Tecniche di ricerca in strutture dati complesse: Alberi N-ari

Prof. Ing. Loris Penserini, PhD

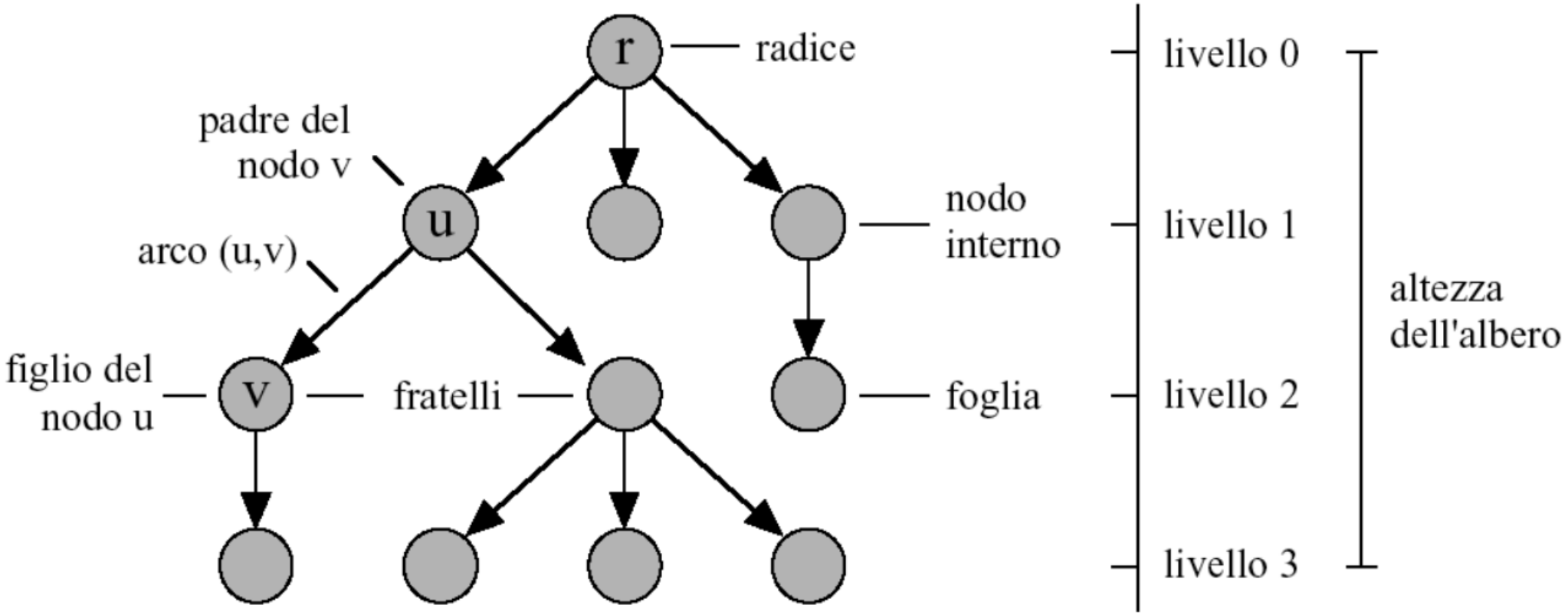
https://orcid.org/0009-0008-6157-0396

Overview

Parleremo di:

- Creazione di un albero N-ario:
  aggiungere/visualizzare nodi
- Depth-First Search: preorder e postorder (ricorsivo)
- Complessità DFS
- Breadth-First Search (iterativo)
- Complessità BFS
- Project Work

Strutture dati ad «albero»



Alberi N-ari…

Lavorare con alberi n-ari e le loro tecniche di visita è un vero salto di livello nella programmazione: si lavora con ricorsione, strutture dati complesse, e si aprono porte verso algoritmi di parsing e di intelligenza artificiale…

Parliamo di strutture dati, ad «albero» appunto, in cui ogni nodo può avere un numero variabile di figli

Creazione di un Albero N-ario

```python
T = TypeVar("T")

@dataclass
class NaryNode(Generic[T]):
    """
    Generic[T] dice: “questa classe è generica sul tipo T”.
    value: T e children: List[NaryNode[T]] significano che tutti i valori
    del nodo e dei suoi figli hanno lo stesso tipo.
    """
    value: T

    """
    Il default_factory=list garantisce che ogni nodo abbia la sua lista separata,
    e non una lista condivisa da tutti i nodi.
    """
    children: List["NaryNode[T]"] = field(default_factory=list)

    # -- costruzione albero --
    # add_child => nodo foglia
    def add_child(self, value: T) -> "NaryNode[T]":
        child = NaryNode(value)
        self.children.append(child)
        return child

    # add_children => lista di nodi associata un singolo nodo
    def add_children(self, values: Iterable[T]) -> List["NaryNode[T]"]:
        nodes = [NaryNode(v) for v in values]
        # aggiunge questi nuovi nodi "nodes" alla lista dei figli del nodo corrente "self".
        self.children.extend(nodes)
        return nodes
```

Aggiungere nodi dell'Albero

```
30      # add_children => lista di nodi associata un singolo nodo
31      def add_children(self, values: Iterable[T]) -> List["NaryNode[T]"]:
32          nodes = [NaryNode(v) for v in values]
33          # aggiunge questi nuovi nodi "nodes" alla lista dei figli del nodo corrente "self".
34          self.children.extend(nodes)
35          return nodes
```

values: Iterable[T] è una collezione di valori (può essere lista, tupla, range, ecc.).

[NaryNode(v) for v in values] crea un nuovo nodo per ciascun valore.

self.children.extend(nodes) aggiunge questi nuovi nodi alla lista dei figli del nodo corrente (self).

return nodes restituisce la lista (-> List[...]) dei nuovi figli appena creati, così si possono salvare in variabili e continuare a usarli.

Aggiunge nodi foglia

```
24      # add_child => nodo foglia
25      def add_child(self, value: T) -> "NaryNode[T]":
26          child = NaryNode(value)
27          self.children.append(child)
28          return child
29
```

-> "NaryNode[T]":

Significa che il metodo restituisce un singolo nodo, cioè la foglia appena inserita.

Creazione di un Albero N-ario

```python
# ---- esempio d'uso ----
if __name__ == "__main__":
    root = NaryNode("A")
    b, c, d = root.add_children(["B", "C", "D"])
    b.add_children(["E", "F"])
    g = c.add_child("G")
    g.add_children(["I", "L"])
    d.add_child("H")
```

```
Stampa dell'albero creato con la funzione render():
A
├── B
│   ├── E
│   └── F
├── C
│   └── G
│       ├── I
│       └── L
└── D
    └── H
```

Overview

Parleremo di:

- Creazione di un albero N-ario:
  aggiungere/visualizzare nodi
- Depth-First Search: preorder e postorder (ricorsivo)
- Complessità DFS
- Breadth-First Search (iterativo)
- Complessità BFS
- Project Work

Depth-First Search o visita in profondità

In questo ambito di ricerca si distinguono le tipologie:

- **DFS preorder:** prima il nodo e poi i figli
- **DFS postorder:** prima i figli e poi il nodo

DFS Preorder

La funzione **dfs_preorder** implementa una ricerca in profondità (preordine). Essa visita il nodo prima dei figli, utilizzando la ricorsione. La complessità temporale è O(N), e quella spaziale O(H), considerando: N => numero dei nodi e H => (altezza albero) il numero dei livelli di profondità dell'albero.

Depth-First Search

```
Stampa dell'albero creato con la funzione render():
A
├── B
│   ├── E
│   └── F
├── C
│   └── G
│       ├── I
│       └── L
└── D
    └── H
```

```python
52      def print_dfs_preorder(self):
53          """Stampa i nodi in ordine DFS-preorder: Nodo -> Figli"""
54          print(f"Visito nodo: {self.value}")
55          for c in self.children:
56              c.print_dfs_preorder()
```

```
Stampa con print_dfs_preorder():
Visito nodo: A
Visito nodo: B
Visito nodo: E
Visito nodo: F
Visito nodo: C
Visito nodo: G
Visito nodo: I
Visito nodo: L
Visito nodo: D
Visito nodo: H
```

DFS postorder

La funzione **dfs_postorder** implementa una ricerca in profondità (postordine). Essa visita i figli prima del nodo padre, utilizzando la ricorsione. La complessità temporale è O(N), e quella spaziale O(H), considerando: N => numero dei nodi e H => (altezza albero) il numero dei livelli di profondità dell'albero.

DFS postorder

```
Stampa dell'albero creato con la funzione render():
A
├── B
│   ├── E
│   └── F
├── C
│   └── G
│       ├── I
│       └── L
└── D
    └── H
```

```python
58      def print_dfs_postorder(self):
59          """Stampa i nodi in ordine DFS-postorder: Figli -> Nodo"""
60          for c in self.children:
61              c.print_dfs_postorder()
62          print(f"Visito nodo: {self.value}")
```

```
Stampa con print_dfs_postorder():
Visito nodo: E
Visito nodo: F
Visito nodo: B
Visito nodo: I
Visito nodo: L
Visito nodo: G
Visito nodo: C
Visito nodo: H
Visito nodo: D
Visito nodo: A
```

DFS: Complessità temporale $O(N)$

DFS visita tutti i nodi dell'albero (o del grafo connesso); cioè, **ogni nodo viene visitato una volta** => $O(1)$ per fare l'operazione (es. stampare, confrontare, ecc.).

Non ci sono ripetizioni (a differenza di un grafo con cicli, qui gli archi padre-figlio sono unici).

Si itera sui suoi figli una volta ciascuno, per cui se hai N nodi totali, l'operazione di visita costa $O(N)$.

DFS: Complessità Spaziale O(H)

La DFS è tipicamente ricorsiva (oppure usa una pila).

In ogni chiamata ricorsiva tiene in memoria lo stato del nodo corrente e deve ricordarsi dove tornare.

La profondità massima della ricorsione è pari all'altezza dell'albero, indicata con H.

- **Se l'albero è bilanciato, $H \approx \log N$.**
- **Se l'albero è degenerato (cioè una lista), $H = N$.**

Quindi lo spazio aggiuntivo usato dalla DFS è proporzionale a H, non a N: $O(H)$.

Ovviamente devi contare anche lo spazio per contenere i dati stessi dell'albero, ma quello è $O(N)$ e "già c'è", indipendente dall'algoritmo.

Overview

Parleremo di:

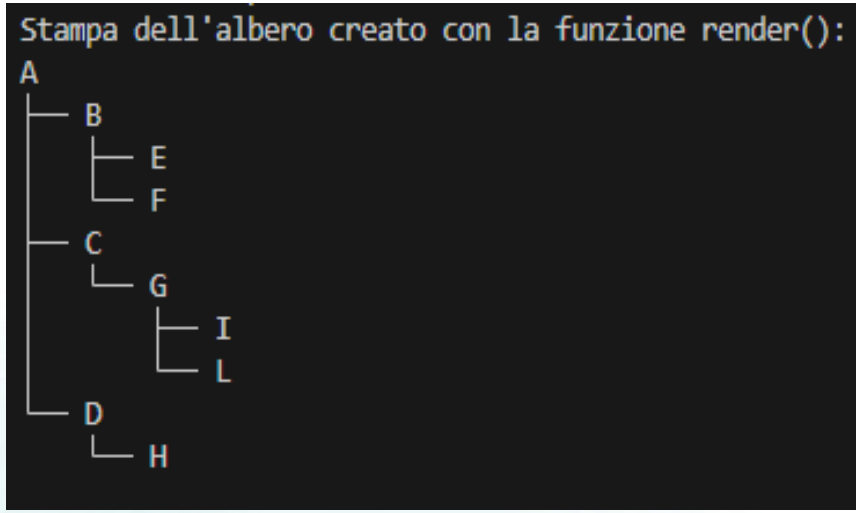
- Creazione di un albero N-ario:
  aggiungere/visualizzare nodi
- Depth-First Search: preorder e postorder (ricorsivo)
- Complessità DFS
- Breadth-First Search (iterativo)
- Complessità BFS
- Project Work

Breadth-First Search “visita in ampiezza”

Idea di base, sfrutta un approccio iterativo:

- Si parte da un nodo iniziale (es. la radice di un albero).
- Si visitano prima tutti i nodi di un livello, poi si passa al livello successivo.
- Si usa tipicamente una coda (queue/deque) per tenere traccia dei nodi ancora da visitare.

BFS Iterativa




```python
    def print_bfs(self):
        """Stampa i nodi in ordine BFS (livelli)"""
        q: Deque[NaryNode[T]] = deque([self]) # q è una coda doppia (double-ended queue),
                                              # che conterrà elementi di tipo NaryNode[T]
        while q:
            node = q.popleft()
            print(f"Visito nodo: {node.value}")
            q.extend(node.children)
```

```
Stampa con print_bfs():
Visito nodo: A
Visito nodo: B
Visito nodo: C
Visito nodo: D
Visito nodo: E
Visito nodo: F
Visito nodo: G
Visito nodo: H
Visito nodo: I
Visito nodo: L
```

BFS Iterativa

La visita in ampiezza (BFS, Breadth-First Search) non scende "in profondità" come la DFS, quindi non si appoggia naturalmente alla chiamata ricorsiva. I passi principali sono:

- BFS tiene una coda esplicita (**q**) con i nodi da visitare.
- Si estrae un nodo alla volta (popleft()), si visita, e si accodano i suoi figli (**q.extend(node.children)**).
- Il ciclo **while q:** continua fino a svuotare la coda.

Quindi è una implementazione iterativa basata su una struttura dati ausiliaria (la coda), non ricorsiva..

Le Code in BFS

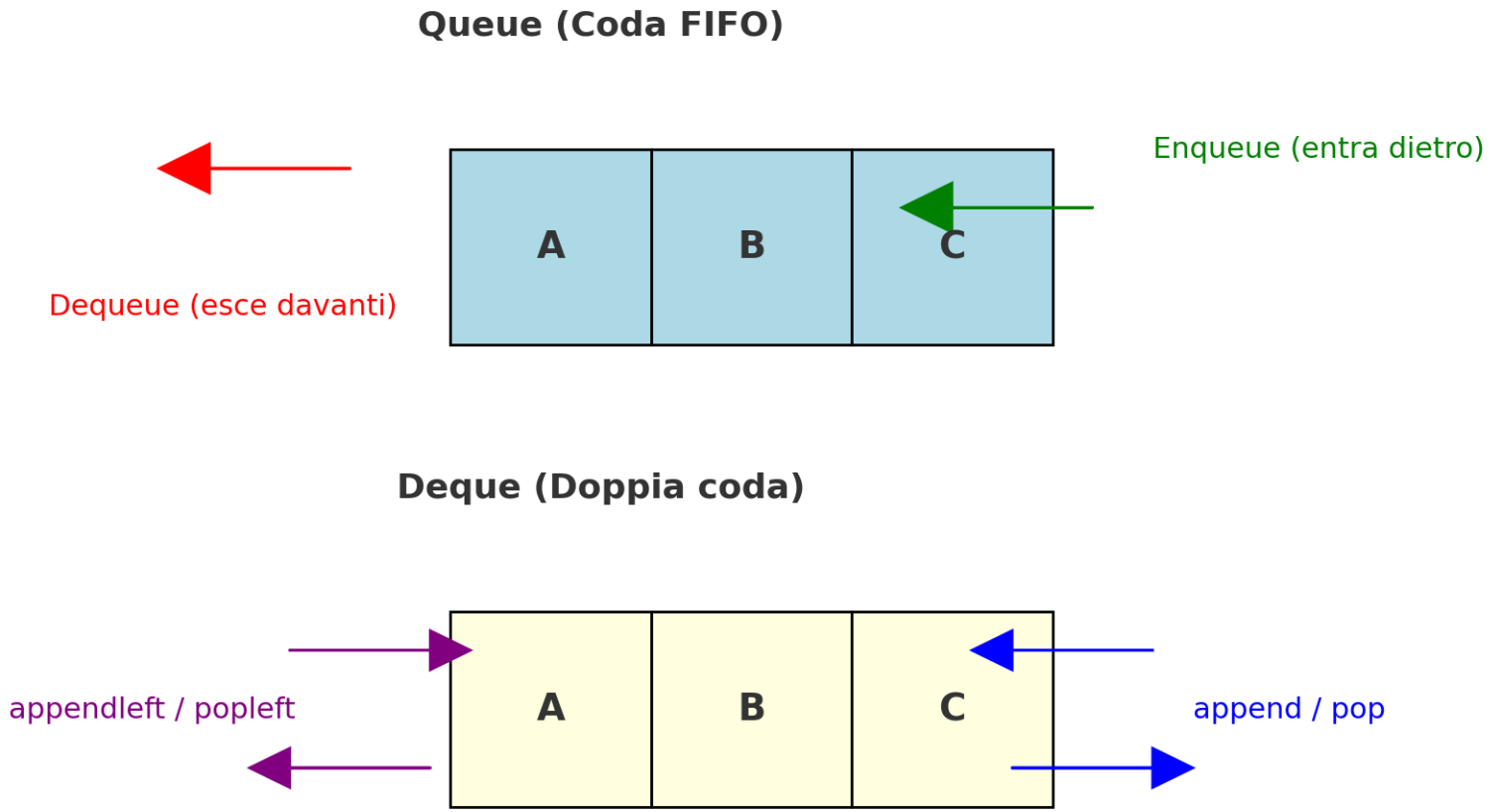
Queue (coda FIFO). Entra solo da destra (enqueue), esce solo da sinistra (dequeue).

Deque (doppia coda). Si può inserire ed estrarre da entrambe le estremità:

- destra: append, pop
- sinistra: appendleft, popleft

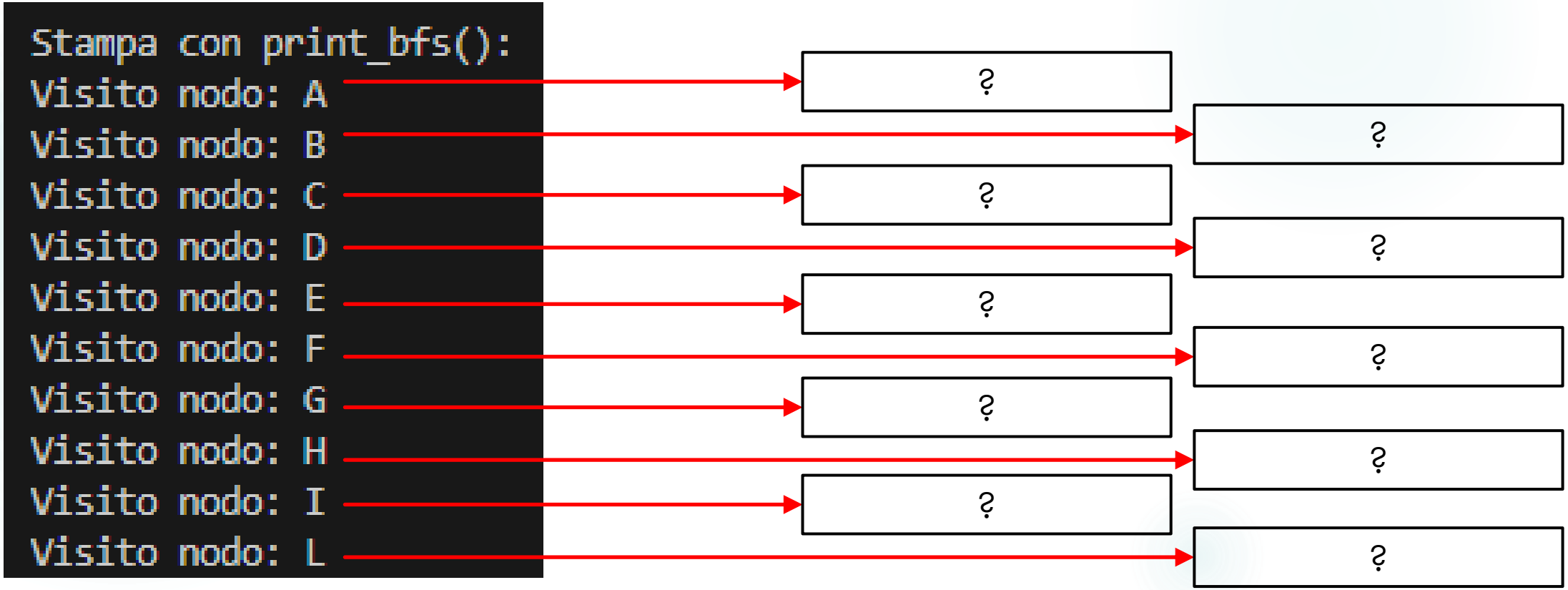
In Python, per BFS si usa «deque» perché l'operazione **popleft() è molto efficiente** (O(1)), mentre farlo su una lista costerebbe O(n).

Le Code


Queue (Coda FIFO)
Enqueue (entra dietro)
A
B
C
Dequeue (esce davanti)
Deque (Doppia coda)
appendleft / popleft
A
B
C
append / pop

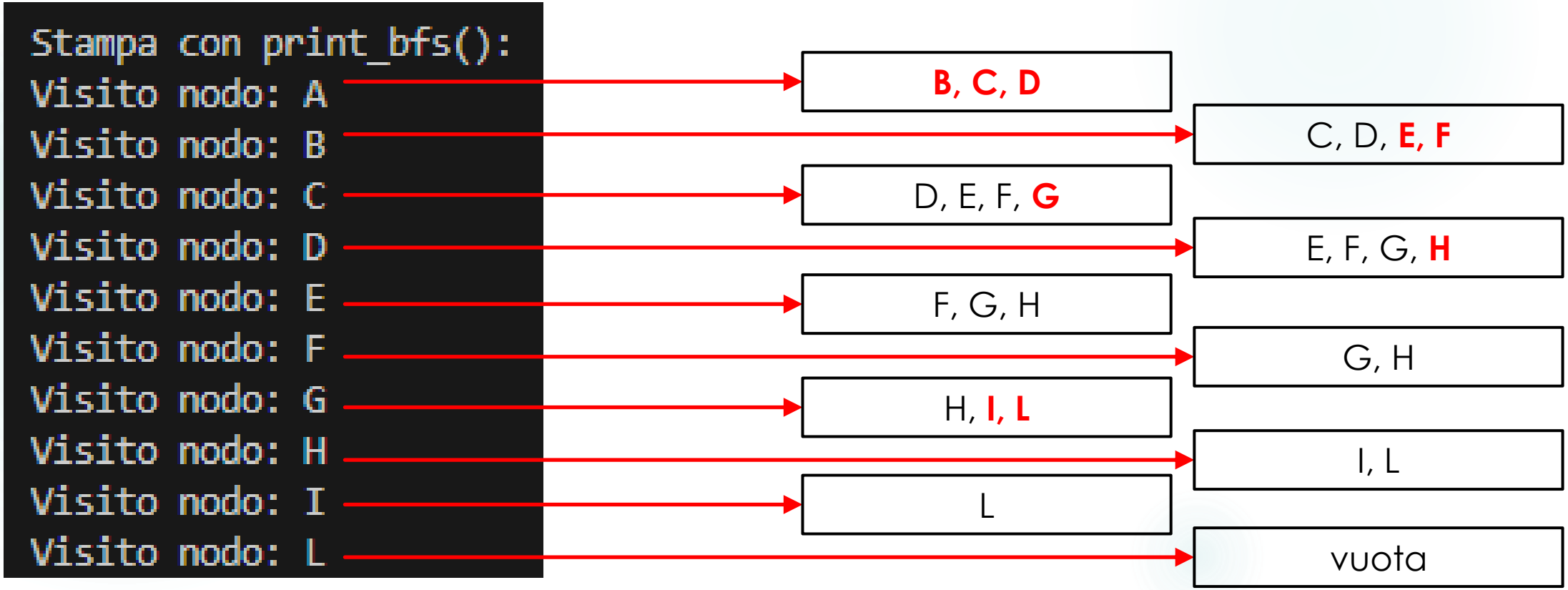
PW_code

Disegnare come evolve la «coda» **q**, cioè i suoi elementi interni, per ogni nodo visitato.



PW_code: Soluzione

Disegnare come evolve la «coda» **q**, cioè i suoi elementi interni, per ogni nodo visitato.



Bibliografia

[Raschka et al., book 2021] Sebastian Raschka, Vahid Mirjalili, «Machine Learning con Python – Nuova edizione – Costruire algoritmi per generare conoscenza», Apogeo. ISBN: 978-88-503-3524-4. 2021.

[Lambert, book 2024] Kenneth A. Lambert, «Programmazione in Python – Terza edizione», Apogeo education – Maggioli Editore. ISBN: 978-88-916-7143-1. 2024.

[Penserini, GitHub 2025] Loris Penserini, «Corso ITS Academy Turismo Marche: AI Specialist», https://github.com/penserini/ITS_AI_Specialist, 2025.