

# Corso “Programmazione 1”

## Capitolo 12: Strutture Dati Astratte

Docente: **Marco Roveri** - `marco.roveri@unitn.it`  
Esercitori: **Stefano Berlatto** - `stefano.berlatto-1@unitn.it`  
**Marco Robol** - `marco.robol@unitn.it`  
C.D.L.: Informatica (INF)  
A.A.: 2022-2023  
Luogo: DISI, Università di Trento  
URL: <https://bit.ly/2VgfYwJ>



Ultimo aggiornamento: 16 novembre 2022

# Terms of Use and Copyright

## USE

This material (including video recording) is intended solely for students of the University of Trento registered to the relevant course for the Academic Year 2022-2023.

## SELF-STORAGE

Self-storage is permitted only for the students involved in the relevant courses of the University of Trento and only as long as they are registered students. Upon the completion of the studies or their abandonment, the material has to be deleted from all storage systems of the student.

## COPYRIGHT

The copyright of all the material is held by the authors. Copying, editing, translation, storage, processing or forwarding of content in databases or other electronic media and systems without written consent of the copyright holders is forbidden. The selling of (parts) of this material is forbidden. Presentation of the material to students not involved in the course is forbidden. The unauthorised reproduction or distribution of individual content or the entire material is not permitted and is punishable by law.

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.

## 1 Tipo di Dato Astratto

## 2 Strutture Dati Astratte Importanti

- Le Pile (Realizzate Tramite Array)
- Le Code (Realizzate Tramite Array)
- Le Pile (Realizzate Tramite Liste Concatenate)
- Le Code (Realizzate Tramite Liste Concatenate)
- Gli alberi Binari (Realizzati Tramite Grafi)
- Gli alberi Binari (Realizzati Tramite Array)

## 3 Esempi

- Calcolatrice RPN
- Coda a Priorità
- Rubbrica
- Rubbrica Doppia
- Calcolatrice Standard

# Tipo di Dato Astratto/Abstract Data Type

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.)
  - **stessa interfaccia** (stessi header di funzioni, riportati in un file `.h`)  
⇒ 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:

$\left\{ \begin{array}{l} \text{TDA/point.h} \\ \text{TDA/point.cc} \\ \text{TDA/point\_main.cc} \end{array} \right\}$

# Esempi Molto Importanti di Tipi di Dato Astratto

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

# Le Pile (Stack)

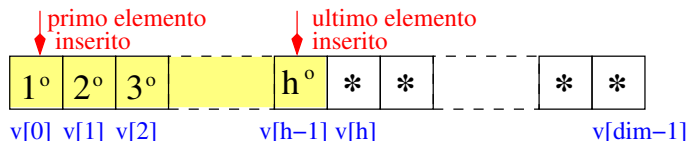
- Una **pila** è una collezione di dati omogenei (e.g., puntatori a struct) in cui gli elementi sono gestiti in modo **LIFO** (**Last In First Out**)
  - Viene visualizzato/estratto l'elemento inserito più recentemente
  - Es: una scatola alta e stretta contenente documenti
- Operazioni tipiche definite 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

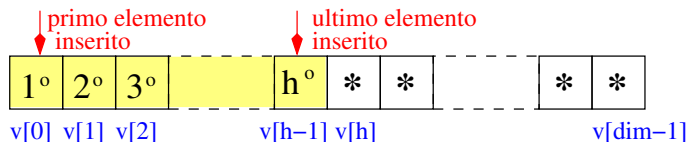


- **Dati:** un intero  $h$  e un array  $v$  di  $dim$  elementi di tipo  $T$ 
  - $v$  allocato staticamente o dinamicamente
  - $h$  indice del prossimo elemento da inserire (inizialmente 0)
- ⇒ numero di elementi contenuti nella pila:  $h$ 
  - pila vuota:  $h==0$
  - pila piena:  $h==dim$
- ⇒ massimo 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:

$\left\{ \begin{array}{l} \text{STACK\_QUEUE\_ARRAY/struct\_stack.h} \\ \text{STACK\_QUEUE\_ARRAY/struct\_stack.cc} \\ \text{STACK\_QUEUE\_ARRAY/struct\_stack\_main.cc} \end{array} \right\}$

- uso di stack per invertire l'ordine:

$\left\{ \begin{array}{l} \text{STACK\_QUEUE\_ARRAY/struct\_stack.h} \\ \text{STACK\_QUEUE\_ARRAY/struct\_stack.cc} \\ \text{STACK\_QUEUE\_ARRAY/struct\_reverse\_main.cc} \end{array} \right\}$

(struct\_stack.h | .cc **stessi** del caso precedente)

# Le Code (Queue)

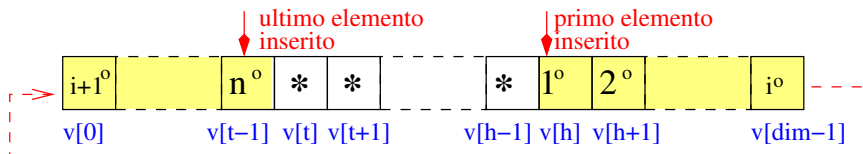
- Una **coda** è una collezione di dati omogenei in cui gli elementi sono gestiti in modo **FIFO** (**First In First Out**)
  - Viene visualizzato/estratto l'elemento inserito meno recentemente
  - Es: una coda ad uno sportello
- Operazioni tipiche definite su una coda di oggetti di tipo **T**:
  - `init()/deinit()`: 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!

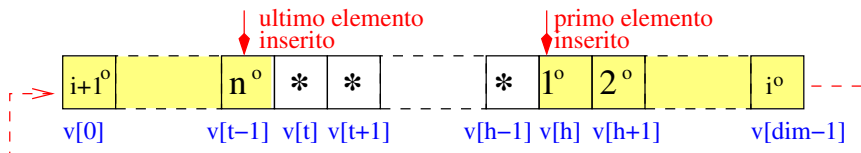
# Implementazione di una coda mediante array



- **Idea:** **buffer circolare:**  $\text{succ}(i) == (i+1) \% \text{dim}$
  - **Dati:** due interi  $h, t$  e un array  $v$  di  $\text{dim}$  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)
- ⇒ num. di elementi contenuti nella coda:  $n = (t \geq h ? t - h : t - h + \text{dim})$
- coda vuota:  $t == h$
  - coda piena:  $\text{succ}(t) == h$
- ⇒ massimo numero di elementi contenuti nella coda:  $\text{dim} - 1$

N.B.:  $\text{dim}$  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"  $t$  ( $t=\text{succ}(t)$ )
- `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:

$\left\{ \begin{array}{l} \text{STACK\_QUEUE\_ARRAY/struct\_queue.h} \\ \text{STACK\_QUEUE\_ARRAY/struct\_queue.cc} \\ \text{STACK\_QUEUE\_ARRAY/struct\_queue\_main.cc} \end{array} \right\}$

# Esercizi proposti

Esercizi proposti!:

{ STACK\_QUEUE\_ARRAY/ESERCIZI\_PROPOSTI.txt }



# Liste Concatenate



- (Nella sua versione più semplice) una **lista concatenata**  $l$  di oggetti di tipo  $T$  è definita come segue:

- $l$  è un puntatore **NULL** (lista vuota) oppure
- $l$  è un puntatore ad un nodo (**struct**) contenente:
  - un campo `value` di tipo  $T$
  - un campo `next` di tipo lista concatenata

```
struct node;
```

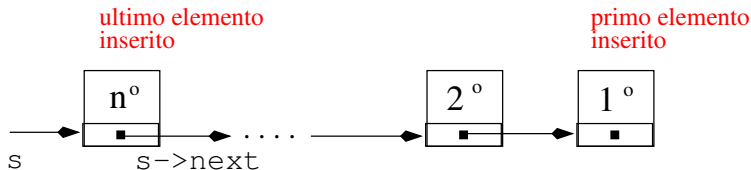
```
typedef node * lista;
```

```
struct node { T value;  lista next; };
```

- Opzionalmente, possono esserci puntatori ad altri elementi

Una lista concatenata è una **struttura dati dinamica**, la cui struttura si evolve con l'immissione e estrazione di elementi.

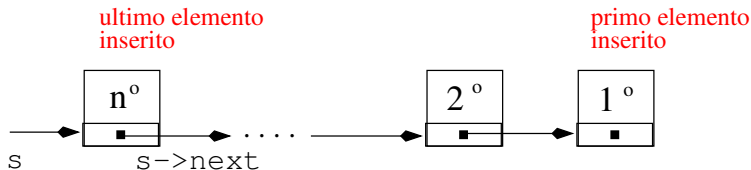
# Implementazione di una pila come lista concatenata



- **Dati:** una lista concatenata  $s$  di  $n$  elementi
    - $s$  punta all'ultimo elemento inserito nella pila (inizialmente NULL)
    - [ Opzionalmente un intero  $n$  con il numero di elementi nella lista ]
    - l'ultimo elemento della lista contiene il primo elemento inserito.
    - pila vuota:  $s == \text{NULL}$
    - pila piena: out of memory
- ⇒ numero di elementi contenuti nella pila limitato dalla memoria

N.B.: allocati solo gli  $n$  nodi necessari a contenere gli elementi

# Implementazione di una pila come lista concatenata II



## ● Funzionalità:

- `init()`: pone  $s = \text{NULL}$
- `push(T)`:
  1. alloca un nuovo nodo ad un puntatore  $\text{tmp}$
  2. copia l'elemento in  $\text{tmp} \rightarrow \text{value}$
  3. assegna  $\text{tmp} \rightarrow \text{next} = s$ , e  $s = \text{tmp}$
- `pop()`:
  1. fa puntare un nuovo puntatore  $\text{first}$  al primo nodo:  $\text{first} = s$
  2.  $s$  aggira il primo nodo:  $s = s \rightarrow \text{next}$
  3. dealloca (l'ex) primo nodo: **delete**  $\text{first}$
- `top(T &)`: restituisce  $s \rightarrow \text{value}$
- `deinit()`: ripete `pop()` finché la pila non è vuota

# Esempi su pile di interi

- semplice stack di interi come struct:

```
{  STACK_QUEUE_PUNT/struct_stack.h  
  STACK_QUEUE_PUNT/struct_stack.cc  
  STACK_QUEUE_PUNT/struct_stack_main.cc }
```

- uso di stack per invertire l'ordine:

```
{  STACK_QUEUE_PUNT/struct_stack.h  
  STACK_QUEUE_PUNT/struct_stack.cc  
  STACK_QUEUE_PUNT/struct_reverse_main.cc }
```

N.B. I “main” e gli header delle funzioni identici a quelli in STACK\_QUEUE\_ARRAY ⇒

Tipo di Dato Astratto

# Implementazione di una coda come lista concatenata



- **Dati:** una lista concatenata  $h$  di  $n$  elementi di tipo  $T$ , un puntatore  $t$  all'ultimo elemento
    - $h$  punta al primo elemento inserito nella coda (inizialmente NULL)
    - $t$  punta all'ultimo elemento inserito nella coda
    - [ Opzionalmente un intero  $n$  con il numero di elementi nella lista ]
    - coda vuota:  $h == \text{NULL}$
    - coda piena: `out of memory`
- ⇒ numero di elementi contenuti nella coda limitato dalla memoria

N.B.: allocati solo gli  $n$  nodi necessari a contenere gli elementi

# Implementazione di una coda come lista conc. II



## ● Funzionalità:

- `init()`: pone `h=NULL`
- `enqueue(T)`:
  1. alloca un nuovo nodo ad un puntatore `tmp`
  2. copia l'elemento in `tmp->value` e pone `tmp->next=NULL`
  3. (se coda non vuota) assegna `t->next=tmp`, e `t=tmp`  
(se coda vuota) assegna `h=tmp`, e `t=tmp`
- `dequeue()`: come `pop()` della pila con il puntatore `h`
- `first(T &)`: come `top()` della pila con il puntatore `h`
- `deinit()`: ripete `dequeue()` finché la coda non è vuota

# Esempi su code di interi

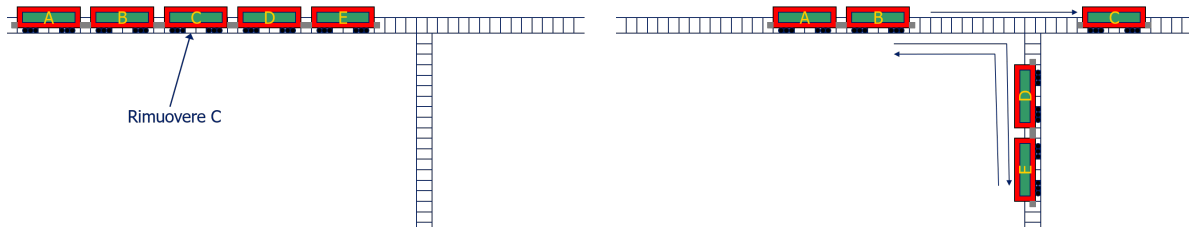
- semplice queue di interi come struct:

$$\left\{ \begin{array}{l} \text{STACK\_QUEUE\_PUNT/struct\_queue.h} \\ \text{STACK\_QUEUE\_PUNT/struct\_queue.cc} \\ \text{STACK\_QUEUE\_PUNT/struct\_queue\_main.cc} \end{array} \right\}$$

N.B. I “main” e gli header delle funzioni identici a quelli in STACK\_QUEUE\_ARRAY  $\Rightarrow$   
Tipo di Dato Astratto

# Esempi uso stack

- Ho un treno composto da vagoni A B C D E, ho un binario di supporto e voglio rimuovere vagone C mantenendo ordine degli altri vagoni.



- Conversione di una espressione da infissa a postfissa
  - infissa :  $((1 + 10) * (20 + 30)) + 40$
  - postfissa:  $1\ 10\ +\ 20\ 30\ +\ *\ 40\ +$



# Esempio stack di **char**

- TDA stack di **char** implementato con array:

$\left\{ \begin{array}{l} \text{STACK\_QUEUE\_PUNT/cstack.h} \\ \text{STACK\_QUEUE\_PUNT/cstack\_arr.cc} \\ \text{STACK\_QUEUE\_PUNT/cstack\_main.cc} \end{array} \right\}$

- TDA stack di **char** implementato con liste concatenate:

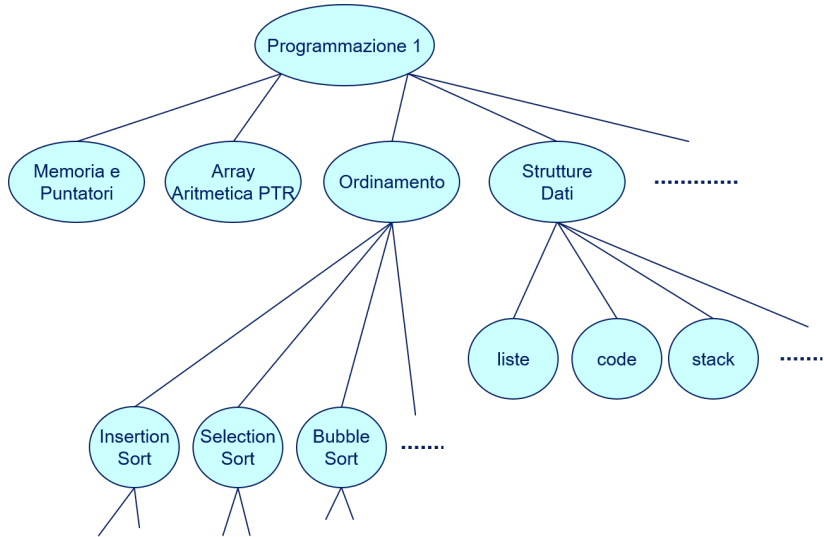
$\left\{ \begin{array}{l} \text{STACK\_QUEUE\_PUNT/cstack.h} \\ \text{STACK\_QUEUE\_PUNT/cstack\_list.cc} \\ \text{STACK\_QUEUE\_PUNT/cstack\_main.cc} \end{array} \right\}$

# Esercizi proposti

Esercizi proposti!:

{ STACK\_QUEUE\_PUNT/ESERCIZI\_PROPOSTI.txt }

- Gli alberi sono una struttura matematica che gioca un ruolo molto importante nella progettazione e nell'analisi di algoritmi:
  - Sono spesso utilizzati per descrivere proprietà dinamiche degli algoritmi.
  - Spesso utilizziamo strutture dati che rappresentano implementazioni concrete di alberi.
- Questo tipo di ADT lo incontriamo nella vita di tutti i giorni:
  - L'albero genealogico della propria famiglia (da cui deriva la maggior parte della terminologia impiegata nella teoria degli alberi).
  - Nei tornei sportivi.
  - Per rappresentare l'organigramma di aziende.
  - Per rappresentare l'analisi sintattica dei linguaggi di programmazione.
  - Il file system di un sistema operativo.
  - Gerarchie
  - ...

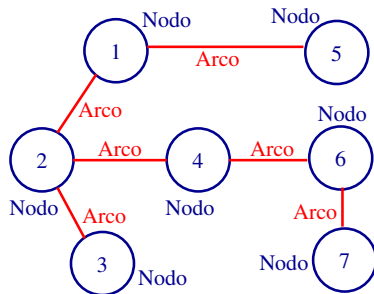


- Esistono diversi tipi di alberi, ed è importante distinguere tra modello astratto e modello concreto (ovvero tra modello matematico e implementazione).
- In ordine di generalità decrescente distinguiamo:
  - Alberi generici.
  - Alberi con radice.
  - Alberi ordinati.
  - Alberi M-ari.
  - **Alberi binari** come caso particolare di albero M-ario.

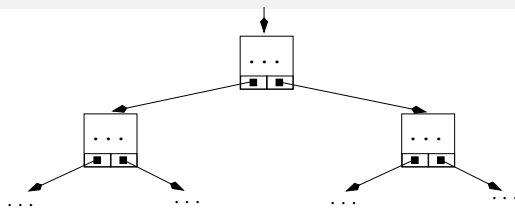
# Alberi (Teoria dei grafi)

In teoria dei grafi un **albero** è un grafo non orientato nel quale due vertici qualsiasi sono connessi da uno e un solo cammino

- Definizione: Un albero è un insieme non vuoto di vertici ed archi (grafo) che soddisfa alcune proprietà:
  - Un vertice (o nodo) è un oggetto semplice che può essere dotato di un nome, e di una informazione associata (denominata spesso chiave o key).
  - Un arco è una connessione tra due nodi.
  - Un grafo **non orientato**, **connesso** e **privo di cicli**



# Alberi binari



- (Nella sua versione più semplice) un **albero binario**  $t$  di oggetti di tipo  $T$  è definito come segue:
  - $t$  è un puntatore **NULL** (albero vuoto) oppure
  - $t$  è un puntatore ad un nodo (**struct**) contenente:
    - un campo `value` di tipo  $T$
    - due campi `left` e `right` di tipo albero

```
struct node;
```

```
typedef node * albero;
```

```
struct node { T value;  albero left, right; };
```

Un albero binario è una **struttura dati dinamica**.

# Alberi binari: terminologia

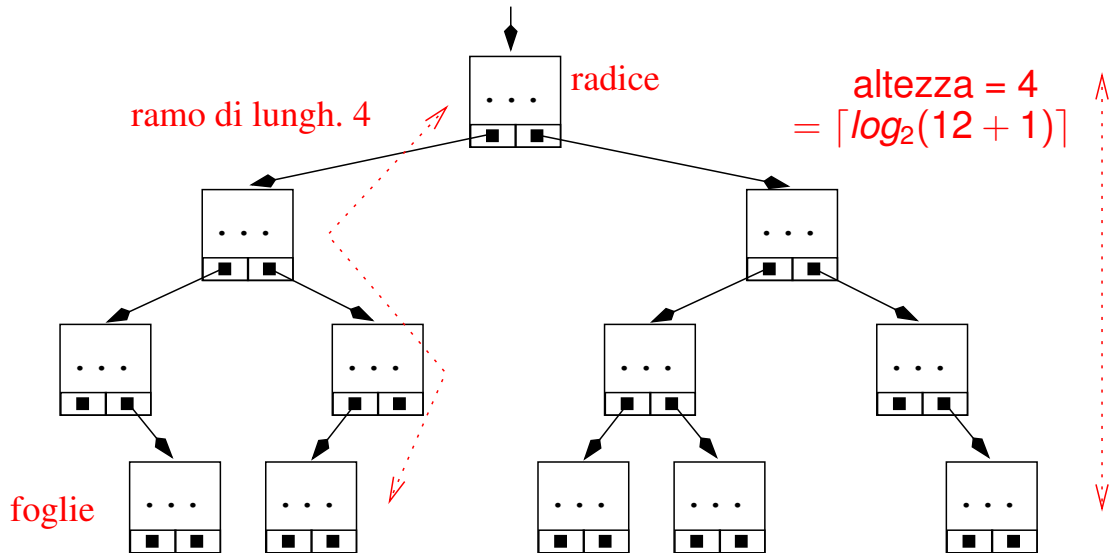
- I sottoalberi (possibilmente vuoti) di un nodo  $N$  sono detti **sottoalbero sinistro** e **sottoalbero destro** di  $N$
- Se un nodo  $N$  punta nell'ordine a due (eventuali) nodi  $N1$ ,  $N2$ 
  - $N1$  e  $N2$  sono detti rispettivamente **figlio sinistro** e **figlio destro** di  $N$
  - $N$  è detto **nodo padre** di  $N1$  e  $N2$
- In un albero binario ci possono essere tre tipi di nodi:
  - Il **nodo radice**, che non ha padre
  - I **nodi foglia**, che non hanno figli
  - I **nodi intermedi**, che hanno padre e almeno un figlio



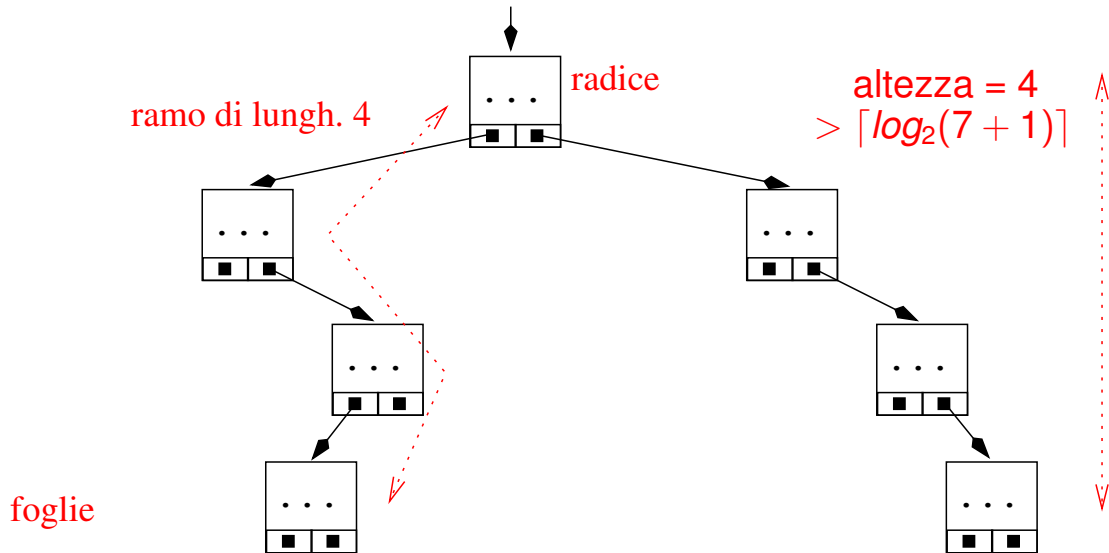
## Alberi binari: terminologia - II

- Una catena di nodi dalla radice a una foglia è detta **ramo**
  - Il numero di nodi in un ramo è detto **lunghezza** del ramo
  - La massima lunghezza di un ramo è detta **altezza** dell'albero
  - L'altezza di un albero binario di  $N$  elementi è  $h \in [\lceil \log_2(N+1) \rceil, N]$
- Un albero binario di  $N$  elementi è **bilanciato** se la sua altezza è  $h = \lceil \log_2(N+1) \rceil$   
 $\implies$  tutti i rami hanno lunghezza  $h$  o  $h-1$
- Un albero binario di  $N$  elementi è **completo** se la sua altezza è tale che  $N = 2^h - 1$   
 $\implies$  tutti i rami hanno lunghezza  $h$

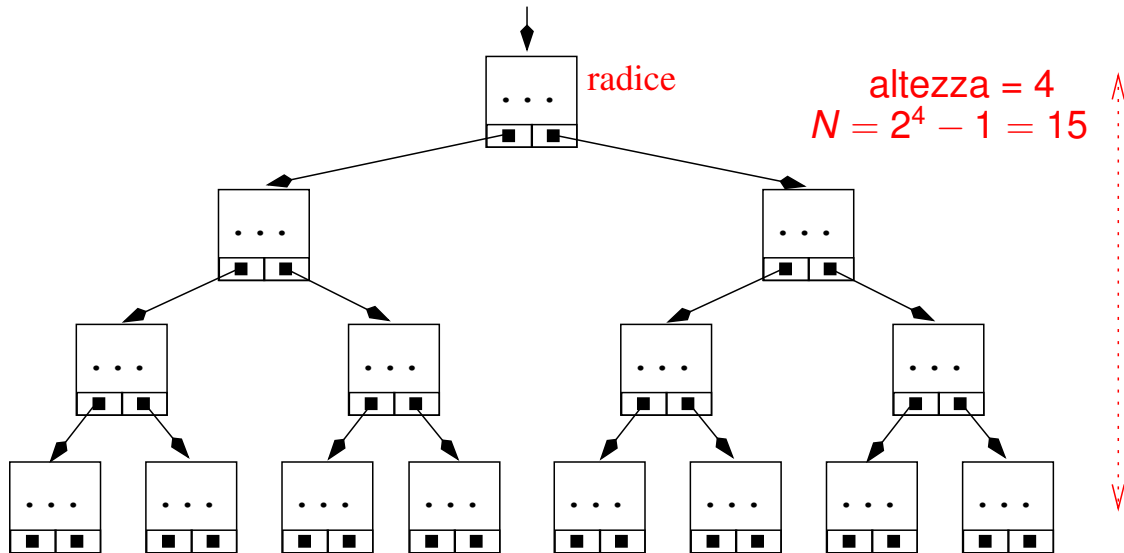
# Esempio: albero binario bilanciato



# Esempio: albero binario non bilanciato



# Esempio: albero binario completo



## Esempio: albero binario completo (2)

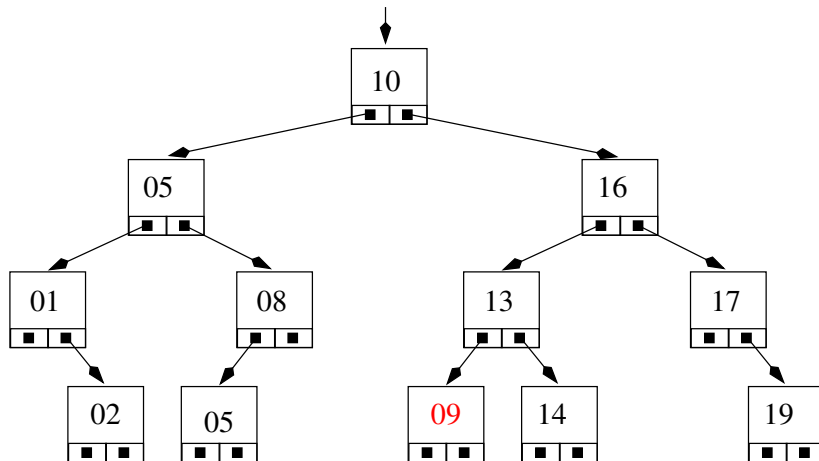


# Albero di ricerca binaria

- Un albero di ricerca binaria è una struttura dati utile a mantenere **dati ordinati**.
- Assumiamo una relazione di ordine totale di precedenza " $\preceq$ " tra gli elementi  $T$ 
  - Es: ordine numerico, ordine alfabetico del campo "cognome", ecc.
- Un albero binario è un **albero di ricerca binaria** se ogni nodo  $N$  dell'albero verifica la seguente proprietà:
  - Tutti i nodi del sottoalbero di sinistra **precedono strettamente**  $N$
  - Tutti i nodi del sottoalbero di destra **sono preceduti** da  $N$(è possibile invertire lo "strettamente" tra sinistra e destra)

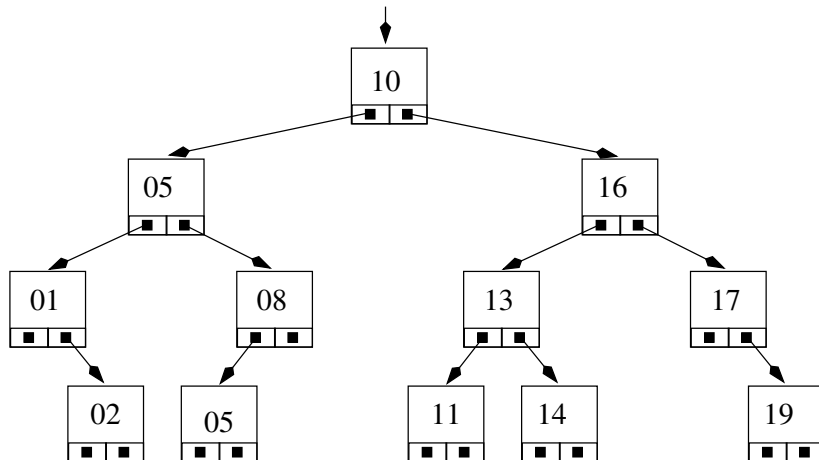
Nota: in alcuni casi non è previsto che ci possano essere due valori uguali nel valore valutato dalla relazione di precedenza (**valore chiave**)

## Esempio: albero di ricerca binaria



- Questo è un albero di ricerca binaria?
- **No**, 09 non può stare nel sottoalbero di destra di 10

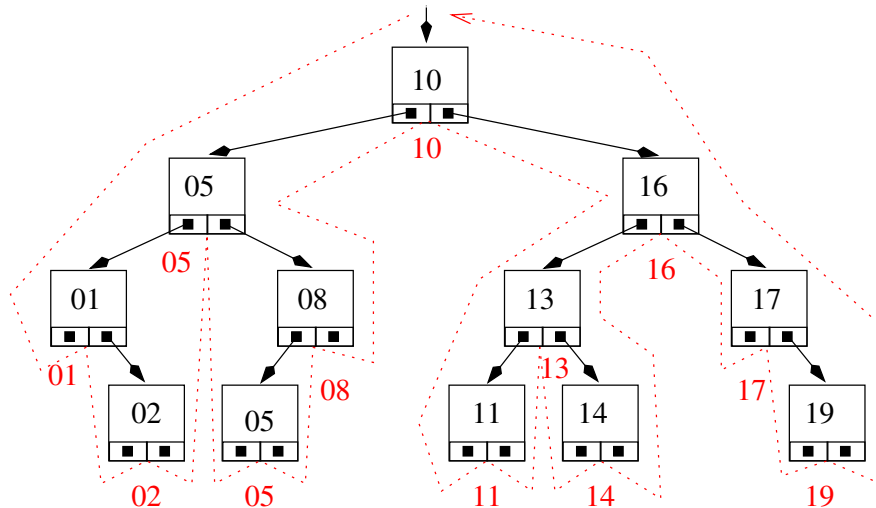
## Esempio: albero di ricerca binaria



- Questo è un albero di ricerca binaria?
- Sì

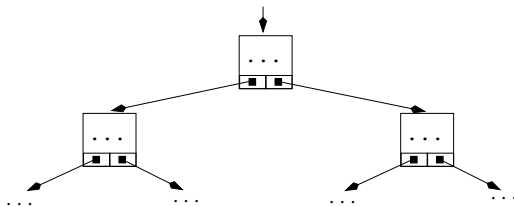


## Esempio: Visita ordinata di un albero di ricerca binaria



- Visita: 01, 02, 05, 05, 08, 10, 11, 13, 14, 16, 17, 19  $\Rightarrow$  ordinati!

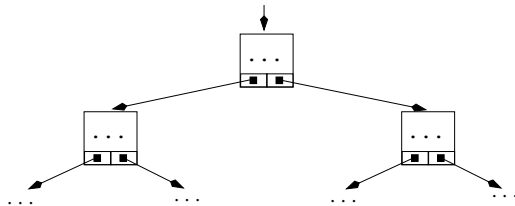
# Implementazione di un albero di ricerca binaria



- **Dati:** un albero di ricerca binaria  $t$ 
    - $t$  punta al primo elemento inserito nell'albero (inizialmente **NULL**)
    - albero vuoto:  $t == \mathbf{NULL}$
    - albero pieno: `out of memory`
- ⇒ numero di elementi contenuti nell'albero limitato solo dalla memoria

N.B.: allocati solo gli  $n$  nodi necessari a contenere gli elementi

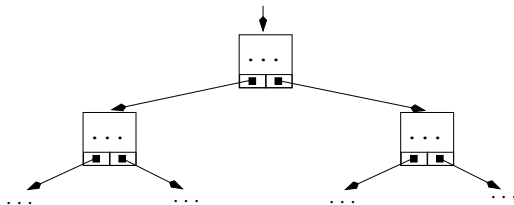
# Implementazione di un albero di ricerca binaria II



## ● Funzionalità:

- `init`: pone `t=NULL`
- `search` (cerca un elemento `val` in `t`):
  1. se `t == NULL`, restituisce `NULL`
  2. se `val == t->value`, restituisce `t`
  3. se `val < t->value`, cerca ricorsivamente in `t->left`
  4. se `val > t->value`, cerca ricorsivamente in `t->right`
- ...

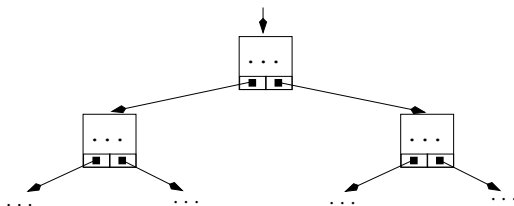
# Implementazione di un albero di ricerca binaria III



## ● Funzionalità:

- ...
- `insert` (inserisce un elemento `val` in `t`):
  1. se `t` è vuoto, `t == NULL`:
    - crea un nuovo nodo per il puntatore `tmp`
    - pone `tmp->value=val`, `tmp->left=NULL`, `tmp->right=NULL`,
    - pone `t=tmp`
  2. se `val < t->value`, inserisci ricorsivamente in `t->left`
  3. se `val >= t->value`, inserisci ricorsivamente in `t->right`
- ...

# Implementazione di un albero di ricerca binaria III



## ● Funzionalità:

- ...
- `print` (stampa in modo ordinato l'albero `t`): Se l'albero non è vuoto
  - stampa ricorsivamente il sottoalbero sinistro `t->left`
  - stampa il contenuto del nodo puntato da `t`: `t->value`
  - stampa ricorsivamente il sottoalbero destro `t->right`
- `deinit`: se l'albero non è vuoto:
  - applica ricorsivamente `deinit` ai sottoalberi sinistro `t->left` e destro `t->right`
  - applica `delete` al nodo puntato da `t`
- [ `remove` non analizzata qui ]

# Esempi su alberi di ricerca binaria su interi

- albero di **char**:

- { TREE/tree.h
  - { TREE/tree.cc
  - { TREE/tree\_main.cc }

- variante della precedente:

- { TREE/tree1.h
  - { TREE/tree1.cc
  - { TREE/tree1\_main.cc }

# Esempi su alberi di ricerca binaria su tipi generici

- albero di "qualsiasi" tipo si voglia:

$\left\{ \begin{array}{l} \text{MODULAR\_TREE/tree.h} \\ \text{MODULAR\_TREE/tree.cc} \\ \text{MODULAR\_TREE/tree\_main.cc} \end{array} \right\}$

# Esercizi proposti

Esercizi proposti!:

{ TREE/ESERCIZI\_PROPOSTI.txt }

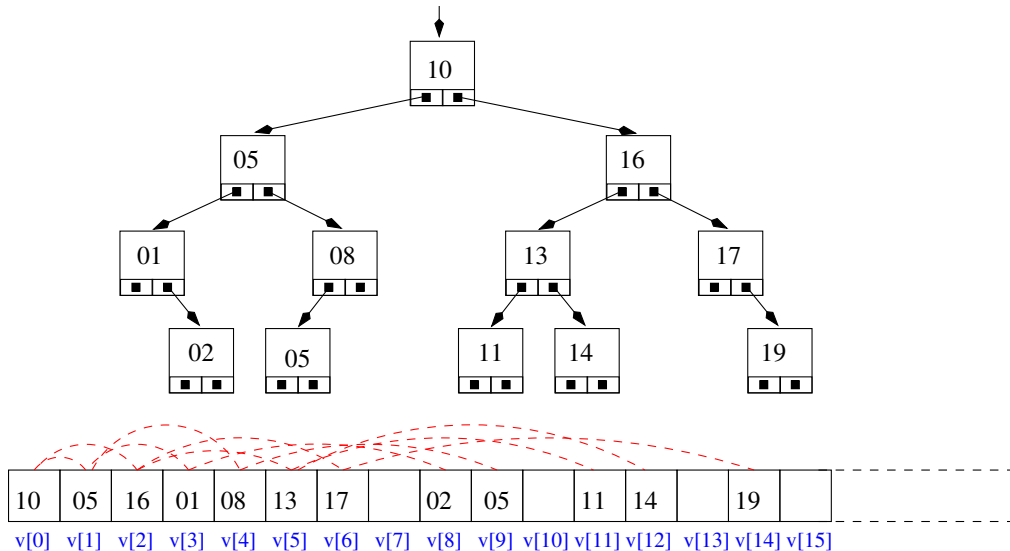


# Implementazione di un albero binario tramite array

- **Dati:** un array  $v$  di  $\dim$  elementi di tipo  $T$ 
  - un (sotto)albero è dato da un puntatore a  $v$  e un indice  $i$   
`struct tree { T * v; int i; };`
  - $v$  allocato dinamicamente
  - l'elemento radice è in  $v[0]$
  - se un elemento è in posizione  $v[i]$ , i suoi due figli sono in posizione  $v[2*i+1]$  e  $v[2*i+2]$
  - necessaria una nozione ausiliaria di “elemento vuoto”
- **Funzionalità:** come nell'implementazione precedente, cambia solo la nozione di figlio sinistro/destro

N.B.: allocati  $\dim$  nodi  $\implies$  efficace solo se ben bilanciato

# Implementazione di un albero binario tramite array II



# Esempi su alberi di interi

- albero di **char**:

$\left\{ \begin{array}{l} \text{TREE\_ARRAY/tree.h} \\ \text{TREE\_ARRAY/tree.cc} \\ \text{TREE\_ARRAY/tree\_main.cc} \end{array} \right\}$

- albero di "qualsiasi" tipo si voglia:

$\left\{ \begin{array}{l} \text{MODULAR\_TREE\_ARRAY/tree.h} \\ \text{MODULAR\_TREE\_ARRAY/tree.cc} \\ \text{MODULAR\_TREE\_ARRAY/tree\_main.cc} \end{array} \right\}$

N.B. Il "main" e gli header delle funzioni identici a quelli in TREE/MODULAR\_TREE  $\Rightarrow$   
**Tipo di Dato Astratto**

# Esempio di uso di Pile: Calcolatrice RPN

- Il metodo di calcolo **Reverse Polish Notation (RPN)** funziona postponendo l'operatore ai due operandi

$34 * 3 \implies 34 \ 3 \ *$

- Permette di effettuare complicate concatenazioni di conti senza usare parentesi:

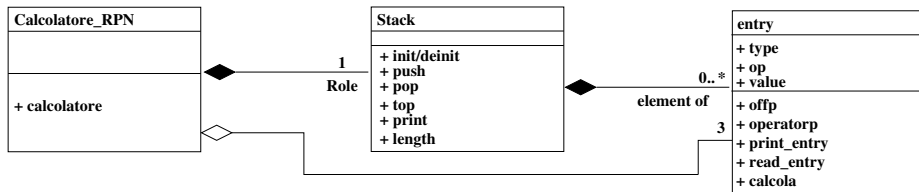
$(34 * 3) / (31 - 5) + (21+3) / (24-12)$

$\implies$

$34 \ 3 \ * \ 31 \ 5 \ - \ / \ 21 \ 3 \ + \ 24 \ 12 \ - \ / \ +$

- Una calcolatrice RPN funziona nel modo seguente:
  - se viene immesso un operando, si inserisce in uno stack di operandi
  - se viene immesso un operatore (binario)  $op$ :
    1. vengono prelevati dallo stack gli ultimi due operandi  $op1$  e  $op2$
    2. viene applicato l'operatore  $op$  a  $op2$  e  $op1$
    3. il risultato viene ri-immesso nello stack
  - l'ultimo elemento nello stack contiene il risultato dell'espressione

# Implementazione della Calcolatrice RPN



Gestione delle entry:

{ CALC\_RPN/entry.h  
CALC\_RPN/entry.cc }

Pila di entry:

{ CALC\_RPN/stack.h  
CALC\_RPN/stack.cc }

Calcolatore RPN:

{ CALC\_RPN/calcolatore\_rpn.h  
CALC\_RPN/calcolatore\_rpn.cc }

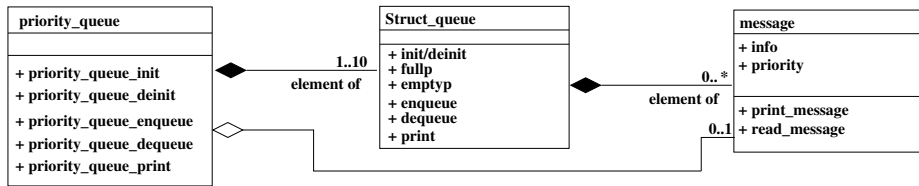
Main:

{ CALC\_RPN/main.cc }

## Esempio di uso di Code: Coda a priorità

- Una **coda a priorità** di messaggi è una struttura dati in cui
  - ogni messaggio arrivato ha una priorità in  $[0..10]$
  - i messaggi vengono estratti in ordine di priorità,  $0 \implies 1 \implies 2 \dots$
  - a parità di priorità vengono estratti in modo FIFO
- Realizzabile con un **array di code**, una per ogni livello di priorità:
  - un messaggio di priorità  $i$  viene inserito nella coda  $i$ -esima
  - l'estrazione avviene a partire dalla coda 0-sima: se vuota si passa alla successiva, ecc.
- Esempio: l'accettazione al Pronto Soccorso di un ospedale

# Implementazione della Coda a Priorità



Gestione delle entità “message”:

```
{ CODA_PRIORITA/message.h }
{ CODA_PRIORITA/message.cc }
```

Coda di (puntatori a) messaggi:

```
{ CODA_PRIORITA/struct_queue.h }
{ CODA_PRIORITA/struct_queue.cc }
```

Coda a priorità di (puntatori a) messaggi:

```
{ CODA_PRIORITA/prio_queue.h }
{ CODA_PRIORITA/prio_queue.cc }
```

Main:

```
{ CODA_PRIORITA/prio_queue_main.cc }
```

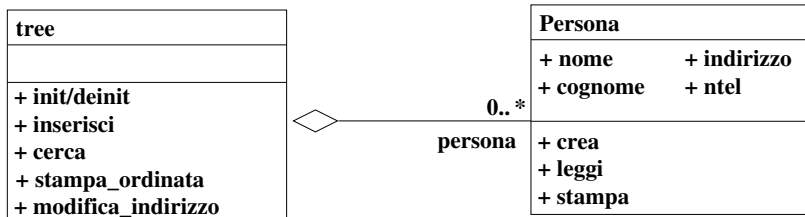




# Esempio di uso di Alberi: una Rubbrica

- Una **Rubbrica** è una lista di (dati di) persone, ordinata con qualche criterio (es per cognome)
  - realizzata come un albero di ricerca binaria di TDA “persona”
  - albero ordinato per il campo “cognome”
  - importante utilizzare **puntatori** a persona:
    - ⇒ ogni copia/passaggio è il passaggio solo di un puntatore
  - per semplicità, non è possibile eliminare una persona dalla rubbrica

# Implementazione della Rubbrica



Gestione del TDA “persona”:

```
{ RUBBRICA/persona.h }
{ RUBBRICA/persona.cc }
```

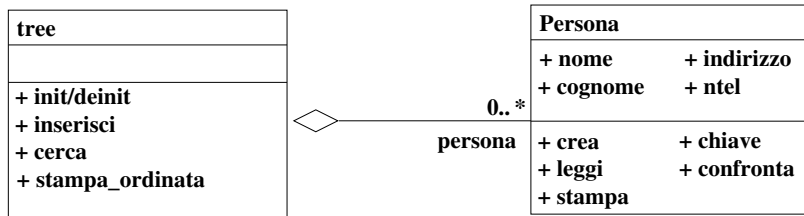
Albero di puntatori a persona, ordinato per cognome:

```
{ RUBBRICA/tree.h }
{ RUBBRICA/tree.cc }
```

Main:

```
{ RUBBRICA/main.cc }
```

# Implementazione della Rubbrica (“modulare”)



Gestione del TDA “persona”:

```
{ MODULAR_RUBBRICA/persona.h }  
{ MODULAR_RUBBRICA/persona.cc }
```

Albero di puntatori a persona, ordinato per cognome:

```
{ MODULAR_RUBBRICA/tree.h }  
{ MODULAR_RUBBRICA/tree.cc }
```

Main:

```
{ MODULAR_RUBBRICA/main.cc }
```

# Esercizi proposti

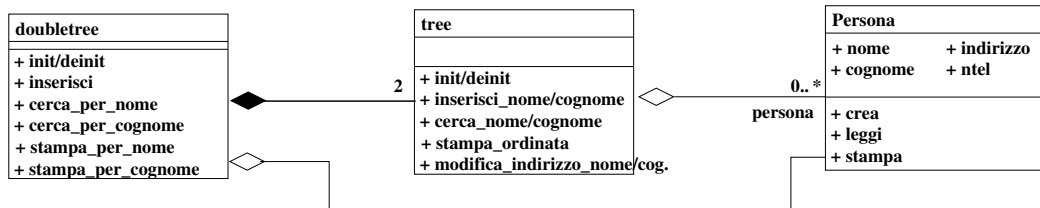
Esercizi proposti!:

{ RUBBRICA/ESERCIZI\_PROPOSTI.txt }

# Esempio di uso di Alberi: una Rubbrica Doppia

- Una **Rubbrica Doppia** è una lista di (dati di) persone, ordinata con un doppio criterio (es per cognome e per nome)
  - realizzata come una coppia di alberi di ricerca binaria di puntatori al TDA “persona”
    - albero ordinato per il campo “cognome”
    - albero ordinato per il campo “nome”
  - importante utilizzare **puntatori** a persona:
    - $\implies$  ogni copia/passaggio è il passaggio solo di un puntatore
    - ogni TDA “persona” è condiviso tra i due alberi
  - per semplicità, non è possibile eliminare una persona dalla rubbrica doppia

# Implementazione della Rubbrica Doppia



Gestione del TDA “persona”:

```
{ RUBBRICA_DOPPIA/persona.h }
{ RUBBRICA_DOPPIA/persona.cc }
```

Albero di puntatori a persona, ordinato per cognome o per nome:

```
{ RUBBRICA_DOPPIA/tree.h }
{ RUBBRICA_DOPPIA/tree.cc }
```

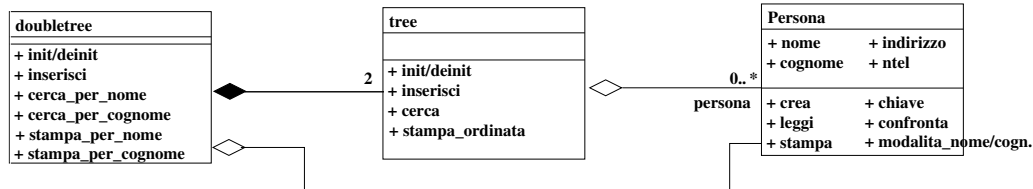
Doppio albero di puntatori a “persona”:

```
{ RUBBRICA_DOPPIA/doubletree.h }
{ RUBBRICA_DOPPIA/doubletree.cc }
```

Main:

```
{ RUBBRICA_DOPPIA/main.cc }
```

# Implementazione della Rubbrica Doppia (“modulare”)



Gestione del TDA “persona”:

```
{ MODULAR_RUBBRICA_DOPPIA/persona.h }
{ MODULAR_RUBBRICA_DOPPIA/persona.cc }
```

Albero di puntatori a persona, ordinato per cognome o per nome:

```
{ MODULAR_RUBBRICA_DOPPIA/tree.h }
{ MODULAR_RUBBRICA_DOPPIA/tree.cc }
```

Doppio albero di puntatori a “persona”:

```
{ RUBBRICA_DOPPIA/doubletree.h }
{ MODULAR_RUBBRICA_DOPPIA/doubletree.cc }
```

Main:

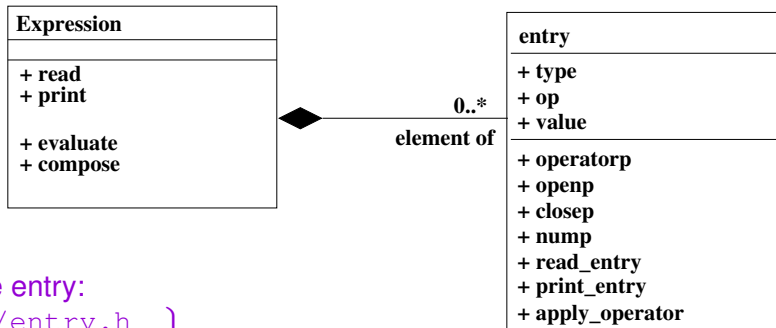
```
{ MODULAR_RUBBRICA_DOPPIA/main.cc }
```

# Esempio di uso di Albero: Calcolatrice Standard

- Un'espressione aritmetica standard utilizza operatori (binari) infissi e parentesi:  
 $((34 * 3) / (31 - 5) + (21+3) / (24-12))$
- Un'espressione aritmetica standard può essere rappresentata da un albero binario
  - le foglie contengano **numeri**
  - gli altri nodi contengano **operatori**
- Un'espressione viene letta e creata ricorsivamente come segue:
  - se viene letto un numero, viene creata un'espressione foglia
  - altrimenti (viene letta una parentesi aperta):
    - viene creato un nodo intermedio
    - viene letta e creata ricorsivamente la prima espressione, e assegnata al figlio sinistro
    - viene letto l'operatore, ed inserito nel nodo
    - viene letta e creata ricorsivamente la seconda espressione, e assegnata al figlio destro
    - viene letta la parentesi chiusa
- Un'espressione viene valutata ricorsivamente come segue:
  - se è una foglia, viene restituito il suo valore
  - altrimenti, si valutano ricorsivamente i due sottoalberi, e vi si applica l'operatore del nodo



# Implementazione della Calcolatrice Standard



Gestione delle entry:

```
{ CALC_STD/entry.h }
{ CALC_STD/entry.cc }
```

TDA espressione aritmetica:

```
{ CALC_STD/expression.h }
{ CALC_STD/expression.cc }
```

Main:

```
{ CALC_STD/calcola_main.cc }
```

# Esercizi proposti

Esercizi proposti!:

{ CALC\_STD/ESERCIZI\_PROPOSTI.txt }