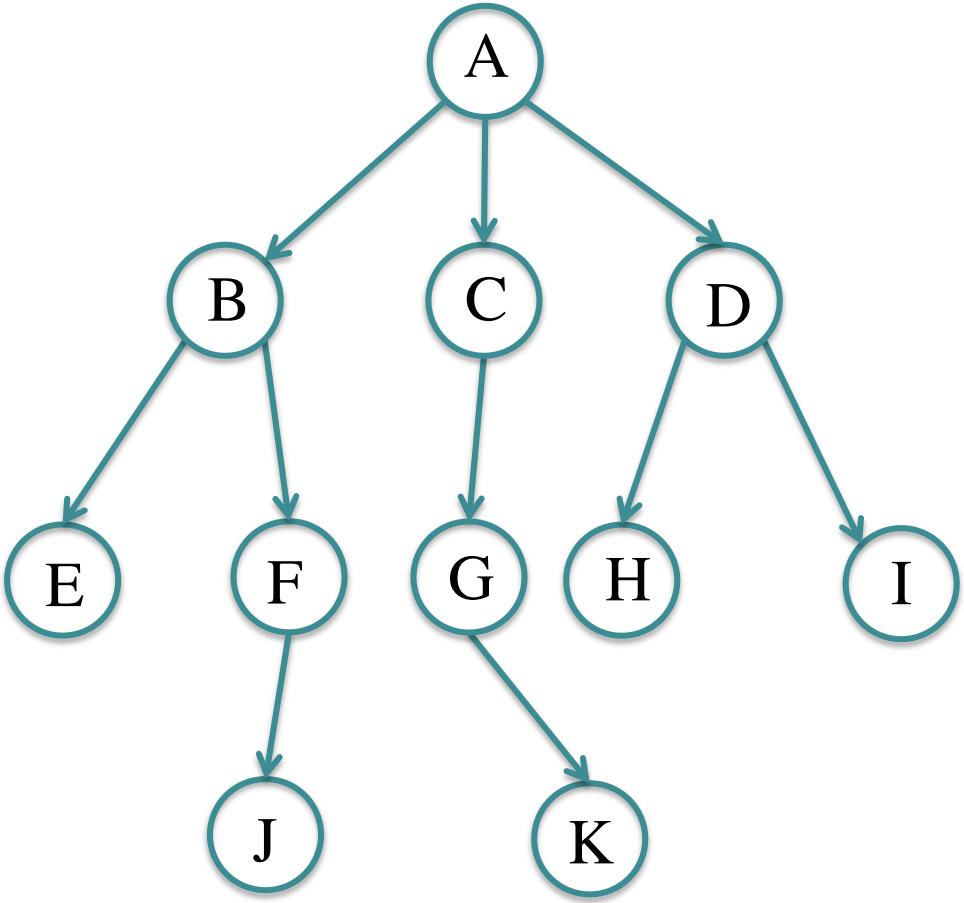
Topic 5

Non linear data structure

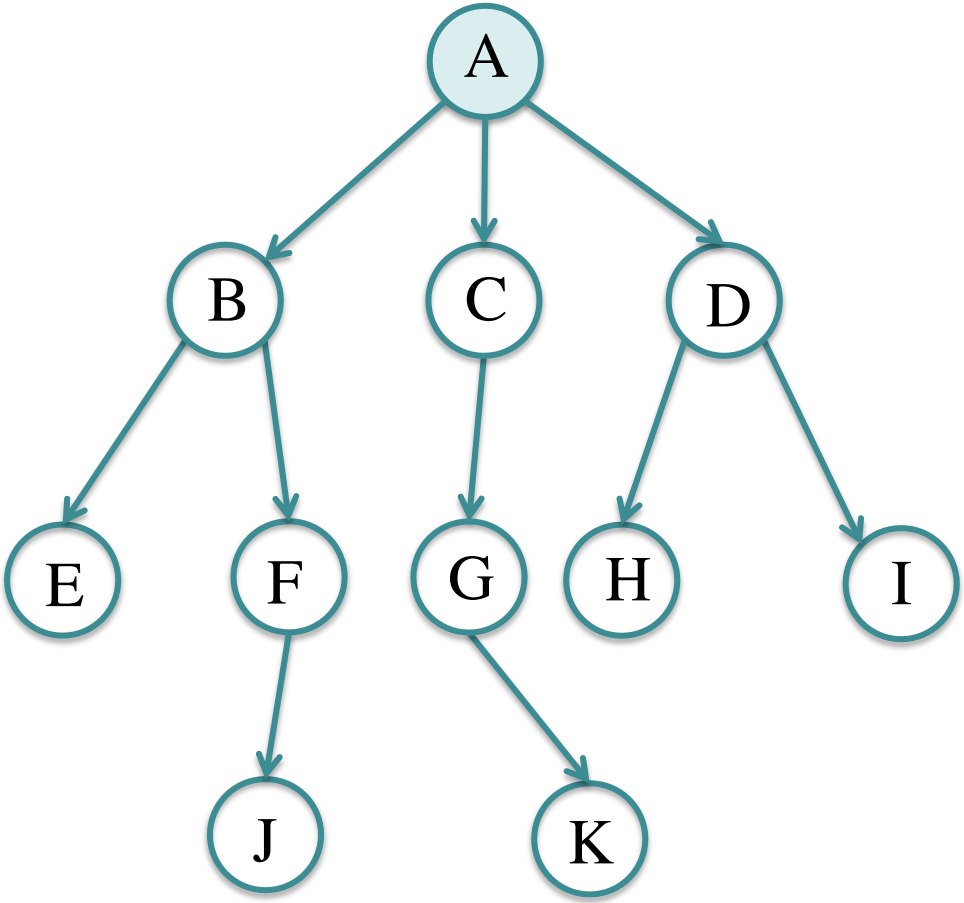
Trees

Intro

Hierarchical data structure

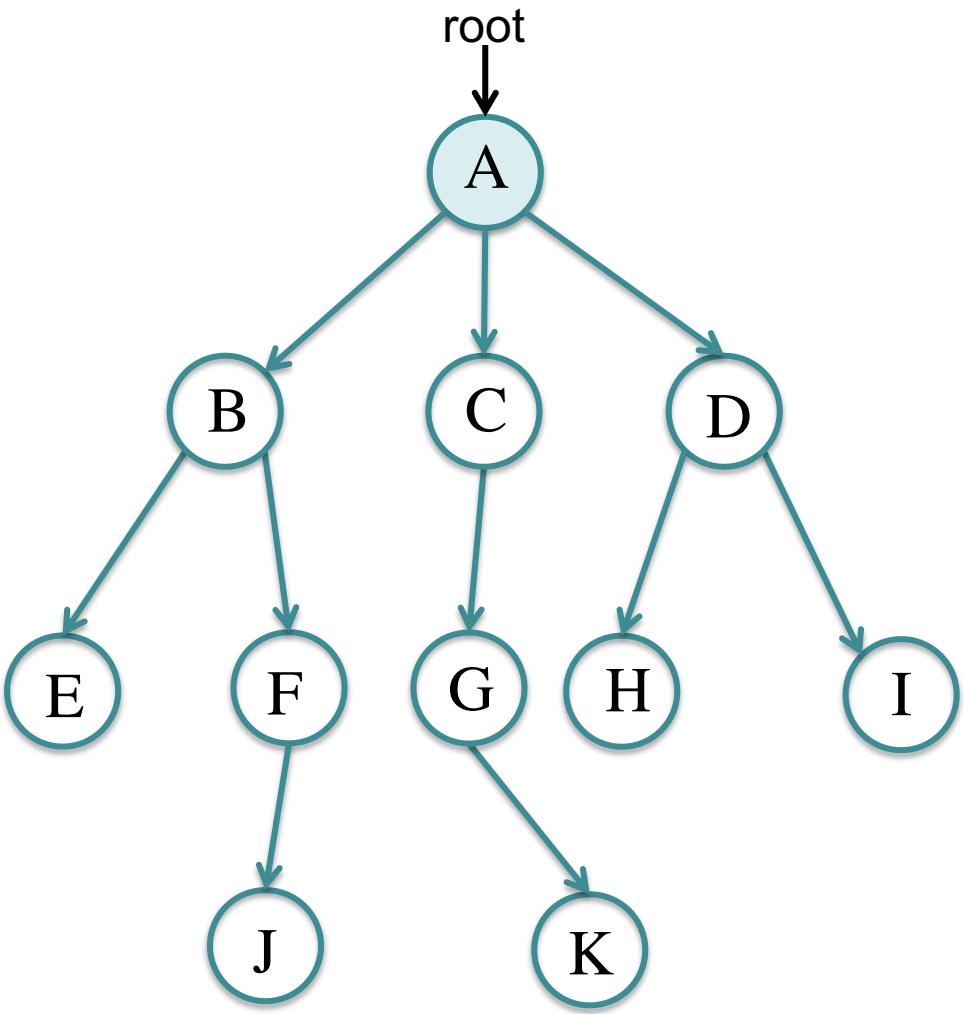


Tree data structure



The node at the top of the tree:

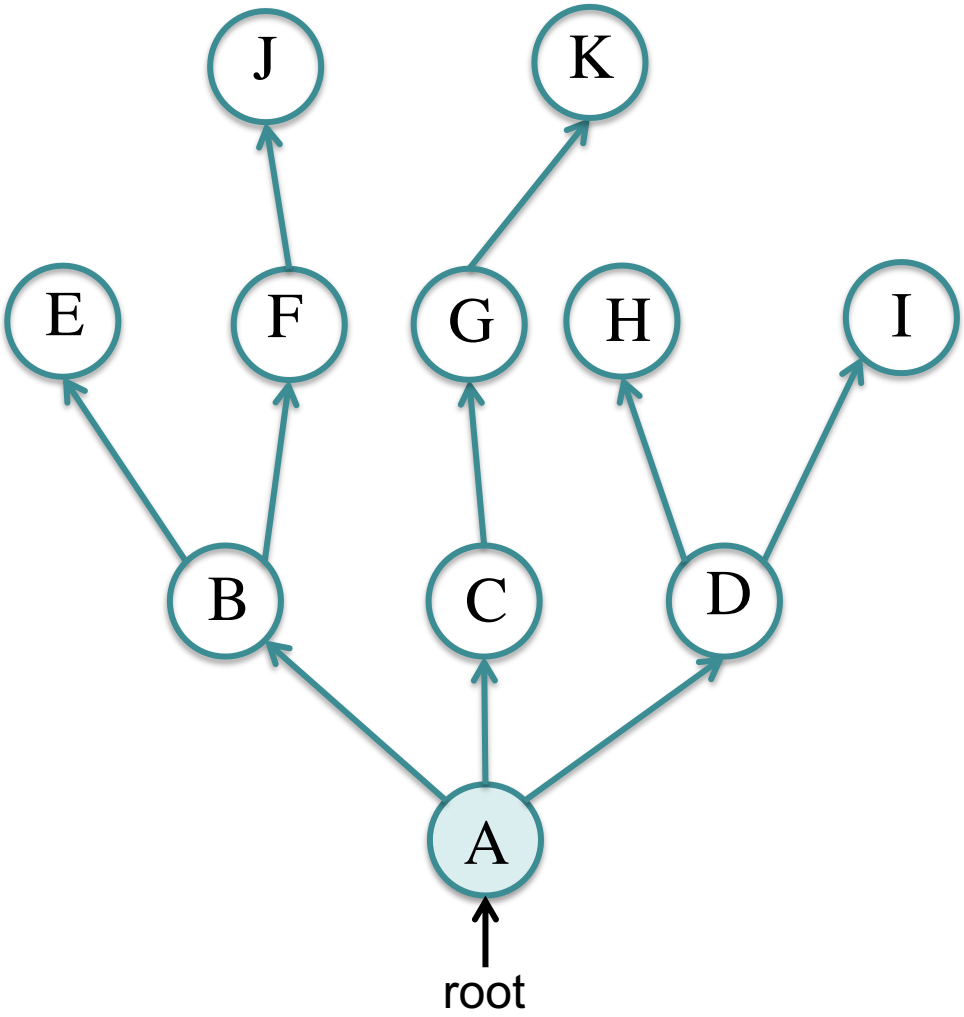
- Initial node of the tree



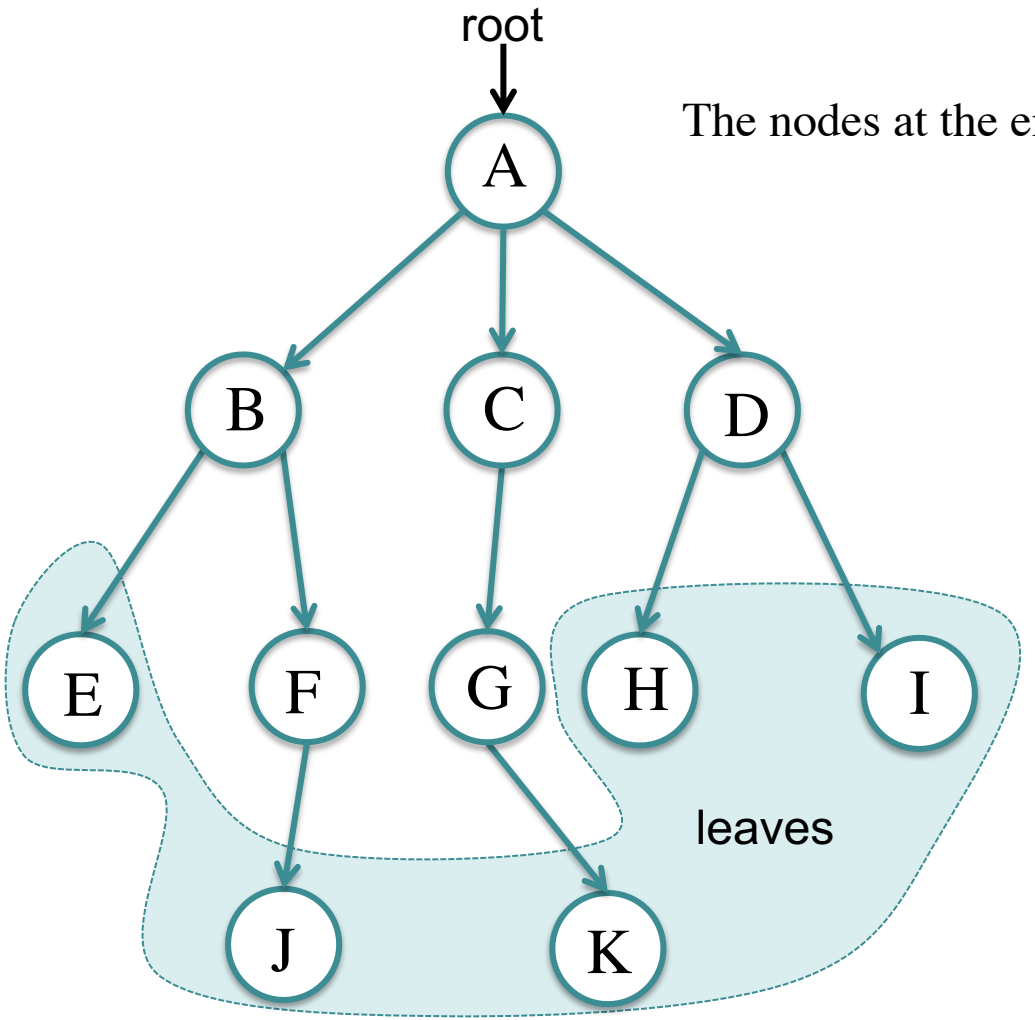
The node at the top of the tree:

- initial node of the tree
- called the root of the tree

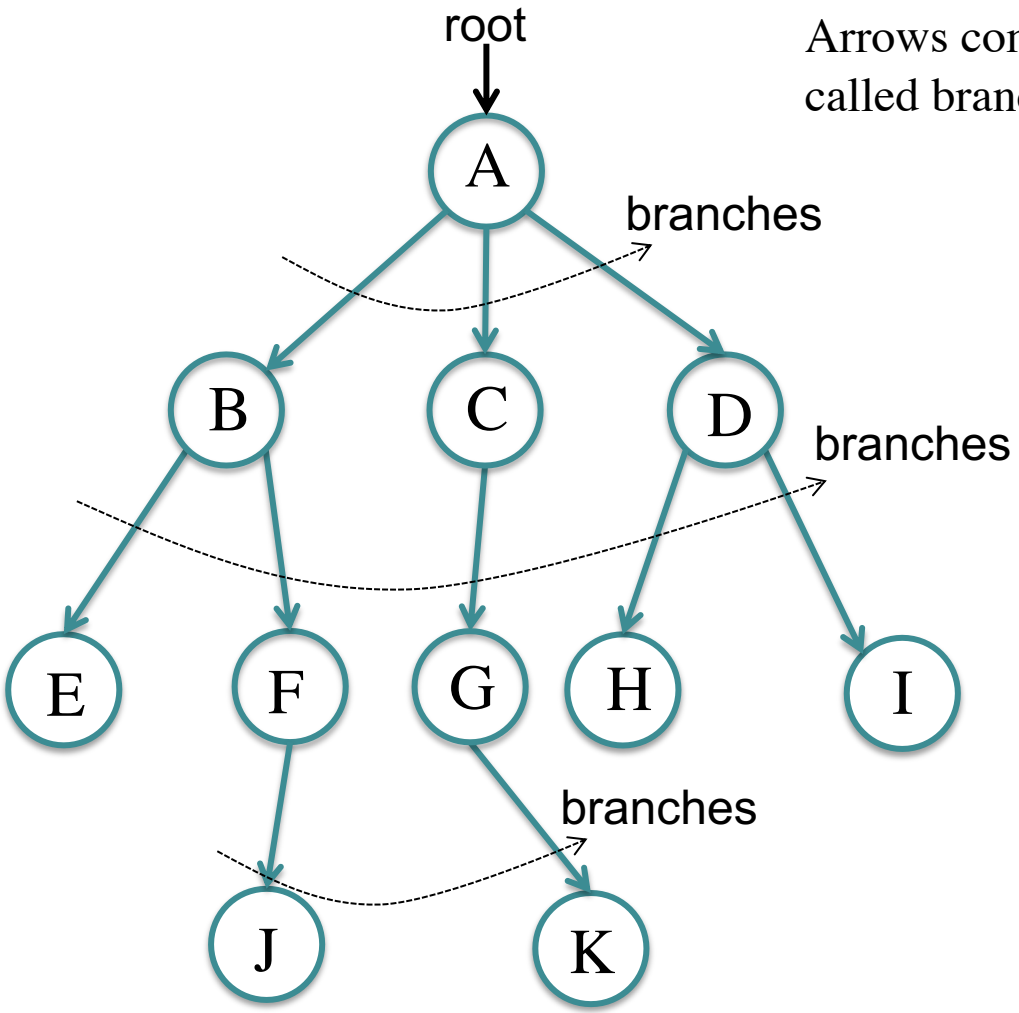
A tree has a pointer that points to the root of the tree



A tree data structure is an upside-down tree !!!

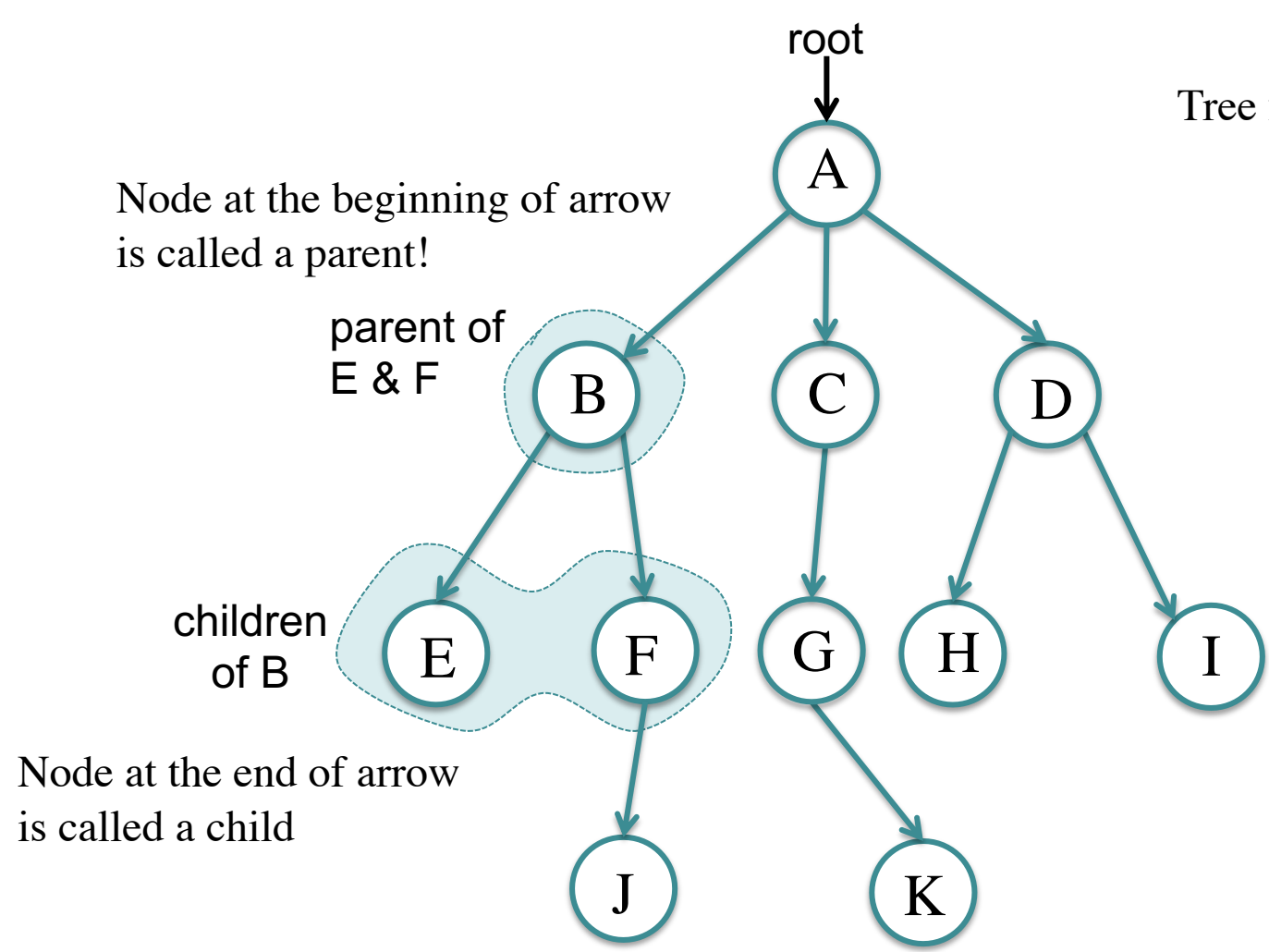


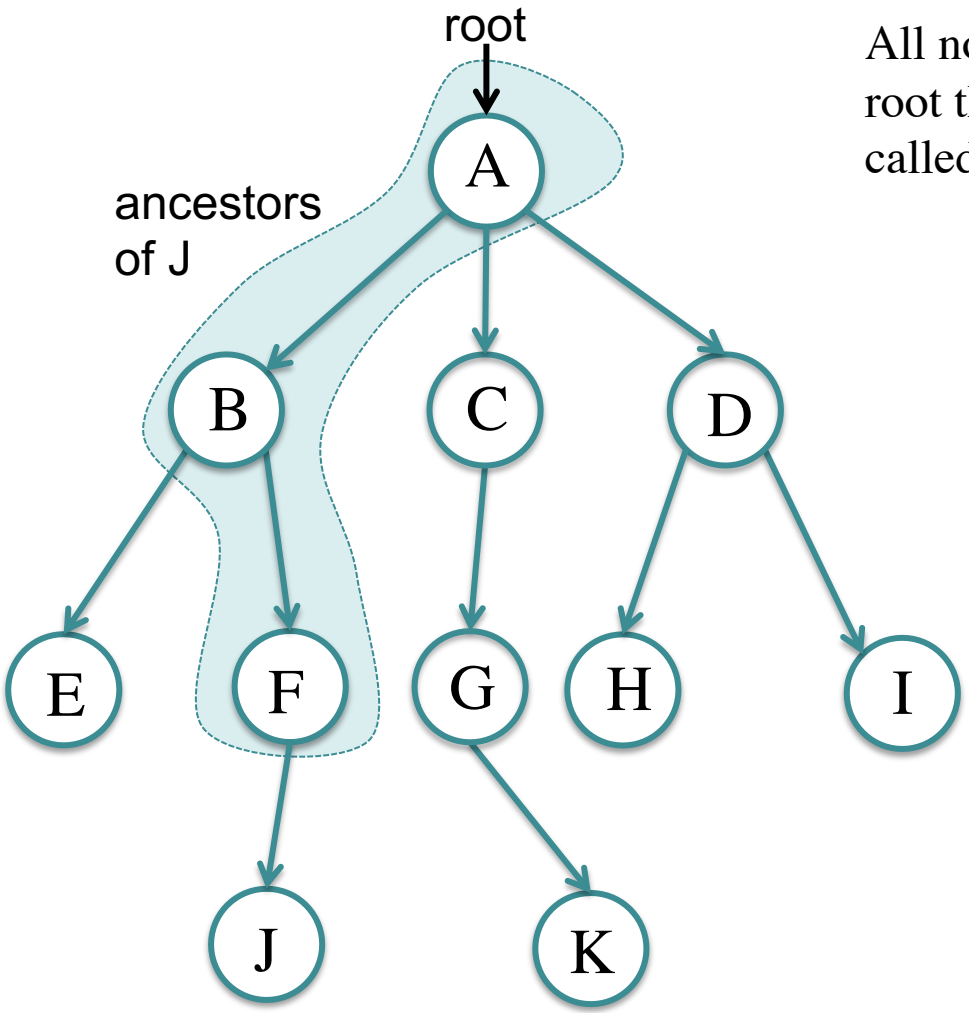
The nodes at the end of tree at called leaves



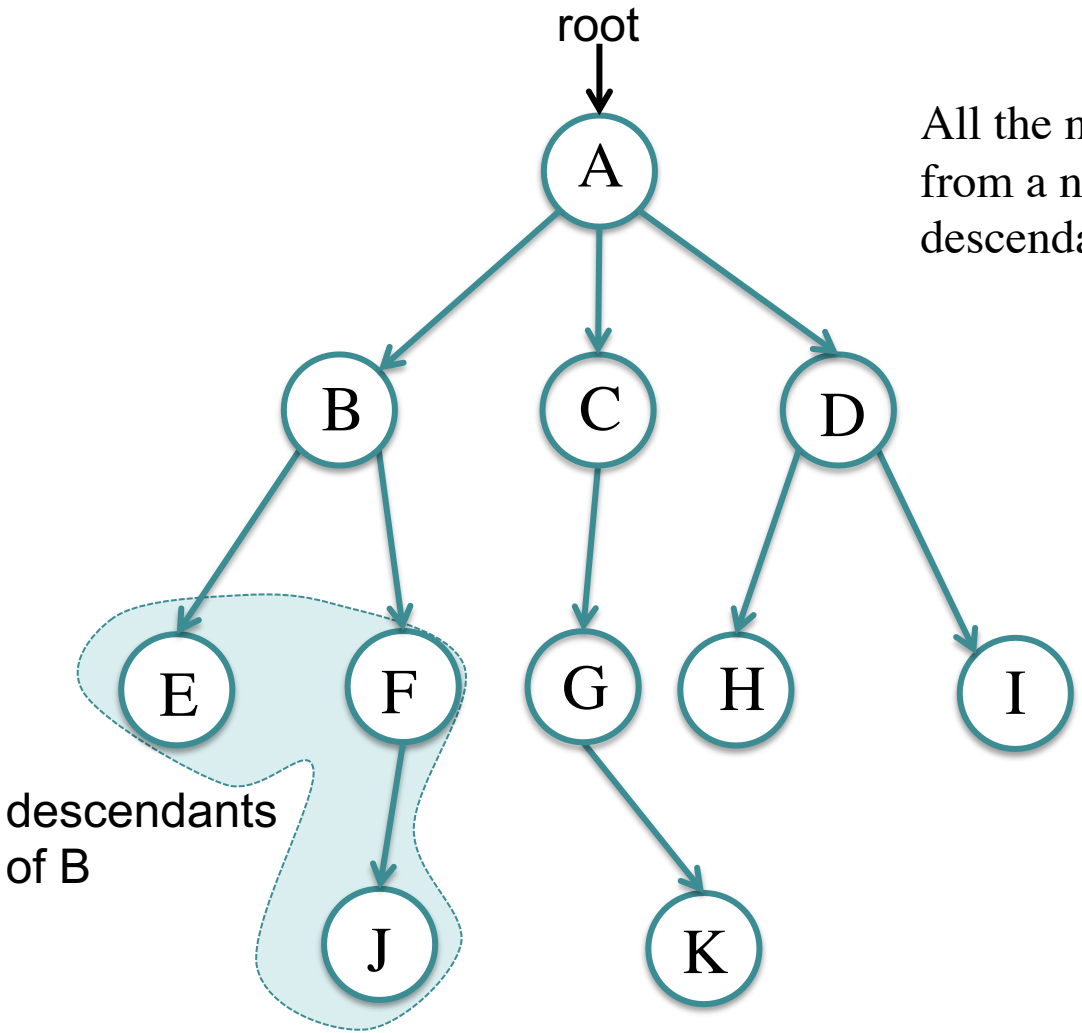
Arrows coming out of node are called branches

Tree family analogy

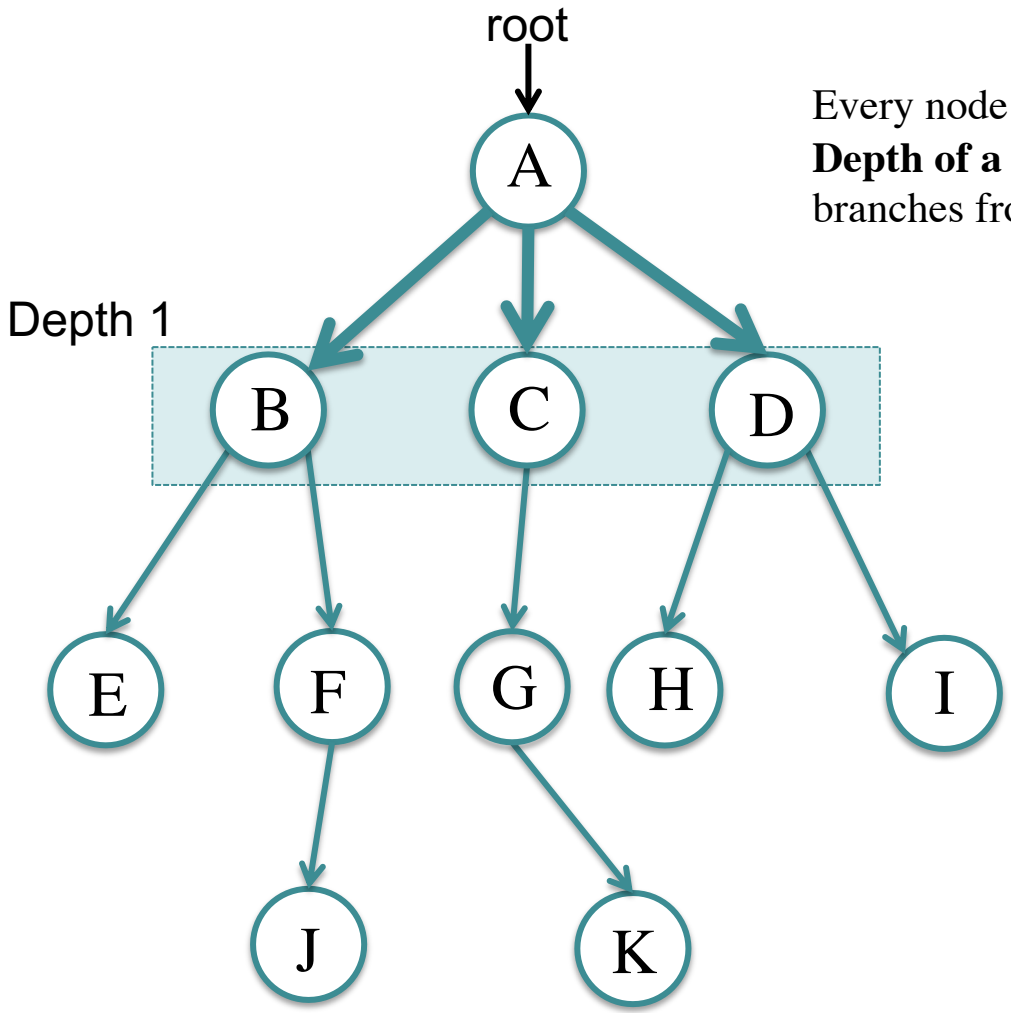




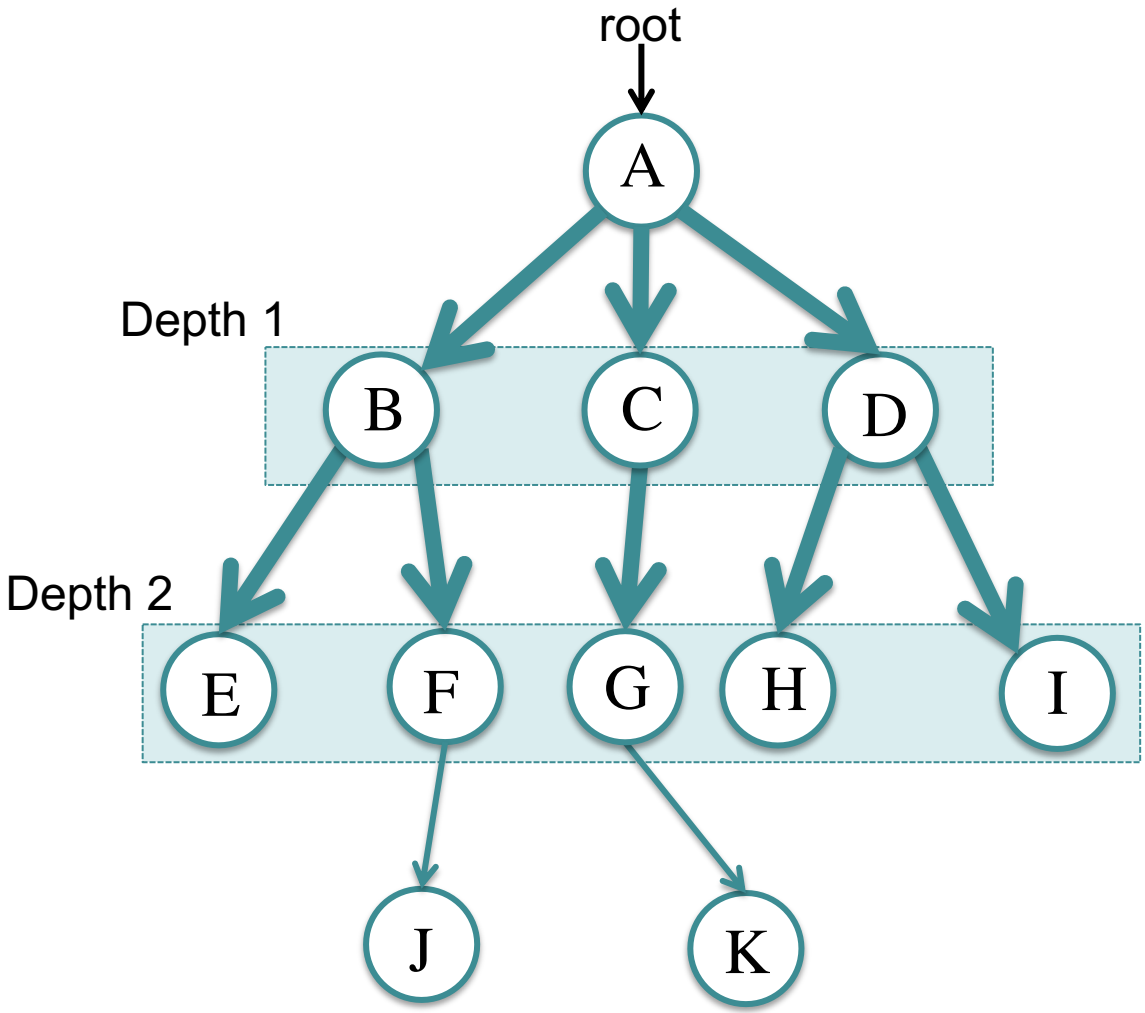
All nodes in the path from the root that leads to a node x are called ancestors of x

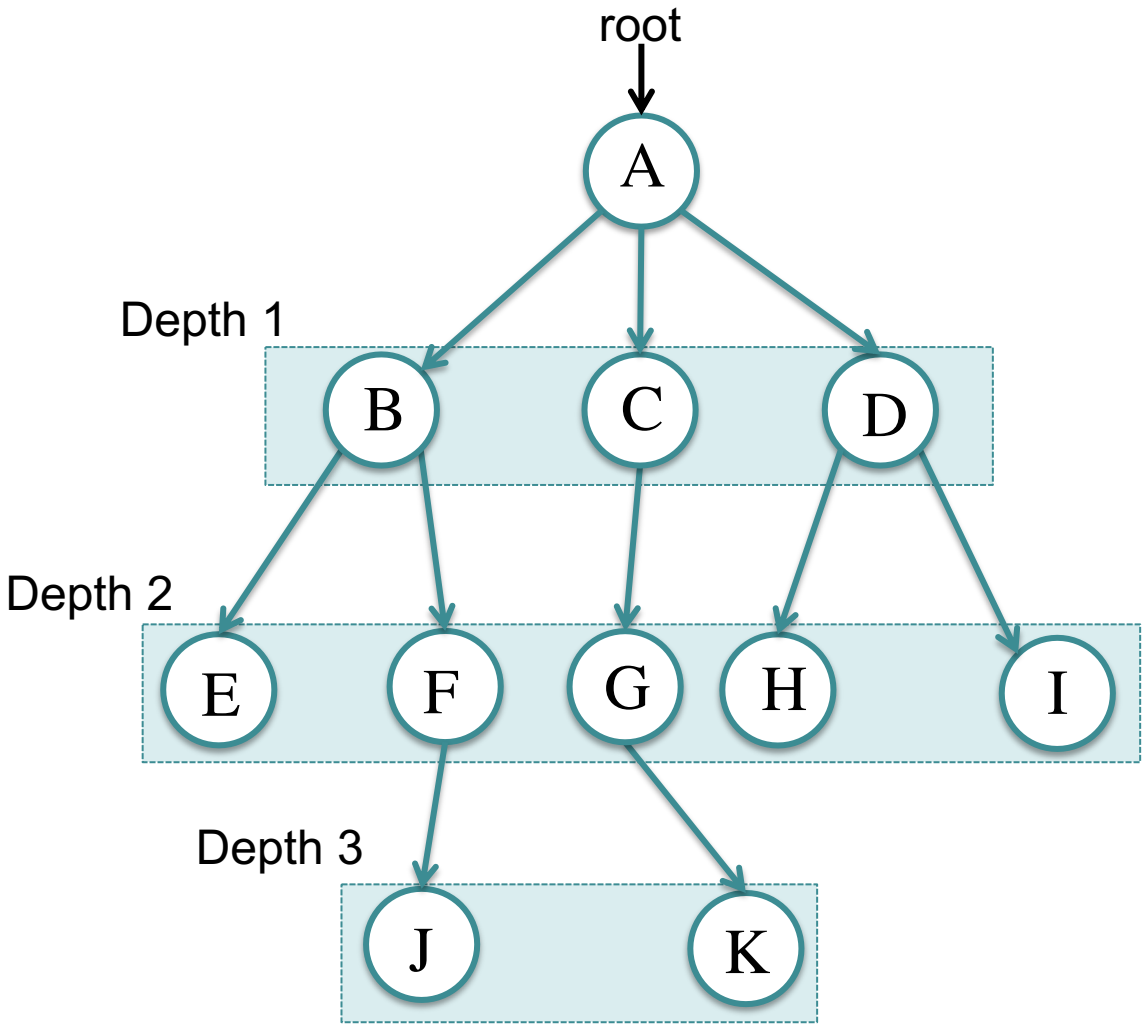


All the nodes on the way down from a node x are called descendants of x.

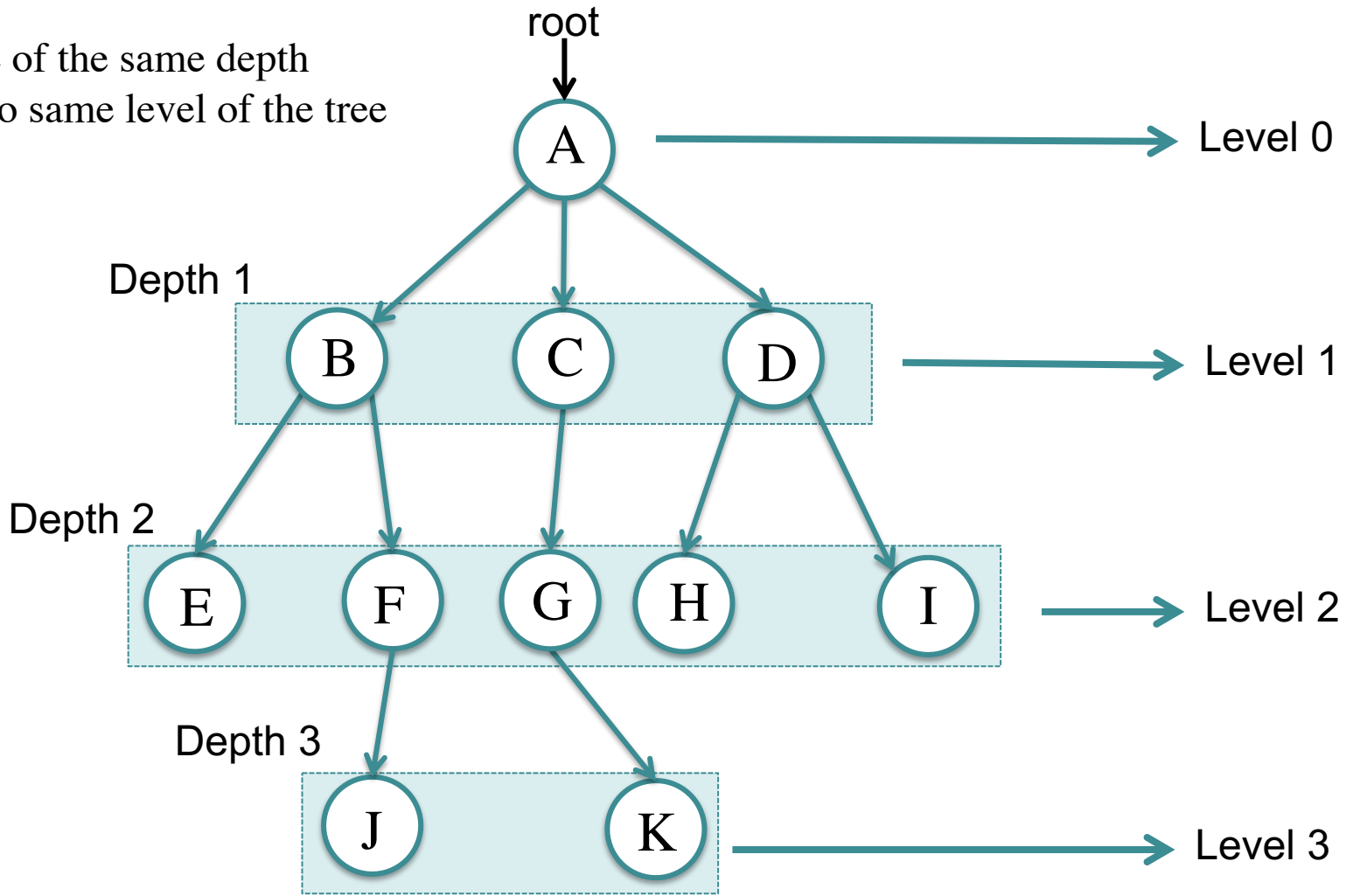


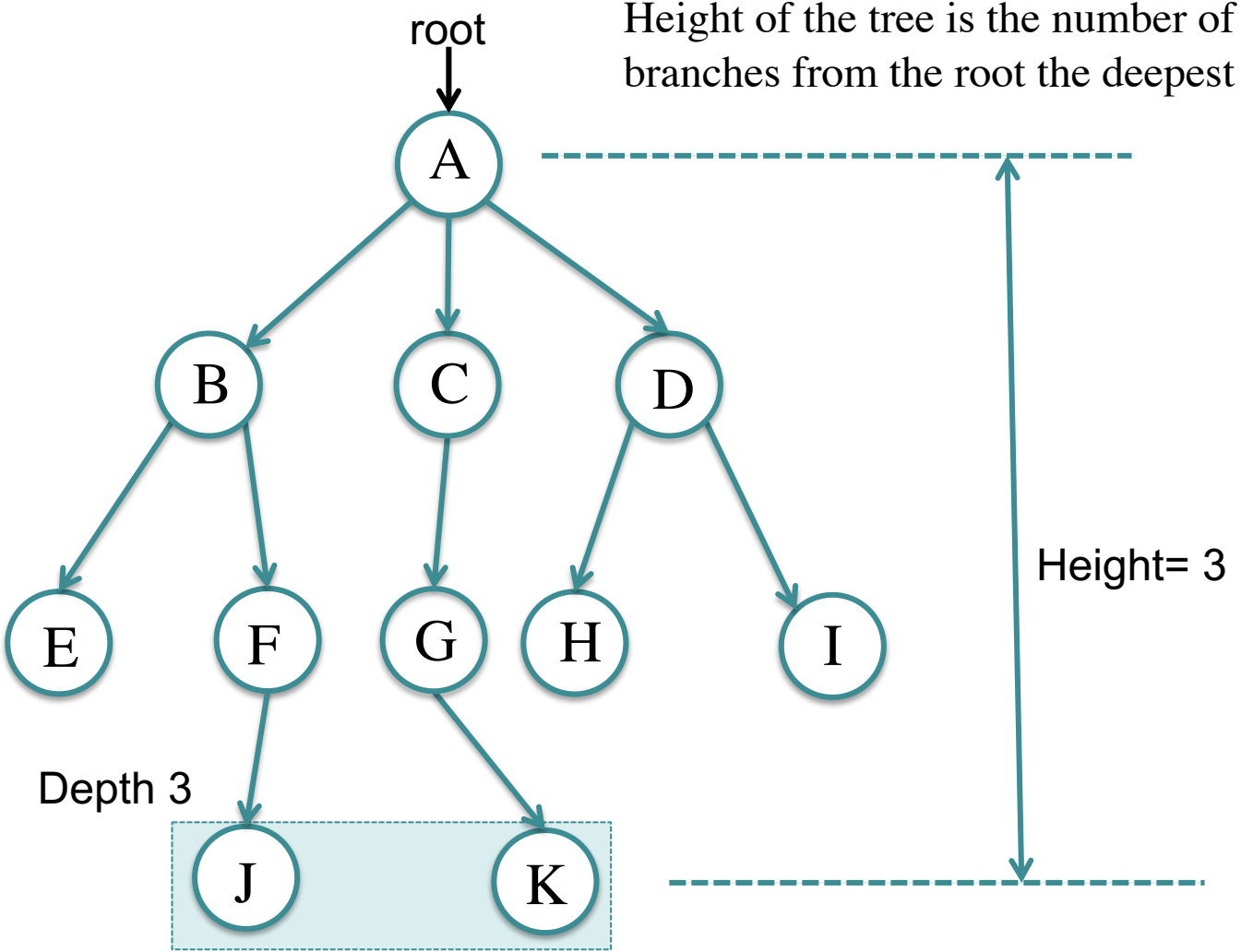
Every node of a tree has a depth!
Depth of a node X: is the number of branches from the root to X.

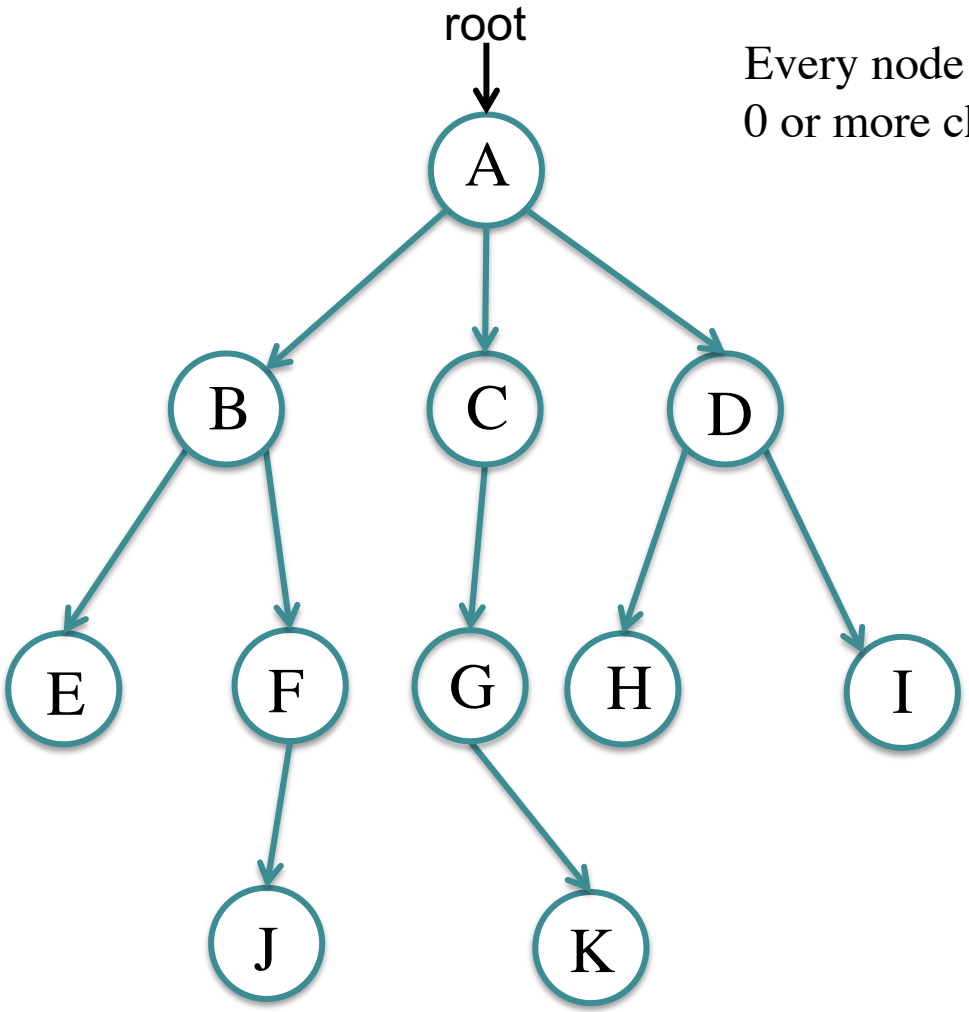

root
A
Depth 1
B
C
D
Depth 2
E
F
G
H
I
J
K


root
A
Depth 1
B
C
D
Depth 2
E
F
G
H
I
Depth 3
J
K

The node of the same depth belongs to same level of the tree

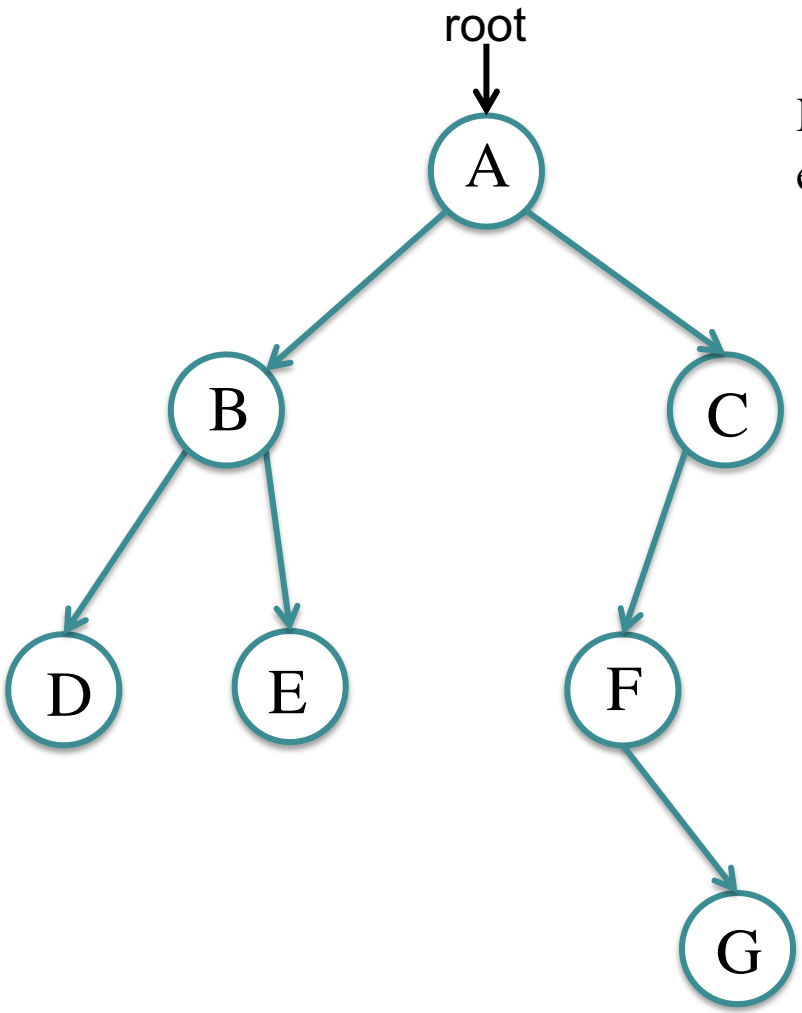



root
Height of the tree is the number of branches from the root the deepest leave.
A
B
C
D
E
F
G
H
I
J
K
Depth 3
Height= 3



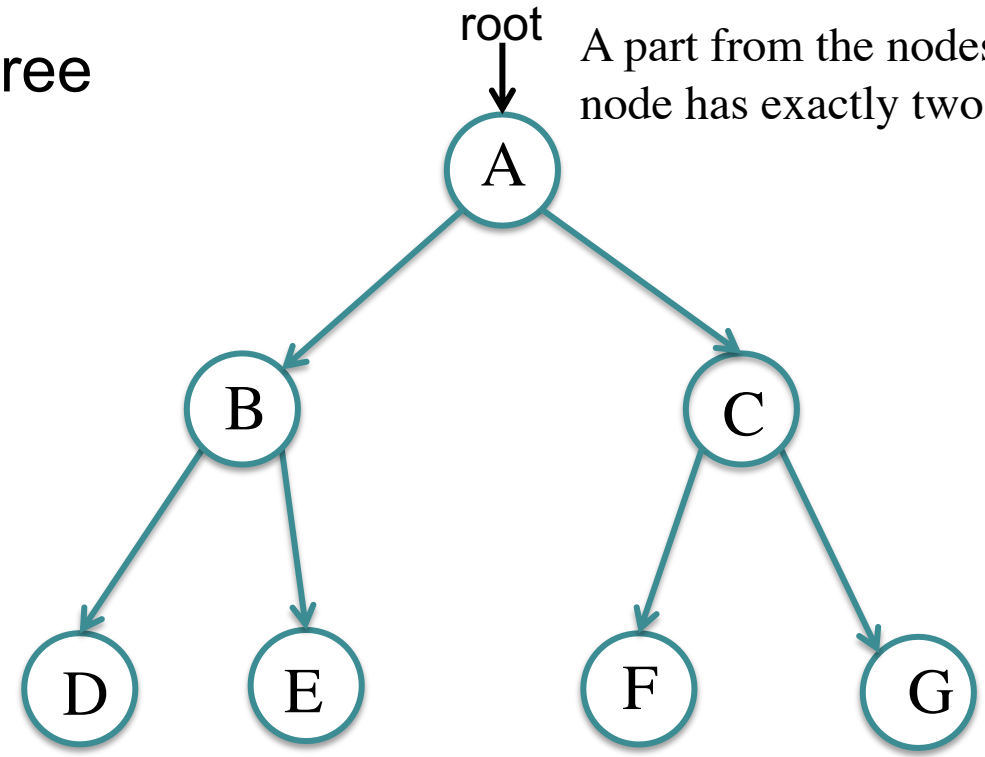
Every node in a generic tree can have 0 or more children!

Binary tree



Number of children is less or equal to 2

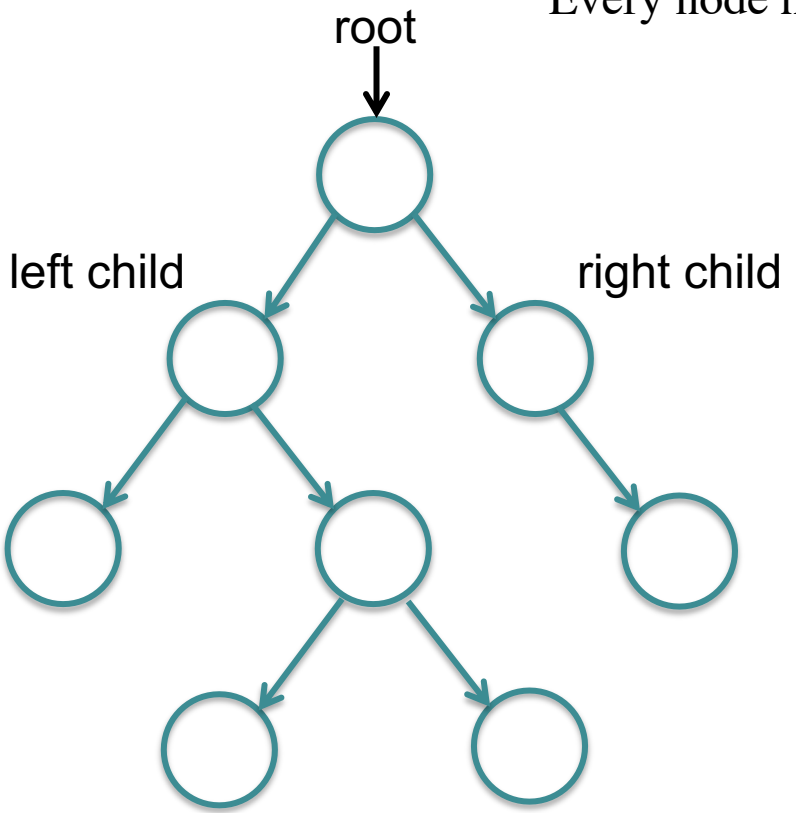
Full binary tree

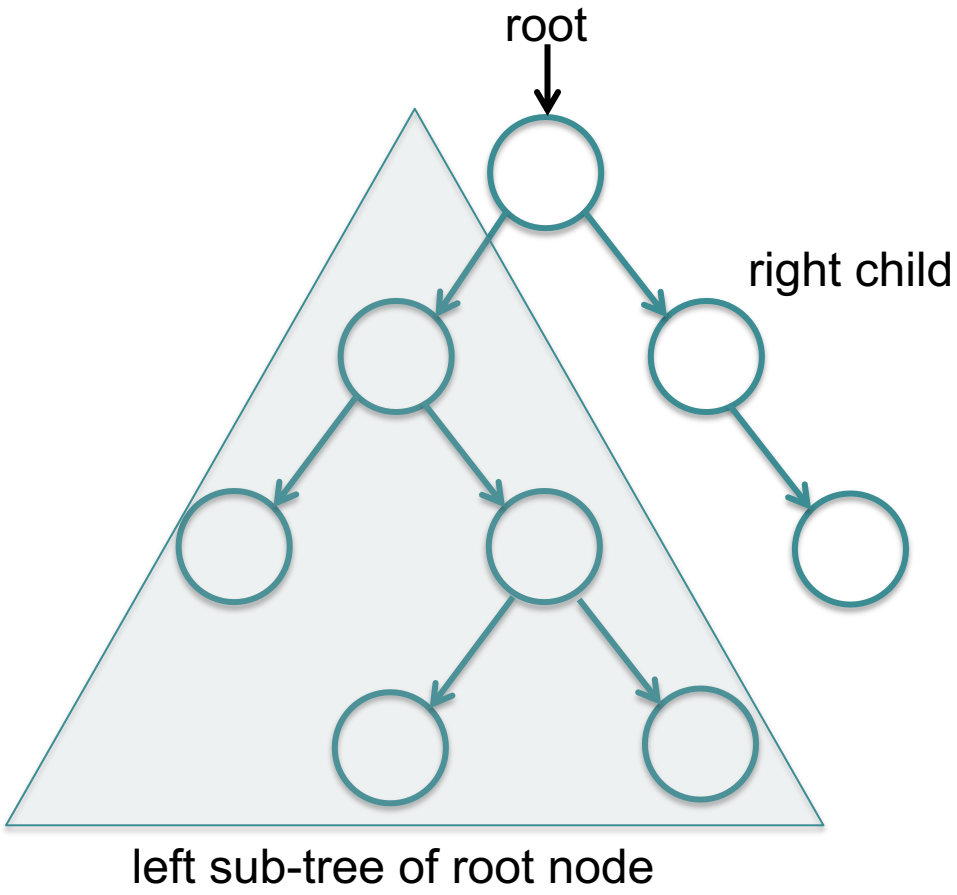


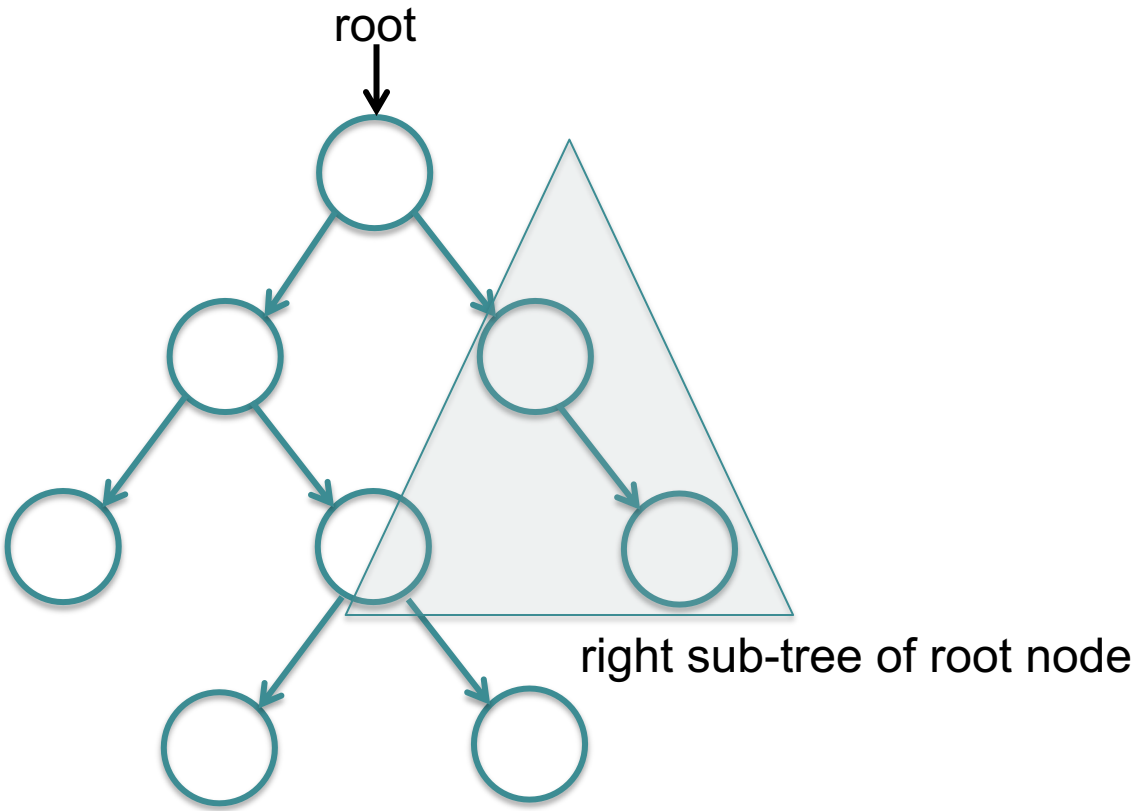
A part from the nodes at the last level, every node has exactly two children!

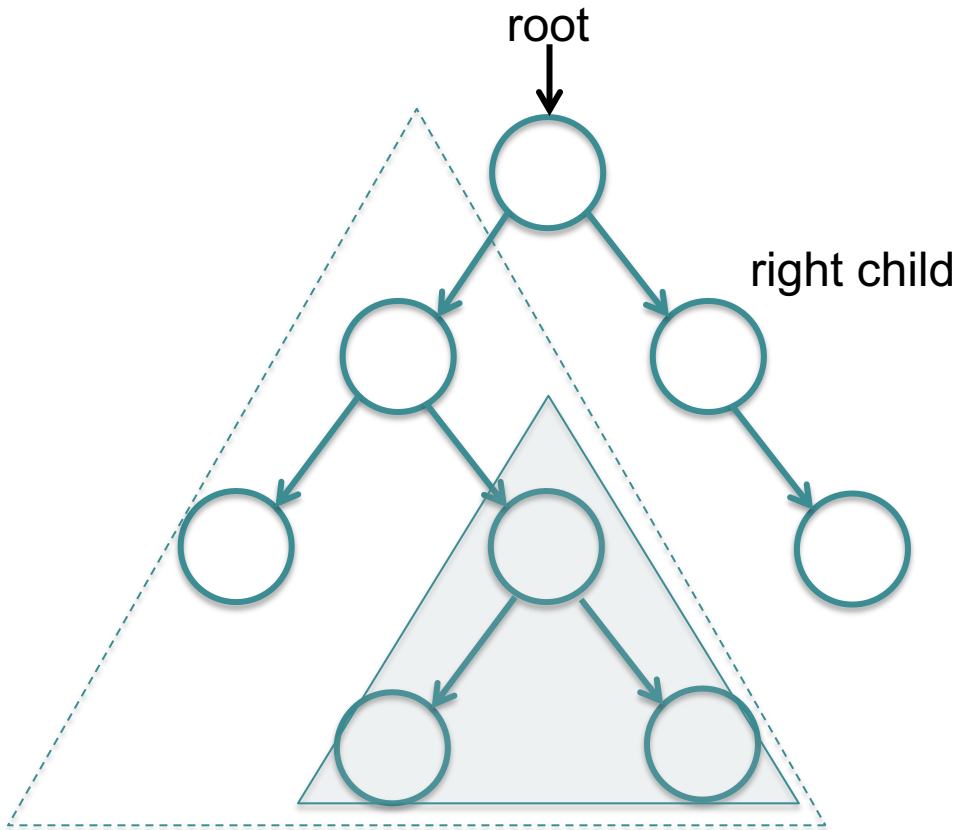
Binary tree

Every node has maximum of 2 children



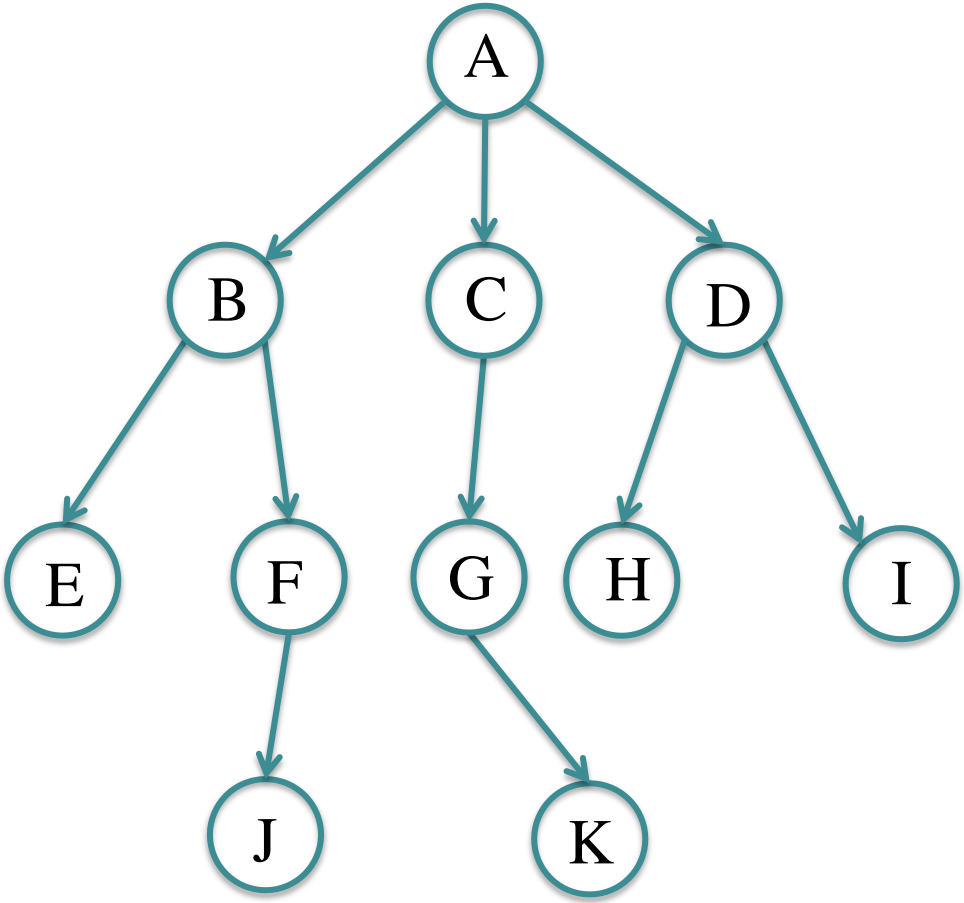

root
right child
left sub-tree of root node


root
right sub-tree of root node



There are other subtrees inside a subtree! Recursion!!!!!!

This tree characteristic allows a recursive implementation!



Summary:

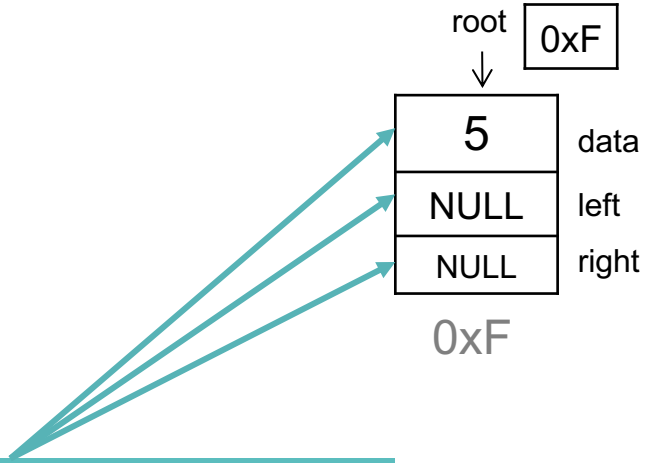
- Definition of a tree
- Branches and leaves
- Ancestors and descendant of a node
- Height of a tree
- Binary trees
- Next, we will focus on binary trees

Binary Trees Implementation

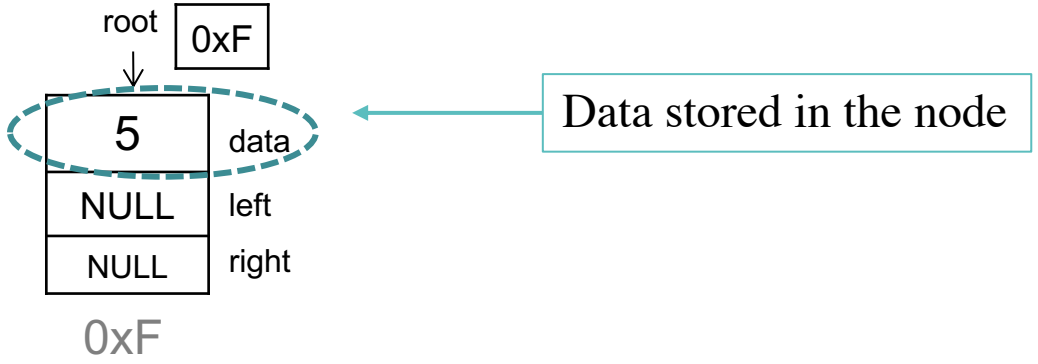
Two ways of implementing a binary tree:

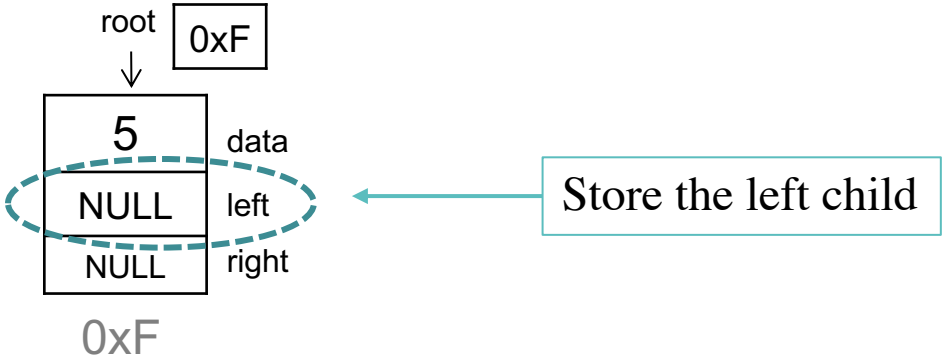
- Memory pointers
- Arrays

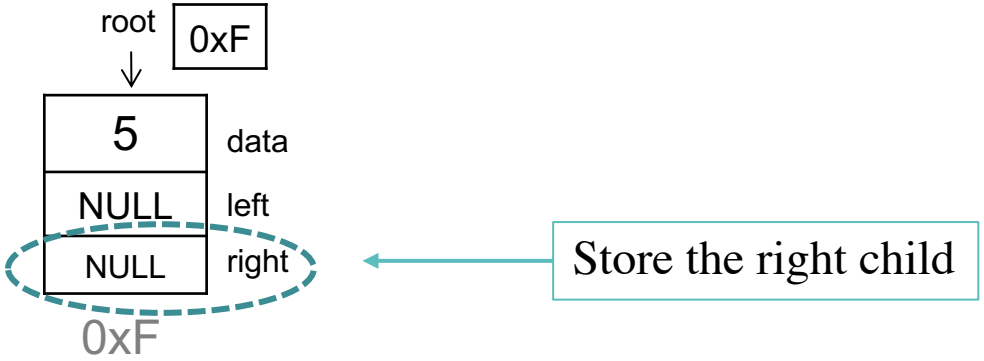
Memory pointer

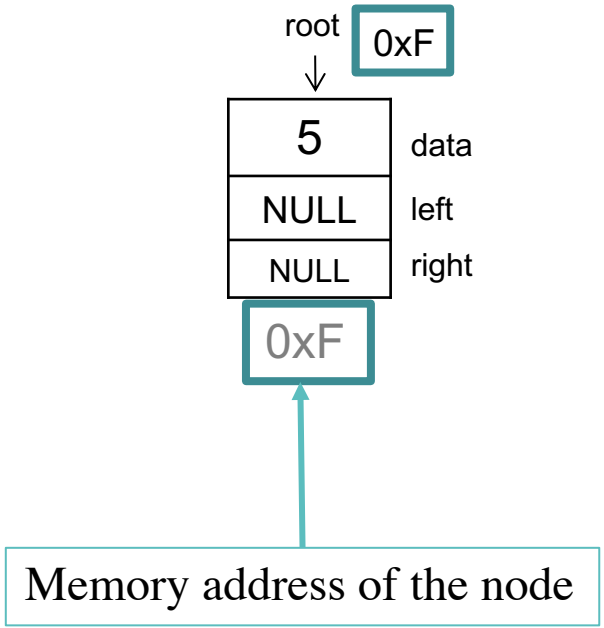


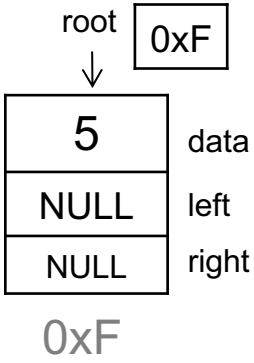
Define a node with 3 memory spaces


root
0xF
5
data
NULL
left
NULL
right
0xF
Data stored in the node

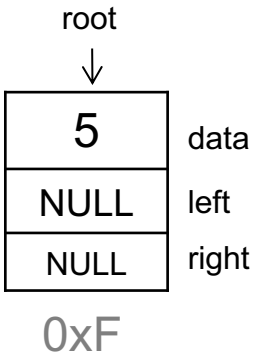

root
0xF
5
data
NULL
left
NULL
right
0xF
Store the left child


root
0xF
5
data
NULL
left
NULL
right
0xF
Store the right child

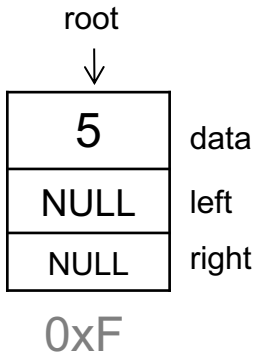

root
0xF
5
data
NULL
left
NULL
right
0xF
Memory address of the node

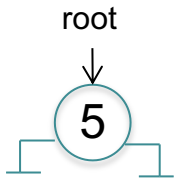


Example:
Node that stores the value 5 and has no children as it is not connected to any other node

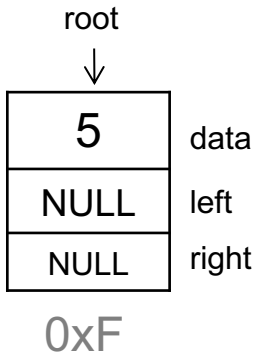


Abstract representation of this tree

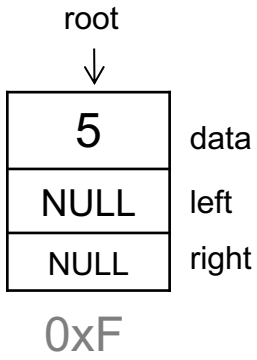


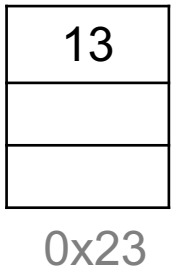


More explicit Abstract representation of this tree

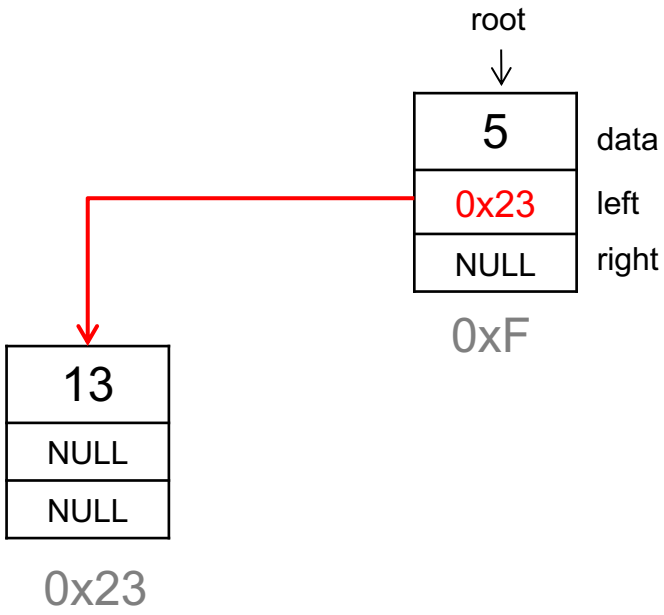


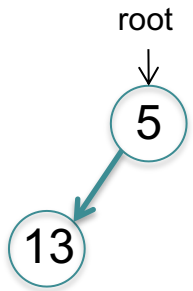
Let’s now try to add nodes to this tree!!!!

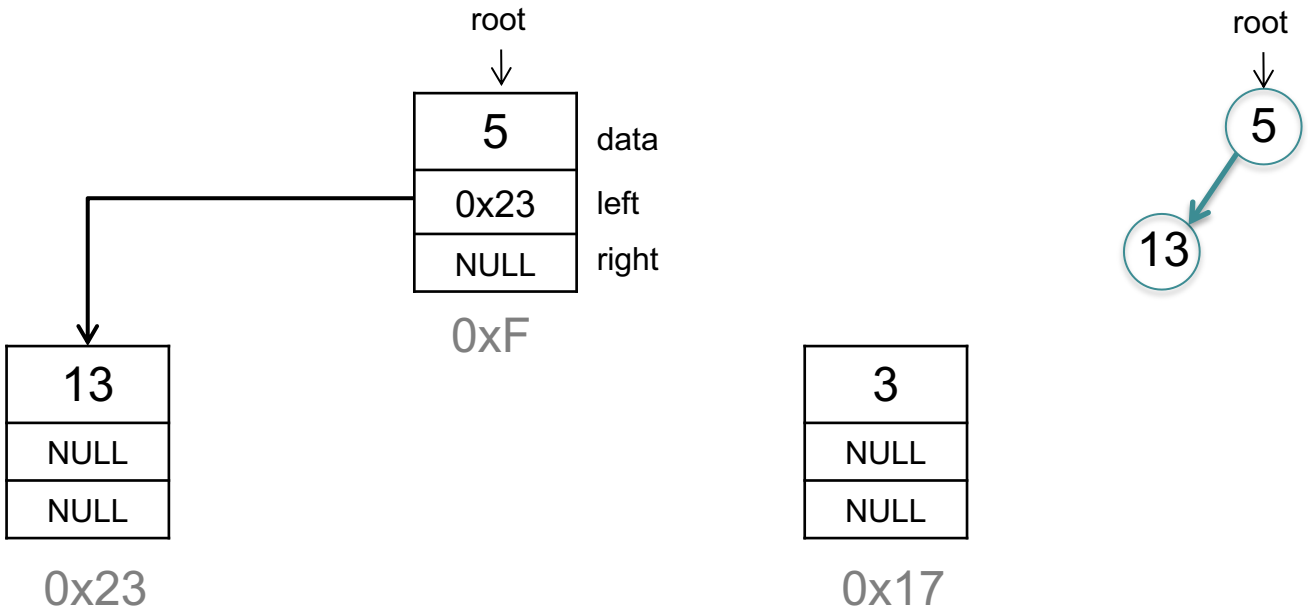




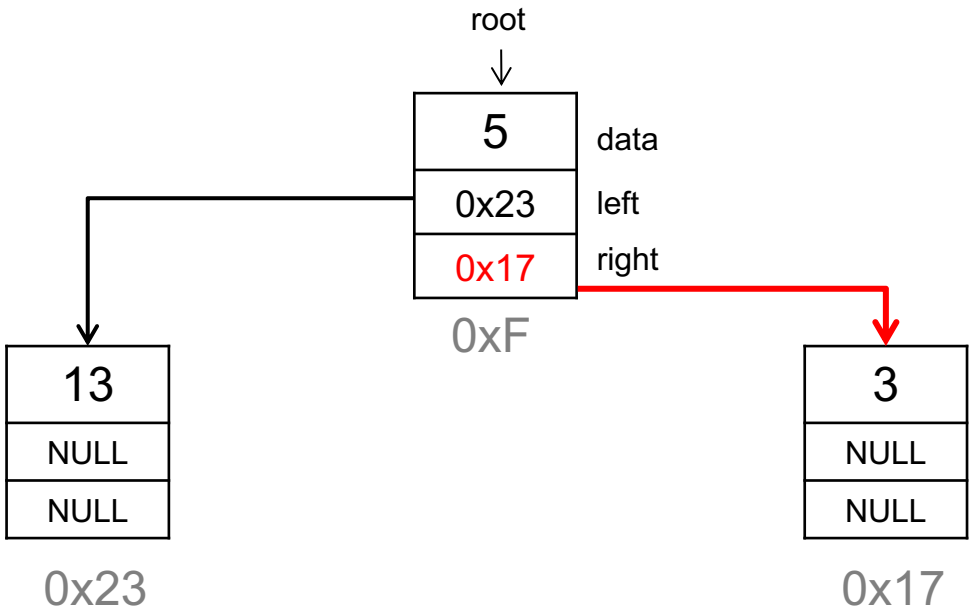
Create a new left child node !

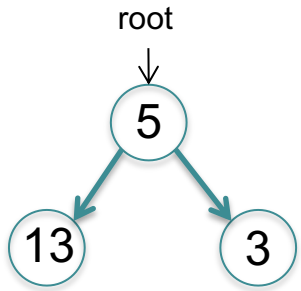

root
5
data
0x23
left
NULL
right
0xF
13
NULL
NULL
0x23


root
5
13

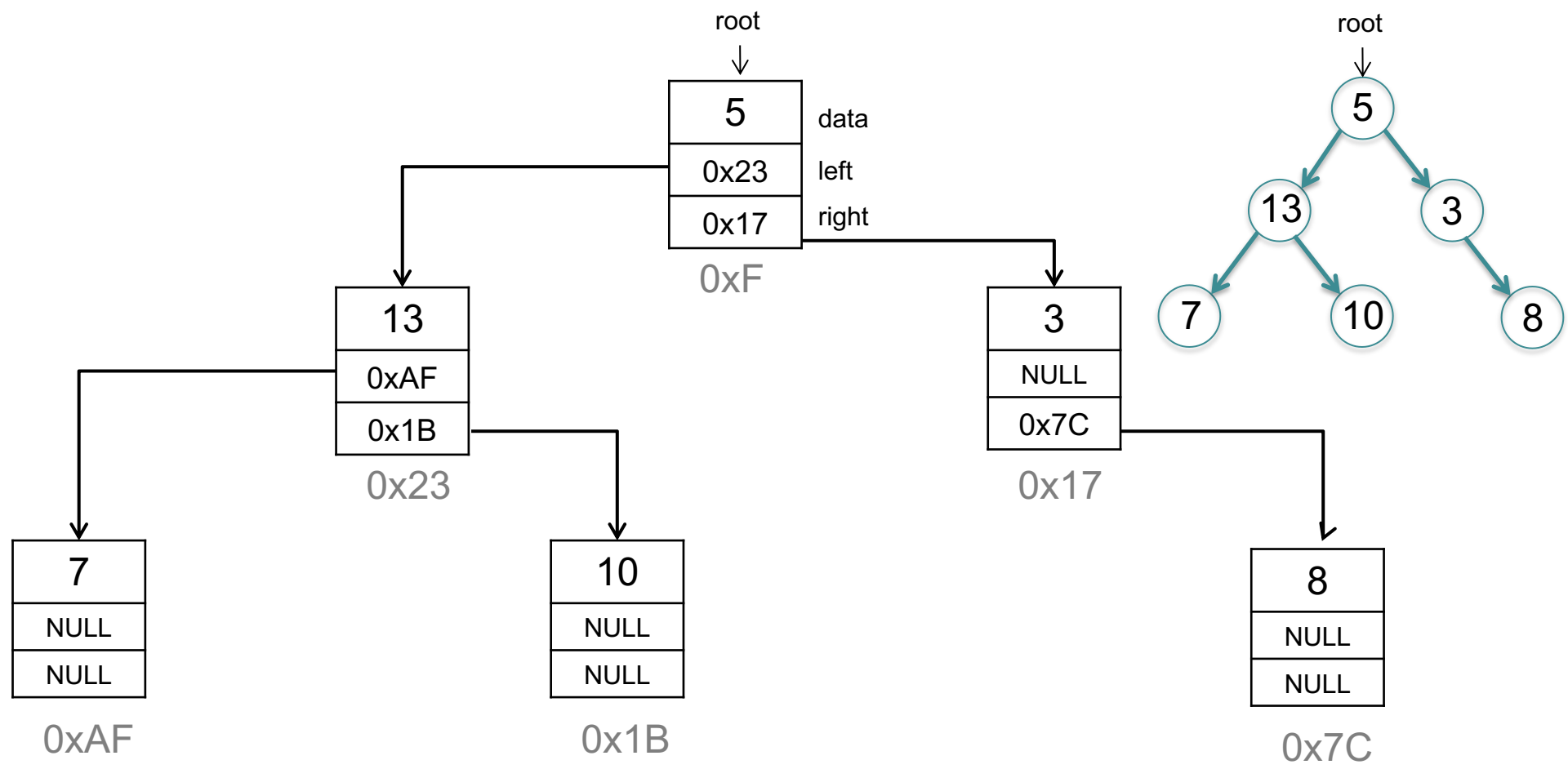


Create a new right child node !



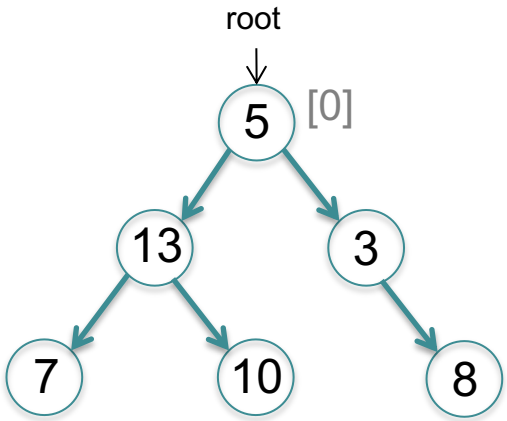


We can repeat this processes to add nodes to this tree

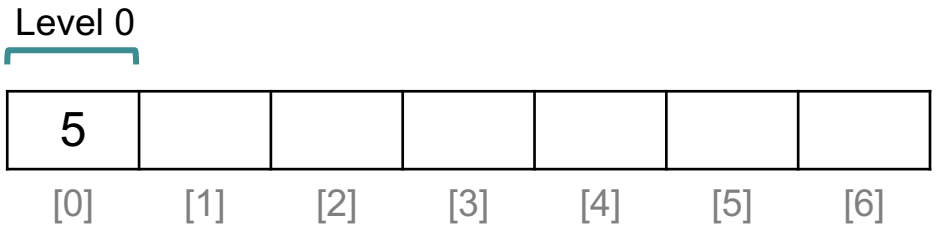


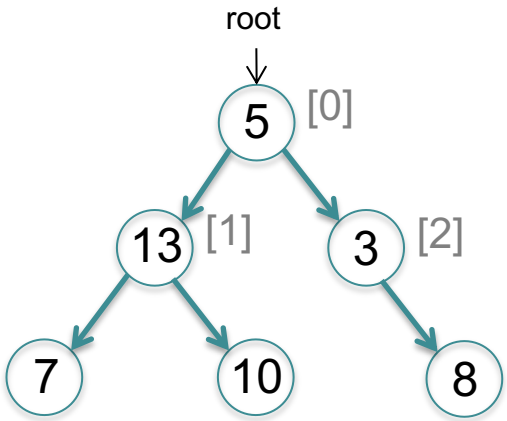
This how use pointers to implement a binary tree!

Arrays

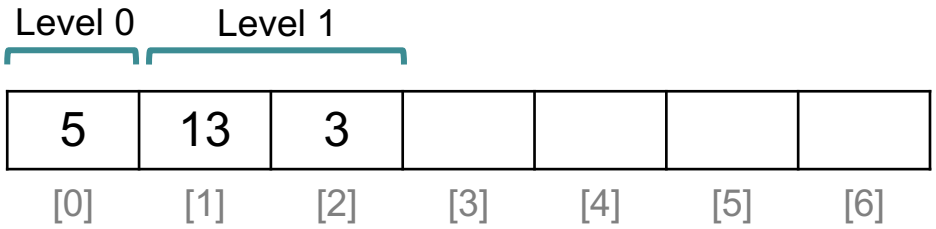


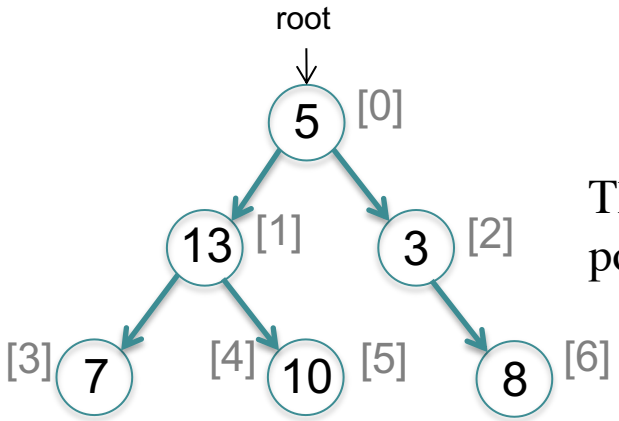
In this case:
The root is stored at position 0



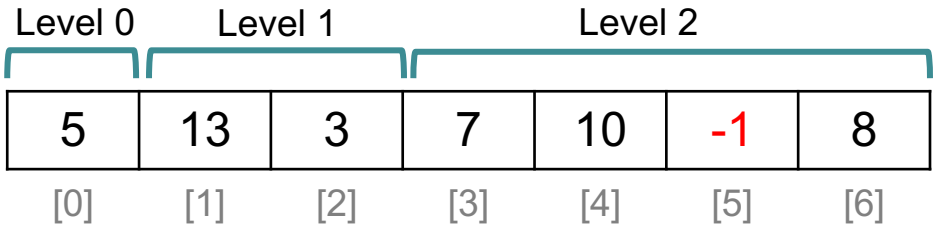


The nodes at level 1 are stored in positions 1 and 2

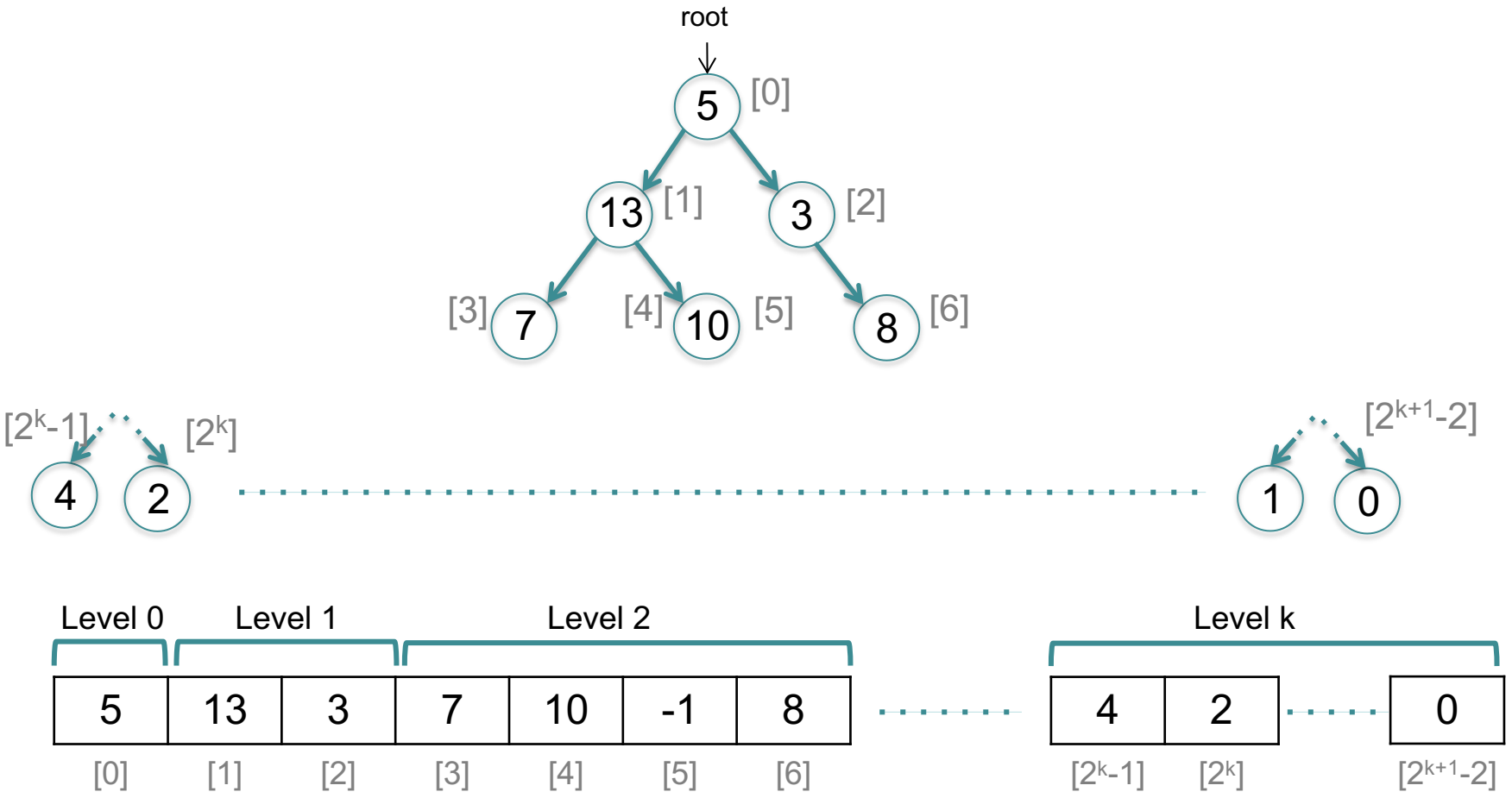




The nodes at level 2 are stored in positions 3, 4, 5 and 6



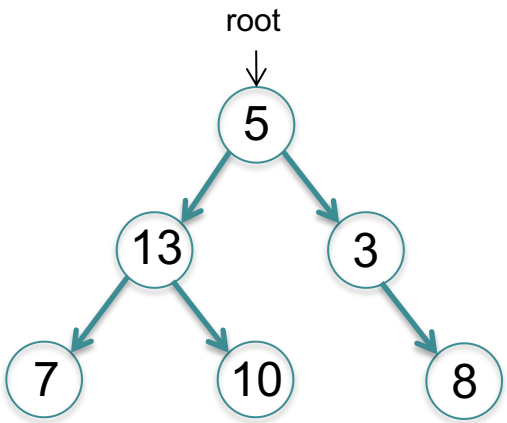
NB: no number is stored in position 5, in this case we store a number that is out of range



In general: the elements at level k are stored at positions $2^k-1, 2^k, \ldots, 2^{k+1}-2$

Arrays implementation

The advantages:
less memory space as we don’t need to use memory positions to store the addresses of the nodes
Drawbacks: Fixed size



Summary

- Memory pointers implementation
- Array implementation
- Next, we will study how to travers a tree

Trees Traversal

Intro

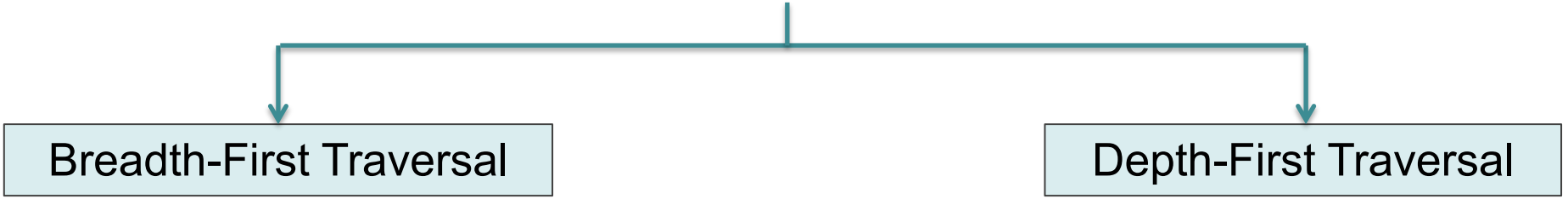
How to travers a tree ?

Traversal:
the process of visiting all the nodes of a tree

Visiting a node to access the data stored in it

When we travers a tree we need to make sure that we can visit every node of the tree.

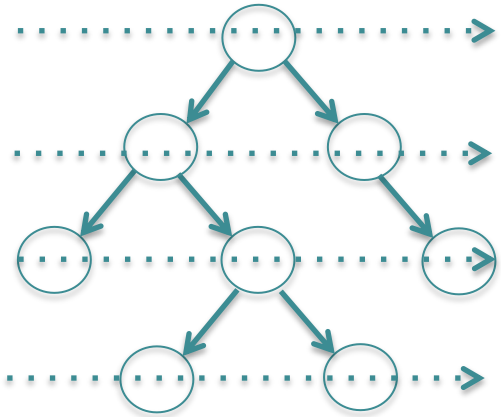
Traversal:
the process of visiting all the nodes of a tree



Traversal:
the process of visiting all the nodes of a tree

Breadth-First Traversal

Depth-First Traversal



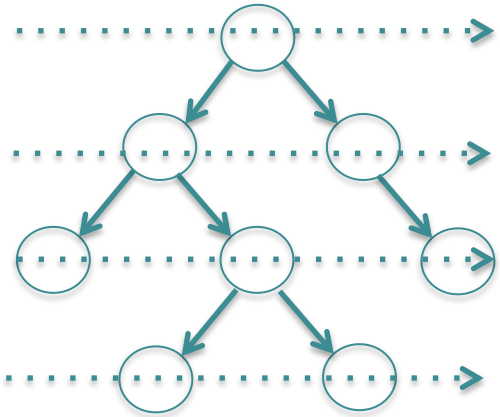
"reading the tree"
(in the western world)

We traverse a tree horizontally,
level by level from left to right

Traversal:
the process of visiting all the nodes of a tree

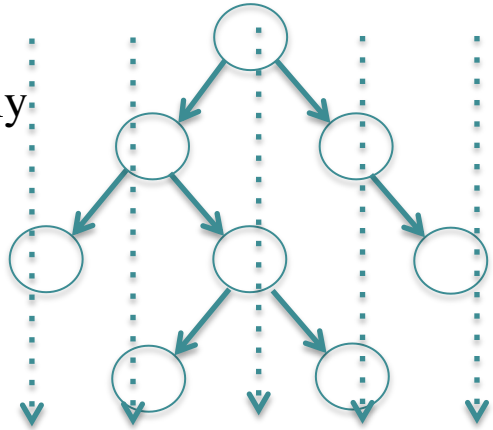
Breadth-First Traversal

Depth-First Traversal



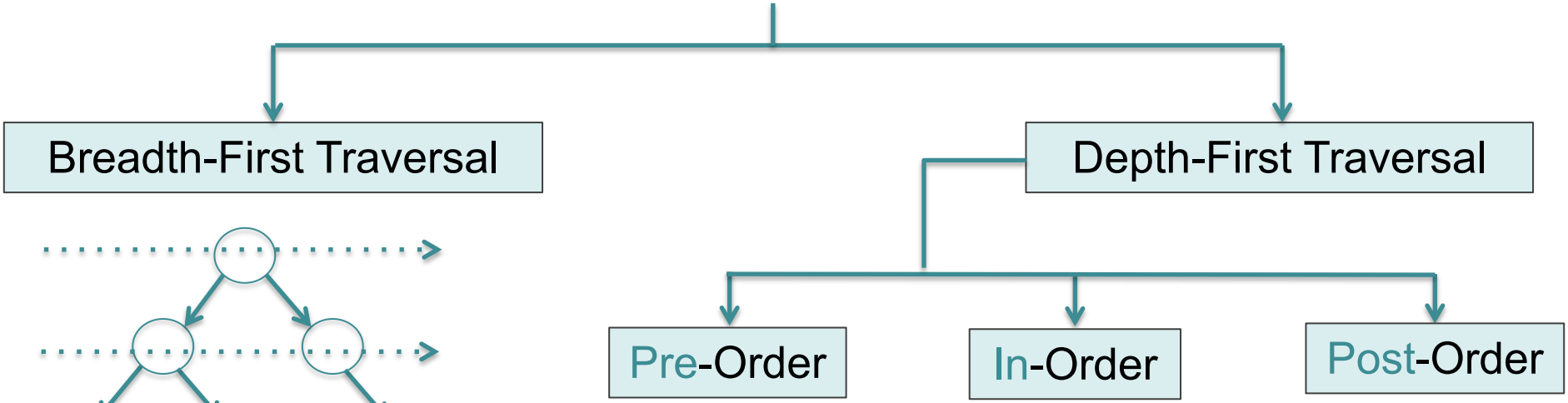
"reading the tree"
(in the western world)

Traverse the tree vertically



"diving the tree"

Traversal:
the process of visiting all the nodes of a tree

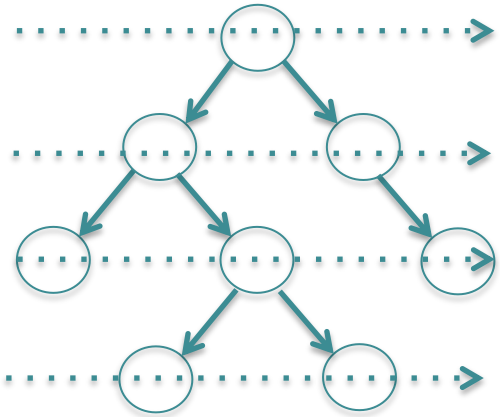


"reading the tree"
(in the western world)

3 different type of Depth-First traversal

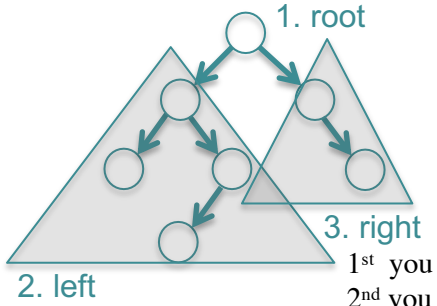
Traversal:
the process of visiting all the nodes of a tree

- Breadth-First Traversal
- Depth-First Traversal
  - Pre-Order
  - In-Order
  - Post-Order



"reading the tree"
(in the western world)

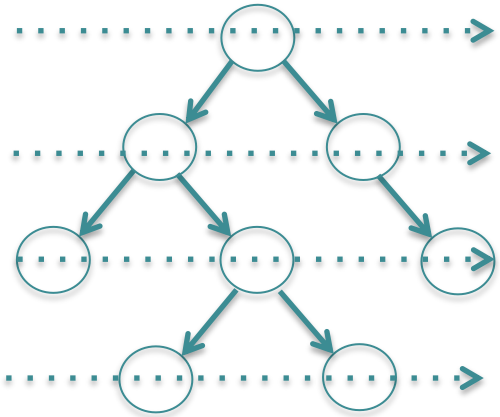
root
left
right



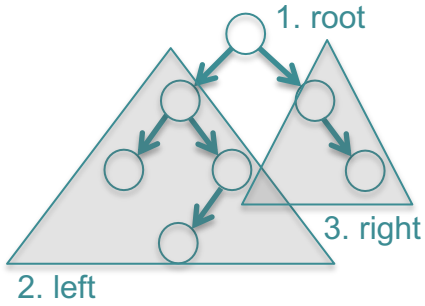
1st you visit the root node
2nd you visit the left-subtree
3rd you visit the right-subtree

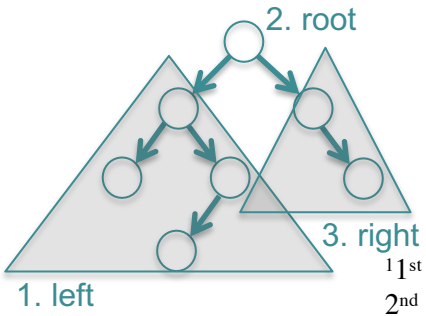
Traversal:
the process of visiting all the nodes of a tree

- Breadth-First Traversal
- Depth-First Traversal
  - Pre-Order (root, left, right)
  - In-Order (left, root, right)
  - Post-Order



"reading the tree"
(in the western world)

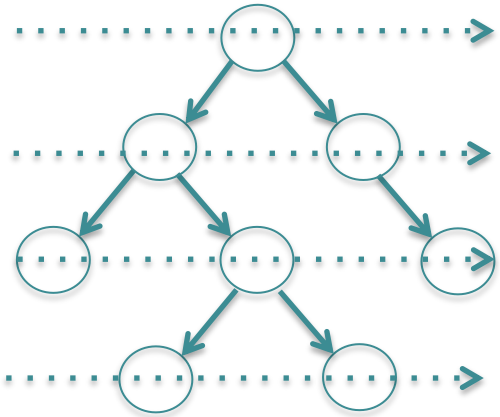




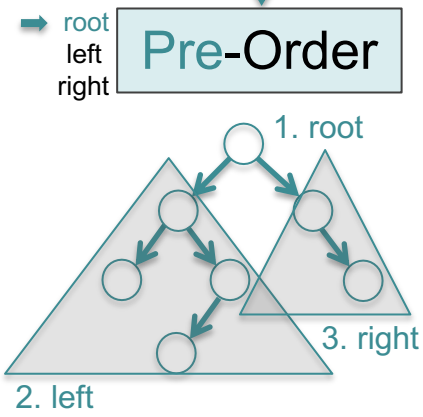
[1] 1st visit the left-subtree
2nd visit the root node
3rd visit the right-subtree

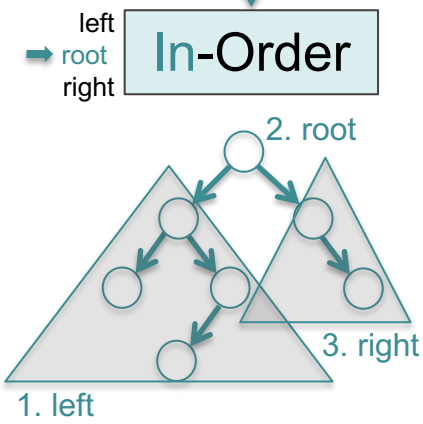
Traversal:
the process of visiting all the nodes of a tree

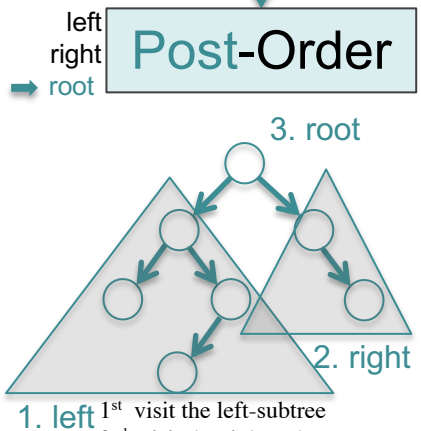
- Breadth-First Traversal
- Depth-First Traversal
  - Pre-Order
  - In-Order
  - Post-Order



"reading the tree"
(in the western world)







1st visit the left-subtree
2nd visit the right-subtree
3rd visit the root

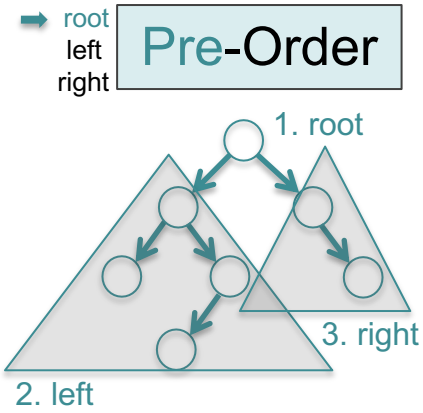
Summary:

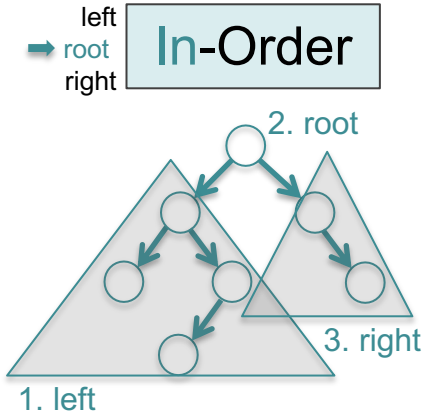
Binary tree Traversal

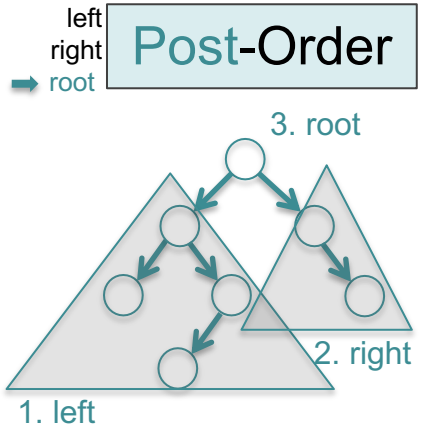
1. Breadth-First Traversal
2. Depth-First Traversal
    1. Pre-Order
    2. In-Order
    3. Post-Order

Depth-First Traversal

3-different ways of traversing a binary using depth-first traversal

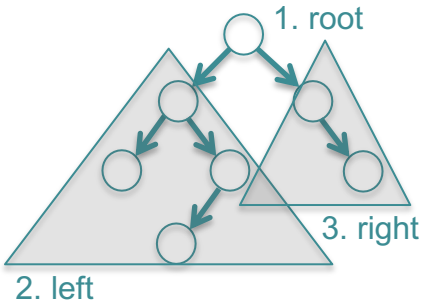

root
left
right
Pre-Order
1. root
3. right
2. left


left
root
right
In-Order
2. root
3. right
1. left


left
right
root
Post-Order
3. root
2. right
1. left

root
left
right

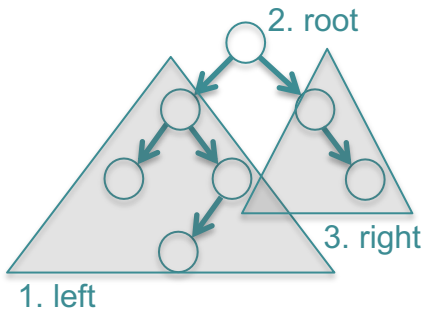
Pre-Order




```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```

left
root
right

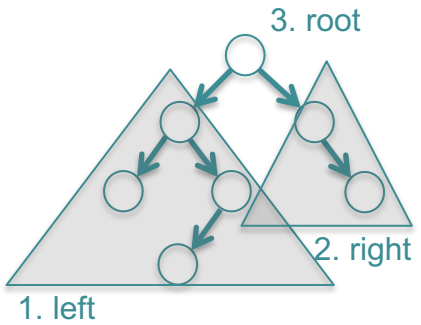
In-Order




```
function in-order(T)
  if !ISEMPTY(T) then
    in-order(left(T))
    visit(root(T))
    in-order(right(T))
  end if
end function
```

left
right
root

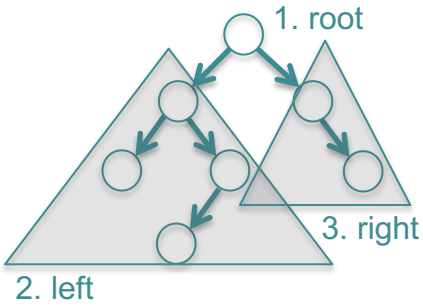
Post-Order




```
function post-order(T)
  if !ISEMPTY(T) then
    post-order(left(T))
    post-order(right(T))
    visit(root(T))
  end if
end function
```

Pre-Order

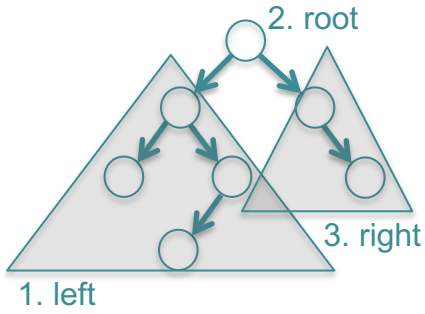
root
left
right




```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```

In-Order

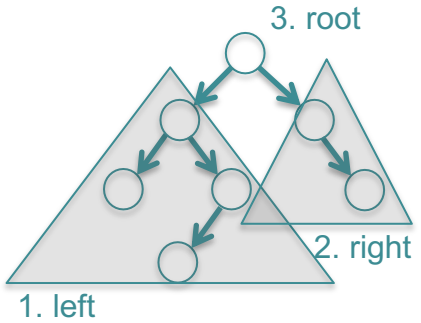
left
root
right




```
function in-order(T)
  if !ISEMPTY(T) then
    in-order(left(T))
    visit(root(T))
    in-order(right(T))
  end if
end function
```

Post-Order

left
right
root

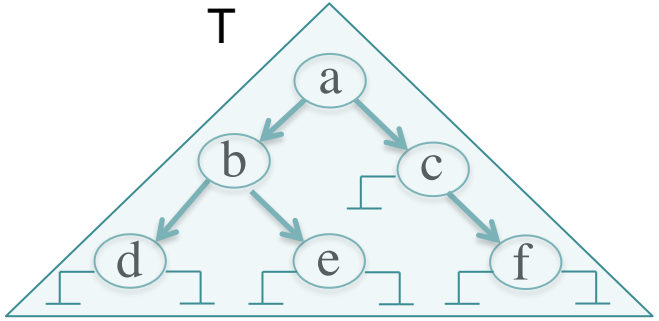



```
function post-order(T)
  if !ISEMPTY(T) then
    post-order(left(T))
    post-order(right(T))
    visit(root(T))
  end if
end function
```

➡ root
left
right

Pre-Order

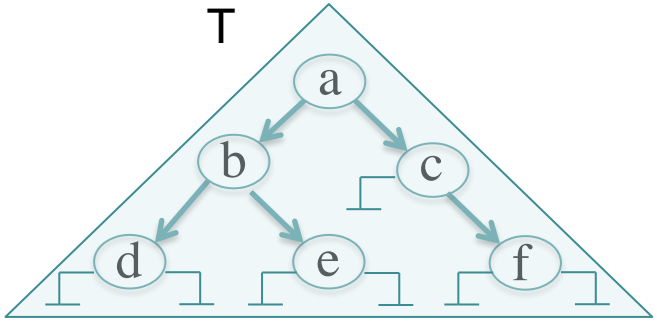
```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```



➡ root
left
right

Pre-Order

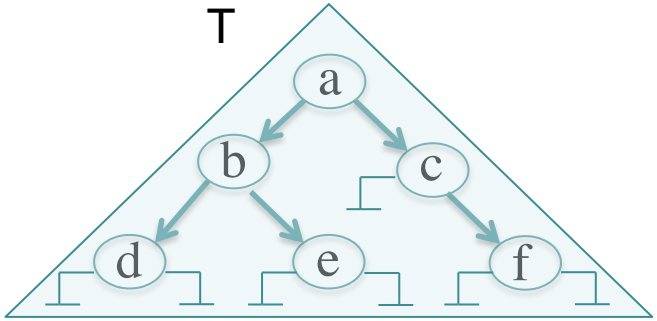
```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```



➡ root
left
right

Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```

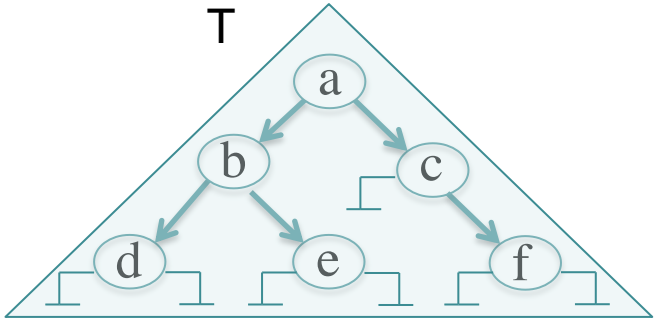


[a]

➡ root
left
right

Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```

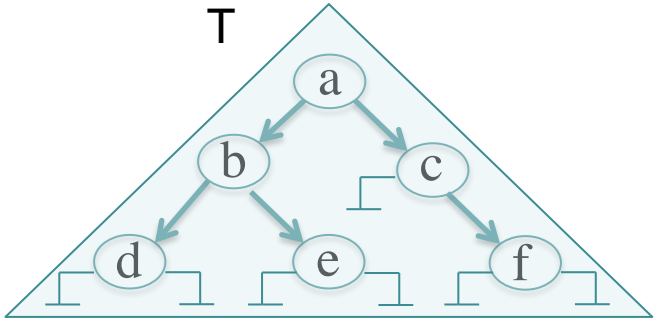


[a]

root
left
right

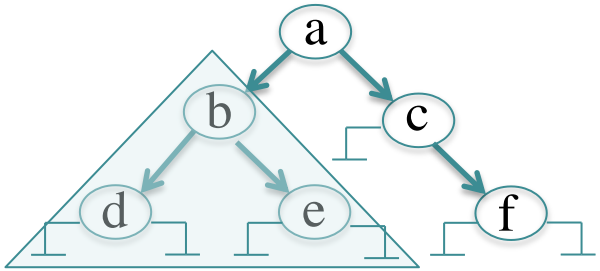
Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```



[a]

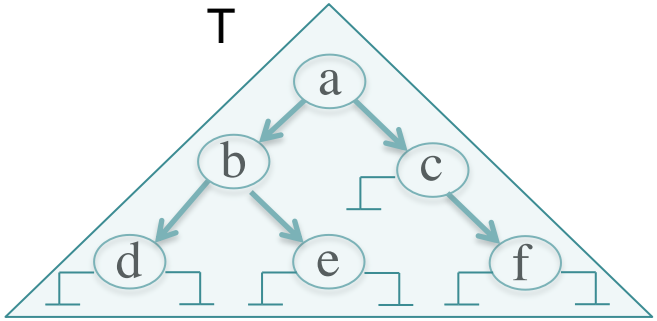
T



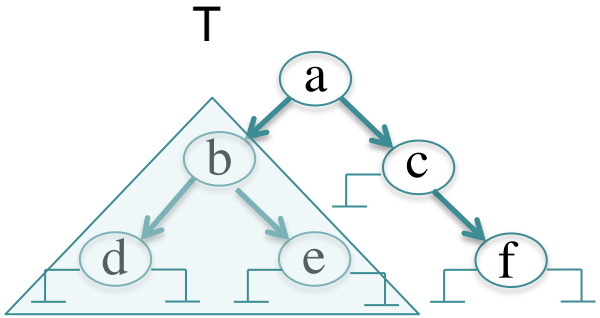
root
left
right

Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```



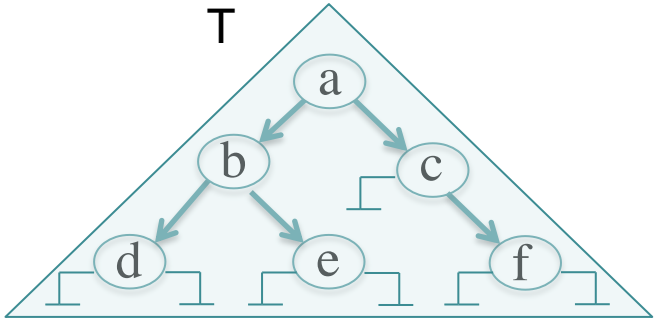
[a b]



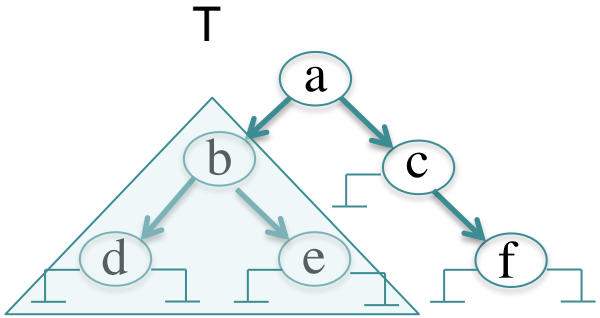
root
left
right

Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```



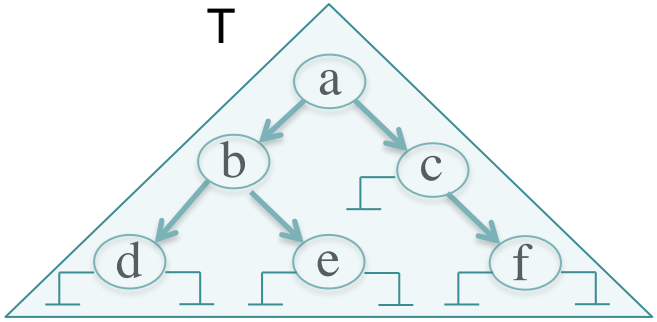
[a b]



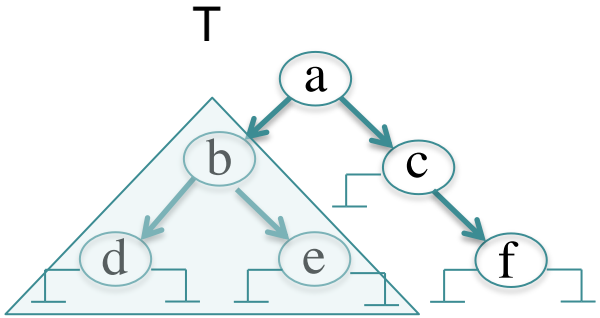
root
left
right

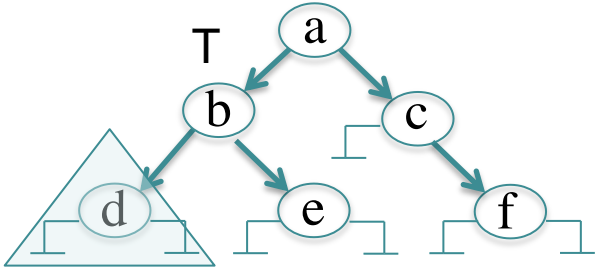
Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```



[a b]

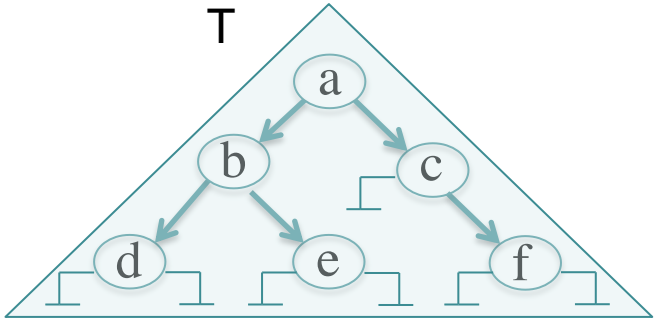




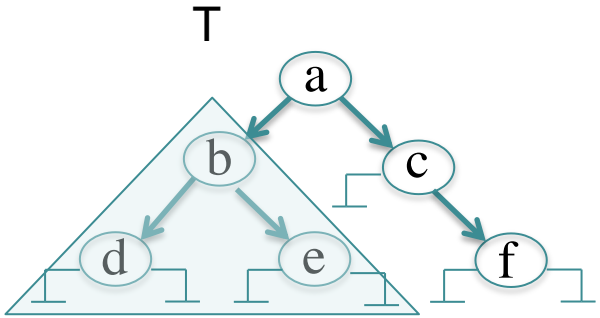
root
left
right

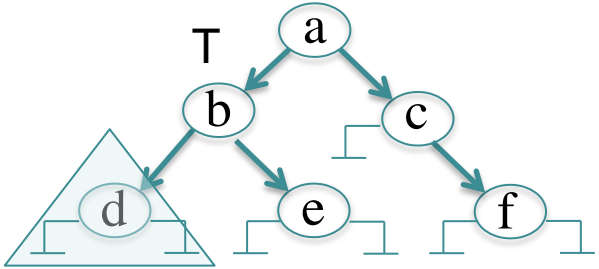
Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```



[a b d]

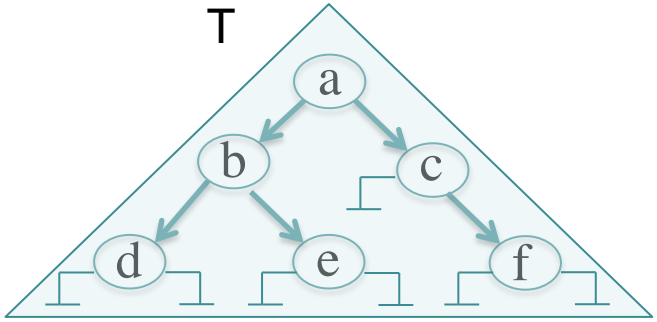




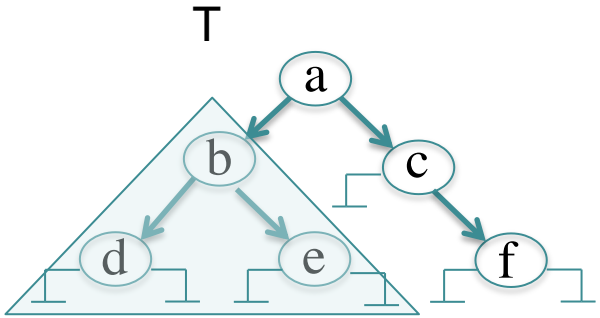
root
left
right

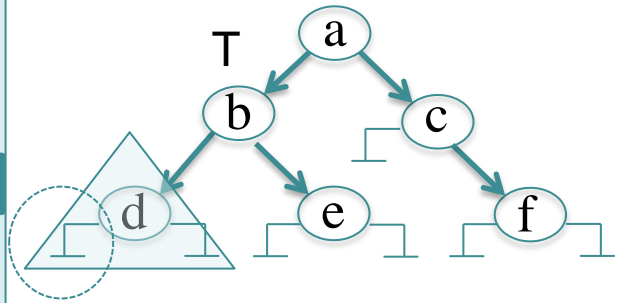
Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```



[a b d]

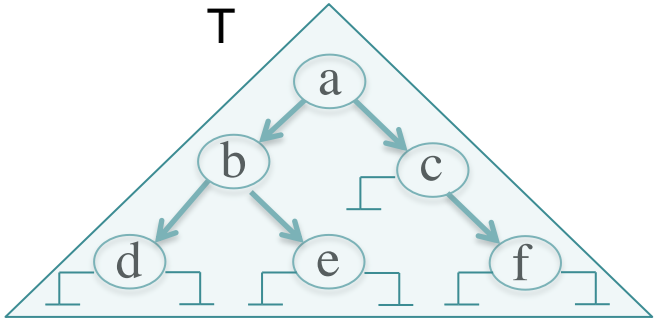




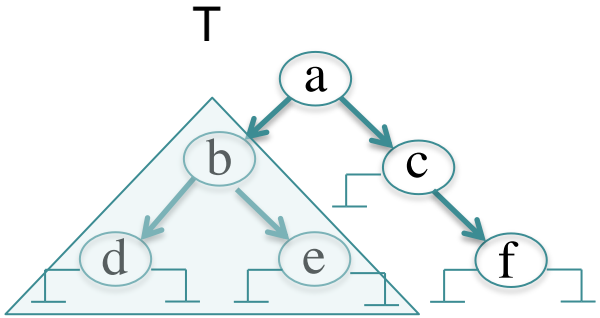
root
left
right

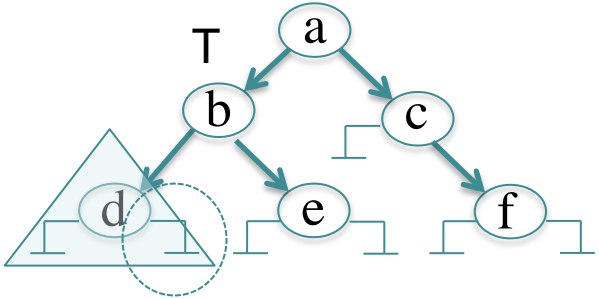
Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```



[a b d]

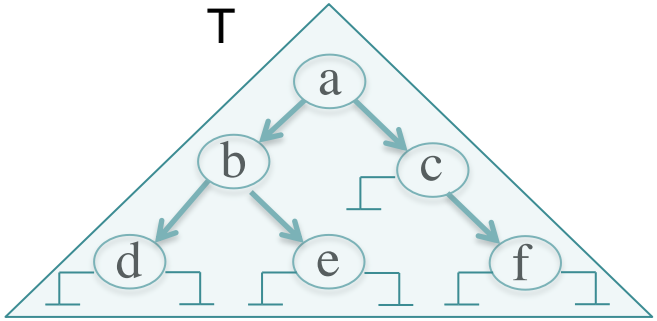




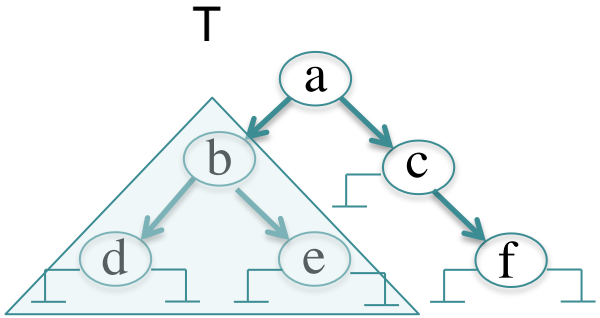
root
left
right

Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```



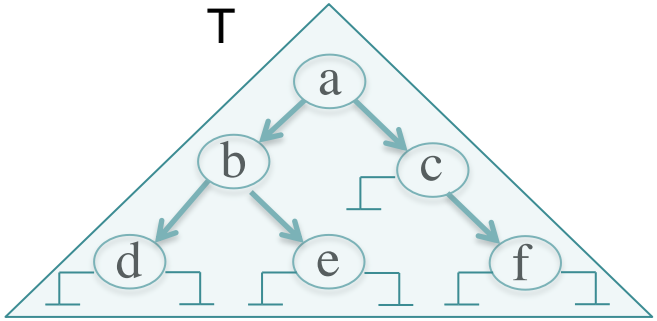
[a b d]



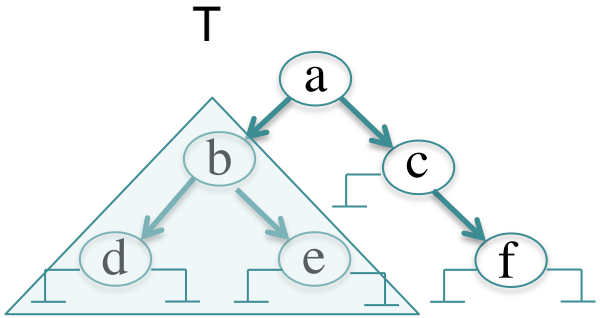
root
left
right

Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```



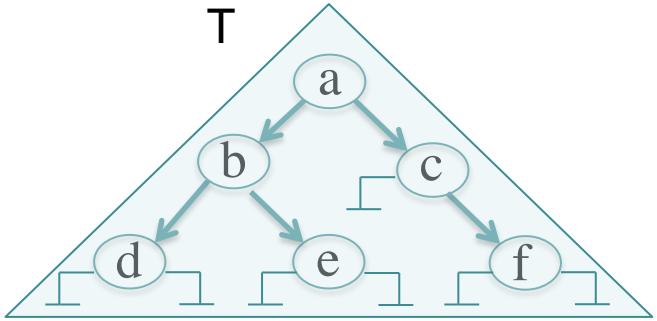
[a b d]



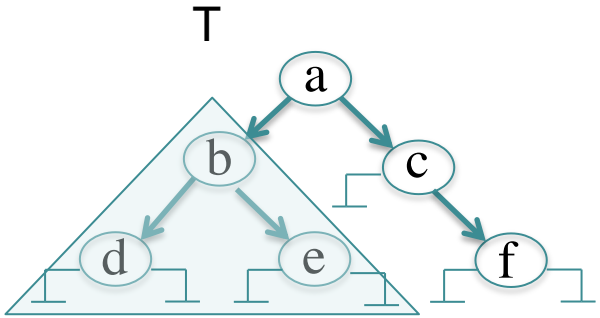
root
left
right

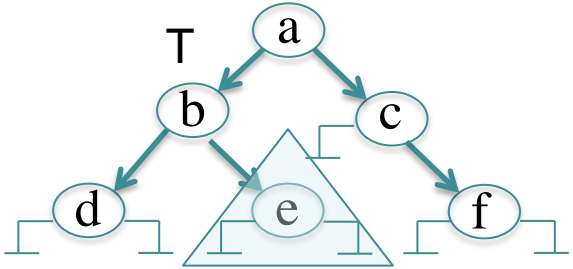
Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```



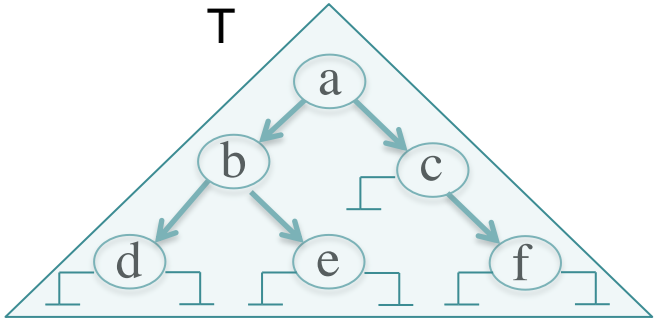
[a b d]





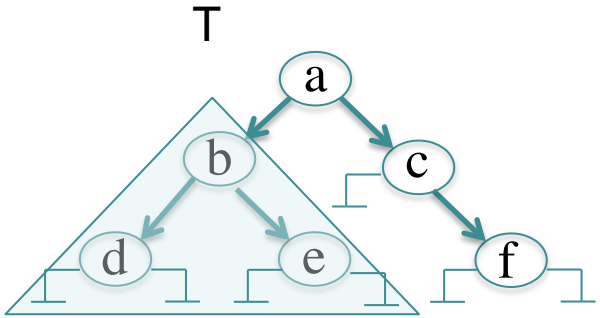
root
left
right

Pre-Order

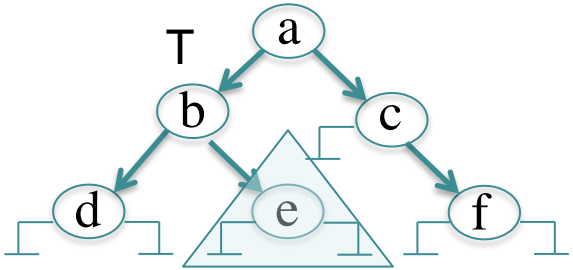


[a b d e]

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```




```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```

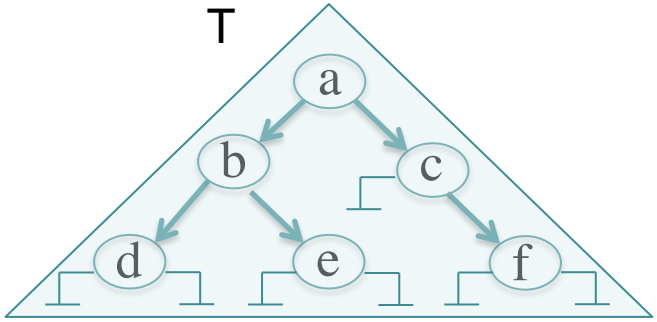



```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```

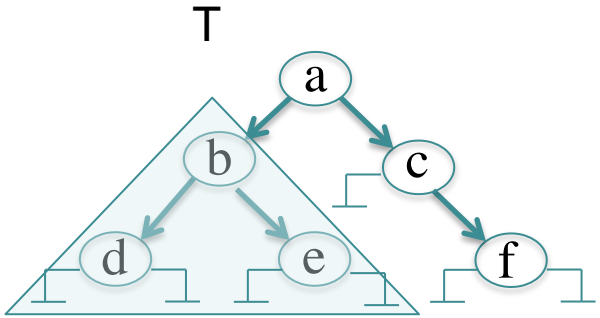
root
left
right

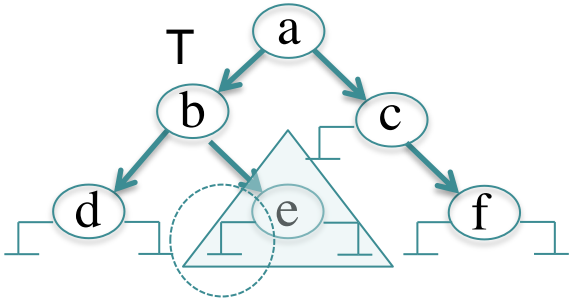
Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```



[a b d e]

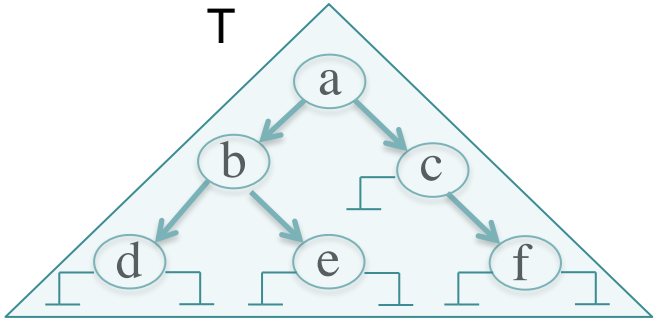




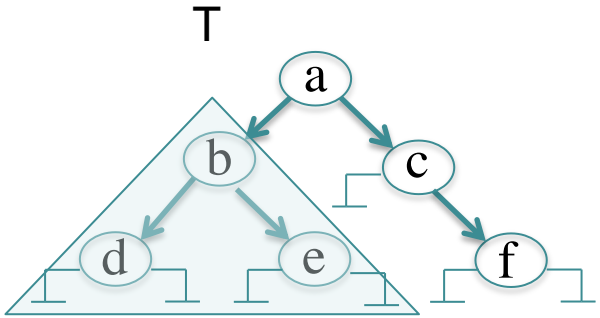
root
left
right

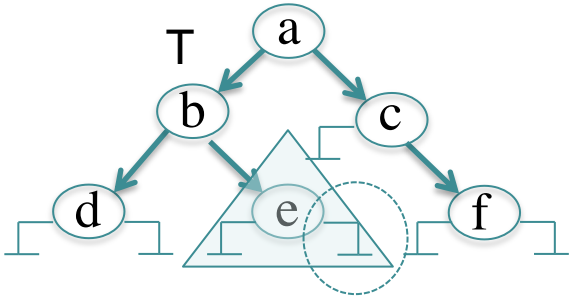
Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```



[a b d e]

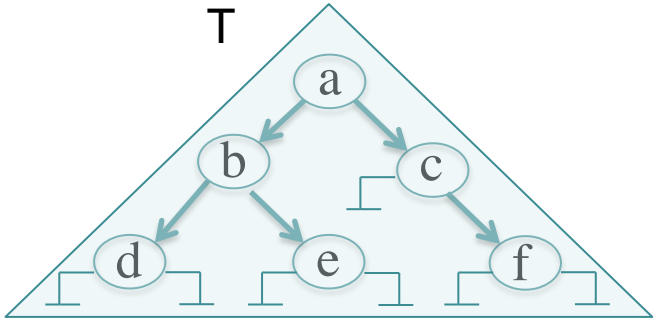




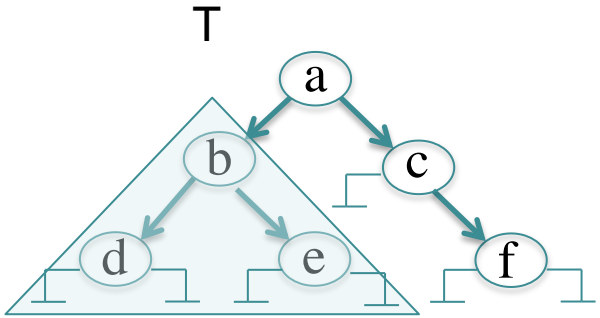
root
left
right

Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```



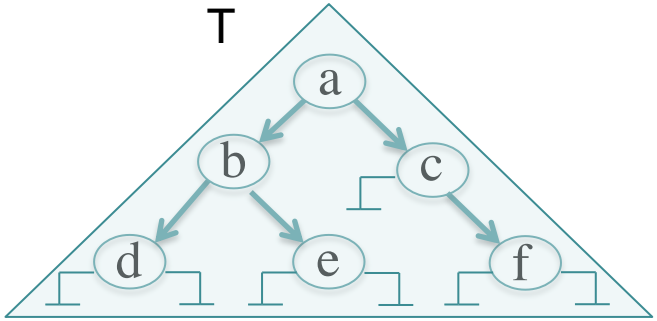
[a b d e]



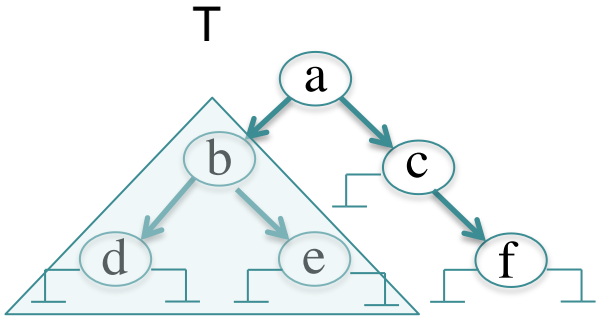
root
left
right

Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```



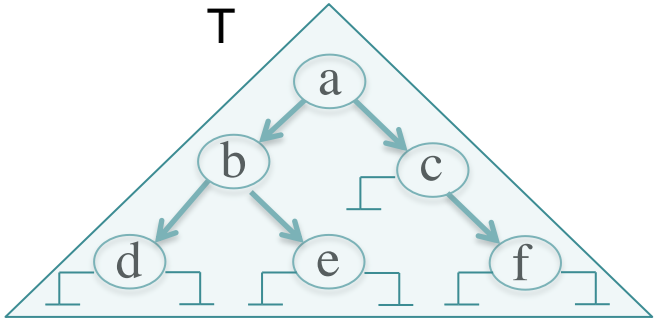
[a b d e]



➡ root
left
right

Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```

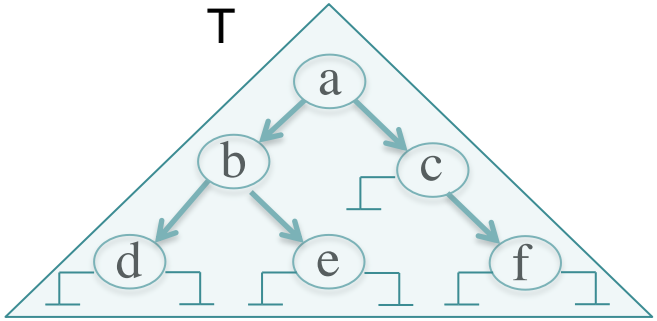


[a b d e]

➡ root
left
right

Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```

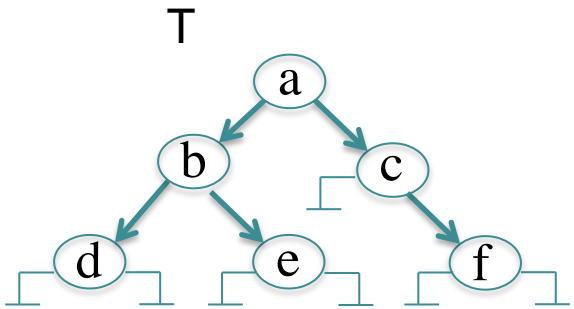


[a b d e]

➡ root
left
right

Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```

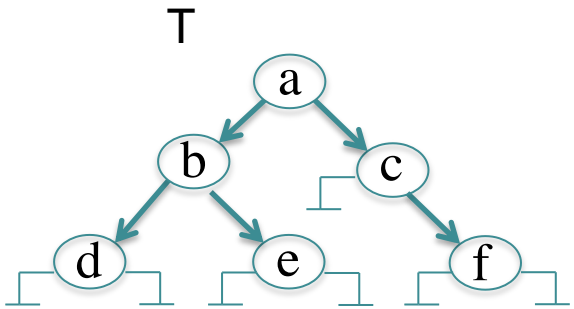


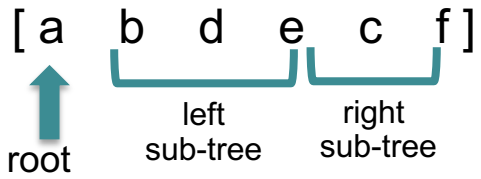
[a b d e c f]

➡ root
left
right

Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```

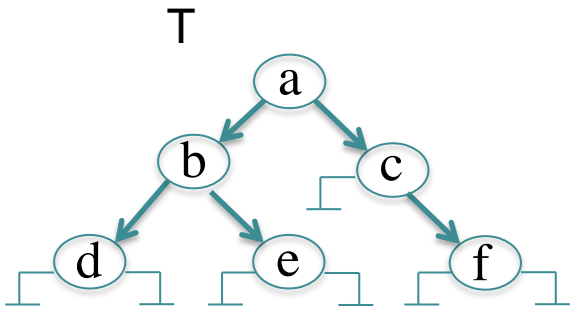


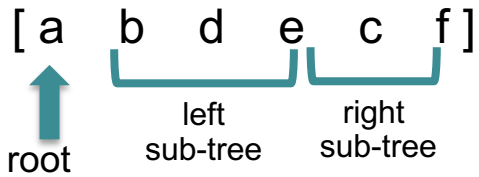


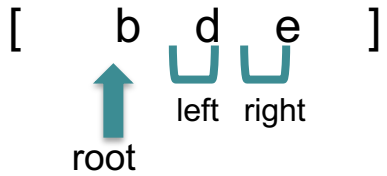
➡ root
left
right

Pre-Order

```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
  end if
end function
```

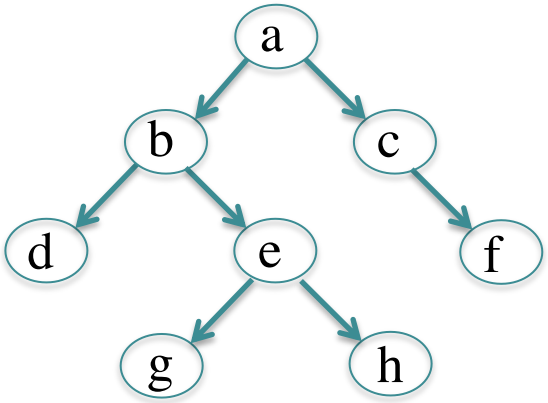






root
left
right

Pre-Order




```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
end function
```

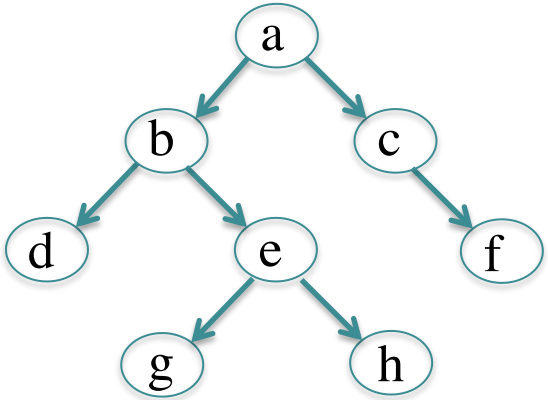
Left sub-tree Right sub-tree

[a ,b,d,e,g,h,c,f]

root precedes everything

left
root
right

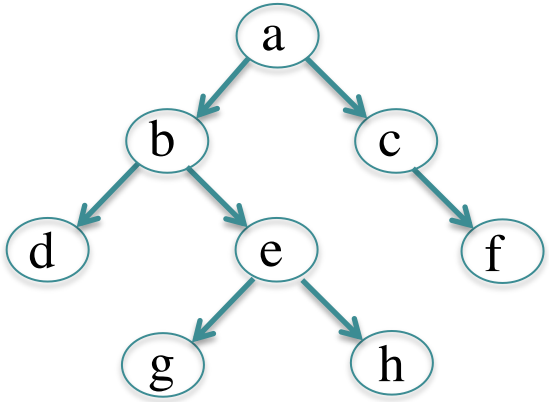
In-Order




```
function in-order(T)
  if !ISEMPTY(T) then
    in-order(left(T))
    visit(root(T))
    in-order(right(T))
end function
```

left
right
root

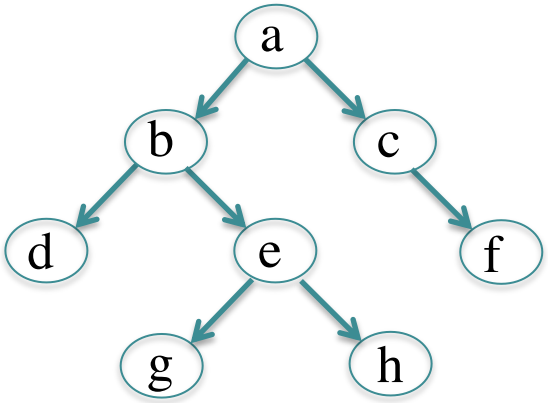
Post-Order




```
function post-order(T)
  if !ISEMPTY(T) then
    post-order(left(T))
    post-order(right(T))
    visit(root(T))
end function
```

root
left
right

Pre-Order




```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
end function
```

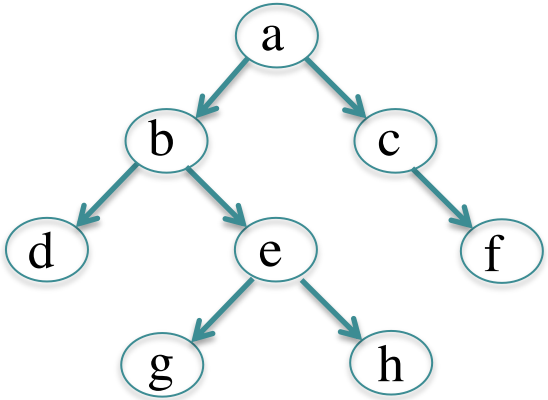
Left sub-tree | Right sub-tree

[a ,b,d,e,g,h,c,f]

root precedes everything

left
root
right

In-Order




```
function in-order(T)
  if !ISEMPTY(T) then
    in-order(left(T))
    visit(root(T))
    in-order(right(T))
end function
```

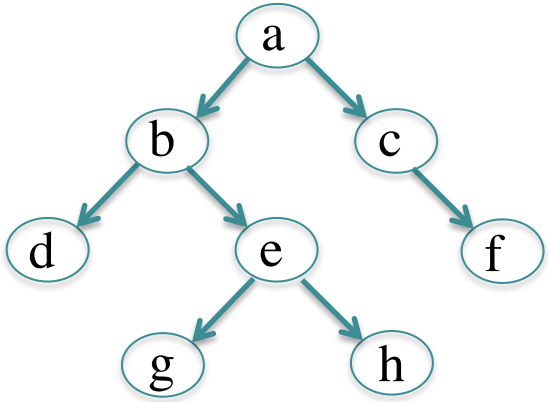
Left sub-tree | Right sub-tree

[d,b,g,e,h, a ,c,f]

root in the middle

left
right
root

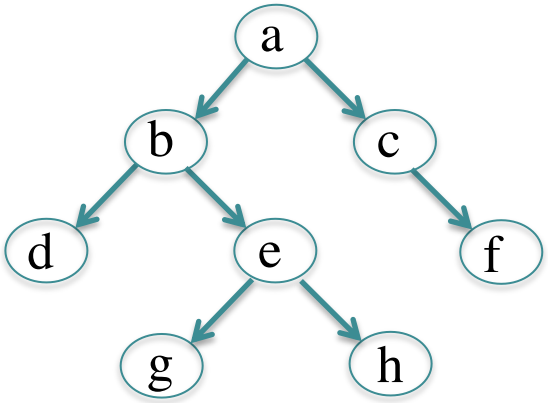
Post-Order




```
function post-order(T)
  if !ISEMPTY(T) then
    post-order(left(T))
    post-order(right(T))
    visit(root(T))
end function
```

root
left
right

Pre-Order




```
function pre-order(T)
  if !ISEMPTY(T) then
    visit(root(T))
    pre-order(left(T))
    pre-order(right(T))
end function
```

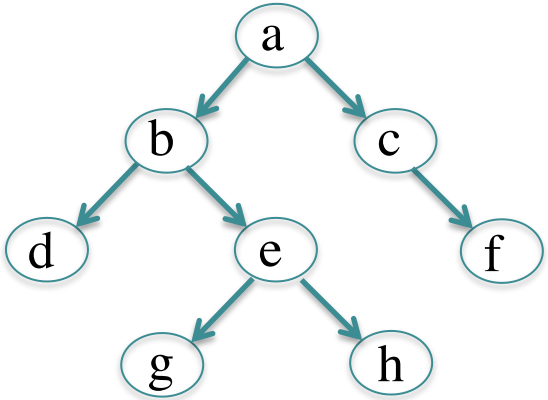
Left sub-tree Right sub-tree

[a ,b,d,e,g,h,c,f]

root precedes everything

left
root
right

In-Order




```
function in-order(T)
  if !ISEMPTY(T) then
    in-order(left(T))
    visit(root(T))
    in-order(right(T))
end function
```

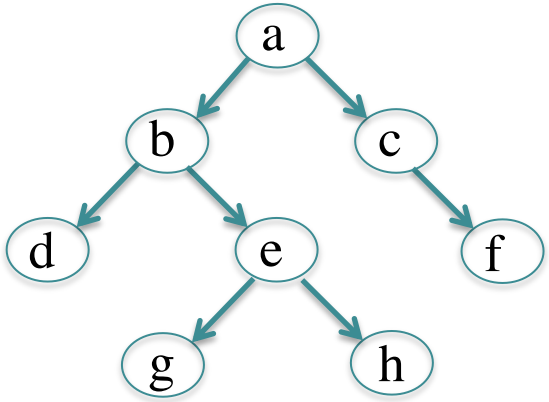
Left sub-tree Right sub-tree

[d,b,g,e,h, a ,c,f]

root in the middle

left
right
root

Post-Order




```
function post-order(T)
  if !ISEMPTY(T) then
    post-order(left(T))
    post-order(right(T))
    visit(root(T))
end function
```

Left sub-tree Right sub-tree

[d,g,h,e,b,f,c, a]

root after everything

Summary

Root precedes everything

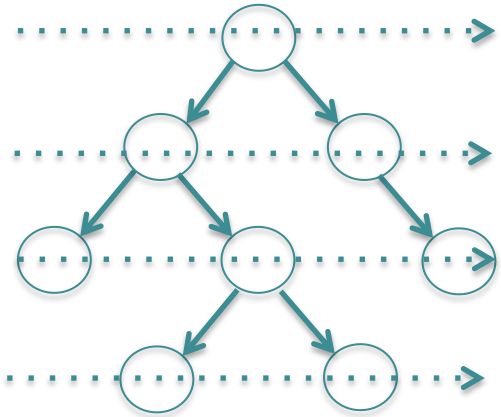
Root in the middle

Root after everything

Breadth-First Traversal

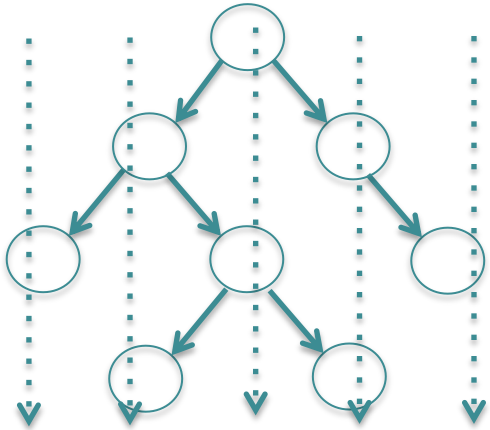
Traversal:
the process of visiting all the nodes of a tree

Breadth-First Traversal



"reading the tree"
(in the western world)

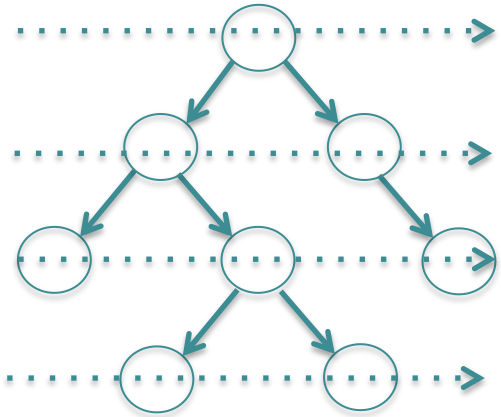
Depth-First Traversal



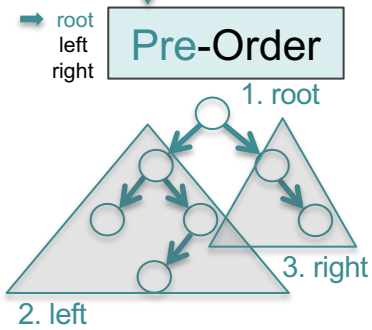
"diving the tree"

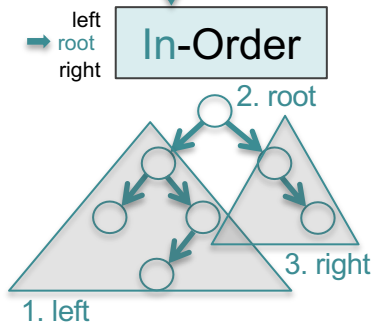
Traversal:
the process of visiting all the nodes of a tree

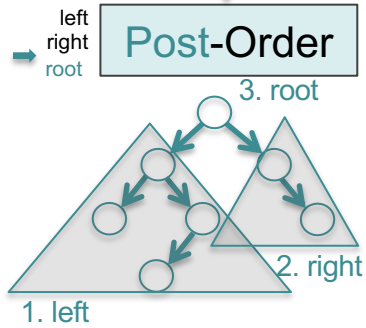
- Breadth-First Traversal
- Depth-First Traversal
  - Pre-Order (root, left, right)
  - In-Order (left, root, right)
  - Post-Order (left, right, root)



“reading the tree”
(in the western world)



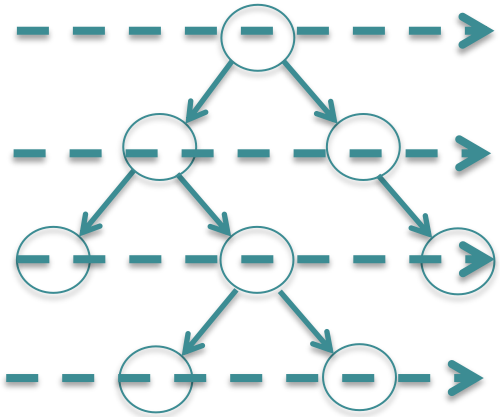




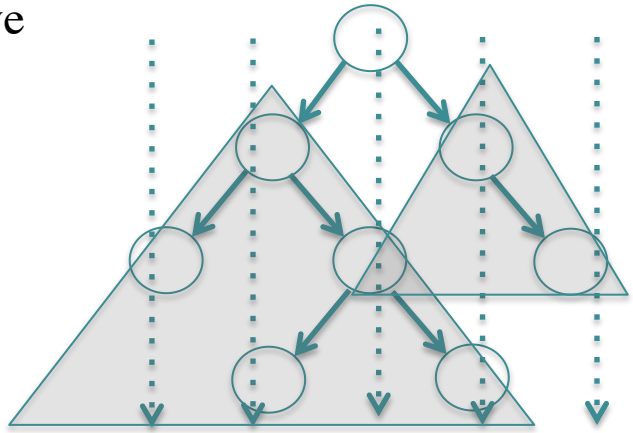
Traversal:
the process of visiting all the nodes of a tree

Breadth-First Traversal

Depth-First Traversal



There is no horizontal recursive structure!
Use iterative algorithm



"reading the tree"
(in the western world)

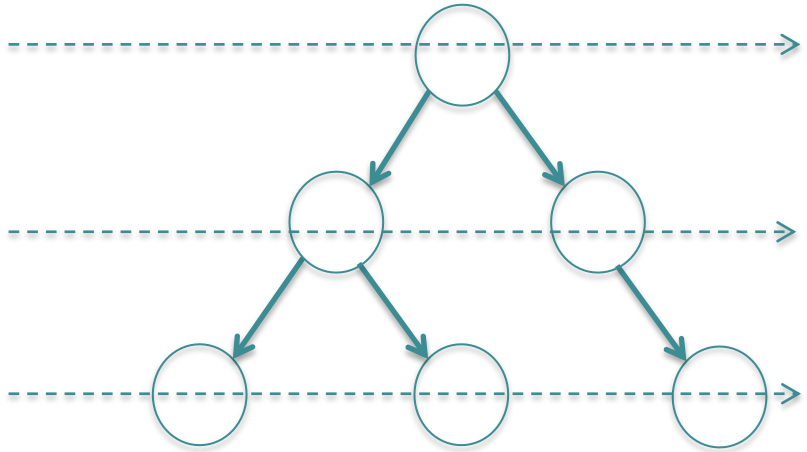
"diving the tree"

Breadth-First Traversal

visit(T.root)

visit(T.root.left) ; visit(T.root.right)

visit(T.root.left.left); visit(T.root.left.right);
visit(T.root.right.left); visit(T.root.right.right);



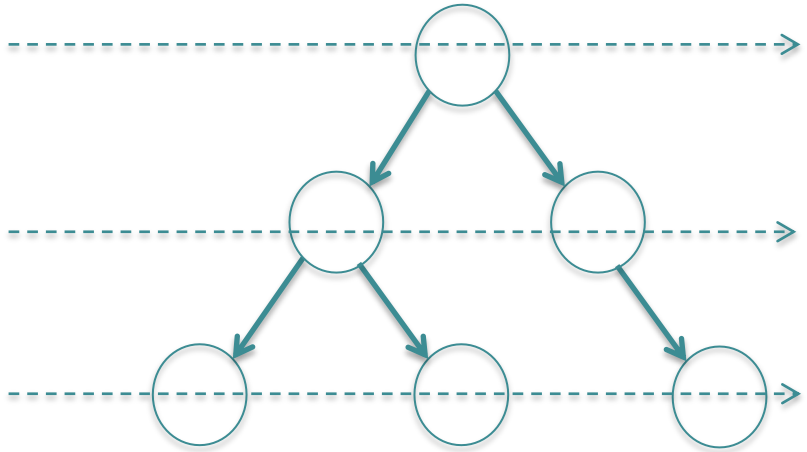
The program that will visit this tree will look like this!

Breadth-First Traversal

visit(T.root)

visit(T.root.left) ; visit(T.root.right)

visit(T.root.left.left); visit(T.root.left.right);
visit(T.root.right.left); visit(T.root.right.right);

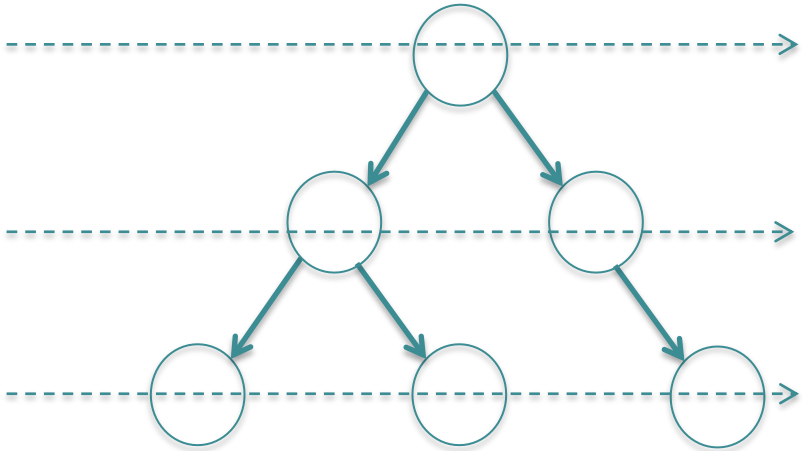


Breadth-First Traversal

visit(T.root)

visit(T.root.left) ; visit(T.root.right)

visit(T.root.left.left); visit(T.root.left.right);
visit(T.root.right.left); visit(T.root.right.right);

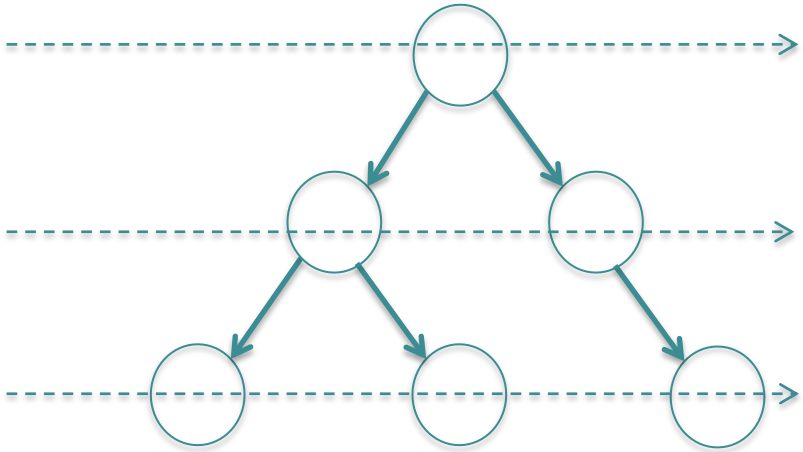


Breadth-First Traversal

visit(T.root)

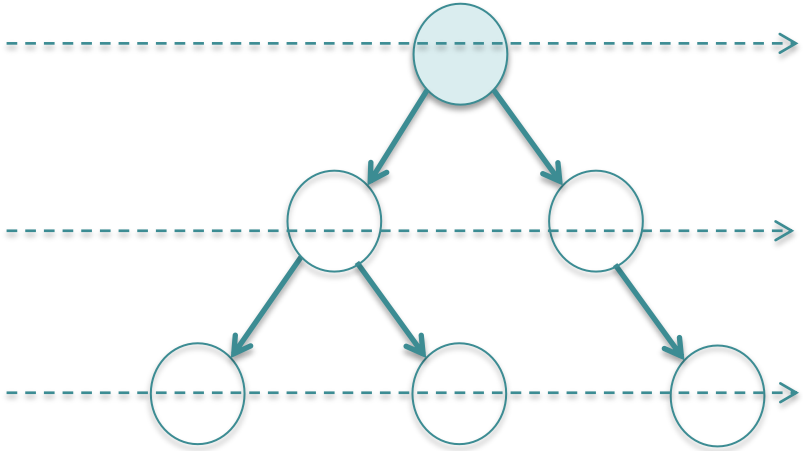
visit(T.root.left) ; visit(T.root.right)

visit(T.root.left.left); visit(T.root.left.right);
visit(T.root.right.left); visit(T.root.right.right);



Breadth-First Traversal

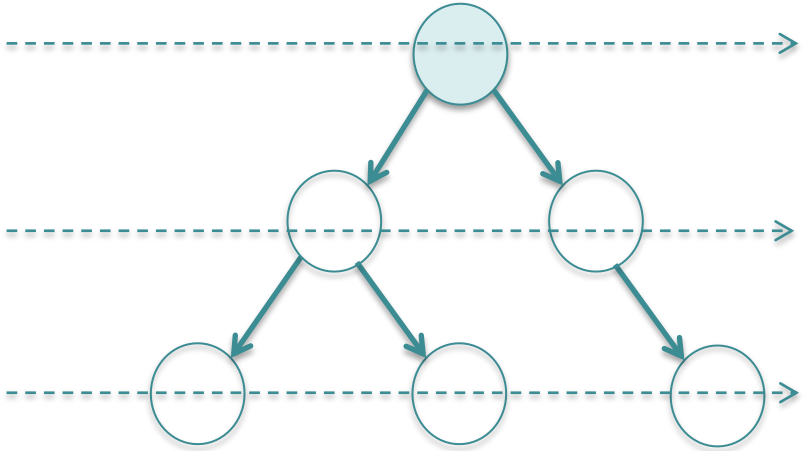
visit(T.root)



Once the root is visited, we know that the next nodes to be visited its children!!

Breadth-First Traversal

visit(T.root)

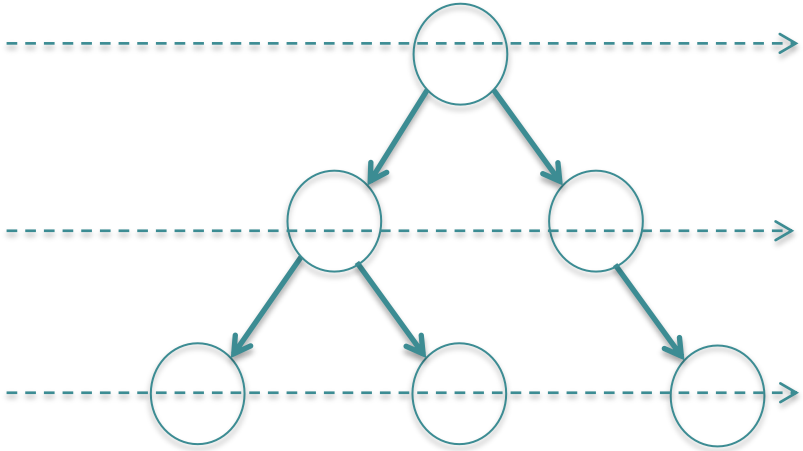


Temporary storage
[T.root.left, T.root.right]

Save the information about its children immediately

Breadth-First Traversal

visit(T.root)



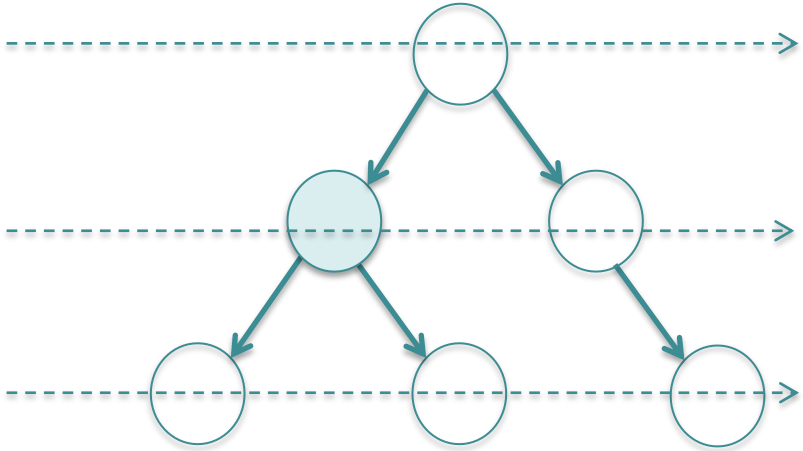
Temporary storage
[T.root.left, T.root.right]

Visiting nodes in level 1 is is easy, we can just extract this information from the temporary storage !

Breadth-First Traversal

visit(T.root)

visit(T.root.left)



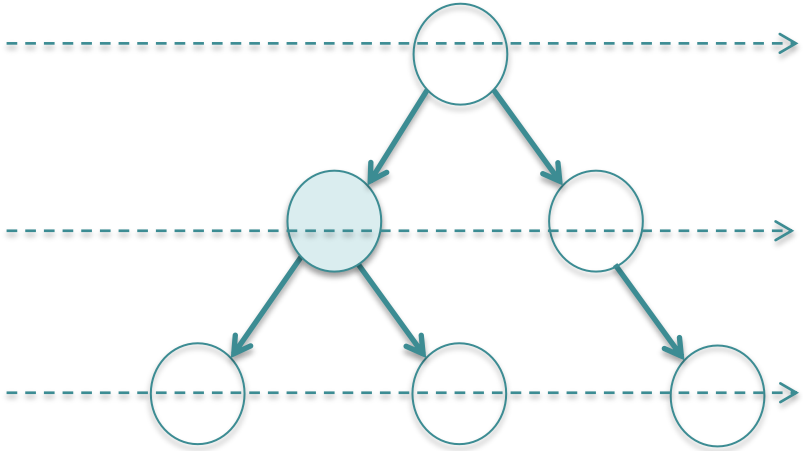
Temporary storage
[T.root.left, T.root.right]

Visiting the the first node in level 1 is easy, we just extract this information from the temporary storage!

Breadth-First Traversal

visit(T.root)

visit(T.root.left)



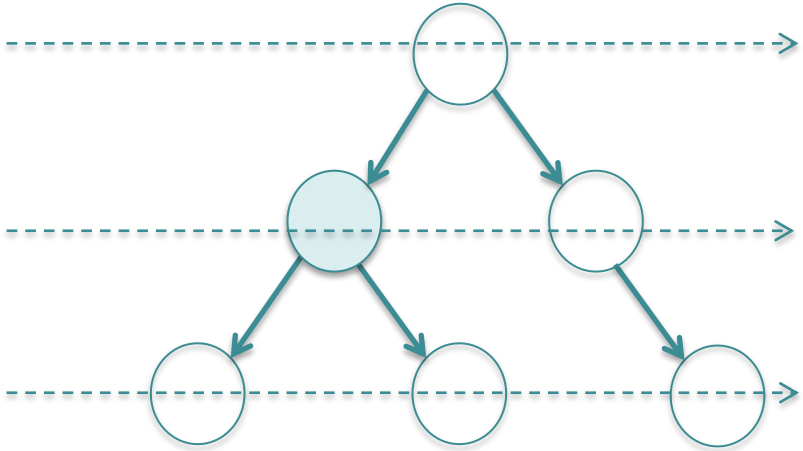
Temporary storage
[T.root.left, T.root.right, T.root.left.left, T.root.left.right]

Add the information about the children in the temporary storage!

Breadth-First Traversal

visit(T.root)

visit(T.root.left)



Temporary storage
[T.root.left, T.root.right, T.root.left.left, T.root.left.right]

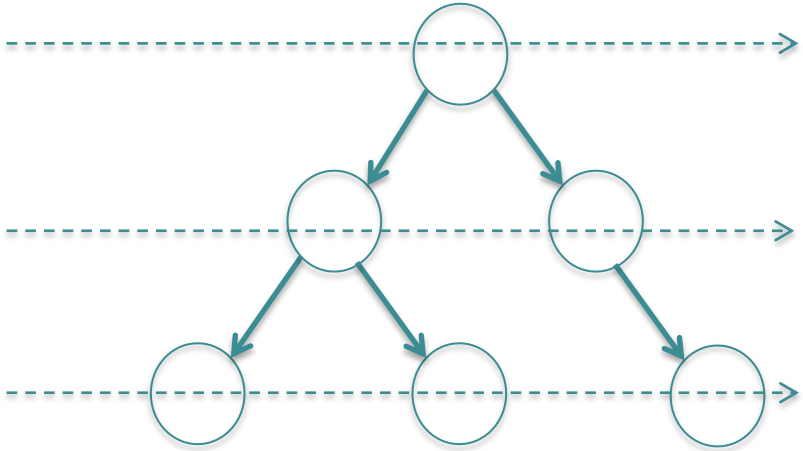
This node is now visited and information about its children is saved in the memory storage!

Hence, it can be removed

Breadth-First Traversal

visit(T.root)

visit(T.root.left)

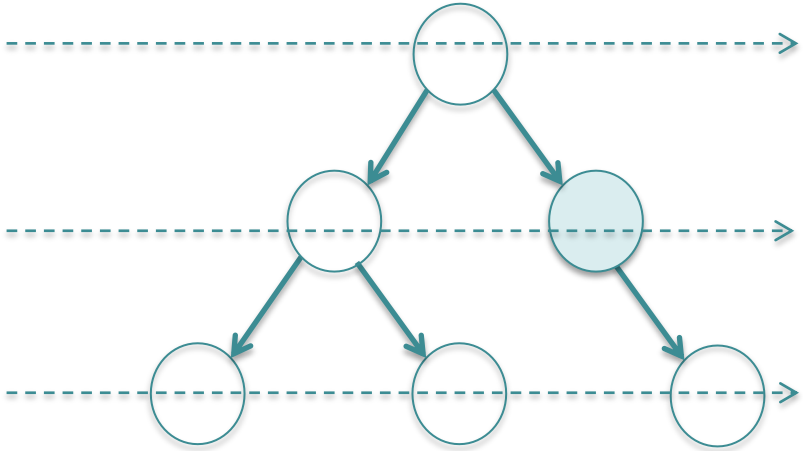


Temporary storage
[T.root.right, T.root.left.left, T.root.left.right]

Breadth-First Traversal

visit(T.root)

visit(T.root.left); visit(T.root.right)



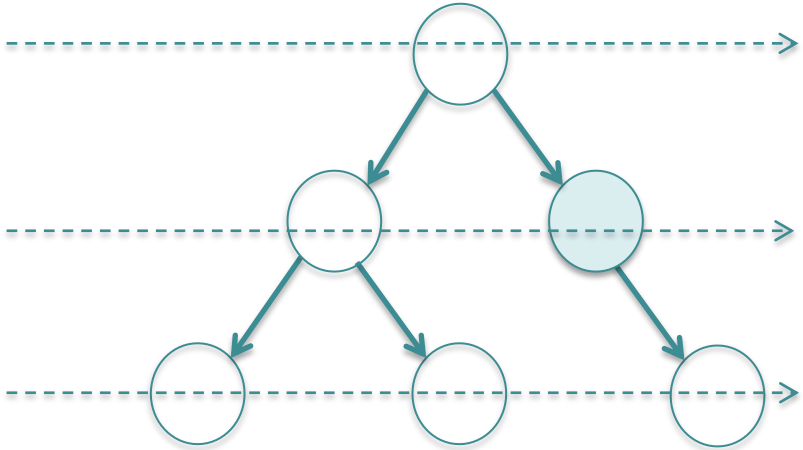
Temporary storage
[T.root.right, T.root.left.left, T.root.left.right]

We know visit the 2nd node in level 1

Breadth-First Traversal

visit(T.root)

visit(T.root.left); visit(T.root.right)



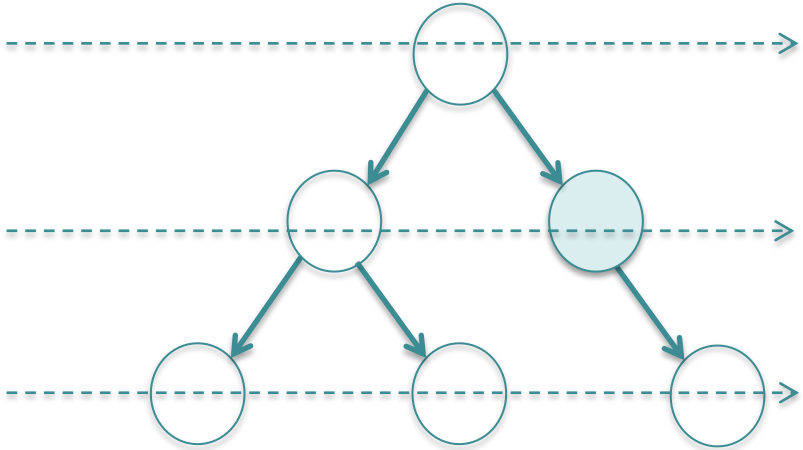
Temporary storage
[T.root.right, T.root.left.left, T.root.left.right, T.root.right.right]

Save information about its children in a temporary storage

Breadth-First Traversal

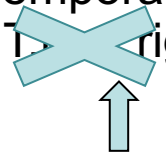
visit(T.root)

visit(T.root.left); visit(T.root.right)



Temporary storage

[T…right, T.root.left.left, T.root.left.right, T.root.right.right]

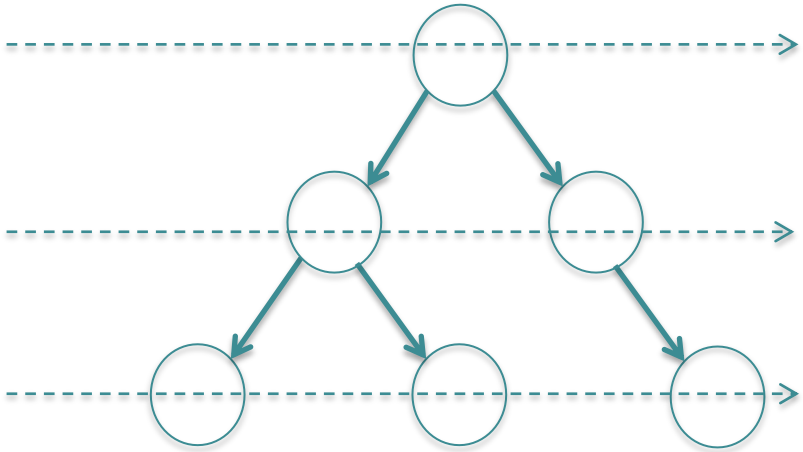


This node is removed from the temporary storage

Breadth-First Traversal

visit(T.root)

visit(T.root.left); visit(T.root.right)



Temporary storage
[T.root.left.left, T.root.left.right, T.root.right.right]

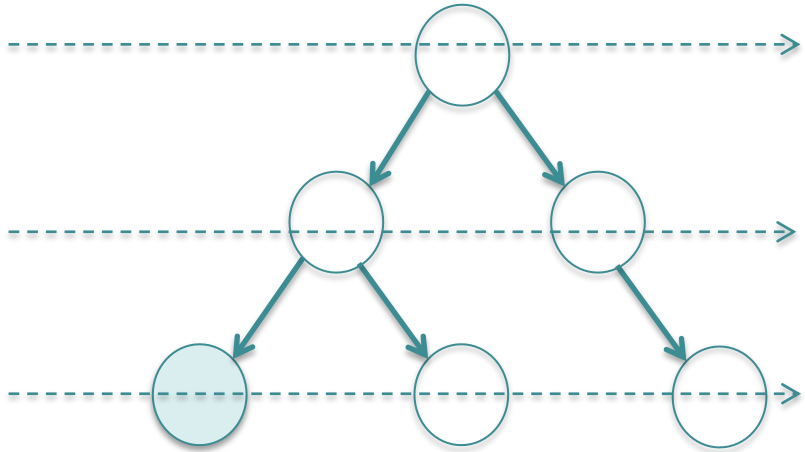
We move to the next element in the temporary storage!

Breadth-First Traversal

visit(T.root)

visit(T.root.left); visit(T.root.right)

visit(T.root.left.left)



Temporary storage
[T.root.left.left, T.root.left.right, T.root.right.right]

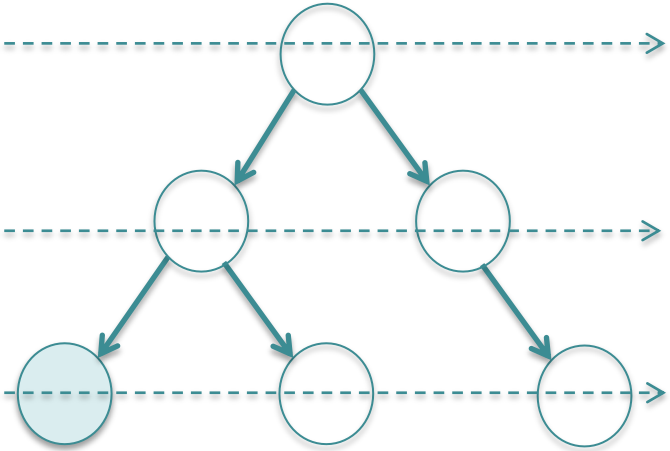
This node has no children, so no information needs to be stored in the temporary storage!

Breadth-First Traversal

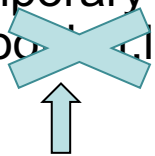
visit(T.root)

visit(T.root.left); visit(T.root.right)

visit(T.root.left.left)



Temporary storage
[T.root.left, T.root.left.right, T.root.right.right]



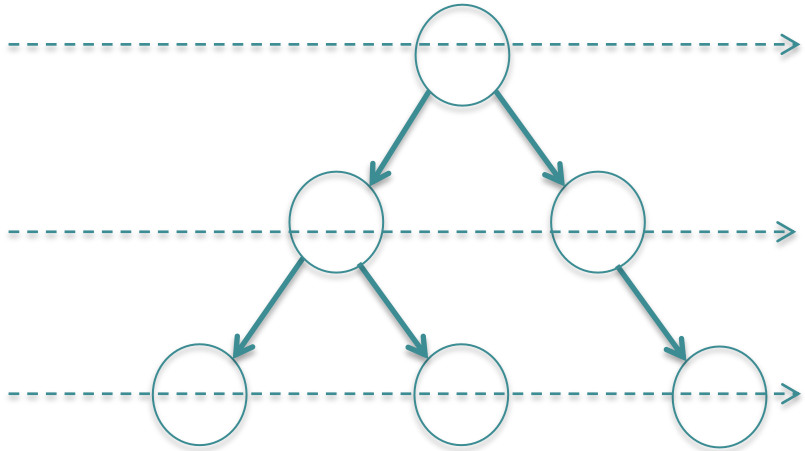
This node is then removed from the temporary storage

Breadth-First Traversal

visit(T.root)

visit(T.root.left); visit(T.root.right)

visit(T.root.left.left)



Temporary storage
[T.root.left.right, T.root.right.right]

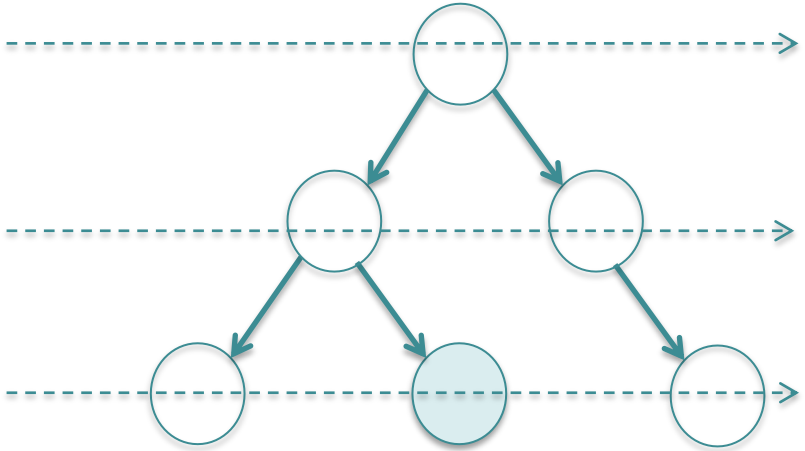
We then move to the next node

Breadth-First Traversal

visit(T.root)

visit(T.root.left); visit(T.root.right)

visit(T.root.left.left); visit(T.root.left.right)



Temporary storage
[T.root.left.right, T.root.right.right]

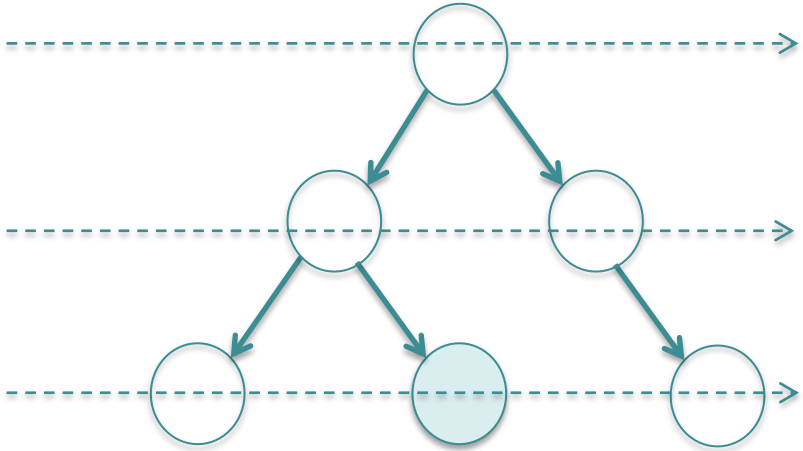
Similarly, this move is visited and removed as as it has no children

Breadth-First Traversal

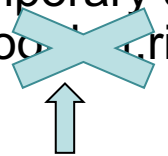
visit(T.root)

visit(T.root.left); visit(T.root.right)

visit(T.root.left.left); visit(T.root.left.right)



Temporary storage
[T.root.left.right, T.root.right.right]

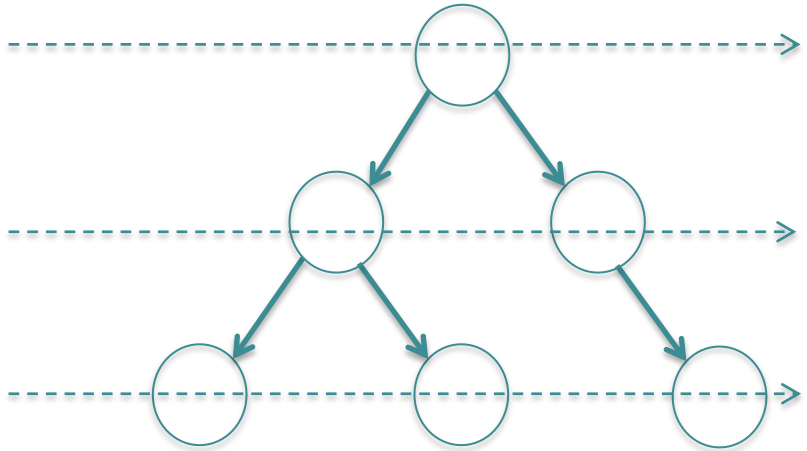


Breadth-First Traversal

visit(T.root)

visit(T.root.left); visit(T.root.right)

visit(T.root.left.left); visit(T.root.left.right)



Temporary storage
[T.root.right.right]

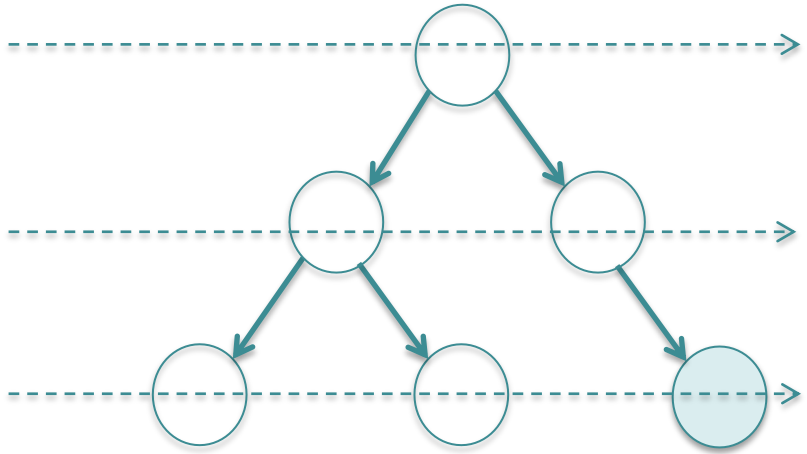
We then move to the last node

Breadth-First Traversal

visit(T.root)

visit(T.root.left); visit(T.root.right)

visit(T.root.left.left); visit(T.root.left.right)
visit(T.root.right.right)



Temporary storage
[T.root.right.right]

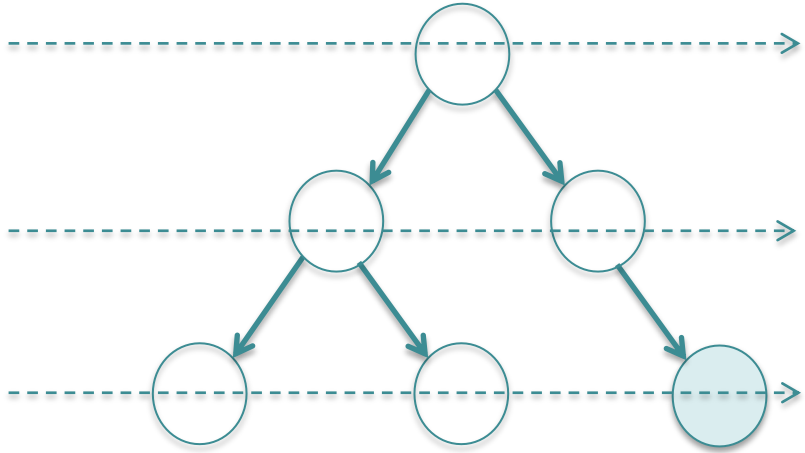
This node is now visited and removed the memory storage!

Breadth-First Traversal

visit(T.root)

visit(T.root.left); visit(T.root.right)

visit(T.root.left.left); visit(T.root.left.right)
visit(T.root.right.right)



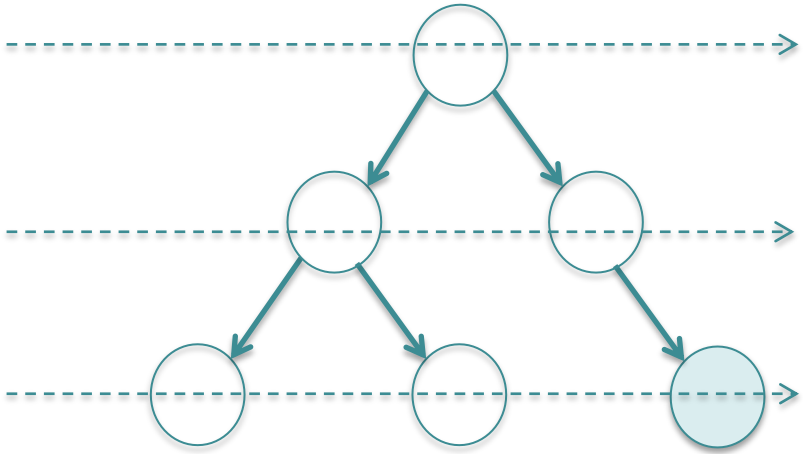
Temporary storage
[T.root.left.right]

Breadth-First Traversal

visit(T.root)

visit(T.root.left); visit(T.root.right)

visit(T.root.left.left); visit(T.root.left.right)
visit(T.root.right.right)



Temporary storage
[ ]

The memory is empty once all the nodes are visited!

The temporary storage used here is queue data structure as nodes are inserted in one side and removed from the other side.

```
function breadth-first(root)
   Q ← new Queue()
   enqueue-if(Q,root)
   while !ISEMPTY(Q)  do
      t ← PEEK(Q)
      visit(t)
      enqueue-if(Q,left(t))
      enqueue-if(Q,right(t))
      DEQUEUE(Q)
   end while
end function
```

```
function enqueue-if(Q,t)
   if !NULL(t)  then
      ENQUEUE(Q,t)
end function
```

```
function breadth-first(root)
    Q ← new Queue()
    enqueue-if(Q,root)
    while !ISEMPTY(Q) do
        t ← PEEK(Q)
        visit(t)
        enqueue-if(Q,left(t))
        enqueue-if(Q,right(t))
        DEQUEUE(Q)
    end while
end function
```

```
function enqueue-if(Q,t)
    if !NULL(t) then
        ENQUEUE(Q,t)
end function
```

```
function breadth-first(root)
    Q ← new Queue()
    enqueue-if(Q,root)
    while !ISEMPTY(Q)  do
        t ← PEEK(Q)
        visit(t)
        enqueue-if(Q,left(t))
        enqueue-if(Q,right(t))
        DEQUEUE(Q)
    end while
end function
```

```
function enqueue-if(Q,t)
    if !NULL(t)  then
        ENQUEUE(Q,t)
end function
```

```
function breadth-first(root)
    Q ← new Queue()
    enqueue-if(Q,root)
    while !ISEMPTY(Q)  do
        t ← PEEK(Q)
        visit(t)
        enqueue-if(Q,left(t))
        enqueue-if(Q,right(t))
        DEQUEUE(Q)
    end while
end function
```

```
function enqueue-if(Q,t)
    if !NULL(t)  then
        ENQUEUE(Q,t)
end function
```

```
function breadth-first(root)
    Q ← new Queue()
    enqueue-if(Q,root)
    while !ISEMPTY(Q)  do
        t ← PEEK(Q)
        visit(t)
        enqueue-if(Q,left(t))
        enqueue-if(Q,right(t))
        DEQUEUE(Q)
    end while
end function
```

```
function enqueue-if(Q,t)
    if !NULL(t)  then
        ENQUEUE(Q,t)
end function
```

```
function breadth-first(root)
    Q ← new Queue()
    enqueue-if(Q,root)
    while !ISEMPTY(Q)  do
        t ← PEEK(Q)
        visit(t)
        enqueue-if(Q,left(t))
        enqueue-if(Q,right(t))
        DEQUEUE(Q)
    end while
end function
```

```
function enqueue-if(Q,t)
    if !NULL(t)  then
        ENQUEUE(Q,t)
end function
```

```
function breadth-first(root)
   Q ← new Queue()
   enqueue-if(Q,root)
   while !ISEMPTY(Q)  do
      t ← PEEK(Q)
      visit(t)
      enqueue-if(Q,left(t))
      enqueue-if(Q,right(t))
      DEQUEUE(Q)
   end while
end function
```

```
function enqueue-if(Q,t)
   if !NULL(t)  then
      ENQUEUE(Q,t)
end function
```

```
function breadth-first(root)
   Q ← new Queue()
   enqueue-if(Q,root)
   while !ISEMPTY(Q)  do
      t ← PEEK(Q)
      visit(t)
      enqueue-if(Q,left(t))
      enqueue-if(Q,right(t))
      DEQUEUE(Q)
   end while
end function
```

```
function enqueue-if(Q,t)
   if !NULL(t)  then
      ENQUEUE(Q,t)
end function
```

```
function breadth-first(root)
   Q ← new Queue()
   enqueue-if(Q,root)
   while !ISEMPTY(Q)  do
      t ← PEEK(Q)
      visit(t)
      enqueue-if(Q,left(t))
      enqueue-if(Q,right(t))
      DEQUEUE(Q)
   end while
end function
```

```
function enqueue-if(Q,t)
   if !NULL(t)  then
      ENQUEUE(Q,t)
end function
```

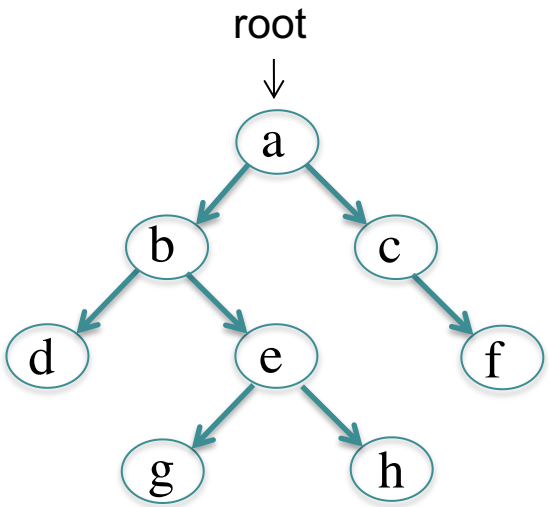
```
function breadth-first(root)
    Q ← new Queue()
    enqueue-if(Q,root)
    while !ISEMPTY(Q)  do
        t ← PEEK(Q)
        visit(t)
        enqueue-if(Q,left(t))
        enqueue-if(Q,right(t))
        DEQUEUE(Q)
    end while
end function
```

```
function enqueue-if(Q,t)
    if !NULL(t)  then
        ENQUEUE(Q,t)
end function
```

```
function breadth-first(root)
    Q ← new Queue()
    enqueue-if(Q,root)
    while !ISEMPTY(Q)  do
        t ← PEEK(Q)
        visit(t)
        enqueue-if(Q,left(t))
        enqueue-if(Q,right(t))
        DEQUEUE(Q)
    end while
end function
```

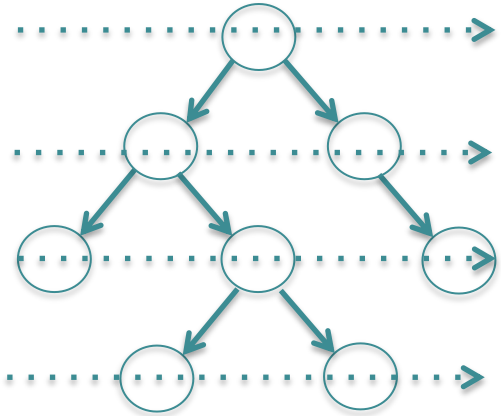
```
function enqueue-if(Q,t)
    if !NULL(t)  then
        ENQUEUE(Q,t)
end function
```

Your task:



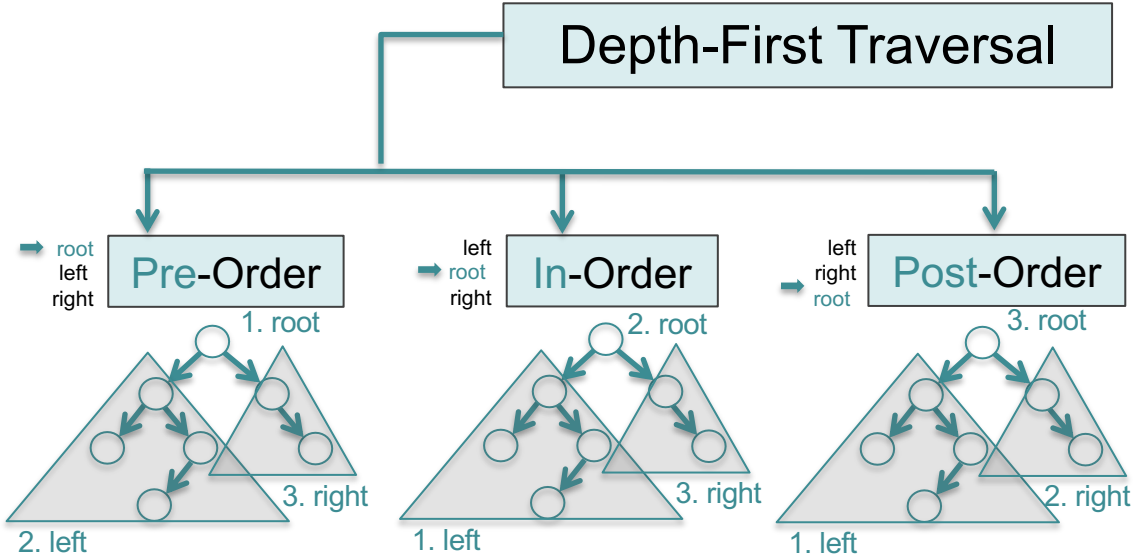
Summary

Breadth-First Traversal



Iterative algorithm

Depth-First Traversal



Recursive algorithm