

# Tree Traversal (1A)

---

Copyright (c) 2015 - 2018 Young W. Lim.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Please send corrections (or suggestions) to [youngwlim@hotmail.com](mailto:youngwlim@hotmail.com).

This document was produced by using LibreOffice and Octave.

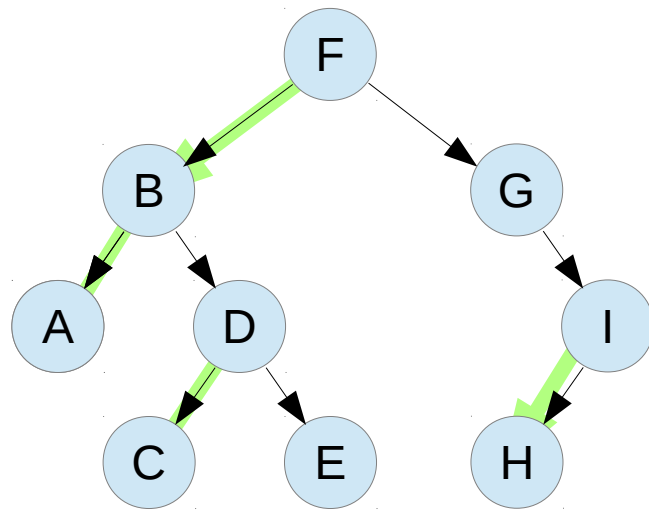
# Tree Traversal

## Depth First Search

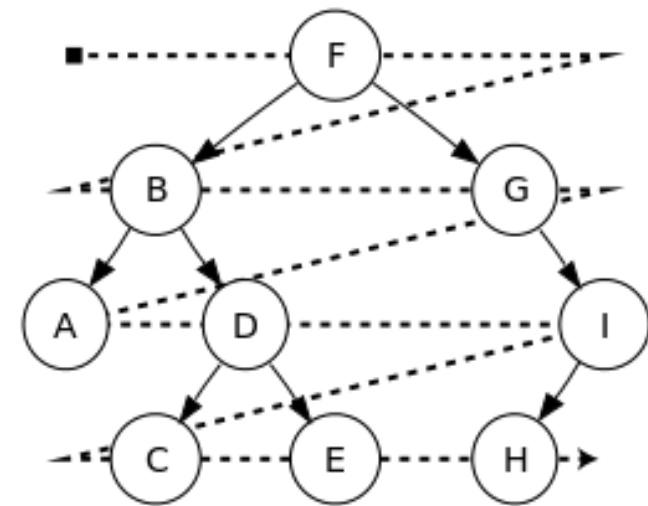
Pre-Order

In-order

Post-Order



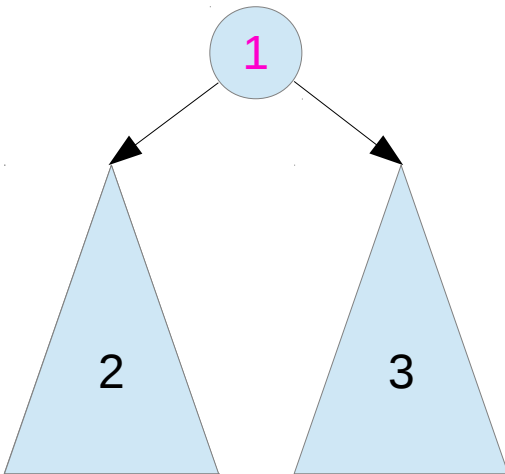
## Breadth First Search



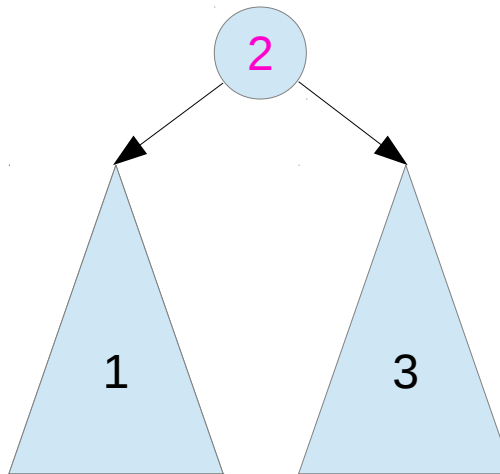
<https://en.wikipedia.org/wiki/Morphism>

# Recursive Algorithms

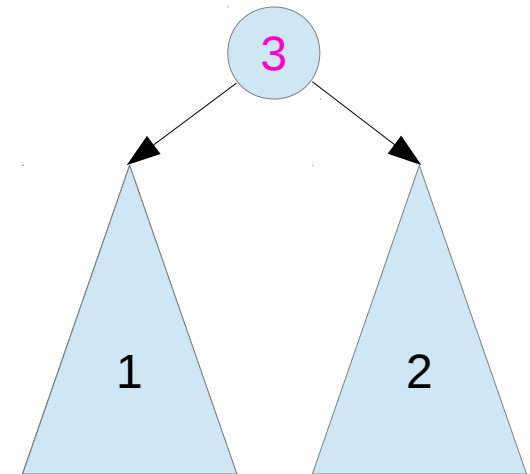
```
preorder(node)
if (node = null)
  return
visit(node)
preorder(node.left)
preorder(node.right)
```



```
inorder(node)
if (node = null)
  return
inorder(node.left)
visit(node)
inorder(node.right)
```



```
postorder(node)
if (node = null)
  return
postorder(node.left)
postorder(node.right)
visit(node)
```



[https://en.wikipedia.org/wiki/Tree\\_traversal](https://en.wikipedia.org/wiki/Tree_traversal)

# Iterative Algorithms

**iterativePreorder**(node)

```
if (node = null)
  return
s ← empty stack
s.push(node)
```

**while** (not s.isEmpty())

```
node ← s.pop()
visit(node)
// right child is pushed first
// so that left is processed first
if (node.right ≠ null)
  s.push(node.right)
if (node.left ≠ null)
  s.push(node.left)
```

[https://en.wikipedia.org/wiki/Tree\\_traversal](https://en.wikipedia.org/wiki/Tree_traversal)

**iterativeInorder**(node)

```
s ← empty stack

while (not s.isEmpty() or
  node ≠ null)
  if (node ≠ null)
    s.push(node)
    node ← node.left
  else
    node ← s.pop()
    visit(node)
    node ← node.right
```

**iterativePostorder**(node)

```
s ← empty stack
lastNodeVisited ← null

while (not s.isEmpty() or node ≠ null)
  if (node ≠ null)
    s.push(node)
    node ← node.left
  else
    peekNode ← s.peek()
    // if right child exists and traversing
    // node from left child, then move right
    if (peekNode.right ≠ null and
      lastNodeVisited ≠ peekNode.right)
      node ← peekNode.right
    else
      visit(peekNode)
      lastNodeVisited ← s.pop()
```

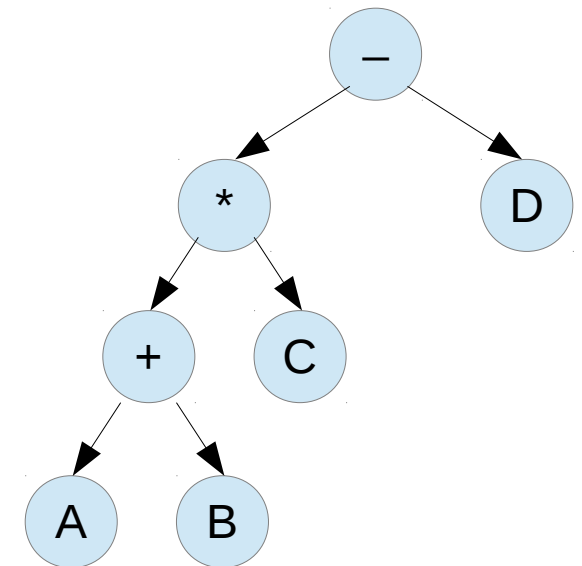
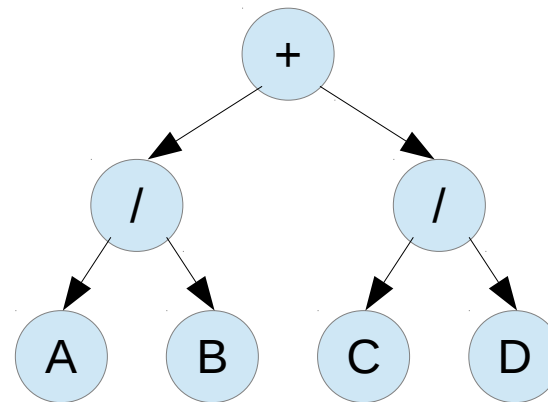
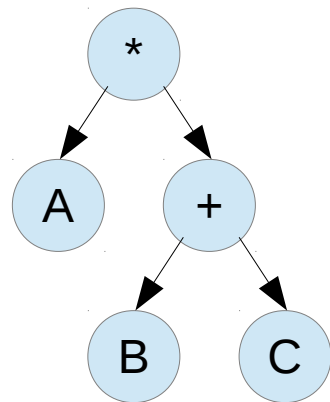
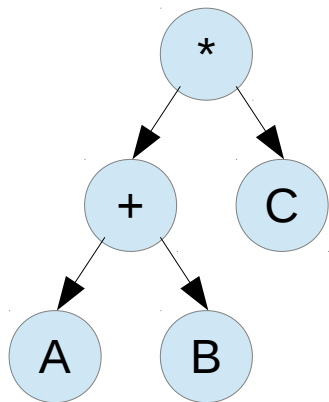
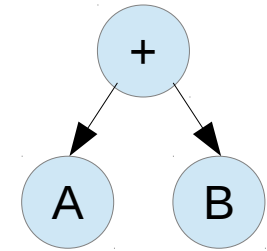
# Infix, Prefix, Postfix Notations

Infix Notation	Prefix Notation	Postfix Notation
$A + B$	$+ A B$	$A B +$
$(A + B) * C$	$* + A B C$	$A B + C *$
$A * (B + C)$	$* A + B C$	$A B C + *$
$A / B + C / D$	$+ / A B / C D$	$A B / C D / +$
$((A + B) * C) - D$	$- * + A B C D$	$A B + C * D -$

[https://www.tutorialspoint.com/data\\_structures\\_algorithms/expression\\_parsing.html](https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.html)

# Infix, Prefix, Postfix Notations and Binary Trees

Infix Notation	Prefix Notation	Postfix Notation
$A + B$	$+ A B$	$A B +$
$(A + B) * C$	$* + A B C$	$A B + C *$
$A * (B + C)$	$* A + B C$	$A B C + *$
$A / B + C / D$	$+ / A B / C D$	$A B / C D / +$
$((A + B) * C) - D$	$- * + A B C D$	$A B + C * D -$

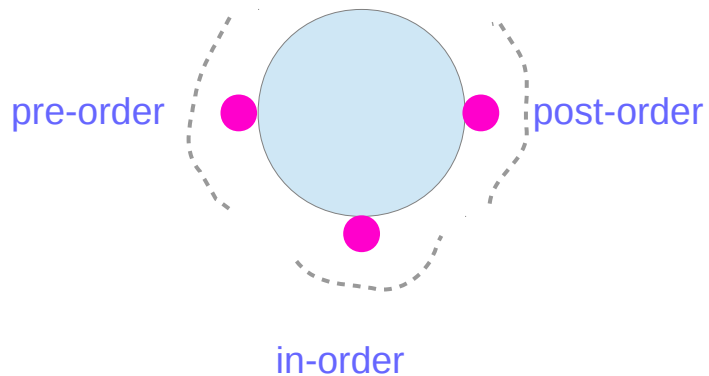
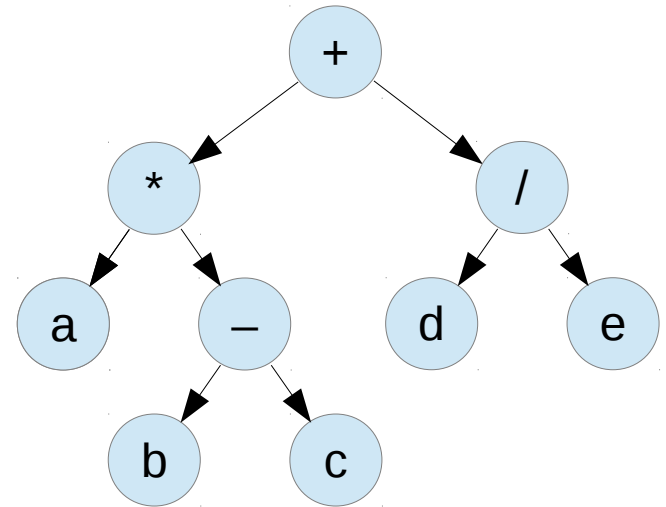


[https://www.tutorialspoint.com/data\\_structures\\_algorithms/expression\\_parsing.htm](https://www.tutorialspoint.com/data_structures_algorithms/expression_parsing.htm)

# In-Order, Pre-Order, Post-Order Binary Tree Traversals

Depth First Search  
Pre-Order  
In-order  
Post-Order

Breadth First Search



$(a*(b-c))+d/e$

$a * b - c + d / e$

$+ * a - b c / d e$

$a b c - * d e / +$

Infix notation

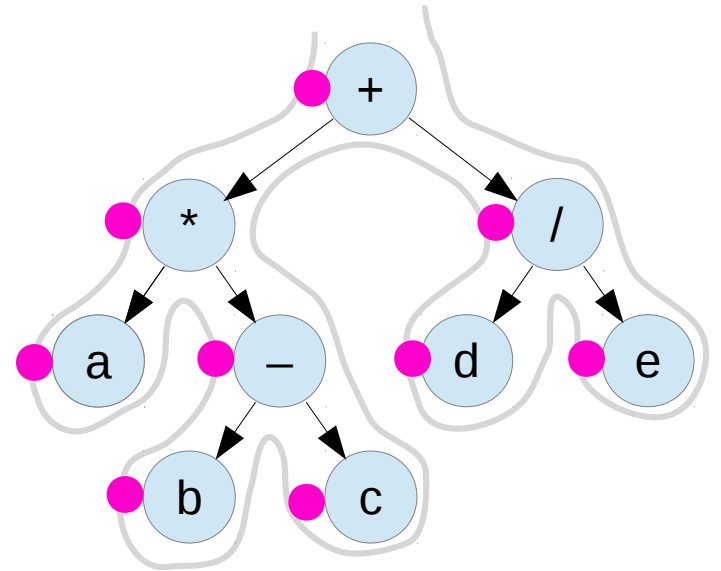
Prefix notation

Postfix notation

[https://en.wikipedia.org/wiki/Tree\\_traversal](https://en.wikipedia.org/wiki/Tree_traversal)



# Pre-Order Binary Tree Traversals



$(a*(b-c))+d/e$

$a * b - c + d / e$

$+ * a - b c / d e$

$a b c - * d e / +$

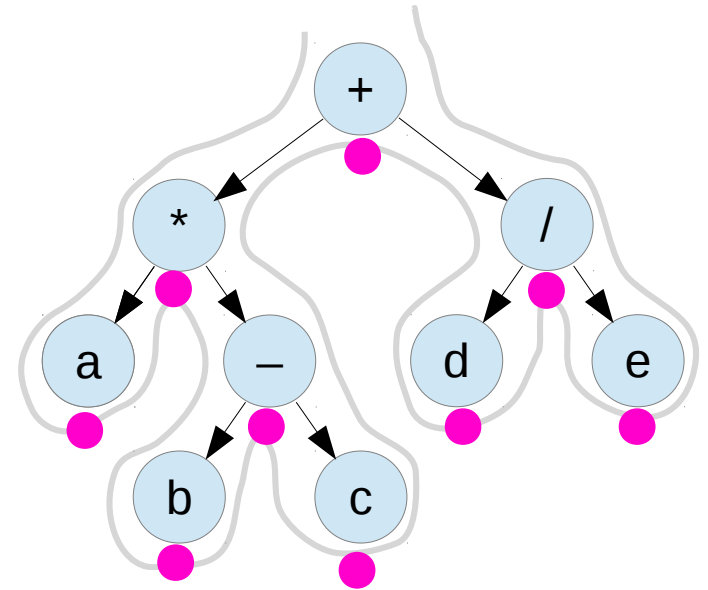
Infix notation

Prefix notation

Postfix notation

[https://en.wikipedia.org/wiki/Tree\\_traversal](https://en.wikipedia.org/wiki/Tree_traversal)

# In-Order Binary Tree Traversals



$(a*(b-c))+d/e$

$a * b - c + d / e$

$+ * a - b c / d e$

$a b c - * d e / +$

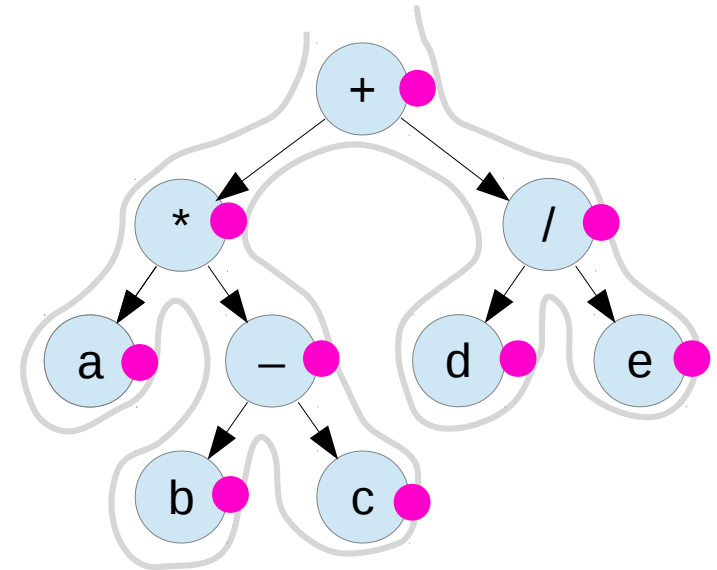
Infix notation

Prefix notation

Postfix notation

[https://en.wikipedia.org/wiki/Tree\\_traversal](https://en.wikipedia.org/wiki/Tree_traversal)

# Post-Order Binary Tree Traversals



$(a*(b-c))+d/e$

$a * b - c + d / e$

$+ * a - b c / d e$

$a b c - * d e / +$

Infix notation

Prefix notation

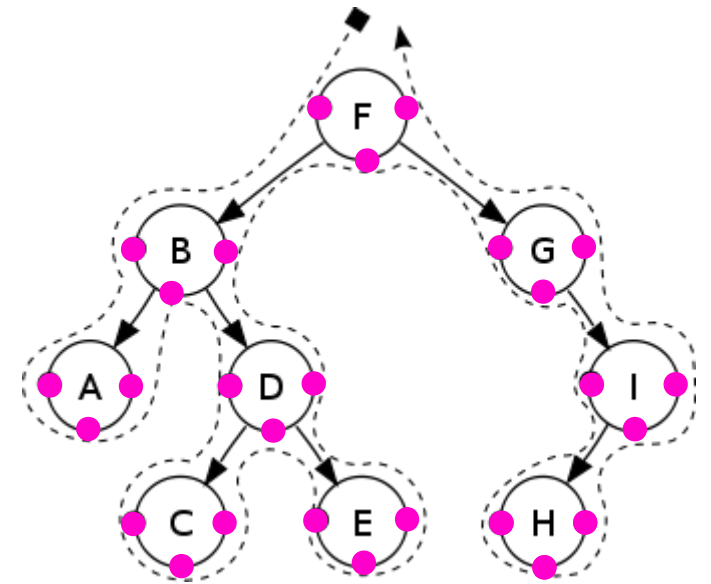
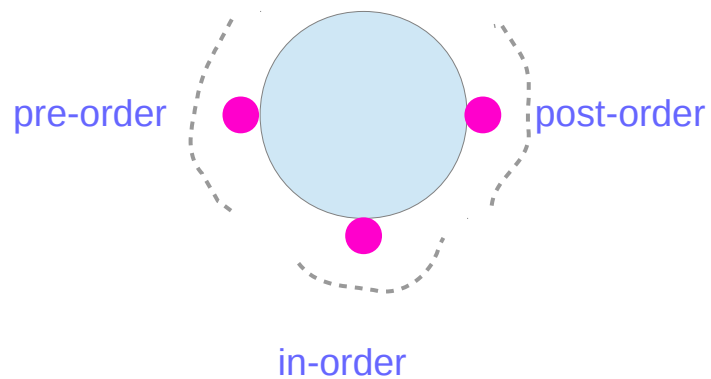
Postfix notation

[https://en.wikipedia.org/wiki/Tree\\_traversal](https://en.wikipedia.org/wiki/Tree_traversal)

# Tree Traversal

Depth First Search  
Pre-Order  
In-order  
Post-Order

Breadth First Search



[https://en.wikipedia.org/wiki/Tree\\_traversal](https://en.wikipedia.org/wiki/Tree_traversal)

# Pre-Order

## pre-order function

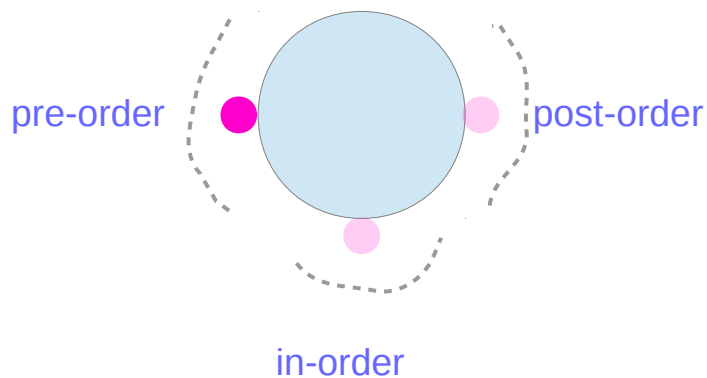
Check if the current node is empty / null.

**Display** the data part of the root (or current node).

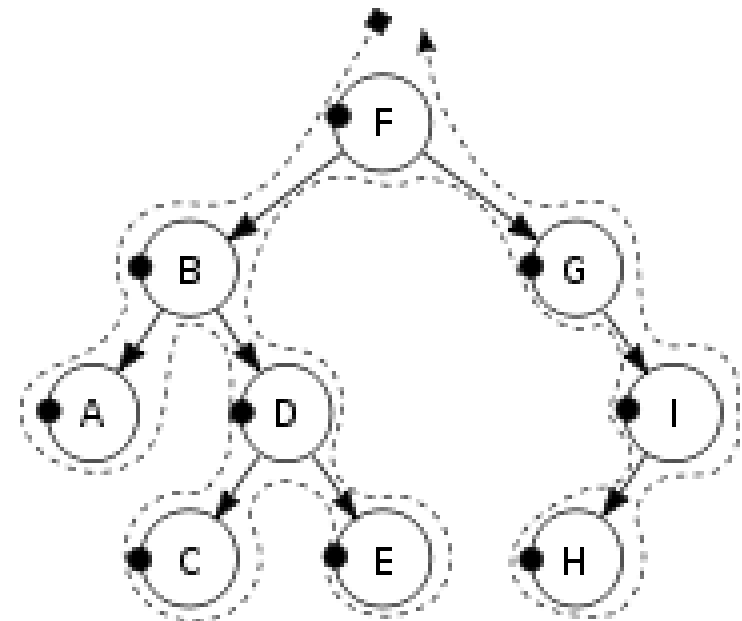
**Traverse** the **left** subtree by recursively calling the **pre-order** function.

**Traverse** the **right** subtree by recursively calling the **pre-order** function.

**FBADCEGIH**



[https://en.wikipedia.org/wiki/Tree\\_traversal](https://en.wikipedia.org/wiki/Tree_traversal)



# In-Order

## in-order function

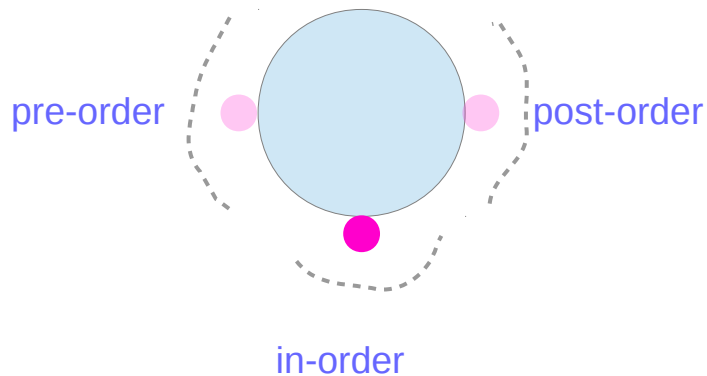
Check if the current node is empty / null.

**Traverse** the left subtree by recursively calling the **in-order** function.

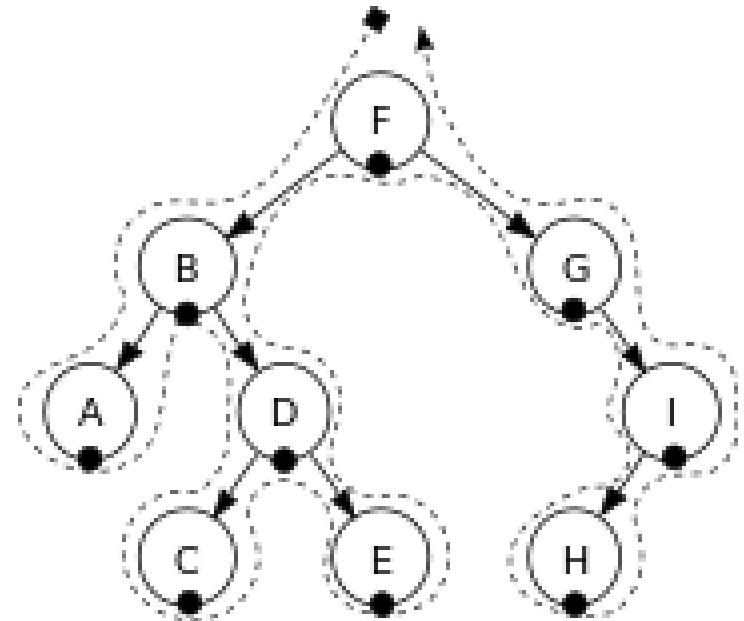
**Display** the data part of the root (or current node).

**Traverse** the right subtree by recursively calling the **in-order** function.

ABCDEFGHI



[https://en.wikipedia.org/wiki/Tree\\_traversal](https://en.wikipedia.org/wiki/Tree_traversal)



# Post-Order

## post-order function

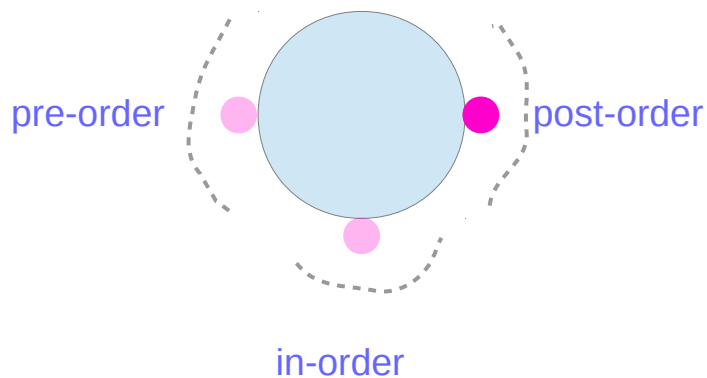
Check if the current node is empty / null.

**Traverse** the left subtree by recursively calling the **post-order** function.

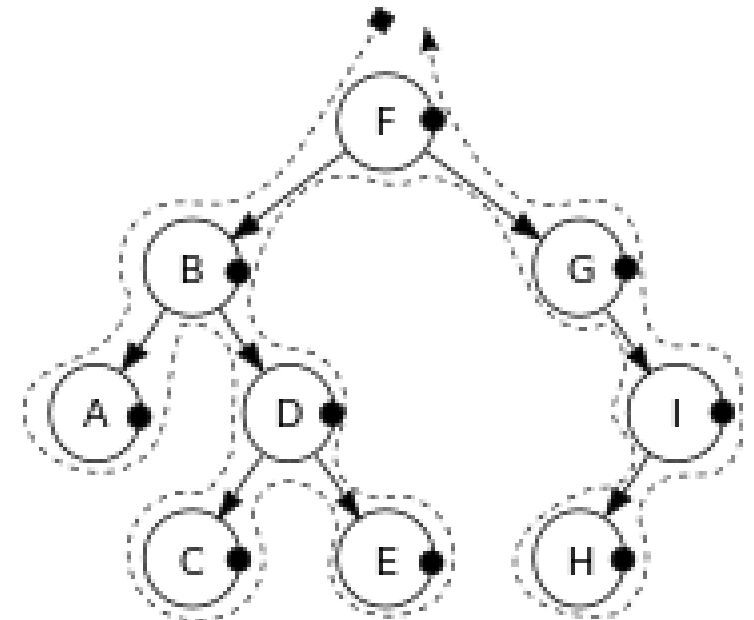
**Traverse** the right subtree by recursively calling the **post-order** function.

**Display** the data part of the root (or current node).

ACEDBHIGH

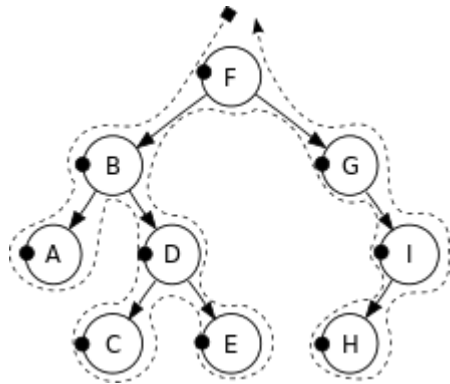


[https://en.wikipedia.org/wiki/Tree\\_traversal](https://en.wikipedia.org/wiki/Tree_traversal)

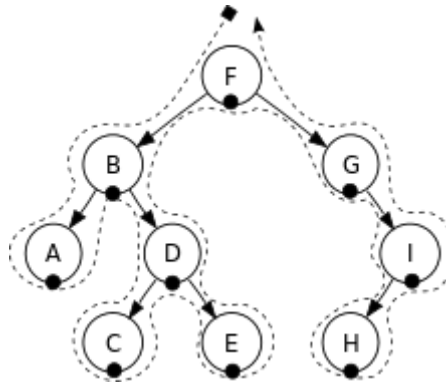


# Recursive Algorithms

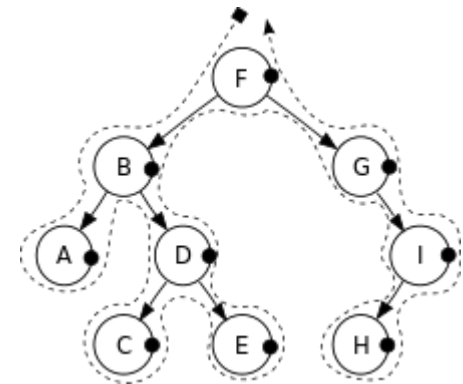
```
preorder(node)
  if (node = null)
    return
  visit(node)
  preorder(node.left)
  preorder(node.right)
```



```
inorder(node)
  if (node = null)
    return
  inorder(node.left)
  visit(node)
  inorder(node.right)
```



```
postorder(node)
  if (node = null)
    return
  postorder(node.left)
  postorder(node.right)
  visit(node)
```

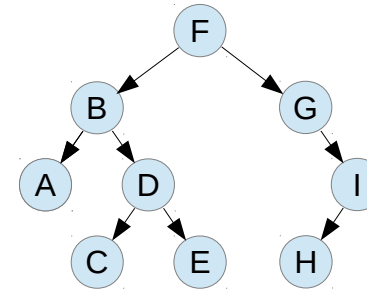
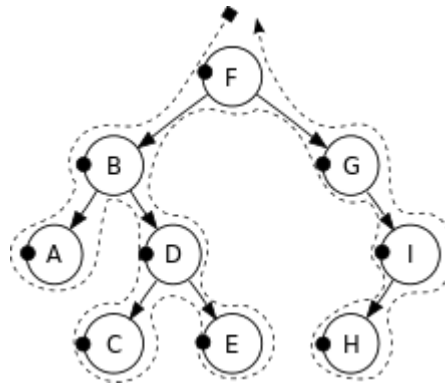


[https://en.wikipedia.org/wiki/Tree\\_traversal](https://en.wikipedia.org/wiki/Tree_traversal)

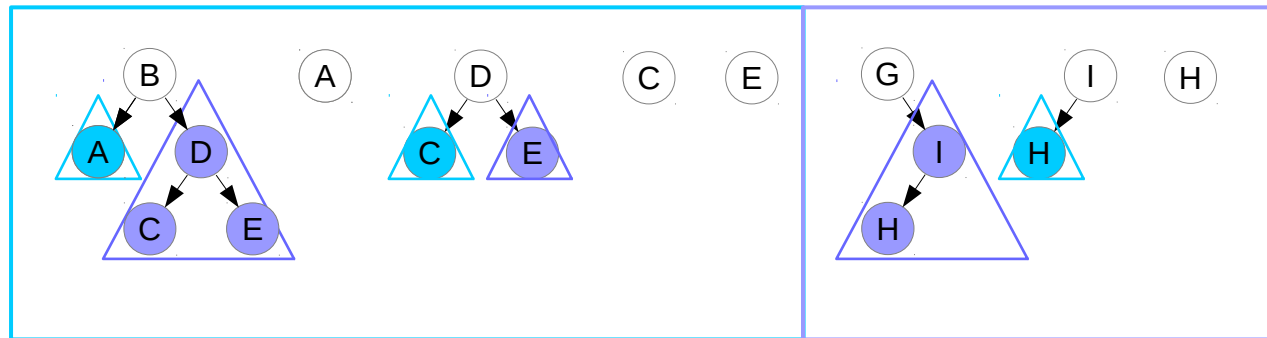
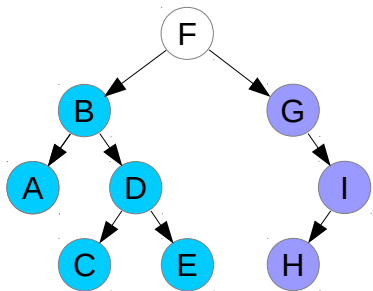


# Pre-Order recursive algorithm

```
preorder(node)
  if (node = null)
    return
  visit(node)
  preorder(node.left)
  preorder(node.right)
```



F — B — A — D — C — E — G — I — H



[https://en.wikipedia.org/wiki/Tree\\_traversal](https://en.wikipedia.org/wiki/Tree_traversal)

# Iterative Algorithms

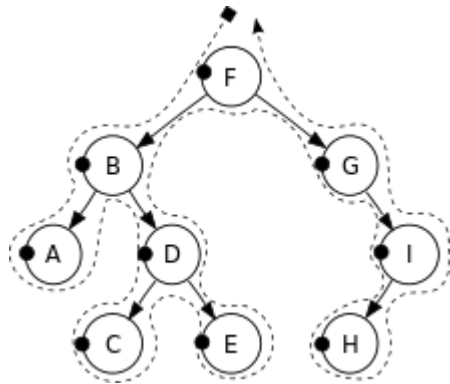
**iterativePreorder**(node)

```
if (node = null)
  return
s ← empty stack
s.push(node)
```

**while** (not s.isEmpty())

```
node ← s.pop()
visit(node)
// right child is pushed first
// so that left is processed first
if (node.right ≠ null)
  s.push(node.right)
if (node.left ≠ null)
  s.push(node.left)
```

[https://en.wikipedia.org/wiki/Tree\\_traversal](https://en.wikipedia.org/wiki/Tree_traversal)

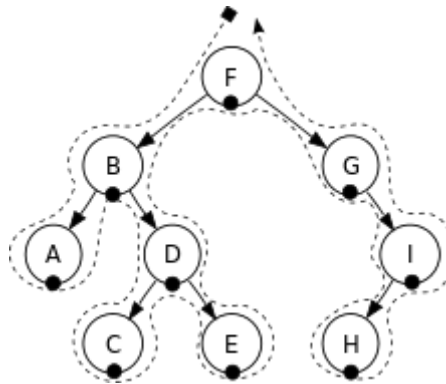


**iterativeInorder**(node)

```
s ← empty stack
```

**while** (not s.isEmpty() or node ≠ null)

```
if (node ≠ null)
  s.push(node)
  node ← node.left
else
  node ← s.pop()
  visit(node)
  node ← node.right
```

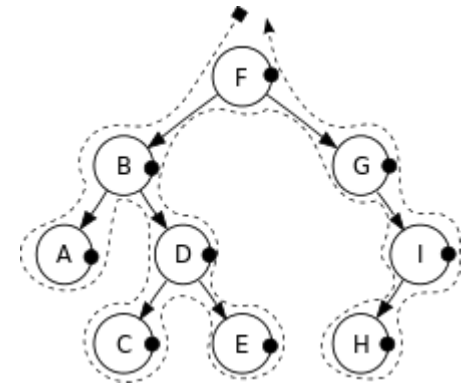


**iterativePostorder**(node)

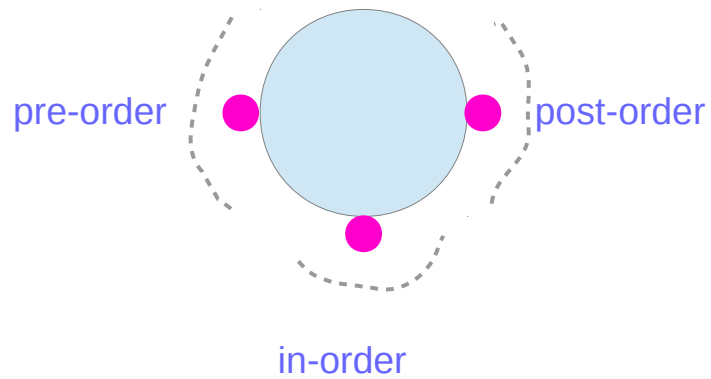
```
s ← empty stack
lastNodeVisited ← null
```

**while** (not s.isEmpty() or node ≠ null)

```
if (node ≠ null)
  s.push(node)
  node ← node.left
else
  peekNode ← s.peek()
  // if right child exists and traversing
  // node from left child, then move right
  if (peekNode.right ≠ null and
      lastNodeVisited ≠ peekNode.right)
    node ← peekNode.right
  else
    visit(peekNode)
    lastNodeVisited ← s.pop()
```

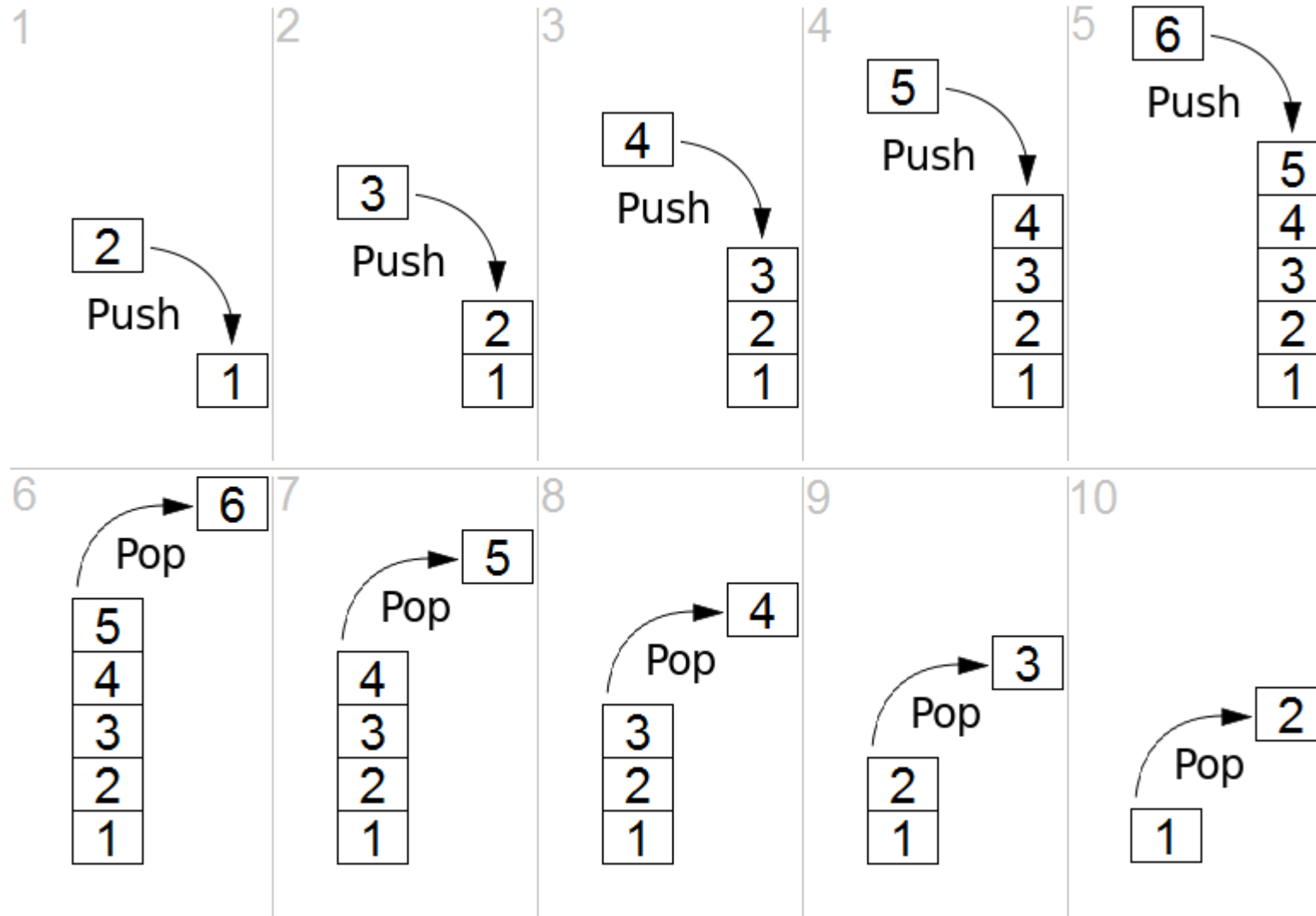


# Tree Traversal



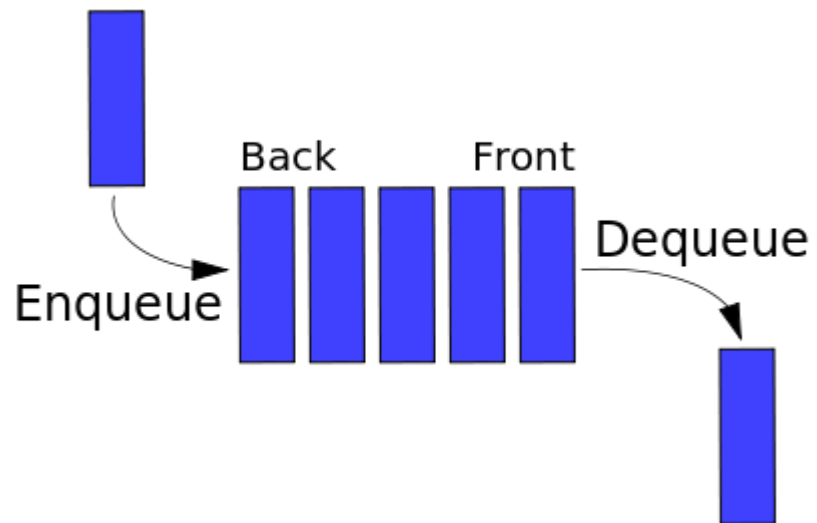
[https://en.wikipedia.org/wiki/Tree\\_traversal](https://en.wikipedia.org/wiki/Tree_traversal)

# Stack



[https://en.wikipedia.org/wiki/Stack\\_\(abstract\\_data\\_type\)](https://en.wikipedia.org/wiki/Stack_(abstract_data_type))

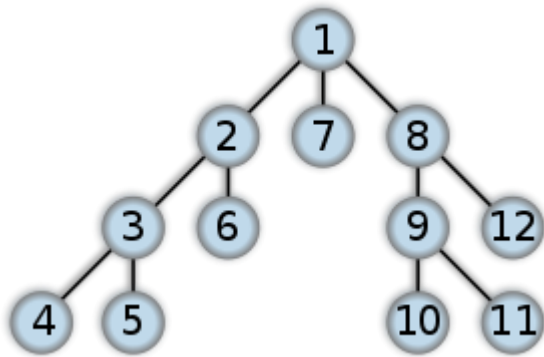
# Queue



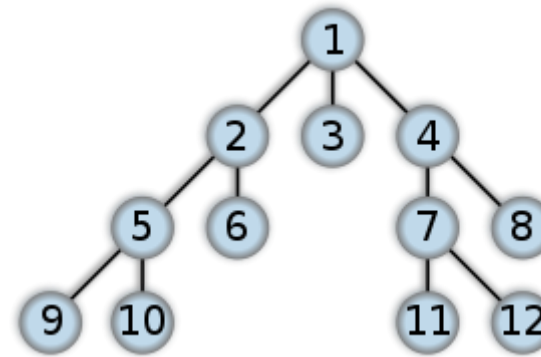
[https://en.wikipedia.org/wiki/Queue\\_\(abstract\\_data\\_type\)#/media/File:Data\\_Queue.svg](https://en.wikipedia.org/wiki/Queue_(abstract_data_type)#/media/File:Data_Queue.svg)

# Search Algorithms

DFS (Depth First Search)



BFS (Breadth First Search)



[https://en.wikipedia.org/wiki/Breadth-first\\_search](https://en.wikipedia.org/wiki/Breadth-first_search), /Depth-first\_search

# DFS Algorithm

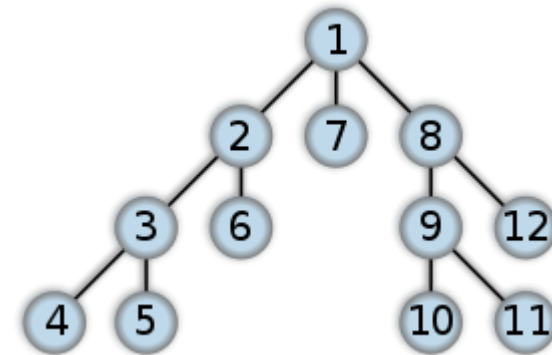
A recursive implementation of DFS:

```
procedure DFS(G,v):  
  label v as discovered  
  for all edges from v to w in G.adjacentEdges(v) do  
    if vertex w is not labeled as discovered then  
      recursively call DFS(G,w)
```

A non-recursive implementation of DFS:

```
procedure DFS-iterative(G,v):  
  let S be a stack  
  S.push(v)  
  while S is not empty  
    v = S.pop()  
    if v is not labeled as discovered:  
      label v as discovered  
      for all edges from v to w in G.adjacentEdges(v) do  
        S.push(w)
```

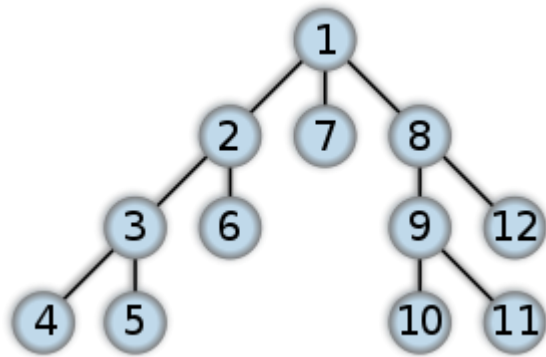
DFS (Depth First Search)



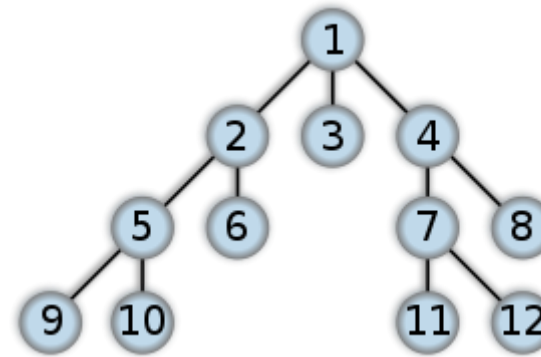
[https://en.wikipedia.org/wiki/Breadth-first\\_search, /Depth-first\\_search](https://en.wikipedia.org/wiki/Breadth-first_search, /Depth-first_search)

# Search Algorithms

DFS (Depth First Search)



BFS (Breadth First Search)



[https://en.wikipedia.org/wiki/Breadth-first\\_search](https://en.wikipedia.org/wiki/Breadth-first_search), /Depth-first\_search



# BFS Algorithm

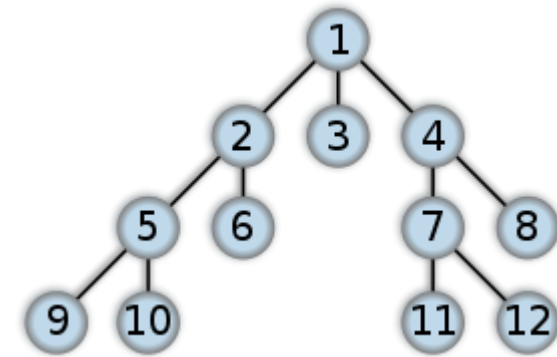
Breadth-First-Search(Graph, root):

create empty set S  
create empty queue Q

add root to S  
Q.enqueue(root)

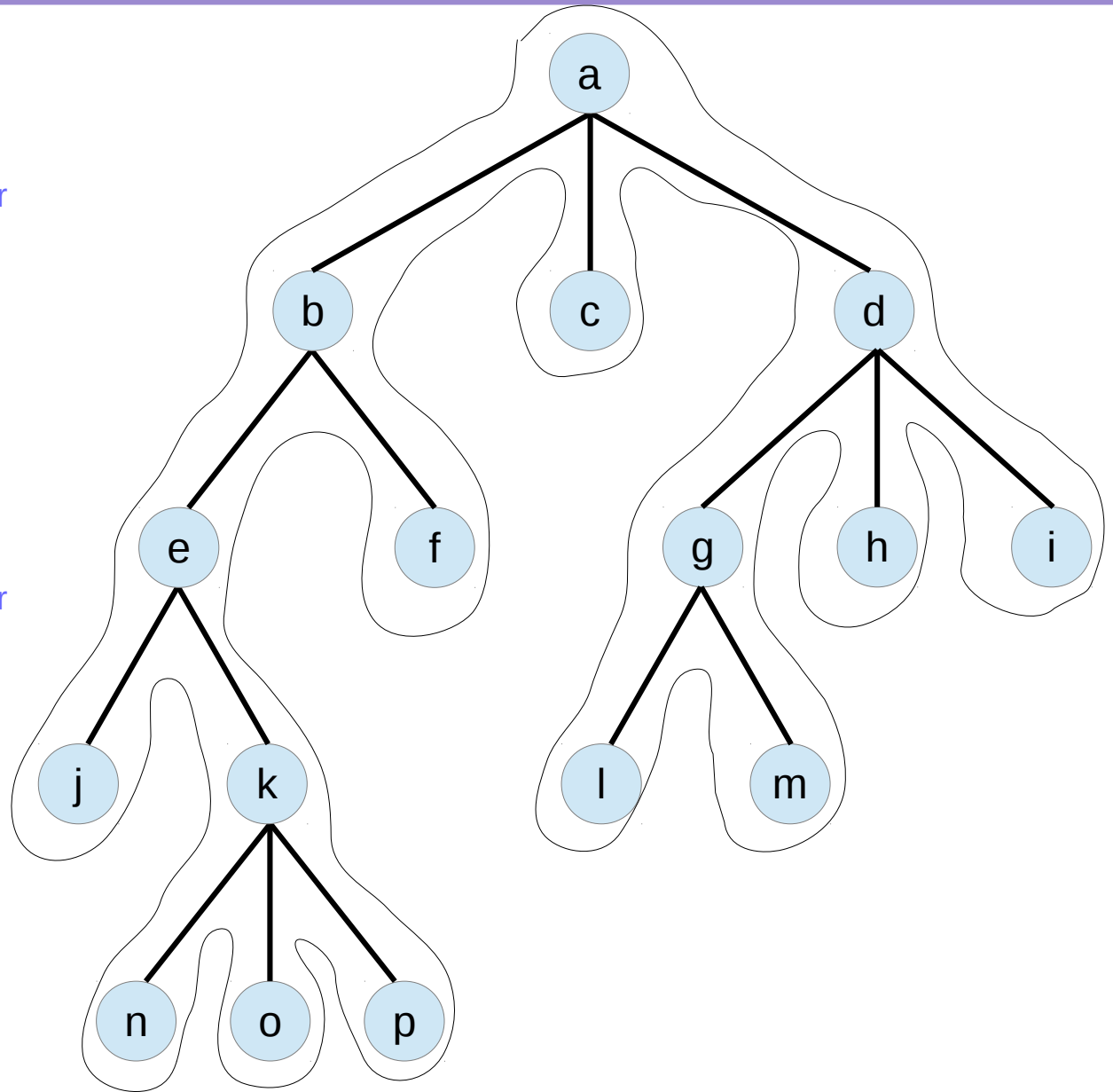
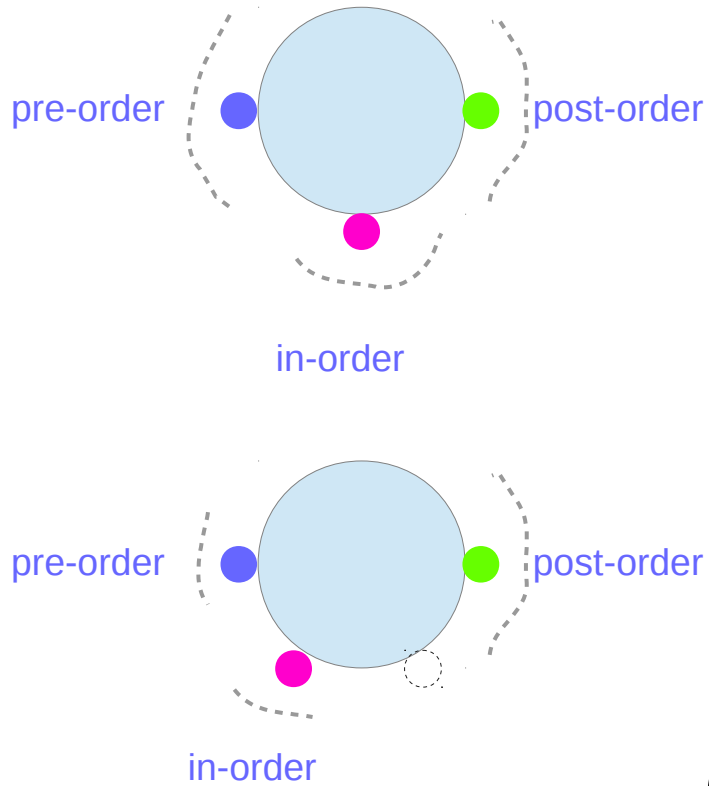
while Q is not empty:  
  current = Q.dequeue()  
  if current is the goal:  
    return current  
  for each node n that is adjacent to current:  
    if n is not in S:  
      add n to S  
      n.parent = current  
      Q.enqueue(n)

BFS (Breadth First Search)



[https://en.wikipedia.org/wiki/Breadth-first\\_search](https://en.wikipedia.org/wiki/Breadth-first_search), /Depth-first\_search

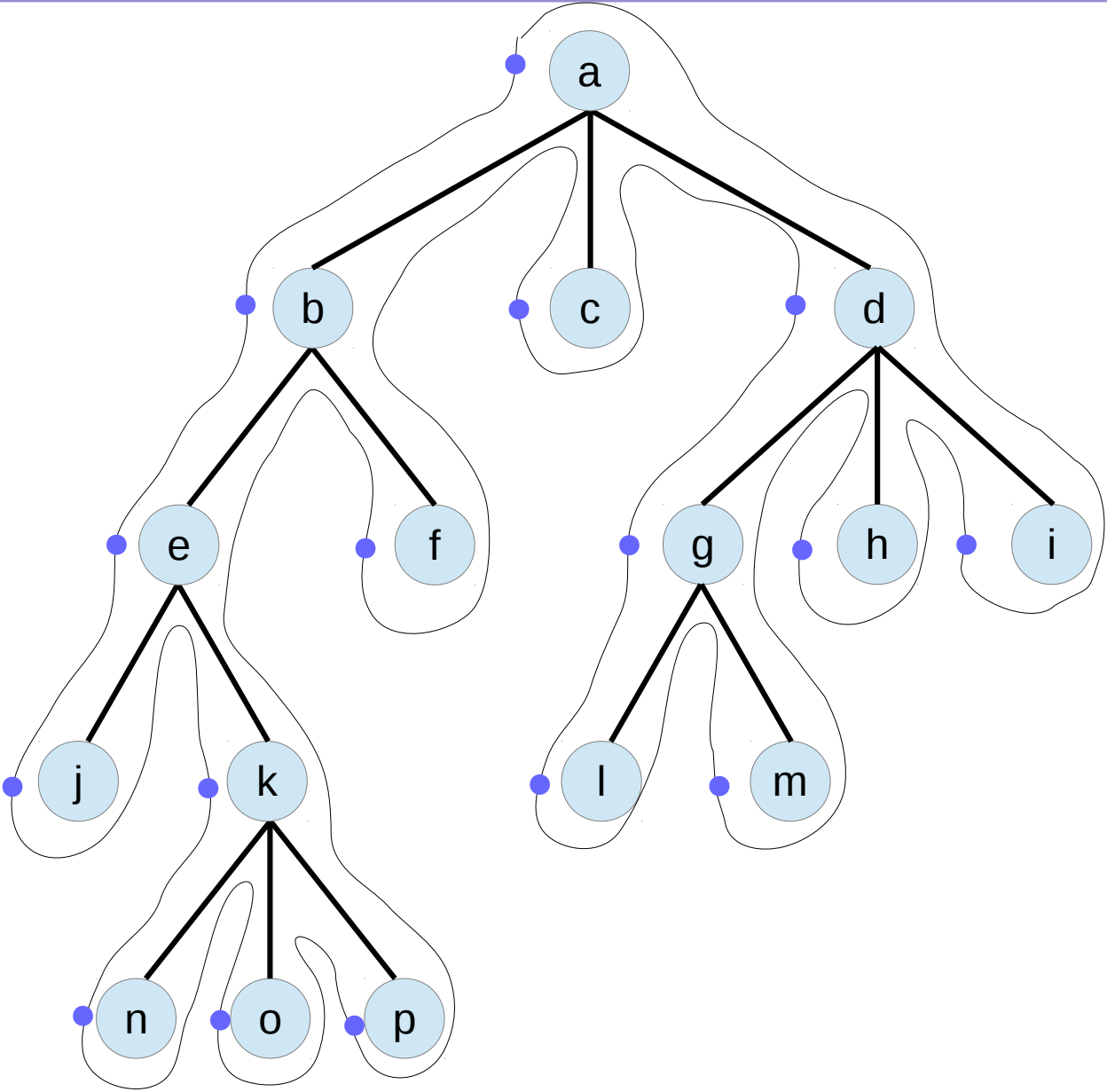
# In-Order



Rosen

# Ternary Tree

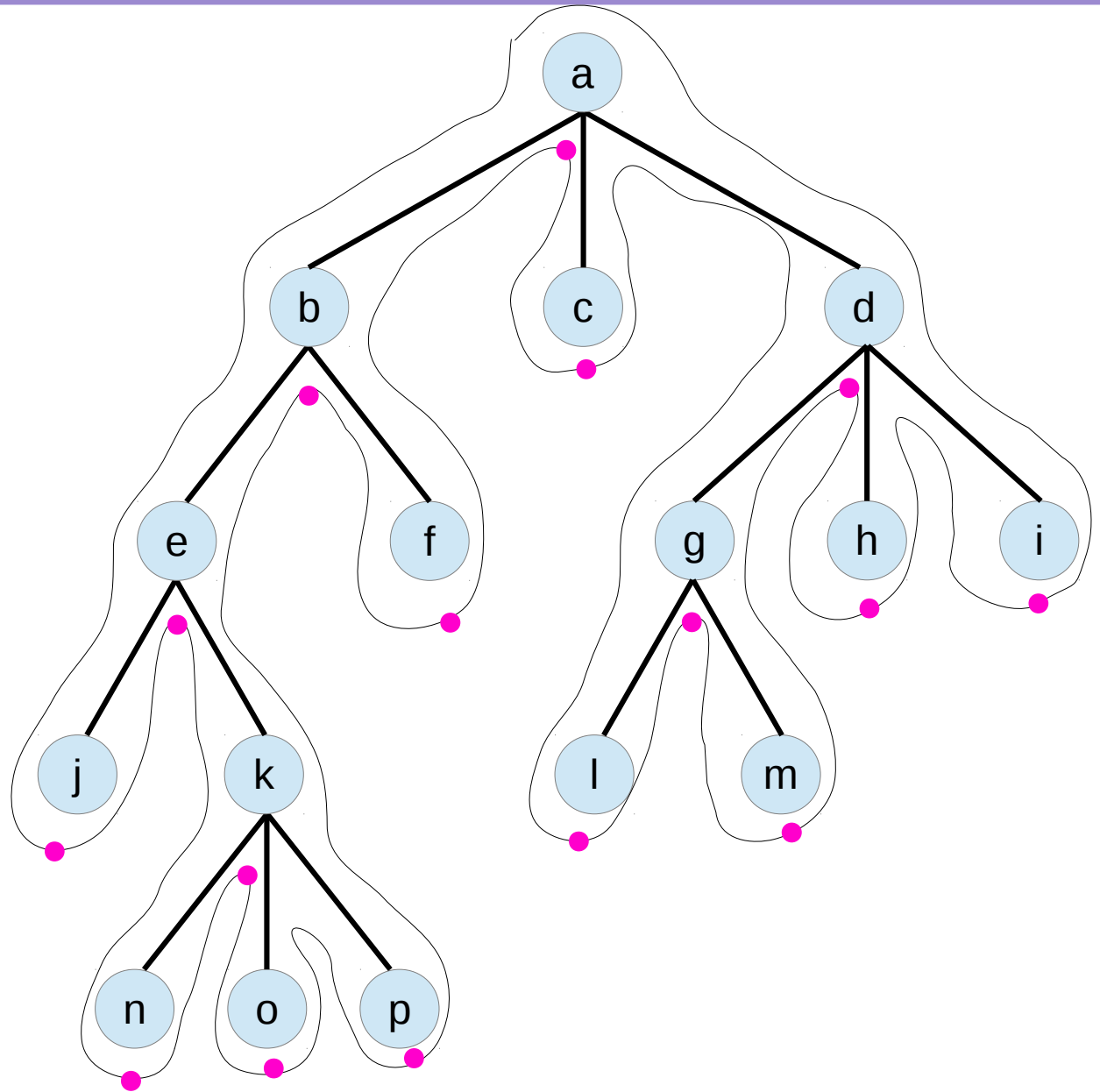
a-b-e-j-k-n-o-p-f-c-d-g-l-m-h-i



Rosen

# In-Order

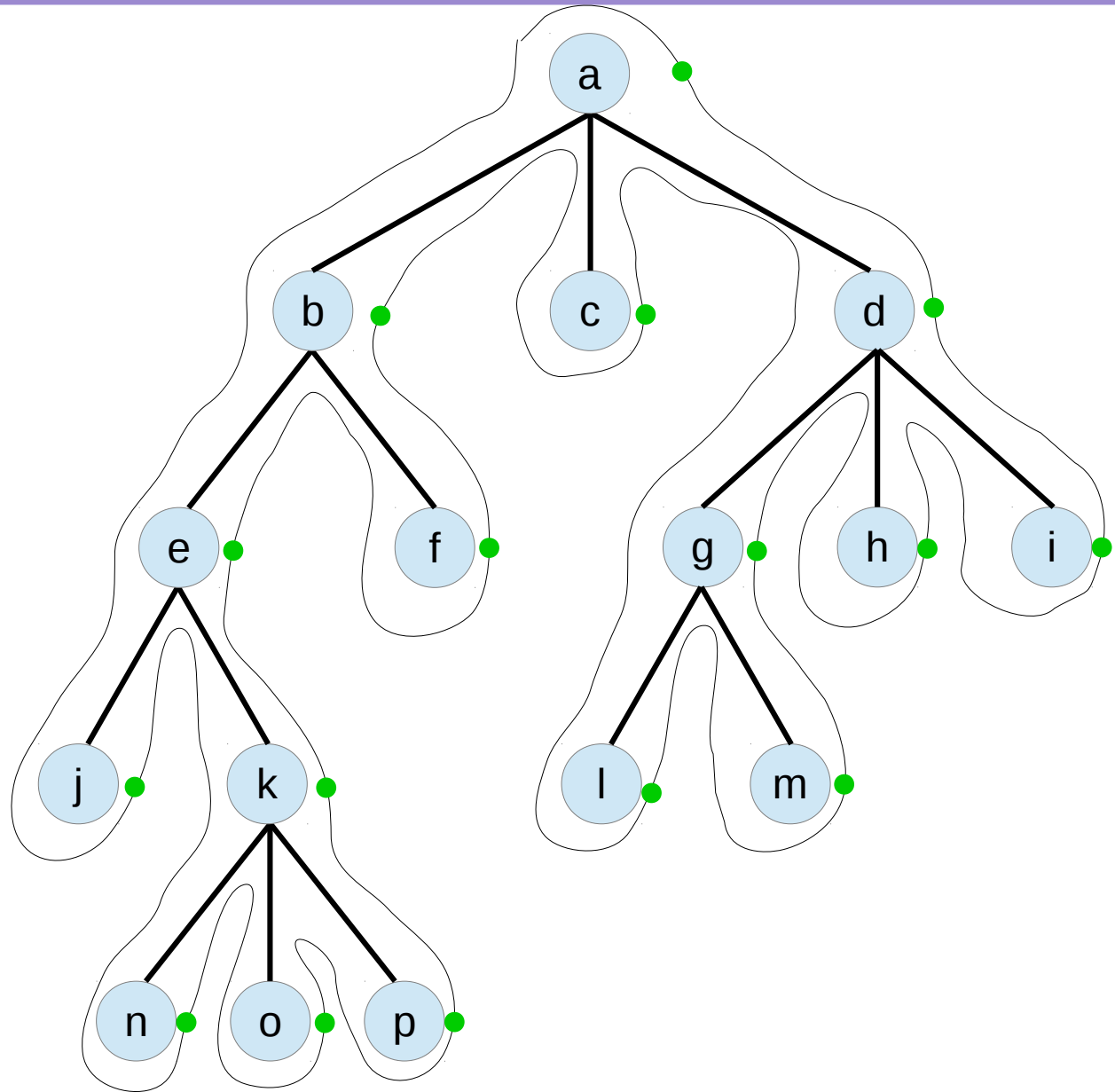
j-e-n-k-o-p-b-f-a-c-l-g-m-d-h-i



Rosen

# Post-Order

j-n-o-p-k-e-f-b-c-l-m-g-h-i-d-a



Rosen

# Ternary

## Ternary

### Etymology

Late Latin ternarius (“consisting of three things”), from terni (“three each”).

### Adjective

ternary (not comparable)

Made up of three things; treble, triadic, triple, triplex

Arranged in groups of three

(mathematics) To the base three [quotations ▼]

(mathematics) Having three variables

<https://en.wiktionary.org/wiki/ternary>

The sequence continues with **quaternary**, **quinary**, **senary**, **septenary**, **octonary**, **nonary**, and **denary**, although most of these terms are rarely used. There's no word relating to the number eleven but there is one that relates to the number twelve: **duodenary**.

<https://en.oxforddictionaries.com/explore/what-comes-after-primary-secondary-tertiary>

## References

- [1] <http://en.wikipedia.org/>
- [2]