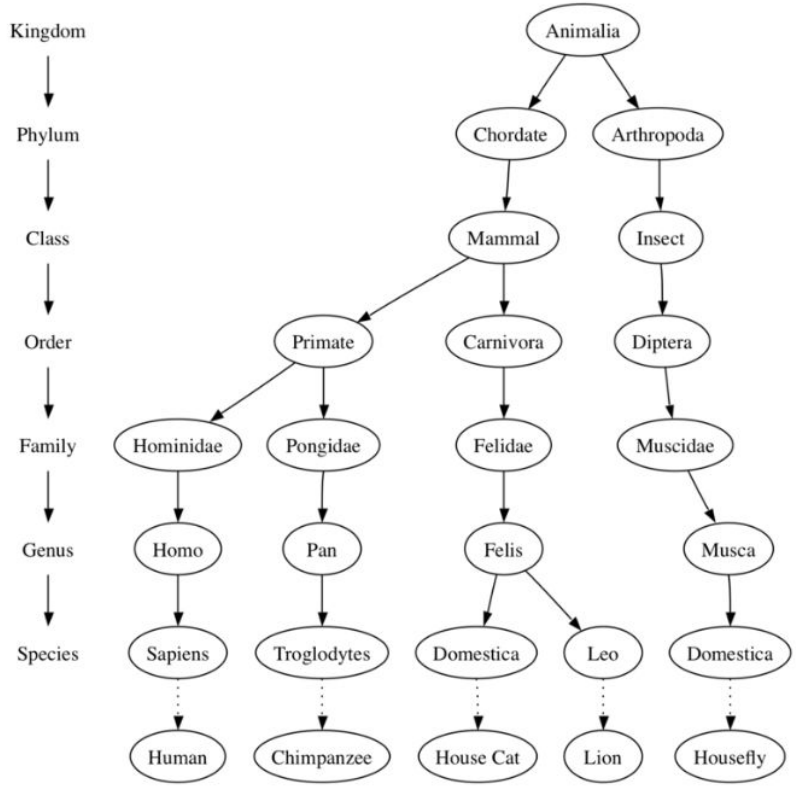
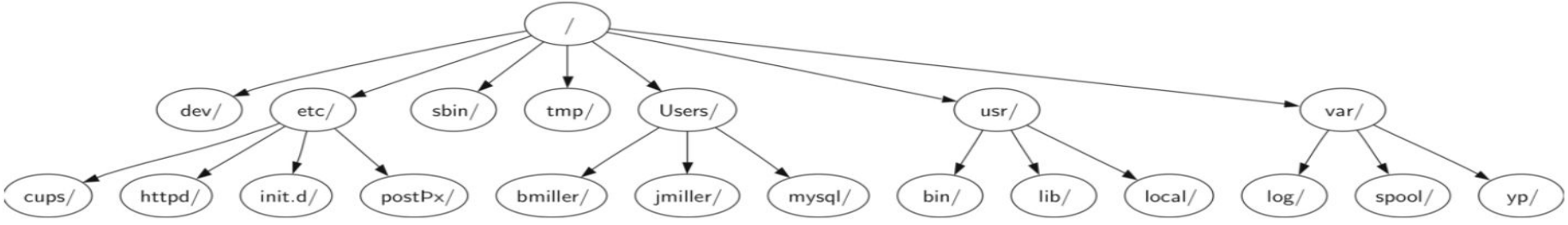
TECNICO SUPERIORE WEB DEVELOPER FULL STACK

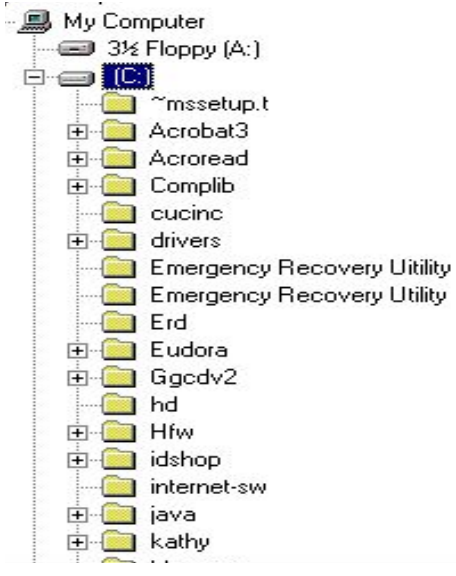
#9 - Alberi

Alcuni esempi di alberi - 1: La natura



Alcuni esempi di alberi - 2: File system

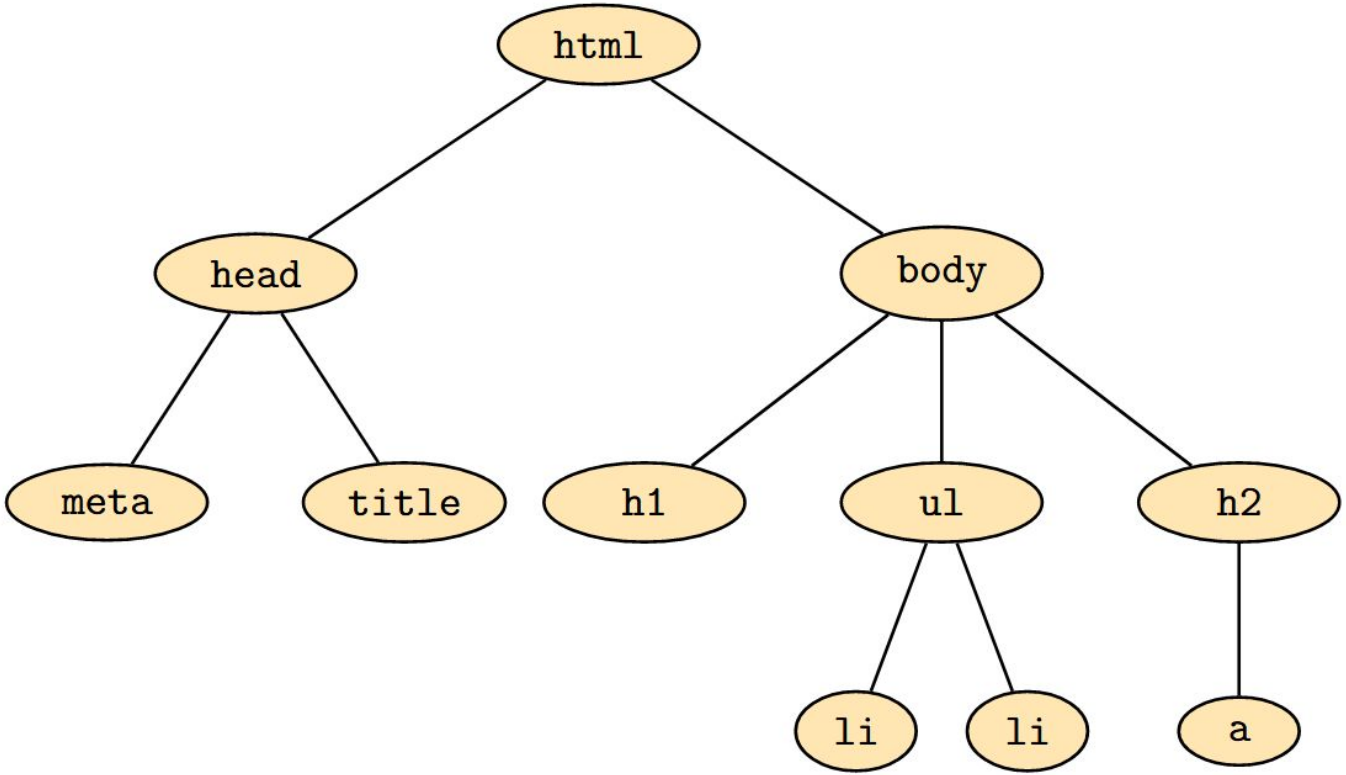




Alcuni esempi di alberi - 3: HTML

```
<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html"/>
        <title>simple</title>
    </head>
    <body>
        <h1>A simple web page</h1>
        <ul>
            <li>List item one</li>
            <li>List item two</li>
        </ul>
        <h2>
            <a href="http://www.google.com">Google</a>
        </h2>
    </body>
</html>
```

Alcuni esempi di alberi - 3: HTML



Albero radicato: definizione (1)

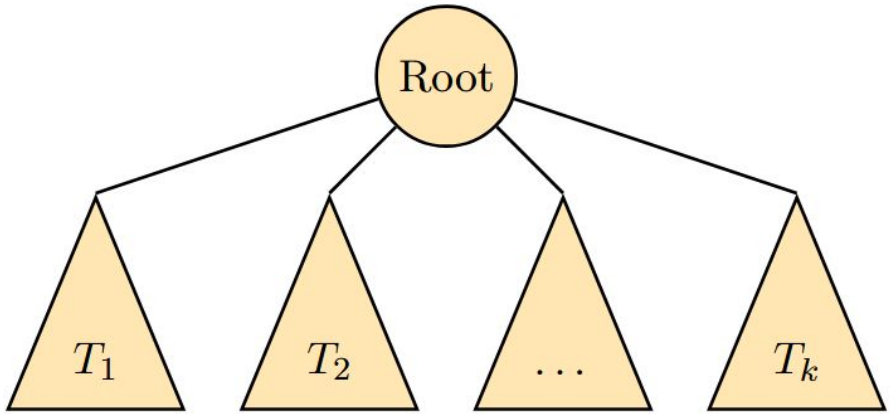
Un albero consiste di un insieme di nodi e un insieme di archi orientati che connettono coppie di nodi, con le seguenti proprietà:

- Un nodo dell'albero è designato come nodo radice;
- Ogni nodo `n`, a parte la radice, ha esattamente un arco entrante;
- Esiste un cammino unico dalla radice ad ogni nodo;
- L'albero è connesso.

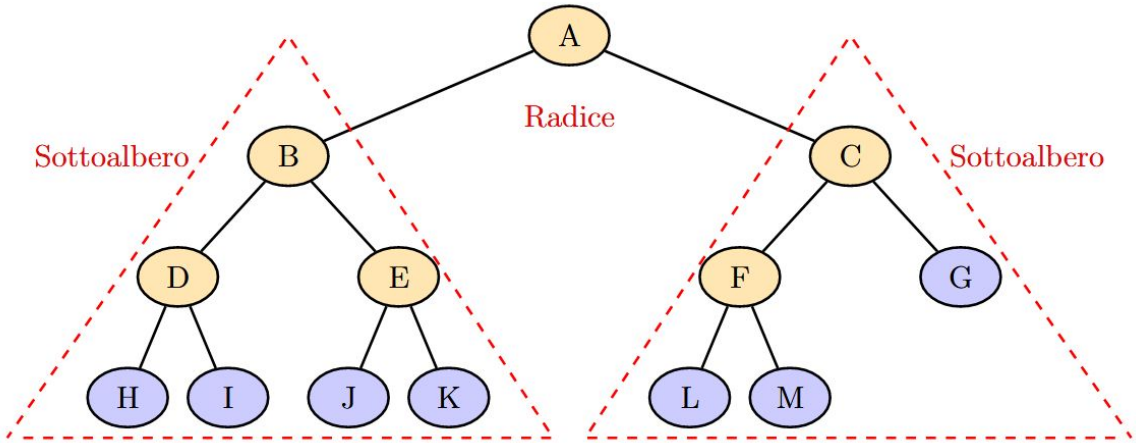
Albero radicato: definizione (2)

Un albero è dato da:

- un insieme vuoto, *oppure*
- un nodo radice e zero o più sottoalberi, ognuno dei quali è un albero; la radice è connessa alla radice di ogni sottoalbero con un arco orientato



Alberi, un po’ di terminologia



- A è la radice
- B, C sono radici dei sottoalberi
- D, E sono fratelli
- D, E sono figli di B
- B è il padre di D, E
- I nodi viola sono foglie
- Gli altri nodi sono nodi interni

Alberi, un po’ di terminologia

Profondità nodi (Depth)

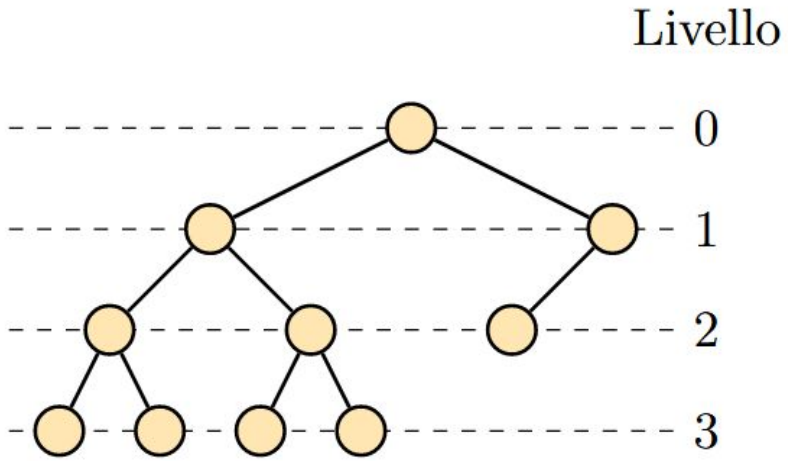
La lunghezza del cammino semplice dalla radice al nodo (misurato in numero di archi).

Livello (Level)

L’insieme di nodi alla stessa profondità.

Altezza albero (Height)

La profondità massima delle foglie

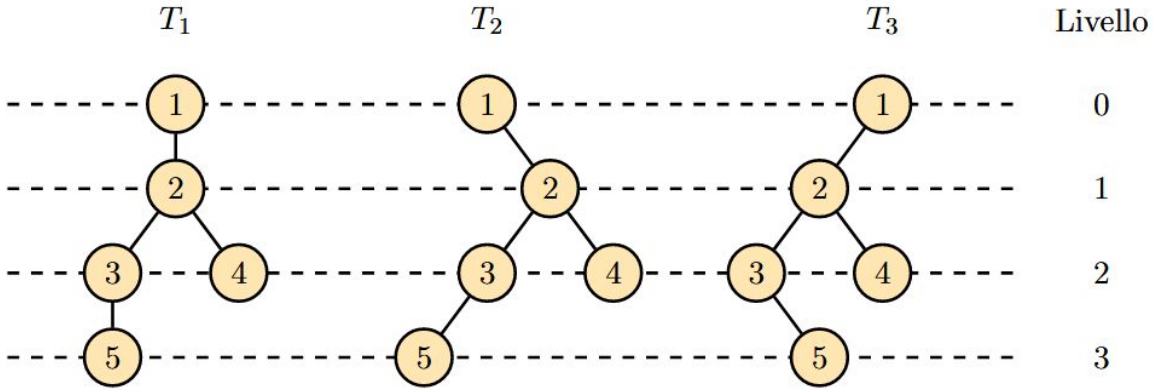


Alberi binari

Albero binario

Un albero binario è un albero radicato in cui ogni nodo ha al massimo due figli, identificati come figlio sinistro e figlio destro.

Nota: Due alberi T e U che hanno gli stessi nodi, gli stessi figli per ogni nodo e la stessa radice, sono distinti qualora un nodo k sia designato come figlio sinistro di un nodo v in T e come figlio destro di v in U.



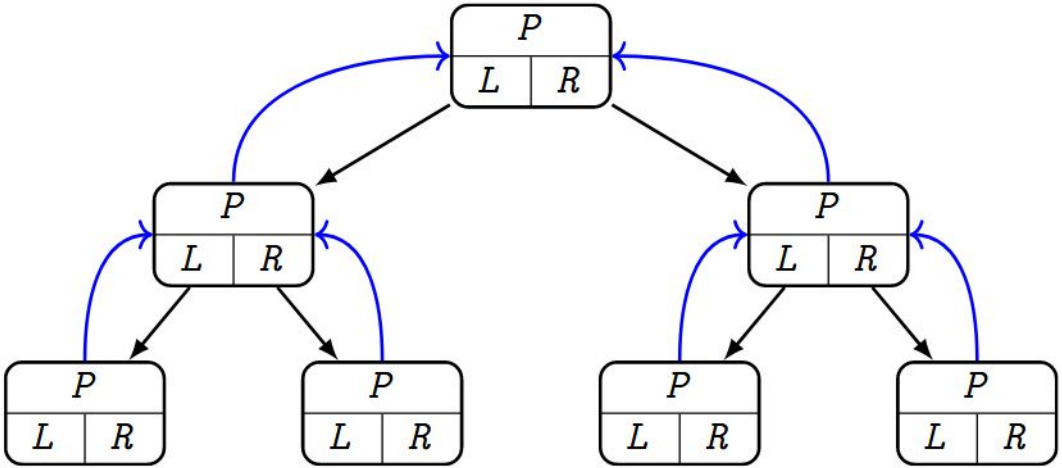
Albero binario, la specifica

Tree (Item v)	Costruisce un nuovo nodo, contenente v, senza figli o genitori
Item read()	Legge il valore memorizzato nel nodo
write (Item v)	Modifica il valore memorizzato nel nodo
Tree parent()	Restituisce il padre, oppure **nil** se questo nodo è radice

Albero binario, la specifica

Tree left() Tree right()	Restituisce il figlio sinistro (destro) di questo nodo; restituisce **nil** se assente
insertLeft (Tree t) insertRight (Tree t)	Inserisce il sottoalbero radicato in t come figlio sinistro (destro) di questo nodo
deleteLeft () deleteRight ()	Distrugge (ricorsivamente) il figlio sinistro (destro) di questo nodo

Memorizzare un albero binario



Campi memorizzati nei nodi

- parent: reference al nodo padre
- left: reference al figlio sinistro
- right: reference al figlio destro

Albero binario, implementazione

```
TREE
Tree(ITEM v)
    TREE t = new TREE
    t.parent = nil
    t.left = t.right = nil
    t.value = v
    return t

insertLeft(TREE T)
    if left == nil then
        T.parent = this
        left = T

insertRight(TREE T)
    if right == nil then
        T.parent = this
        right = T
```

```
deleteLeft()
    if left ≠ nil then
        left.deleteLeft()
        left.deleteRight()
        delete left
        left = nil

deleteRight()
    if right ≠ nil then
        right.deleteLeft()
        right.deleteRight()
        delete right
        right = nil
```

Visita di un albero

Una visita è una strategia per analizzare (visitare) tutti i nodi di un albero, ossia “scansionarne” tutti i valori.

Visità in profondità, o Depth-First Search (DFS)

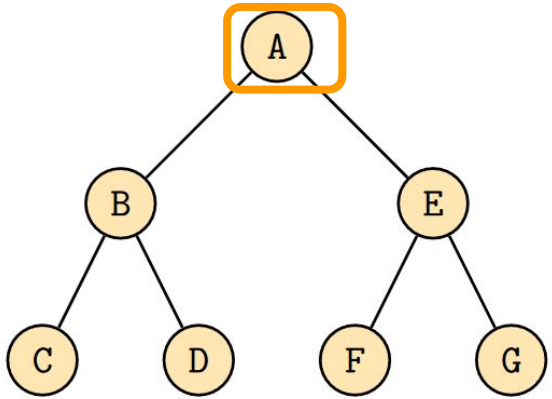
- Per visitare un albero, si visita ricorsivamente ognuno dei suoi sottoalberi
- Tre varianti: pre/in/post visita (pre/in/post order)
- Richiede uno stack di appoggio

Visita in ampiezza, o Breadth First Search (BFS)

- Ogni livello dell’albero viene visitato, uno dopo l’altro
- Richiede una queue

DFS: Pre-order

```
dfs(TREE t)
if t ≠ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    dfs(t.right())
```

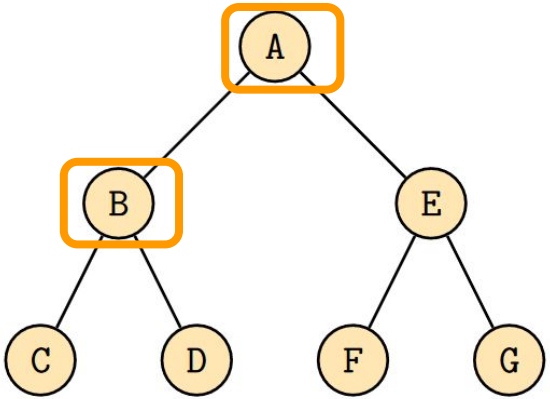


Sequenza: **A**

Stack: **A**

DFS: Pre-order

```
dfs(TREE t)
if t ≠ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    dfs(t.right())
```

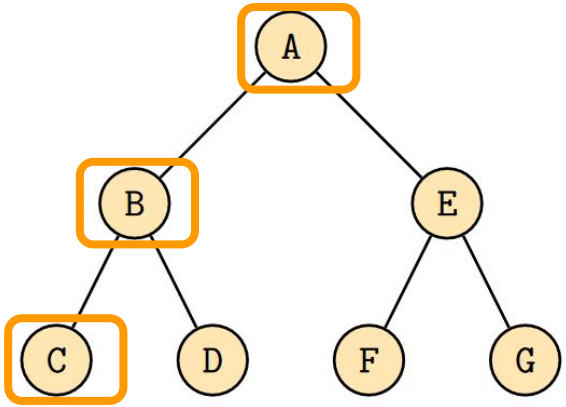


Sequenza: **A** **B**

Stack: **A** **B**

DFS: Pre-order

```
dfs(TREE t)
if t ≠ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    dfs(t.right())
```

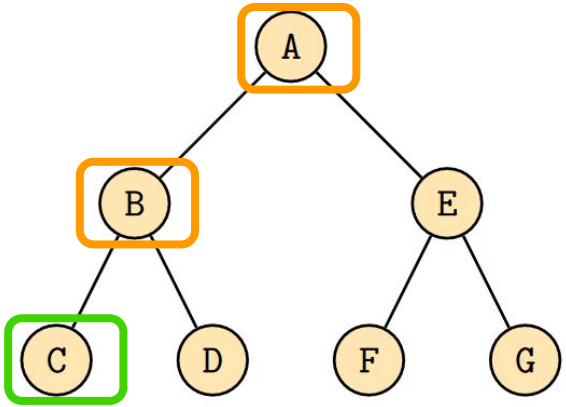


Sequenza: **A B C**

Stack: **A B C**

DFS: Pre-order

```
dfs(TREE t)
if t ≠ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    dfs(t.right())
```

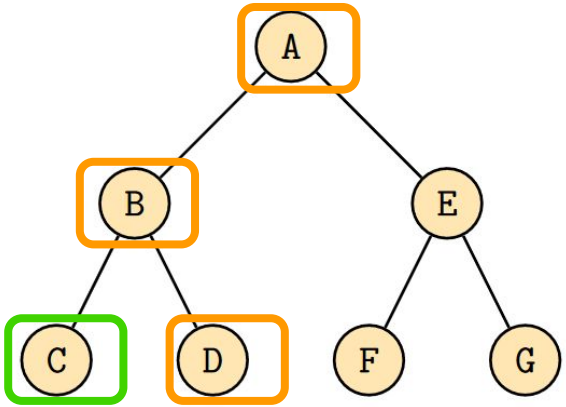


Sequenza: **A B C**

Stack: **A B**

DFS: Pre-order

```
dfs(TREE t)
if t ≠ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    dfs(t.right())
```

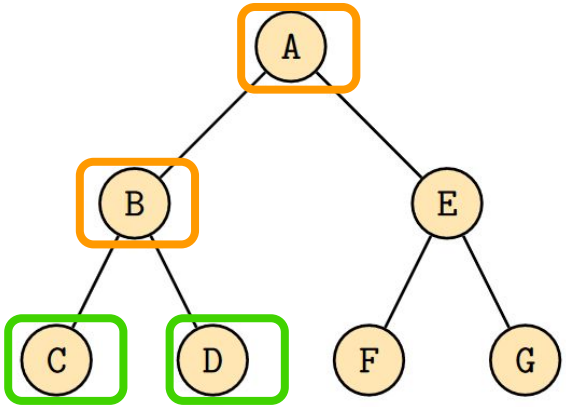


Sequenza: **A B C D**

Stack: **A B D**

DFS: Pre-order

```
dfs(TREE t)
if t ≠ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    dfs(t.right())
```

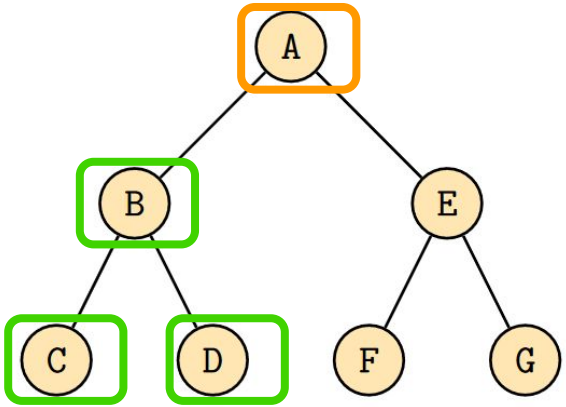


Sequenza: **A B C D**

Stack: **A B**

DFS: Pre-order

```
dfs(TREE t)
if t ≠ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    dfs(t.right())
```

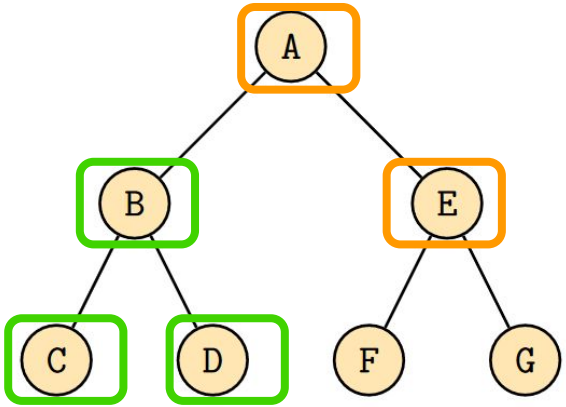


Sequenza: **A B C D**

Stack: **A**

DFS: Pre-order

```
dfs(TREE t)
if t ≠ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    dfs(t.right())
```

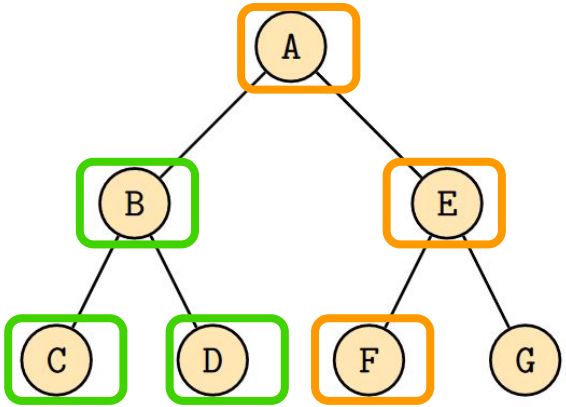


Sequenza: **A B C D E**

Stack: **A E**

DFS: Pre-order

```
dfs(TREE t)
if t ≠ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    dfs(t.right())
```

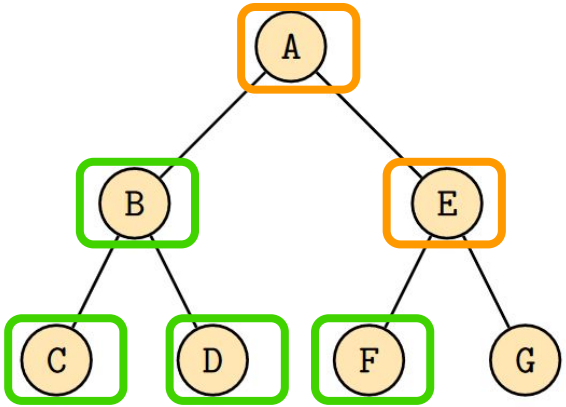


Sequenza: **A B C D E F**

Stack: **A E F**

DFS: Pre-order

```
dfs(TREE t)
if t ≠ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    dfs(t.right())
```

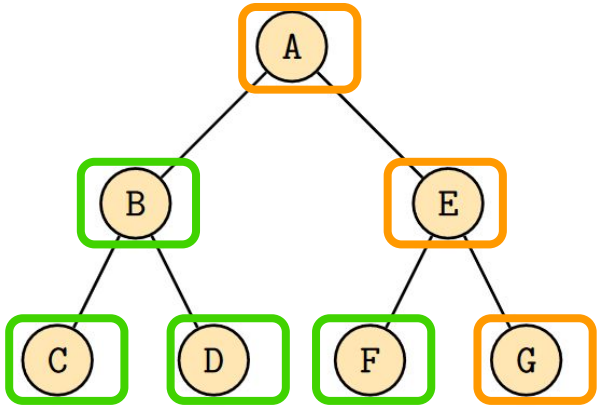


Sequenza: **A B C D E F**

Stack: **A E**

DFS: Pre-order

```
dfs(TREE t)
if t ≠ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    dfs(t.right())
```

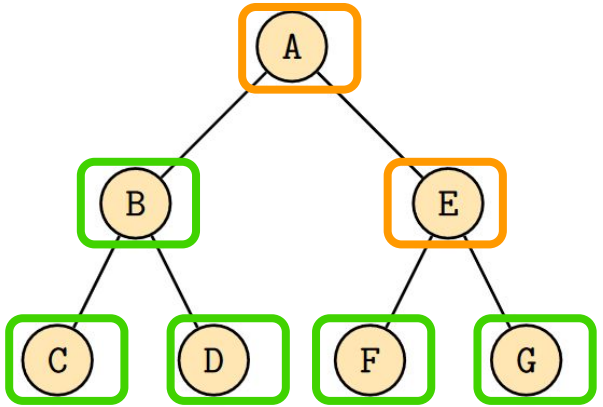


Sequenza: **A B C D E F G**

Stack: **A E G**

DFS: Pre-order

```
dfs(TREE t)
if t ≠ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    dfs(t.right())
```

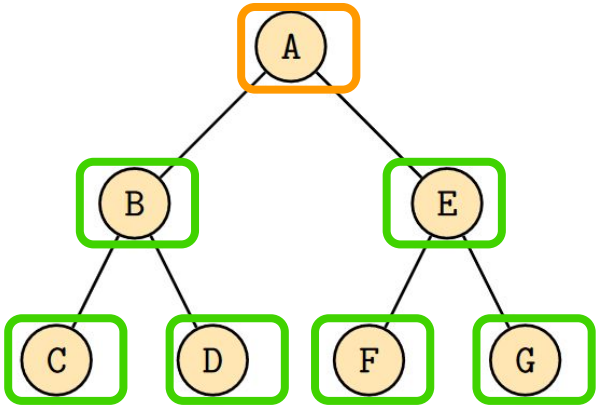


Sequenza: **A B C D E F G**

Stack: **A E**

DFS: Pre-order

```
dfs(TREE t)
if t ≠ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    dfs(t.right())
```

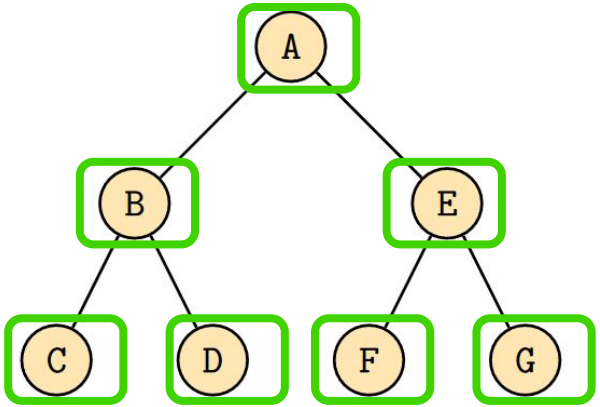


Sequenza: **A B C D E F G**

Stack: **A**

DFS: Pre-order

```
dfs(TREE t)
if t ≠ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    dfs(t.right())
```



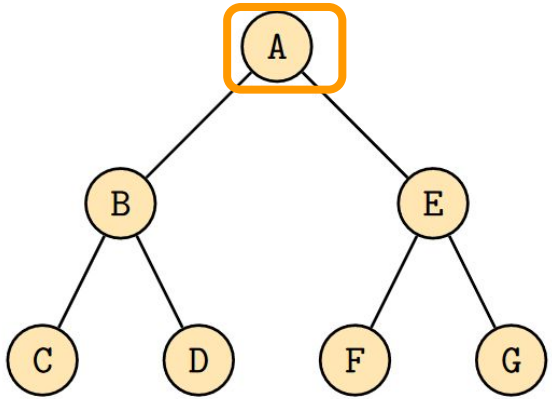
Sequenza: **A B C D E F G**

Stack:

...e restituisco al chiamante di dfs.

DFS: In-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
```

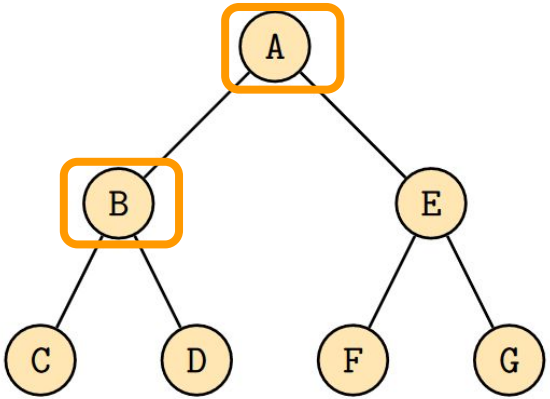


Sequenza:

Stack: **A**

DFS: In-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
```

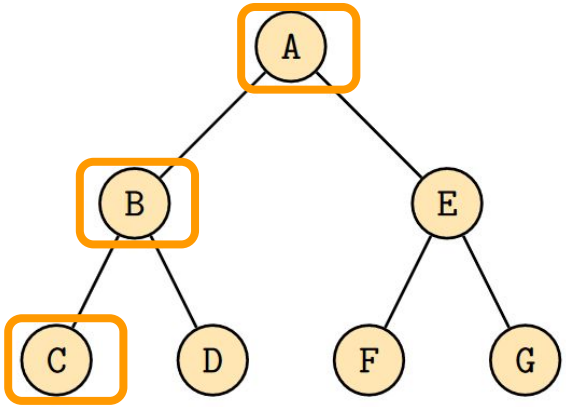


Sequenza:

Stack: **A** **B**

DFS: In-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
```

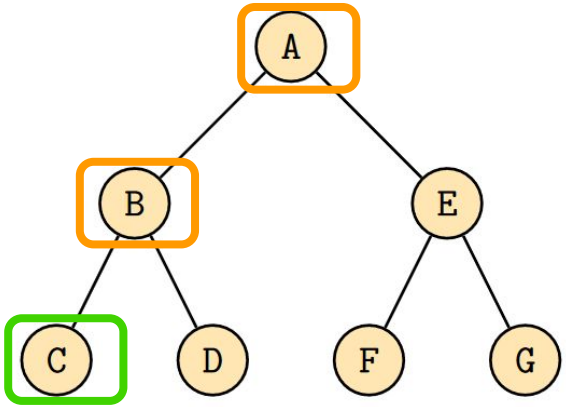


Sequenza: **C**

Stack: **A B C**

DFS: In-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
```

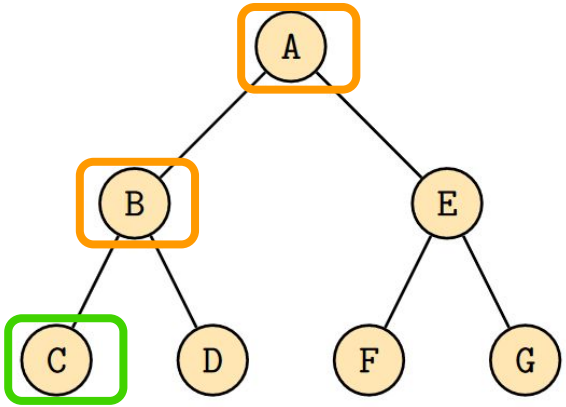


Sequenza: **C**

Stack: **A B**

DFS: In-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
```

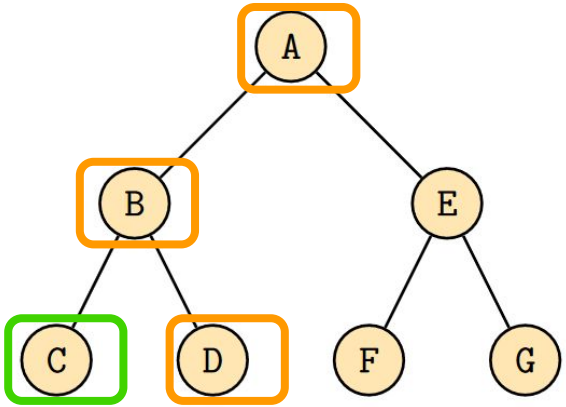


Sequenza: **C** **B**

Stack: **A** **B**

DFS: In-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
```

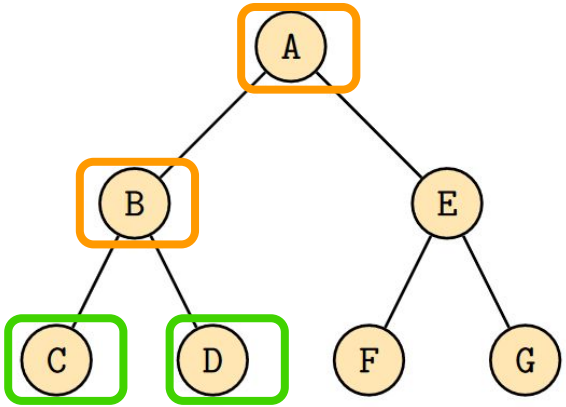


Sequenza: **C** **B** **D**

Stack: **A** **B** **D**

DFS: In-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
```

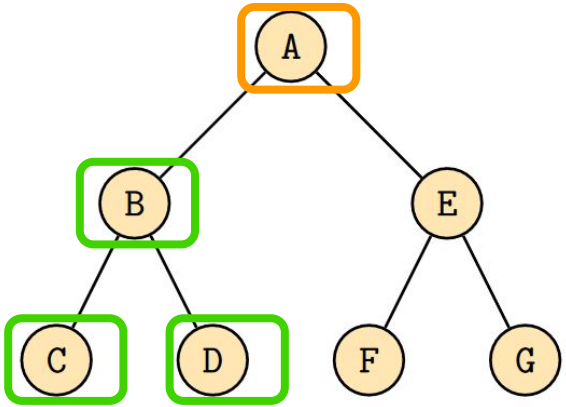


Sequenza: **C B D**

Stack: **A B**

DFS: In-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
```

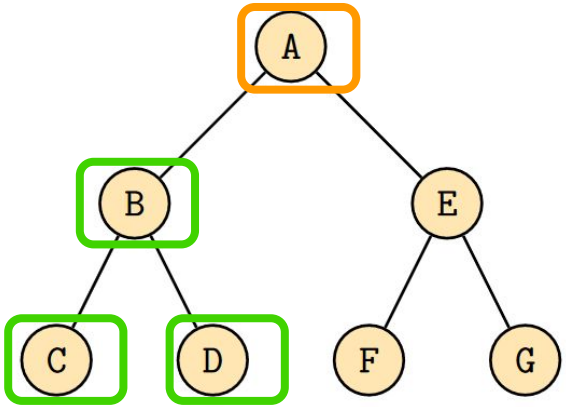


Sequenza: **C B D**

Stack: **A**

DFS: In-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
```

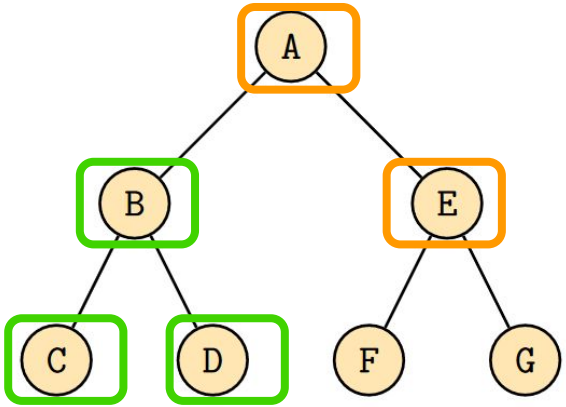


Sequenza: **C B D A**

Stack: **A**

DFS: In-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
```

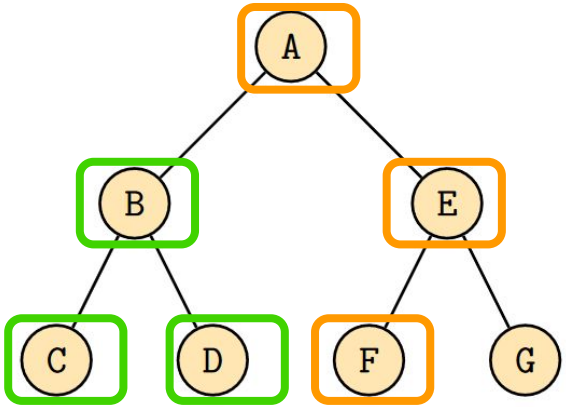


Sequenza: **C B D A**

Stack: **A E**

DFS: In-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
```

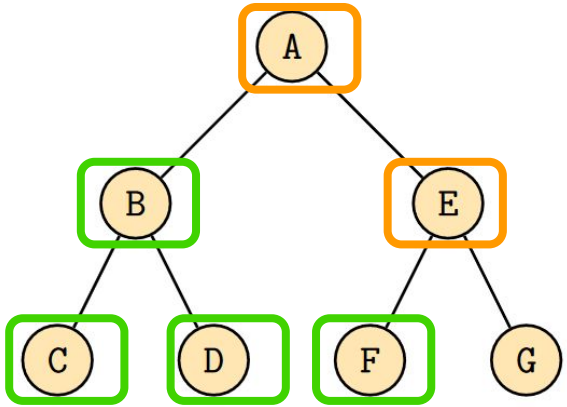


Sequenza: **C B D A F**

Stack: **A E F**

DFS: In-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
```

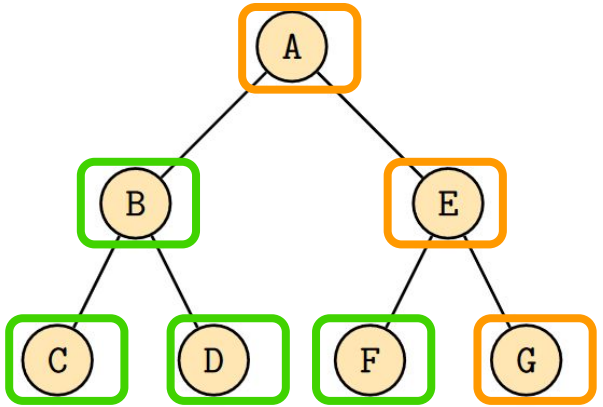


Sequenza: **C B D A F E**

Stack: **A E**

DFS: In-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
```

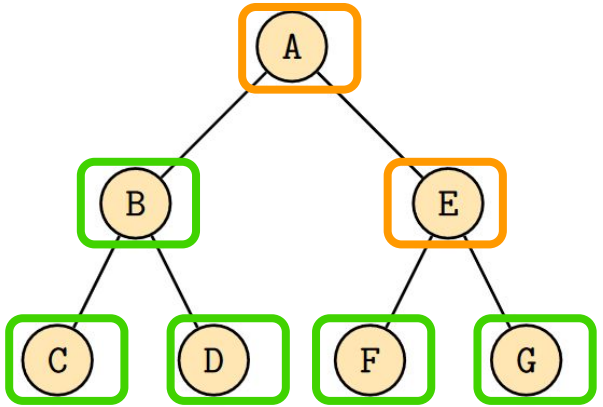


Sequenza: **C B D A F E G**

Stack: **A E G**

DFS: In-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
```

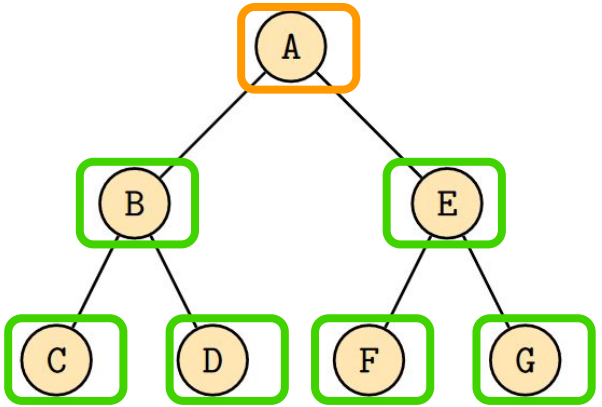


Sequenza: **C B D A F E G**

Stack: **A E**

DFS: In-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
```

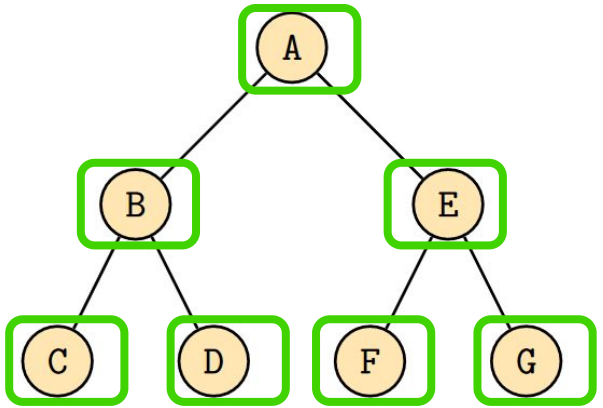


Sequenza: **C B D A F E G**

Stack: **A**

DFS: In-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
```



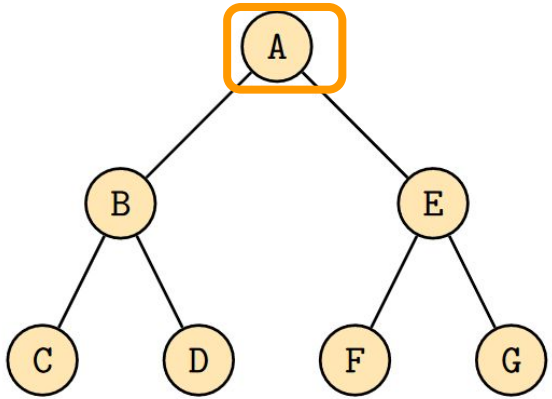
Sequenza: **C B D A F E G**

Stack:

...e restituisco al chiamante di dfs.

DFS: Post-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    dfs(t.right())
    % post-order visit of t
    print t
```

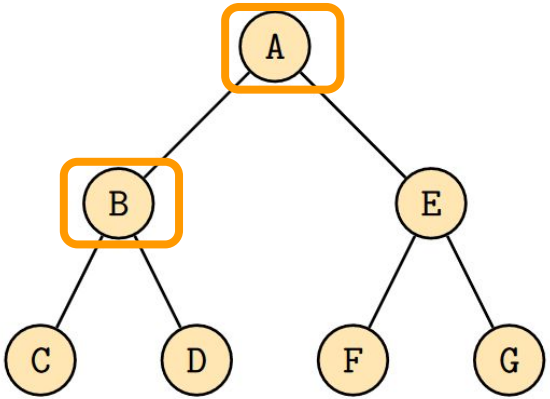


Sequenza:

Stack: **A**

DFS: Post-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    dfs(t.right())
    % post-order visit of t
    print t
```

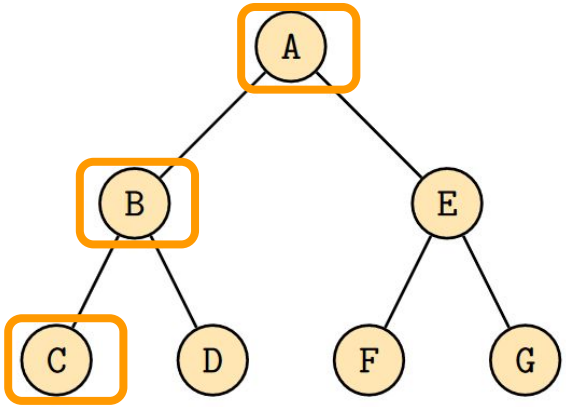


Sequenza:

Stack: **A** **B**

DFS: Post-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    dfs(t.right())
    % post-order visit of t
    print t
```

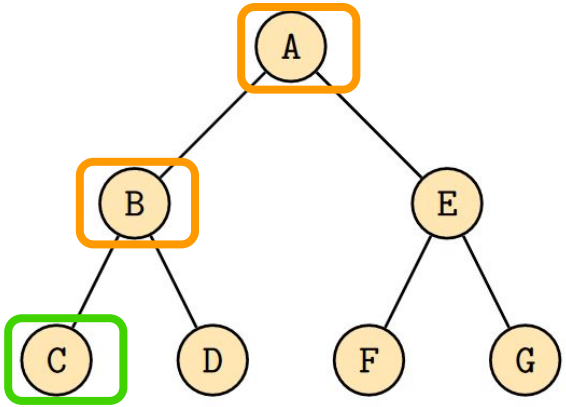


Sequenza: **C**

Stack: **A B C**

DFS: Post-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    dfs(t.right())
    % post-order visit of t
    print t
```

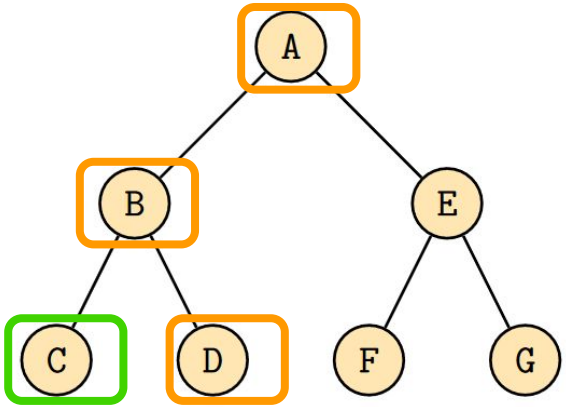


Sequenza: **C**

Stack: **A B**

DFS: Post-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    dfs(t.right())
    % post-order visit of t
    print t
```

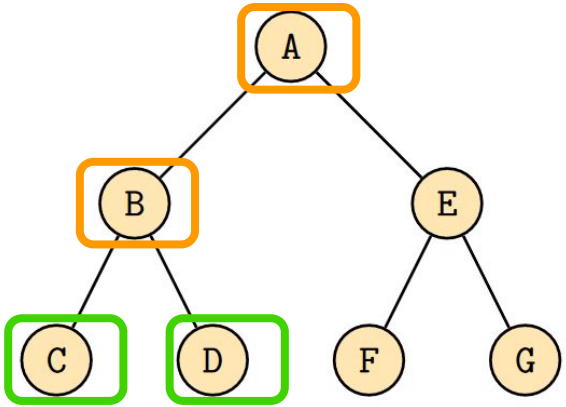


Sequenza: **C** **D**

Stack: **A** **B** **D**

DFS: Post-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    dfs(t.right())
    % post-order visit of t
    print t
```

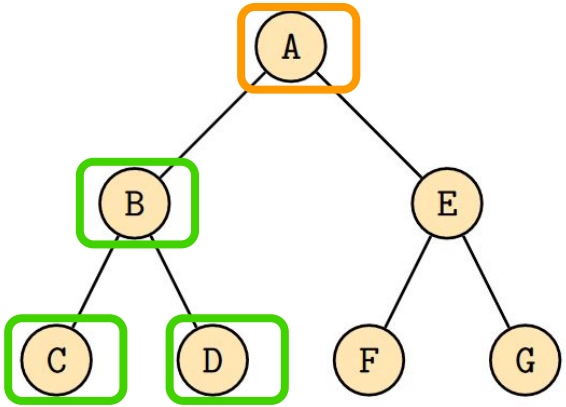


Sequenza: **C D**

Stack: **A B**

DFS: Post-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    dfs(t.right())
    % post-order visit of t
    print t
```

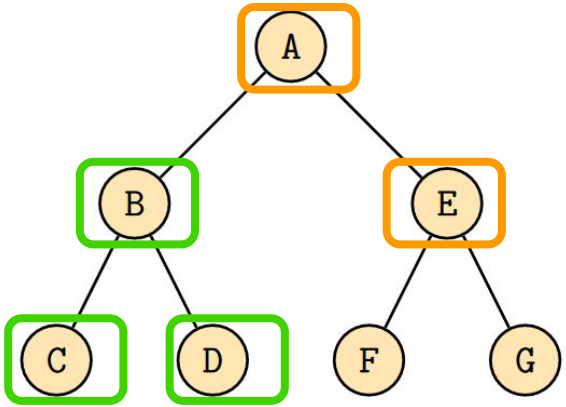


Sequenza: **C D B**

Stack: **A**

DFS: Post-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    dfs(t.right())
    % post-order visit of t
    print t
```

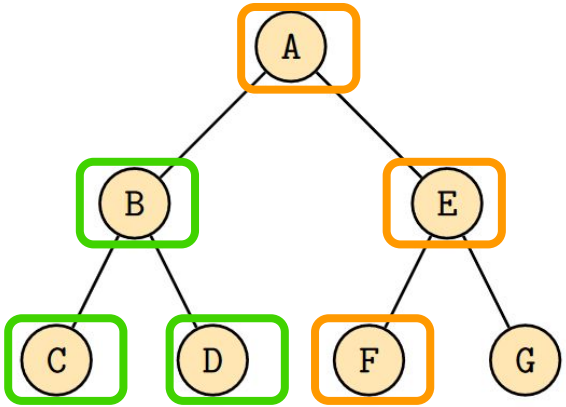


Sequenza: **C D B**

Stack: **A E**

DFS: Post-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    dfs(t.right())
    % post-order visit of t
    print t
```

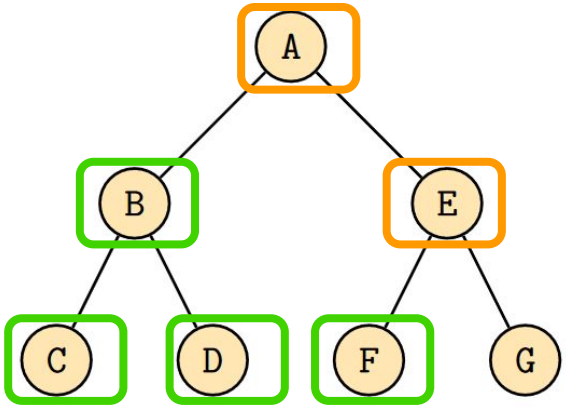


Sequenza: **C D B F**

Stack: **A E F**

DFS: Post-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    dfs(t.right())
    % post-order visit of t
    print t
```

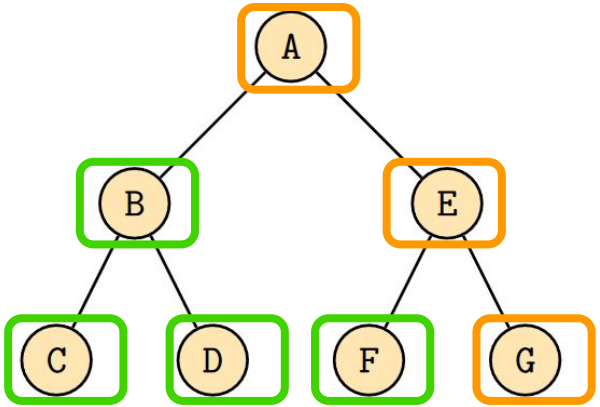


Sequenza: **C D B F**

Stack: **A E**

DFS: Post-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    dfs(t.right())
    % post-order visit of t
    print t
```

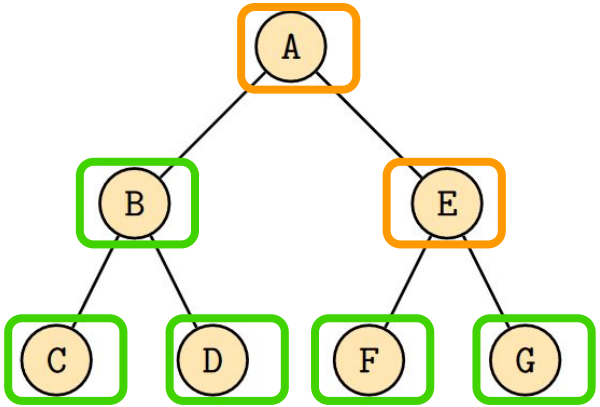


Sequenza: **C D B F G**

Stack: **A E G**

DFS: Post-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    dfs(t.right())
    % post-order visit of t
    print t
```

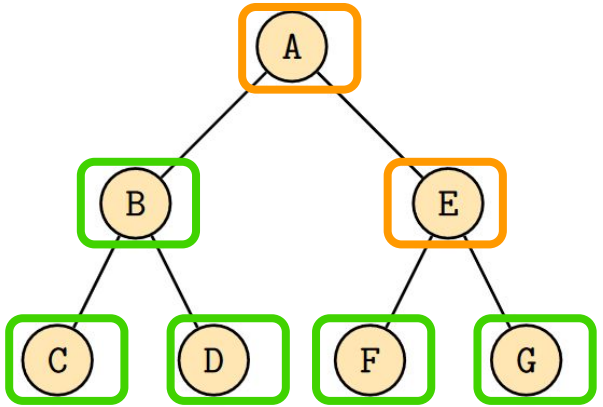


Sequenza: **C D B F G**

Stack: **A E**

DFS: Post-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    dfs(t.right())
    % post-order visit of t
    print t
```

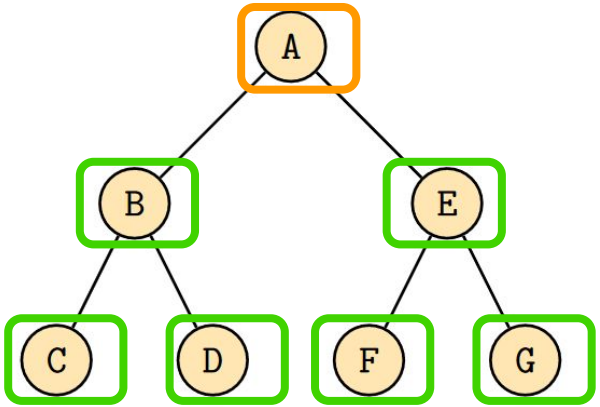


Sequenza: **C D B F G E**

Stack: **A E**

DFS: Post-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    dfs(t.right())
    % post-order visit of t
    print t
```

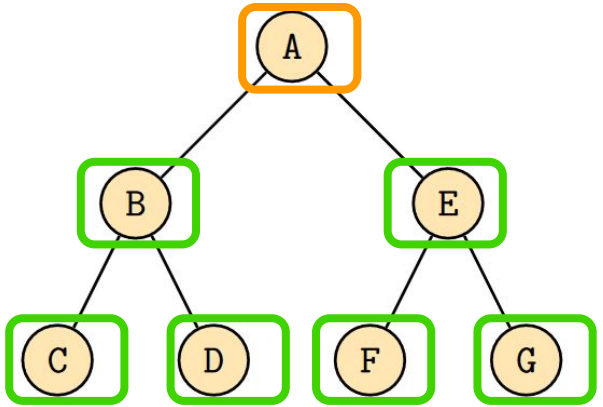


Sequenza: **C D B F G E**

Stack: **A**

DFS: Post-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    dfs(t.right())
    % post-order visit of t
    print t
```

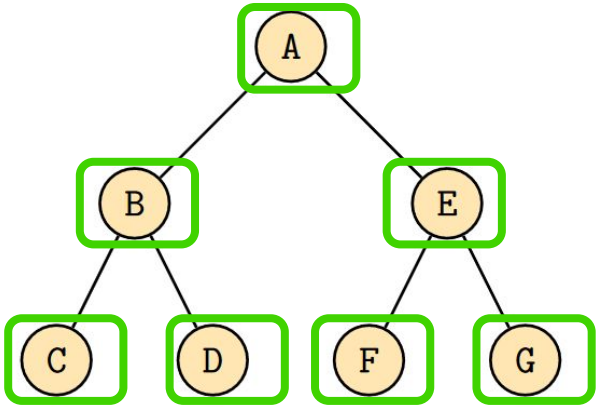


Sequenza: **C D B F G E A**

Stack: **A**

DFS: Post-order

```
dfs(TREE t)
if t ≠ nil then
    dfs(t.left())
    dfs(t.right())
    % post-order visit of t
    print t
```

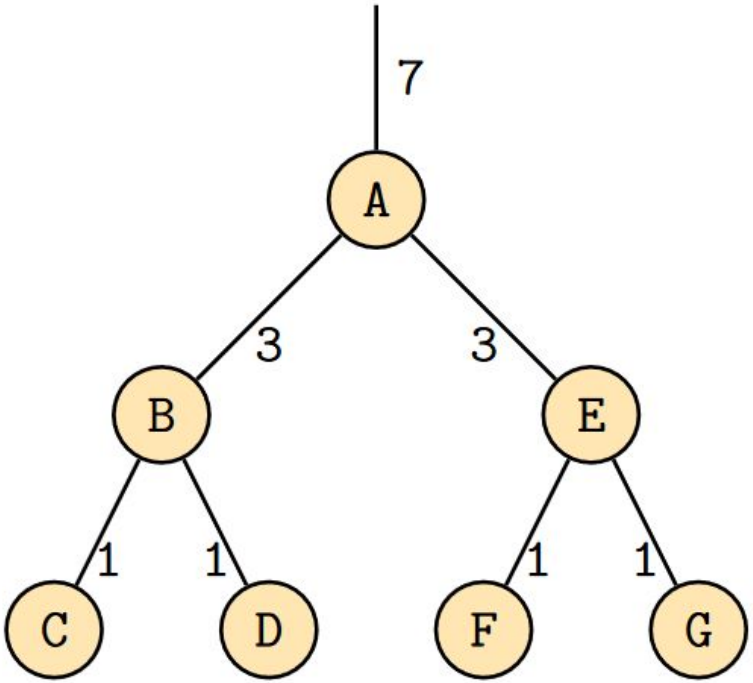


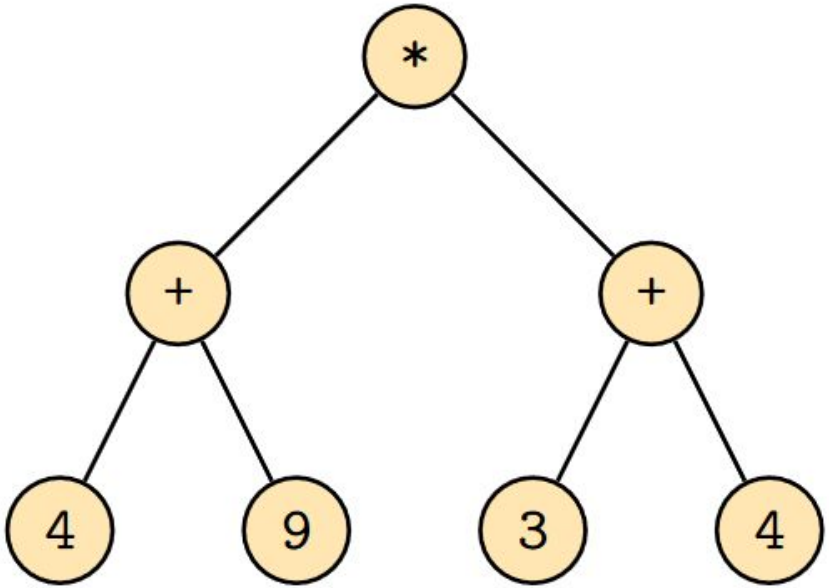
Sequenza: **C D B F G E A**

Stack:

...e restituisco al chiamante di dfs.

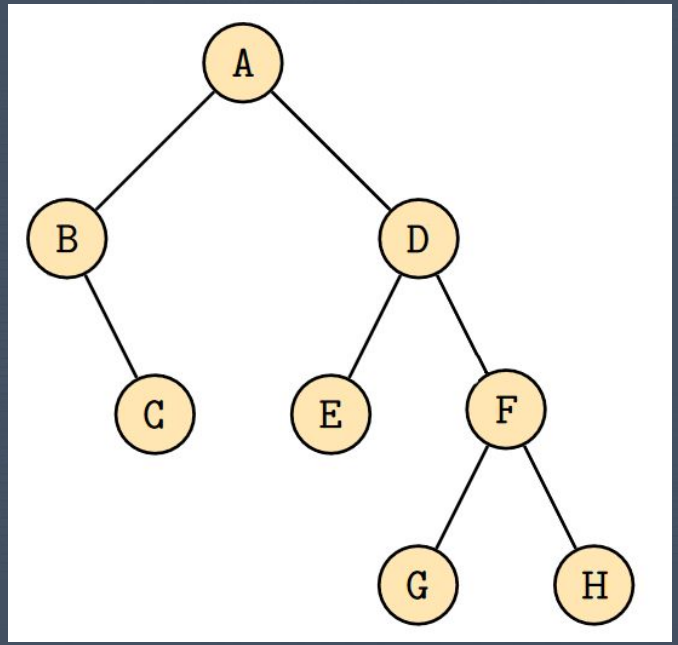
Esempi di applicazione





Esercizi

- Quali sequenze di nodi vengono stampate visitando nelle tre modalità il seguente albero?



Esercizi

Pre-order:

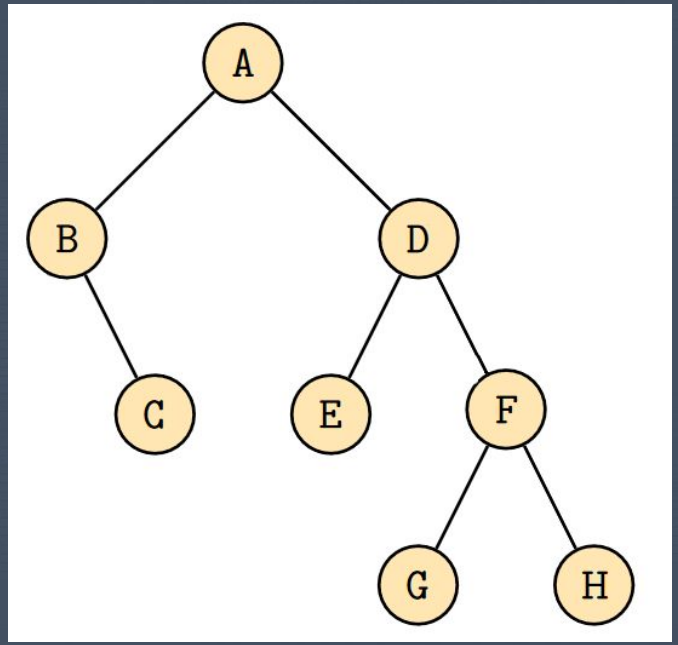
A B C D E F G H

In-order:

B C A E D G F H

Post-order:

C B E G H F D A



Esercizi

- Dato un albero binario, i cui nodi contengono elementi interi, si scriva una procedura per ottenere l'albero inverso, ovvero un albero in cui il figlio destro (con relativo sottoalbero) è scambiato con il figlio sinistro (con relativo sottoalbero).
- Dato un albero binario T, definiamo "altezza minimale" di un nodo v la minima distanza di v da una delle foglie del suo sottoalbero.
  Descrivere un algoritmo che riceve in input un nodo v e restituisce la sua altezza minimale, e calcolarne la complessità.

Alberi generici

Albero generico

Un albero generico è un albero radicato in cui ogni nodo ha un numero arbitrario di figli, o nessuno.

Albero generico, la specifica

Tree (Item v)	Costruisce un nuovo nodo, contenente v, senza figli o genitori
Item read()	Legge il valore memorizzato nel nodo
write (Item v)	Modifica il valore memorizzato nel nodo
Tree parent()	Restituisce il padre, oppure **nil** se questo nodo è radice

La specifica delle operazioni sul singolo nodo è uguale a quella per gli alberi binari.

Albero generico, la specifica

Tree leftmostChild()	Restituisce il primo figlio, oppure **nil** se questo nodo è una foglia
Tree rightSibling()	Restituisce il prossimo fratello, oppure **nil** se assente
insertChild (Tree t) insertSibling (Tree t)	Inserisce il sottoalbero t come primo nodo (o come prossimo fratello) di questo nodo
deleteChild ()	Distruggi l'albero radicato identificato dal primo figlio
deleteSibling ()	Distruggi l'albero radicato identificato dal prossimo fratello

Visita di un albero generico

```
dfs(TREE t)
if t ≠ nil then
    % pre-order visit of node t
    print t
    TREE u = t.leftmostChild()
    while u ≠ nil do
        dfs(u)
        u = u.rightSibling()
```

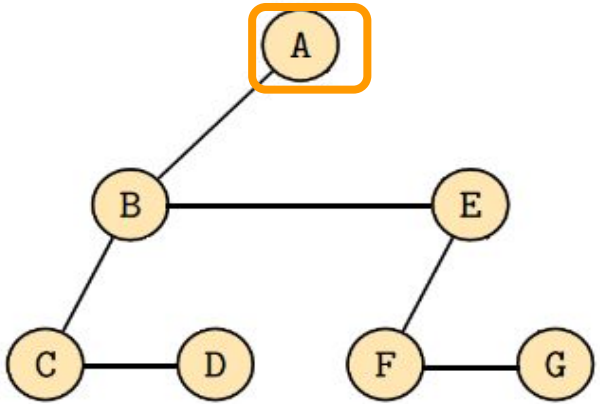
```
dfs(TREE t)
if t ≠ nil then
    TREE u = t.leftmostChild()
    while u ≠ nil do
        dfs(u)
        u = u.rightSibling()
    % post-order visit of node t
    print t
```

Le visite pre-order e post-order si adattano in pochi passaggi.

Ma quella in-order?

Breadth-First Search (BFS)

```
bfs(TREE t)
QUEUE Q = Queue()
Q.enqueue(t)
while not Q.isEmpty() do
    TREE u = Q.dequeue()
    % visita per livelli nodo u
    print u
    u = u.leftmostChild()
    while u ≠ nil do
        Q.enqueue(u)
        u = u.rightSibling()
```

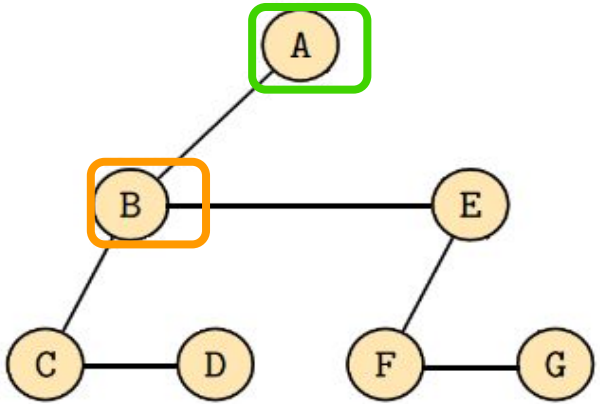


Seq.:

Queue: A

Breadth-First Search (BFS)

```
bfs(TREE t)
QUEUE Q = Queue()
Q.enqueue(t)
while not Q.isEmpty() do
    TREE u = Q.dequeue()
    % visita per livelli nodo u
    print u
    u = u.leftmostChild()
    while u ≠ nil do
        Q.enqueue(u)
        u = u.rightSibling()
```

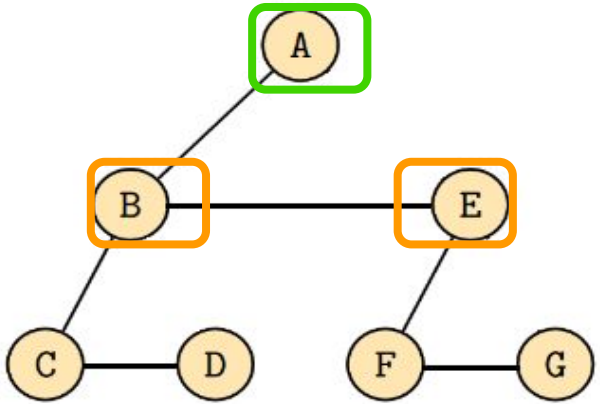


Seq.: A

Queue: B

Breadth-First Search (BFS)

```
bfs(TREE t)
QUEUE Q = Queue()
Q.enqueue(t)
while not Q.isEmpty() do
    TREE u = Q.dequeue()
    % visita per livelli nodo u
    print u
    u = u.leftmostChild()
    while u ≠ nil do
        Q.enqueue(u)
        u = u.rightSibling()
```

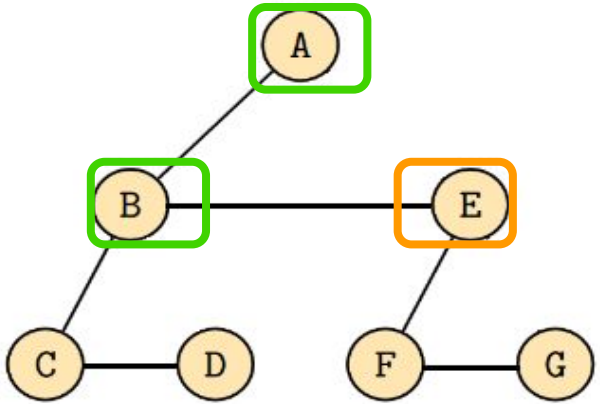


Seq.: A

Queue: B E

Breadth-First Search (BFS)

```
bfs(TREE t)
QUEUE Q = Queue()
Q.enqueue(t)
while not Q.isEmpty() do
    TREE u = Q.dequeue()
    % visita per livelli nodo u
    print u
    u = u.leftmostChild()
    while u ≠ nil do
        Q.enqueue(u)
        u = u.rightSibling()
```

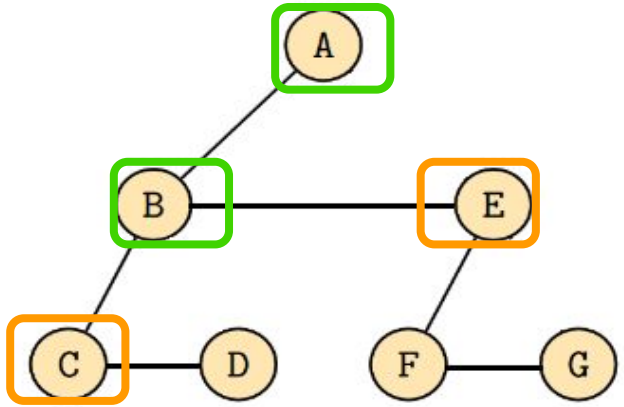


Seq.: A B

Queue: E

Breadth-First Search (BFS)

```
bfs(TREE t)
QUEUE Q = Queue()
Q.enqueue(t)
while not Q.isEmpty() do
    TREE u = Q.dequeue()
    % visita per livelli nodo u
    print u
    u = u.leftmostChild()
    while u ≠ nil do
        Q.enqueue(u)
        u = u.rightSibling()
```

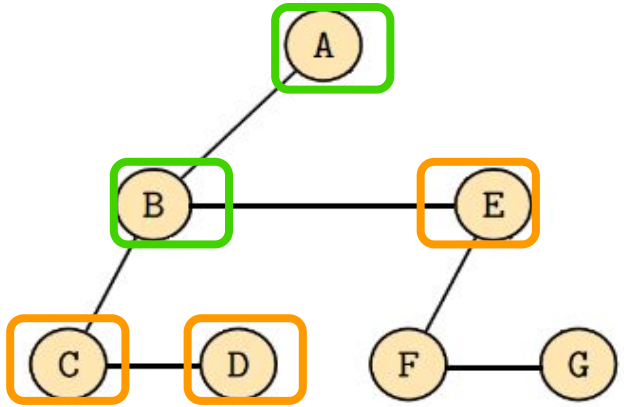


Seq.: A B

Queue: E C

Breadth-First Search (BFS)

```
bfs(TREE t)
QUEUE Q = Queue()
Q.enqueue(t)
while not Q.isEmpty() do
    TREE u = Q.dequeue()
    % visita per livelli nodo u
    print u
    u = u.leftmostChild()
    while u ≠ nil do
        Q.enqueue(u)
        u = u.rightSibling()
```

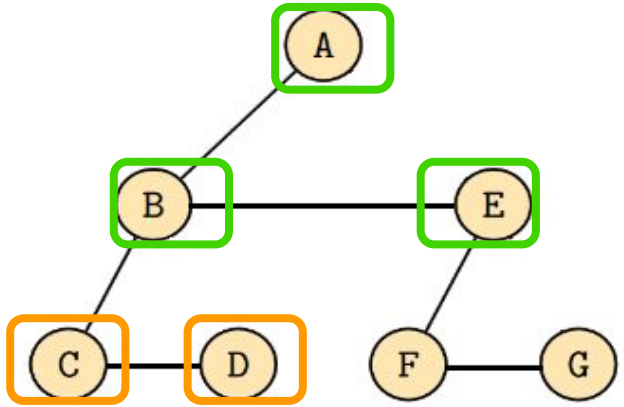


Seq.: A B

Queue: E C D

Breadth-First Search (BFS)

```
bfs(TREE t)
QUEUE Q = Queue()
Q.enqueue(t)
while not Q.isEmpty() do
    TREE u = Q.dequeue()
    % visita per livelli nodo u
    print u
    u = u.leftmostChild()
    while u ≠ nil do
        Q.enqueue(u)
        u = u.rightSibling()
```

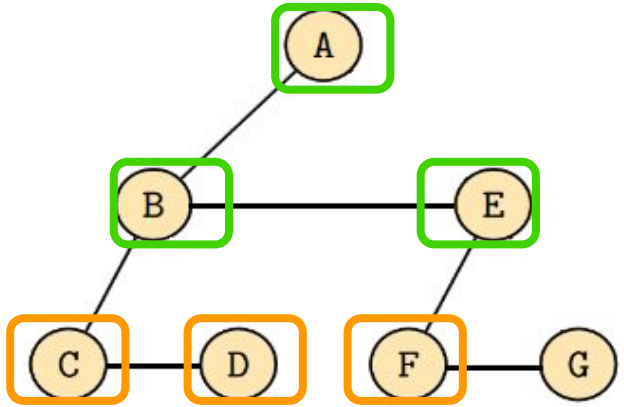


Seq.: A B E

Queue: C D

Breadth-First Search (BFS)

```
bfs(TREE t)
QUEUE Q = Queue()
Q.enqueue(t)
while not Q.isEmpty() do
    TREE u = Q.dequeue()
    % visita per livelli nodo u
    print u
    u = u.leftmostChild()
    while u ≠ nil do
        Q.enqueue(u)
        u = u.rightSibling()
```

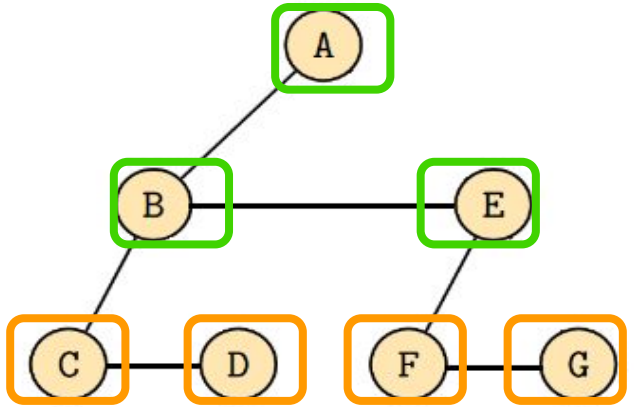


Seq.: A B E

Queue: C D F

Breadth-First Search (BFS)

```
bfs(TREE t)
QUEUE Q = Queue()
Q.enqueue(t)
while not Q.isEmpty() do
    TREE u = Q.dequeue()
    % visita per livelli nodo u
    print u
    u = u.leftmostChild()
    while u ≠ nil do
        Q.enqueue(u)
        u = u.rightSibling()
```

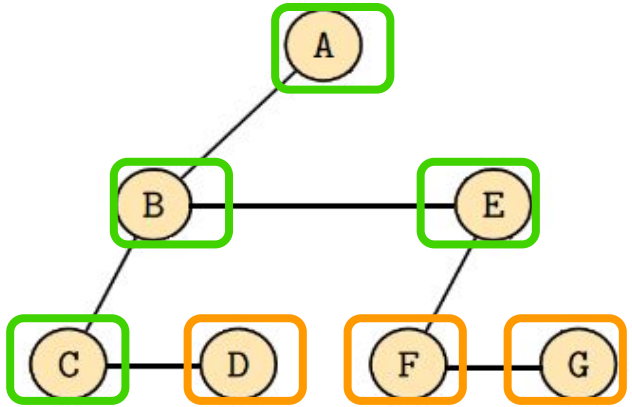


Seq.: A B E

Queue: C D F G

Breadth-First Search (BFS)

```
bfs(TREE t)
QUEUE Q = Queue()
Q.enqueue(t)
while not Q.isEmpty() do
    TREE u = Q.dequeue()
    % visita per livelli nodo u
    print u
    u = u.leftmostChild()
    while u ≠ nil do
        Q.enqueue(u)
        u = u.rightSibling()
```

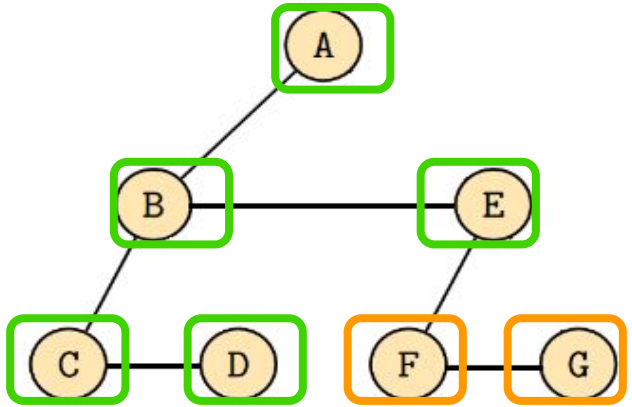


Seq.: A B E C

Queue: D F G

Breadth-First Search (BFS)

```
bfs(TREE t)
QUEUE Q = Queue()
Q.enqueue(t)
while not Q.isEmpty() do
    TREE u = Q.dequeue()
    % visita per livelli nodo u
    print u
    u = u.leftmostChild()
    while u ≠ nil do
        Q.enqueue(u)
        u = u.rightSibling()
```

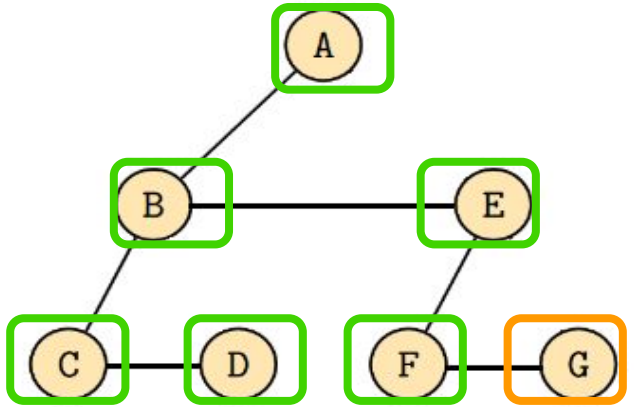


Seq.: A B E C D

Queue: F G

Breadth-First Search (BFS)

```
bfs(TREE t)
QUEUE Q = Queue()
Q.enqueue(t)
while not Q.isEmpty() do
    TREE u = Q.dequeue()
    % visita per livelli nodo u
    print u
    u = u.leftmostChild()
    while u ≠ nil do
        Q.enqueue(u)
        u = u.rightSibling()
```

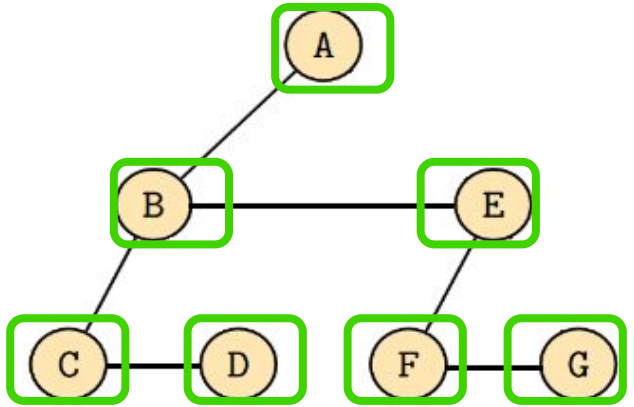


Seq.: A B E C D F

Queue: G

Breadth-First Search (BFS)

```
bfs(TREE t)
QUEUE Q = Queue()
Q.enqueue(t)
while not Q.isEmpty() do
    TREE u = Q.dequeue()
    % visita per livelli nodo u
    print u
    u = u.leftmostChild()
    while u ≠ nil do
        Q.enqueue(u)
        u = u.rightSibling()
```



Seq.: A B E C D F G

Queue:

Costo computazionale delle visite

Potremmo fare il calcolo a mano, trovando l'equazione di ricorrenza (per DFS) o le somme (per BFS), ma possiamo anche ragionare così: visto che ogni nodo viene visitato al massimo una volta (se l'algoritmo è implementato correttamente), il costo di una visita di un albero contenente n nodi è…

Costo computazionale delle visite

Potremmo fare il calcolo a mano, trovando l'equazione di ricorrenza (per DFS) o le somme (per BFS), ma possiamo anche ragionare così: visto che ogni nodo viene visitato al massimo una volta (se l'algoritmo è implementato correttamente), il costo di una visita di un albero contenente n nodi è $\Theta(n)$.

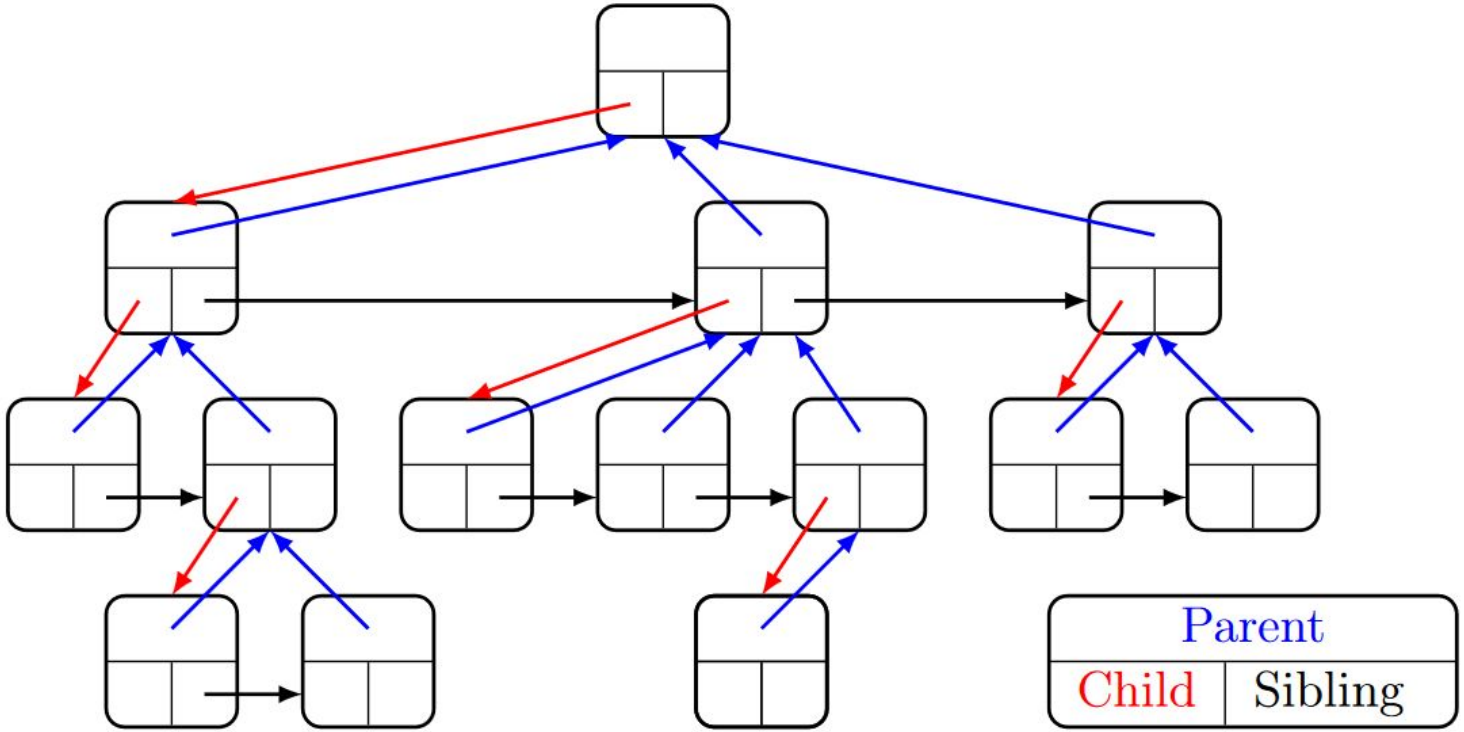
Memorizzazione degli alberi

Memorizzazione degli alberi

Esistono diversi modi per memorizzare un albero a seconda del numero massimo e medio di figli presenti.

- Realizzazione primo figlio, prossimo fratello
  - Se ogni nodo può avere un numero arbitrario di figli
- Realizzazione con vettore dei figli
  - Se so mediamente quanti figli ci sono per ogni nodo
- Realizzazione con vettore dei padri
  - Se mi interessa mantenere solo la relazione Figlio ⟶ Padre

Primo figlio, prossimo fratello

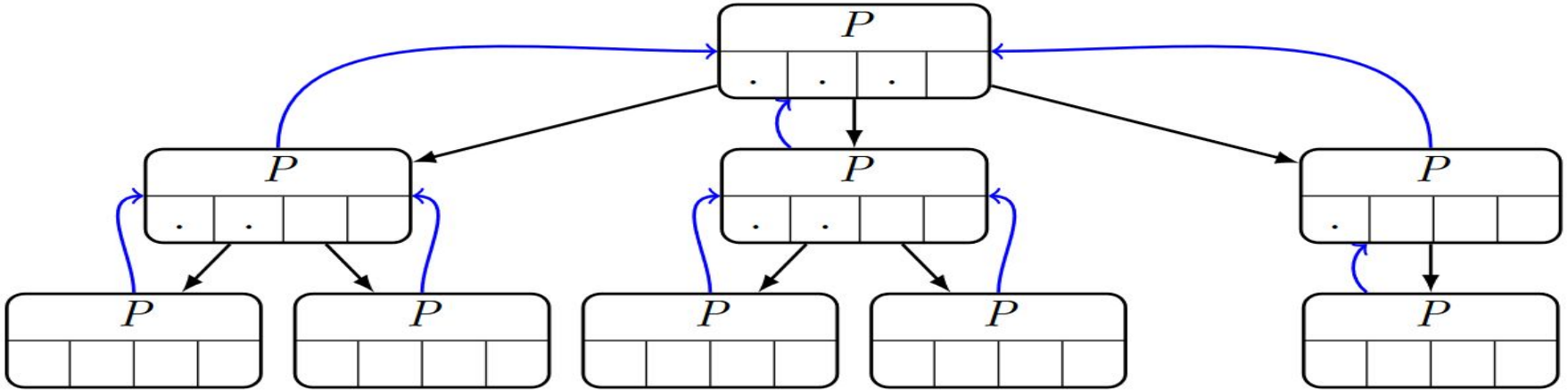


Implementazione

TREE

```
TREE parent
TREE child
TREE sibling
ITEM value

Tree(ITEM v)
    TREE t = new TREE
    t.value = v
    t.parent = t.child = t.sibling = nil
    return t

insertChild(TREE t)
    t.parent = self
    t.sibling = child
    child = t

insertSibling(TREE t)
    t.parent = parent
    t.sibling = sibling
    sibling = t

deleteChild()
    TREE newChild = child.rightSibling()
    delete(child)
    child = newChild

deleteSibling()
    TREE newBrother = sibling.rightSibling()
    delete(sibling)
    sibling = newBrother

delete(TREE t)
    TREE u = t.leftmostChild()
    while u ≠ nil do
        TREE next = u.rightSibling()
        delete(u)
        u = next
    delete t
```

Vettore dei figli

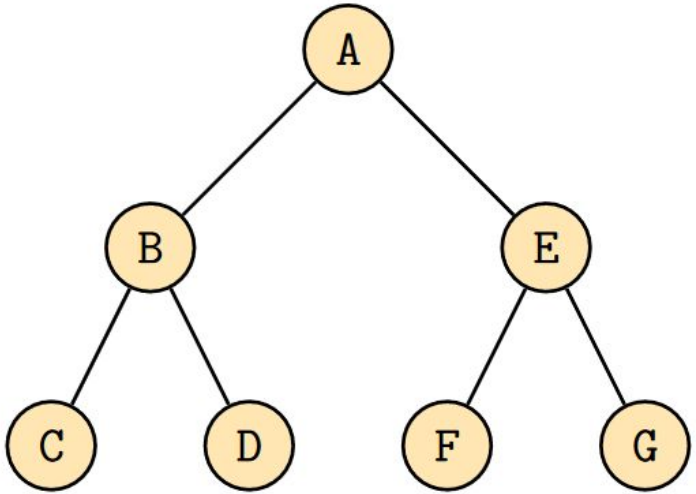


Campi memorizzati nei nodi

- parent: riferimento al nodo padre
- vettore dei figli: a seconda del numero di figli, può comportare una discreta quantità di spazio sprecato

Vettore dei padri

	Valore	Pos. padre
1	A	0/1
2	B	1
3	E	1
4	C	2
5	D	2
6	F	3
7	G	3

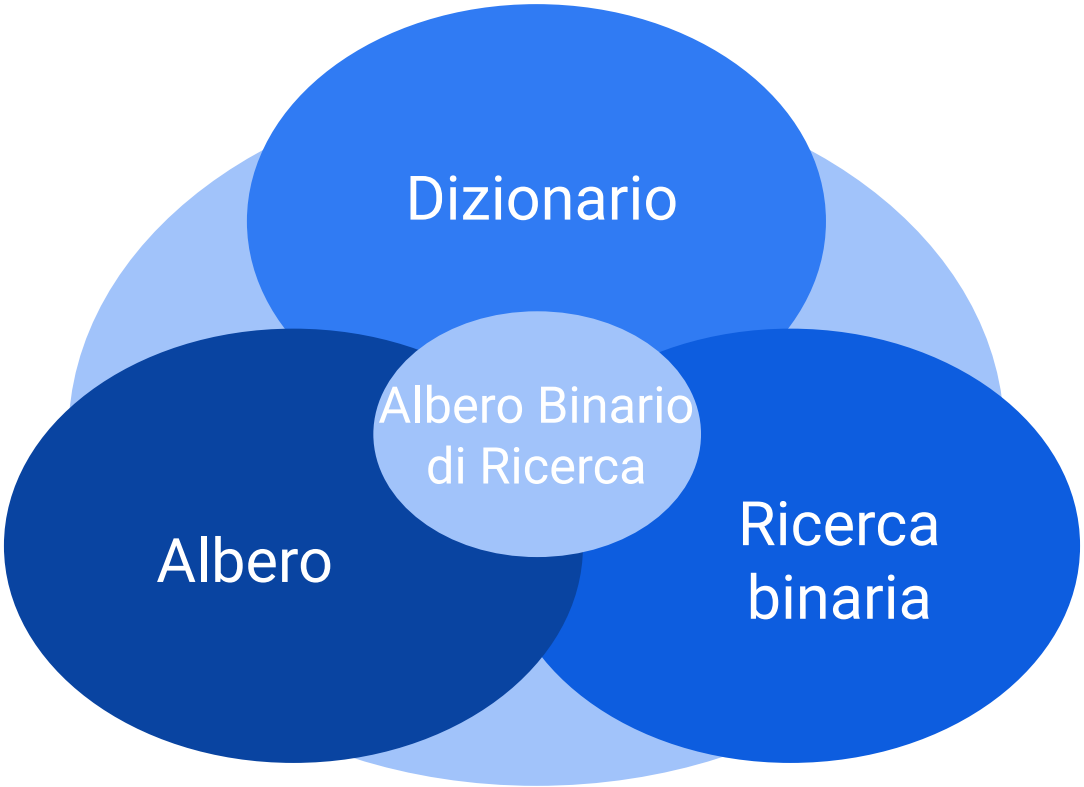


Cenni velocissimi
sugli alberi binari di ricerca (ABR, o BST)

Dizionario - Ripasso della specifica

Item lookup (Item k)	Restituisce il valore associato alla chiave k se presente, **nil** altrimenti
insert (Item k, Item v)	Associa il valore v alla chiave k
remove (Item k)	Rimuove l'associazione della chiave k

Possibili implementazioni			
Struttura dati	lookup	insert	remove
Vettore ordinato	O(logn)	O(n)	O(n)
Vettore non ordinato	O(n)	O(1)*	O(1)*
Lista non ordinata	O(n)	O(1)*	O(1)*

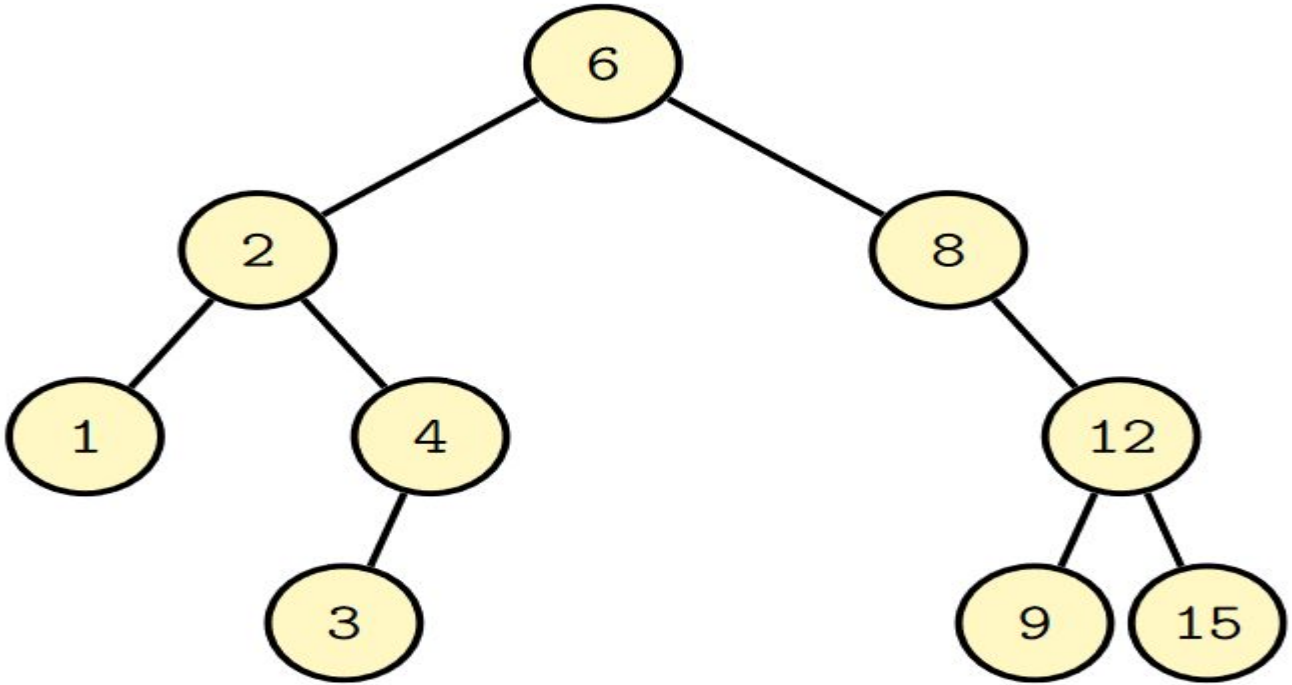

Dizionario
Albero Binario di Ricerca
Albero
Ricerca binaria

Alberi binari di ricerca (ABR)

L’idea è quella di portare la ricerca binaria negli alberi.

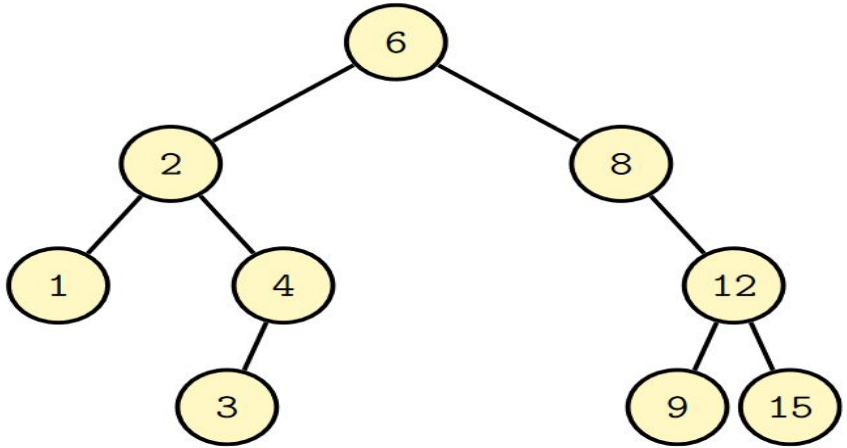
Le associazioni chiave-valore di un dizionario vengono memorizzate in un albero binario, dove ogni nodo *u* contiene una coppia (*u.key*, *u.value*). Le chiavi devono appartenere ad un insieme totalmente ordinato, cioè devono essere sempre confrontabili.

Esempio di ABR



La ricerca all'interno di un ABR

```
TREE lookupNode(TREE T, ITEM k)
TREE u = T
while u ≠ nil and u.key ≠ k do
    if k < u.key then
        u = u.left
    else
        u = u.right
return u
```



Costo computazionale degli ABR

Tutte le operazioni sono confinate all'interno di un cammino semplice dalla radice dell'albero fino ad una foglia. Definiamo h l'altezza dell'albero, ossia il numero di nodi che devo attraversare dalla radice alla foglia "più bassa".

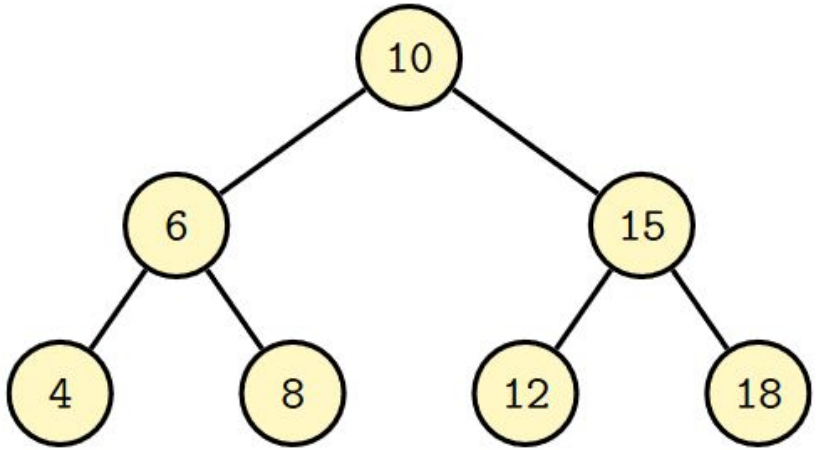
Il costo computazionale della ricerca è, in generale, $O(h)$, ossia più l'albero è alto più tempo ci vuole per raggiungerne il fondo (le foglie). La domanda centrale è:

che relazione esiste tra h e n (il numero di nodi dell'albero)?

Caso ottimo

Nel caso ottimo l'albero è perfettamente bilanciato, ossia ho lo stesso numero di nodi a destra e a sinistra della radice.

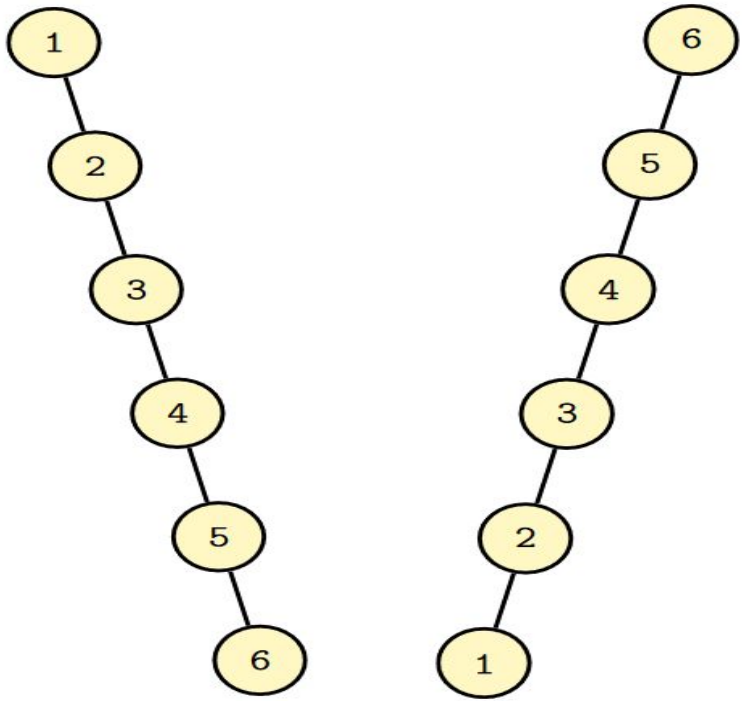
L'altezza h è quindi $\log n$ e il costo computazionale della ricerca è $O(\log n)$



Caso pessimo

Nel caso pessimo l'albero è assolutamente sbilanciato, con tutti i nodi a destra o a sinistra della radice.

L'altezza h è quindi n e il costo computazionale della ricerca è $O(n)$, equivalente ad una ricerca lineare.



Caso medio

Il caso “medio” si può suddividere in due categorie:

- **Caso medio “semplice”**: gli inserimenti avvengono in ordine casuale; è possibile dimostrare che l’altezza media dell’albero così generato è $O(\log n)$;
- **Caso medio “generale”**: se non ci sono vincoli per quanto riguarda l’ordine di esecuzione di inserimenti e cancellazioni diventa molto difficile da trattare.

Nella realtà non ci si affida al caso, ma si utilizzano tecniche per mantenere l’albero bilanciato o, meglio, per ri-bilanciarlo.

Quali tecniche esistono?

- Alberi AVL (Adelson-Velskii e Landis, 1962)
- B-Alberi (Bayer, McCreight, 1972)
- Alberi 2-3 (Hopcroft, 1983)
- Alberi Red-Black (Guibas and Sedgewick, 1978)
  - Complessità di insert: $O(\log n)$
  - Complessità di delete: $O(\log n)$

Esercizi

- La larghezza di un albero radicato è il numero massimo di nodi che stanno tutti al medesimo livello. Ideare un algoritmo che calcoli la larghezza di un albero T composto di n nodi e stimarne la complessità.
- Gli ordini di visita di un albero binario di 9 nodi sono i seguenti:
  - A, E, B, F, G, C, D, I, H (pre-order)
  - B, G, C, F, E, H, I, D, A (post-order)
  - B, E, G, F, C, A, D, H, I (in-order).

Ricostruite la struttura dell'albero binario.