Algoritmi e Strutture Dati

Alberi

Alberto Montresor and Davide Rossi

Università di Bologna

19 settembre 2024

Sommario

Albero radicato – Definizione 1

Albero radicato (Rooted tree)

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

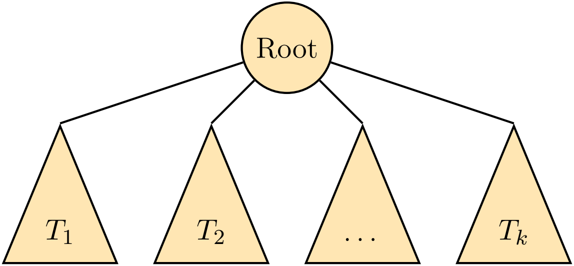
- Un nodo dell'albero è designato come nodo radice;
- Ogni nodo n, a parte la radice, ha esattamente un arco entrante;
- Esiste un cammino unico dalla radice ad ogni nodo;
- L'albero è connesso.

Albero radicato – Definizione 2 (Ricorsiva)

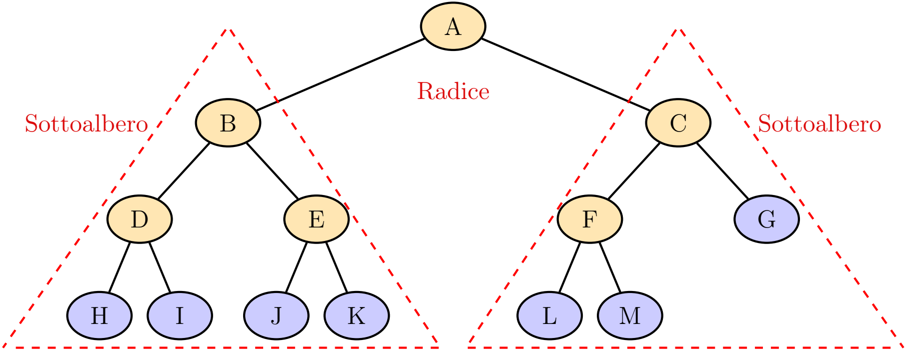
Albero radicato (Rooted tree)

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.

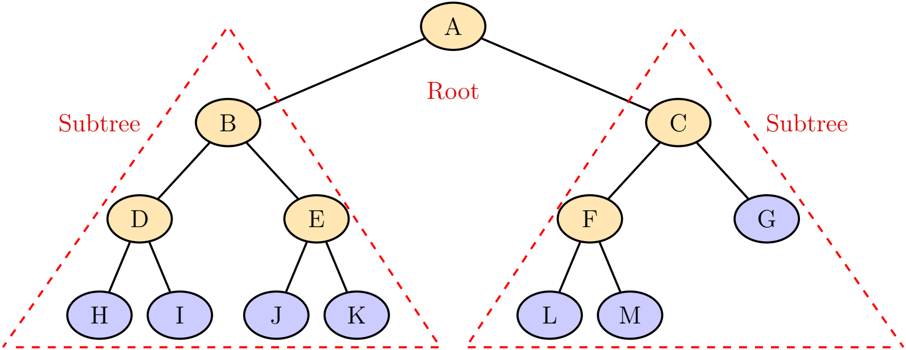


Terminologia



- 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

Terminology (English)



- A is the tree root
- B, C are roots of their subtrees
- D, E are siblings

fratelli

figli

- D, E are children of B
- B is the parent of D, E

- Purple nodes are leaves
- The other nodes are internal nodes

Terminologia

Profondità nodi (Depth)

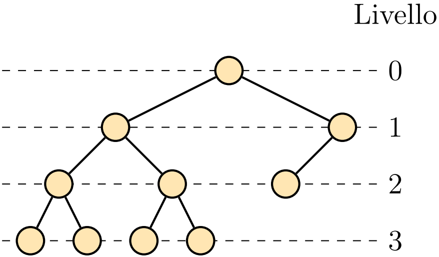
La lunghezza del cammino semplice dalla radice al nodo (misurato in numero di archi)

Livello (Level)

L'insieme di nodi alla stessa profondità

Altezza albero (Height)

La profondità massima della sue foglie



Altezza di questo albero = 3

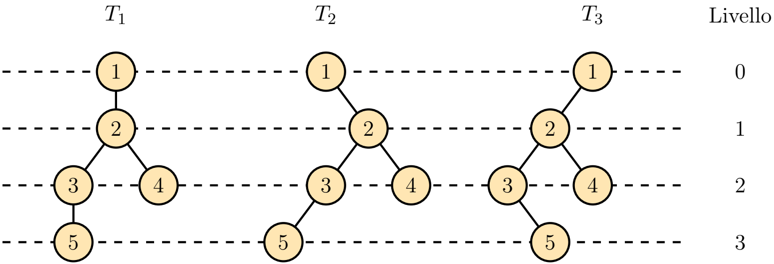
Sommario

Albero binario

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 u sia designato come figlio sinistro di v in T e come figlio destro di v in U.*



Specifica (Albero binario)

```
TREE
% Costruisce un nuovo nodo, contenente v, senza figli o genitori
Tree(ITEM v)

% Legge il valore memorizzato nel nodo
ITEM read()

% Modifica il valore memorizzato nel nodo
write(ITEM v)

% Restituisce il padre, oppure nil se questo nodo è radice
TREE parent()
```

Specifica (Albero binario)

TREE

% Restituisce il figlio sinistro (destro) di questo nodo; restituisce **nil** se assente

TREE left()

TREE right()

% Inserisce il sottoalbero radicato in t come figlio sinistro (destro) di questo nodo

insertLeft(TREE t)

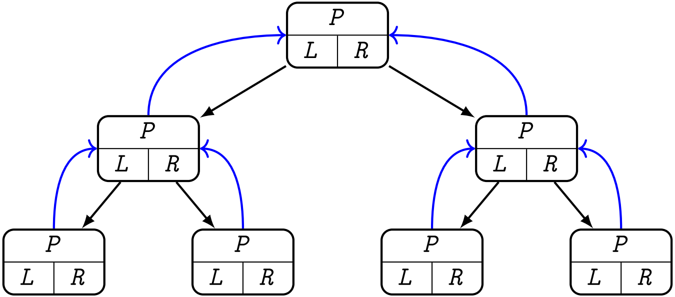
insertRight(TREE t)

% Distrugge (ricorsivamente) il figlio sinistro (destro) di questo nodo

deleteLeft()

deleteRight()

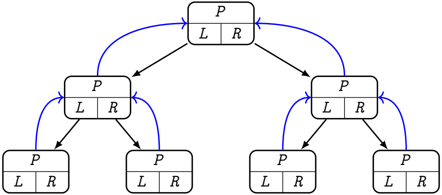
Memorizzare un albero binario



Campi memorizzati nei nodi

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

Implementazione



TREE

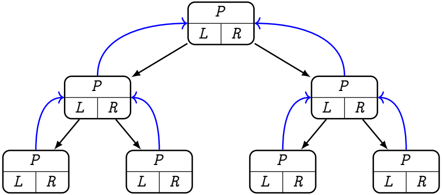
```
Tree(ITEM $v$)
    TREE $t$ = new TREE
    $t.parent$ = nil
    $t.left = t.right$ = nil
    $t.value = v$
    return $t$

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

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

Inizializza l'albero
↓
crea la radice (create root)

Ipotizzo che i metodi:
insertLeft, insertright, deleteleft, deleteright
funzionano solo quando fanno riferimento
a un nodo

aggiunge se il nodo non ha figli
sinistri, un sottoalbero T come figlio
sinistro (stessa cosa per dx)

Implementazione



```
TREE
deleteLeft()
    if $left \neq$ nil then
        $left$.deleteLeft()
        $left$.deleteRight()
        delete $left$
        $left$ = nil

deleteRight()
    if $right \neq$ nil then
        $right$.deleteLeft()
        $right$.deleteRight()
        delete $right$
        $right$ = nil
```

Visite di alberi

Visita di un albero / ricerca

Una strategia per analizzare (visitare) tutti i nodi di un albero.

Visità in profondità
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

Visita in ampiezza
Breadth First Search (BFS)

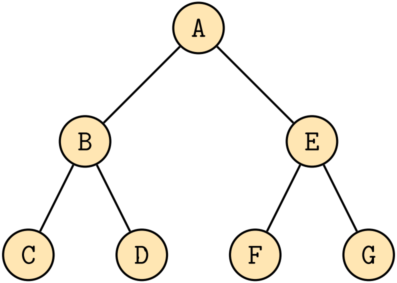
- Ogni livello dell'albero viene visitato, uno dopo l'altro
- Si parte dalla radice
- Richiede una queue .

queue → FIFO

stack → LIFO

Depth-First Search (DFS)

```
dfs(TREE t)
if $t \neq$ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
    % post-order visit of t
    print t
```



print t = A

dfs(t.left()) -> print t = B

print t = B

dfs(t.r())

dfs(t.l()) -> prt = C

prt = B

dfs(t.r())

prt = B

dfs(t.l()) = ϕ

prt = C

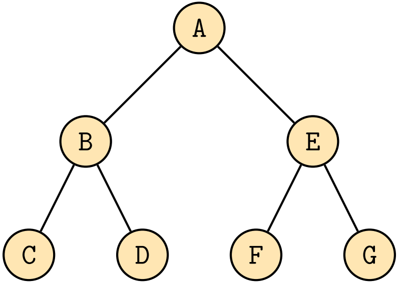
dfs(t.r()) = ϕ

prt = C

-> uguale a left ma con prt = D

Depth-First Search - Pre-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    % pre-order visit of $t$
    print $t$
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```



Sequence: A

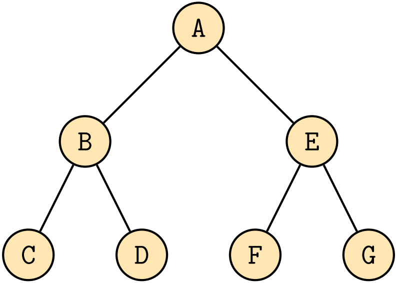
Stack: A

↳ LIFO

Printa solo caso Pre-Order

Depth-First Search - Pre-Order

```
dfs(Tree $t$)
if $t \neq$ nil then
    % pre-order visit of $t$
    print $t$
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```

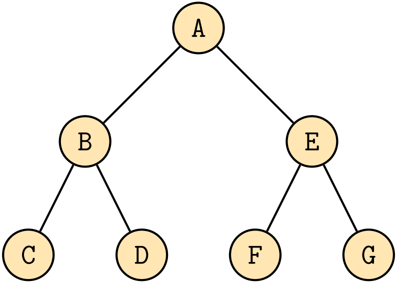


Sequence: A B

Stack: A B

Depth-First Search - Pre-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    % pre-order visit of $t$
    print $t$
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```

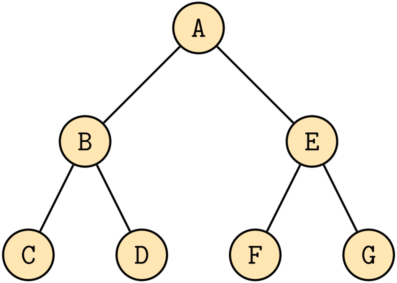


Sequence: A B C

Stack: A B C

Depth-First Search - Pre-Order

```
dfs(TREE t)
if $t \neq$ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    ~~% in-order visit of t~~
    ~~print t~~
    dfs(t.right())
    ~~% post-order visit of t~~
    ~~print t~~
```



Sequence: A B C

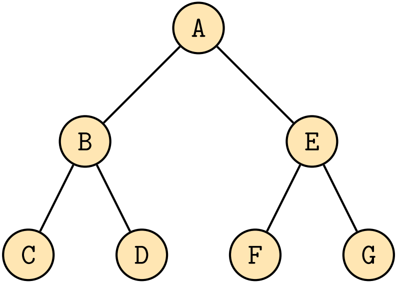
Stack: A B

non c'e' la C pk e' una foglia (leaves) -> caso tutto return null

↳ NO FIGLI !!

Depth-First Search - Pre-Order

```
dfs(TREE t)
if $t \neq$ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    ~~% in-order visit of t~~
    ~~print t~~
    dfs(t.right())
    ~~% post-order visit of t~~
    ~~print t~~
```

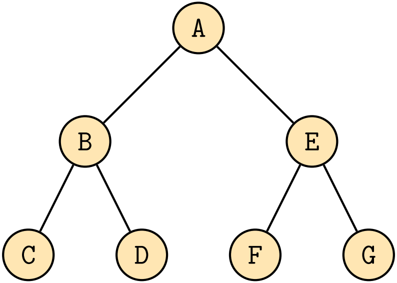


Sequence: `A B C D`

Stack: `A B D`

Depth-First Search - Pre-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    % pre-order visit of $t$
    print $t$
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```

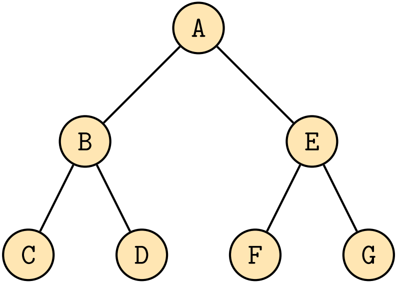


Sequence: `A B C D`

Stack: `A B`

Depth-First Search - Pre-Order

```
dfs(Tree t)
if $t \neq$ nil then
    % pre-order visit of t
    print t
    dfs(t.left())
    ~~% in-order visit of t~~
    ~~print t~~
    dfs(t.right())
    ~~% post-order visit of t~~
    ~~print t~~
```

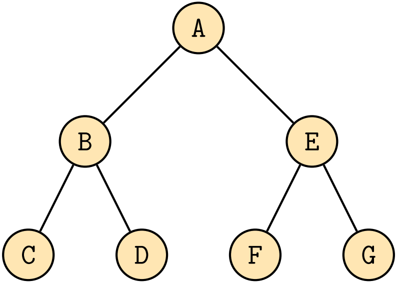


Sequence: A B C D

Stack: A

Depth-First Search - Pre-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    % pre-order visit of $t$
    print $t$
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```



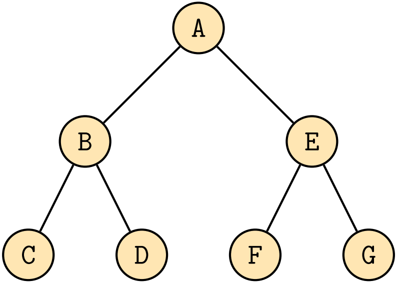
Sequence: A B C D E

Stack: A E

manca B
perché abbiamo
visitato tutti i figli
di B

Depth-First Search - Pre-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    % pre-order visit of $t$
    print $t$
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```

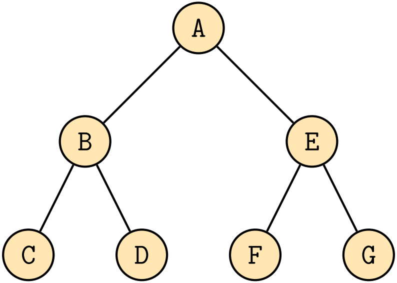


Sequence: A B C D E F

Stack: A E F

Depth-First Search - Pre-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    % pre-order visit of $t$
    print $t$
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```

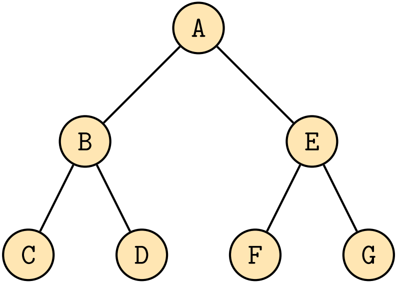


Sequence: A B C D E F

Stack: A E

Depth-First Search - Pre-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    % pre-order visit of $t$
    print $t$
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```

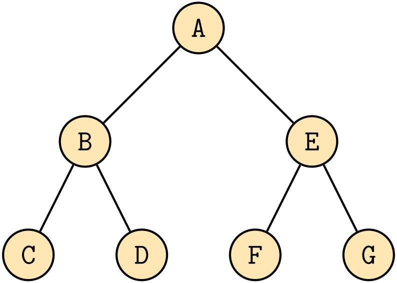


Sequence: A B C D E F G

Stack: A E G

Depth-First Search - Pre-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    % pre-order visit of $t$
    print $t$
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```

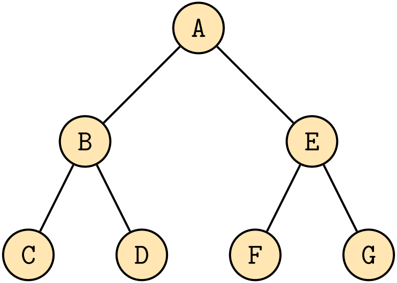


Sequence: A B C D E F G

Stack: A E

Depth-First Search - Pre-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    % pre-order visit of $t$
    print $t$
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```

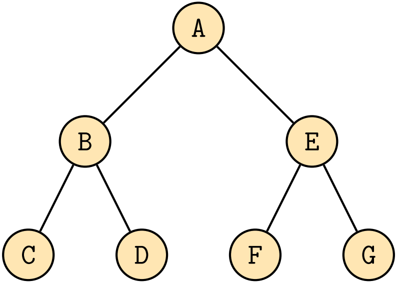


Sequence: A B C D E F G

Stack: A

Depth-First Search - Pre-Order

```
dfs(Tree $t$)
if $t \neq$ nil then
    % pre-order visit of $t$
    print $t$
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```

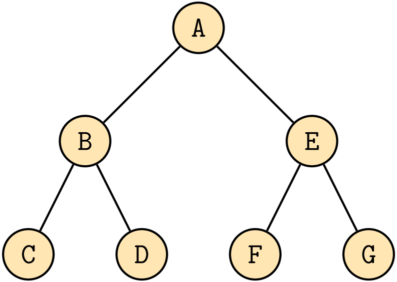


Sequence: A B C D E F G

Stack:

Depth-First Search - In-Order

```
dfs(TREE t)
if $t \neq$ nil then
    ~~% pre-order visit of t~~
    ~~print t~~
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
    ~~% post-order visit of t~~
    ~~print t~~
```



Sequence:

Stack: A

dfs(tl) → t ≠ nil

dfs(tl) → t ≠ nil

pr B

dfs(tr)

dfs(tl) → nil

pr C

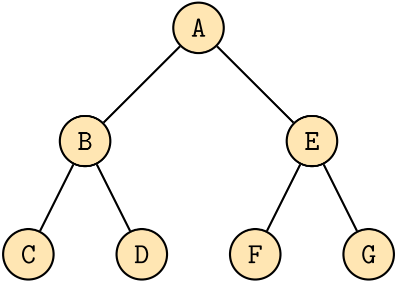
dfs(tr) → nil

dfs(tl)

pr D

Depth-First Search - In-Order

```
dfs(TREE t)
if $t \neq$ nil then
    ~~% pre-order visit of t~~
    ~~print t~~
    dfs(t.left()) →
    % in-order visit of t
    print t
    dfs(t.right())
    ~~% post-order visit of t~~
    ~~print t~~
```

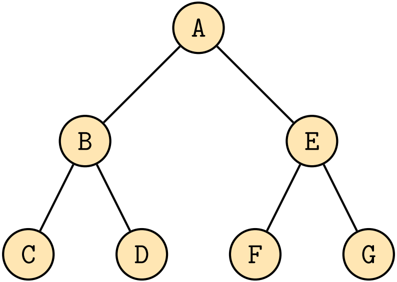


Sequence:

Stack: A B

Depth-First Search - In-Order

```
dfs(TREE t)
if $t \neq$ nil then
    ~~% pre-order visit of t~~
    ~~print t~~
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
    ~~% post-order visit of t~~
    ~~print t~~
```

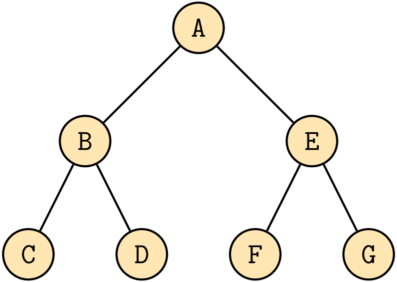


Sequence: C
Stack: A B C

Printa solo in cui c'è scritto in-order

Depth-First Search - In-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    % in-order visit of $t$
    print $t$
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```

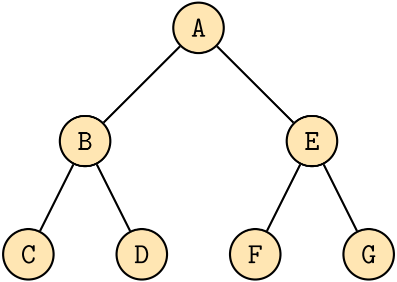


Sequence: C B

Stack: A B

Depth-First Search - In-Order

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

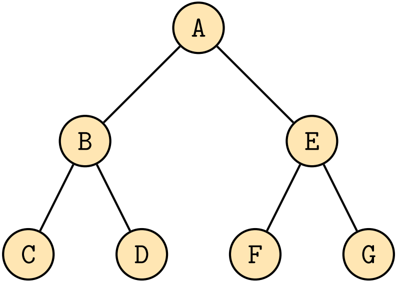


Sequence: C B D

Stack: A B D

Depth-First Search - In-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    % in-order visit of $t$
    print $t$
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```

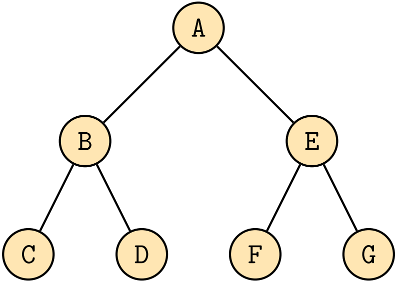


Sequence: C B D

Stack: A B

Depth-First Search - In-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    % in-order visit of $t$
    print $t$
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```

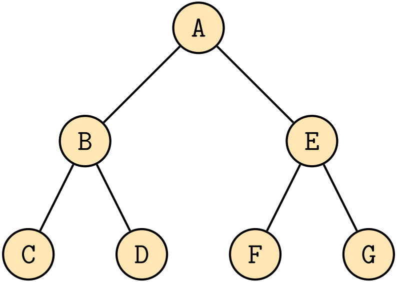


Sequence: `C B D A`

Stack: `A`

Depth-First Search - In-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    % in-order visit of $t$
    print $t$
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```

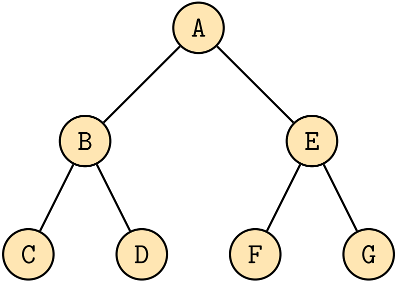


Sequence: C B D A

Stack: A E

Depth-First Search - In-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    % in-order visit of $t$
    print $t$
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```

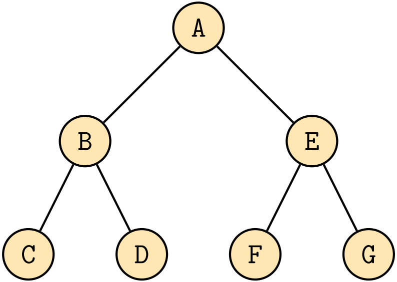


Sequence: C B D A F

Stack: A E F

Depth-First Search - In-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    % in-order visit of $t$
    print $t$
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```

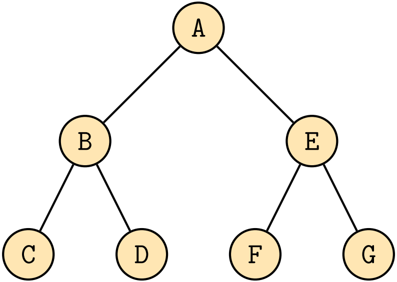


Sequence: C B D A F E

Stack: A E

Depth-First Search - In-Order

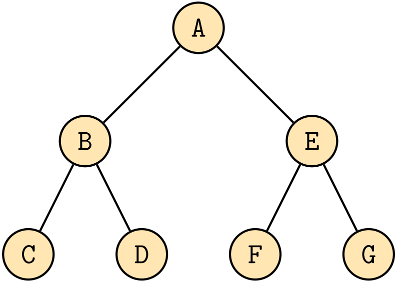
```
dfs(TREE t)
if t ≠ nil then
    ~~% pre-order visit of t~~
    ~~print t~~
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
    ~~% post-order visit of t~~
    ~~print t~~
```



Sequence: C B D A F E G
Stack: A E G

Depth-First Search - In-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    % in-order visit of $t$
    print $t$
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```

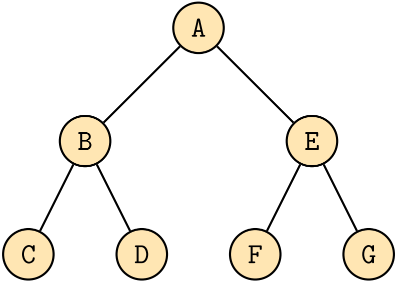


Sequence: `C B D A F E G`

Stack: `A E`

Depth-First Search - In-Order

```
dfs(TREE t)
if $t \neq$ nil then
    ~~% pre-order visit of t~~
    ~~print t~~
    dfs(t.left())
    % in-order visit of t
    print t
    dfs(t.right())
    ~~% post-order visit of t~~
    ~~print t~~
```

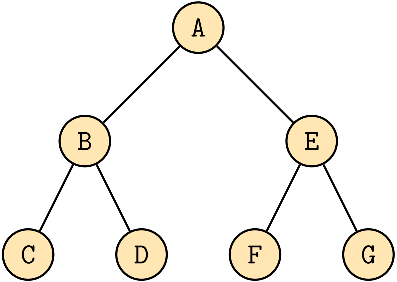


Sequence: C B D A F E G

Stack: A

Depth-First Search - In-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    % in-order visit of $t$
    print $t$
    dfs($t$.right())
    ~~% post-order visit of $t$~~
    ~~print $t$~~
```

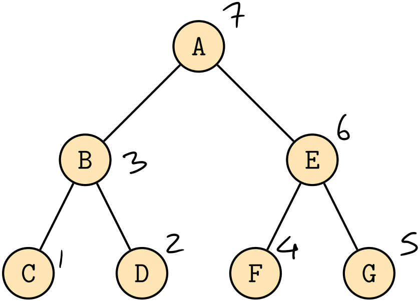


Sequence: C B D A F E G

Stack:

Depth-First Search - Post-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    % post-order visit of $t$
    print $t$
```

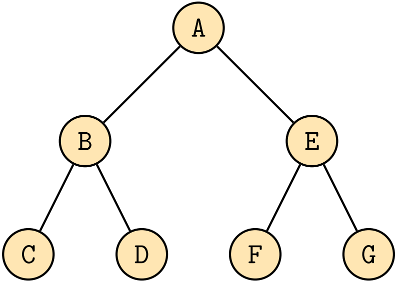


Sequence:

Stack: A

Depth-First Search - Post-Order

```
dfs(TREE t)
if $t \neq$ nil then
    ~~% pre-order visit of t~~
    ~~print t~~
    dfs(t.left())
    ~~% in-order visit of t~~
    ~~print t~~
    dfs(t.right())
    % post-order visit of t
    print t
```

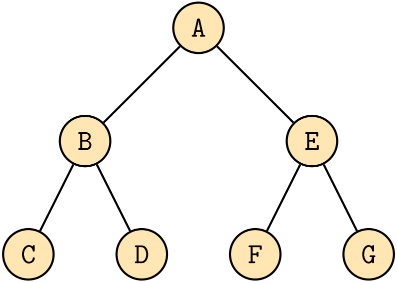


Sequence:

Stack: A B

Depth-First Search - Post-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    % post-order visit of $t$
    print $t$
```

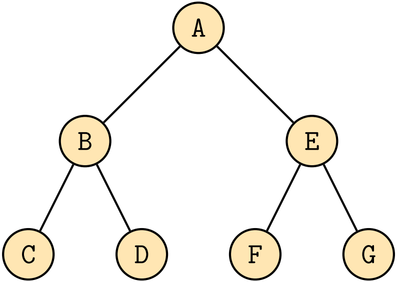


Sequence: C

Stack: A B C

Depth-First Search - Post-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    % post-order visit of $t$
    print $t$
```

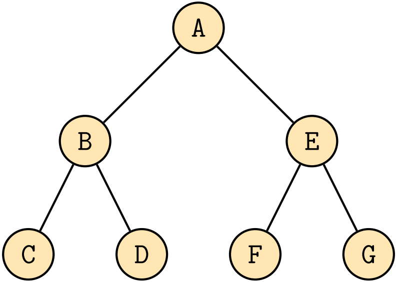


Sequence: C

Stack: A B

Depth-First Search - Post-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    % post-order visit of $t$
    print $t$
```

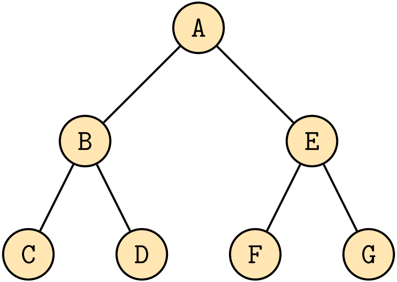


Sequence: C D

Stack: A B D

Depth-First Search - Post-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    % post-order visit of $t$
    print $t$
```

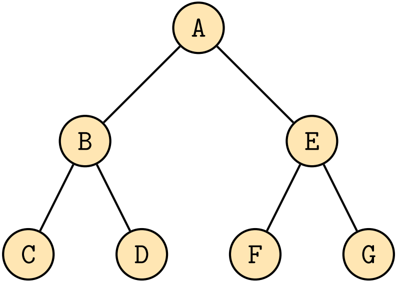


Sequence: `C D B`

Stack: `A B`

Depth-First Search - Post-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    % post-order visit of $t$
    print $t$
```

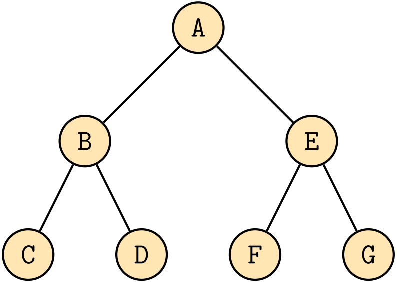


Sequence: C D B

Stack: A

Depth-First Search - Post-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    % post-order visit of $t$
    print $t$
```

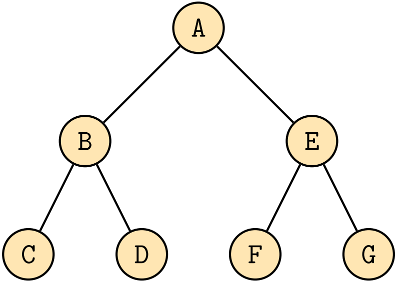


Sequence: C D B

Stack: A E

Depth-First Search - Post-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    % post-order visit of $t$
    print $t$
```

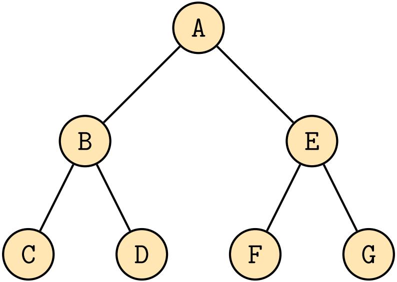


Sequence: C D B F

Stack: A E F

Depth-First Search - Post-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    % post-order visit of $t$
    print $t$
```

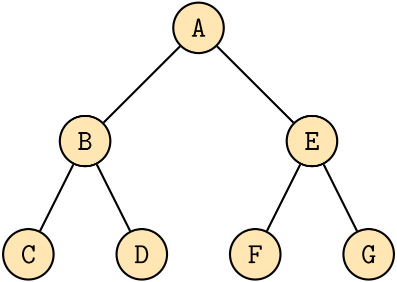


Sequence: C D B F

Stack: A E

Depth-First Search - Post-Order

```
dfs(Tree $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    % post-order visit of $t$
    print $t$
```

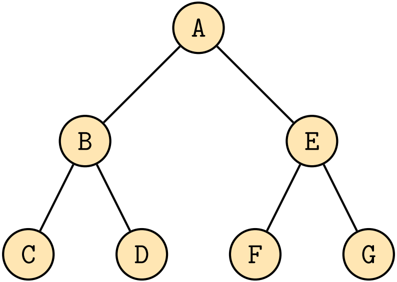


Sequence: C D B F G

Stack: A E G

Depth-First Search - Post-Order

```
dfs(TREE $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    % post-order visit of $t$
    print $t$
```

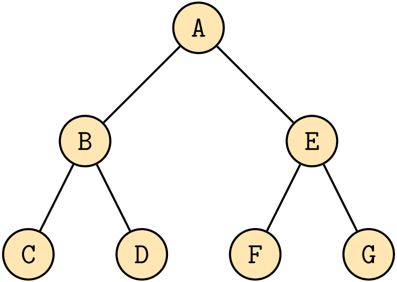


Sequence: C D B F G E

Stack: A E

Depth-First Search - Post-Order

```
dfs(Tree $t$)
if $t \neq$ nil then
    ~~% pre-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.left())
    ~~% in-order visit of $t$~~
    ~~print $t$~~
    dfs($t$.right())
    % post-order visit of $t$
    print $t$
```

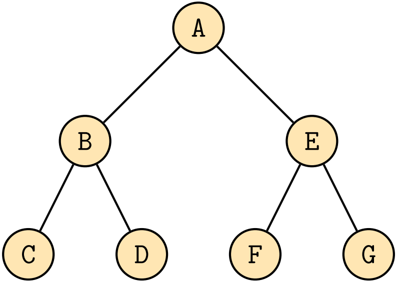


Sequence: `C D B F G E A`

Stack: `A`

Depth-First Search - Post-Order

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



Sequence: C D B F G E A
Stack:

Esempi di applicazione

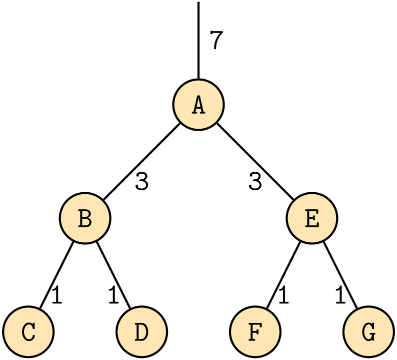
Obbiettivo -> contare il numero totale di nodi in un albero binario utilizzando una visita in post ordine

POST ORDINE -> prima si esplorano completamente i sottoalberi sx e dx -> poi si elabora la radice

Contare nodi – Post-visita

-> quanto e' grande il sottoalbero?

```
int count(TREE $T$)
if $T$ == nil then
    return 0
else
    $C_\ell$ = count($T$.left())
    $C_r$ = count($T$.right())
    return $C_\ell + C_r + 1$
```



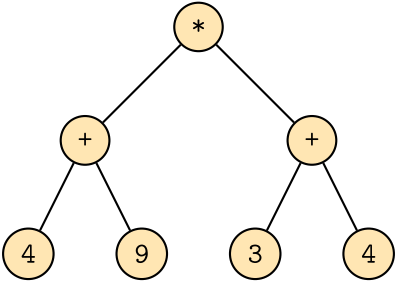
Esempi di applicazione

Obbiettivo → Stampare un'espressione matematica memorizzata in un albero binario utilizzando una visita **in-ordine**

L'albero rappresenta l'espressione in forma gerarchica dove:
Nodi foglia: Numeri (operandi)
Nodi interni: Operatori (+, •, ...)

Stampare espressioni – In-visita

```
int printExp(TREE $T$)
if $T$.left() == nil and $T$.right() == nil then
    print $T$.read()
else
    print "("
    printExp($T$.left())
    print $T$.read()
    printExp($T$.right())
    print ")"
```



((4+9) * (3+4))

Costo computazionale

Il costo di una visita di un albero contenente n nodi è $\Theta(n)$, in quanto ogni nodo viene visitato al massimo una volta.

Sommario

Specifica (Albero generico)

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

Specifica (Albero generico)

Primo figlio -> left child

left

TREE

% Restituisce il primo figlio, oppure **nil** se questo nodo è una foglia

TREE leftmostChild()

% Restituisce il prossimo fratello, oppure **nil** se assente

TREE rightSibling()

% Inserisce il sottoalbero t come primo figlio di questo nodo

insertChild(TREE t)

% Inserisce il sottoalbero t come prossimo fratello di questo nodo

insertSibling(TREE t)

% Distruggi l'albero radicato identificato dal primo figlio

(DISTRUGGE L'ALBERO DEL PRIMO FIGLIO !!)

deleteChild()

% Distruggi l'albero radicato identificato dal prossimo fratello

(DISTRUGGE ALBERO DEL FRATELLO !!)

deleteSibling()

Esempio: Class Node (Java 8)

```
package org.w3c.dom;
public interface Node {

  /** The parent of this node. */
  public Node   getParentNode();

  /** The first child of this node. */
  public Node   getFirstChild()

  /** The node immediately following this node. */
  public Node   getNextSibling()

  /** Inserts the node newChild before the existing child node refChild. */
  public Node   insertBefore(Node newChild, Node refChild)

  /** Adds the node newChild to the end of the list of children of this node. */
  public Node   appendChild(Node newChild)

  /** Removes the child node indicated by oldChild from the list of children. */
  public Node   removeChild(Node oldChild)

  [...]
}
```

Depth-First Search

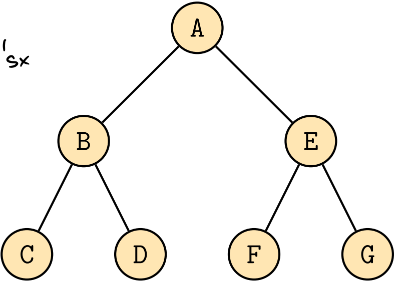
Pre-order → classica visita in profondità

post-order → (es. contare nodi) → misuro la profondità dei sottoalberi

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

prendo il figlio sx

fratello (dx)



print A
u = B
while
dfs(u) → print B
u = C
while
dfs(u) → print c
while ∅
u = D → continuo il while → print D while ∅

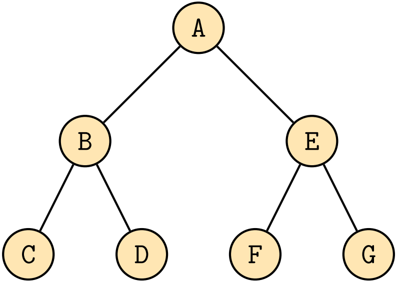
Breadth-First Search

Esplora un albero livello per livello, partendo dalla radice e procedendo verso i nodi più profondi in ordine crescente di distanza dalla radice

↳ A differenza del DFS, usa una coda (Queue) per gestire i nodi da visitare

FIFO

```
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 $u \neq$ nil do
        $Q$.enqueue($u$)
        $u$ = $u$.rightSibling()
```

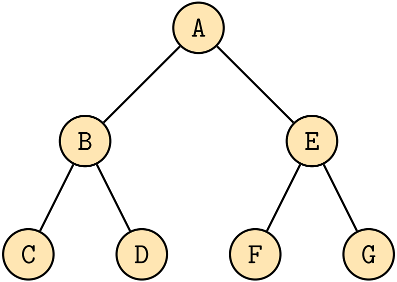


Sequence:

Queue: A

Breadth-First Search

```
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 $u \neq$ nil do
        $Q$.enqueue($u$)
        $u = u$.rightSibling()
```

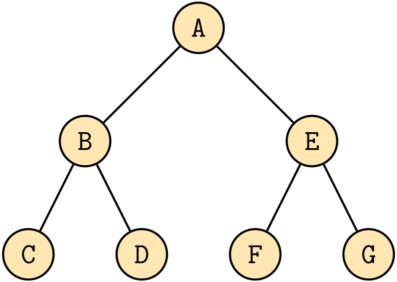


Sequence: A

Queue: B E

Breadth-First Search

```
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 $u \neq$ nil do
        $Q$.enqueue($u$)
        $u = u$.rightSibling()
```

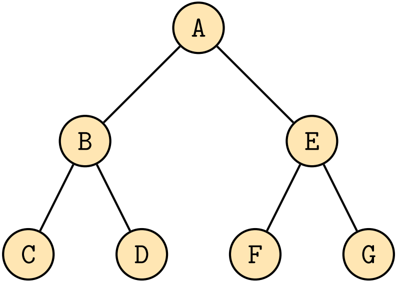


Sequence: A B

Queue: E C D

Breadth-First Search

```
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 $u \neq$ nil do
        $Q$.enqueue($u$)
        $u = u$.rightSibling()
```

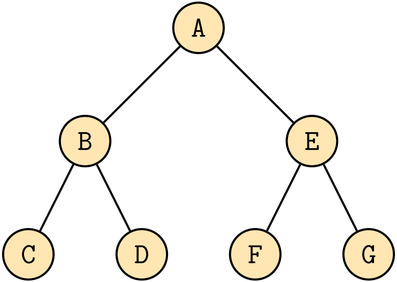


Sequence: A B E

Queue: C D F G

Breadth-First Search

```
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 $u \neq$ nil do
        $Q$.enqueue($u$)
        $u = u$.rightSibling()
```

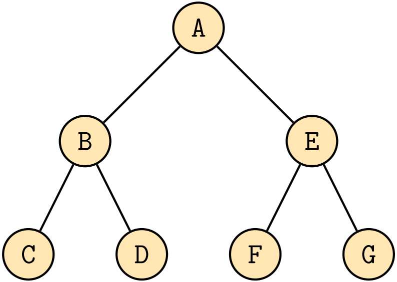


Sequence: A B E C

Queue: D F G

Breadth-First Search

```
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 $u \neq$ nil do
        $Q$.enqueue($u$)
        $u = u$.rightSibling()
```

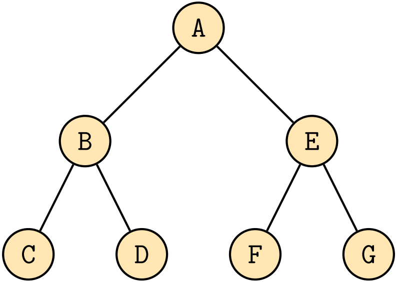


Sequence: A B E C D

Queue: F G

Breadth-First Search

```
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 $u \neq$ nil do
        $Q$.enqueue($u$)
        $u$ = $u$.rightSibling()
```

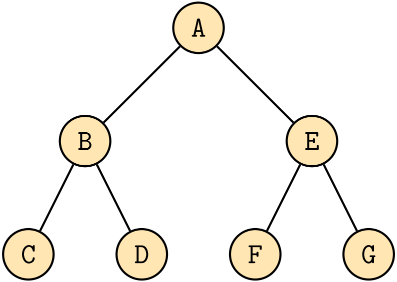


Sequence: A B E C D F

Queue: G

Breadth-First Search

```
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 $u \neq$ nil do
        $Q$.enqueue($u$)
        $u$ = $u$.rightSibling()
```



Sequence: A B E C D F G
Queue:

Differenze tra DFS e BFS

1) BFS → usa una coda (Queue)
DFS → usa una stack

2) BFS → visita per livello
DFS → visita in profondità

3) BFS → ottima per trovare cammini minimi (es. alberi di decisione)
DFS → Utile per esplorare tutti i rami (es. backtraking)

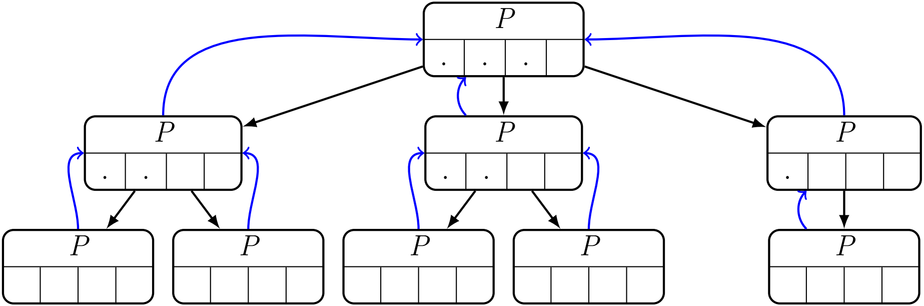
Tempo → $O(n)$ → n e' il numero di nodi (ogni nodo viene visitato una volta)

Memorizzazione

Esistono diversi modi per memorizzare un albero, più o meno indicati a seconda del numero massimo e medio di figli presenti.

- Realizzazione con vettore dei figli
- Realizzazione primo figlio, prossimo fratello
- Realizzazione con vettore dei padri

Realizzazione con vettore dei figli



Campi memorizzati nei nodi

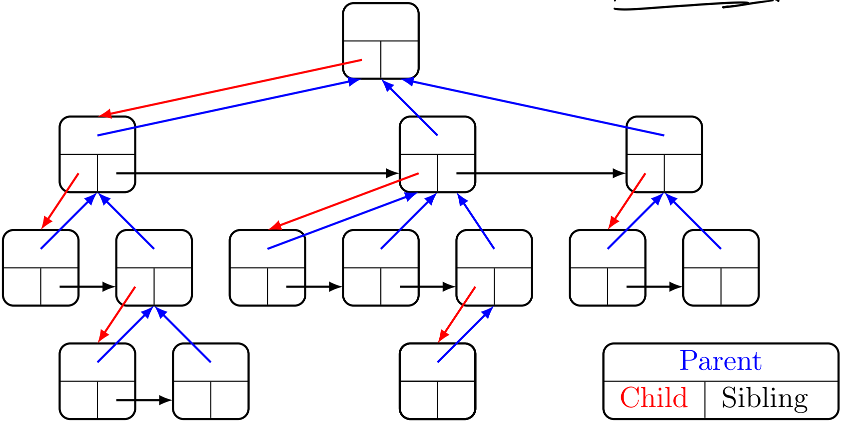
- *parent*: reference al nodo padre
- Vettore dei figli: a seconda del numero di figli, può comportare una discreta quantità di spazio sprecato

Realizzazione basata su Primo figlio, prossimo fratello

Implementato come una lista di fratelli

Padre

FIGLIO | FRATELLO



Implementazione

```
TREE
TREE parent                                          % Reference al padre
TREE child                                           % Reference al primo figlio
TREE sibling                                         % Reference al prossimo fratello
ITEM value                                           % Valore memorizzato nel nodo

Tree(ITEM v)                                         % Crea un nuovo nodo
    TREE t = new TREE
    t.value = v
    t.parent = t.child = t.sibling = nil   → rappresenta la radice
    return t

insertChild(TREE t)
    t.parent = self
    t.sibling = child                                % Inserisce t prima dell'attuale primo figlio
    child = t   ← prima dell' attuale figlio

insertSibling(TREE t)
    t.parent = parent
    t.sibling = sibling                              % Inserisce t prima dell'attuale prossimo fratello
    sibling = t   ← prima dell' attuale fratello
```

Implementazione

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

deleteSibling()
    TREE $newSibling = sibling$.rightSibling()
    delete($sibling$)
    $sibling = newSibling$

delete(TREE $t$)
    TREE $u = t$.leftmostChild()
    while $u \neq$ nil do
        TREE $next = u$.rightSibling()
        delete($u$)
        $u = next$
    delete $t$
```

- cancella il primo figlio
- cancella il primo fratello

elimina l'albero t partendo dalla foglia più a Sx

Realizzazione con vettore dei padri

L'albero è rappresentato da un vettore i cui elementi contengono il valore associato al nodo e l'indice della posizione del padre nel vettore.

1	A	0
2	B	1
3	E	1
4	C	2
5	D	2
6	F	3
7	G	3