

# Strutture dati elementari

Damiano Macedonio  
Università Ca' Foscari di Venezia

[mace@unive.it](mailto:mace@unive.it)

Original work Copyright © Alberto Montresor, University of Trento  
(<http://www.dit.unitn.it/~montreso/asd/index.shtml>)

Modifications Copyright © 2009, 2010, Moreno Marzolla, Università di Bologna

Modifications Copyright © 2012,2014 Damiano Macedonio, Università Ca' Foscari di Venezia

*This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike License. To view a copy of this license, visit*

*<http://creativecommons.org/licenses/by-nc-sa/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.*

# Introduzione

- **Tipo di dato astratto**
  - Un modello matematico, dato da una collezione di valori e un insieme di operazioni ammesse su questi valori
- **Tipo di dato primitivo**
  - Forniti direttamente dal linguaggio
  - Esempi: int (+, -, \*, /, %), boolean (!, &&, ||)
- **Tipo di dato**
  - “Specifica” e “implementazione” di un tipo di dato astratto
  - **Specifica**: “manuale d'uso”, nasconde i dettagli implementativi all'utilizzatore
  - **Implementazione**: realizzazione vera e propria

# Strutture dati

- I dati sono spesso riuniti in insiemi detti **strutture dati**
  - sono caratterizzati più dall'organizzazione dei dati che dal tipo dei dati stessi
  - il tipo dei dati contenuti può essere parametrico (*templates* in C++, o *generics* in Java)
- Una struttura dati è composta da:
  - un modo sistematico di organizzare i dati
  - un insieme di operazioni per manipolare la struttura
- Alcune tipologie di strutture dati:
  - lineari / non lineari (presenza di una sequenza)
  - statiche / dinamiche (variazione di dimensione, contenuto)
  - omogenee / disomogenee (dati contenuti)

# Esempio: Dizionario (Insieme dinamico)

- Struttura dati “generale”: **insieme dinamico**
  - Può crescere, contrarsi, cambiare contenuto
  - Operazioni base: inserimento, cancellazione, ricerca
- Esempio di specifica:

```
public interface Dizionario {  
    public void insert(Object e, Comparable k);  
    public void delete(Comparable k);  
    public Object search(Comparable k);  
}
```

# Alcune possibili implementazioni Array ordinati

- search(Key k)
  - Cerca la chiave k mediante ricerca binaria
  - Costo:  $O(\log n)$
- insert(Key k, Item o)
  - Inserisci un nuovo elemento, spostando a destra di una posizione tutti gli elementi successivi
  - Costo:  $O(n)$
- delete(Key k)
  - Cerca la chiave k mediante ricerca binaria
  - Elimina l'elemento contenente la chiave k, spostando a sinistra di una posizione tutti gli elementi successivi
  - Costo:  $O(n)$

# Alcune implementazioni possibili

## Liste concatenate (non ordinate)

- search(Key k)
  - Cerca la chiave k mediante ricerca sequenziale
  - Costo:  $O(n)$
- insert(Key k, Item o)
  - Inserisci l'elemento all'inizio della lista
  - Costo complessivo:  $O(1)$
- delete(Key k)
  - Cerca la chiave k mediante ricerca sequenziale (costo  $O(n)$ )
  - Elimina l'elemento (costo  $O(1)$ )
  - Costo complessivo:  $O(n)$

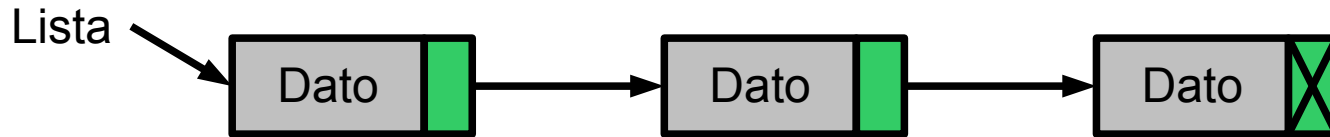
# Strutture dati elementari



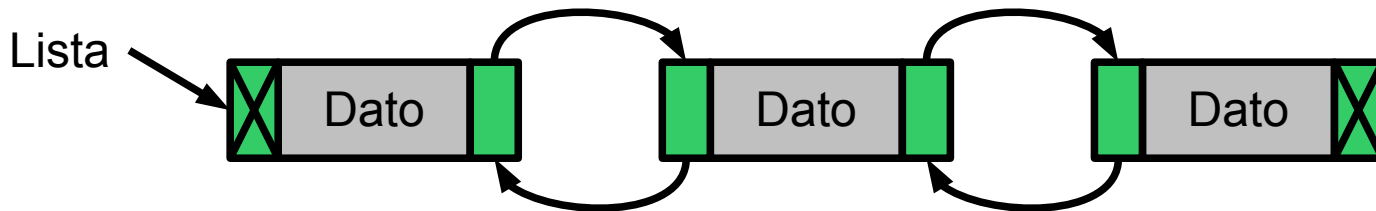
# Strutture collegate

alcune implementazioni possibili

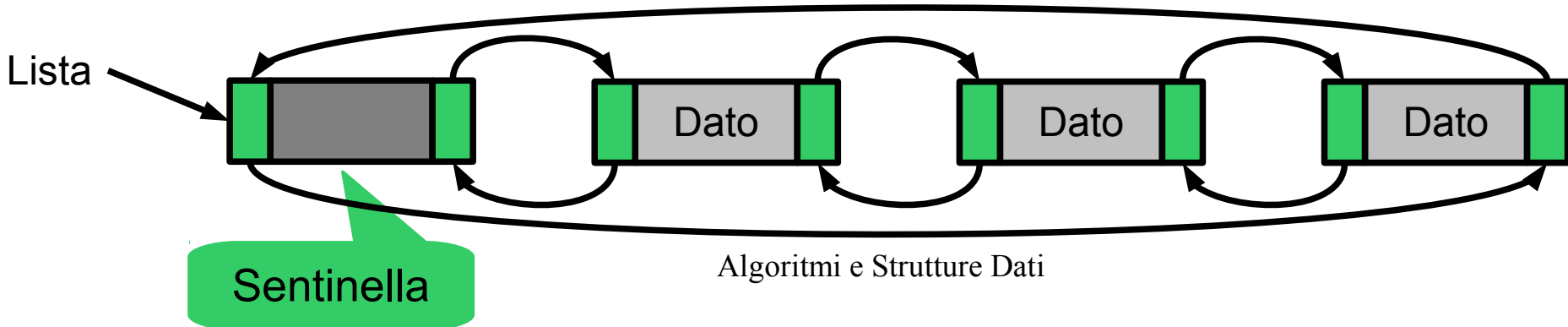
- Liste semplici



- Liste doppiamente concatenate



- Liste circolari con sentinella



# Esempio

```
// Implementazione basata su lista circolare
// SENZA sentinella
public class StrutturaCollegata implements Dizionario {

    private Record list = null;

    private final class Record { ... }

    public void insert(Object e, Comparable k)
    { ... }

    public void delete(Comparable k)
    { ... }

    public Object search(Comparable k)
    { ... }

}
```

# Esempio

```
private final class Record {
    public Object      elem;
    public Comparable chiave;
    public Record     next;
    public Record     prev;

    public Record(Object e, Comparable k) {
        elem = e;
        chiave = k;
        next = prev = null;
    }
}
```

# Esempio

```
public void insert(Object e, Comparable k) {  
    Record p = new Record(e, k);  
    if (list == null)  
        list = p.prev = p.next = p;  
    else {  
        p.next = list.next;  
        list.next.prev = p;  
        list.next = p;  
        p.prev = list;  
    }  
}
```

L'uso della sentinella eviterebbe la gestione di questo caso particolare

Il nuovo elemento viene inserito immediatamente dopo la testa della lista

Costo:  $O(1)$

# Esempio

```
public void delete(Comparable k) {
    Record p = null;
    if (list != null)
        for (p = list.next; ; p = p.next){
            if (p.chiave.equals(k)) break;
            if (p == list) { p = null; break; }
        }
    if (p == null)
        throw new EccezioneChiaveNonValida();
    if (p.next == p)
        list = null;
    else {
        if (list == p) list = p.next;
        p.next.prev = p.prev;
        p.prev.next = p.next;
    }
}
```



**Costo: O(n)**

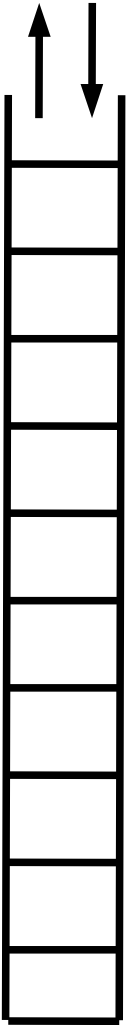
# Esempio

```
public Object search(Comparable k) {  
    if (list == null)  
        return null;  
    for (Record p = list.next; ; p = p.next){  
        if (p.chiave.equals(k))  
            return p.elem;  
        if (p == list)  
            return null;  
    }  
}
```

**Costo:  $O(n)$**

# Stack

- Insieme dinamico in cui l'elemento rimosso dall'operazione di cancellazione è l'ultimo inserito
  - politica “Last In, First Out” (LIFO)
- Operazioni previste
  - void push(Item) # inserisce un elemento in cima
  - Item pop() # rimuove l'elemento in cima
  - Item top() # non rimuove; legge solamente
  - boolean isEmpty()



# Stack

- Possibili utilizzi
  - Nei linguaggi con procedure: gestione dei record di attivazione
  - Nei linguaggi stack-oriented:
    - Tutte le operazioni elementari lavorano prendendo uno-due operandi dallo stack e inserendo il risultato nello stack
    - Es: Java bytecode
- Possibili implementazioni
  - Tramite liste puntate doppie
    - puntatore all'elemento top, per estrazione/inserimento
  - Tramite array
    - dimensione limitata, overhead più basso



# Interfaccia Pila

```
public interface Pila {
    /**
     * Verifica se la pila è vuota.
     */
    public boolean isEmpty();
    /**
     * Aggiunge l'elemento in cima
     */
    public void push(Object e);
    /**
     * Restituisce l'elemento in cima
     */
    public Object top();
    /**
     * Cancella l'elemento in cima
     */
    public Object pop();
}
```

# Implementare una pila tramite array

(metodo del raddoppiamento/dimezzamento)

```
public class PilaArray implements Pila
{
    private Object[] S = new Object[1];
    private int n = 0;

    public boolean isEmpty() {
        return n == 0;
    }

    public void push(Object e) { ... }
    public Object top() { ... }
    public Object pop() { ... }
}
```

# PilaArray: metodo top()

```
public Object top() {  
    if (this.isEmpty())  
        throw new EccezioneStrutturaVuota("Pila vuota");  
    return S[n - 1];  
}
```



**Costo:  $O(1)$**

# PilaArray: metodo push()

```
public void push(Object e) {  
    if (n == S.length) {  
        Object[] temp = new Object[2 * S.length];  
        for (int i = 0; i < n; i++)  
            temp[i] = S[i];  
        S = temp;  
    }  
    S[n] = e;  
    n = n + 1;  
}
```



**Costo:???**

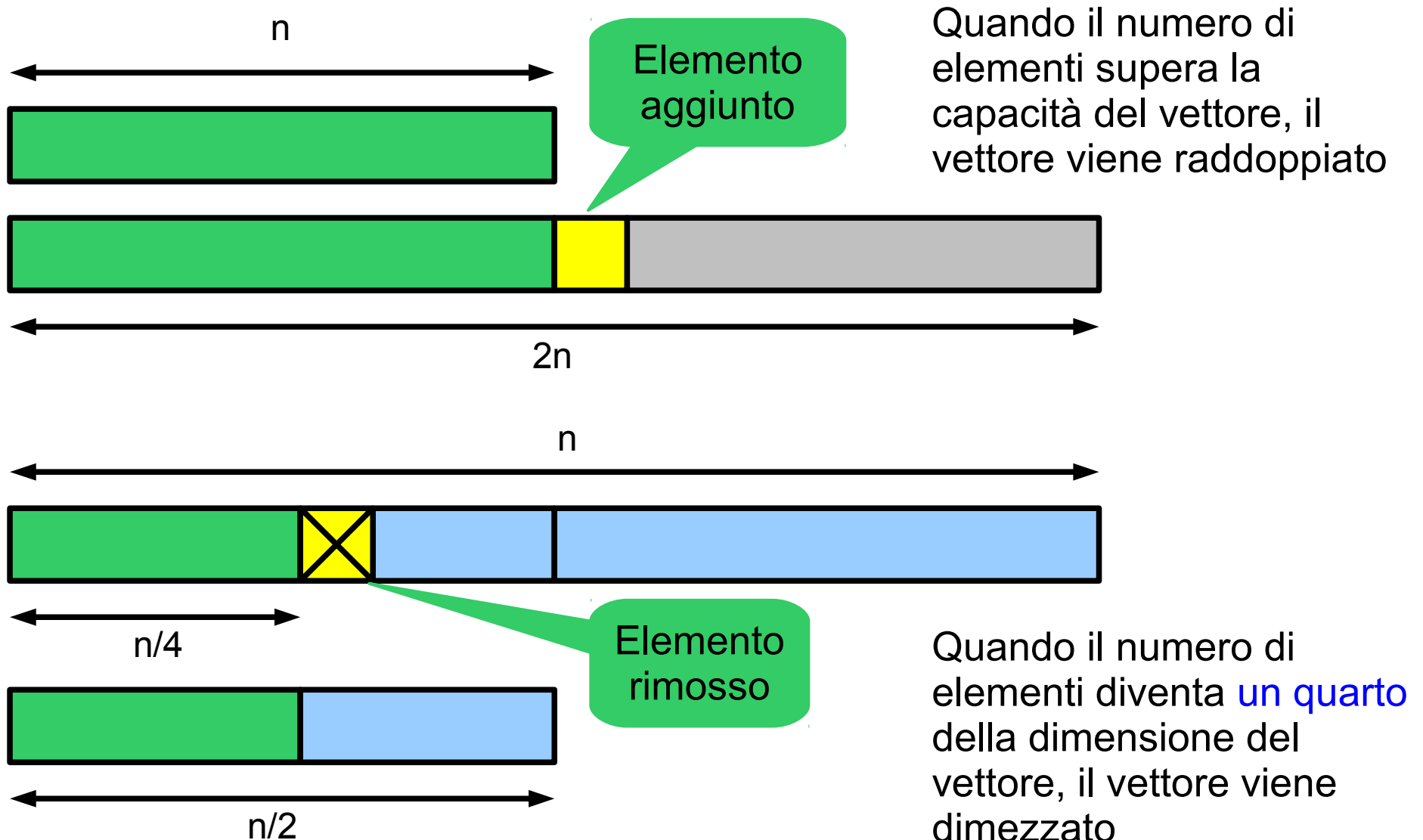
# PilaArray: metodo pop()

```
public Object pop() {
    if (this.isEmpty())
        throw new EccezioneStrutturaVuota("Pila vuota");
    n = n - 1;
    Object e = S[n];
    if (n > 1 && n == S.length / 4) {
        Object[] temp = new Object[S.length / 2];
        for (int i = 0; i < n; i++)
            temp[i] = S[i];
        S = temp;
    }
    return e;
}
```

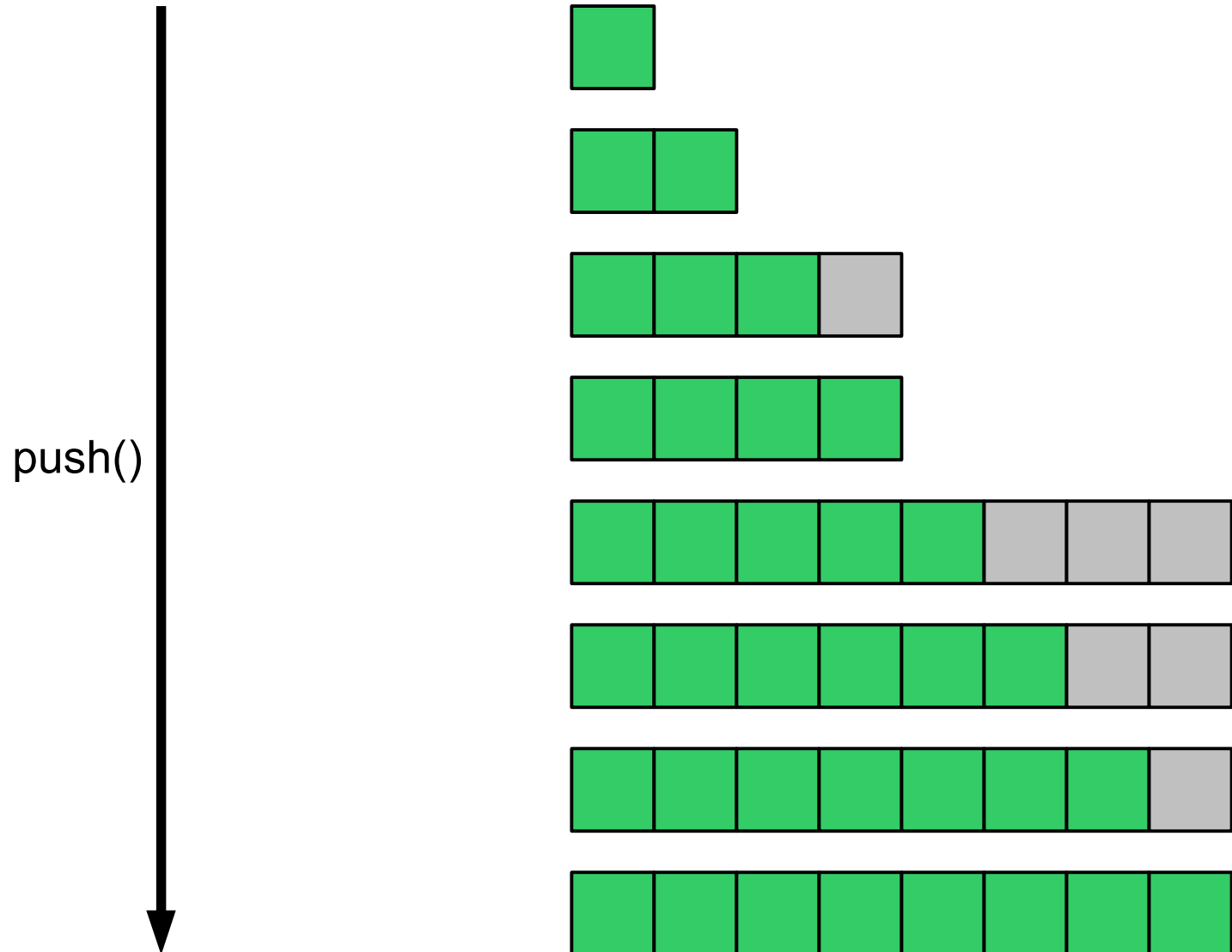


**Costo: ???**

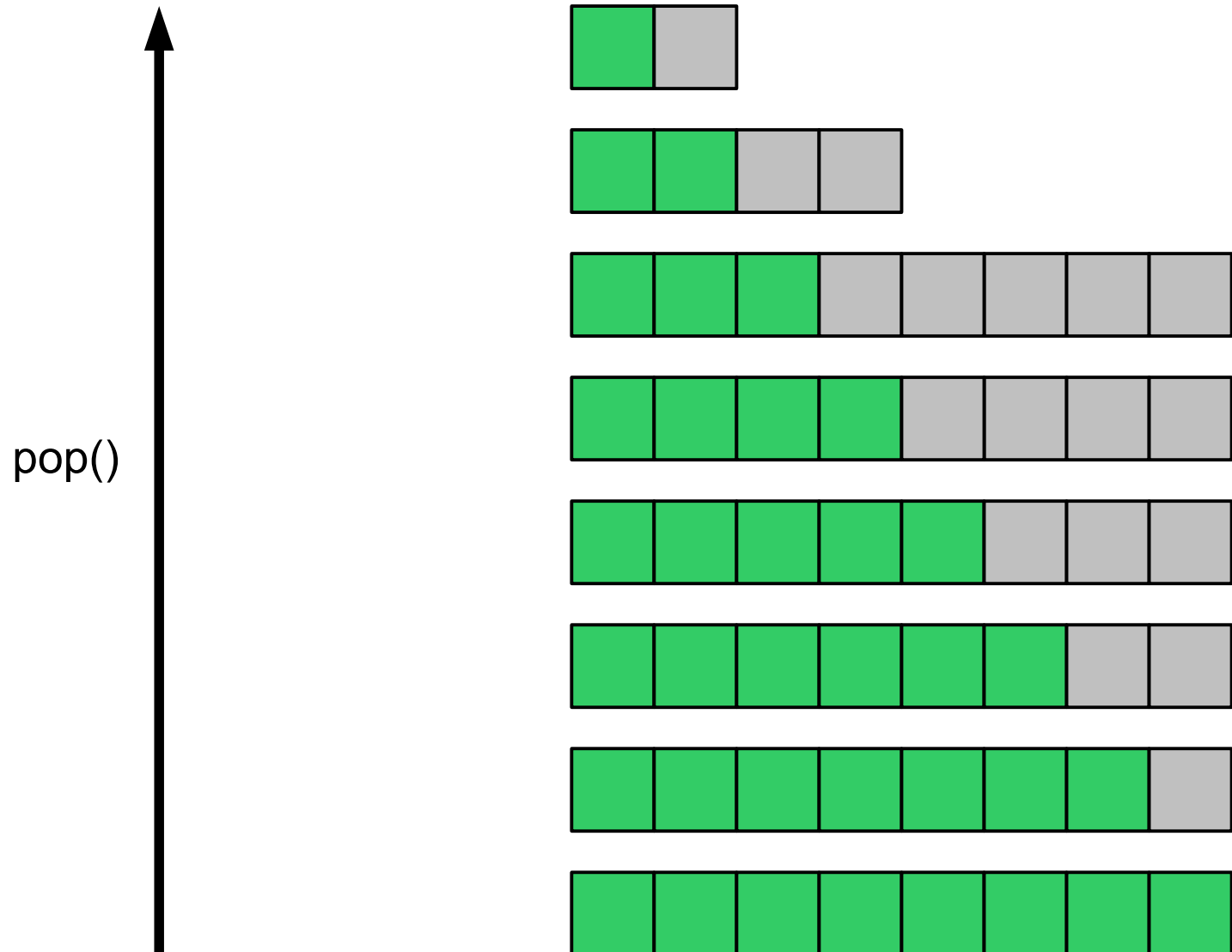
# Metodo del raddoppiamento/dimezzamento



# Analisi delle operazioni push() e pop()



# Analisi delle operazioni push() e pop()





# Analisi delle operazioni push() e pop()

- Nel caso peggiore, entrambe sono  $O(n)$
- Nel caso migliore, entrambe sono  $O(1)$
- Partendo dallo stack vuoto, quanto costano  $n$  operazioni push() consecutive?
  - $O(n)$
- Partendo da uno stack con  $n$  elementi, quanto costano  $n$  operazioni pop() consecutive?
  - $O(n)$
- Partendo da uno stack con numero *arbitrario* di elementi, quanto costano  $k$  operazioni (scelte *arbitrariamente* tra push() e pop()) consecutive?
  - Analisi ammortizzata

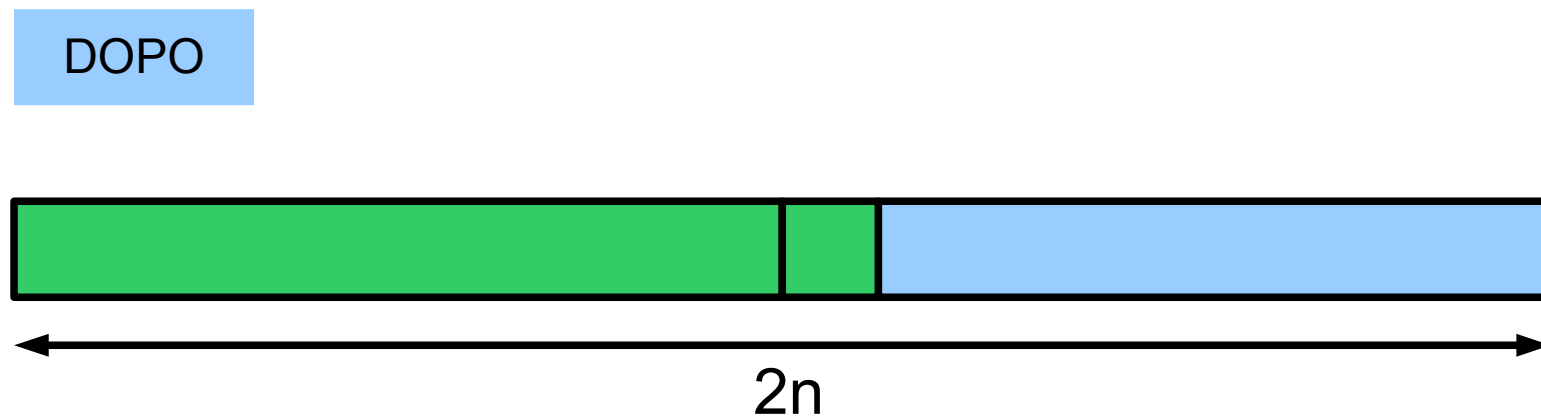
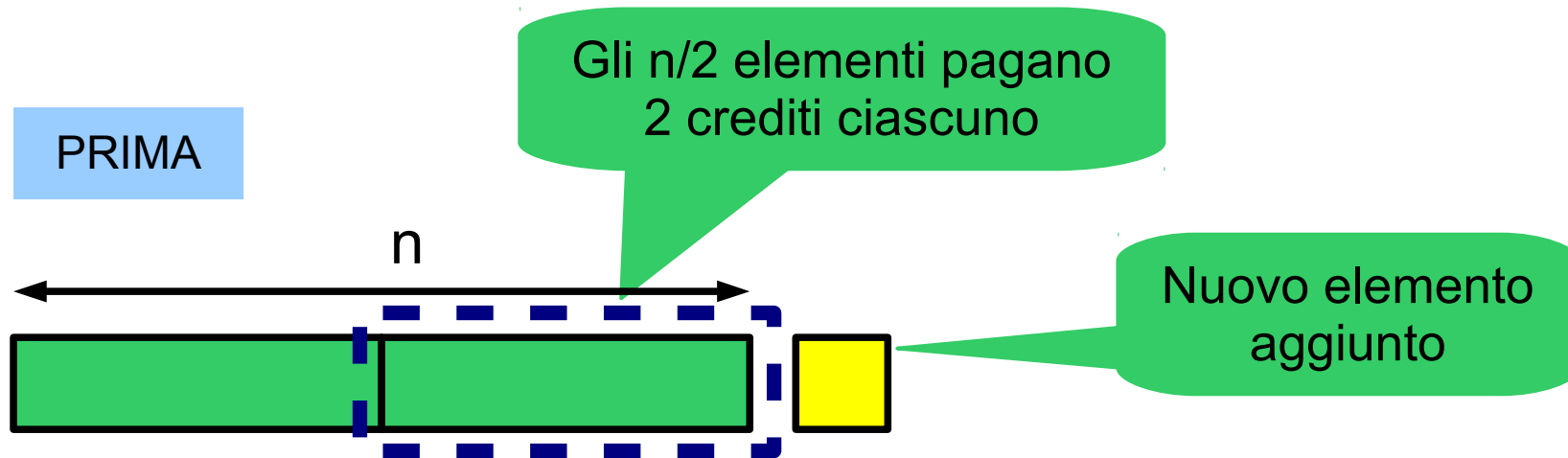
# Analisi ammortizzata: il metodo dei crediti

- Associamo a ciascun elemento della struttura dati un numero di **crediti**
  - Un credito può essere utilizzato per eseguire  $O(1)$  operazioni elementari
- Quando creo un elemento la prima volta, “pago” un certo numero di crediti
- Userò quei crediti per pagare ulteriori operazioni su quell'elemento, in futuro

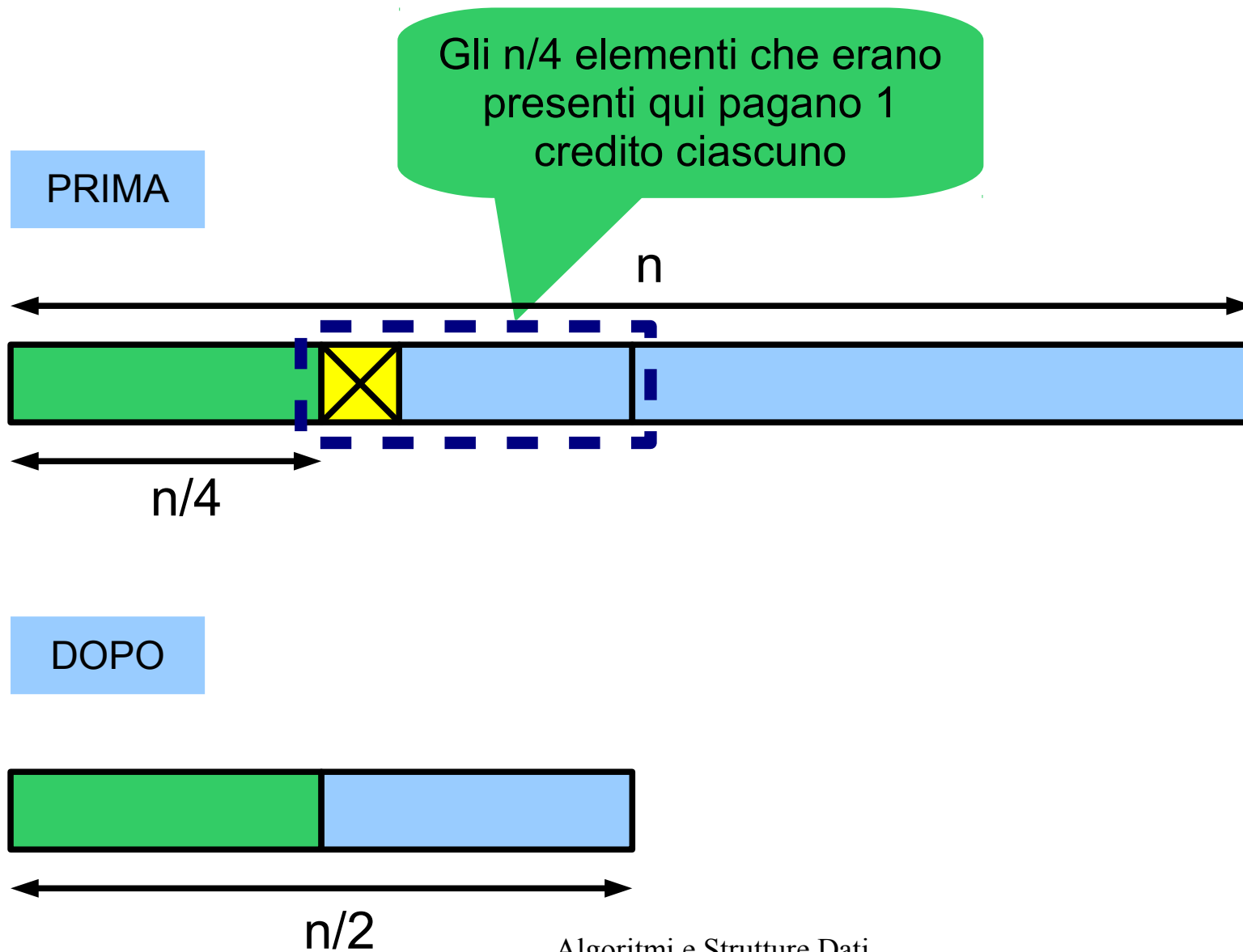
# Analisi ammortizzata delle operazioni push() e pop()

- L'inserimento di un elemento nello stack deposita **3 crediti** sulla cella dell'array che viene usata
- Quando devo **raddoppiare**
  - Sottraggo **2 crediti** dalle celle nella seconda metà dell'array (prima del raddoppio);
  - Uso questi crediti per “pagare” l'operazione di copia dei valori dall'array originale a quello “raddoppiato”
- Quando devo **dimezzare**
  - Sottraggo **1 credito** dalle celle nel secondo quarto dell'array (prima del dimezzamento)
  - Uso questi crediti per “pagare” l'operazione di copia

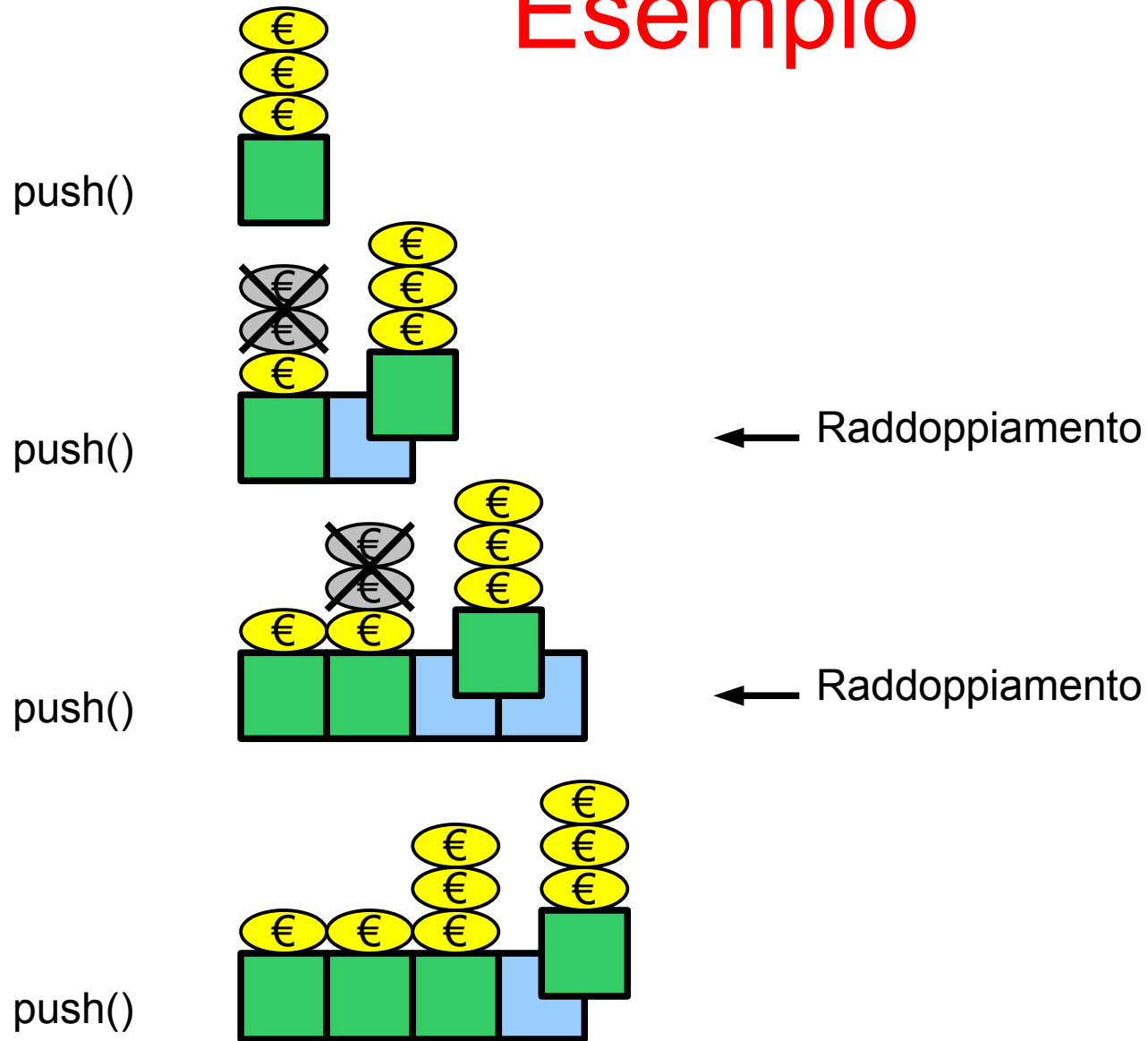
# Analisi ammortizzata



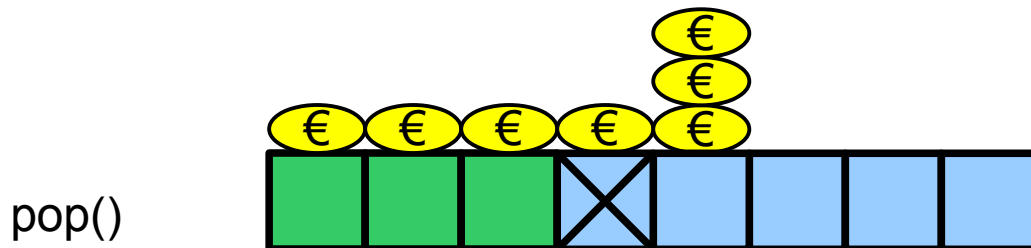
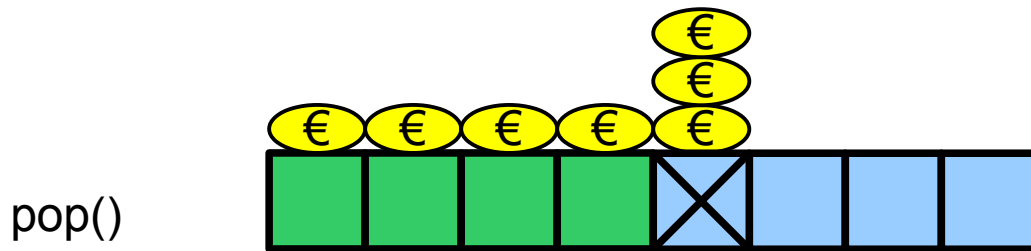
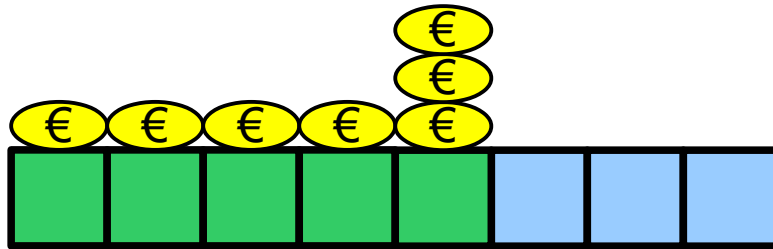
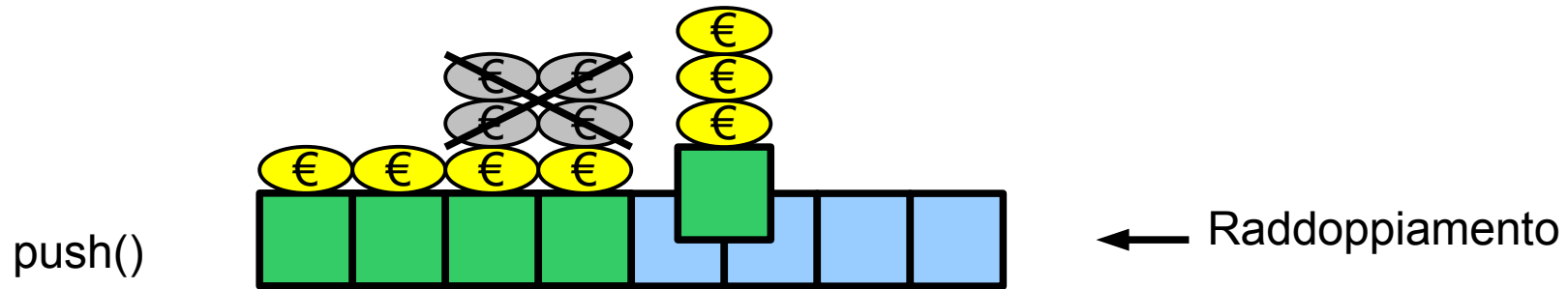
# Analisi ammortizzata



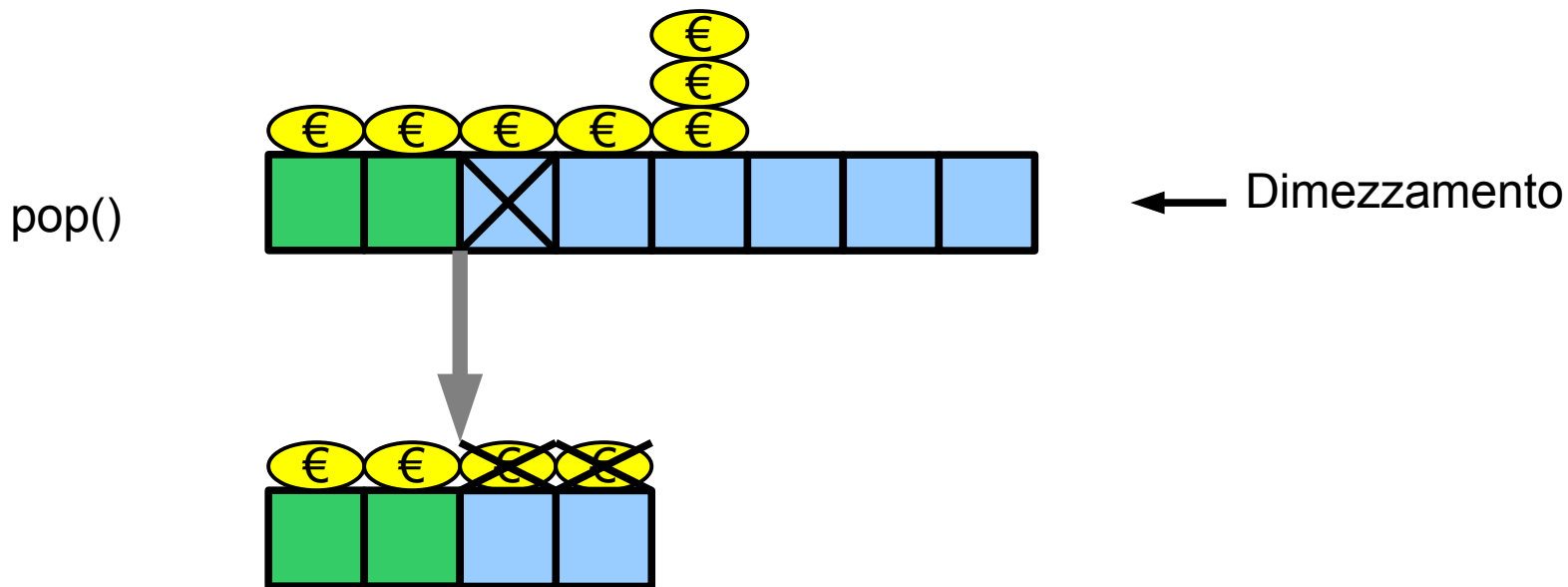
# Esempio



# Esempio (cont.)



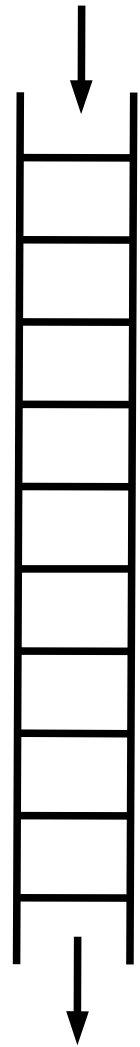
# Esempio (cont.)





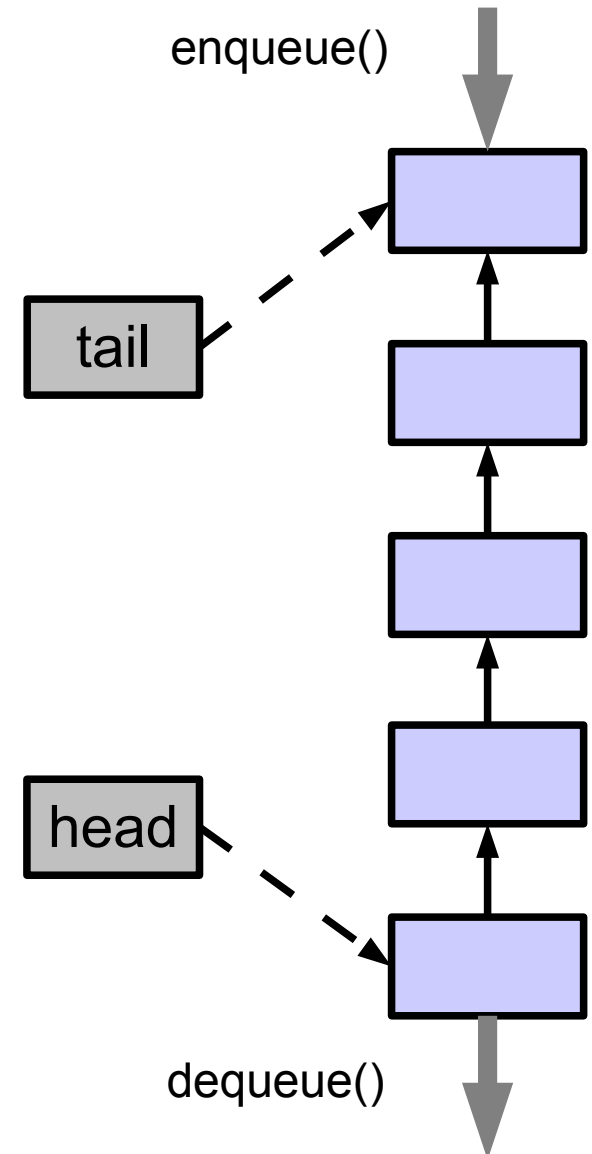
# Coda (Queue)

- Insieme dinamico in cui l'elemento rimosso dall'operazione di cancellazione è quello che da più tempo è presente
  - politica “first in, first out” (FIFO)
- Operazioni previste (tutte  $O(1)$ )
  - void enqueue(Item)
  - Item dequeue()
  - Item first()
  - boolean isEmpty()

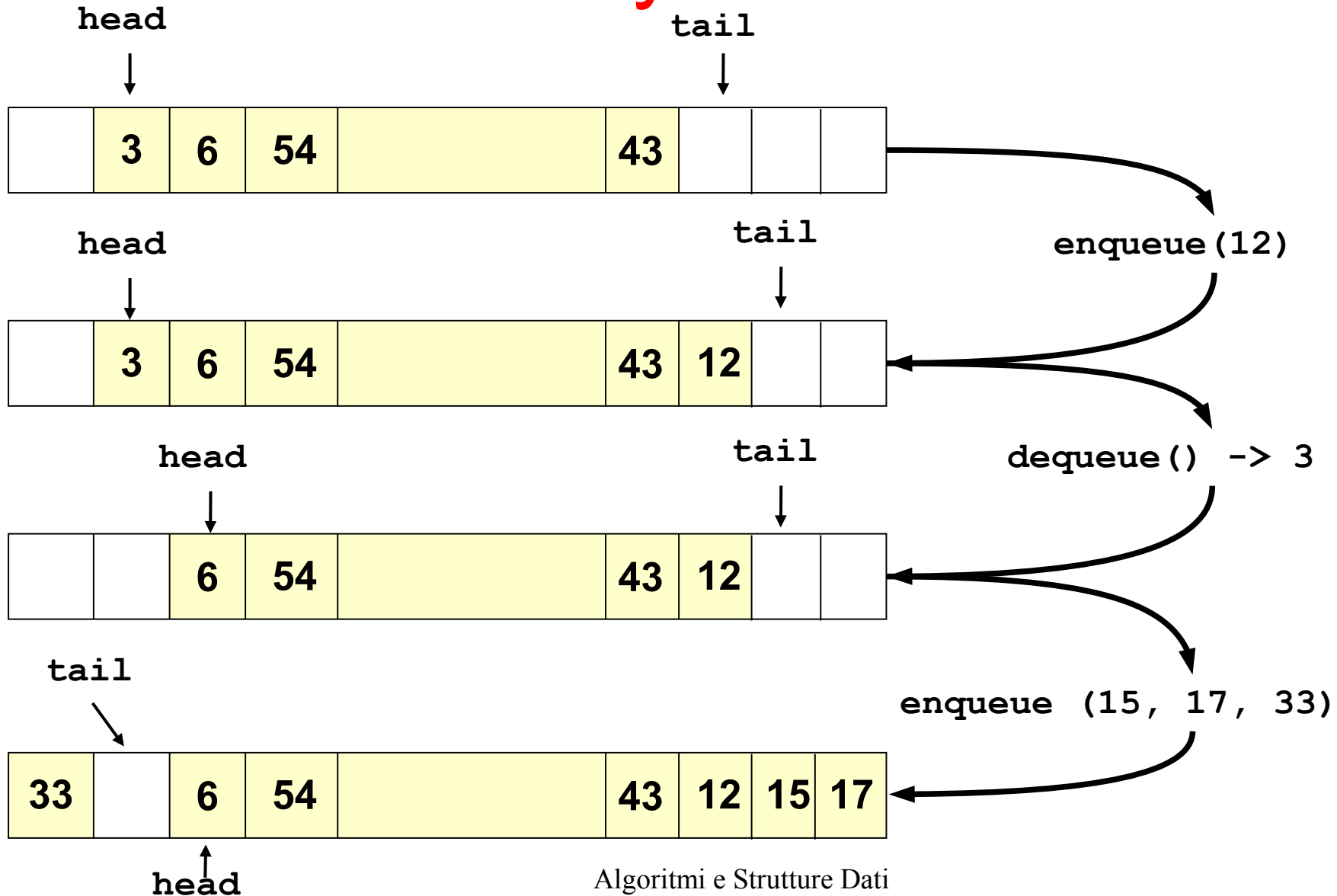


# Coda

- Possibili utilizzi
  - Nei sistemi operativi, i processi in attesa di utilizzare una risorsa vengono gestiti tramite una coda
- Possibili implementazioni
  - Tramite **liste puntate semplici**
    - puntatore *head* (inizio della coda), per estrazione
    - puntatore *tail* (fine della coda), per inserimento
    - Esempio: classe `CodaCollegata` in `asdlab.libreria.StruttureElem`
  - Tramite **array circolari**
    - dimensione limitata, overhead più basso



# Queue: Implementazione tramite array circolari



# Queue: Implementazione tramite array circolari (Java)

```
public class Queue {  
  
    private Object[] buffer = new int[MAX_SIZE];  
    private int head;        // "Dequeuing" index  
    private int tail;       // "Enqueuing" index  
    private int size;       // Number of elements  
  
    public Queue() { head = tail = size = 0; }  
    public boolean isEmpty() { return size == 0; }  
  
    public Object head() {  
        if (size == 0) throw new Exception("Empty");  
        else return buffer[head];  
    }  
}
```

Costo:  $O(1)$

# Queue: Implementazione tramite array circolari (Java)

Costo:  $O(1)$

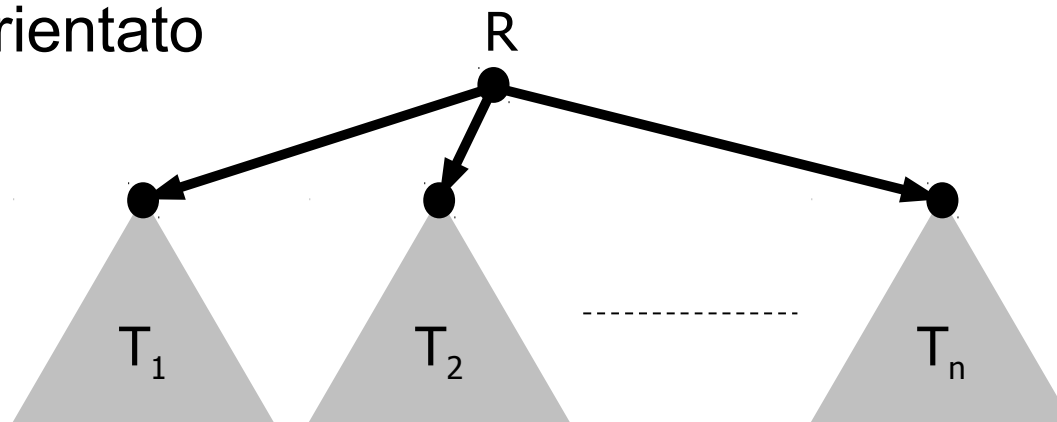
```
public void enqueue(Object o) {  
    if (size == buffer.length)  
        throw new Exception("Full");  
    buffer[tail] = o;  
    tail = (tail+1) % buffer.length;  
    size++;  
}
```

Costo:  $O(1)$

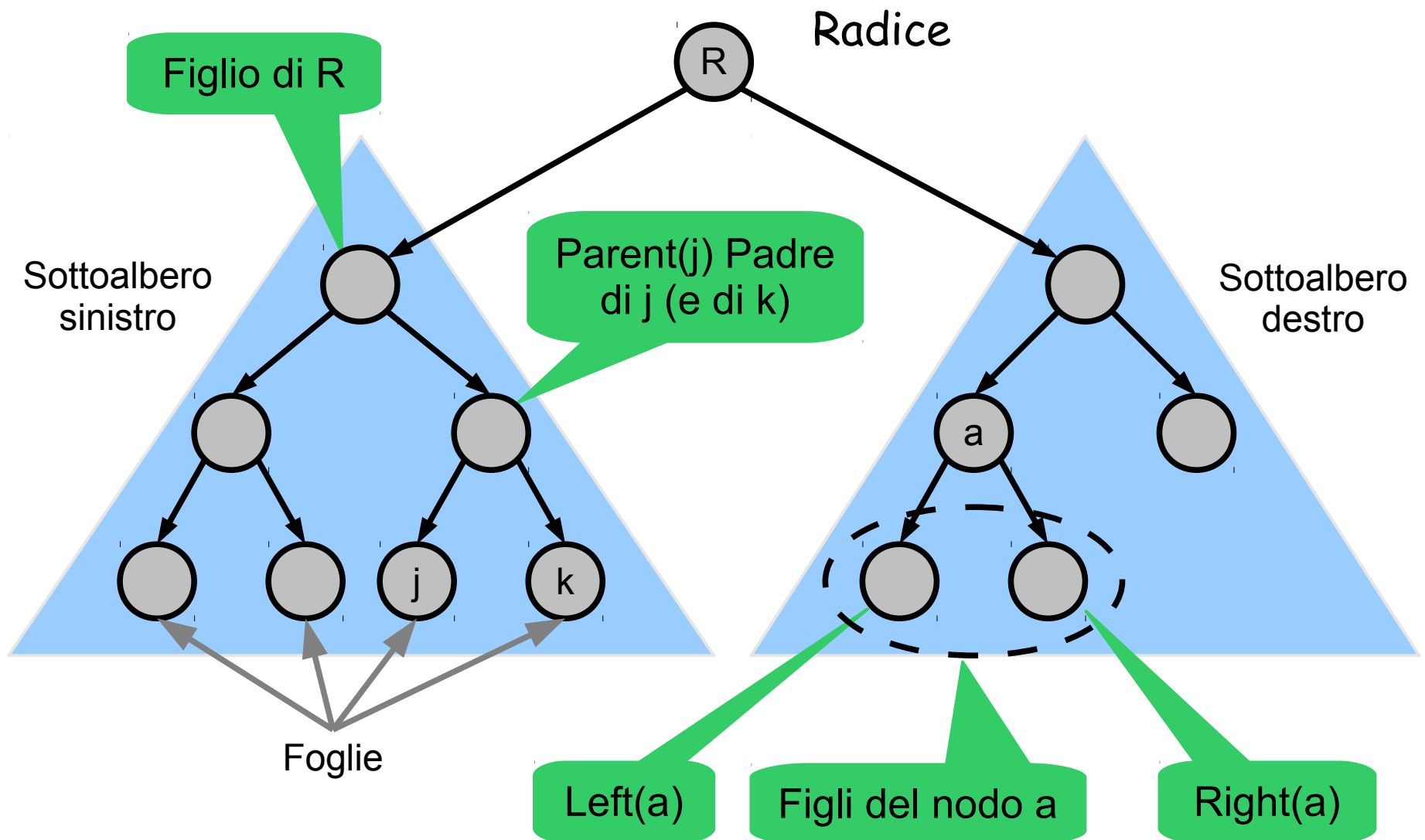
```
public Object dequeue() {  
    if (size == 0) throw new Exception("Empty");  
    Object res = buffer[head];  
    head = (head+1) % buffer.length;  
    size--;  
    return res;  
}
```

# Alberi radicati

- Albero: definizione informale
  - È un insieme dinamico i cui elementi hanno relazioni di tipo gerarchico
- Albero: definizione ricorsiva
  - Insieme vuoto di nodi, oppure
  - Una radice  $R$  e zero o più alberi disgiunti (detti *sottoalberi*), con la radice  $R$  collegata alla radice di ogni sottoalbero da un arco orientato

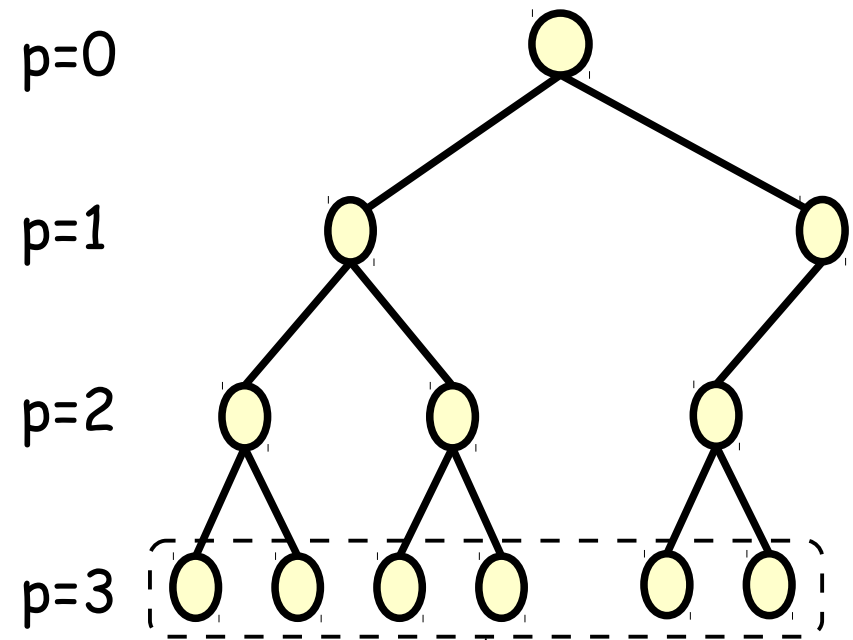


# Alberi binari radicati



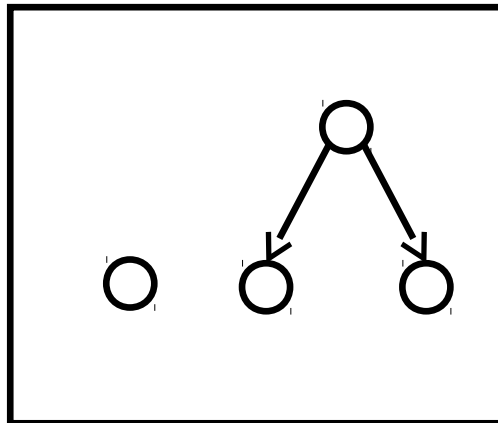
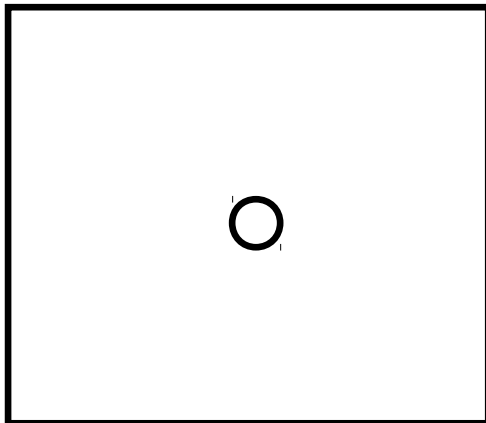
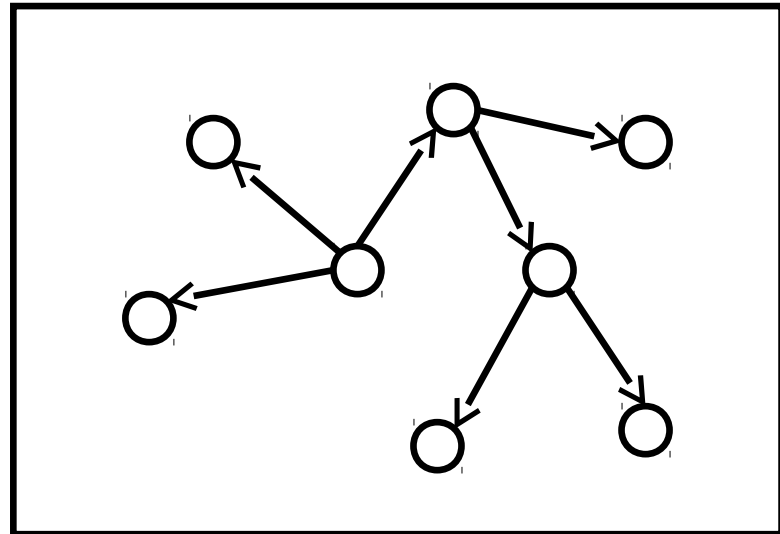
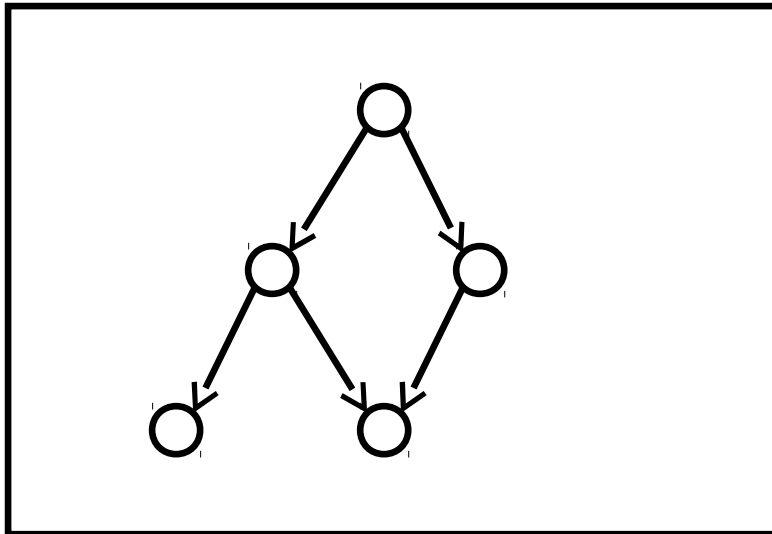
# Alberi: definizioni

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





# Alberi?



# Operazioni sugli alberi

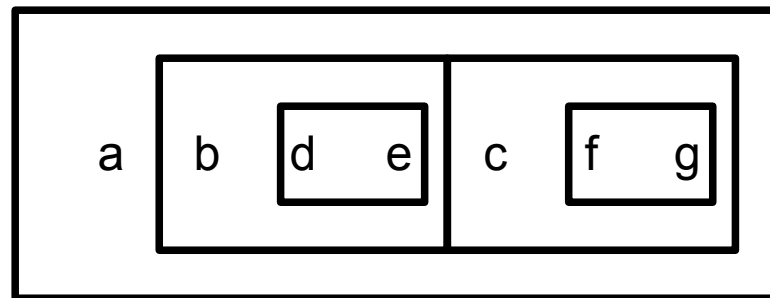
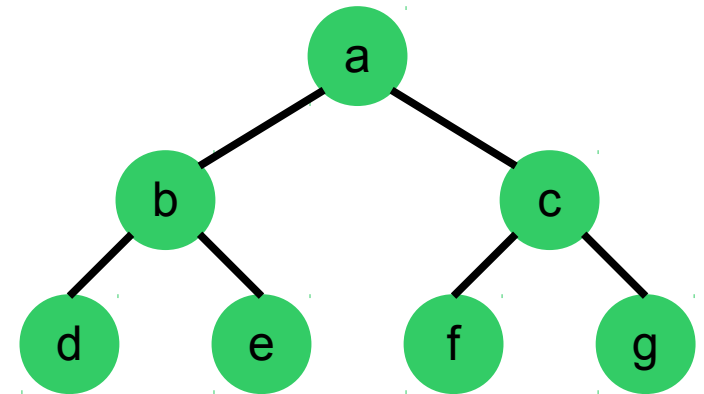
- `numNodi()` → intero
- `grado(nodo v)` → intero (*numero di figli*)
- `padre(nodo v)` → nodo
- `figli(nodo v)` → <nodo, nodo, ... nodo>
- `aggiungiNodo(nodo u)` → nodo
  - Aggiunge un nuovo nodo *v* come figlio di *u*; restituisce *v*
- `aggiungiSottoalbero(Albero a, nodo u)`
  - Aggiunge il sottoalbero *a* in modo che la radice di *a* diventi figlia di *u*
- `rimuoviSottoalbero(nodo v)` → albero
  - stacca e restituisce il sottoalbero radicato in *v*

# Algoritmi di visita degli alberi

- Visita (o attraversamento) di un albero:
  - Algoritmo per “visitare” tutti i nodi di un albero
- **In profondità** (*depth-first search*, DFS)
  - Vengono visitati i rami, uno dopo l'altro
  - Esistono tre varianti (pre-ordine, in-ordine, post-ordine)
- **In ampiezza** (*breadth-first search*, BFS)
  - A livelli, partendo dalla radice

# Visita alberi binari in profondità pre-ordine

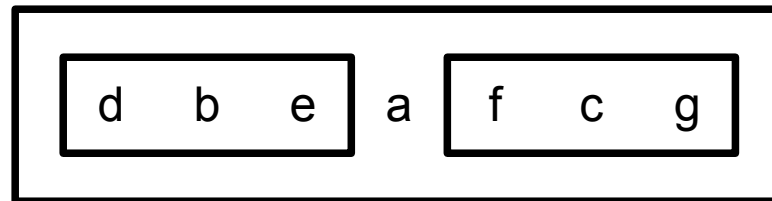
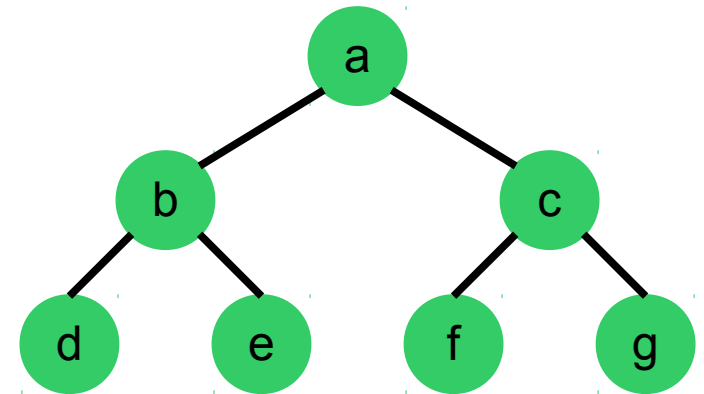
```
Algoritmo visita-preordine(T)
if ( T ≠ nil ) {
  Visita T
  visita-preordine(T.left())
  visita-preordine(T.right())
}
```



- Nota: gli algoritmi di visita si possono facilmente generalizzare al caso di albero non binario

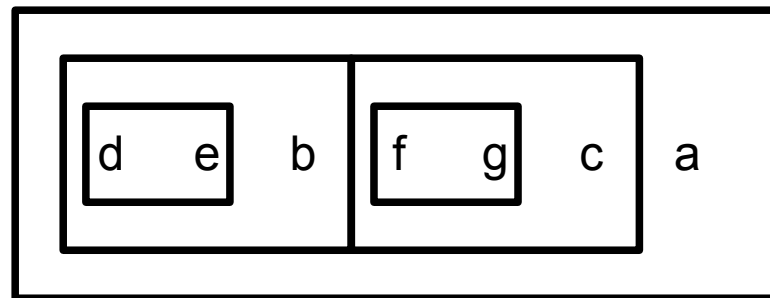
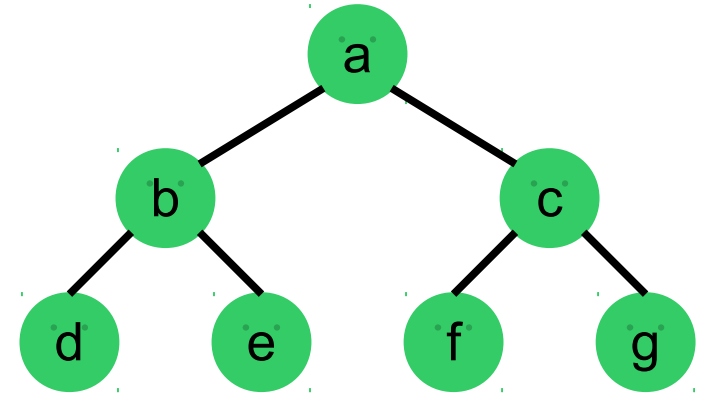
# Visita alberi binari in profondità in-ordine (simmetrica)

```
Algoritmo visita-inordine(T)
if ( T ≠ null ) {
    visita-inordine(T.left())
    Visita T
    visita-inordine(T.right())
}
```



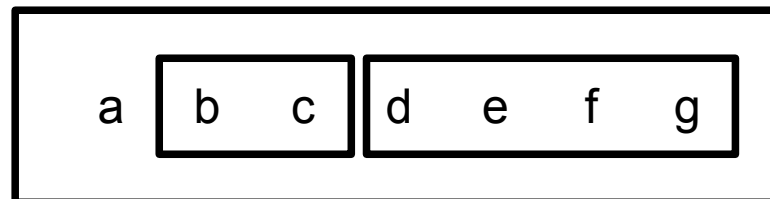
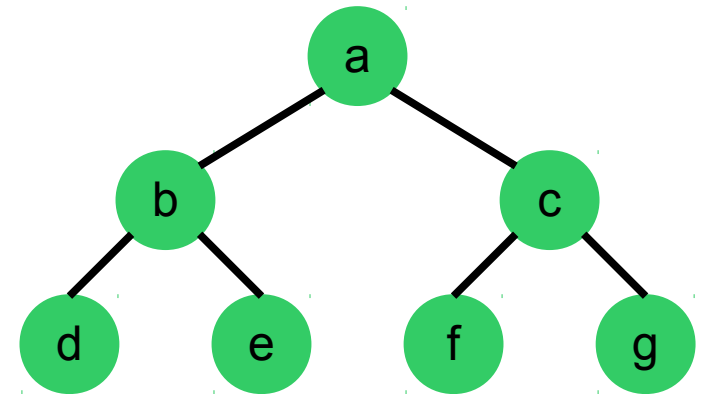
# Visita alberi binari in profondità post-ordine

```
Algoritmo visita-postordine(T)
if ( T ≠ null ) {
    visita-postordine(T.left())
    visita-postordine(T.right())
    Visita T
}
```

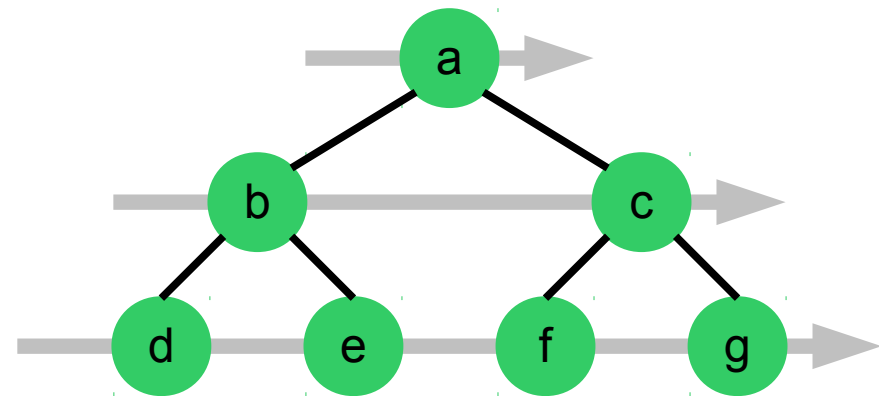
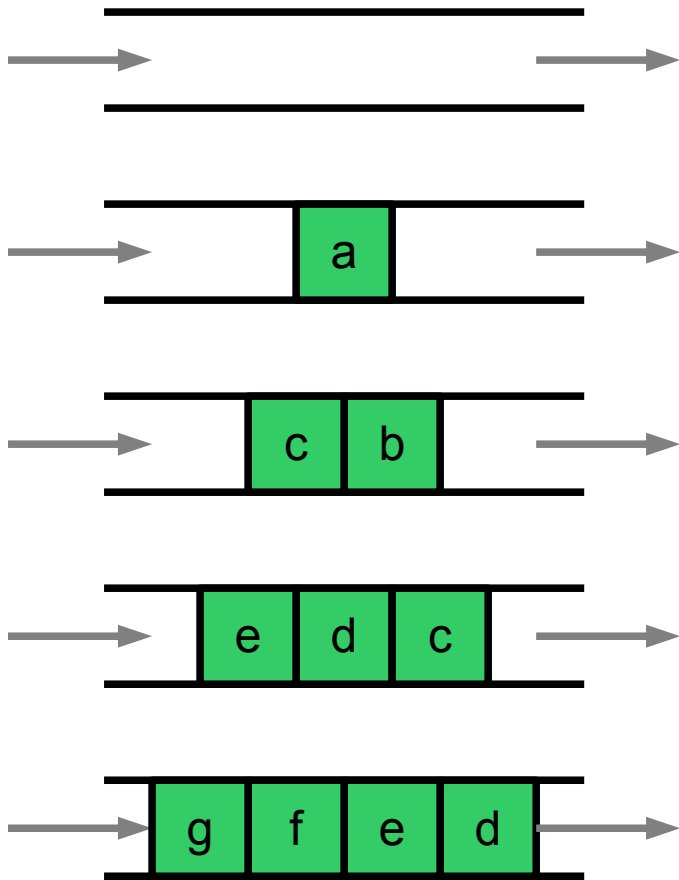


# Visita alberi binari: in ampiezza

```
Algoritmo visita-ampiezza(T)
q = new Queue()
q.insert(T)
while ( !q.empty() ) {
  p := q.dequeue()
  visita p
  if ( p.left() != null ) then
    q.enqueue(p.left());
  endif
  if ( p.right() != null) then
    q.enqueue(p.right())
  endif
}
```

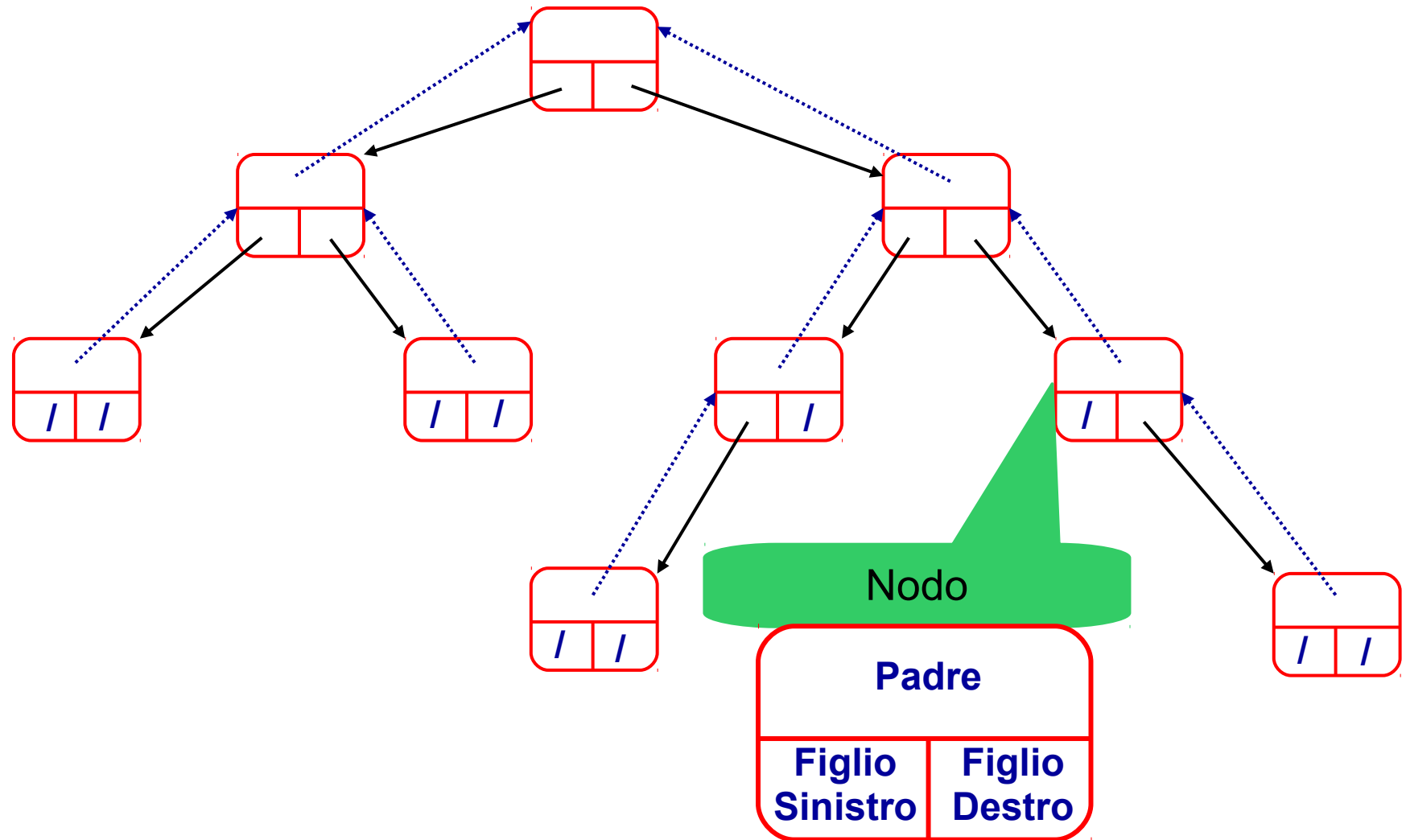


# Visita alberi binari: in ampiezza





# Implementazione di alberi binari (strutture con puntatori)



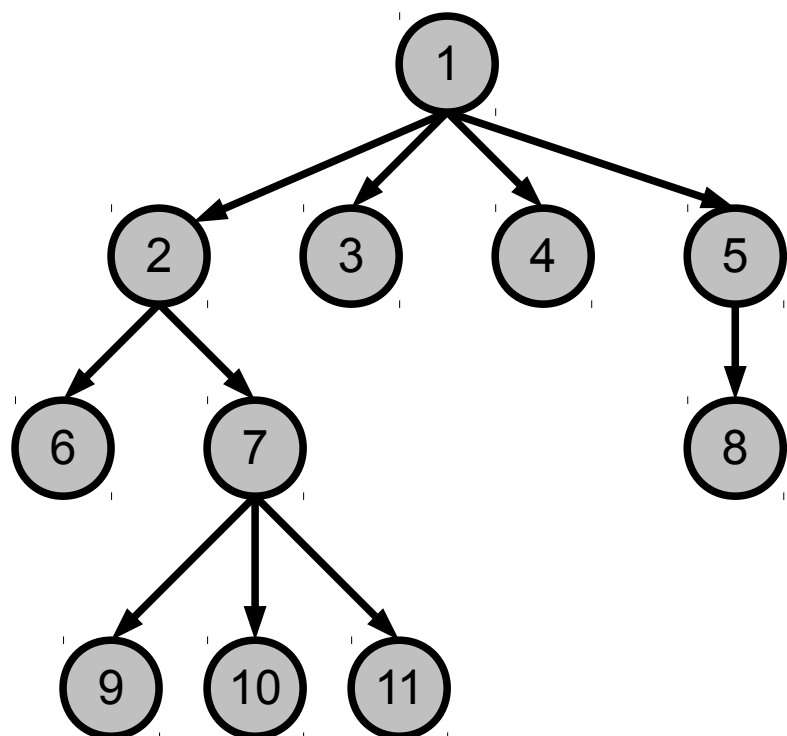
# Alcune operazioni sugli alberi binari

## conteggio dei nodi

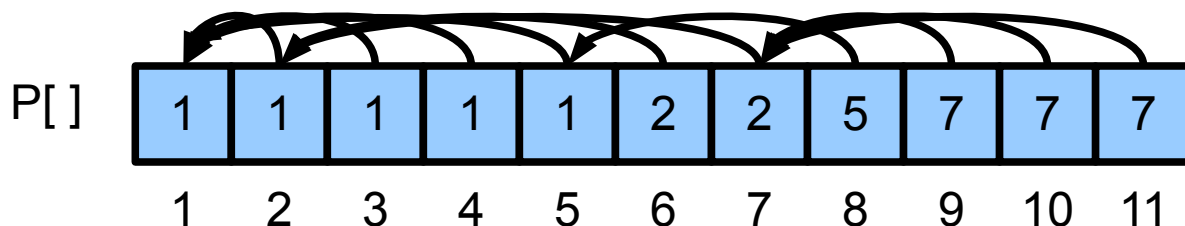
```
Algoritmo numNodi(T)
  if (T == null) then
    return 0;
  else
    return 1 + numNodi(T.left()) + numNodi(T.right());
  endif
```

- Quale è il costo computazionale di numNodi() ?
- È possibile modificare le strutture dati affinché numNodi() richieda tempo  $O(1)$ ?
  - Quale è l'impatto di tale modifica sulla complessità computazionale delle altre operazioni che è possibile effettuare su un albero binario (es, aggiungere una foglia, fondere o separare alberi, ecc.)?

# Implementazione di alberi generali: vettore padri



- I nodi sono numerati da **1** a **n**
- L'albero è rappresentato da un vettore **P**, ove **P[i]** indica il padre del nodo **i**
- Per convenzione, si indica come padre della radice la radice stessa



# Implementazione d-alberi: vettore posizionale

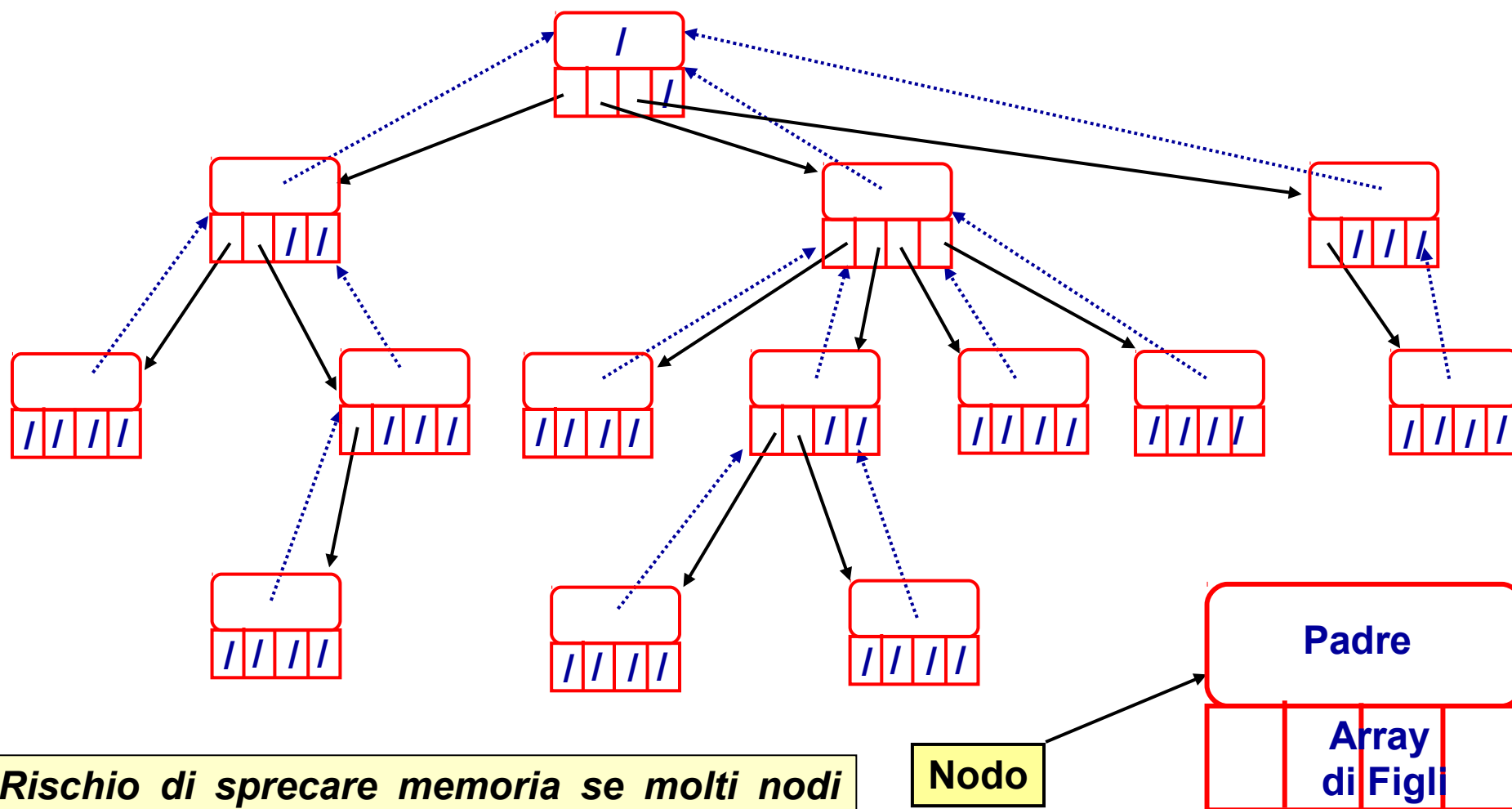
Sia  $T$  un albero  $d$ -ario di  $n$  nodi, numerati da 0 a  $n-1$ .

Un **vettore posizionale** è un array  $A$  di dimensione  $n$  (con le celle numerate da 0 a  $n-1$ ) tale che  $A[i]$  contiene l'informazione associata al nodo  $i$ .

I figli del nodo  $i$ :  $i$  nodi  $d \cdot i + k$  con  $k=1, \dots, d$ .

Dato il nodo  $i$ , con  $i \neq 0$ ,  $\text{Parent}(i) = \lfloor (i-1)/d \rfloor$

# Implementazione di alberi generali: array di figli



*Rischio di sprecare memoria se molti nodi hanno grado minore del grado massimo  $k$ .*

# Implementazione di alberi generali: nodo fratello

