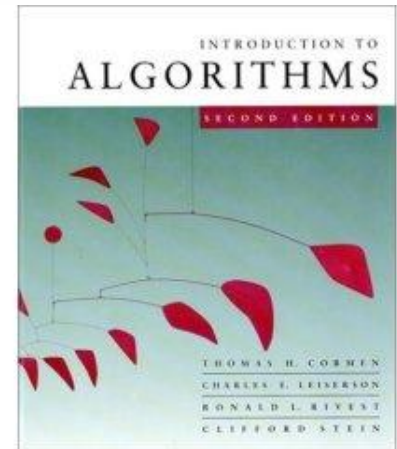




Strutture Dati Elementari

Algoritmi, Strutture Dati e Calcolo Parallelo

- ❑ Questo materiale è tratto dalle trasparenze del corso Introduction to Algorithms (2005-fall-6046) tenuto dal Prof. Leiserson all'MIT (<http://people.csail.mit.edu/cel/>)
- ❑ E dalle trasparenze del corso "Algoritmi e Strutture Dati" del prof. Alberto Montresor dell'Università di Trento
- ❑ T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein Introduction to Algorithms, Second Edition, The MIT Press, Cambridge, Massachusetts London, England McGraw-Hill Book Company
- ❑ Queste trasparenze sono disponibili sui siti <http://www.pierlucalanzi.net>
<http://www.slideshare.net/pierluca.lanzi>



dato = valore che una variabile può assumere

tipo di dato astratto = collezione di valori +
operazioni ammesse su questi valori

incapsula i dettagli dell'implementazione

Esempi: int (+, -, *, /, %), boolean (!, &&, ||)

Dati tipi di dati astratti...

- ❑ Separa la specifica dall'implementazione

- ❑ Specifica
 - ▶ "manuale d'uso"
 - ▶ nasconde i dettagli implementativi all'utilizzatore

- ❑ Implementazione
 - ▶ realizzazione vera e propria

- ❑ I dati sono spesso organizzati in insiemi detti strutture dati

- ❑ **Strutture dati**
 - ▶ Focus sull'organizzazione dei dati, non sul tipo di dato
 - ▶ Il tipo dei dati contenuti può essere parametrico
 - ▶ Strutture dati astratte, quando si vuole distinguere le strutture stesse dalle implementazioni

- ❑ **Una struttura dati è composta da**
 - ▶ un modo sistematico di organizzare i dati
 - ▶ un insieme di operatori per manipolare la struttura

- ❑ **Tipologie di strutture dati**
 - ▶ lineari/non lineari
 - ▶ statiche/dinamiche (variazione di dimensione, contenuto)
 - ▶ omogenee/disomogenee (dati contenuti)

- ❑ Struttura dati "generale": insieme dinamico
 - ▶ Può crescere, contrarsi, cambiare contenuto
 - ▶ Operazioni base: inserimento, cancellazione, ricerca
 - ▶ Il tipo di insieme (= struttura) dipende dalle operazioni

- ❑ Elemento
 - ▶ Uno degli oggetti memorizzato nella struttura
 - ▶ Campo chiave di identificazione
 - ▶ Dati satellite (aggiuntivi oltre alla chiave)
 - ▶ Campi che fanno riferimento ad altri elementi dell'insieme

❑ Operazioni di interrogazione

- ▶ Item search(Key k)
- ▶ Item successor(Item x)
- ▶ Item predecessor(Item x)
- ▶ Item minimum()
- ▶ Item maximum()

❑ Operazioni di modifica

- ▶ void insert(Item x)
- ▶ void delete(Item x)

Liste...

Una lista è una sequenza ordinata di elementi

Gli elementi della lista hanno una posizione

$$\langle a_0, a_1, \dots, a_{n-1} \rangle$$

È definito il concetto di posizione corrente (fence)

$$\langle a_0, a_1, \dots, a_k, \mid a_{k+1}, \dots, a_{n-1} \rangle$$

Quali operazioni dobbiamo implementare?

Inserimento (testa, posizione corrente, coda),
cancellazione, posizionamento (setpos),
prev, next, ecc.

Lista (e puntatore corrente)

$\langle a_0, a_1, \dots, a_k, \mid a_{k+1}, \dots, a_{n-1} \rangle$

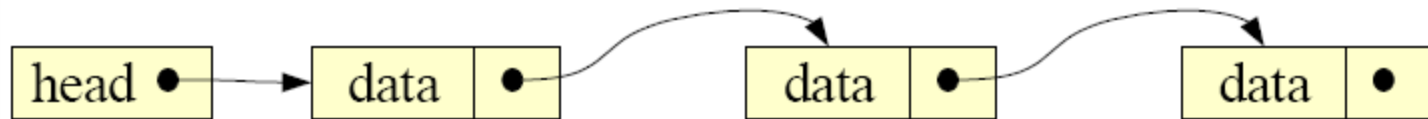
Quale implementazione?

Array?

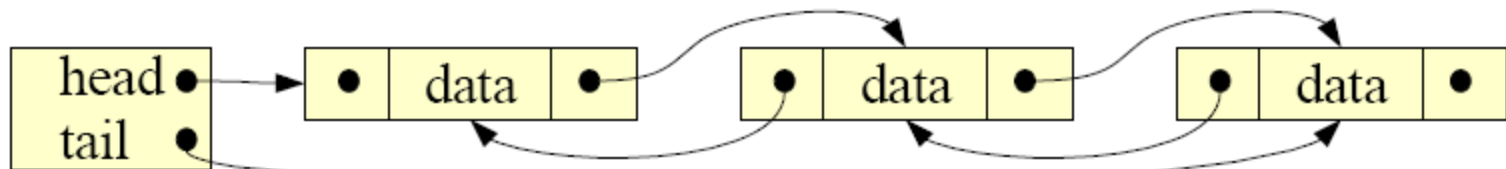
Liste puntate?

- ❑ Sequenza di nodi, contenenti dati (qualsiasi) arbitrari e 1-2 reference (puntatori, link) all'elemento successivo e/o precedente

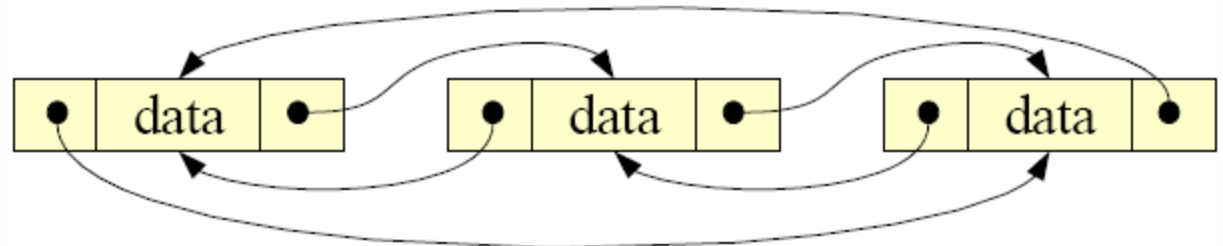
- ❑ Liste puntate semplici

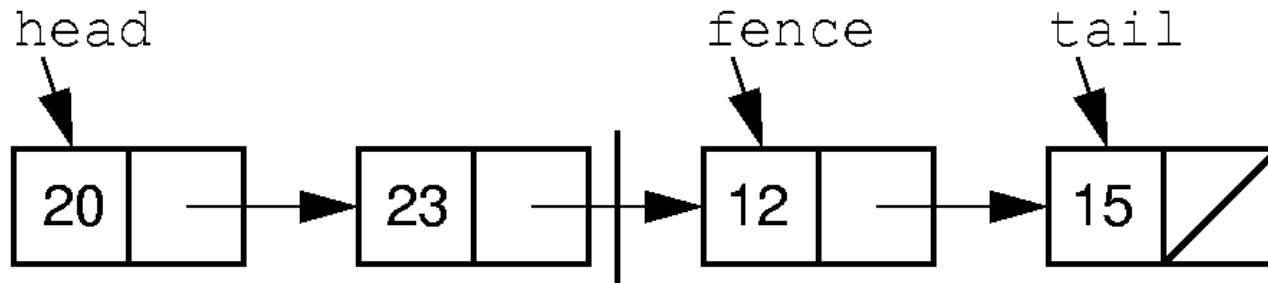


- ❑ Liste puntate doppie

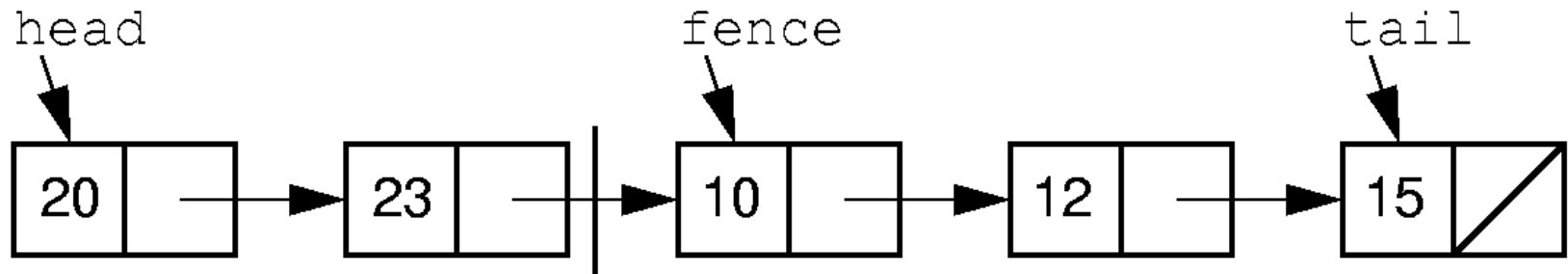


- ❑ Liste circolari





(a)



(b)

Ricerca

LIST-SEARCH(L, k)

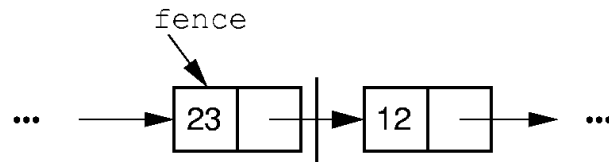
```
1 x ← head[L]
2 while x ≠ NIL
  and key[x] ≠ k
3   do x ← next[x]
4 return x
```

Inserimento in testa

LIST-INSERT(L, x)

```
1 next[x] ← head[L]
2 if head[L] ≠ NIL
3   then prev[head[L]] ← x
4 head[L] ← x
5 prev[x] ← NIL
```

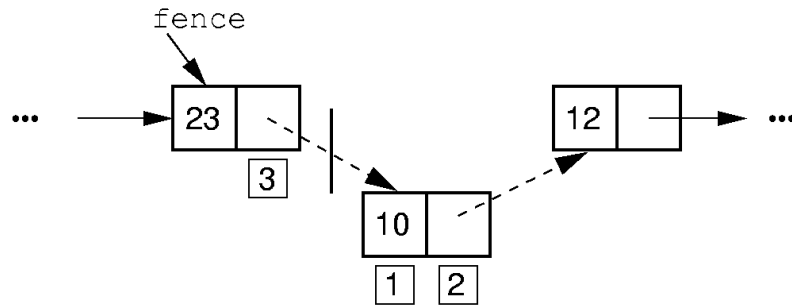
Inserimento



Insert 10:

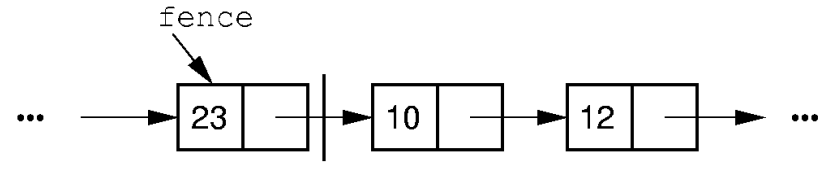
10	
----	--

(a)

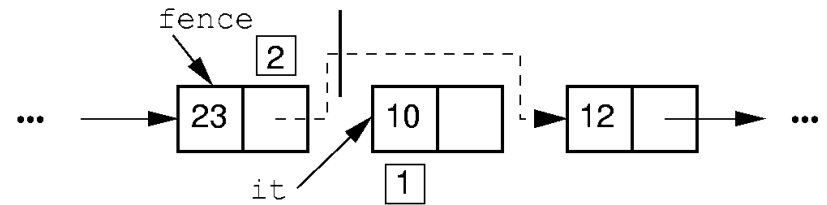


(b)

Cancellazione



(a)

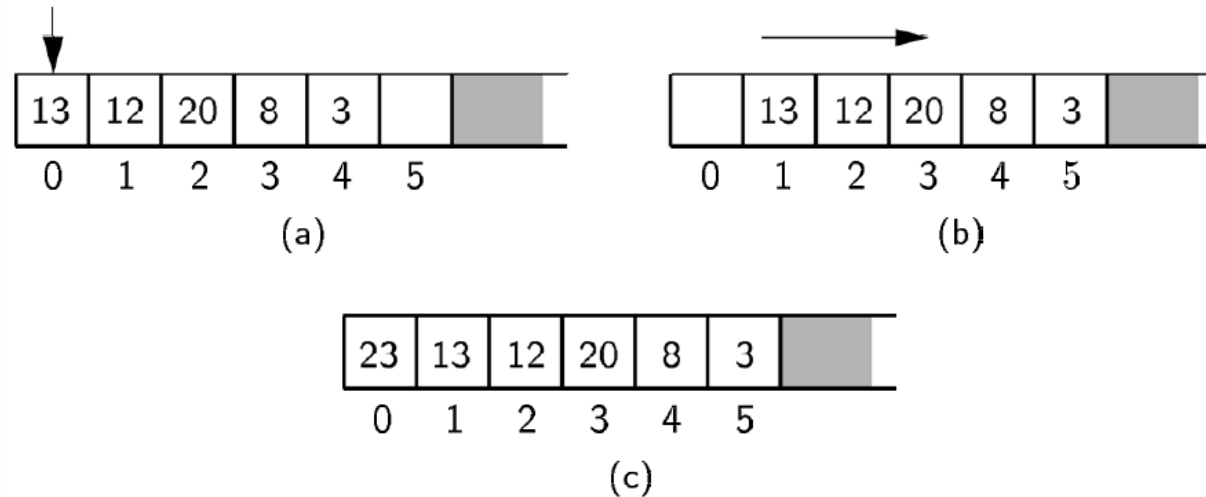


(b)

- ❑ Qual è la complessità della ricerca?
- ❑ Qual è la complessità dell'operazione **prev** che sposta il puntatore corrente (fence) alla posizione precedente?
- ❑ Qual è la complessità dell'operazione **setpos** che sposta il puntatore corrente (fence) a una posizione specificata?

- ❑ Una lista può essere implementata utilizzando una lista puntata, però potremmo anche decidere di usare gli array
- ❑ Ad esempio, l'inserimento in testa,

Insert 23:



- ❑ Quanto costa? Quanto costava con le liste puntate?

- ❑ Implementazione delle liste
 - ▶ Strutture dati dinamiche (linked list)
 - ▶ Array

- ❑ Implementazione basata su array
 - ▶ L'array potrebbe dover essere allocato tutto insieme
 - ▶ No overhead se tutte le posizioni sono piene
 - ▶ Inserimento e cancellazione, complessità?
 - ▶ Prev e setpos, complessità?

- ❑ Implementazione basata su linked lists:
 - ▶ Lo spazio richiesto cresce con il numero di elemento
 - ▶ Ogni elemento richiede overhead
 - ▶ Inserimento e cancellazione, complessità?
 - ▶ Prev e setpos, complessità?

confronto sull'occupazione di memoria

"Break-even" point: $DE = n(P + E)$;

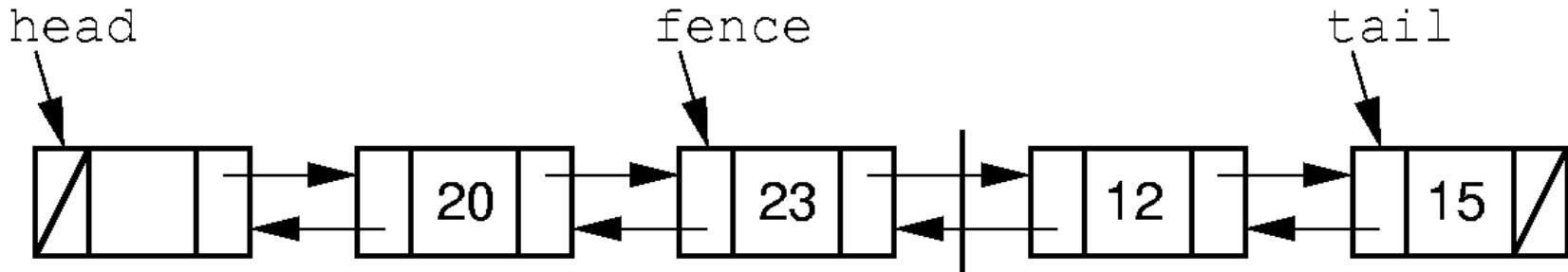
$$n = DE / (P + E)$$

E: Spazio per un elemento

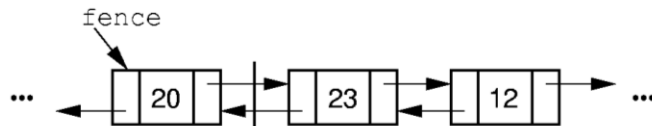
P: Spazio per un puntatore

D: Numero di elementi nell'array

Liste Doppie Puntate



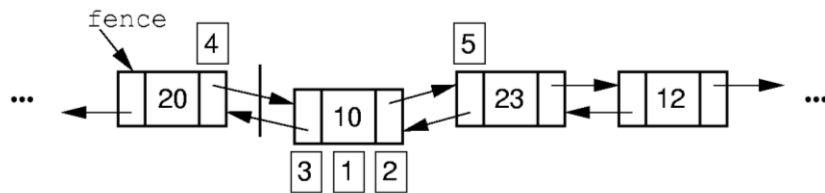
Inserimento



Insert 10:

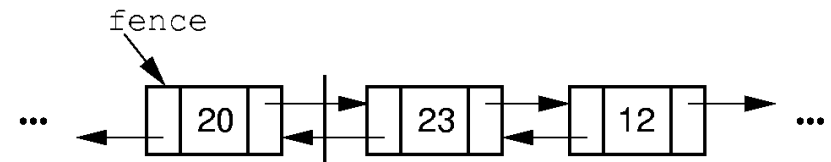
	10	
--	----	--

(a)

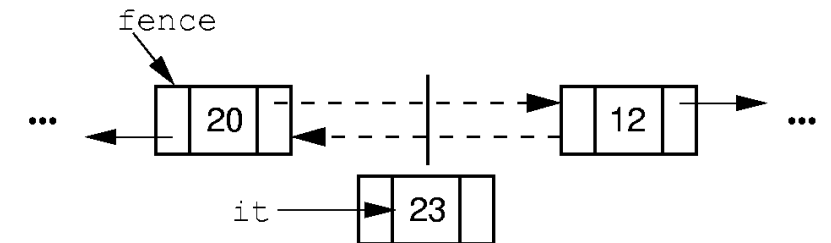


(b)

Cancellazione



(a)



(b)

Dobbiamo definire un tipo di dato astratto
(tipo dei dati + operazioni possibili)

Vogliamo astrarre dall'implementazione

Lista = sequenza ordinata di elementi

Differenti implementazioni (array, liste puntate)
Ma stesse funzionalità!

Definizione dei dati insieme alle funzionalità
Separazione fra definizione e implementazione

Programmazione a oggetti

```
typedef struct {
    Elem lista[20];    // array che contiene il # di elementi
    int  n;           // # di elementi nella lista
} ListElem;

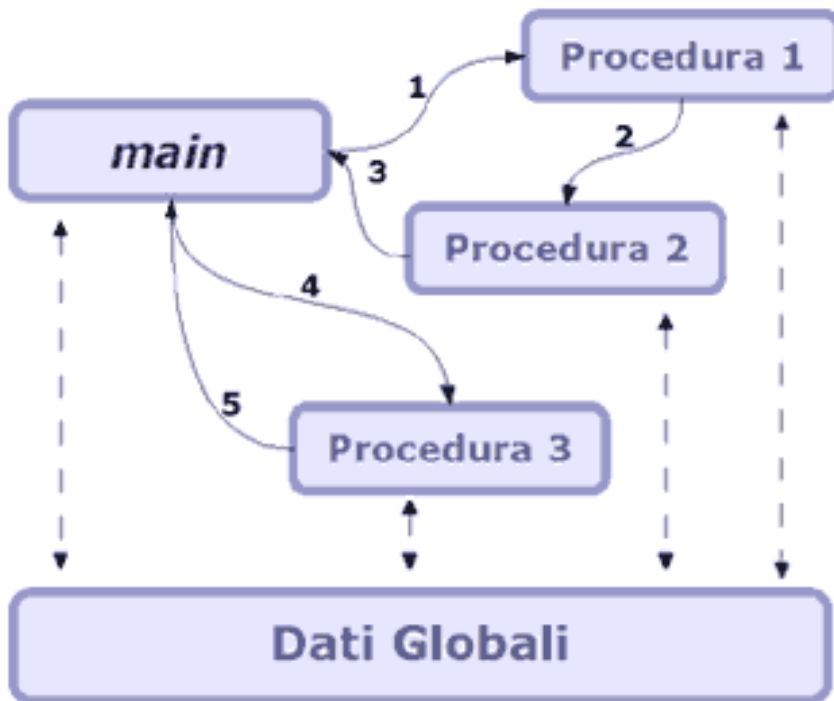
void InitListaElem(ListaElem &lista)
{
    lista.n = 0;
}

void InsListaElem(ListaElem &lista, Elem x)
{
    ...
}

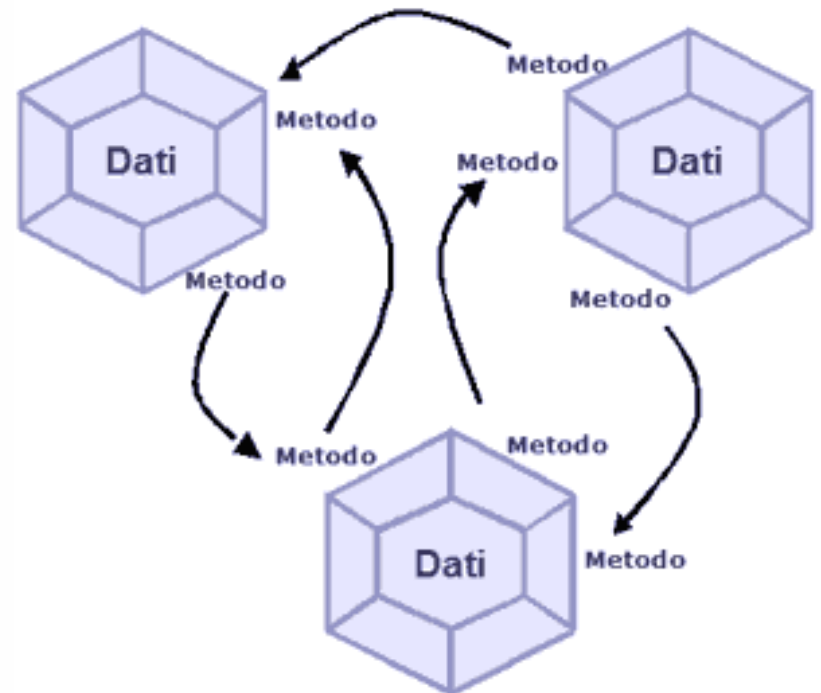
ListaElem MyList;
```

Paradigmi di Programmazione

Programmazione Procedurale



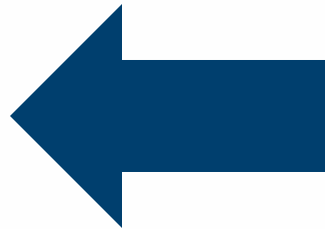
Programmazione Orientata agli Oggetti



- Oggetti
- Classi
- Interazione tra oggetti
- Incapsulamento
- Interfaccia
- Accesso agli attributi

Automobile
Velocità Colore # porte Marca
Avvia Accelera Frena Gira ...

Attributi



Metodi

Velocità = 80
Colore = blu
porte = 4
Marca = ...
Avvia/Accelera/
Frena/Gira



Velocità = 120
Colore = grigio
porte = 2
Marca = ...
Avvia/Accelera/
Frena/Gira



Classe

Oggetti/Istanze

- ❑ Specifica gli attributi, senza indicarne il valore
- ❑ Specifica i metodi che devono avere gli oggetti appartenenti alla classe
- ❑ La classe serve per generare più oggetti con gli stessi attributi e gli stessi metodi
- ❑ Gli oggetti creati a partire da una classe sono chiamati istanze della classe
- ❑ Due istanze della stessa classe
 - ▶ Hanno valori diversi
 - ▶ Stesso comportamento (i metodi sono gli stessi)

- ❑ Indica la proprietà degli oggetti di incorporare al loro interno sia gli attributi (le caratteristiche) che i metodi (il comportamento)
- ❑ Tutto ciò a cui si riferisce ad un certo oggetto è racchiuso e contenuto all'interno dell'oggetto stesso
- ❑ Gli attributi e i metodi sono incapsulati nell'oggetto.
- ❑ In questo modo tutte le informazioni utili che riguardano un oggetto sono al suo interno. Questo è uno dei vantaggi della programmazione ad oggetti

- ❑ Elenco dei metodi
 - ▶ Nome
 - ▶ Numero e il tipo dei parametri
 - ▶ Il valore di ritorno del metodo

- ❑ Chi utilizza l'oggetto deve conoscere solo la sua interfaccia.

- ❑ Può sapere,
 - ▶ Quali metodi possono essere invocati,
 - ▶ Quali sono i parametri da passare,
 - ▶ Che tipo di dato ricaverà come risultato

Non deve conoscere
i dettagli dell'implementazione

- ❑ Attributi e metodi sono
 - ▶ Pubblici: quando chiunque li può modificare e invocare
 - ▶ Privati: quando possono essere modificati e invocati solo dall'oggetto stesso

- ❑ Attributi e metodi privati sono nascosti nell'oggetto

Esempio: tipo di dato astratto in C++

```
class point {
private:
    float x, y;
public:
    // costruttore
    point (float a, float b)
        {x = a; y = b;};
    // distruttore
    ~point() {};
    void x_move (float a)
        {x += a;};
    void y_move (float b)
        {y += b;};
    void reset ()
        { x = y = 0.0;};
};

point p1 (1.3, 3.7);

point p2 (53.3, 0.0);

point * p3 =
    new point(0.0, 0.0);

p1.x_move(9.3);

*p3 = p2;

p1.X = 3.4;
```

❑ Costruttori

- ▶ Hanno lo stesso nome della classe “**point (float a, float b)**”
- ▶ Possono essere molti, con prototipi differenti
- ▶ Costruttore di default, usato se non ne sono stati definiti

❑ Distruttori

- ▶ Stesso nome della classe preceduto da una ~
- ▶ Fa il cleanup delle informazioni contenute nell'oggetto dopo il suo ultimo utilizzo, prima della deallocazione
- ▶ C'è un solo distruttore per ogni classe
- ▶ Distruttore di default, usato se non ne è stato definito uno

- ❑ Quando l'oggetto viene creato
 - ▶ L'oggetto è allocato
 - ▶ Gli attributi sono inizializzati come specificato nel costruttore
 - ▶ L'allocazione può essere automatica (nelle dichiarazioni) o esplicita (via new)

- ❑ Quando l'oggetto viene distrutto
 - ▶ Gli attributi sono ripuliti
 - ▶ L'oggetto viene deallocato (automaticamente o esplicitamente)

Costruttori per Copia

- ❑ Tipo speciale di costruttore: crea un'oggetto copiandolo da un altro oggetto dello stesso tipo
- ❑ Non è un assegnamento, l'oggetto non esiste (ancora)
- ❑ È anche utilizzato per il passaggio dei parametri

```
point(const point& p)
{
    // x and y private parts
    // of the object itself
    x = p.x;
    y = p.y;
}
```

- Due notazioni

```
point p1=p2;  
point p1(p2);
```

- La dichiarazione è del tipo

```
point(const point&)
```

Tipo di Dato Astratto per le Liste

```
template <class Elem> class List {  
public:  
    virtual void clear() = 0;  
    virtual bool insert(const Elem&) = 0;  
    virtual bool append(const Elem&) = 0;  
    virtual bool remove(Elem&) = 0;  
    virtual void setStart() = 0;  
    virtual void setEnd() = 0;  
    virtual void prev() = 0;  
    virtual void next() = 0;  
    virtual bool setPos(int pos) = 0;  
    virtual bool getValue(Elem&) const = 0;  
};
```



pure
virtual

Nessun metodo è implementato
La classe non è utilizzabile in questa forma!

```
template <class Elem> // Array-based list
class AList : public List<Elem> {
private:
    int maxSize;        // Maximum size of list
    int listSize;      // Actual elem count
    int fence;         // Position of fence
    Elem* listArray;   // Array holding list
public:
    AList(int size=DefaultListSize) {
        maxSize = size;
        listSize = fence = 0;
        listArray = new Elem[maxSize];
    }
    ...
}
```

**Classe derivata
dalla classe
precedente**

```
~AList() { delete [] listArray; }

void clear() {
    delete [] listArray;
    listSize = fence = 0;
    listArray = new Elem[maxSize];
}

void setStart() { fence = 0; }
void setEnd() { fence = listSize; }
void prev() { if (fence != 0) fence--; }
void next() { if (fence <= listSize) fence++; }

int leftLength() const { return fence; }

int rightLength() const { return listSize - fence; }
```

Classe per Liste Basate su Array: Inserimento in testa

```
// Insert at front of right partition

template <class Elem>
bool AList<Elem>::insert(const Elem& item) {

    if (listSize == maxSize) return false;

    for(int i=listSize; i>fence; i--)
        // Shift Elems up to make room
        listArray[i] = listArray[i-1];
    listArray[fence] = item;
    listSize++; // Increment list size

    return true;
}
```

Classe per Liste Basate su Array: Append

```
// Append Elem to end of the list
template <class Elem>
bool AList<Elem>::append(const Elem& item) {
    if (listSize == maxSize) return false;
    listArray[listSize++] = item;
    return true;
}
```

Stack...

- Una pila o stack è un insieme dinamico in cui l'elemento rimosso dall'operazione di cancellazione è predeterminato: "quello che per meno tempo è rimasto nell'insieme" politica "last in, first out" (LIFO)

- **Operazioni previste**

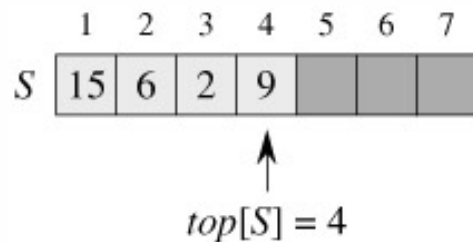
- ▶ void push(Item) // inserisce un elemento
- ▶ Item pop() // rimuove l'elemento
// in cima
- ▶ Item top() // legge l'item in cima
// non rimuove l'item
- ▶ boolean isEmpty() // true se la pila è vuota



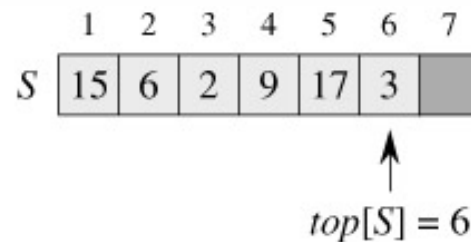
- ❑ Reverse Polish Notation (RPN), o notazione postfissa
- ❑ Espressioni aritmetiche in cui gli operatori seguono gli operandi
- ❑ Esempi
 - ▶ $(7 + 3) \times 5$ si traduce in $7\ 3\ +\ 5\ \times$
 - ▶ $7 + 3 \times 5$ si traduce in $7\ 3\ 5\ \times\ +$

□ Possibili implementazioni

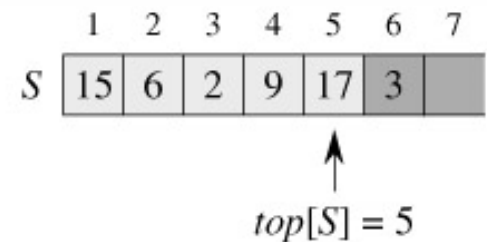
- ▶ Tramite liste puntate doppie con puntatore all'elemento top, per estrazione/inserimento
- ▶ Tramite array, dimensione limitata, ma semplice



(a)



(b)



(c)

STACK-EMPTY(S)

- 1 if $\text{top}[S] = 0$
- 2 then return TRUE
- 3 else return FALSE

PUSH(S, x)

- 1 $\text{top}[S] \leftarrow \text{top}[S] + 1$
- 2 $S[\text{top}[S]] \leftarrow x$

POP(S)

- 1 if STACK-EMPTY(S)
- 2 then error "underflow"
- 3 else $\text{top}[S] \leftarrow \text{top}[S] - 1$
- 4 return $S[\text{top}[S] + 1]$

Stack: Definizione del Tipo di Dato Astratto

```
// Stack abstract class
template <class Elem> class Stack {
public:
    // Reinitialize the stack
    virtual void clear() = 0;
    // Push an element onto the top of the stack.
    virtual bool push(const Elem&) = 0;
    // Remove the element at the top of the stack.
    virtual bool pop(Elem&) = 0;
    // Get a copy of the top element in the stack
    virtual bool topValue(Elem&) const = 0;
    // Return the number of elements in the stack.
    virtual int length() const = 0;
};
```

Implementazione dello Stack: Array e Liste Puntate

```
// Array-based stack implementation
private:
    int size;        // Maximum size of stack
    int top;        // Index for top element
    Elem *listArray; // Array holding elements

// Linked stack implementation
private:
    Link<Elem>* top; // Pointer to first elem
    int size;        // Count number of elems
```

Quali scelte implementative?
Quanto costano le operazioni?
Confronto dei requisiti di memoria?

Code...

- ❑ insieme dinamico in cui l'elemento rimosso dall'operazione di cancellazione è predeterminato: "quello che per più tempo è rimasto nell'insieme"
- ❑ La politica di funzionamento è "first in, first out" (FIFO)
- ❑ Operazioni previste
 - ▶ void enqueue(Item) // sinonimi: put, add, insert
 - ▶ Item dequeue() // sinonimi: removeFirst, extract
 - ▶ Item head() // non rimuove l'item; sinonimi get
 - ▶ boolean isEmpty()



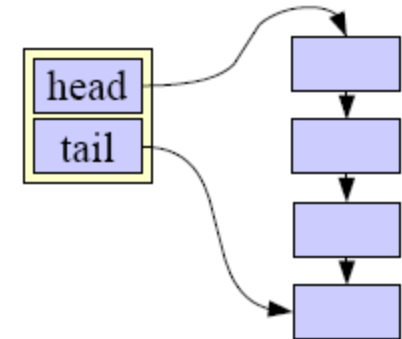
Quanto costano le operazioni?

□ Esempi

- ▶ Nei sistemi operativi, i processi in attesa di utilizzare una risorsa vengono gestiti tramite una coda
- ▶ Prenotazione dei biglietti dei concerti in linea

□ Possibili implementazioni

- ▶ Liste puntate semplici
- ▶ Array

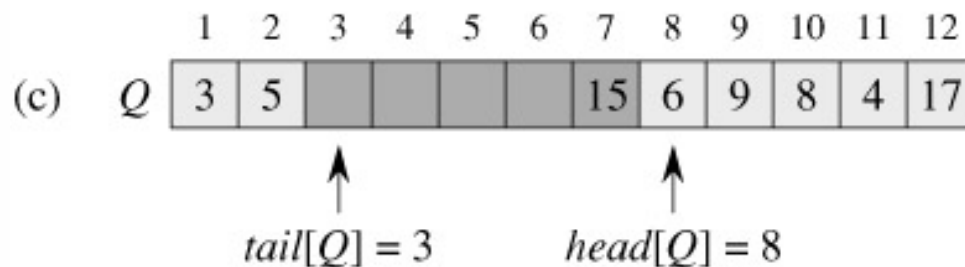
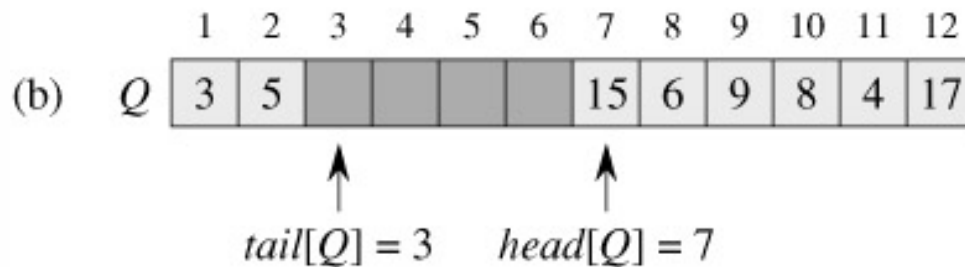
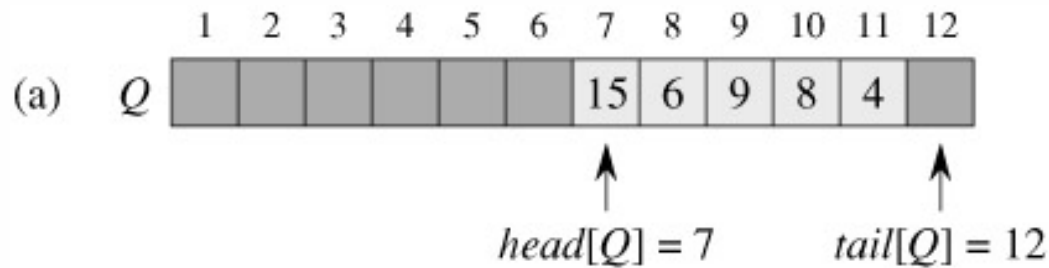


□ Liste puntate semplici

- ▶ Puntatore head (inizio della coda), per estrazione
- ▶ puntatore tail (inizio della coda), per inserimento

□ Tramite array circolari

- ▶ Dimensione limitata, overhead più basso



a) Coda iniziale

b) Dopo aver svolto
 ENQUEUE(Q , 17),
 ENQUEUE(Q , 3), e
 ENQUEUE(Q , 5)

c) Dopo aver svolto
 DEQUEUE(Q) che
 ritorna il valore 15

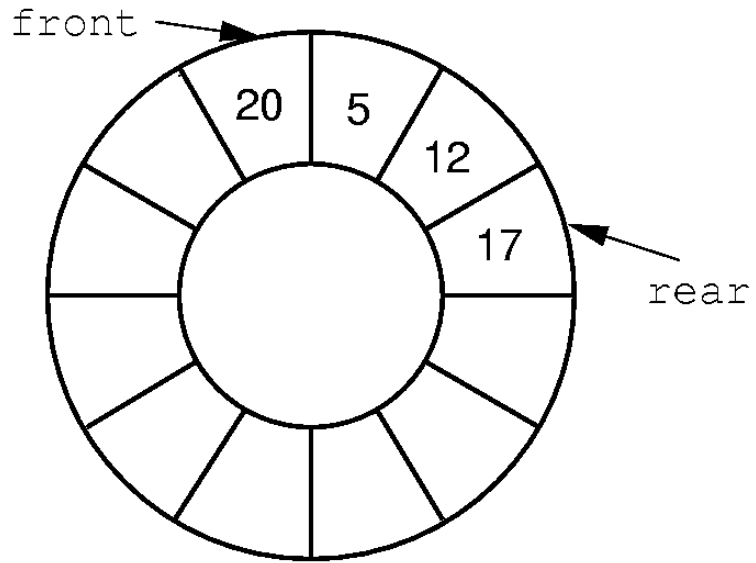
ENQUEUE (Q, x)

```
1  Q[tail[Q]] ← x
2  if tail[Q] = length[Q]
3     then tail[Q] ← 1
4  else
5     tail[Q] ← tail[Q]+1
```

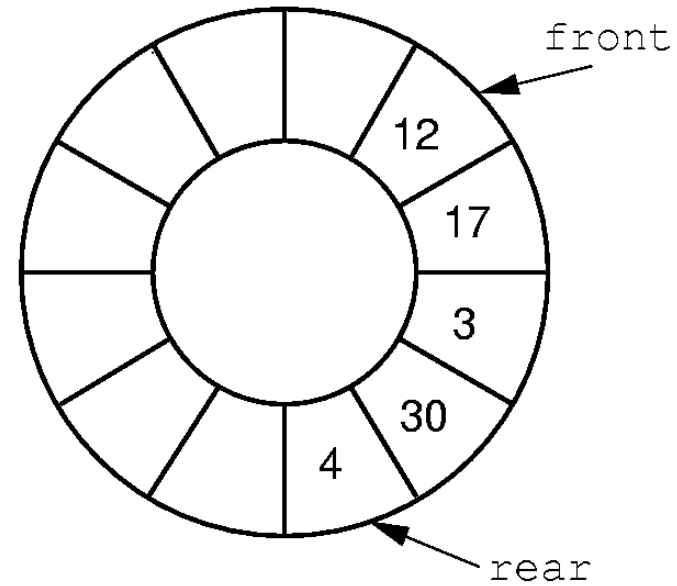
DEQUEUE (Q)

```
1  x ← Q[head[Q]]
2  if head[Q] = length[Q]
3     then head[Q] ← 1
4  else
5     head[Q] ← head[Q]+1
6  return x
```

Implementazione delle Code: Array Circolari



(a)

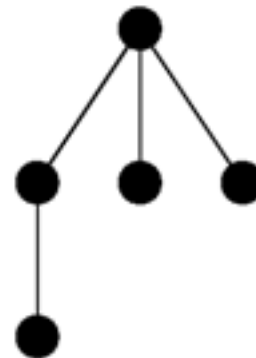
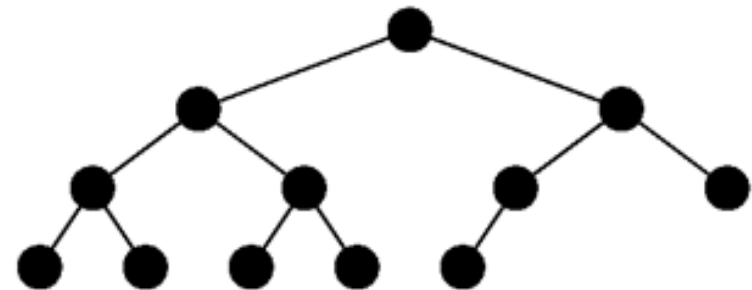
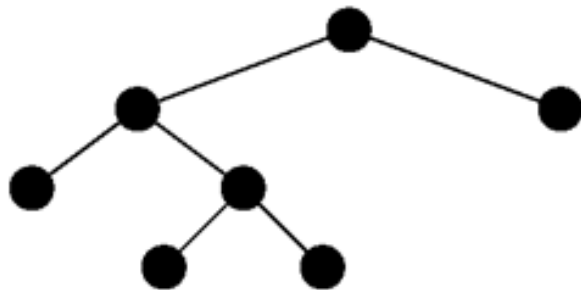


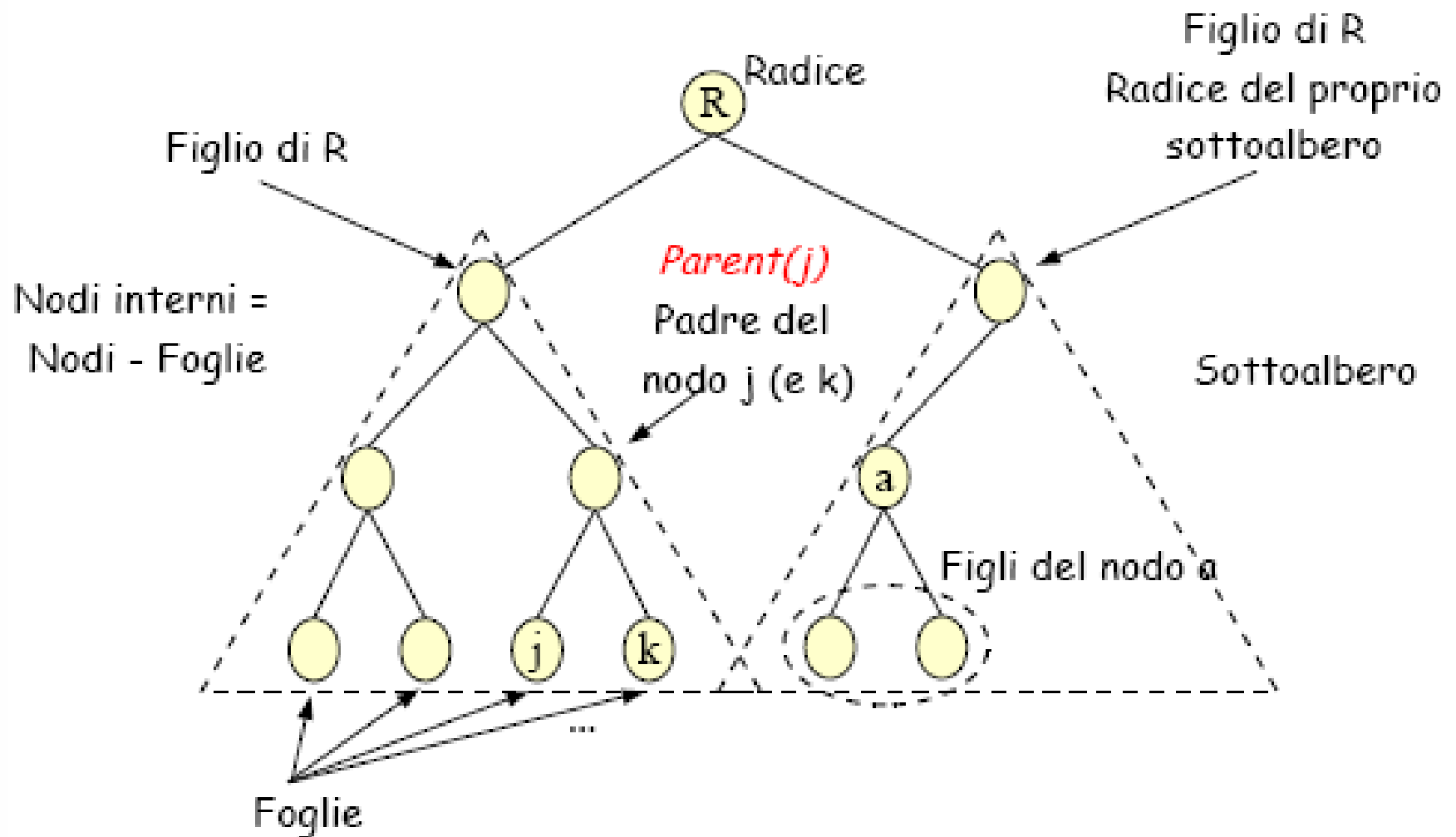
(b)

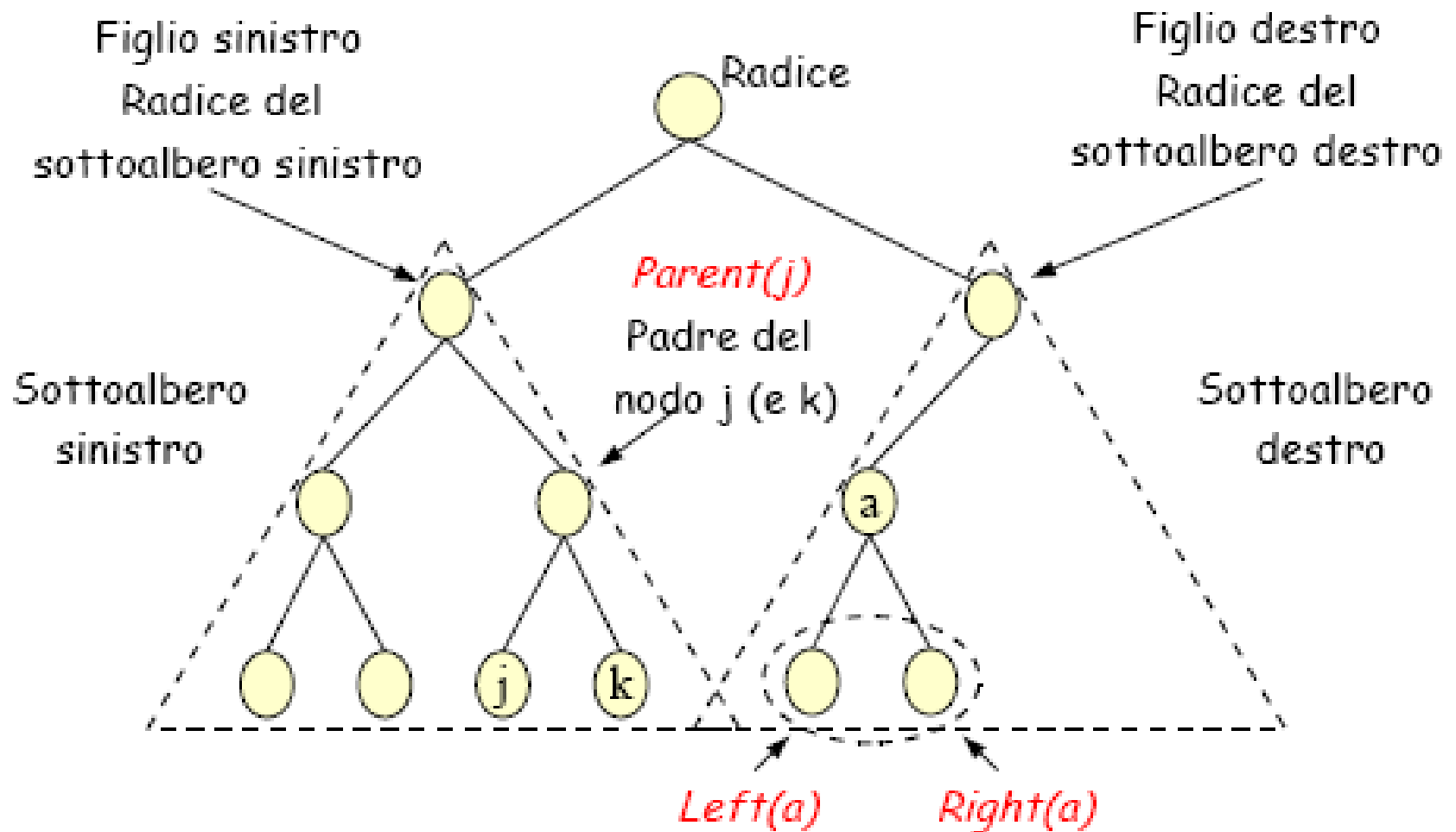
Alberi radicati...

□ Albero radicato

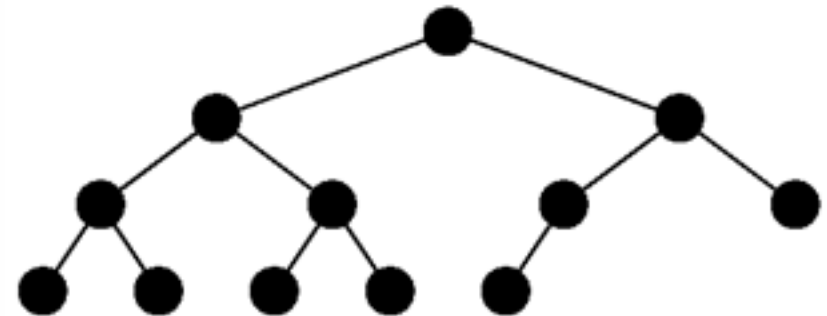
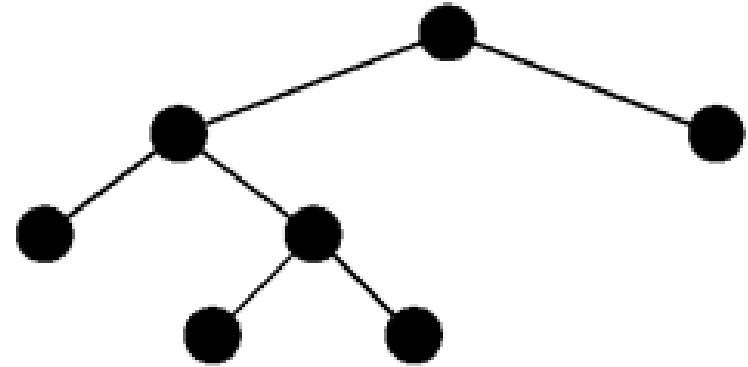
- ▶ Un insieme vuoto di nodi
- ▶ Un nodo radice R collegato a 0 o più alberi (sottoalberi)





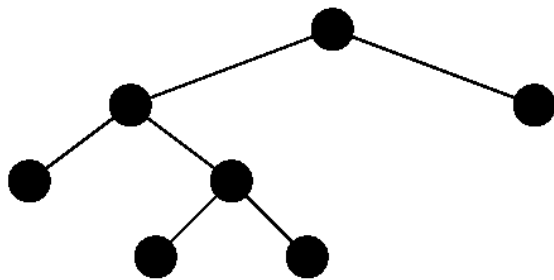


- Profondità di un nodo: la lunghezza del percorso dalla radice al nodo (numero archi attraversati)
- Livello: l'insieme dei nodi alla stessa profondità
- Altezza dell'albero: massima profondità+1
- Grado di un nodo: numero dei figli

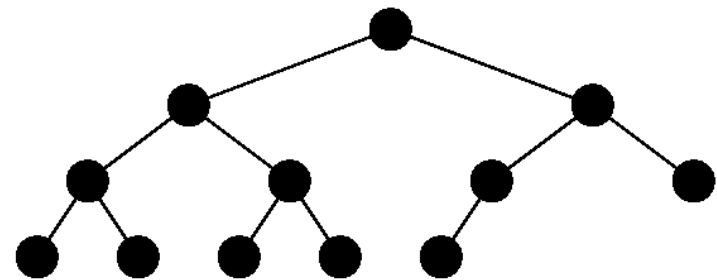


Alberi Binari: Pieni e Completi

- ❑ Alberi binari pieni: Ogni nodo è una foglia oppure è un nodo interno con esattamente due figli non vuoti
- ❑ Alberi binari completi: se l'altezza dell'albero è d , allora tutte le foglie eccetto possibilmente il libello d sono completamente piene. L'ultimo livello ha tutti i nodi sul lato sinistro



(a)



(b)

- **Teorema:** Il numero di foglie in un albero binario pieno non vuoto è uno di più del numero di nodi interni
- **Dimostrazione:** per induzione sul numero di nodi.

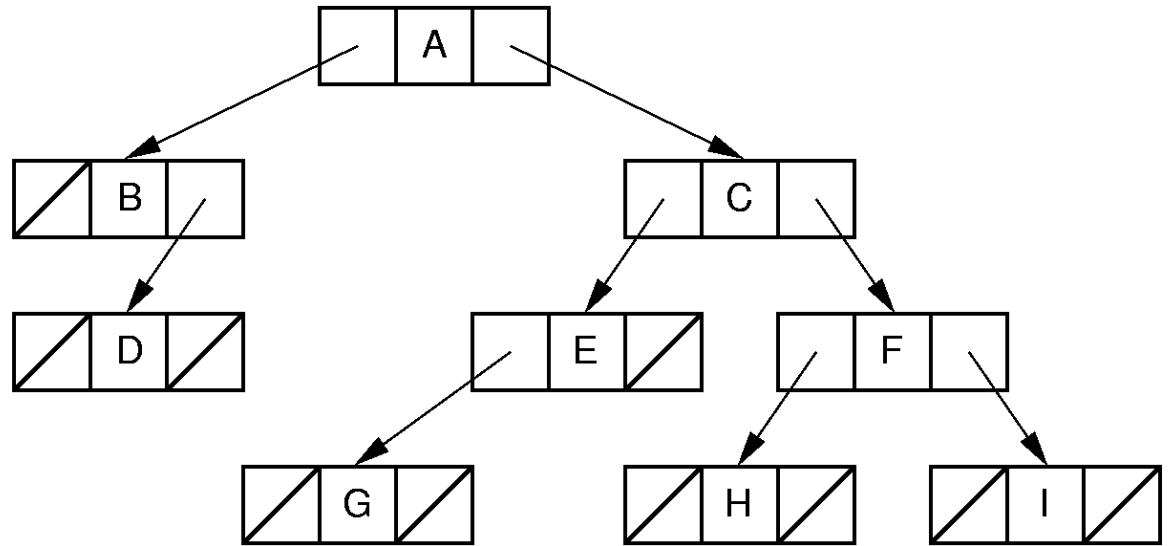
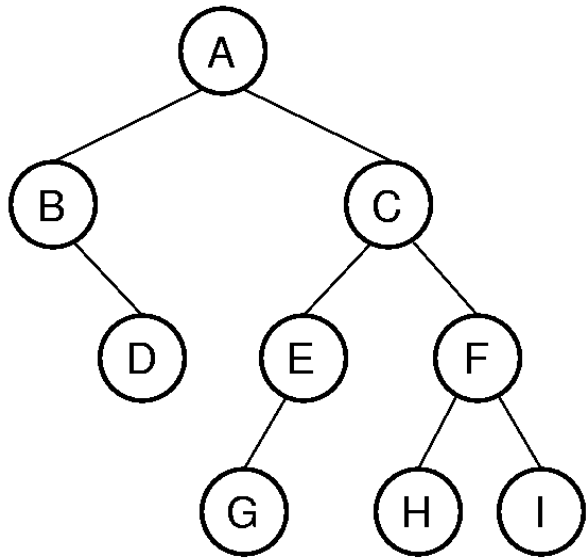
Caso base: un albero binario pieno con 1 nodo interno deve avere due foglie

Ipotesi induttiva: assumiamo che un albero binario T con $n-1$ nodi interni abbia n foglie

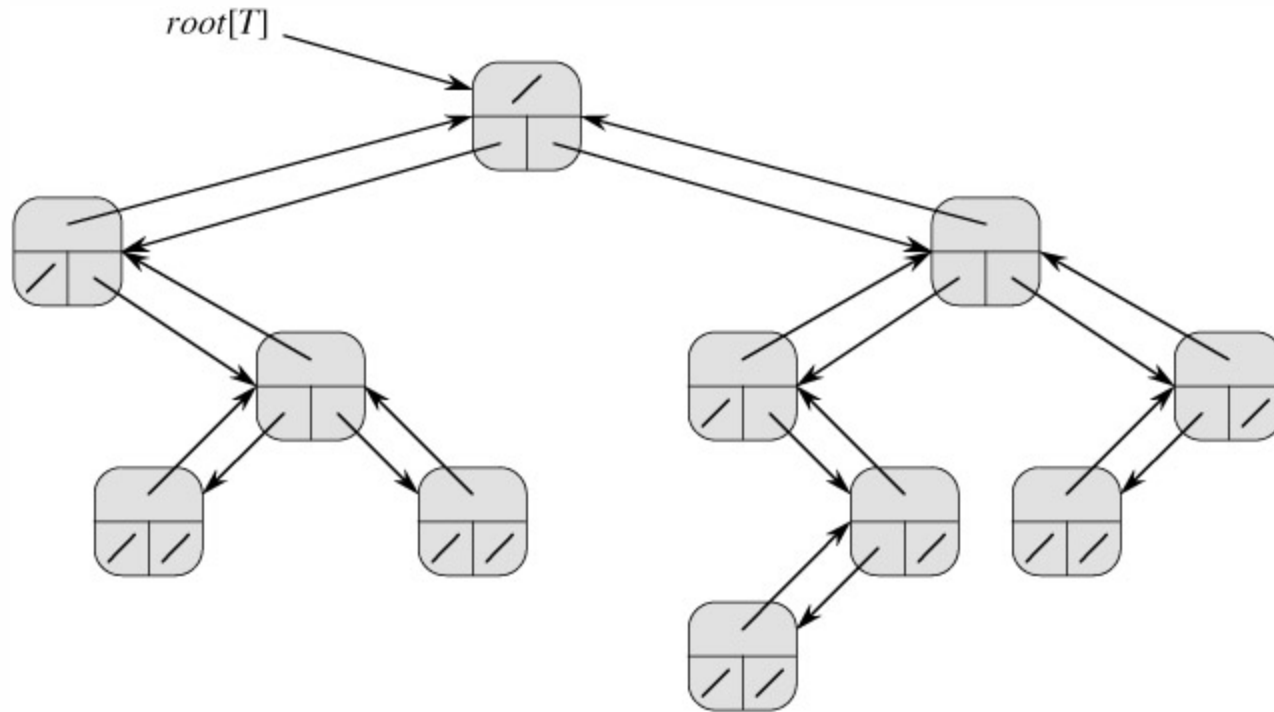
Passo induttivo: dato un albero T con n nodi interni, prendiamo un nodo interno con due figli, rimuovendo i due figli otteniamo l'albero T' (che ha n foglie)

Il numero di nodi interni è aumentato di uno per raggiungere e il numero di foglie è anche aumentato di uno.

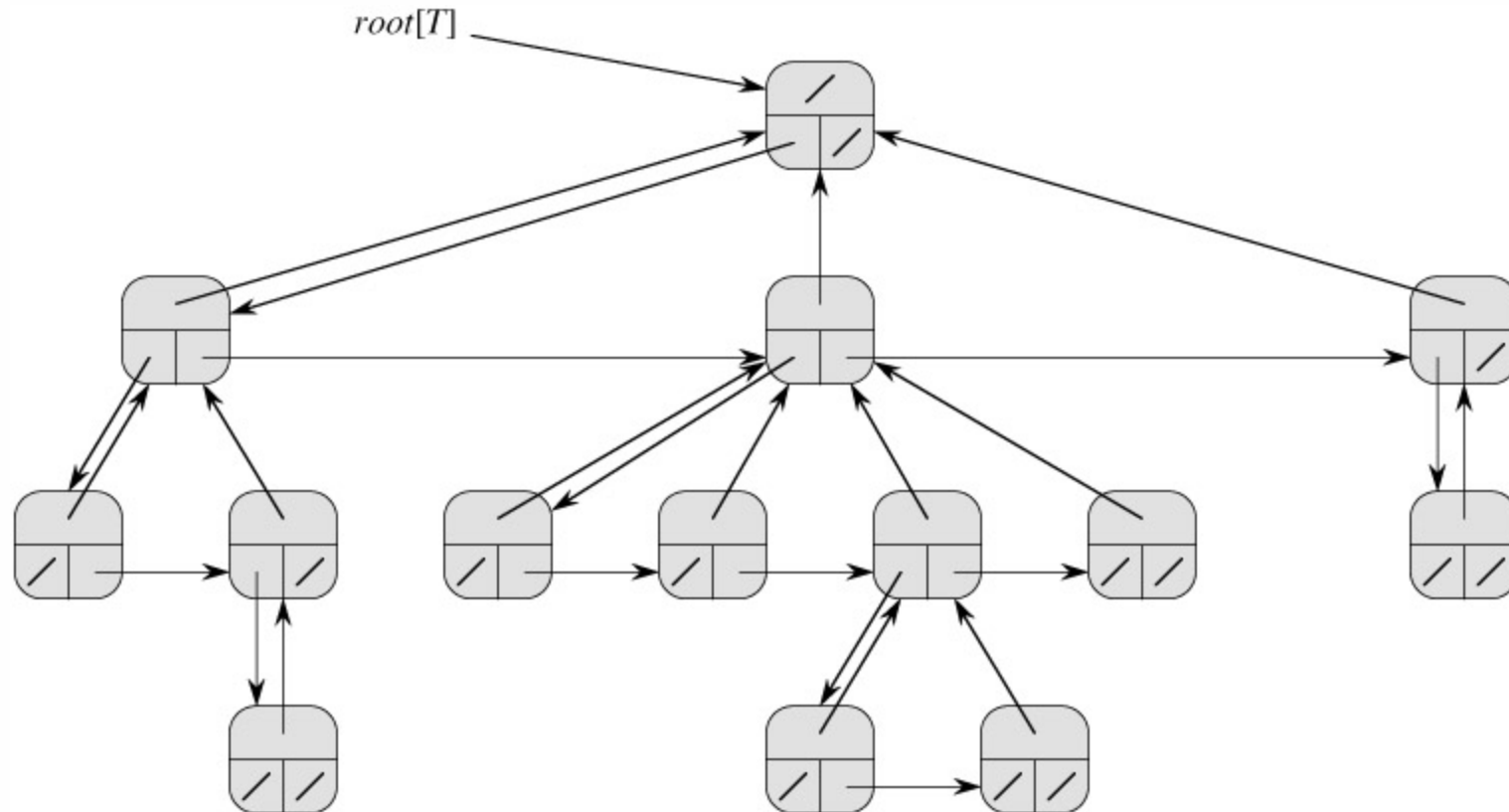
Rappresentazione degli Alberi Binari

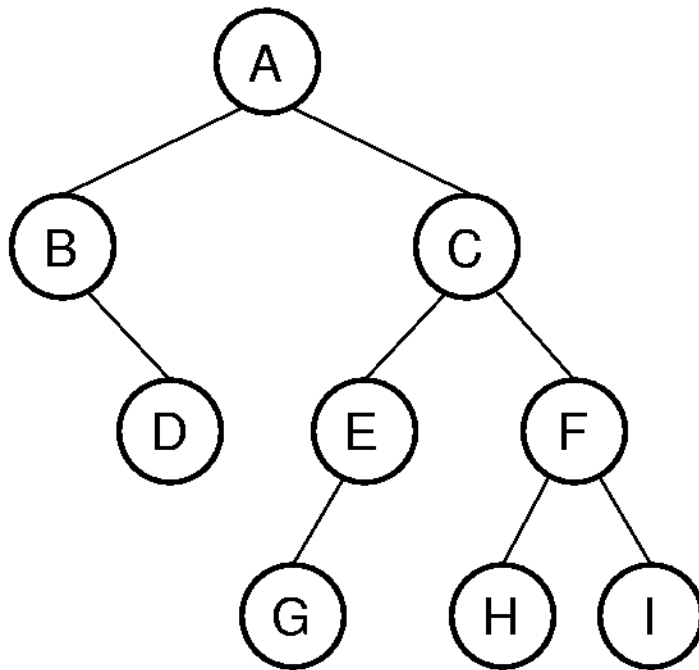


Altre Rappresentazioni: Con puntatore al parente



Altre Rappresentazioni: Con puntatori ai nodi dello stesso livello





	Left	Key	Right	Par
0	1	A	3	-1
1	-1	B	2	0
2	-1	D	-1	1
3	4	C	6	0
4	5	E	-1	3
5	-1	G	-1	4
6	7	F	8	3
7	-1	H	-1	6
8	-1	I	-1	6

```
// Binary tree node class
template <class Elem>
class BinNodePtr : public BinNode<Elem> {
private:
    Elem it;           // The node's value
    BinNodePtr* lc;   // Pointer to left child
    BinNodePtr* rc;   // Pointer to right child
public:
    BinNodePtr() { lc = rc = NULL; }
    BinNodePtr(Elem e, BinNodePtr* l =NULL,
               BinNodePtr* r =NULL)
        { it = e; lc = l; rc = r; }
```

```
Elem& val() { return it; }
void setVal(const Elem& e) { it = e; }
inline BinNode<Elem>* left() const
    { return lc; }
void setLeft(BinNode<Elem>* b)
    { lc = (BinNodePtr*)b; }
inline BinNode<Elem>* right() const
    { return rc; }
void setRight(BinNode<Elem>* b)
    { rc = (BinNodePtr*)b; }
bool isLeaf()
    { return (lc == NULL) && (rc == NULL); }
};
```

- ❑ **Visita (o attraversamento) di un albero:**
 - ▶ Algoritmo per “visitare” tutti i nodi di un albero

- ❑ **In profondità (depth-first search, a scandaglio): DFS**
 - ▶ Vengono visitati i rami, uno dopo l’altro
 - ▶ Tre varianti: pre-ordine, post-ordine, in-ordine

- ❑ **In ampiezza (breadth-first search, a ventaglio): BFS**
 - ▶ A livelli, partendo dalla radice

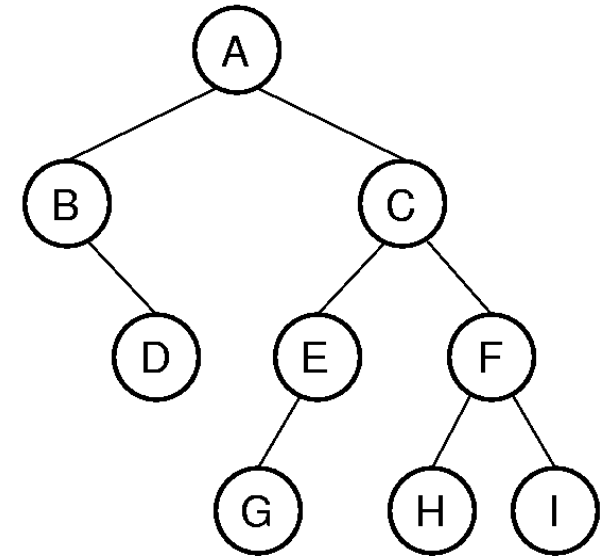
```
template <class Elem> // Pre-ordine
void preorder(BinNode<Elem>* subroot) {
    if (subroot == NULL) return; // Empty
    visit(subroot); // Perform some action
    preorder(subroot->left());
    preorder(subroot->right());
}
```

```
template <class Elem> // In-ordine
void inorder(BinNode<Elem>* subroot) {
    if (subroot == NULL) return; // Empty
    preorder(subroot->left());
    visit(subroot); // Perform some action
    preorder(subroot->right());
}
```

```
template <class Elem> // Post-ordine
void preorder(BinNode<Elem>* subroot) {
    if (subroot == NULL) return; // Empty
    preorder(subroot->left());
    preorder(subroot->right());
    visit(subroot); // Perform some action
}
```

```
VisitaAmpiezza(T)
  q = new Queue()
  q.insert(T)
  while not q.empty() do
    p := q.dequeue()
    visita p
    q.enqueue(p.left())
    q.enqueue(p.right())
```

- Stampare il risultato della
 - ▶ Visita in pre-ordine
 - ▶ Visita in-ordine
 - ▶ Visita in post-ordine
 - ▶ Visita in ampiezza



- Scrivere un algoritmo per
 - ▶ Calcolare l'altezza di un albero binario T
 - ▶ Calcolare il numero di nodi di un albero binario T
 - ▶ Stampare tutti i nodi di profondità h di un albero binario T

Sommario

- ❑ Tipo di dato astratto = collezione di valori + operazioni ammesse su questi valori

- ❑ Strutture dati vs Implementazione

- ❑ Incapsula i dettagli dell'implementazione
 - ▶ Elencare le proprietà degli oggetti
 - ▶ Incorporare al loro interno gli attributi (le caratteristiche) che i metodi (il comportamento)
 - ▶ Oggetti/Classi/Interfaccia
 - ▶ Costruttori/Distruttori

- ❑ Strutture Dati
 - ▶ lineari/non lineari
 - ▶ statiche/dinamiche (variazione di dimensione, contenuto)
 - ▶ omogenee/disomogenee (dati contenuti)