

Strutture dati elementari

Davide Rossi
Dip. di Scienze dell'Informazione
Università di Bologna

rossi@cs.unibo.it
<http://www.cs.unibo.it/~rossi>

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

Modifications Copyright © 2009, Moreno Marzolla, INFN Padova

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

- Dato
 - In un linguaggio di programmazione, un dato è un valore che una variabile può assumere
- Tipo di dato astratto
 - Un modello matematico, dato da una collezione di valori e un insieme di operazioni ammesse su questi valori
- Tipi di dato primitivi
 - Forniti direttamente dal linguaggio
 - Esempi: int (+, -, *, /, %), boolean (!, &&, ||)

Tipi di dati

- “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 addirittura parametrico (*templates* in C++, o *generics* in Java)
- Una struttura dati è composta da:
 - un modo sistematico di organizzare i dati
 - un insieme di operatori che permettono di 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)

Insiemi dinamici

- Struttura dati “generale”: **insieme dinamico**
 - Può crescere, contrarsi, cambiare contenuto
 - Operazioni base: inserimento, cancellazione, ricerca
 - Il tipo di insieme (= struttura) dipende dalle operazioni
- Elementi
 - Elemento: oggetto “puntato” da un riferimento/puntatore
 - Composto da:
 - campo **chiave** di identificazione (visibile dall'utente)
 - campo **dati** (visibile dall'utente)
 - campi che fanno riferimento ad altri elementi dell'insieme (normalmente non visibili dall'utente)

Esempio (specifica)

```
public interface Dizionario {  
    /**  
     * Aggiunge al dizionario la coppia (e,k).  
     */  
    public void insert(Object e, Comparable k);  
  
    /**  
     * Rimuove dal dizionario l'elemento con chiave k.  
     * In caso di duplicati, l'elemento cancellato  
     * è scelto arbitrariamente tra quelli con chiave k.  
     */  
    public void delete(Comparable k);  
  
    /**  
     * Restituisce l'elemento e con chiave k.  
     * In caso di duplicati, l'elemento restituito  
     * è scelto arbitrariamente tra quelli con chiave k.  
     */  
    public Object search(Comparable k);  
}
```

Insiemi dinamici

- Operazioni di interrogazione
 - Item search(Key k)
- Operazioni di modifica
 - void insert(Item x)
 - void delete(Item x)
- Ulteriori operazioni (non presenti nell'interfaccia)
 - Item successor(Item x)
 - Item predecessor(Item x)
 - Item minimum()
 - Item maximum()

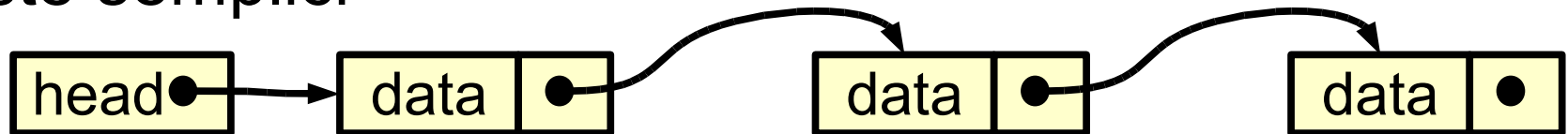
Linked List

- Liste puntate (Linked List)
 - Una sequenza di nodi, contenenti dati arbitrari e 1-2 reference (puntatori, link) all'elemento successivo e/o precedente.
- Tipo di accesso

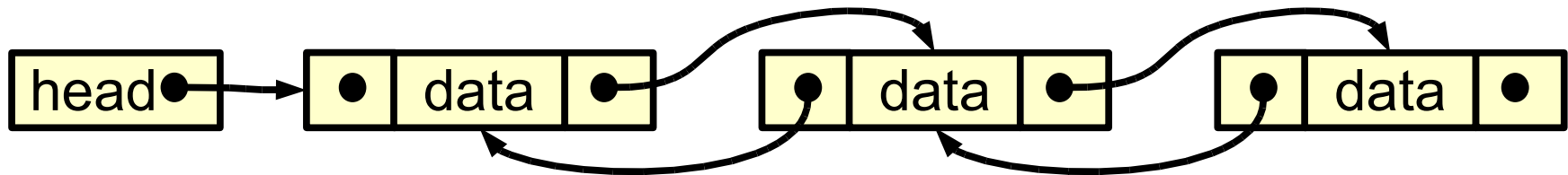
	Liste	Array
Random	$O(n)$	$O(1)$
Inserimento	$O(1)$	$O(n)$
Cancellazione	$O(1)$	$O(n)$

Alcune possibili implementazioni

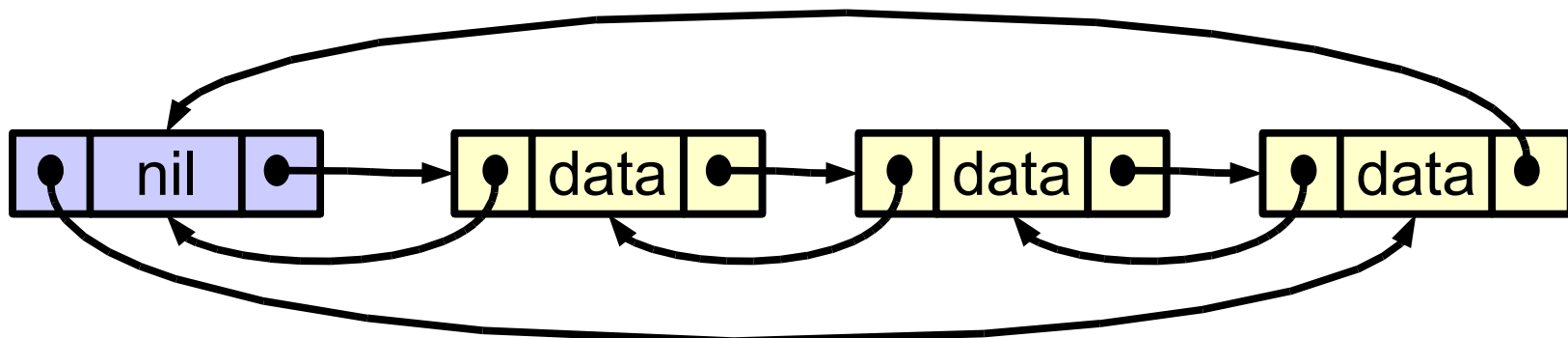
- Liste semplici



- Liste doppiamente collegate



- Liste con sentinella



Esempio

asdlab.libreria.StrutturaElem

```
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

asdlab.libreria.StruttureElem

```
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

asdlab.libreria.StrutturaElem

```
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

Costo: $O(1)$

Esempio

asdlab.libreria.StruttureElem

```
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

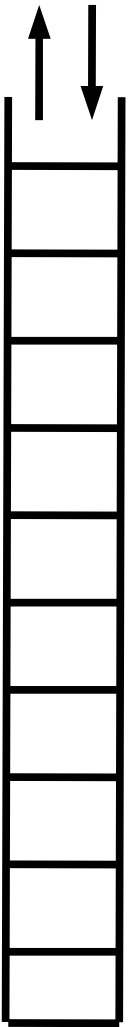
asdlab.libreria.StrutturaElem

```
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 (tutte $O(1)$)
 - 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: Postscript, Java bytecode
- Possibili implementazioni
 - Tramite liste puntate doppie
 - puntatore all'elemento top, per estrazione/inserimento
 - Tramite array
 - dimensione limitata, overhead più basso

Esempio Stack

Reverse Polish Notation (RPN)

- Espressioni aritmetiche in cui gli operatori seguono gli operandi; notazione usata ad es. nelle calcolatrici HP
- Definita dalla grammatica:
 $\langle \text{expr} \rangle ::= \langle \text{numeral} \rangle \mid \langle \text{expr} \rangle \langle \text{expr} \rangle \langle \text{operator} \rangle$

- Esempi:

- 7 3 + 5 x

- 7 3 5 x +

- Esempio di funzionamento

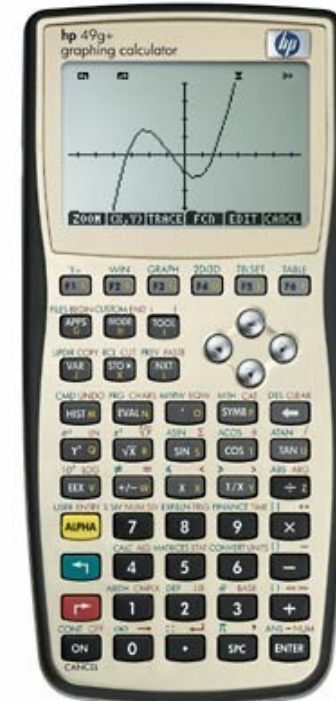
- push 7 7

- push 3 3 / 7

- push 5 5 / 3 / 7

- push x 15 / 7

- push + 22



Interfaccia Pila

```
package asdlab.libreria.StruttureElem;  
  
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

```
package asdlab.libreria.StruttureElem;

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;  
}
```



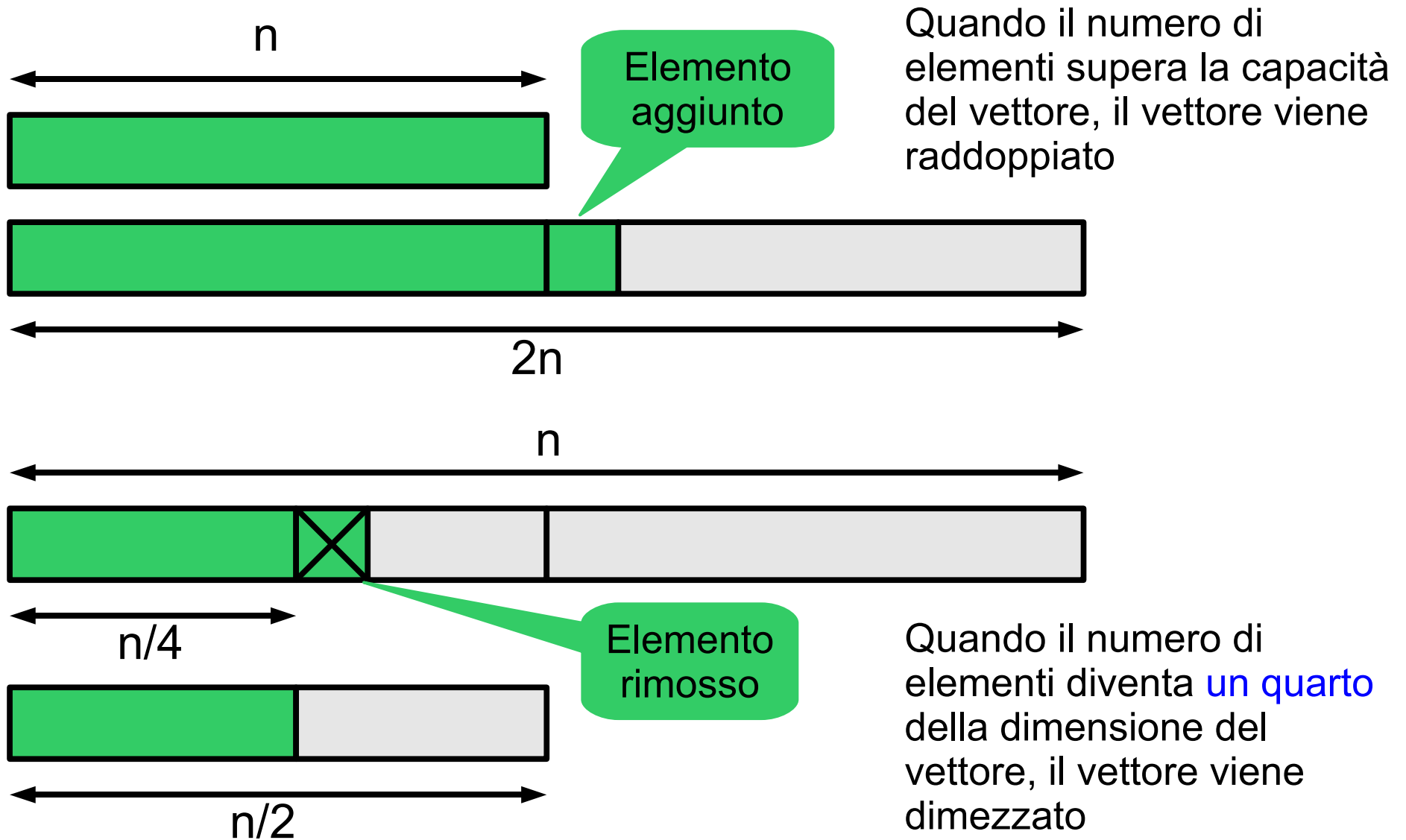
Costo:???

PilaArray: metodo pop()

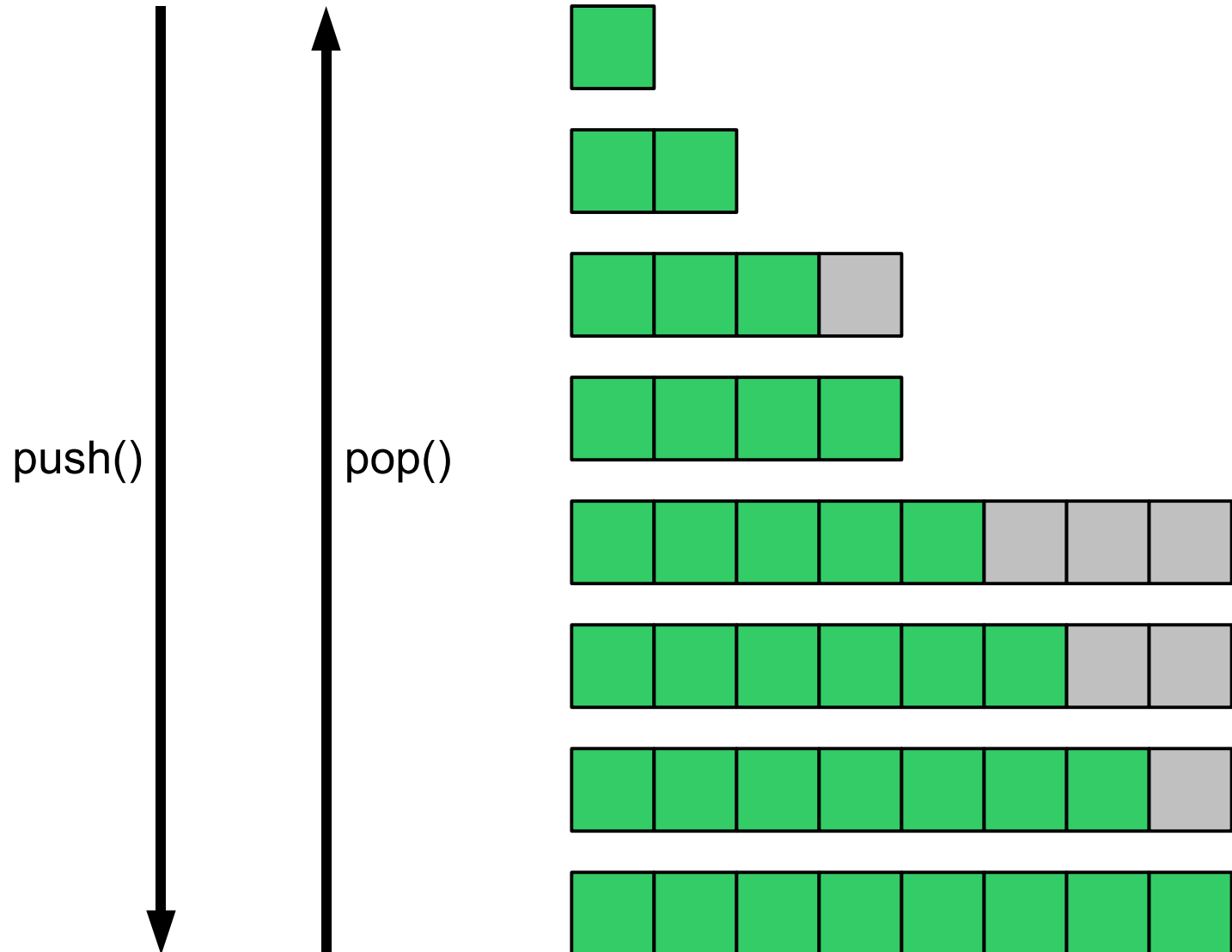
```
public Object pop() {
    if (this.isEmpty())
        throw new EccezioneStrutturaVuota("Pila vuota");
    n = n - 1;
    Object e = S[n];
    if (n > 1 && n == S.length / 4) {
        Object[] temp = new Object[S.length / 2];
        for (int i = 0; i < n; i++)
            temp[i] = S[i];
     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

Costo: ???

Metodo del raddoppiamento/dimezzamento



Analisi delle operazioni push() e pop()



Analisi delle operazioni push() e pop()

- Nel caso peggiore, entrambe sono $O(n)$
- Nel caso migliore, entrambe sono $O(1)$
- Partendo dallo stack vuoto, quanto costano n operazioni push() consecutive?
 - $1 + 2 + 4 + \dots + n/2^i = O(n)$
- Partendo da uno stack con n elementi, quanto costano n operazioni pop() consecutive?
 - $n/2 + n/4 + n/8 + \dots + 2 + 1 = O(n)$
- Partendo da uno stack con numero arbitrario di elementi, quanto costano k operazioni (arbitrarie) consecutive?
 - Analisi ammortizzata

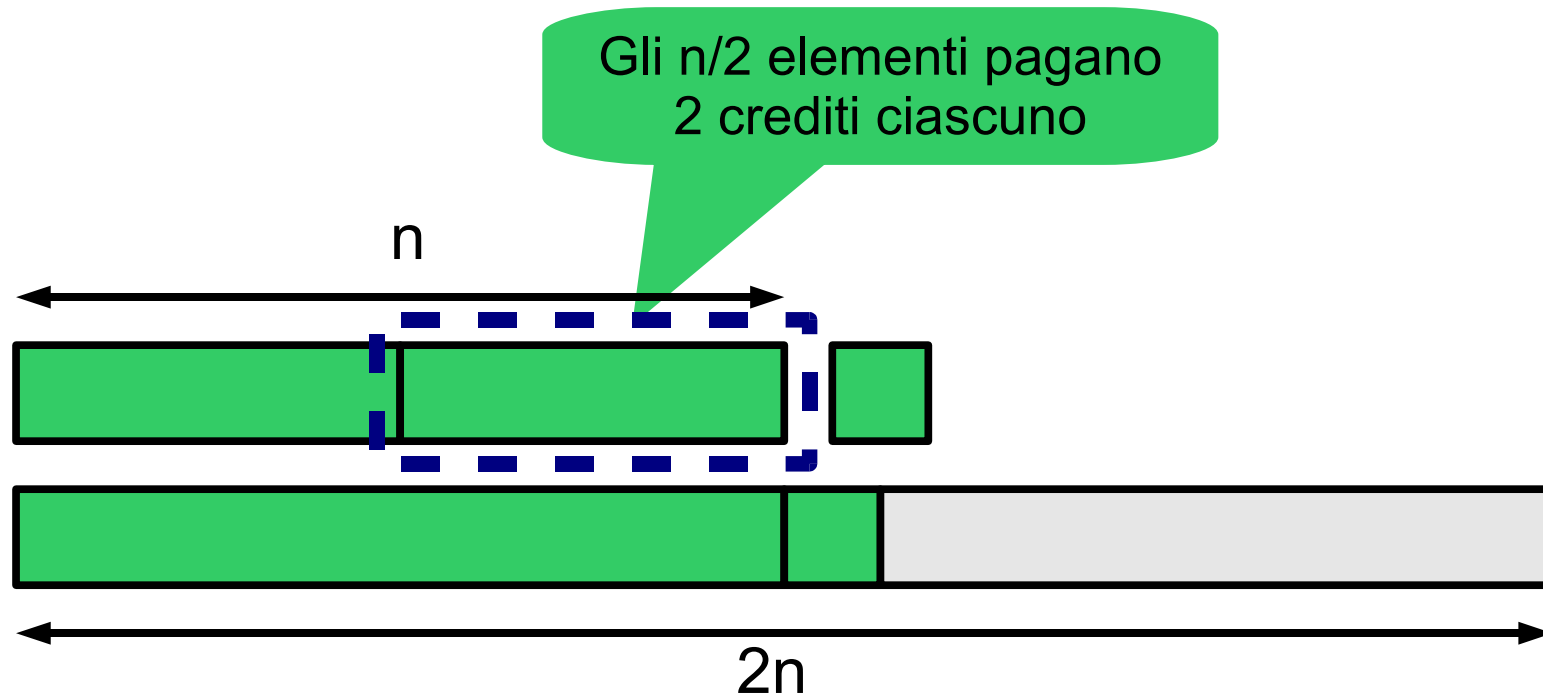
Analisi ammortizzata: il metodo dei crediti

- 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

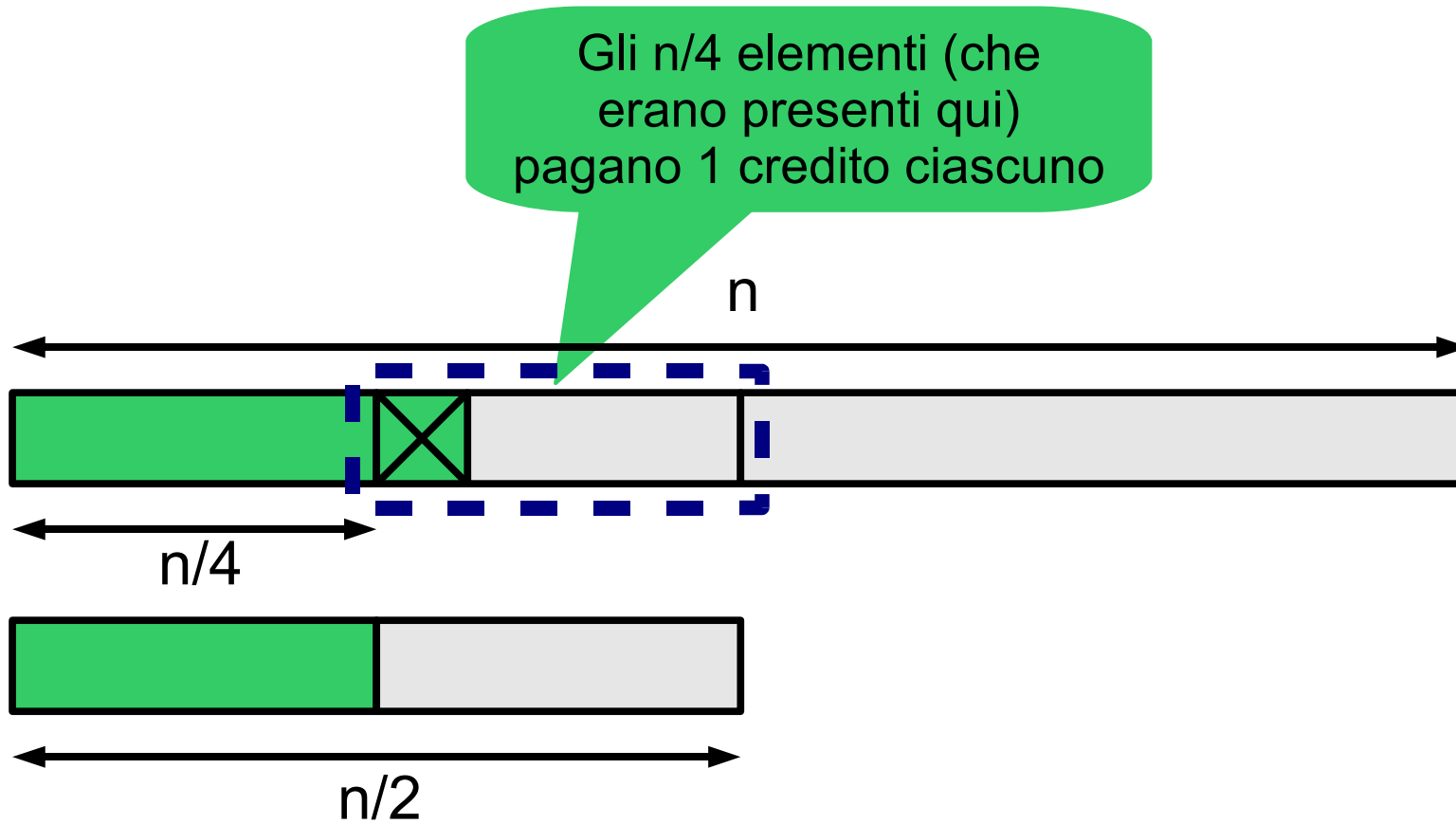
Analisi ammortizzata delle operazioni push() e pop()

- L'inserimento di un elemento nello stack deposita **3 crediti** sulla cella dell'array
- Quando devo raddoppiare
 - Sottraggo 2 crediti dalle celle nella seconda metà dell'array (prima del raddoppio);
 - Uso questi crediti per “pagare” la 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” la copia

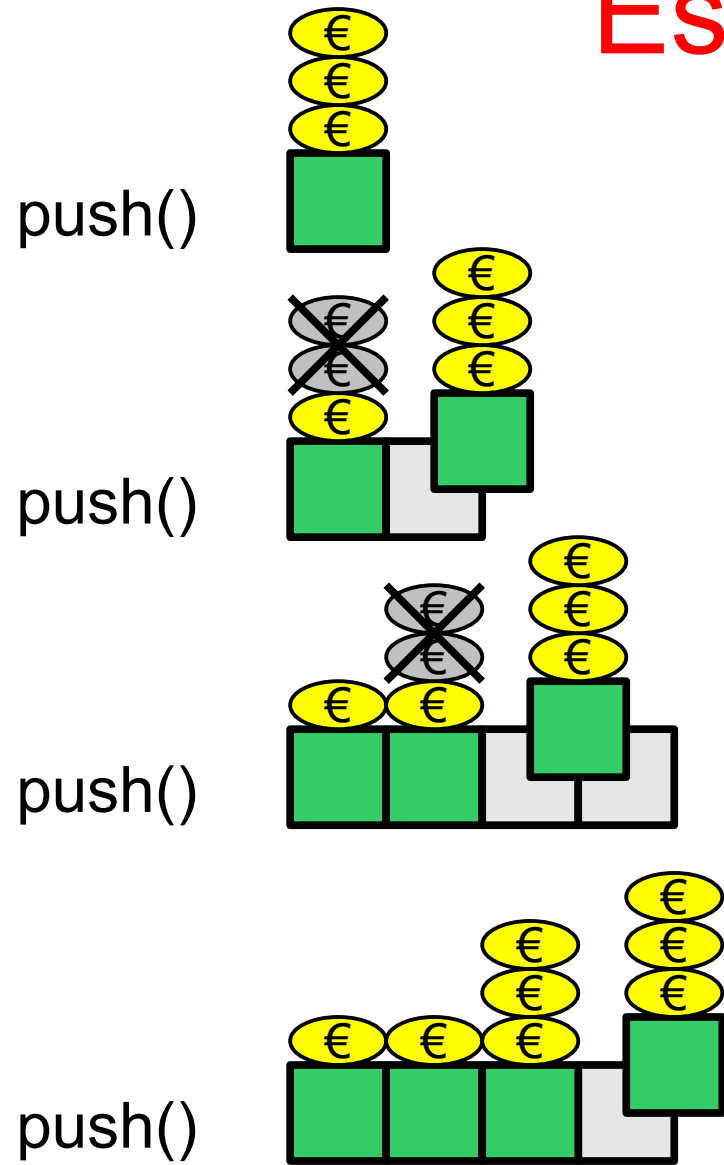
Analisi ammortizzata



Analisi ammortizzata

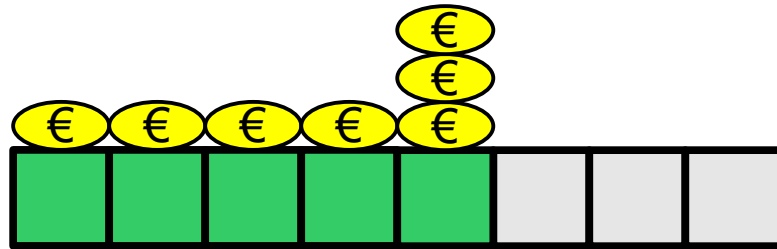
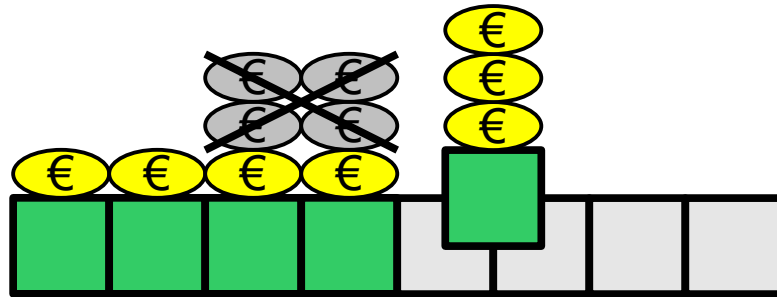


Esempio

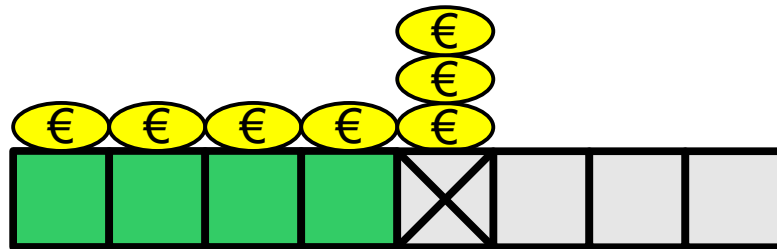


Esempio (cont.)

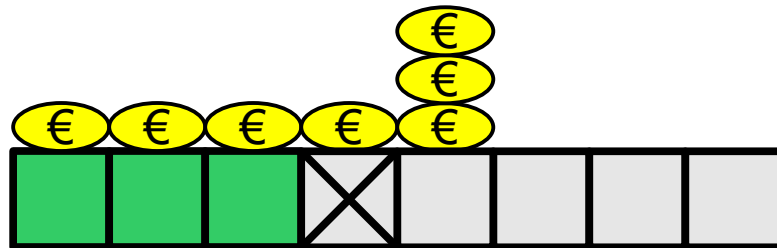
push()



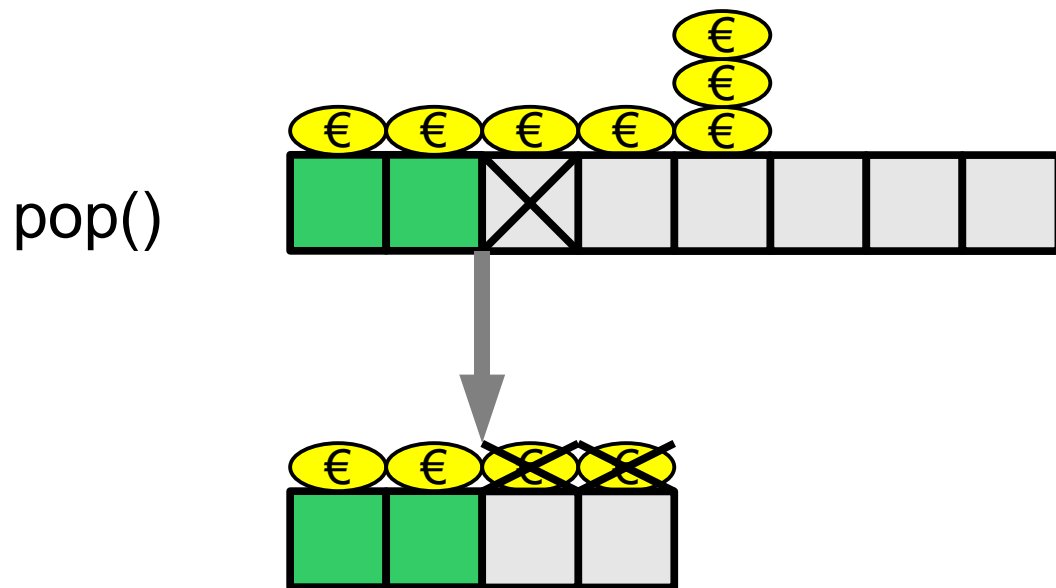
pop()



pop()

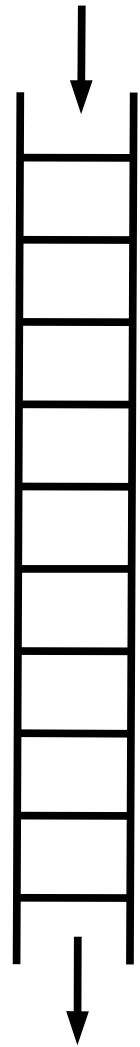


Esempio (cont.)



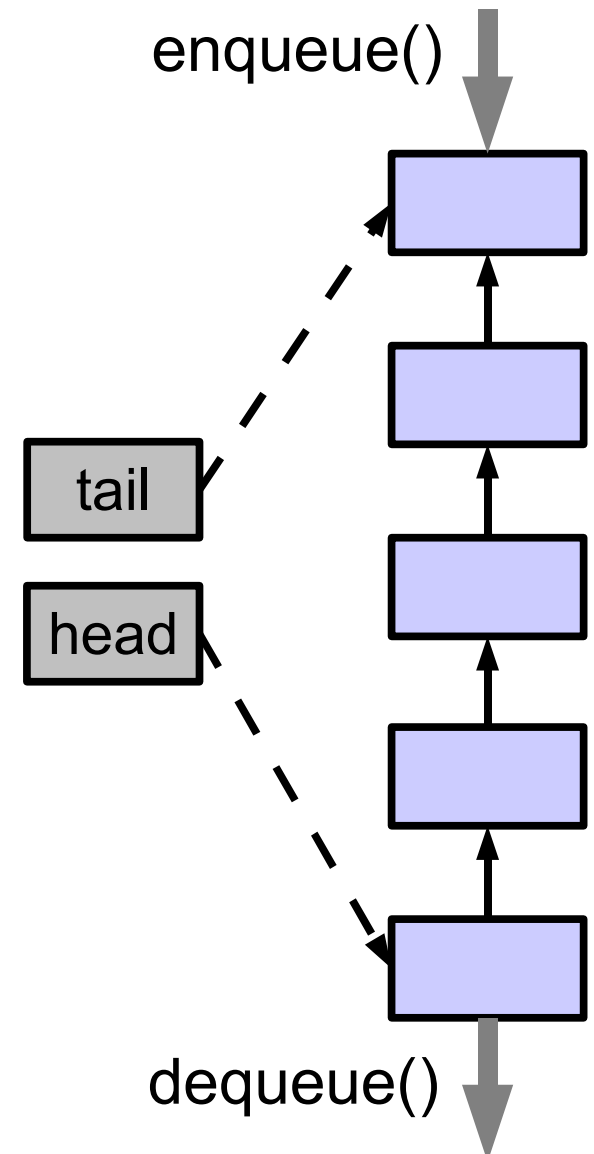
Coda (Queue)

- 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()

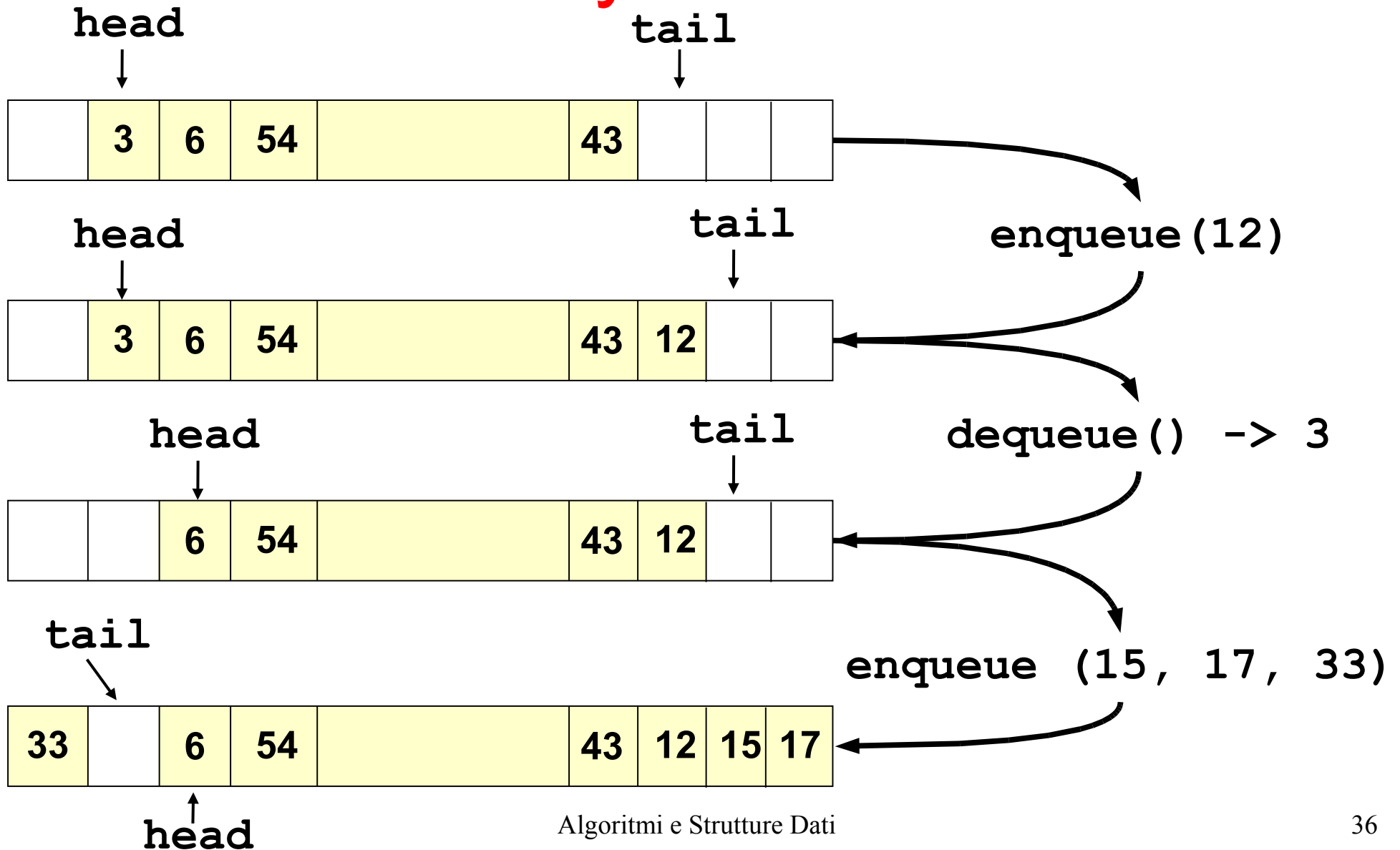


Coda

- 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: $O(1)$

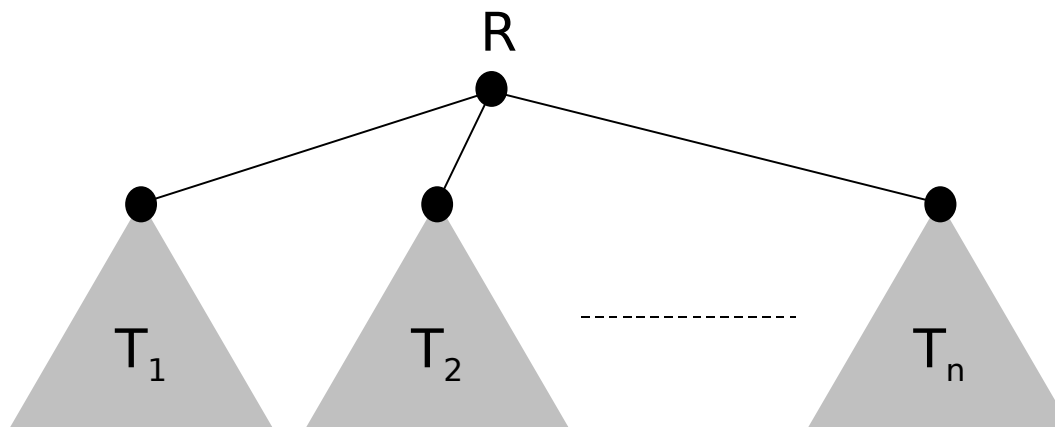
```
public void enqueue(Object o) {  
    if (size == buffer.length)  
        throw new Exception("Full");  
    buffer[tail] = o;  
    tail = (tail+1) % buffer.length;  
    size++;  
}
```

Costo: $O(1)$

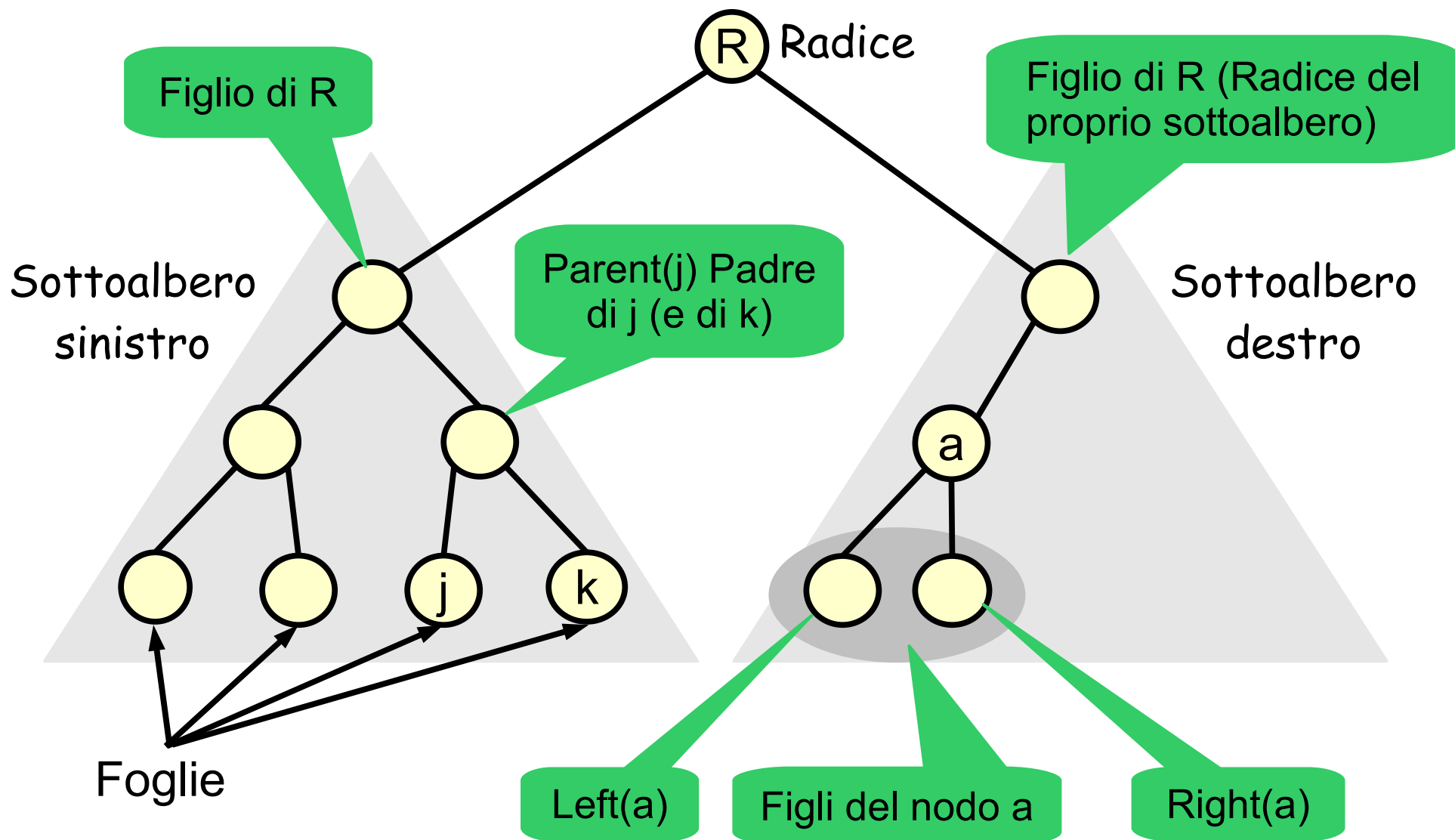
```
public Object dequeue() {  
    if (size == 0) throw new Exception("Empty");  
    Object res = buffer[head];  
    head = (head+1) % buffer.length;  
    size--;  
    return res;  
}
```

Alberi radicati

- Albero: definizione informale
 - E' un insieme dinamico i cui elementi hanno relazioni di tipo gerarchico
- Albero: definizione ricorsiva
 - Insieme vuoto di nodi, oppure
 - Una radice R e 0 o più alberi (detti sottoalberi), con la radice di ogni sottoalbero collegata a R da un arco orientato

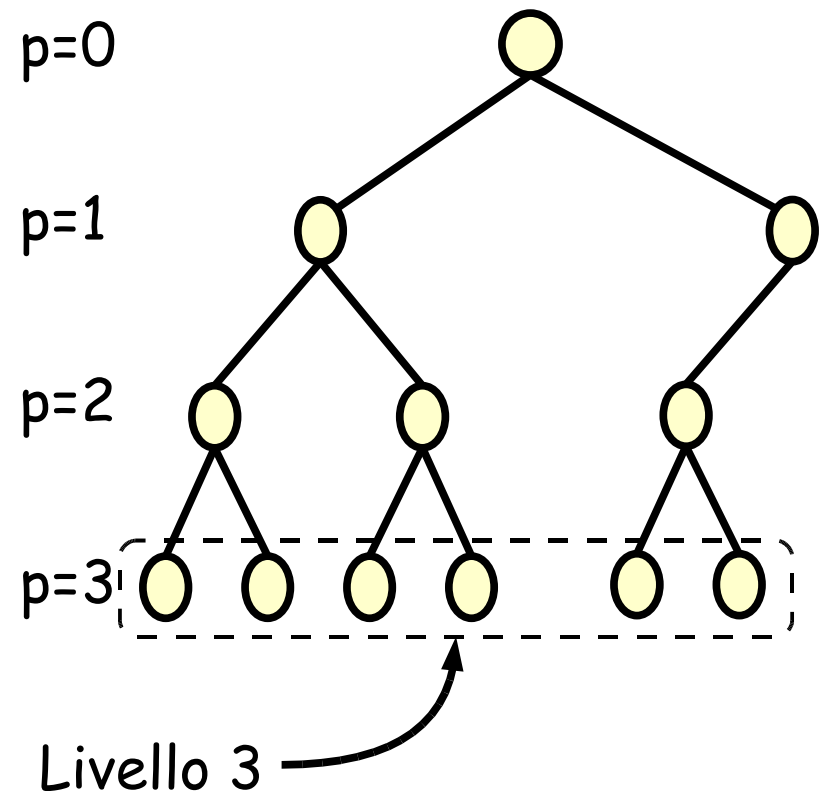


Alberi binari radicati



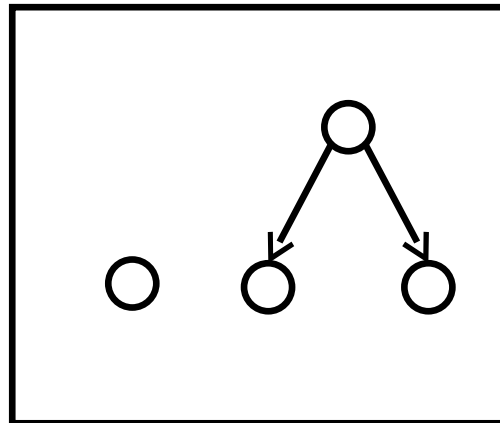
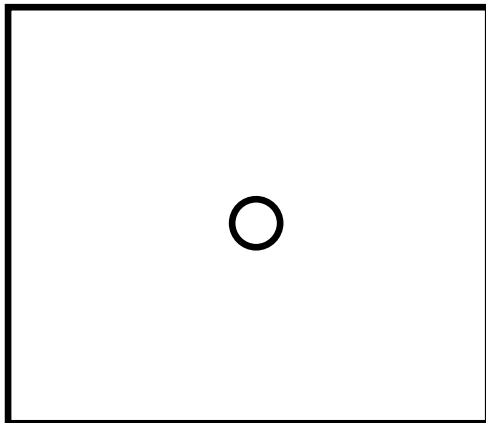
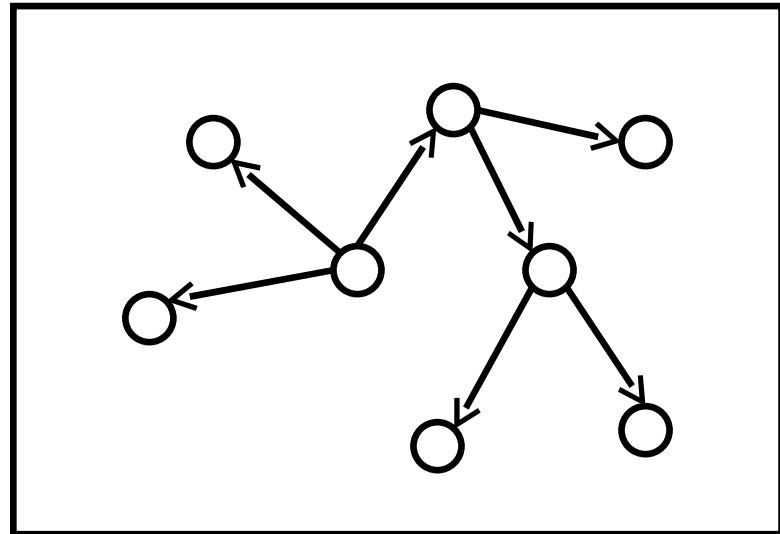
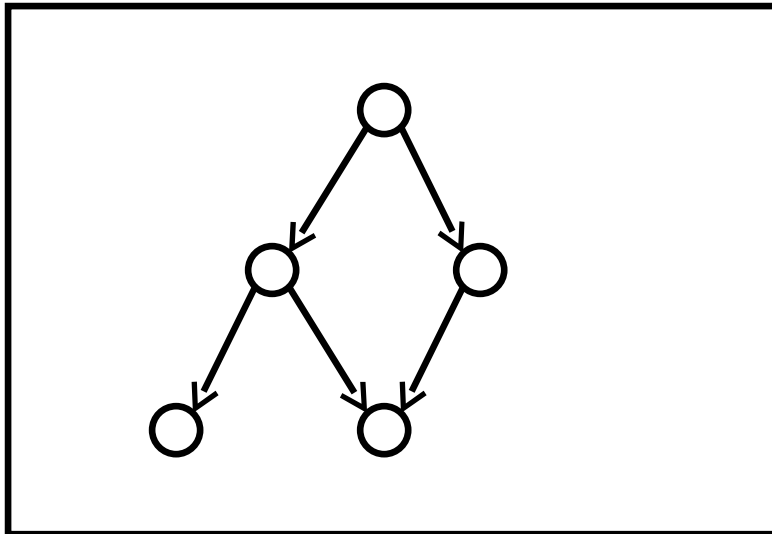
Alberi: definizioni

- 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



Altezza albero: 3

Alberi?

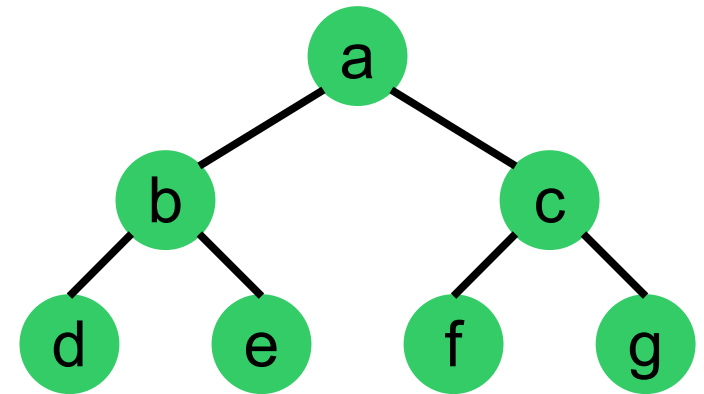


Algoritmi di visita degli alberi

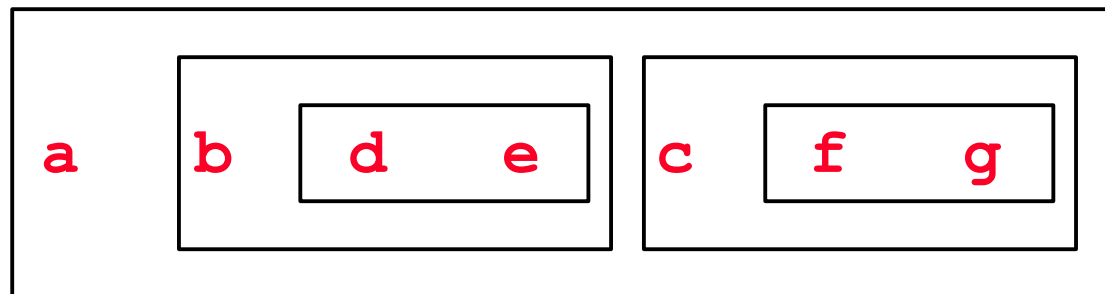
- Visita (o attraversamento) di un albero:
 - Algoritmo per “visitare” tutti i nodi di un albero
- **In profondità** (*depth-first search*, DFS)
 - Vengono visitati i rami, uno dopo l’altro
 - Esistono tre varianti (pre-ordine, in-ordine, post-ordine)
- **In ampiezza** (*breadth-first search*, BFS)
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

Visita alberi binari in profondità pre-ordine

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



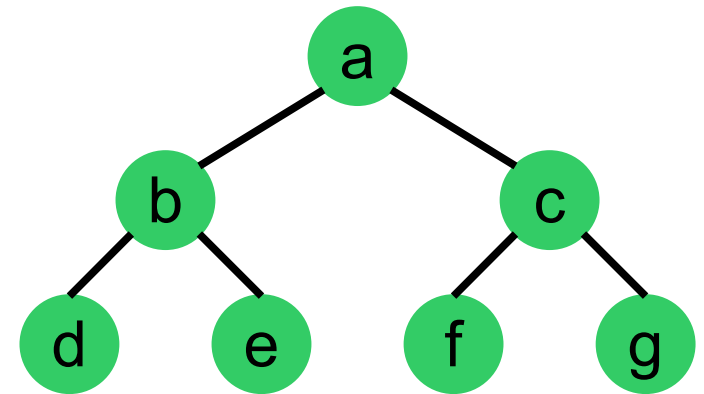
Sequenza:



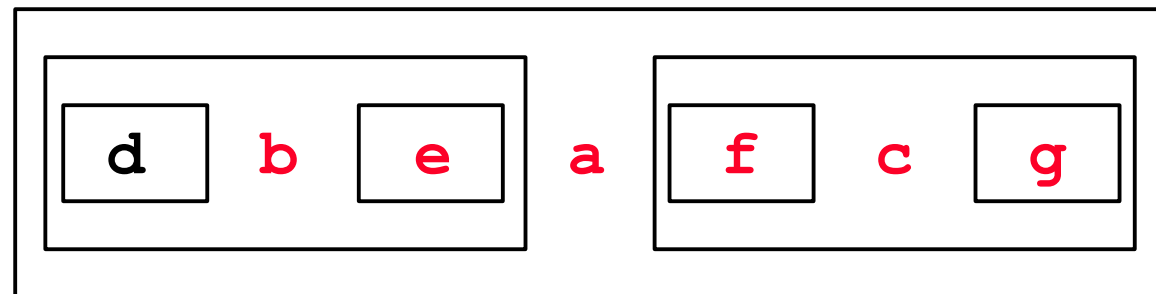
- Nota: gli algoritmi di visita si possono facilmente generalizzare al caso di albero non binario

Visita alberi binari in profondità in-ordine (simmetrica)

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

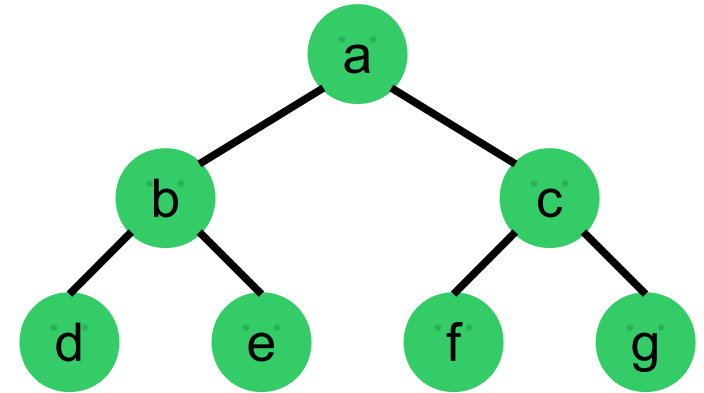


Sequenza:

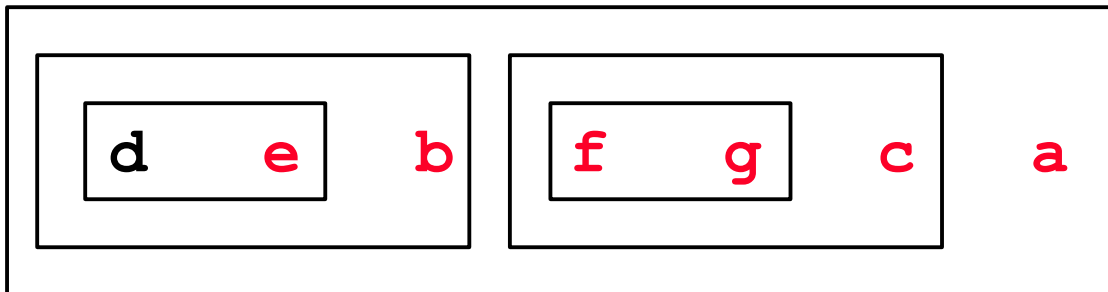


Visita alberi binari in profondità post-ordine

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

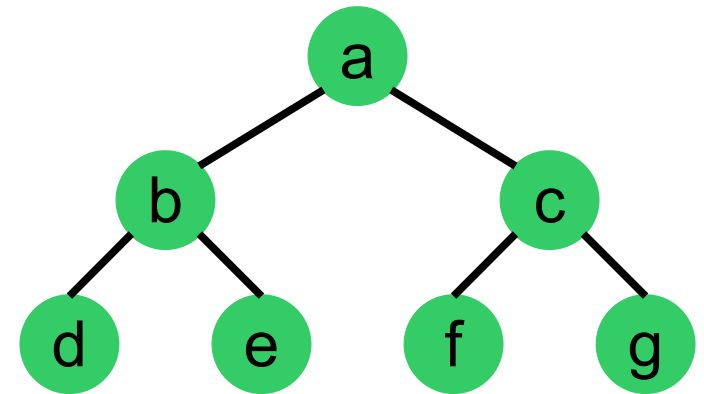


Sequenza:



Visita alberi binari: in ampiezza

```
visita-ampiezza(T)  
q = new Queue()  
q.insert(T)  
while ( !q.empty() ) {  
    p := q.dequeue()  
    if ( p ≠ nil ) {  
        // visita p  
        q.enqueue(p.left())  
        q.enqueue(p.right())  
    }  
}
```



Sequenza:

a

b

c

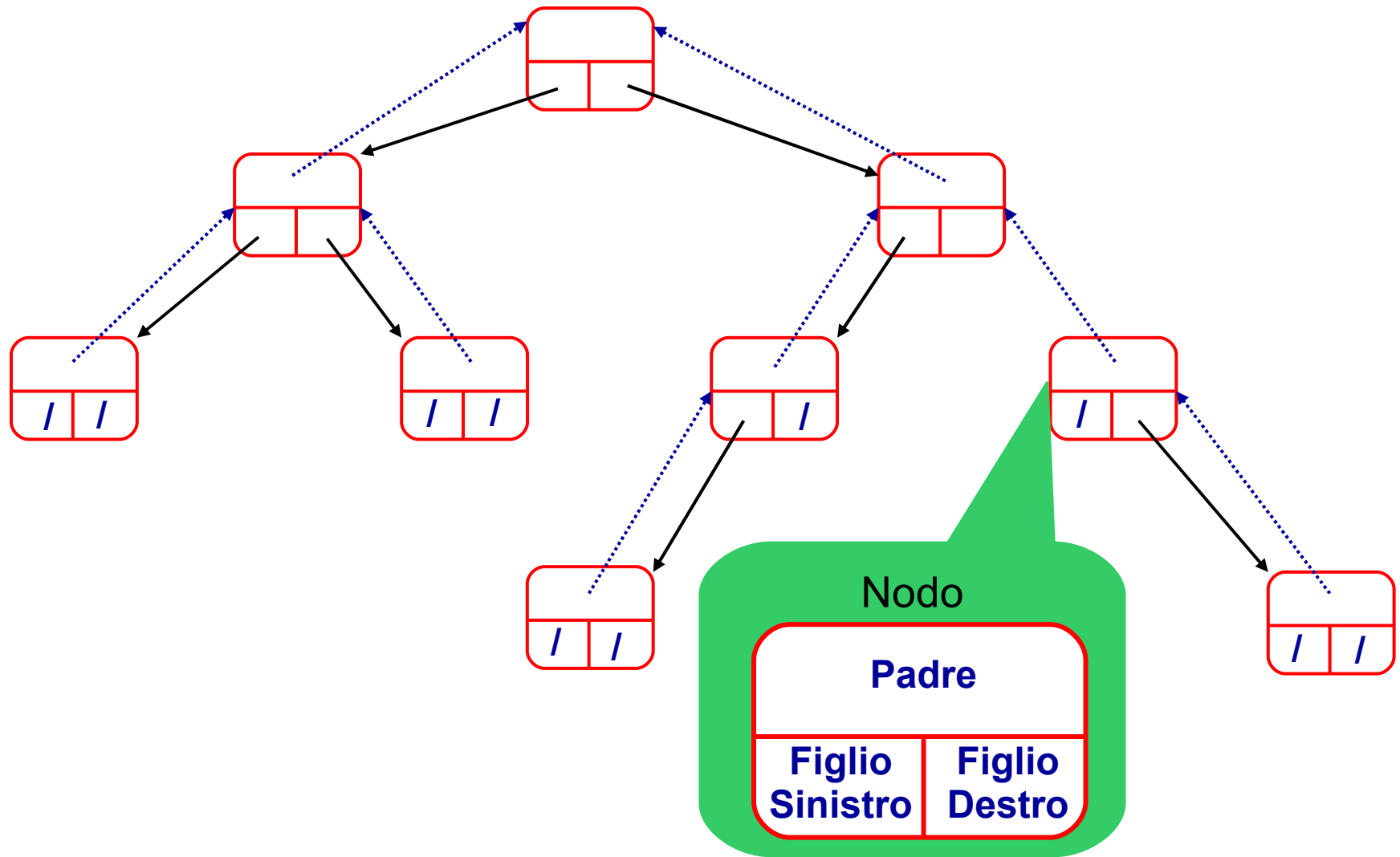
d

e

f

g

Implementazione di alberi binari



Interfaccia AlberoBin

```
public interface AlberoBin {
    public enum TipoVisita { PREORDER, INORDER, POSTORDER };
    public int numNodi();
    public int grado(Nodo v);
    public Object info(Nodo v);
    public Nodo radice();
    public Nodo padre(Nodo v);
    public Nodo sin(Nodo v);
    public Nodo des(Nodo v);
    public void cambiaInfo(Nodo v, Object info);
    public Nodo aggiungiRadice(Object info);
    public Nodo aggiungiFiglioSin(Nodo u, Object info);
    public Nodo aggiungiFiglioDes(Nodo u, Object info);
    public void innestaSin(Nodo u, AlberoBin albero);
    public void innestaDes(Nodo u, AlberoBin albero);
    public AlberoBin pota(Nodo v);
    public List visitaDFS(TipoVisita t);
    public List visitaDFS();
    public List visitaBFS();
}
```

Numero di figli di v

Stacca e restituisce
il sottoalbero avente
v come radice

Classe Nodo

L'interfaccia Rif denota un *riferimento* ad un elemento di una collezione. Non ha attributi né operazioni

```
public abstract class Nodo implements Rif {  
    public Object info;  
    public Nodo(Object info) {this.info = info;}  
    public abstract Object contenitore();  
}
```

Classe NodoBinPF

- Implementazione dei nodi di un albero binario mediante puntatori ai figli

```
public class NodoBinPF extends Nodo {
    public NodoBinPF padre;
    public NodoBinPF sin;
    public NodoBinPF des;
    public AlberoBin albero;

    public NodoBinPF(Object info) {super(info);}
    public AlberoBin contenitore(){
        NodoBinPF n = this;
        while (n.padre != null) n = n.padre;
        return n.albero;
    }
}
```

Alcune operazioni sugli alberi binari

```
public class AlberoBinPF implements AlberoBin {
    private NodoBinPF radice = null;

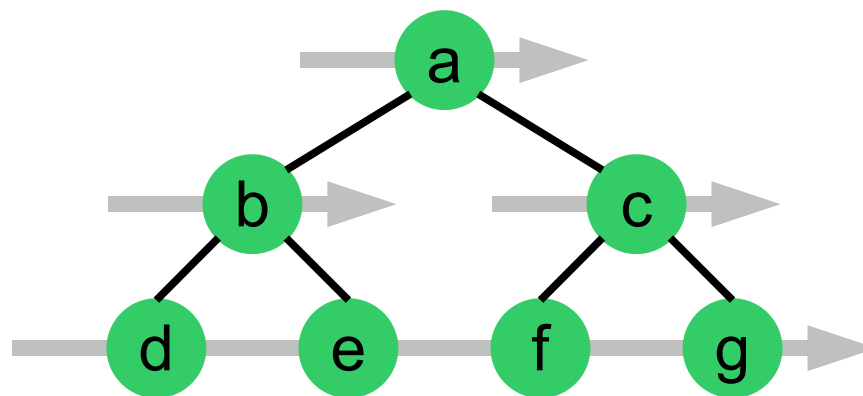
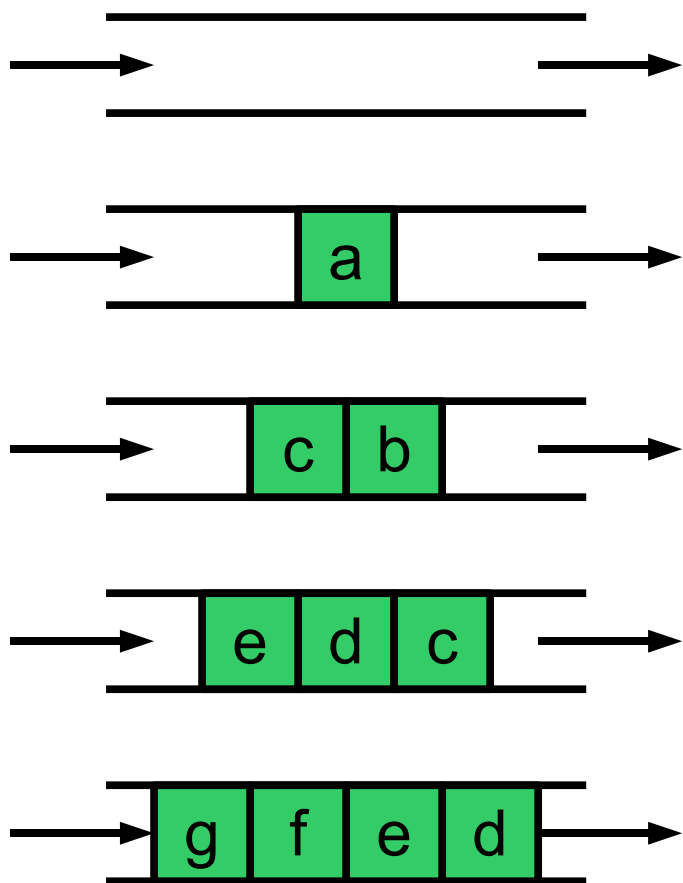
    public AlberoBinPF() {}
    public AlberoBinPF(Object info) {
        this.aggiungiRadice(info);
    }
    public Nodo aggiungiRadice(Object info) {
        if (radice != null) throw new EccezioneNodoEsistente();
        radice = new NodoBinPF(info);
        radice.albero = this;
        return radice;
    }
    /* ... */
}
```

Alcune operazioni sugli alberi binari visita in ampiezza

NB: Uso di
una coda

```
public List visitaBFS() {
    Coda c = new CodaCollegata();
    List listaRitorno = new LinkedList();
    c.enqueue(radice);
    while (!c.isEmpty()) {
        NodoBinPF u = (NodoBinPF) c.dequeue();
        if (u != null) {
            listaRitorno.add(u);
            c.enqueue(u.sin);
            c.enqueue(u.des);
        }
    }
    return listaRitorno;
}
```

Esempio



Alcune operazioni sugli alberi binari visita in profondità

Versione
iterativa

```
public List visitaDFS() {
    PilaArray p = new PilaArray();
    List listaNodi = new LinkedList();
    p.push(radice);
    while (!p.isEmpty()) {
        NodoBinPF u = (NodoBinPF) p.pop();
        if (u != null) {
            listaNodi.add(u);
            p.push(u.des);
            p.push(u.sin);
        }
    }
    return listaNodi;
}
```

NB: Uso di
una pila

Alcune operazioni sugli alberi binari

conteggio dei nodi

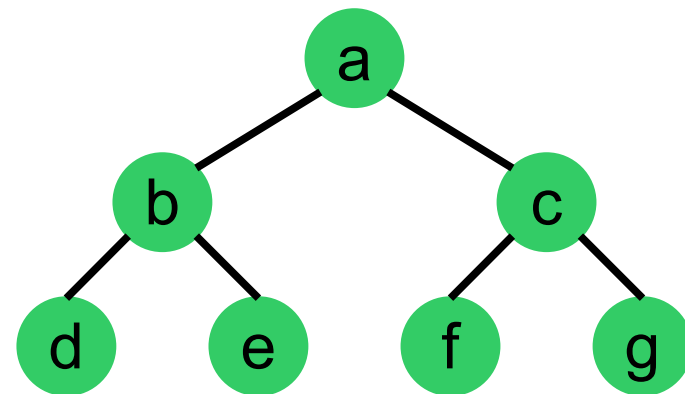
```
public int numNodi() {
    return numNodi(radice);
}
private int numNodi(NodoBinPF r) {
    return r == null ? 0 : numNodi(r.sin) + numNodi(r.des) + 1;
}
```

- Quale è la complessità del metodo numNodi() ?
- E' possibile modificare le strutture dati affinché numNodi() richieda tempo $O(1)$?
 - Quale è l'impatto di tale modifica sulla complessità computazionale delle altre operazioni dell'interfaccia AlberoBin?

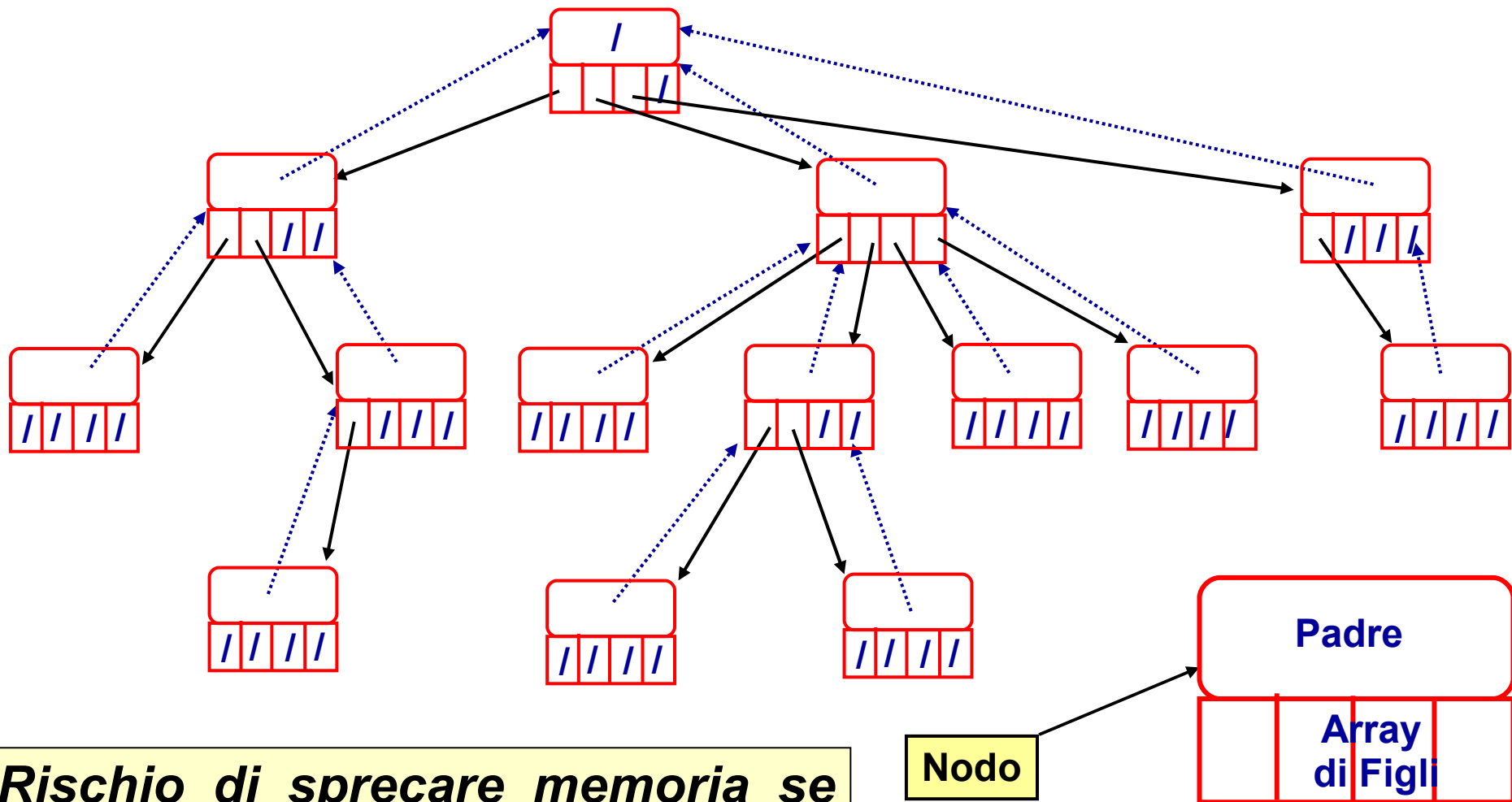
Implementazione di alberi generali: vettore dei padri

- L'albero è rappresentato da un vettore $v[]$ tale che $v[i]$ è l'indice del nodo padre di i
- Esempio (assumendo un vettore $v[1]...v[7]$)

a	b	c	d	e	f	g
0	1	1	2	2	3	3

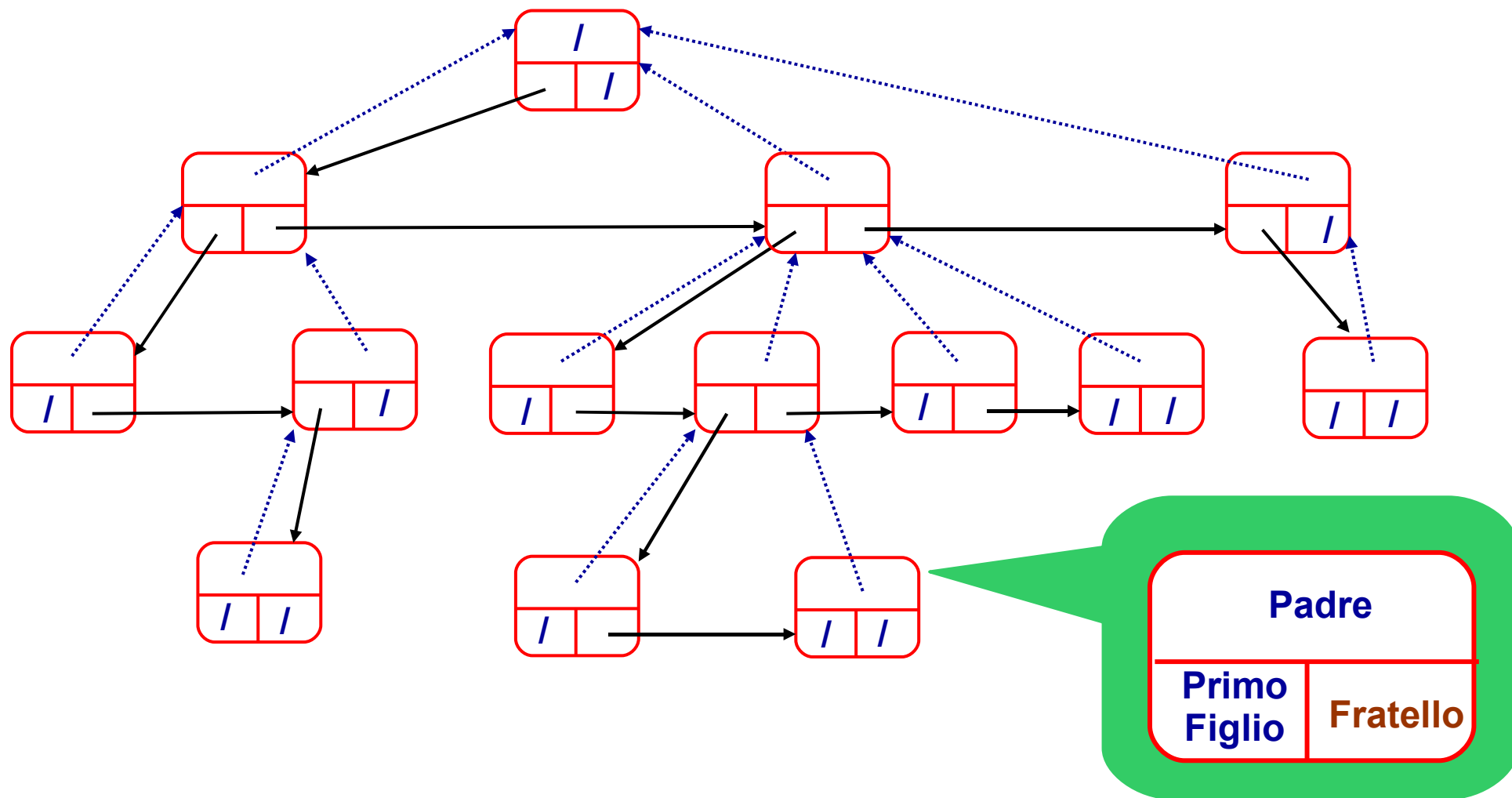


Implementazione di alberi generali: array di figli



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

Implementazione di alberi generali: nodo fratello



Interfaccia Albero (albero generico)

```
public interface Albero {
    public int numNodi();
    public int grado(Nodo v);
    public Object info(Nodo v);
    public Nodo radice();
    public Nodo padre(Nodo v);
    public List figli(Nodo v);
    public void cambiaInfo(Nodo v, Object info);
    public Nodo aggiungiRadice(Object info);
    public Nodo aggiungiFiglio(Nodo u, Object info);
    public void innesta(Nodo u, Albero a);
    public Albero pota(Nodo v);
    public List visitaDFS();
    public List visitaBFS();
}
```

Classe NodoPFFS

primo figlio-fratello successivo

```
public class NodoPFFS extends Nodo {
    public NodoPFFS padre;
    public NodoPFFS primo;
    public NodoPFFS succ;
    public Albero albero;

    public NodoPFFS(Object info) {super(info);}
    public Albero contenitore(){
        NodoPFFS n = this;
        while (n.padre != null) n = n.padre;
        return n.albero;
    }
}
```

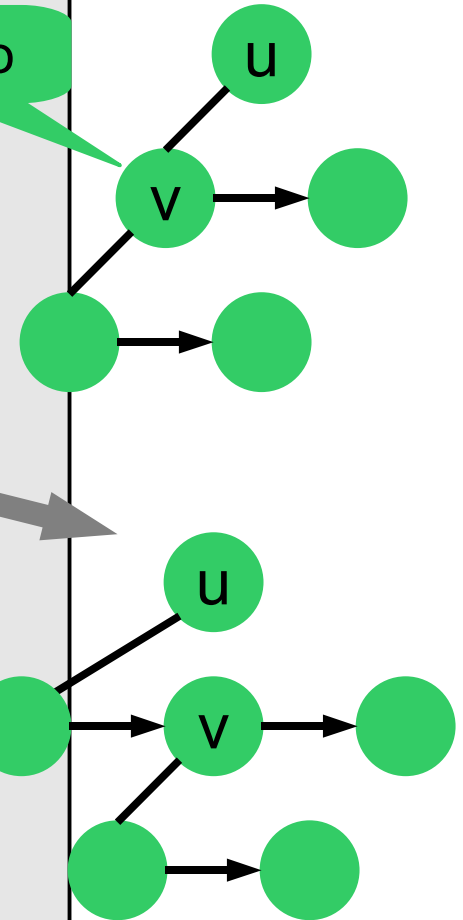
Il metodo pota()

(Stacca e restituisce il sottoalbero avente v come radice)

```
public Albero pota(Nodo v) {  
    if (v == radice) {  
        radice = null;  
        return new AlberoPFFS(v);  
    }  
    NodoPFFS u = ((NodoPFFS) v).padre;  
    if (u.primo == v) {  
        u.primo = u.primo.succ;  
    } else {  
        NodoPFFS temp = u.primo;  
        boolean nodoTrovato = false;  
        for (; temp.succ != null; temp = temp.succ) {  
            if (temp.succ == v) {  
                nodoTrovato = true;  
                break;  
            }  
        }  
        if (nodoTrovato) temp.succ = temp.succ.succ;  
    }  
    ((NodoPFFS) v).succ = null;  
    return new AlberoPFFS(v);  
}
```

u.primo

u.primo



Semplici esercizi basati su visite

- Dato un albero radicato T , calcolare la sua altezza
- Dato un albero radicato T , stampare tutti i nodi a profondità h