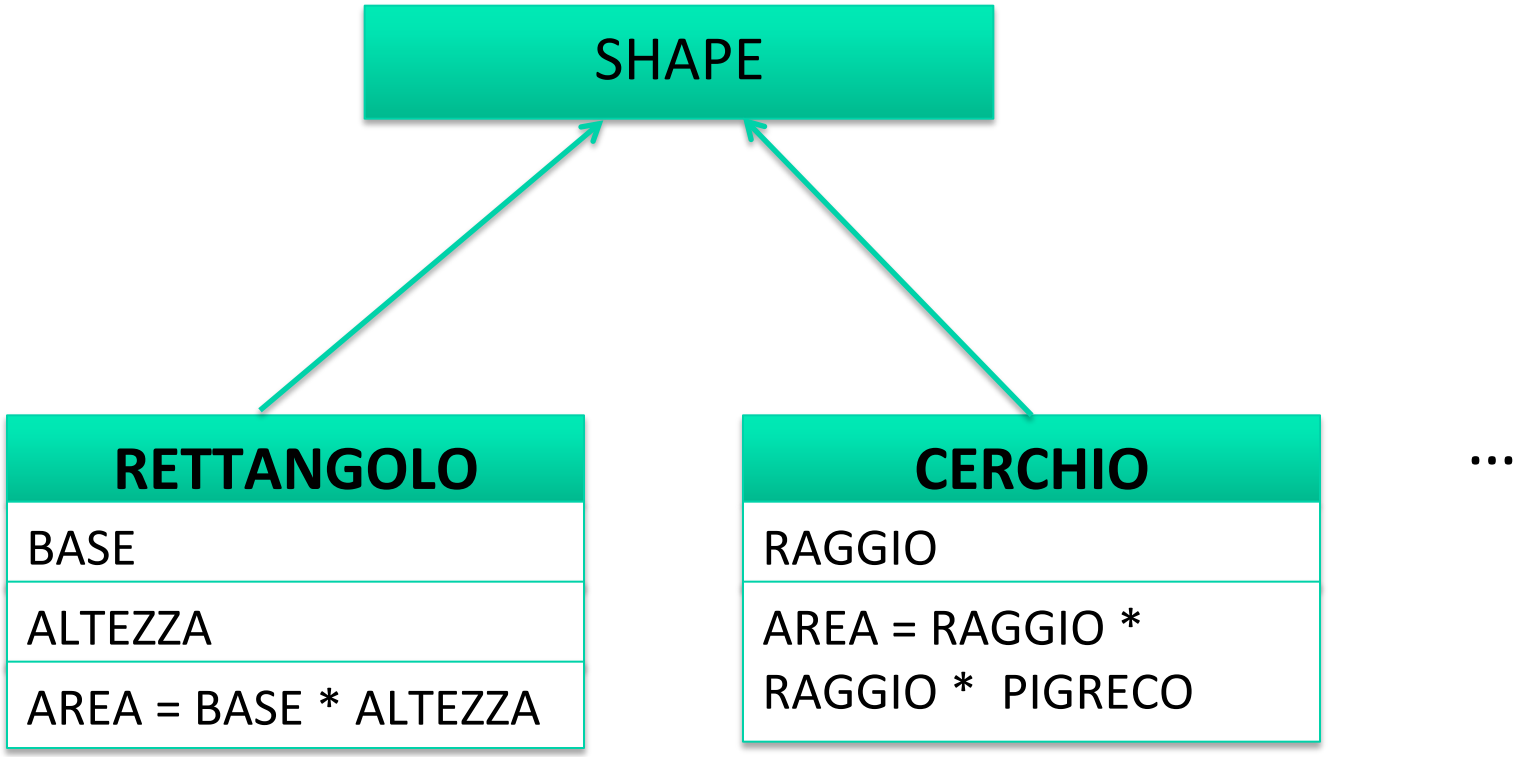
Object-oriented

```
double size( ) {
   double total = 0;
   for (Shape shape: shapes) {
      total += shape.size( );
   }
   return total;
}
```

Distribuiamo la computazione tra le strutture

```
public class Square extends Shape {
   ...
   public double size( ) {
      return width*width;
   }
}
```

Ogni forma geometrica diventa “responsabile” del calcolo dell’area



Oggetto Shape: gestisce del contenuto informativo (dati) e fornisce una funzionalità (il calcolo dell'area)

Object-Oriented Programming

- Object-oriented programming (OOP) is a programming paradigm that represents concepts as “objects” that have data fields (attributes that describe the object) and associated procedures known as methods.

[wiki]

OOP

- Due aspetti importanti
  - Informazione (strutture dati) contenuta nell'oggetto
  - Operazioni significative per operare sulle informazioni presenti nell'oggetto
- Nel mondo della Ingegneria del Software questi due aspetti si riflettono nella nozione di *Information hiding*

Information Hiding

- Information hiding is the principle of segregation of the design decisions in a computer program that are most likely to change, thus protecting other parts of the program from extensive modification if the design decision is changed. The protection involves *providing a stable interface which protects the remainder of the program from the implementation* (the details that are most likely to change)

[wiki]

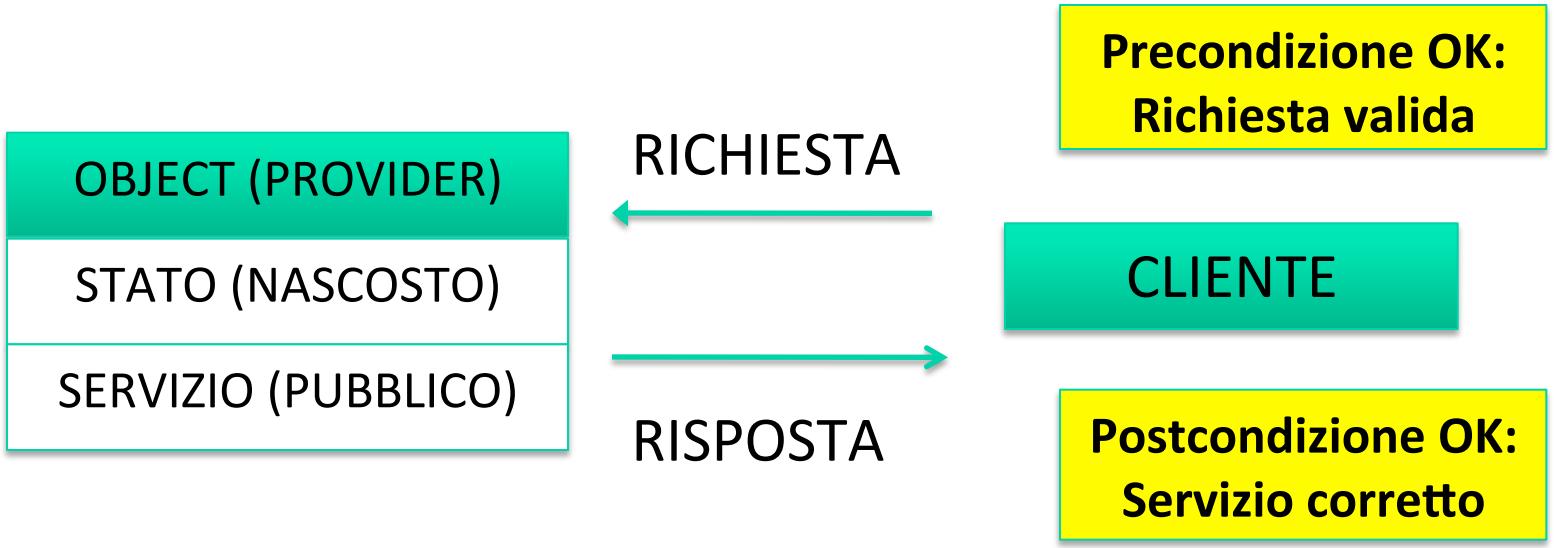
Perché è importante?

- Esposizione della sola informazione che è necessaria a operare
- Dichiarazione di “cosa si fa”, non di “come si fa”
  - *separation of concerns*
- Decomposizione di un sistema in parti
- Protezione dei clienti dalle modifiche riguardanti l’implementazione
- Definizione di contratti di uso
- Facilità per manutenzione e evoluzione del sw

Cosa c'entra con Programmazione II

- La programmazione "by contract" nello spirito OOP usando il linguaggio di programmazione Java come esempio

Programming by contract

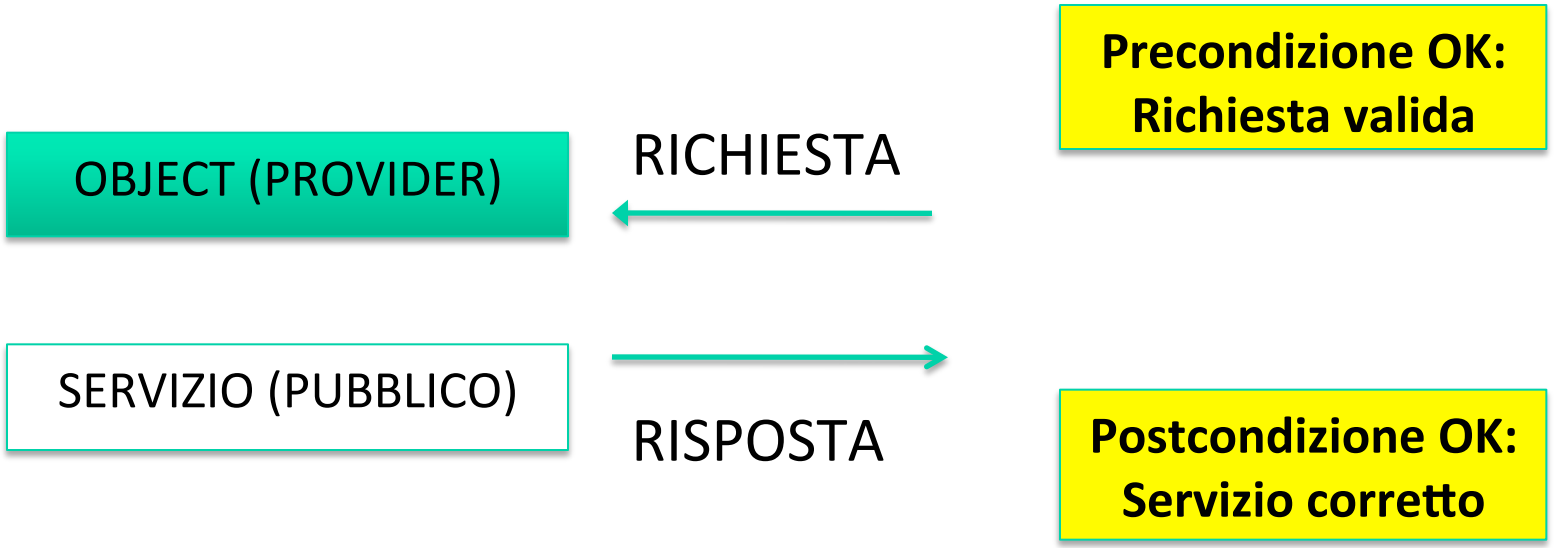


Un contratto definisce il vincolo del servizio

- Visione del cliente: richieste valide
- Visione del provider: fornire correttamente il servizio.

Cosa succede quando viene violato il contratto? **Exception**!!

Programming by contract



Contratto: Invariante di rappresentazione dello stato
Descrive quali sono gli stati validi dell'oggetto
- **alla costruzione**
- **prima e dopo l'invocazione dei metodi (pubblici)**

Cosa succede quando viene violato il contratto? Exception!!!

Java. Perché?

- Modello a oggetti semplice e maneggevole
- Ogni entità è un oggetto
- Ereditarietà singola
- Garbage collection
- Caricamento dinamico delle classi
- Librerie (viviamo in un mondo di API)
- Controllo di tipi statico e dinamico
- Meccanismi per la sicurezza
- Morale: poche novità, ma ragionevolmente pulito, semplice e fruibile

I nostri obiettivi

- Esaminare alcune nozioni fondamentali dei linguaggi di programmazione “moderni”
  - Astrazione sui dati (Data Abstraction)
  - Programmazione “by contract”
  - Generici
  - Modularità
  - Programmazione e verifica
  - Testing, debugging e “defensive programming”

Astrazioni

Meccanismi di astrazione

- L'astrazione sui dati è un esempio significativo di astrazione
- Quali sono i meccanismi di astrazione legati alla programmazione? Lo strumento fondamentale è l'utilizzazione di **linguaggi ad alto livello**
  - enorme semplificazione per il programmatore
  - usando direttamente i costrutti del linguaggio ad alto livello invece che una delle numerosissime sequenze di istruzioni in linguaggio macchina "equivalenti"

Migliori astrazioni nel linguaggio?

- il linguaggio potrebbe avere delle potenti operazioni sull'array del tipo **isIn** e **indexOf**

```
// ricerca indipendente dall'ordine
  found = a.isIn(e);
  if found z = a.indexOf(e);
```

- l'astrazione è scelta dal progettista del linguaggio
  - quali e quanto complicato diventa il linguaggio?
- meglio progettare linguaggi dotati di meccanismi che permettano di definire le astrazioni che servono

Il tipo più comune di astrazione

- ***l'astrazione procedurale*** appare in tutti i linguaggi di program.
  - definizione di procedure, chiamata di procedure
- la separazione tra "definizione" e "chiamata" rende disponibili nel linguaggio i due meccanismi fondamentali di astrazione
- **l'astrazione attraverso parametrizzazione** astrae dall'identità di alcuni dati, rimpiazzandoli con parametri
  - si generalizza un parametro per poterlo usare in situazioni diverse (Shape definito in precedenza)
- **l'astrazione attraverso specifica** astrae dai dettagli implementativi della procedura, per limitarsi a considerare il comportamento che interessa (ciò che fa, non come lo fa)
  - si rende ogni procedura indipendente dalle implementazioni dei moduli che la usano

Astrazione via parametrizzazione

- L'introduzione dei parametri permette di descrivere un insieme (possibilmente infinito) di computazioni diverse con un singolo programma che le astrae tutte
- Il programma seguente descrive una particolare computazione

```
x * x + y * y
```

- La definizione

```
fun(x, y: int) = (x * x + y * y)
```

descrive tutte le computazioni che si possono ottenere chiamando la funzione, cioé applicando la funzione a una opportuna coppia di valori

Astrazione via specifica

- La nozione di procedura si presta a meccanismi di astrazione più potenti della parametrizzazione
  - possiamo astrarre dalla specifica computazione descritta nel corpo della procedura, associando a ogni procedura una specifica (la semantica "intesa" della procedura)
  - …e derivando la semantica della chiamata dalla specifica invece che dal corpo della procedura
  - non è di solito supportata dal linguaggio di programmazione
    - se non in parte tramite le specifiche di tipo, come avviene nei linguaggi funzionali della famiglia di ML

- Si realizza con opportune annotazioni
  - Esempio: Aspect Oriented Programming (AOP)
  - Esempio: JML

```
float sqrt (float coef) {
   // REQUIRES: coef > 0
   // EFFECTS: ritorna una approssimazione
   // della radice quadrata di coef
   float ans = coef/2.0;
   for (int i = 1; i < 7; i++)
         ans = ans-((ans*ans-coef)/
   (2.0*ans));
   return ans;
}
```

- *precondizione* (asserzione requires)
  - deve essere verificata quando si chiama la procedura
- *postcondizione* (asserzione effects)
  - tutto ciò che possiamo assumere che valga quando la chiamata di procedura termina, se al momento della chiamata era verificata la precondizione

Il punto di vista dell'utente

```
float sqrt (float coef) {
   // REQUIRES: coef > 0
   // EFFECTS: ritorna una approssimazione
   // della radice quadrata di coef
   …
}
```

- Gli utenti della procedura non devono preoccuparsi di capire cosa fa, astraendo dalle computazioni descritte nel corpo
  - computazioni che possono essere molto complesse
- Gli utenti della procedura non possono osservare le computazioni descritte dal corpo e da queste dedurre proprietà diverse da quelle specificate nelle asserzioni
  - astraendo dal corpo (implementazione) si "dimentica" informazione evidentemente considerata non rilevante

Tipi di astrazione

- Parametrizzazione e specifica permettono di definire vari tipi di astrazione
  - **astrazione procedurale**
    - si aggiungono nuove operazioni
  - **astrazione di dati**
    - si aggiungono nuovi tipi di dato
  - **iterazione astratta**
    - permette di iterare su elementi di un insieme, senza sapere come questi sono ottenuti
  - **gerarchie di tipo**
    - permette di astrarre da specifici tipi di dato a famiglie di tipi correlati

Astrazione procedurale

- Fornita da tutti i linguaggi ad alto livello
- Aggiunge nuove operazioni a quelle fornite come primitive dal linguaggio di partenza
  - …esempi che avete visto in C e OCaML
- La specifica descrive le proprietà della nuova operazione

Astrazione sui dati

- Fornita da tutti i linguaggi ad alto livello moderni
- Aggiunge nuovi tipi di dato e relative operazioni
  - l'utente non deve interessarsi dell'implementazione, ma fare solo riferimento alle proprietà presenti nella specifica
  - le operazioni sono astrazioni definite da asserzioni logiche (ricordate le specifiche di LPP?)
- La specifica descrive le relazioni fra le operazioni
  - per questo, è diversa da un insieme di astrazioni procedurali

Iterazione astratta

- Permette di iterare su elementi di un insieme, senza sapere come questi sono ottenuti
  - per esempio, potremmo iterare su tutti gli elementi di una matrice senza imporre alcun vincolo sull'ordine con il quale vengono elaborati
- Astrae (nasconde) il flusso di controllo nei cicli

Gerarchie di tipo

- Fornite dai linguaggi ad alto livello moderni
  - per esempio Ocaml, F#, Java, C#...
- I tipi di una famiglia condividono alcune operazioni
  - definite nel supertype, del quale tutti i tipi della famiglia sono subtype
- Una famiglia di tipi astrae i dettagli che rendono diversi tra loro i vari tipi della famiglia

Astrazione e programmazione OO

- Il tipo di astrazione più importante per guidare la decomposizione è l'astrazione sui dati
  - gli iteratori astratti e le gerarchie di tipo sono comunque basati su tipi di dati astratti
- L'astrazione sui dati è il meccanismo fondamentale della programmazione orientata a oggetti
  - anche se esistono altre tecniche per realizzare TD astratti
    - per esempio, all'interno del paradigma di programmazione funzionale

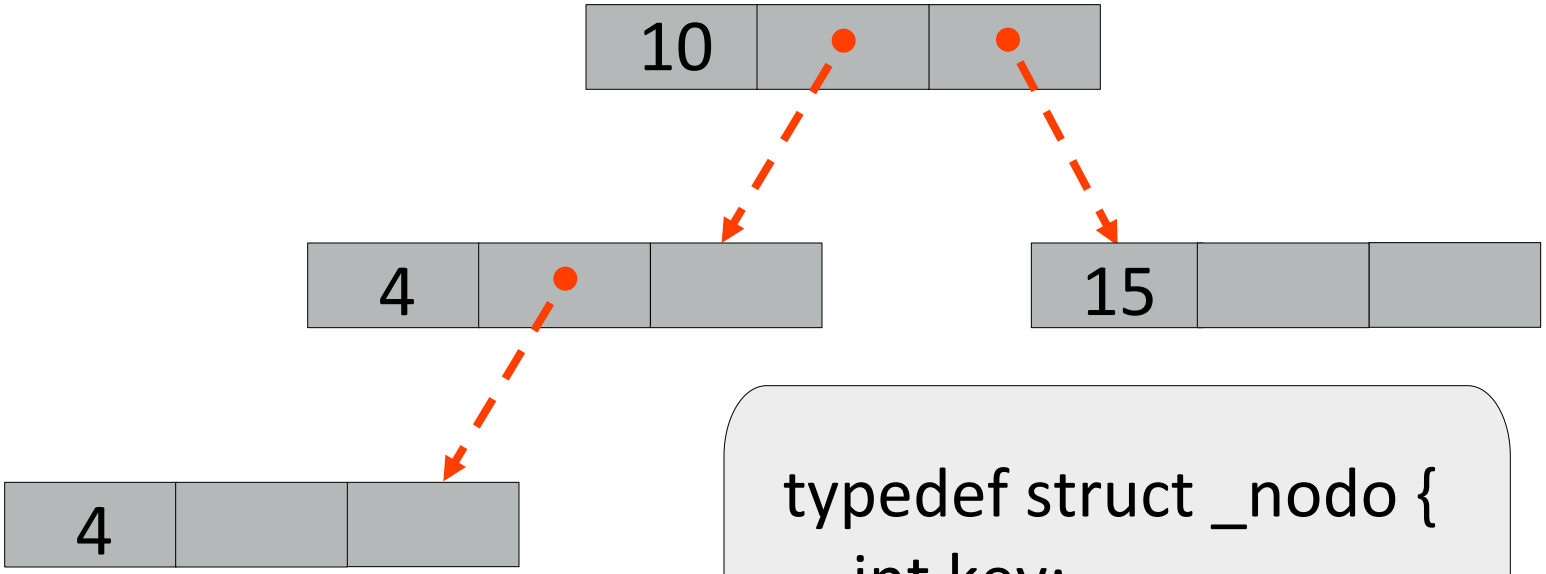
Considerazione finale

- The abstraction principle
  - Each significant piece of functionality in a program should be implemented in just one place in the source code. Where similar functions are carried out by distinct pieces of code, it is generally beneficial to combine them into one by abstracting out the varying parts.

[Benjamin C. Pierce, *Types and Programming Languages* (2002)]

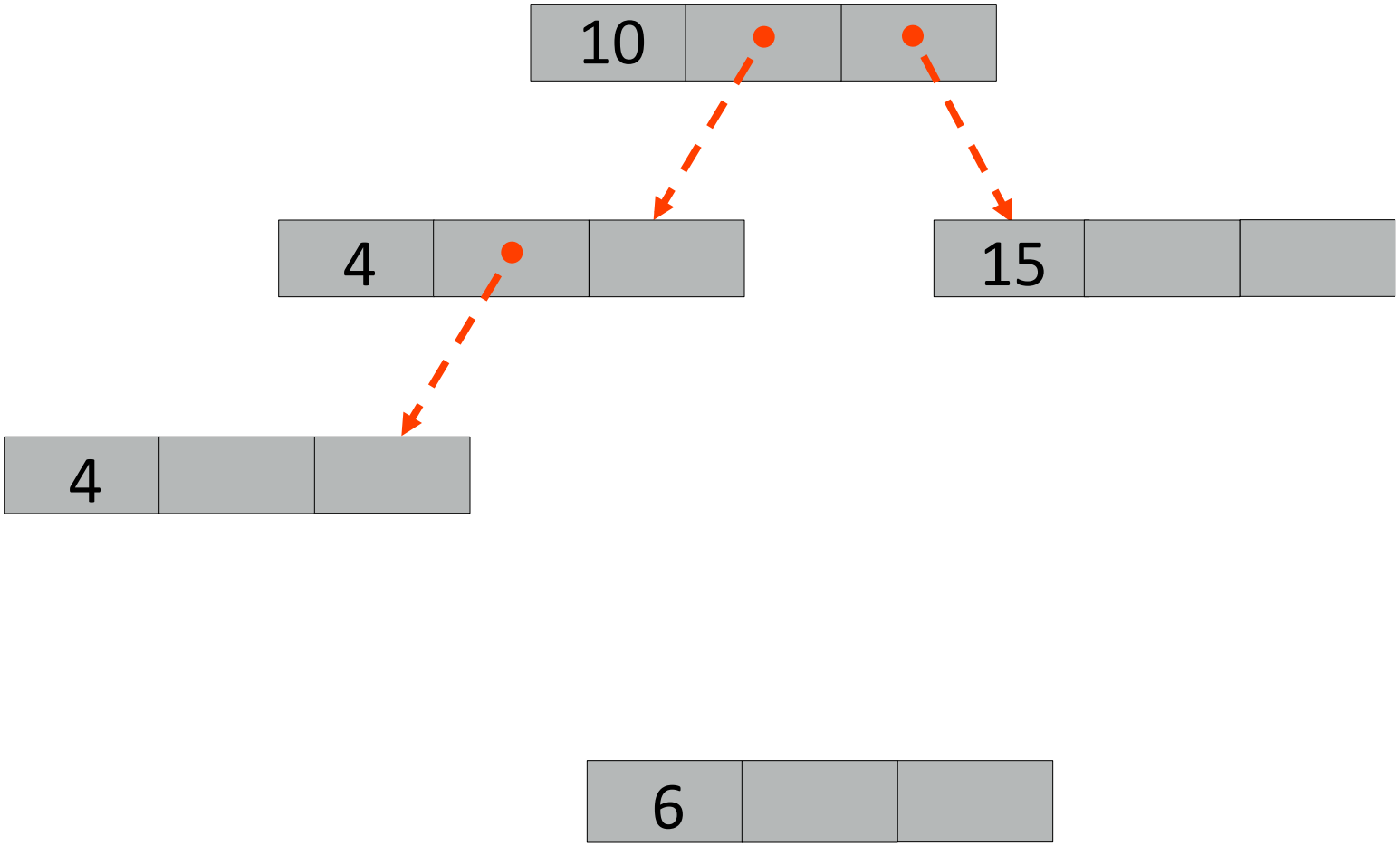
Una implementazione in C degli alberi binari di ricerca

un albero binario di ricerca

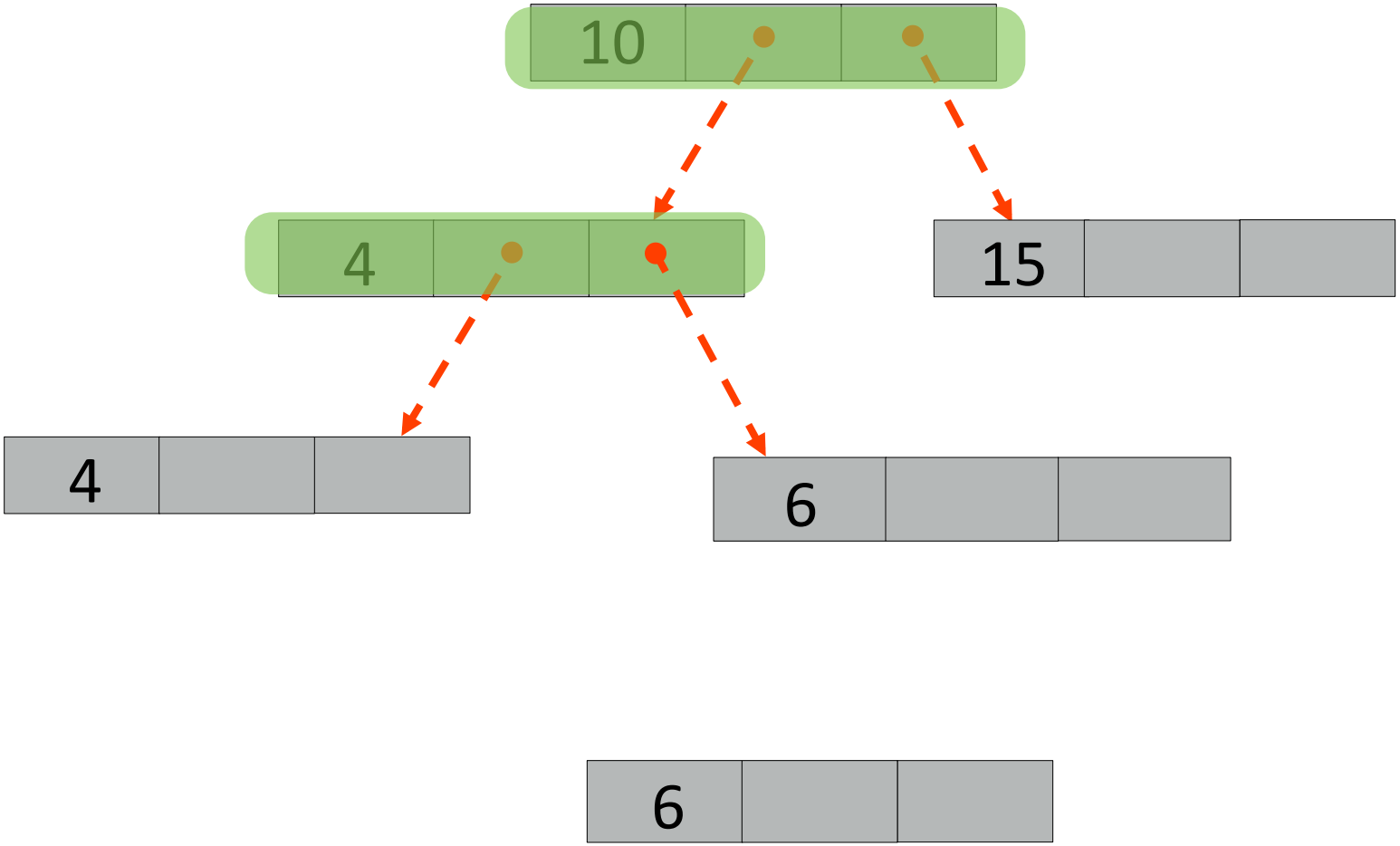



```
typedef struct _nodo {
    int key;
    struct _nodo *left;
    struct _nodo *right;
} nodo;
```

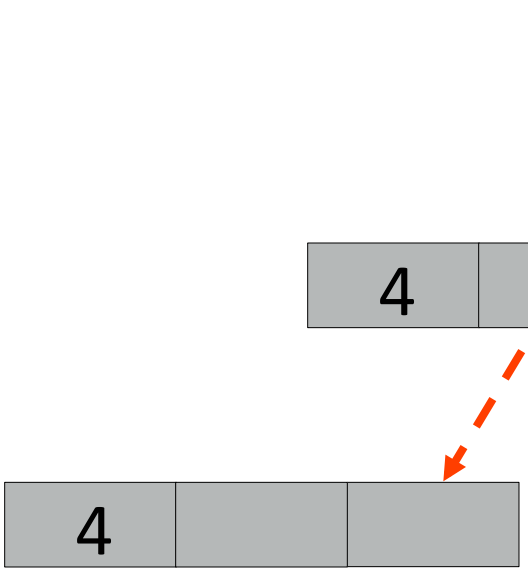
un albero binario di ricerca



un albero binario di ricerca

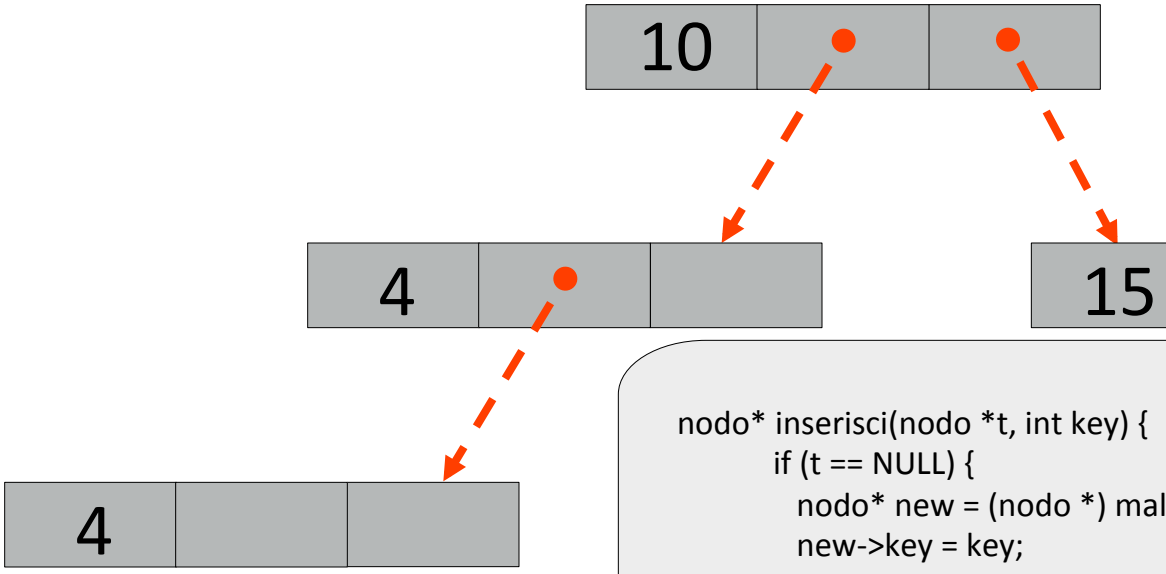


inserimento iterativo



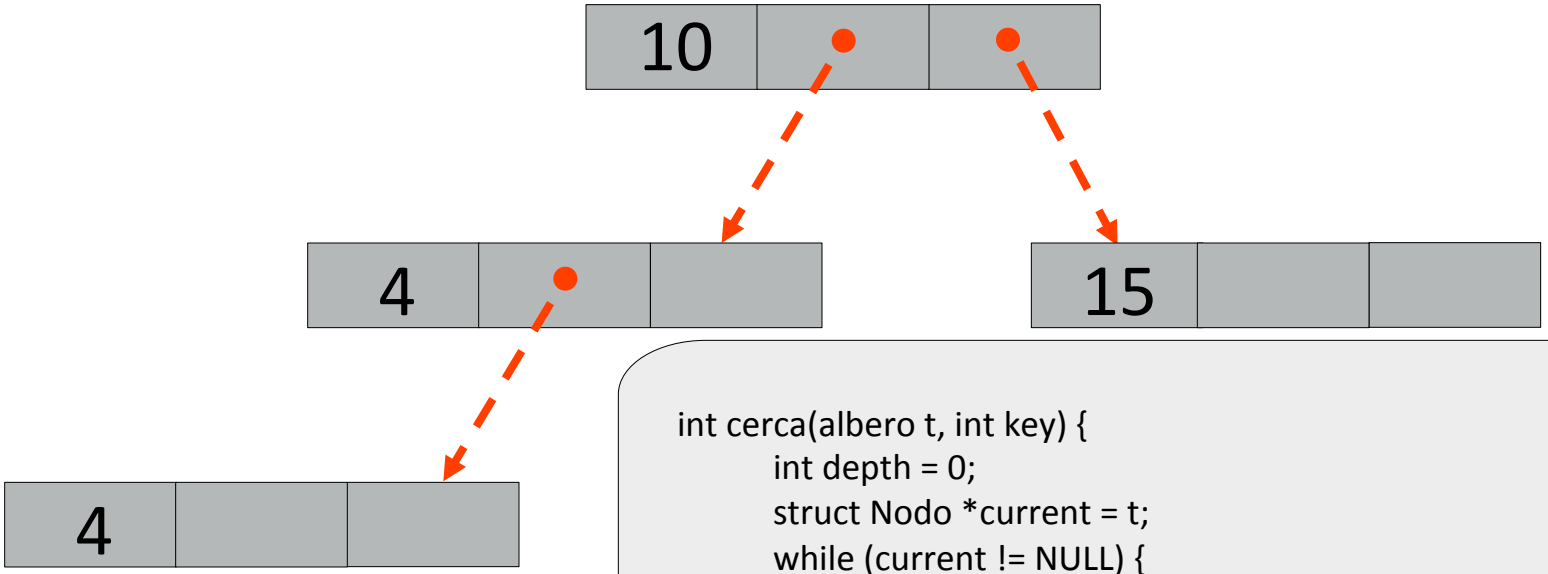

```
nodo* inserisci(nodo *t, int key) {
        nodo* new = (nodo *) malloc(sizeof(nodo));
        new->key = key;
        new->left = new->right = NULL;
        if (t == NULL) return new;

        nodo* parent; nodo* current = t;
        while (current != NULL) {
                parent = current;
                if (current->key < key) current = current->right;
                else current = current->left;
        }
        if (parent->key < key) parent->right = new;
        else parent->left = new;
        return t;
}
```

inserimento ricorsivo



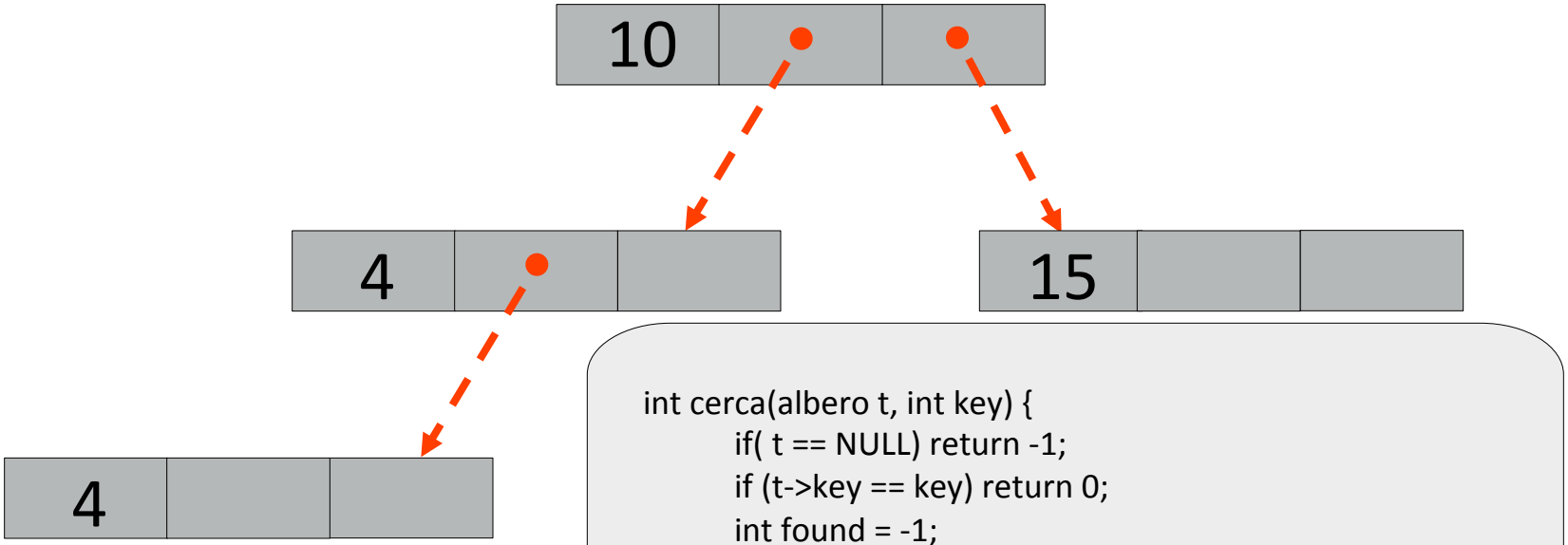

```
nodo* inserisci(nodo *t, int key) {
        if (t == NULL) {
          nodo* new = (nodo *) malloc(sizeof(nodo));
          new->key = key;
          new->left = new->right = NULL;
          return new;
        }

        if (t->key < key)
                        t->right = inserisci(t->right, key);
        else
                        t->left = inserisci(t->left, key);
        return t;
}
```

ricerca iterativa



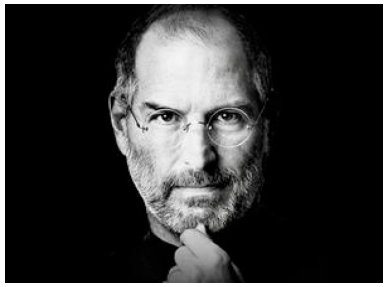

```
int cerca(albero t, int key) {
        int depth = 0;
        struct Nodo *current = t;
        while (current != NULL) {
                if (key == current->key) return depth;
                if (current->key < key)
                        current = current->right;
                else
                        current = current->left;
                depth++;
        }
 return -1;
}
```

ricerca ricorsiva




```
int cerca(albero t, int key) {
        if( t == NULL) return -1;
        if (t->key == key) return 0;
        int found = -1;
        if( t->key < key)
                found = cerca(t->right,key);
        else
                found = cerca(t->left,key);

        if (found >= 0) return 1+found;
        else return -1;
}
```

scenario: ABR modulo condiviso

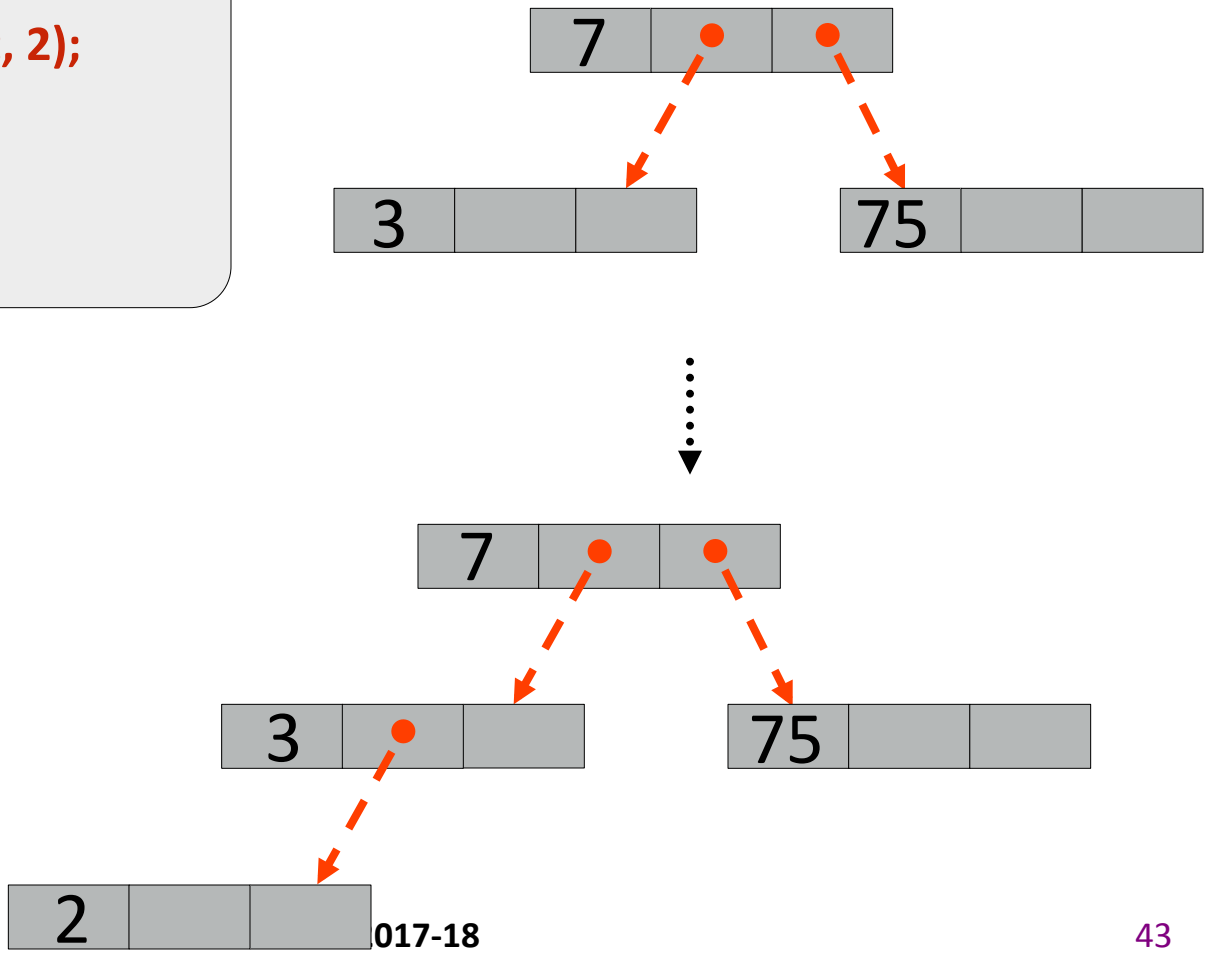


ABR

Programmatori condividono la struttura ABR

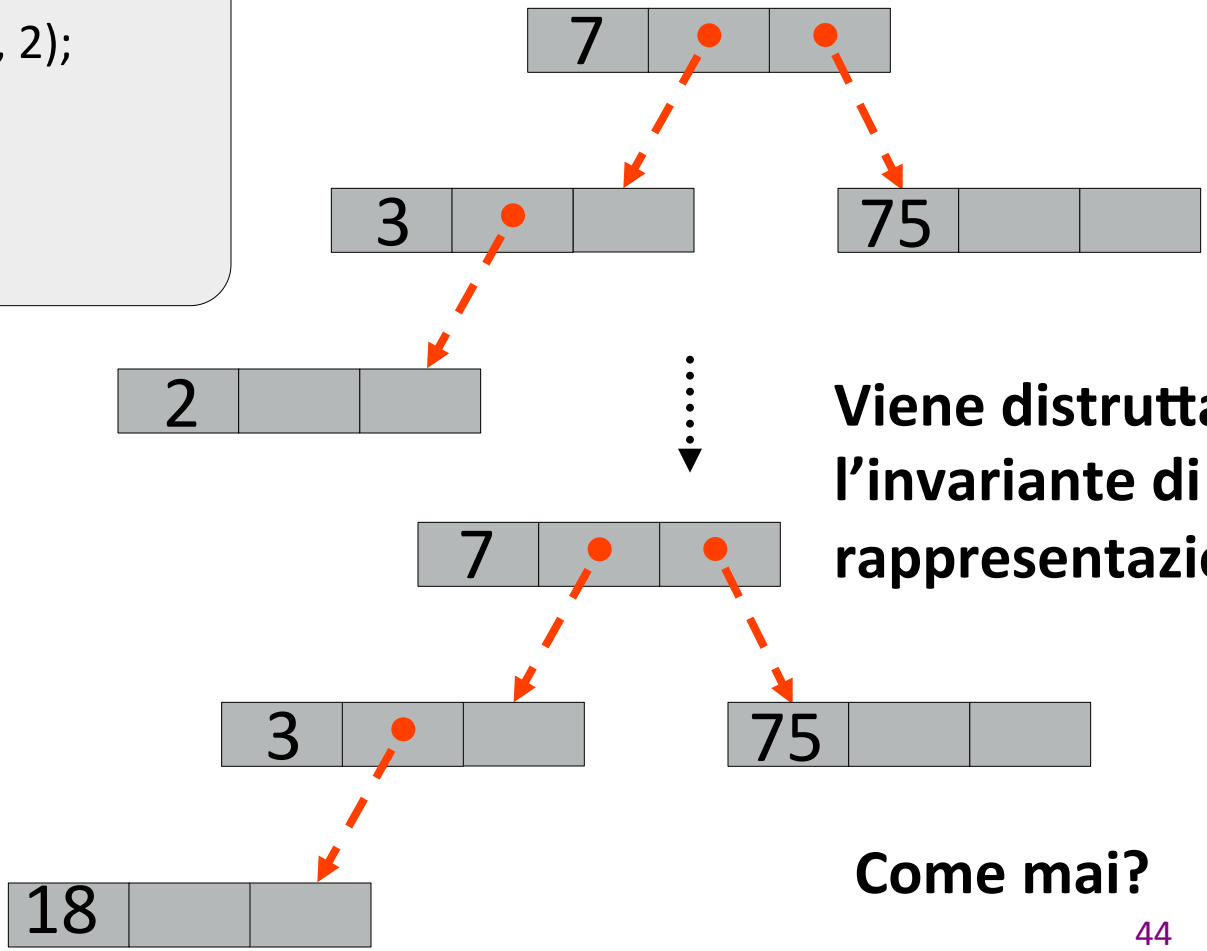
Goofy's code

```
:
G_node = inserisci(t, 2);
:
G_node >key  = 18;
:
```



Goofy's code

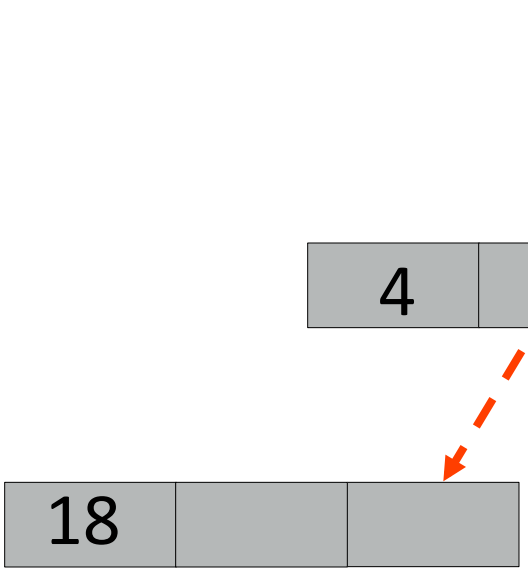
```
:
G_node = inserisci(t, 2);
:
G_node >key  = 18;
:
```



Viene distrutta l'invariante di rappresentazione

Come mai?

invarianti e rappresentazione



RAPPRESENTAZIONE

```
typedef struct _nodo {
    int key;
    struct _nodo *left;
    struct _nodo *right;
} nodo;
```

(nodo ->left)->key < nodo->key &&
nodo->key < (nodo ->right)->key

PUBBLICA: visibile a tutti

scenario 2: ABR e dizionario

- ABR per implementare un dizionario
  - l'estensione richiede di avere una chiave per effettuare la ricerca e una stringa per codificare l'informazione
- Riuso del codice: utilizziamo il vecchio modulo aggiungendo le opportune modifiche per realizzare il dizionario

scenario 2: ABR e dizionario

```
typedef struct _nodo {
    int key;
    string info;
    struct _nodo *left;
    struct _nodo *right;
} nodo;
```

L'invariante è ora una proprietà delle chiavi

ricerca…

```
nodo* inserisci(nodo *t, int key, string info) {
    if (t == NULL) {
      nodo* new = (nodo *) malloc(sizeof(nodo));
      new->key = key;
      new->info = info;
      new->left = new->right = NULL;
      return new;
    }

    if (t->key < key)
            t->right = inserisci(t->right, key, info);
    else
            t->left = inserisci(t->left, key, info);
    return t;
}
```

valutazione della soluzione

- Non abbiamo strumenti linguistici (ovvero previsti nel linguaggio) per estendere il codice alle nuove esigenze
- Cut&Paste Reuse
  - codice debole
  - difficile da mantenere
  - difficile evitare errori

sull'astrazione

- "Abstraction arises from a recognition of similarities between certain objects, situations, or processes in the real world, and the decision to concentrate upon those similarities and to ignore for the time being the differences."

  [Tony Hoare]

- Astrazione: separare le funzionalità offerte dalla loro implementazione

funzionalità vs. implementazione

incapsulamento

- “Encapsulation is the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation.”

[Grady Booch]

sull'ereditarietà

- Passare dal *Cut&Paste reuse* a un metodo supportato da strumenti linguistici nel quale una nuova funzionalità è ottenuta estendendo esplicitamente del codice già implementato

- La nuova implementazione estende la vecchia con funzionalità aggiuntive ma conservando quelle esistenti