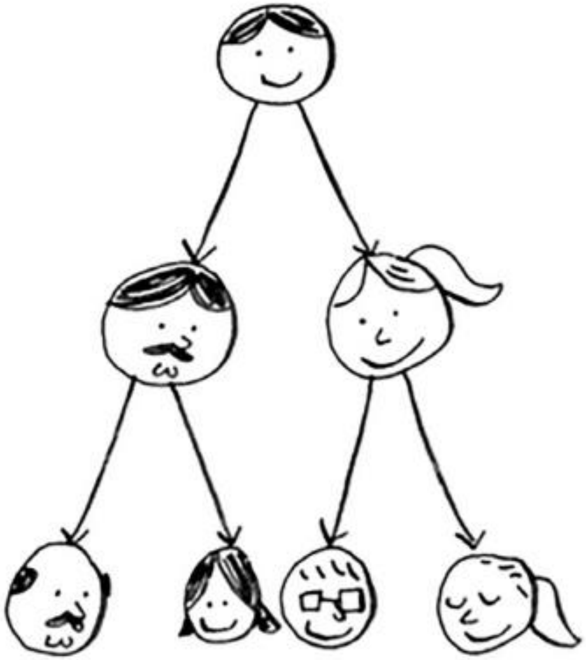
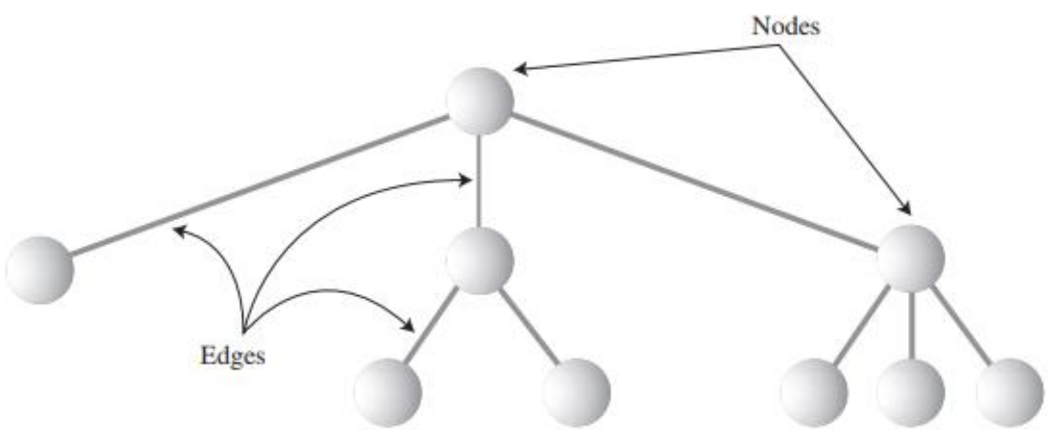
CDK2AAB4
STRUKTUR DATA



Tree Data Structure

What is a Tree?

- A tree consists of *nodes* connected by *edges.*



- A tree is an instance of a more general category called a *graph* (*we'll talk about this later*).

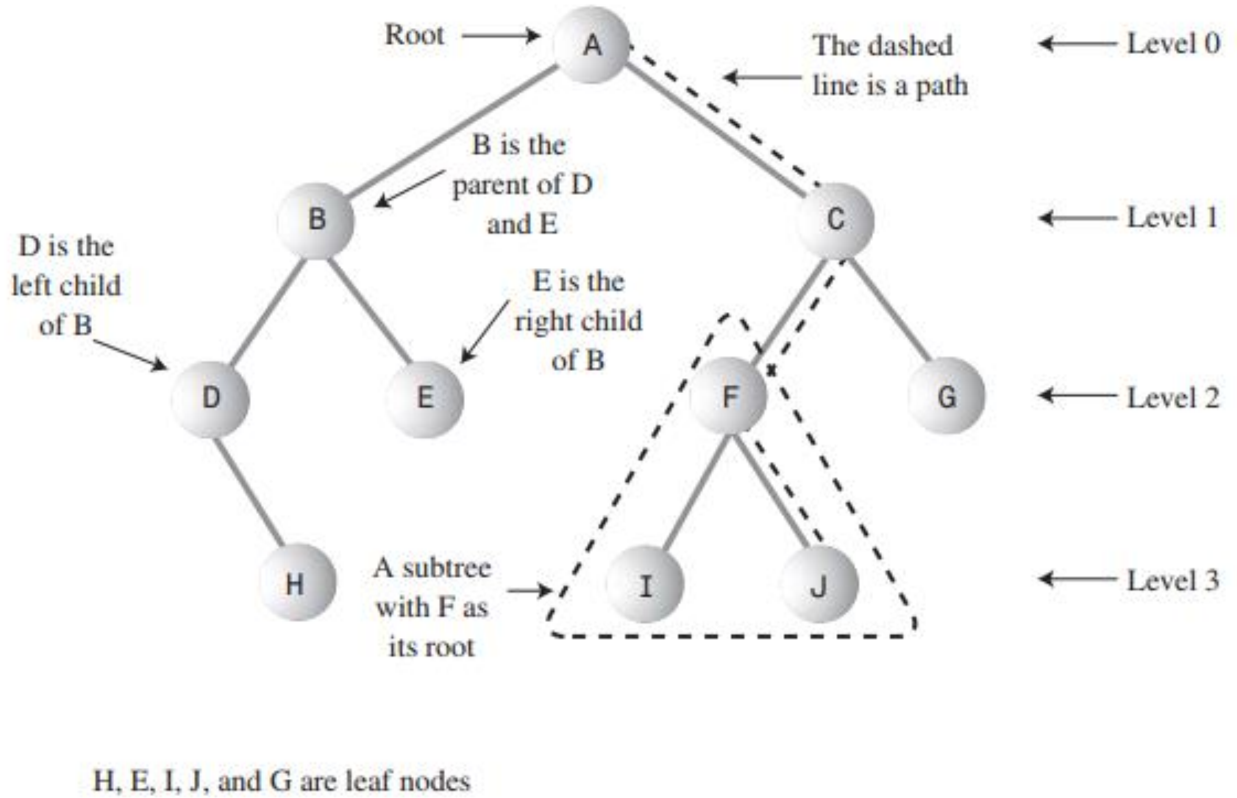
What is a Node?

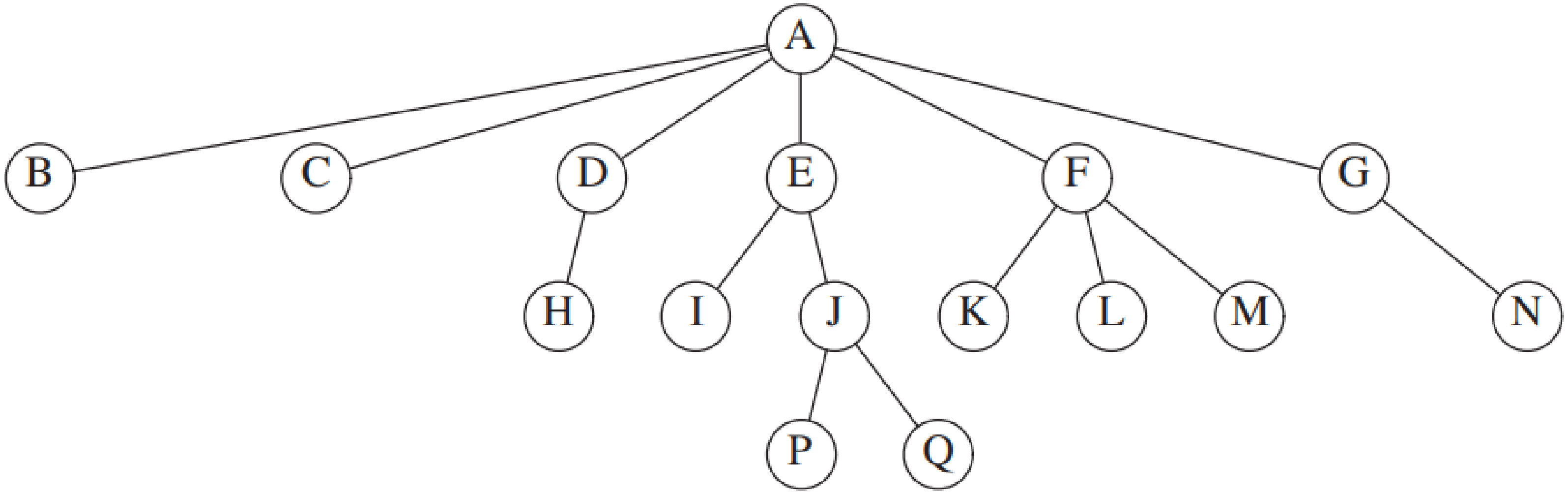
- Nodes often represent such entities as people, car parts, airline reservations, and so on
  - in other words, the typical items we store in any kind of data structure.
  - In an OOP language, these real-world entities are represented by objects.

What is an Edge?

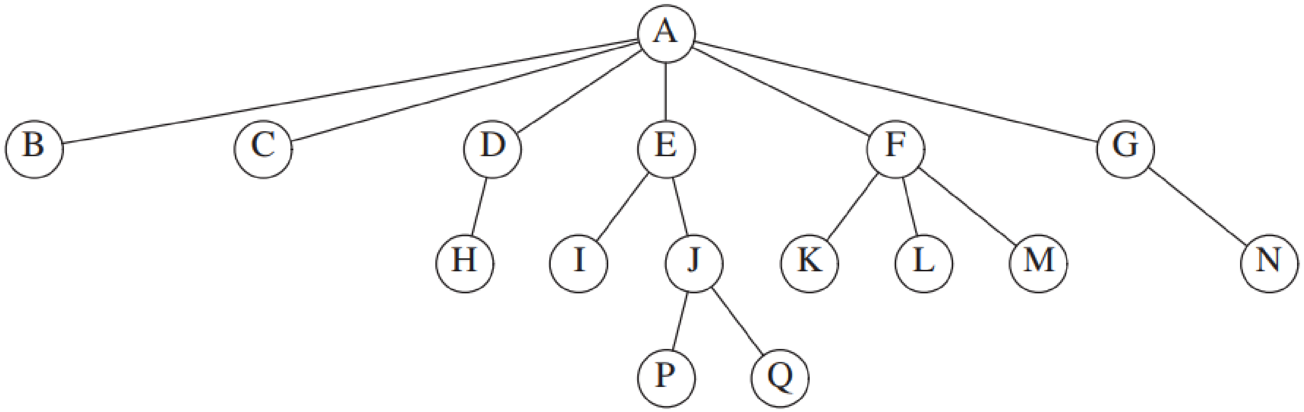
- The lines (edges) between the nodes represent the way the nodes are related.
  - Roughly speaking, the lines represent convenience: It's easy (and fast) for a program to get from one node to another if there is a line connecting them.
  - In fact, the *only* way to get from node to node is to follow a path along the lines.
  - Generally, you are restricted to going in one direction along edges: from the root downward.

Tree Terminology

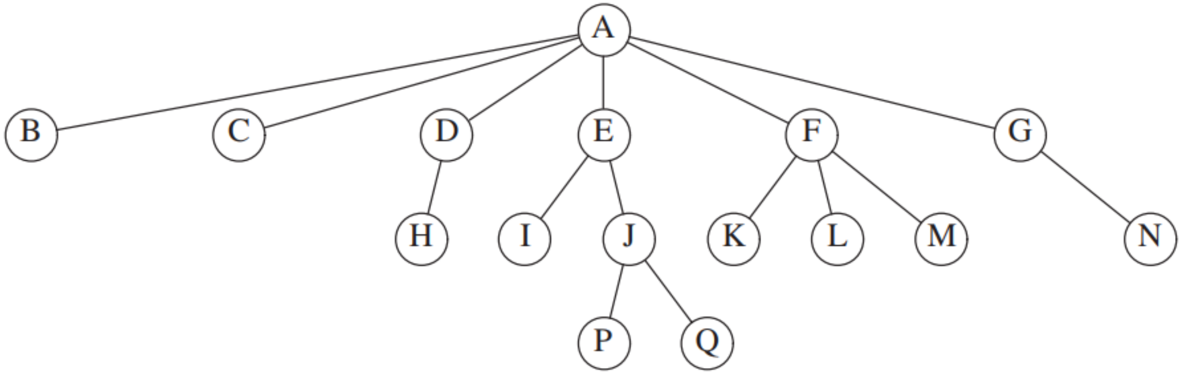




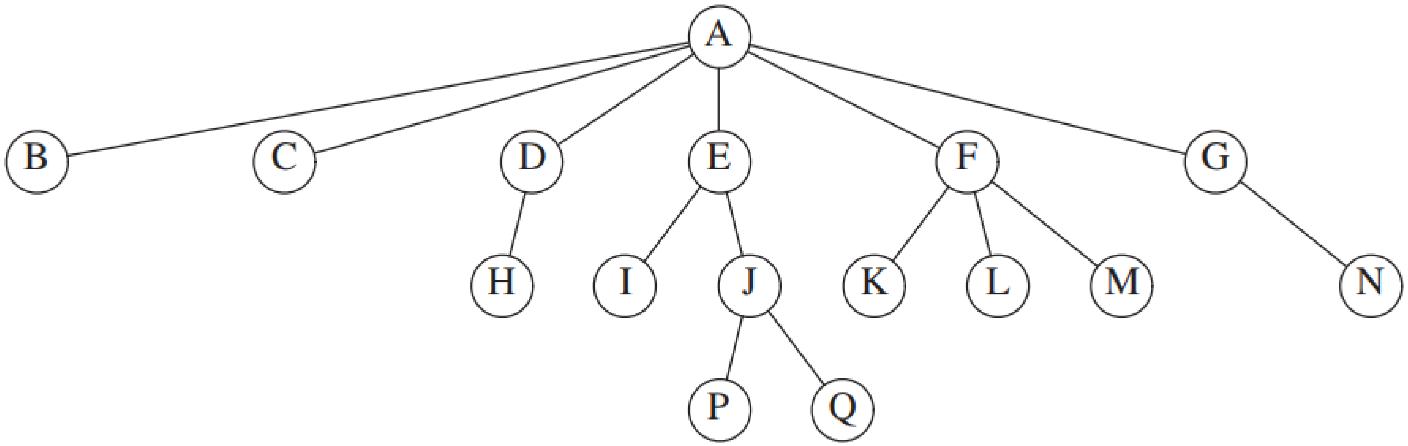
- In the tree above, the **root** is *A*.
- Node *F* has *A* as a **parent** and *K, L,* and *M* as children.
  - Each node may have an arbitrary number of children, possibly zero.



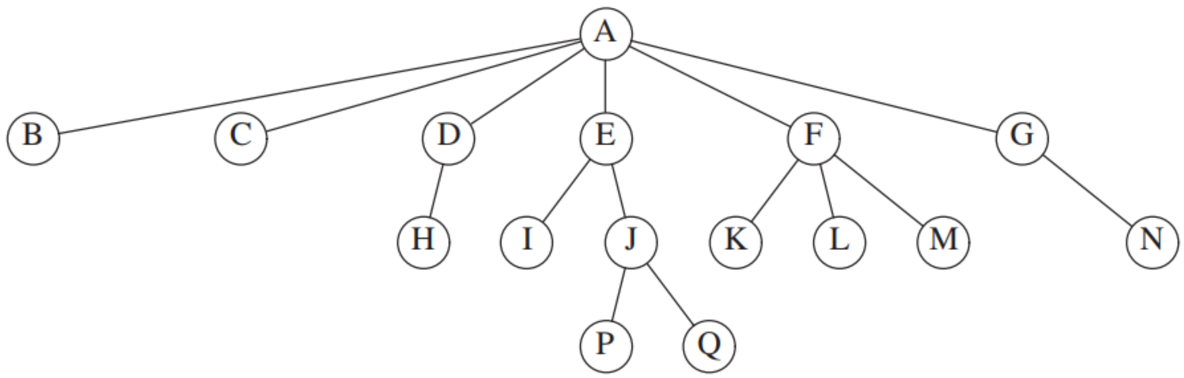
- Nodes with no children are known as **leaves**
  - the leaves in the tree above are *B, C, H, I, P, Q, K, L, M*, and *N*.
  - Nodes with the same parent are **siblings**
    - thus, *K, L,* and *M* are all siblings.
    - **Grandparent** and **grandchild** relations can be defined in a similar manner.



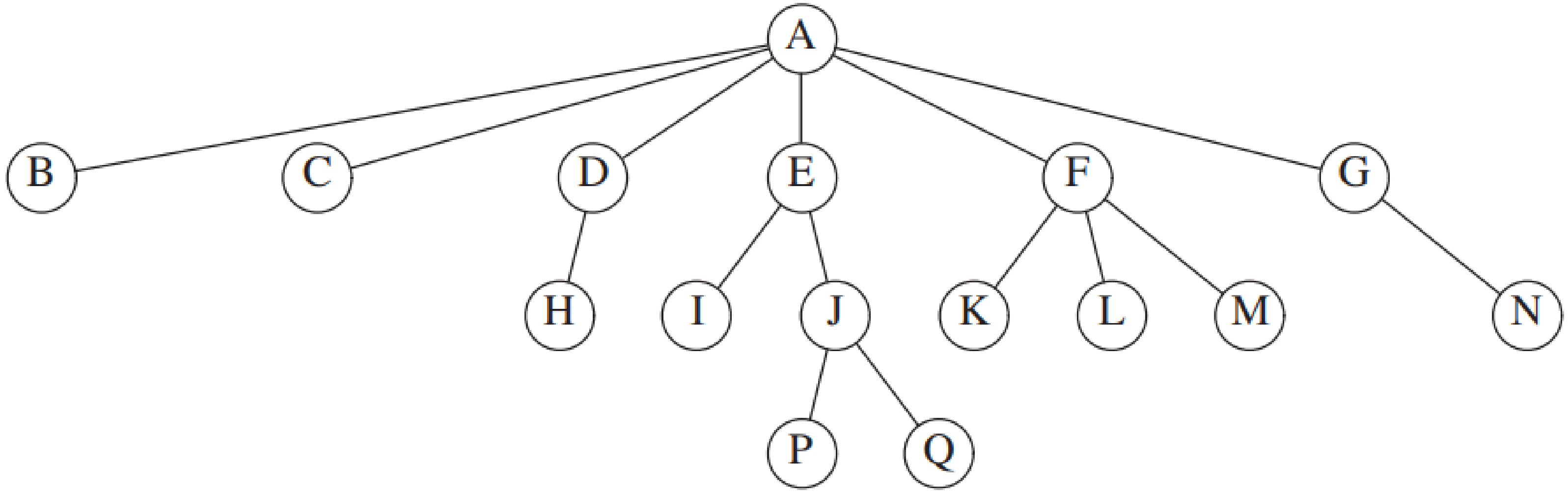
- A **path** from node n_1 to n_k is defined as a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1} for $1 \leq i < k$.
  - The **length** of this path is the number of edges on the path, namely, $k - i$.
  - There is a path of length zero from every node to itself.
  - Notice that in a tree there is exactly one path from the root to each node.



- For any node n_i, the **depth** of n_i is the length of the unique path from the root to n_i.
  - Thus, the root is at depth 0.
- The depth of a tree is equal to the depth of the deepest leaf.
  - this is always equal to the height of the tree



- The **height** of n_i is the length of the longest path from n_i to a leaf.
  - Thus, all leaves are at height 0
- The height of a tree is equal to the height of the root.
  - For the tree in above figure, *E* is at depth 1 and height 2; *F* is at depth 1 and height 1; the height of the tree is 3.



- If there is a path from n_1 to n_2, then n_1 is an **ancestor** of n_2 and n_2 is a **descendant** of n_1.
  - If $n_1 != n_2$, then n_1 is a **proper ancestor** of n_2 and n_2 is a **proper descendant** of n_1.

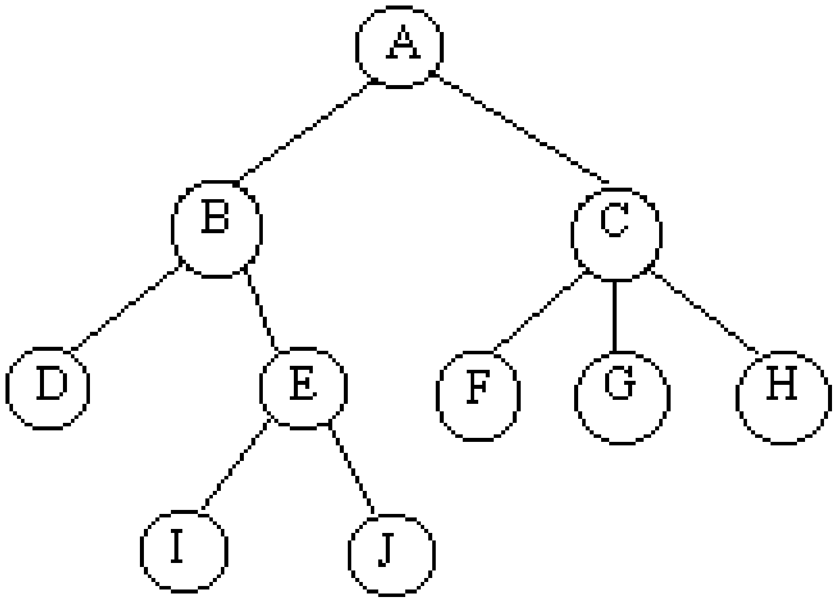
Subtree

- Any node may be considered to be the root of a *subtree*, which consists of its children, and its children's children, and so on.
- If you think in terms of families, a node's subtree contains all its descendants.

Question?

Exercise on Tree Terminology

- Root =
- Sibling C =
- Parent F =
- Child B =
- Leaf =
- Level E =
- Tree height =
- Degree B =
- Ancestor I =
- Descendant B =



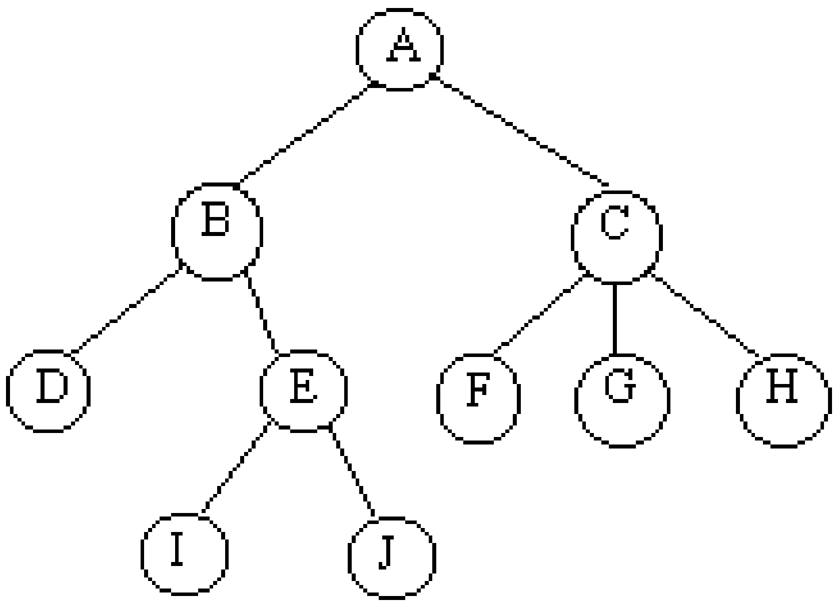
Exercise on Tree Terminology

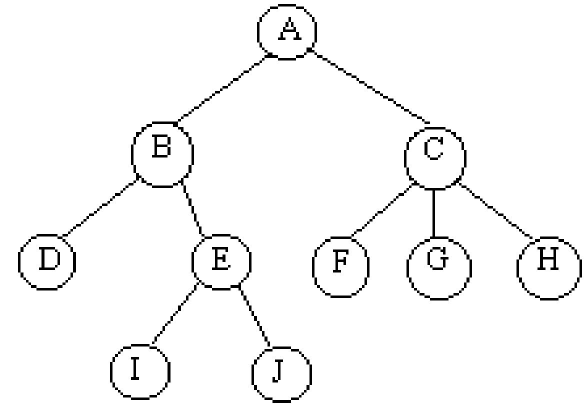
- Create the tree
- Dataset: {A, X, W, H, B, E, S}
- Root: A
- Ancestor of S: {E, A}
- {X, W, E} are siblings
- {H, B} are descendant and both are children of W

Tree Notations / Representing Tree

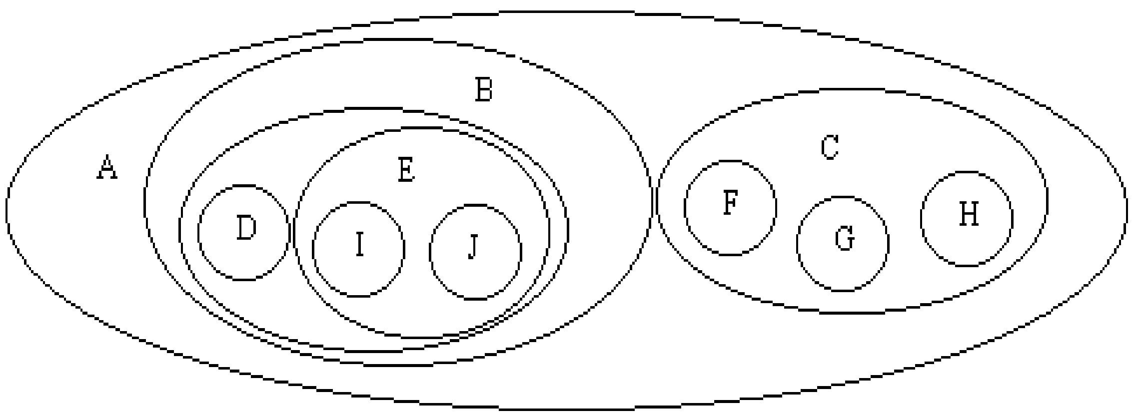
- Tree Diagram Notation
  - Classical node-link diagrams
- Venn Diagram Notation
  - Nested sets / Tree Maps
- Bracket Notation
  - Nested Parentheses
- Level Notation
  - Outlines / tree views

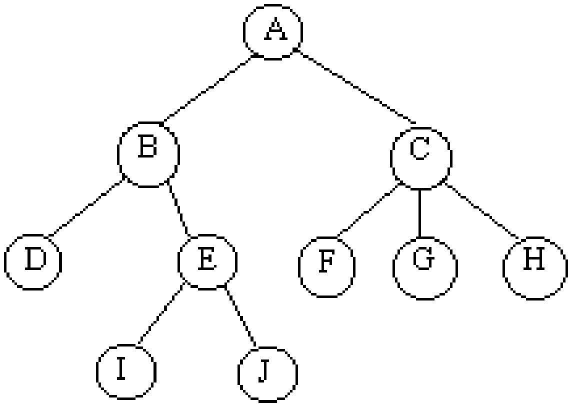
Tree Diagram Notation





Venn Diagram Notation

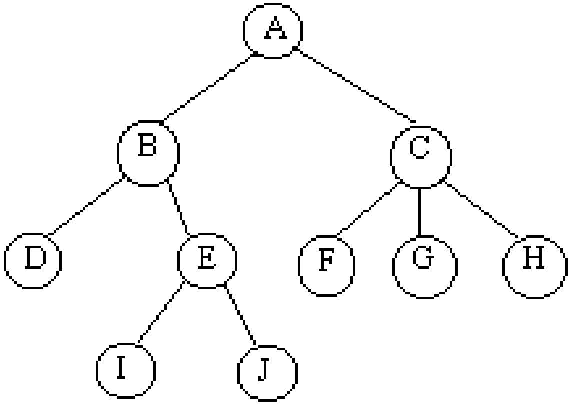


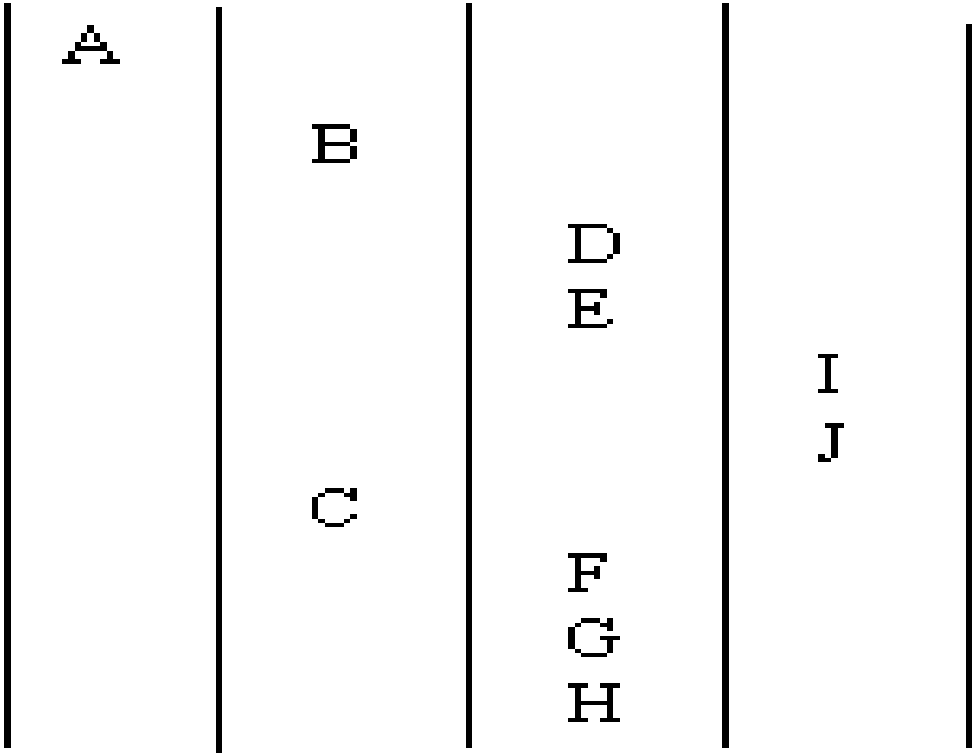


Bracket Notation

A (B (D E (I J)) C (F G H))

Level Notation





Exercise

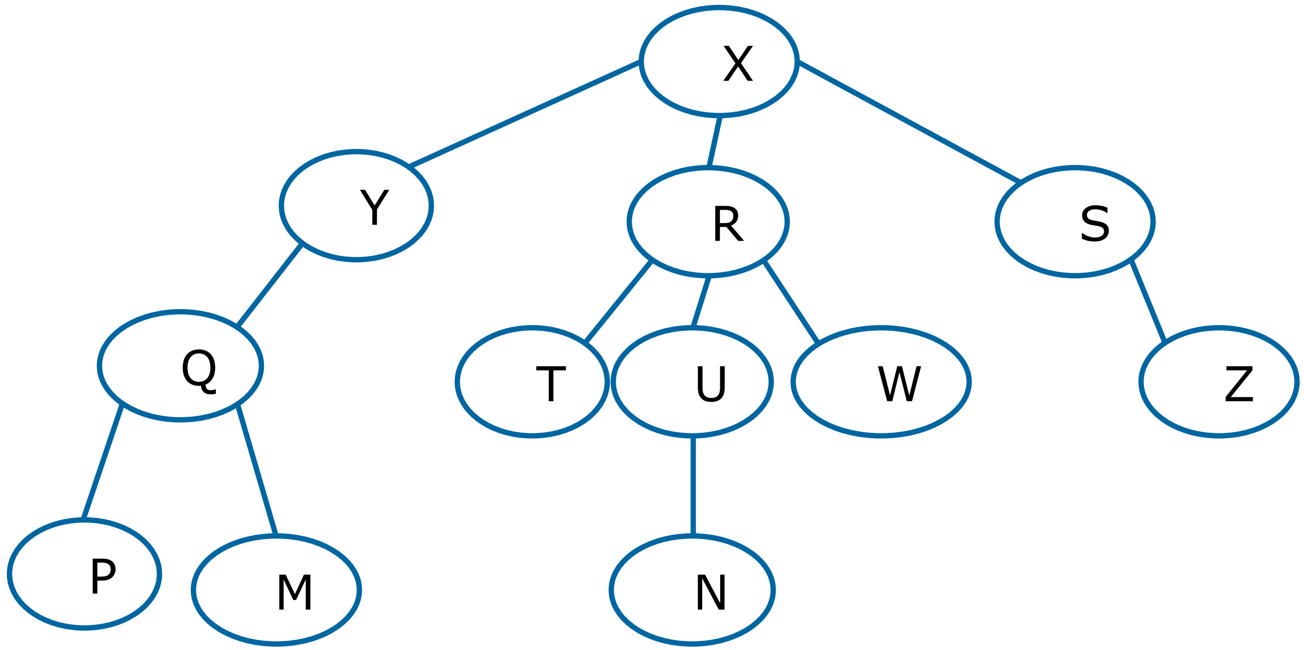
Show the tree representation of the following parenthetical notation:

```
a (b (c d) e f (g h))
```


Reference:
Data Structures: A Pseudocode Approach with C, Second Edition by Richard F. Gilberg & Behrouz A. Forouzan (2005)

Exercise

- Create the tree in Venn Diagram, Bracket, and level notation

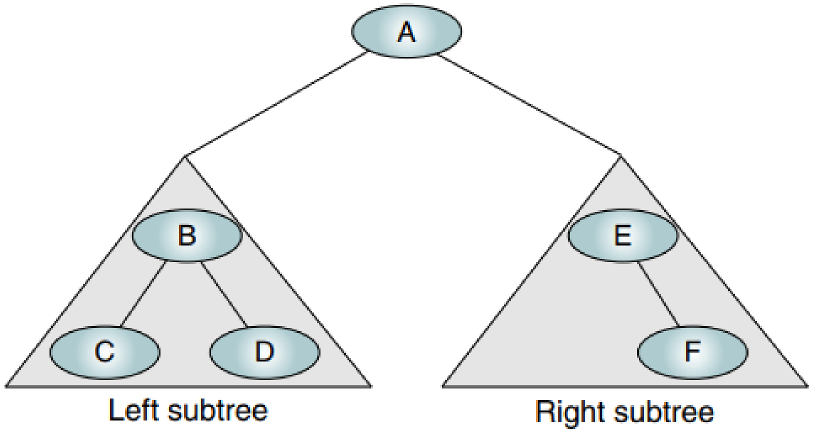


Question?

Binary Tree Data Structure

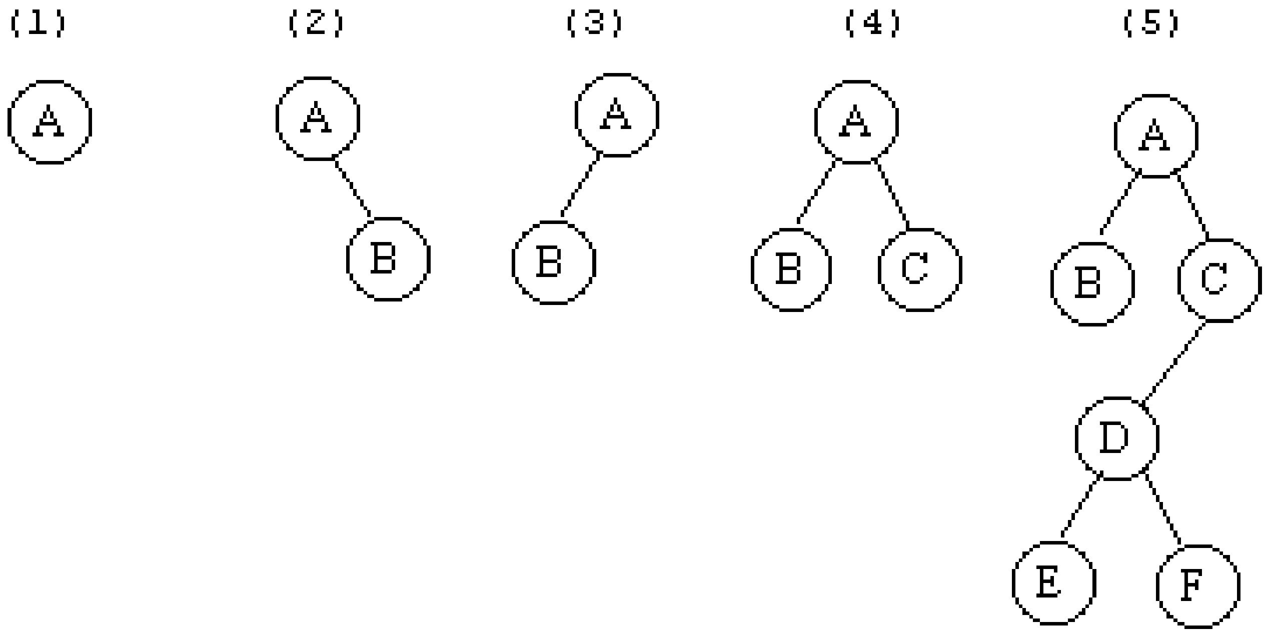
Binary Tree

- A binary tree is a tree in which no node can have more than two subtrees.
- The maximum outdegree for a node is two.
  - In other words, a node can have zero, one, or two subtrees. These subtrees are designated as the left subtree and the right subtree.



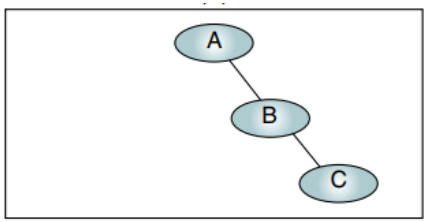
Binary Tree

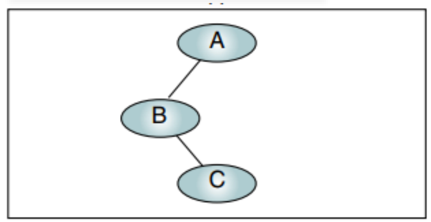
- To better understand the structure of binary trees, study figures below



Properties of Binary Tree

- Height of binary tree
  - Maximum height
    - Given that we need to store N nodes in a binary tree, the maximum height, $H_{max} = N - 1$.





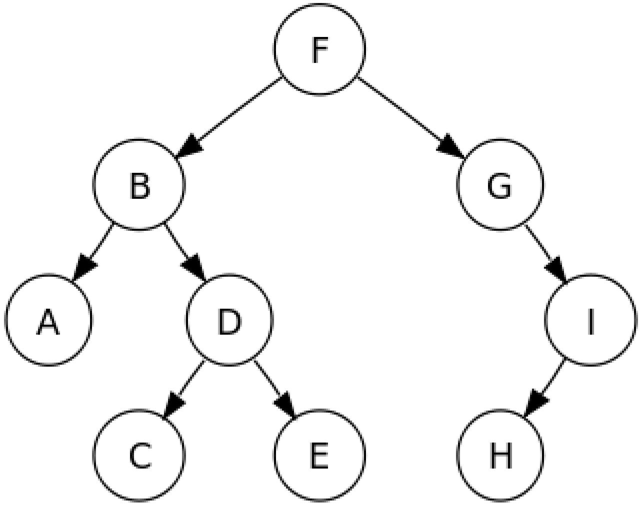
  - Minimum height
    - The minimum height of the tree, H_{min}, is determined by the following formula: $H_{min} = \lfloor \log_2 N \rfloor$.

Properties of Binary Tree

- Number of nodes of binary tree
  - Minimum nodes
    - Given a height of the binary tree, H, the minimum number of nodes in the tree are given as $N_{min} = H + 1$.
  - Maximum nodes
    - The formula for the maximum number of nodes is derived from the fact that each node can have only two descendants. Given a height of the binary tree, H, the maximum number of nodes in the tree is given as $N_{max} = 2^{H+1} - 1$.

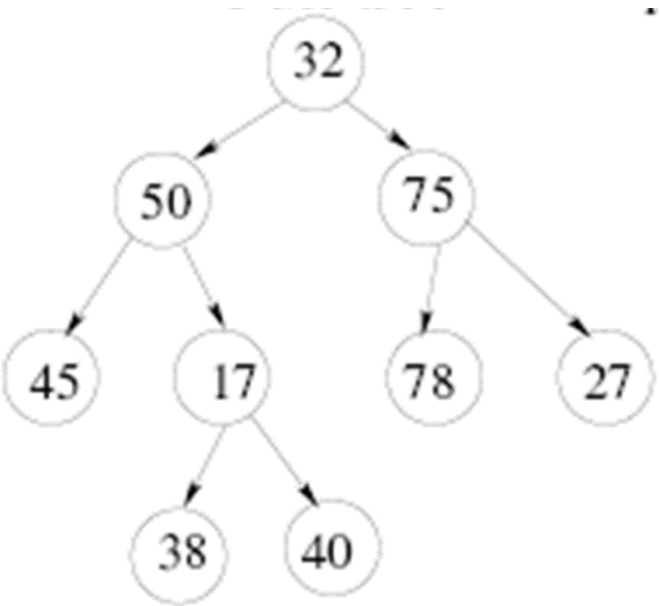
Types of Binary Tree

- Full Binary Tree
- Complete Binary Tree
- Skewed Binary Tree



Full Binary Tree

- Every node has **exactly 2 children** or **0 children** (no nodes have only one child).



Properties of Full Binary Tree

- Number of nodes of full binary tree

  Given a height of full binary tree, H,

  - the **minimum number of nodes**, $N_{min} = 2H + 1$.
  - the **maximum number of nodes**, $N_{max} = 2^{H+1} - 1$.
  - the **minimum nodes** (non root) **of each level**, $Nl_{min} = 2$.
  - the **maximum nodes of each level**, $Nl_{max} = 2^{H}$.

Properties of Full Binary Tree

- Number of leaves of full binary tree

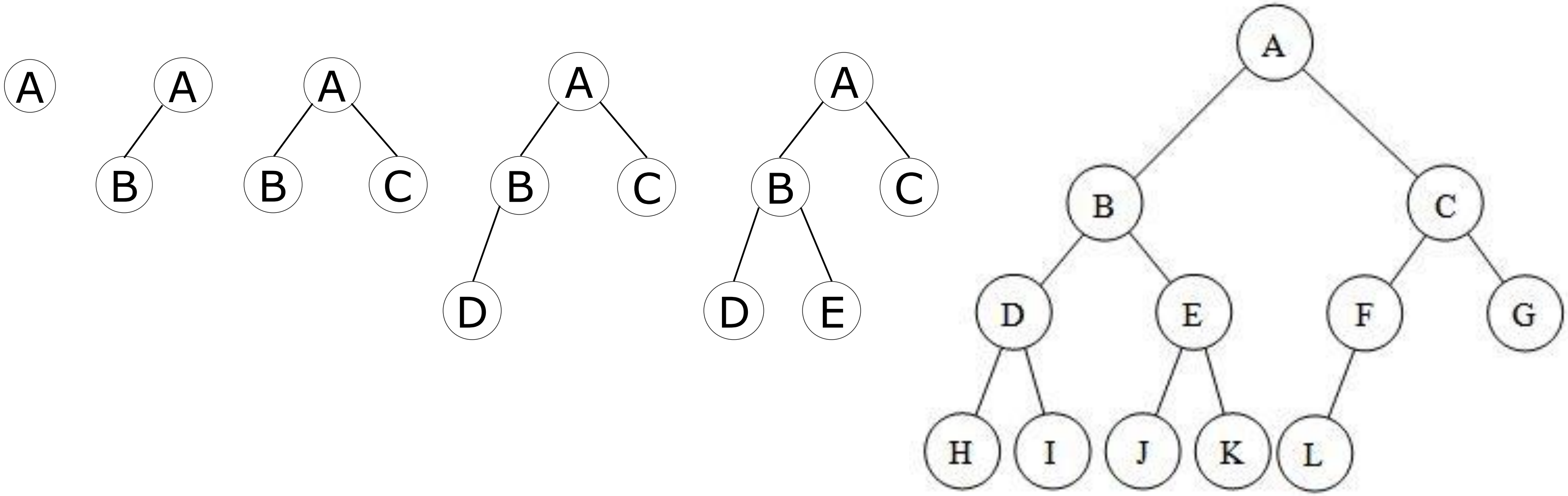
  Given a height of full binary tree, H,

  – the **minimum number of leaves**, $L_{min} = H + 1$.

  – the **maximum number of leaves**, $L_{max} = 2^H$.

Complete Binary Tree

- a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible



Properties of Complete Binary Tree

- Number of nodes of complete binary tree

  Given a height of complete binary tree, H,

  – the **minimum number of nodes**, $N_{min} = 2^H$.

  – the **maximum number of nodes**, $N_{max} = 2^{H+1} - 1$.

  – the **minimum nodes of each level** (not the last level) ,$Nl_{min} = 2^H$.

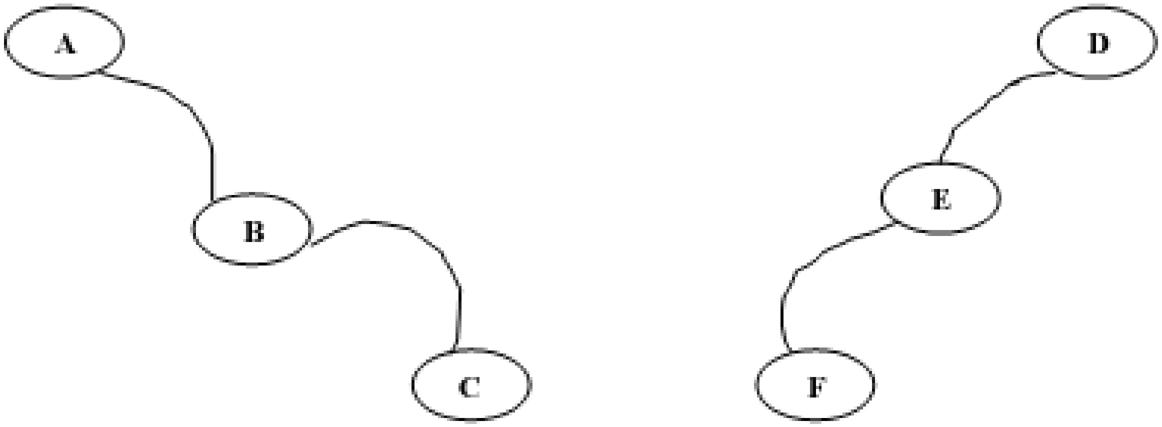
  – the **minimum nodes of the last level**, $Nl_{min} = 1$.

  – the **maximum nodes of each level**, $Nl_{max} = 2^H$.

- Number of NIL links (wasted pointers) of N nodes is $N + 1$.

Skewed Tree

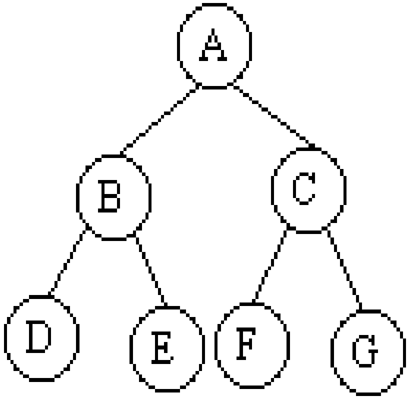
- Binary Tree with an unbalanced left subtree and right subtree.

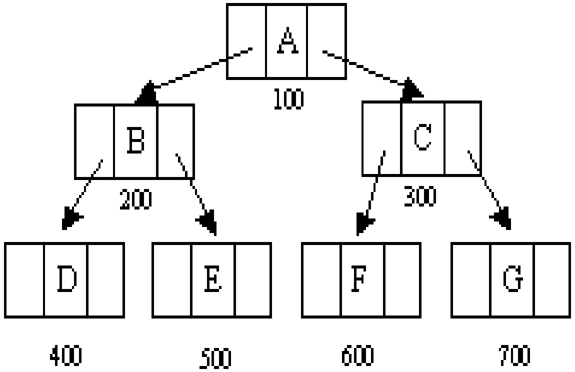


ADT Binary Tree

- Array Representation
- Linked list representation

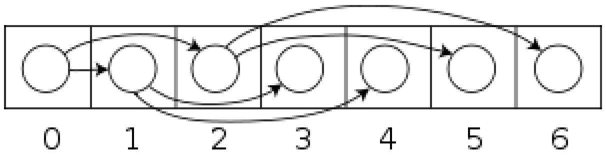
id	value
1	A
2	B
3	C
4	D
5	E
6	F
7	G

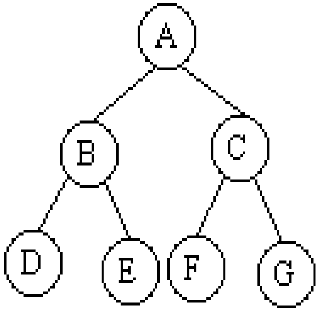




Array Representation

- if a node has an index *i*, its children are found at indices :
  - Left child : $2i + 1$
  - Right child : $2i + 2$
- while its parent (if any) is found at index $\left\lfloor \frac{i-1}{2} \right\rfloor$
  - (assuming the root has index zero)



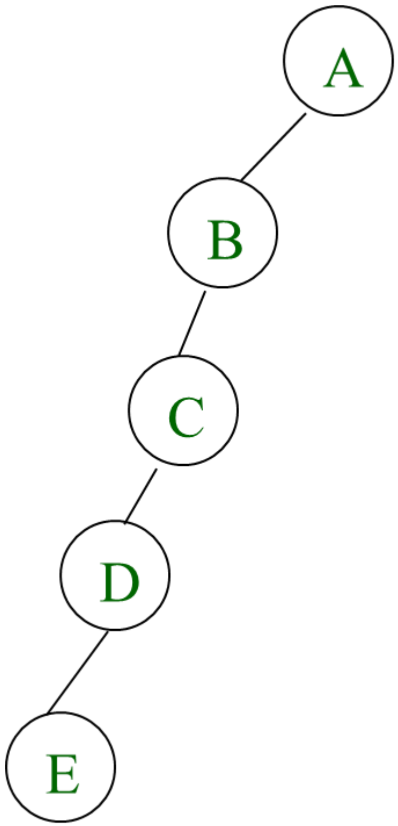


Array Representation

- Problem :

1	2	3	4	5	6	7	8	...	16
A	B	-	C	-	-	-	D	...	E

  - Waste space
  - Insertion / deletion problem



- Array Representation is good for Complete Binary Tree types
  - Binary Heap Tree

Linked List Representation

```
type Infotype : integer
type Address : pointer to Node

type Node <
  info : Infotype
  left : Address
  right : Address
>
```

left	info	right

```
type BinTree : Address

dictionary
  root : BinTree
```

Binary Tree : Create New Node

```
function createNode( x : Infotype ) → Address
dictionary
  procedure allocate( BinTree )
  N : BinTree
algorithm
   allocate( N )
   N→info = x
   N→left = NIL
   N→right = NIL
   return N
```

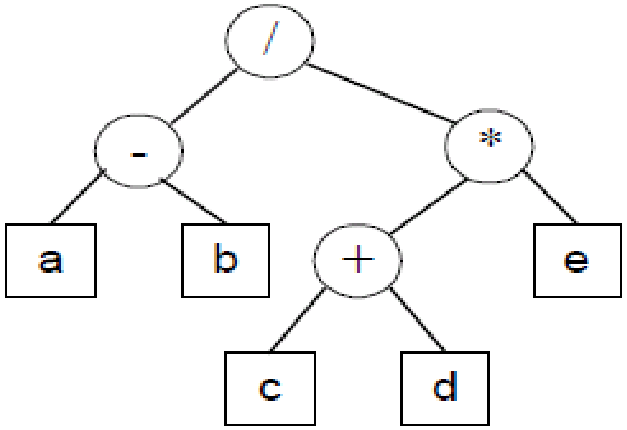
Example Application of Binary Tree

- Arithmetic Expression Tree
- Binary Search Tree
- Decision Tree
- AVL Tree
- Priority Queue
  - Binary Heap Tree

Arithmetic Expression Tree

- The leaves of an expression tree are **operands**, such as constants or variable names, and the other nodes contain **operators**
- This particular tree happens to be binary, because all the operators are binary.
- Example :

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



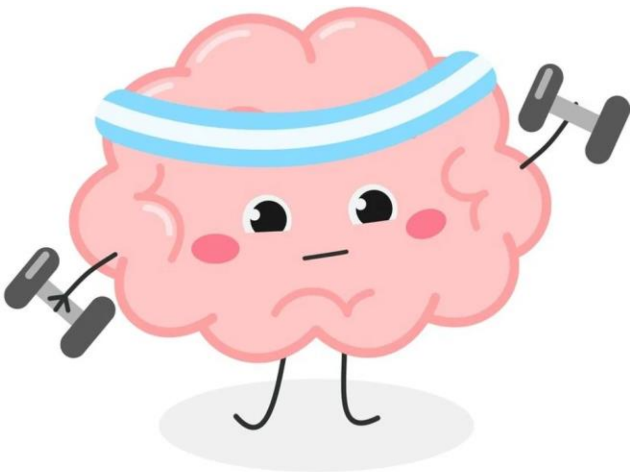
Exercise – create the tree

- (a + b) / (c – d * e) + f + g * h / i
- ((A + B) * (C + D)) / (E + F * H)
- (6 - (12 - (3 + 7))) / ((1 + 0) + 2) * (2 *(3 + 1))

Let's Keep Our Brain Sharp

Write functions that take only a pointer to the root of binary tree, T, and compute:

1. The number of nodes in T
2. The number of leaves in T
3. The number of full nodes in T



Ref.
Data Structures and Algorithm Analysis in C++
by Mark Allen Weiss (2014)

Binary Search Tree (BST)

- Ordered / sorted binary tree
- BST Invariant
  - Left subtree has smaller elements
  - Right subtree has larger elements

BST : Insert new Node

```
procedure insertBST( in x : Infotype, in/out N : BinTree )
dictionary
  function createNode( Infotype ) → Address

algorithm
  if N == NIL then
    N = createNode( x )
  else
    if N→info > x then
      insertBST( x , N→left )
    else if N→info < x then
      insertBST( x , N→right )
    else
      output( 'duplicate' )
    endif
  endif
endprocedure
```

BST : Search Node

```
function findNode( in x : Infotype, N: BinTree )
         → Address
algorithm
  if N→info == x or N == NIL then
    return N
  else
    if N→info > x then
      findNode( x , N→left )
    else if N→info < x then
      findNode( x , N→right )
    endif
  endif
endfunction
```

Binary Search Tree (BST)

- Ordered / sorted binary tree
- BST Invariant
  - Left subtree has smaller elements
  - Right subtree has larger elements

BST : Delete Node

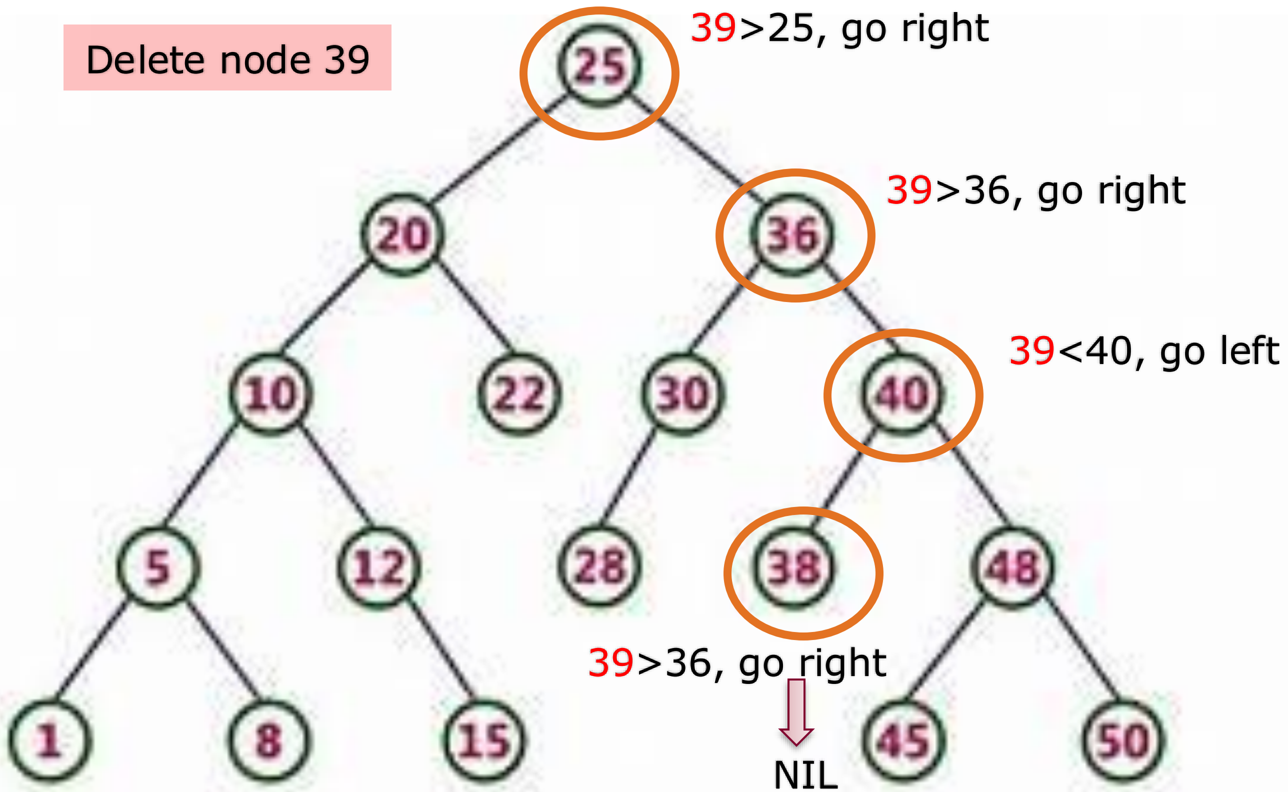
- Two step process of removing elements
  - **FIND** the node to remove (if exists)
  - **REPLACE** the node we want to remove with its successor (if any) to maintain the BST invariant

Delete Node: FIND

- In FIND phase, one of four cases below will happen:
  1. Hit a NIL node: the value does not exist
  2. Equal: found the value!
  3. Less: go to the left subtree
  4. Greater: go to the right subtree

1. Hit a NIL node: the value does not exist
2. Equal: found the value!
3. Less: go to the left subtree
4. Greater: go to the right subtree

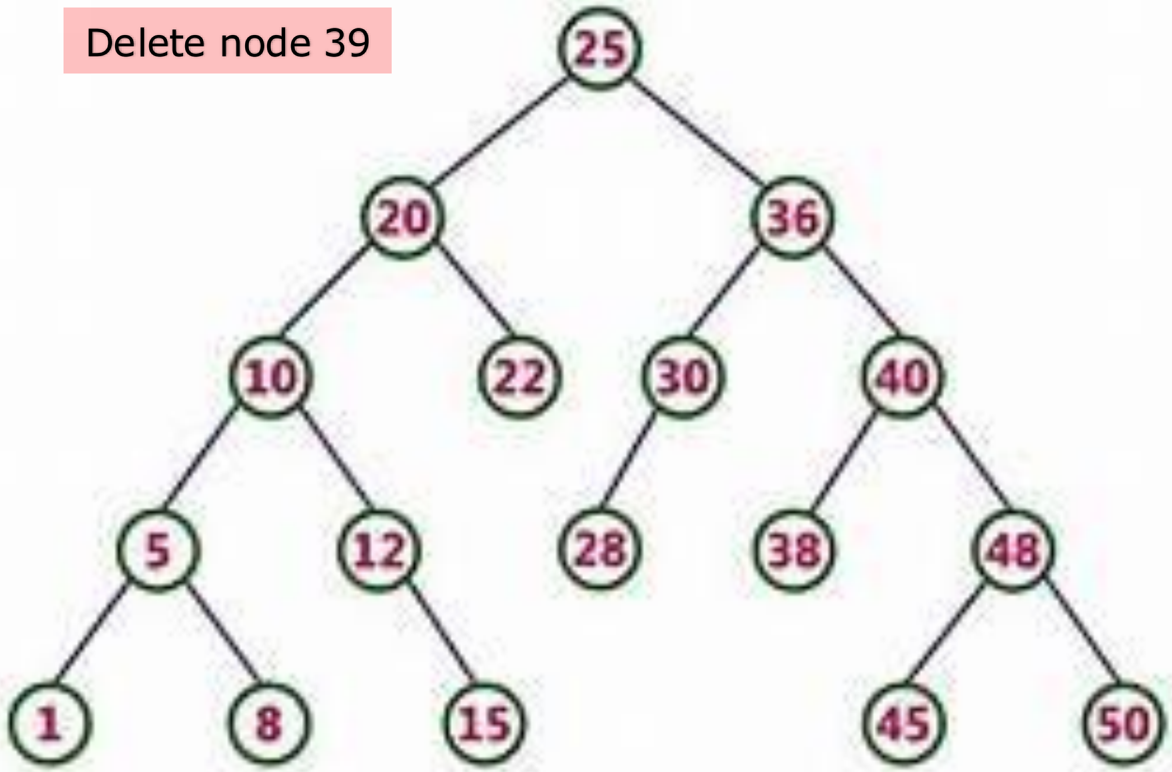
Delete Node: FIND



1. Hit a NIL node: the value does not exist
2. Equal: found the value!
3. Less: go to the left subtree
4. Greater: go to the right subtree

Delete Node: FIND

Delete node 39

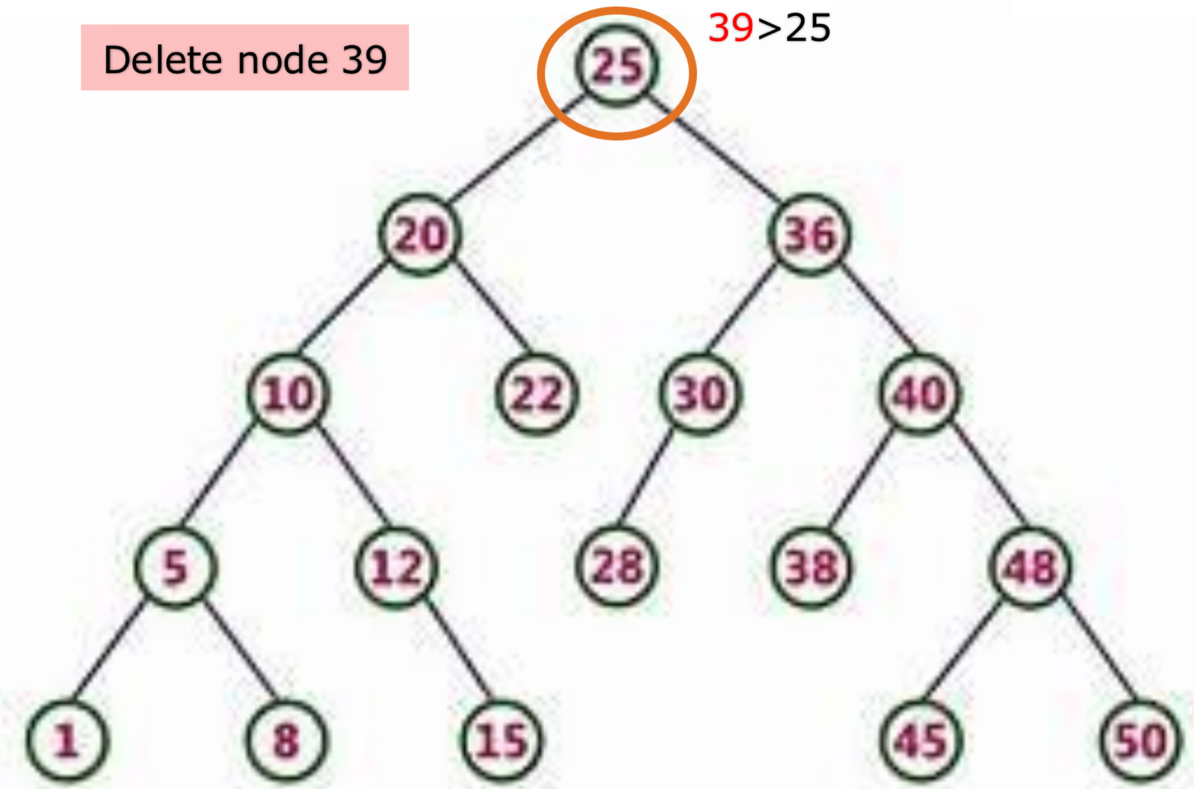


1. Hit a NIL node: the value does not exist
2. Equal: found the value!
3. Less: go to the left subtree
4. Greater: go to the right subtree

Delete Node: FIND

Delete node 39

39>25

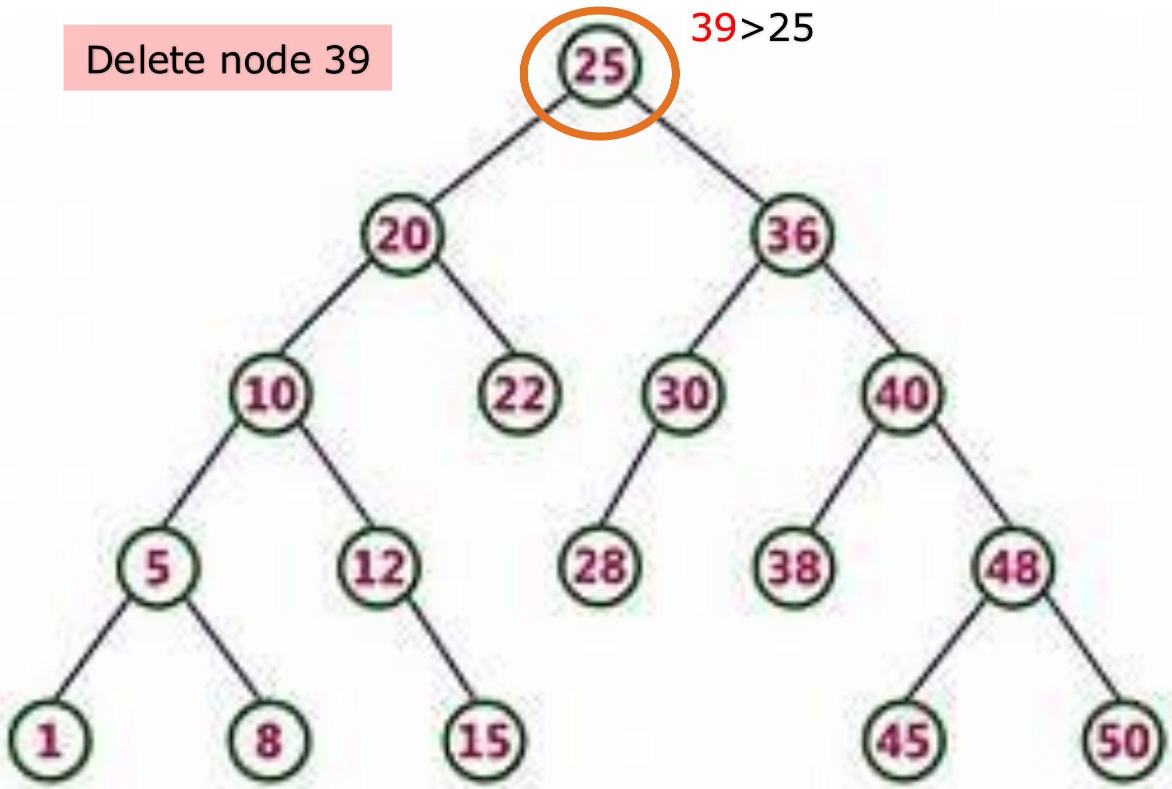


1. Hit a NIL node: the value does not exist
2. Equal: found the value!
3. Less: go to the left subtree
4. Greater: go to the right subtree

Delete Node: FIND

Delete node 39

39>25

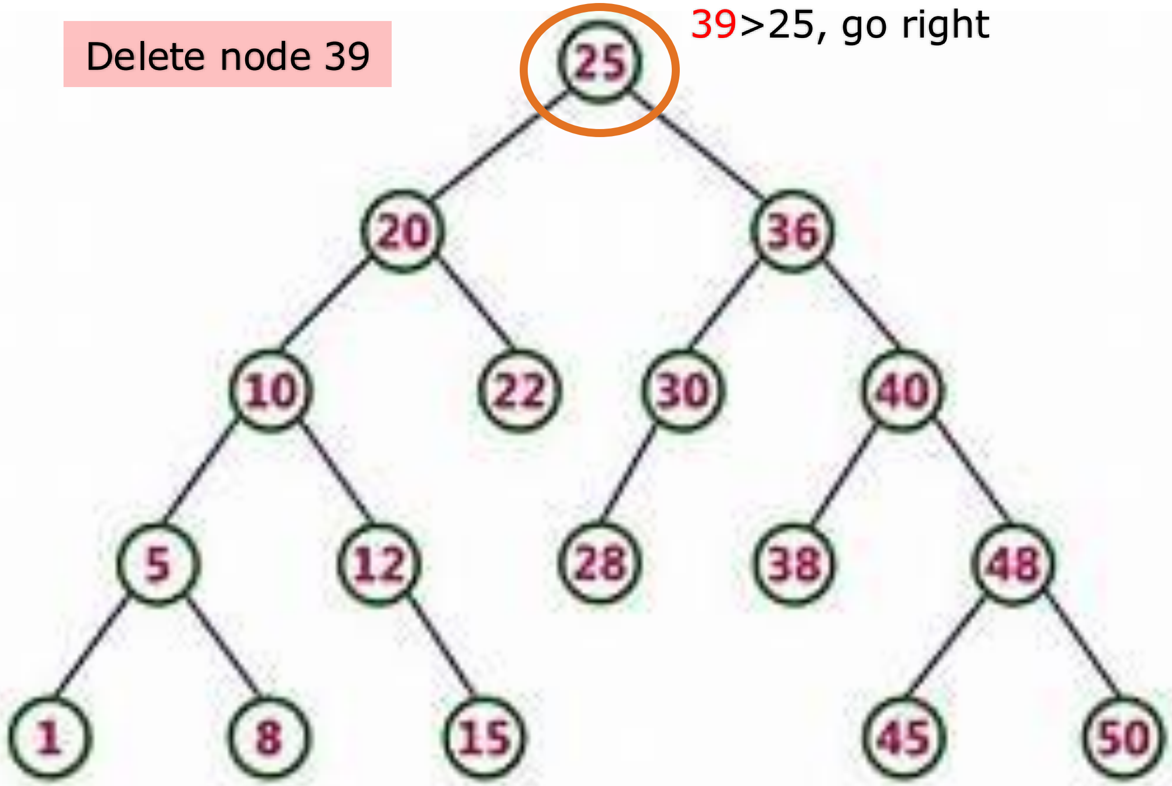


1. Hit a NIL node: the value does not exist
2. Equal: found the value!
3. Less: go to the left subtree
4. Greater: go to the right subtree

Delete Node: FIND

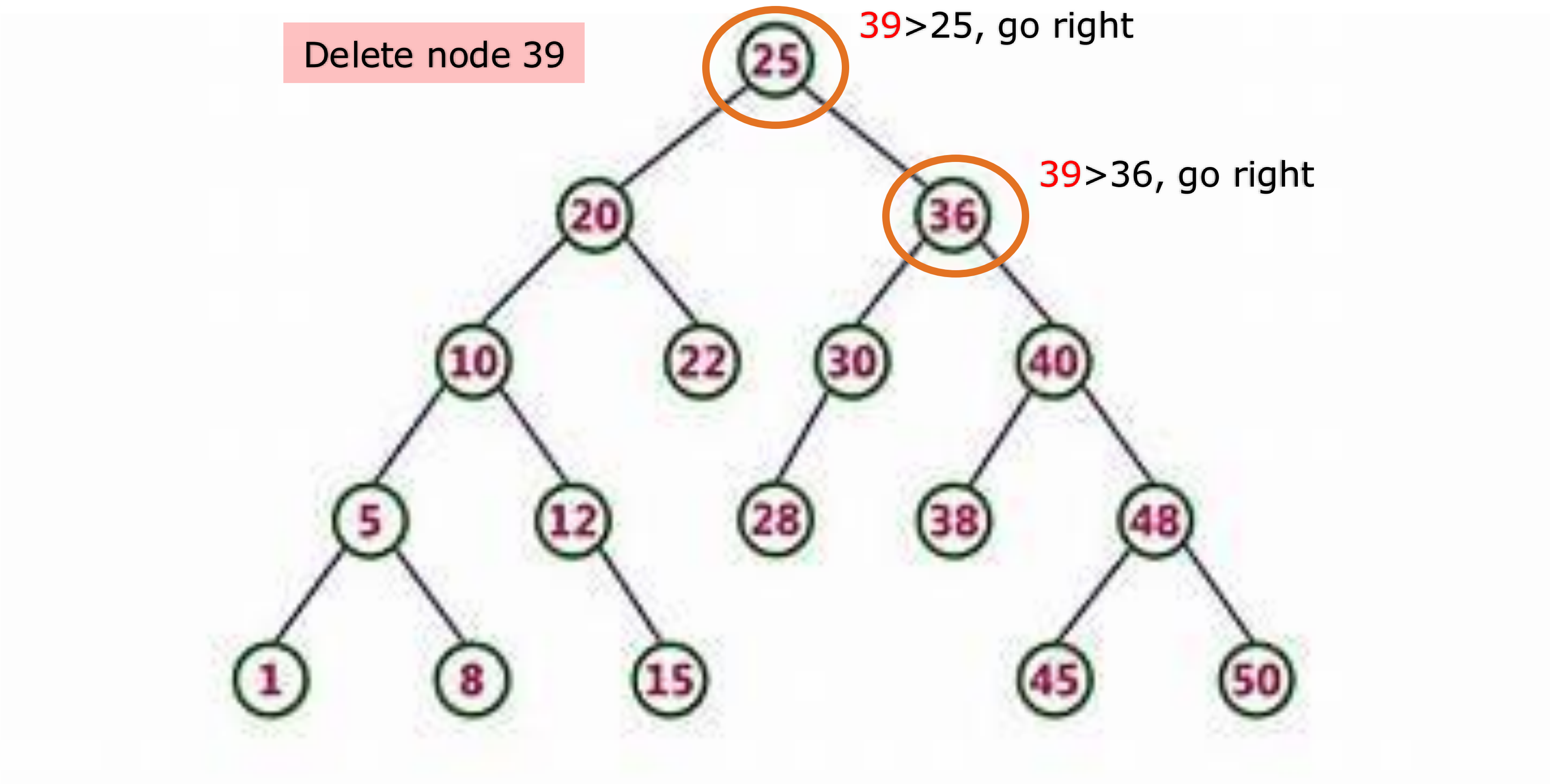
Delete node 39

39>25, go right



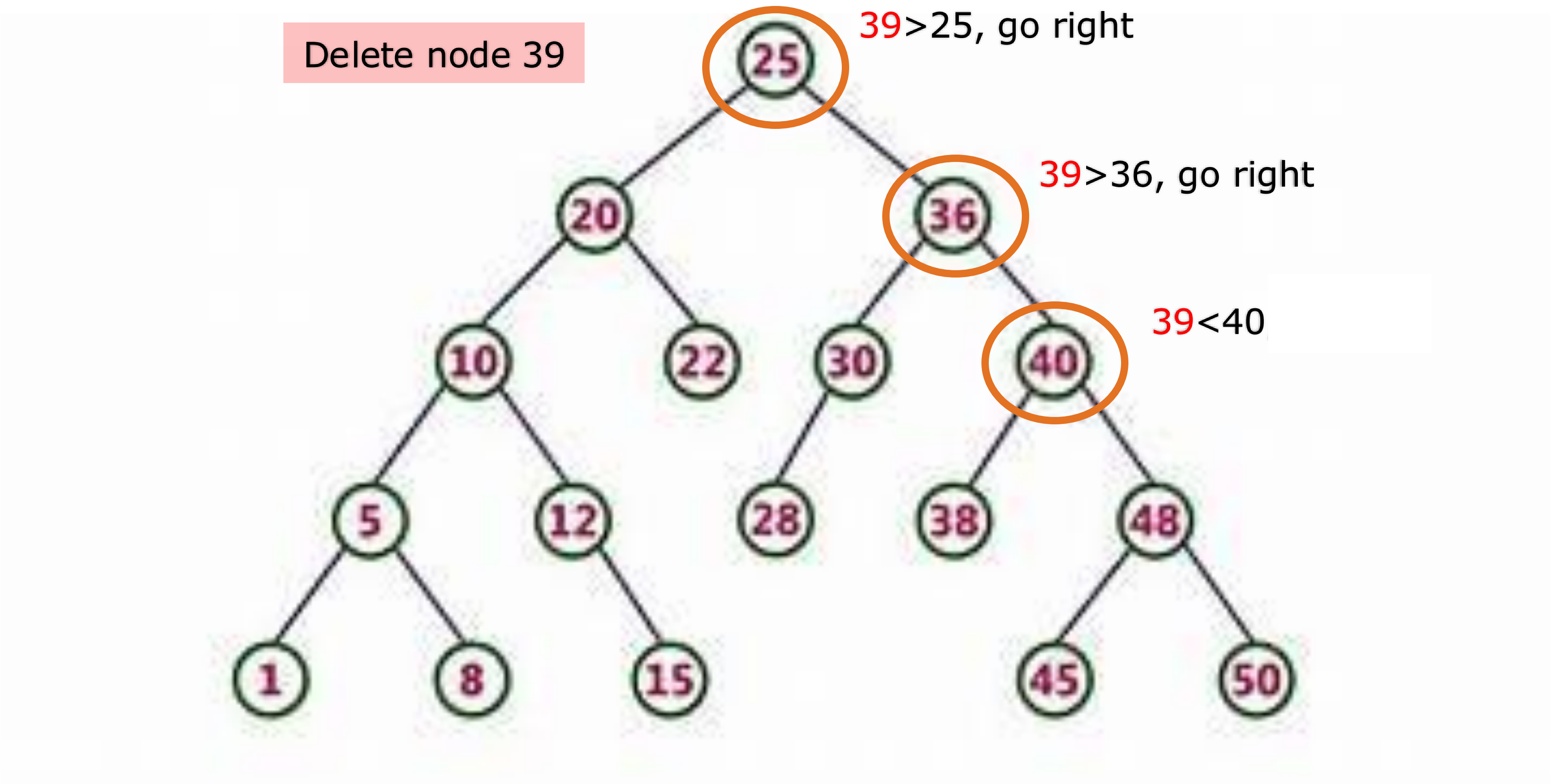
1. Hit a NIL node: the value does not exist
2. Equal: found the value!
3. Less: go to the left subtree
4. Greater: go to the right subtree

Delete Node: FIND



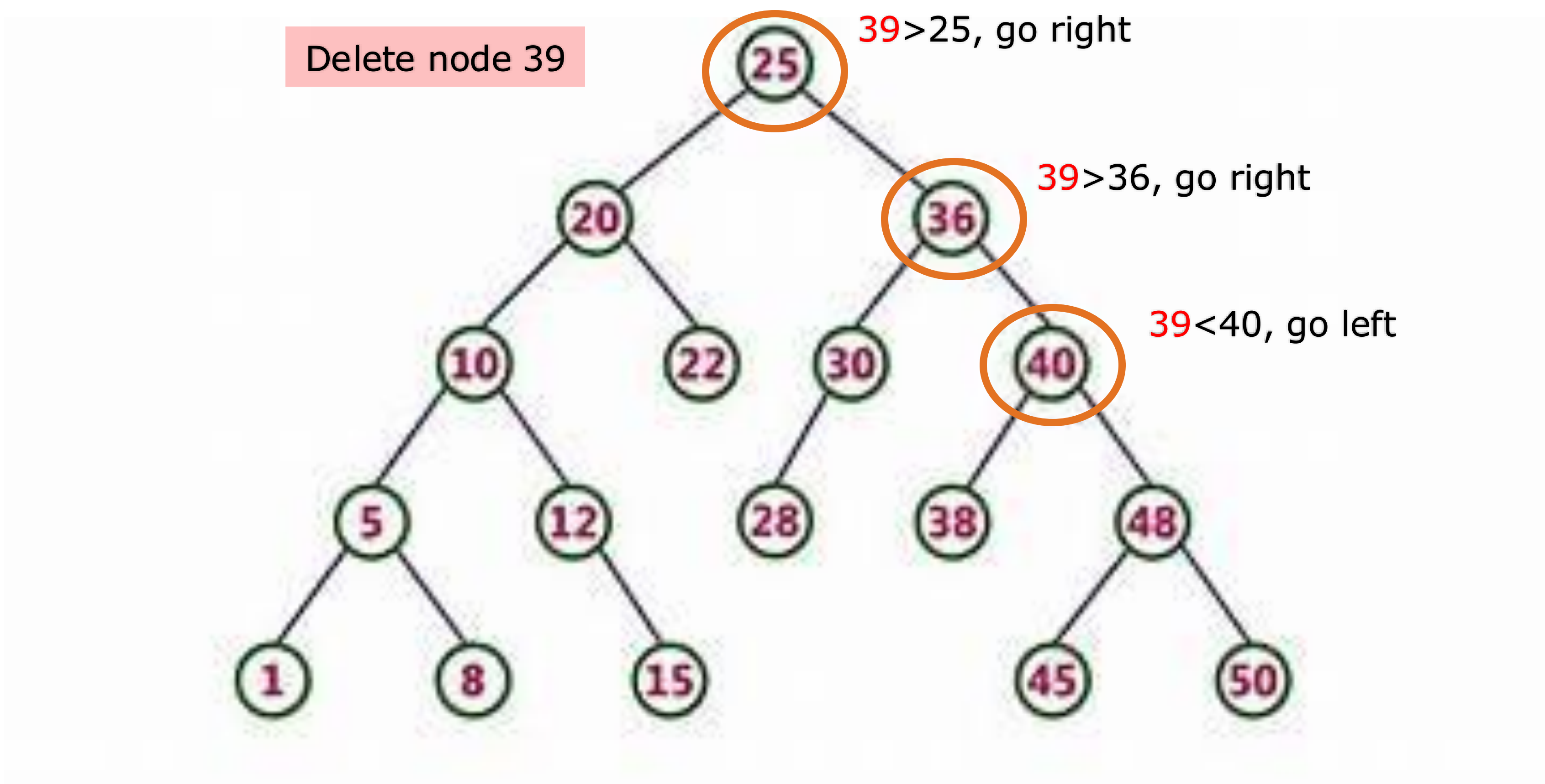
1. Hit a NIL node: the value does not exist
2. Equal: found the value!
3. Less: go to the left subtree
4. Greater: go to the right subtree

Delete Node: FIND



1. Hit a NIL node: the value does not exist
2. Equal: found the value!
3. Less: go to the left subtree
4. Greater: go to the right subtree

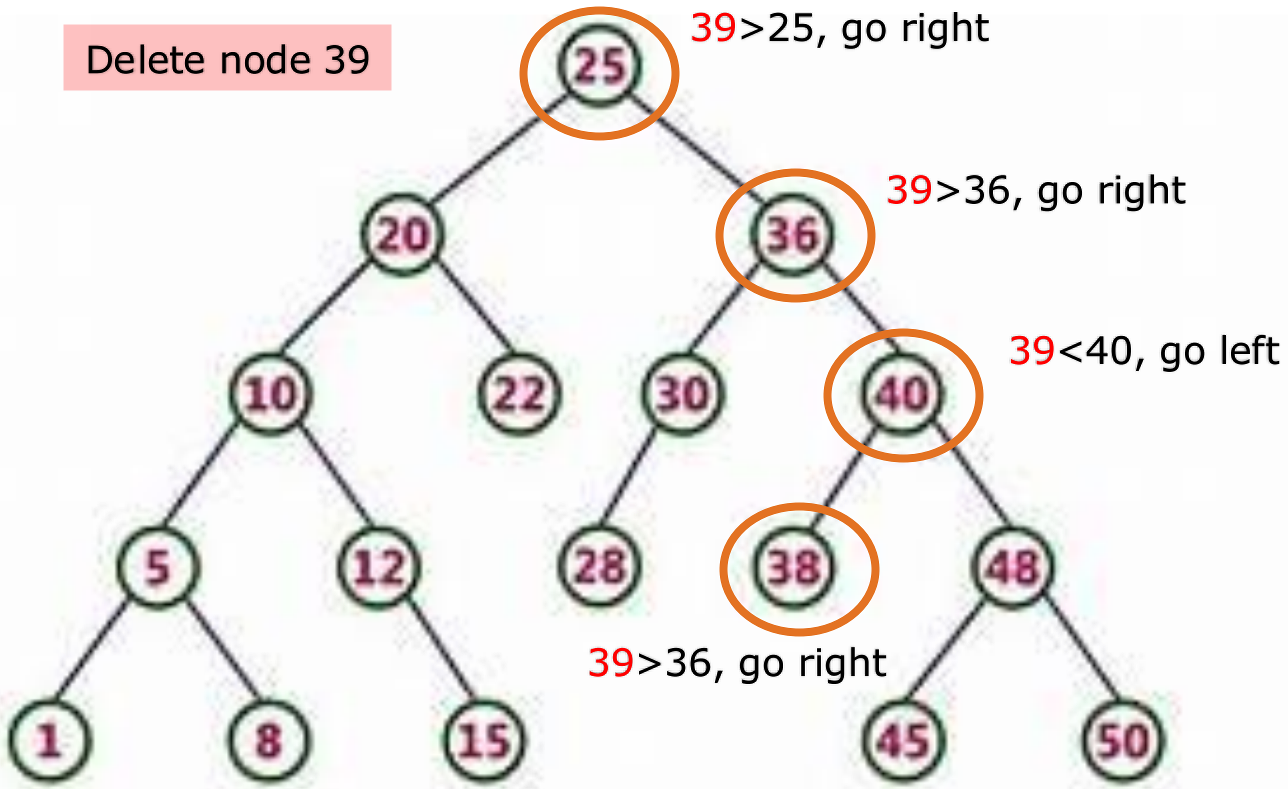
Delete Node: FIND



1. Hit a NIL node: the value does not exist
2. Equal: found the value!
3. Less: go to the left subtree
4. Greater: go to the right subtree

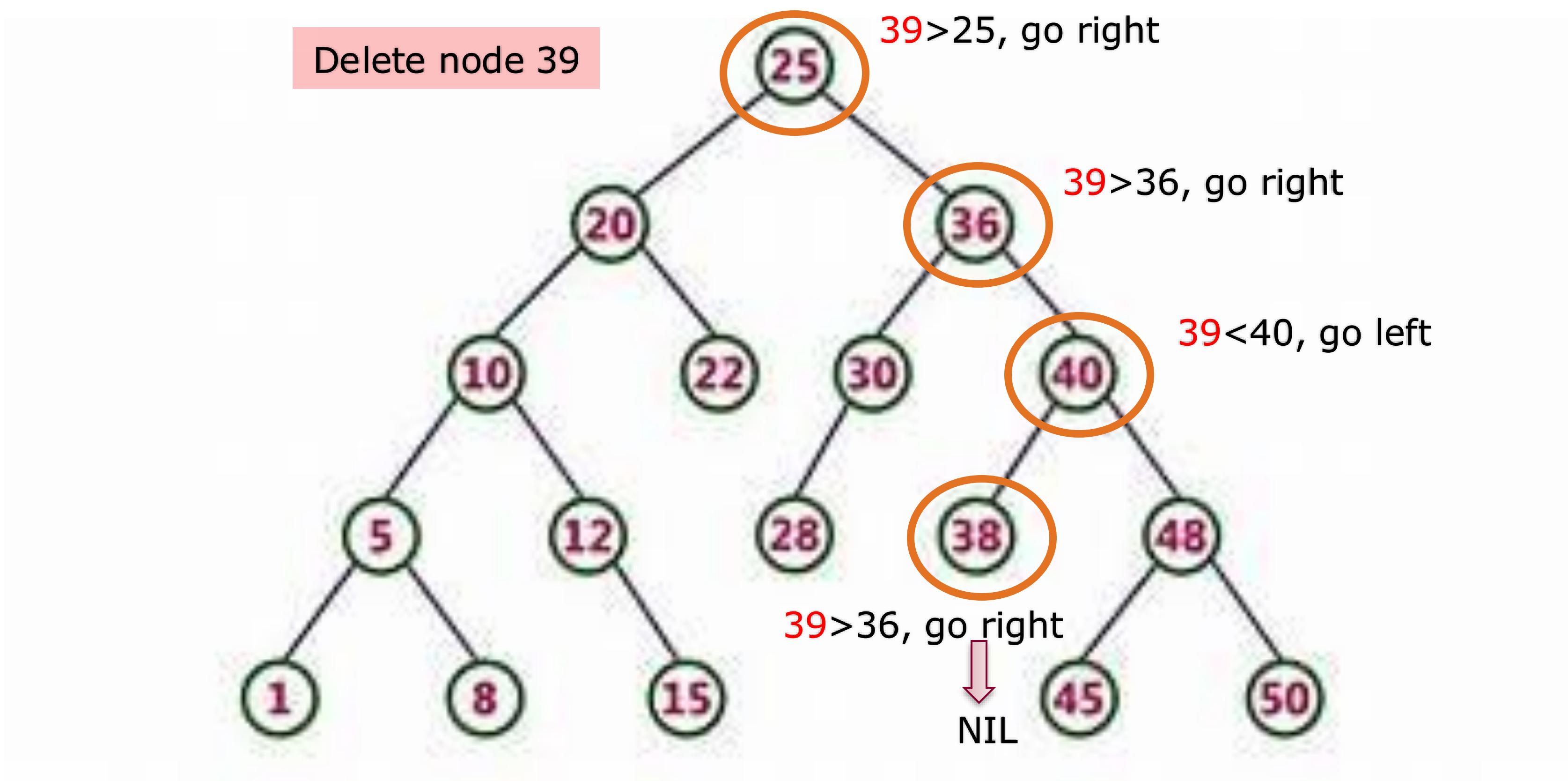
Delete Node: FIND

Delete node 39



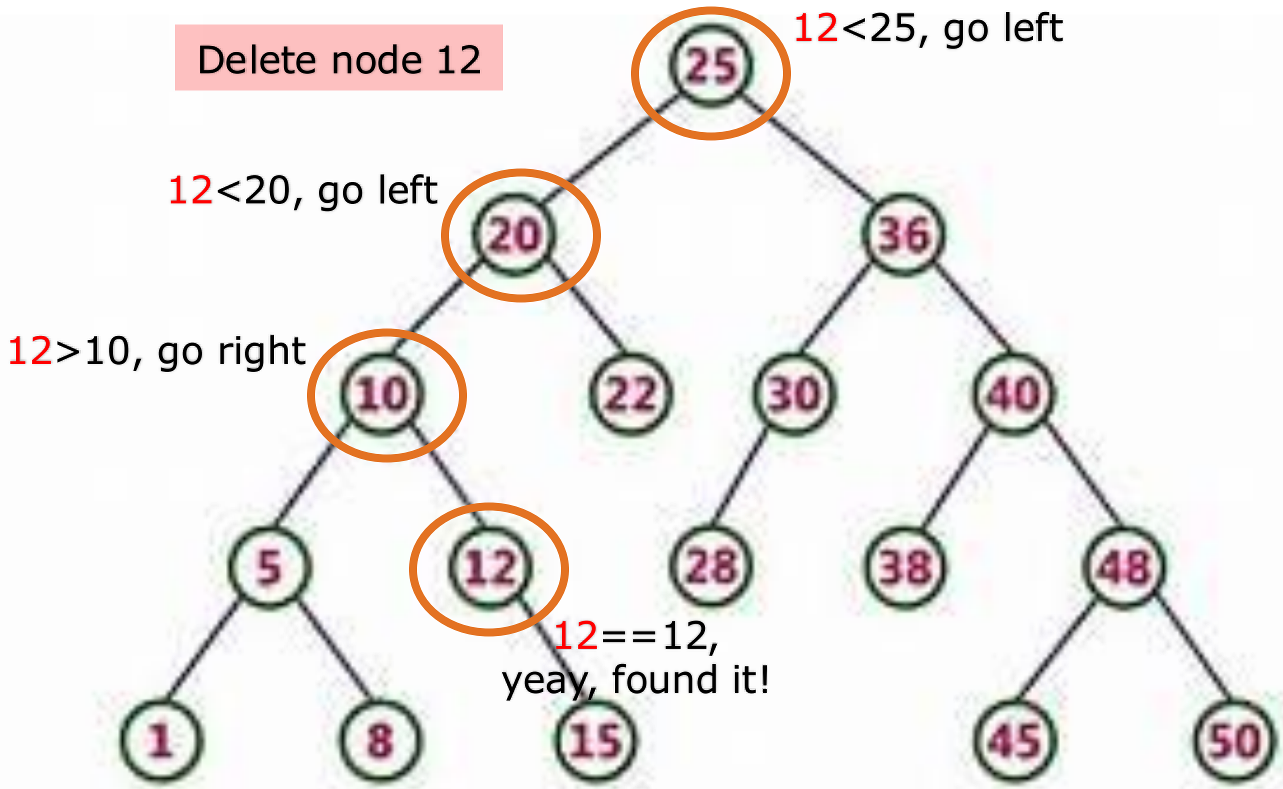
1. Hit a NIL node: the value does not exist
2. Equal: found the value!
3. Less: go to the left subtree
4. Greater: go to the right subtree

Delete Node: FIND



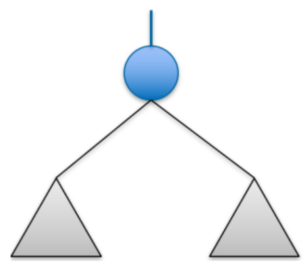
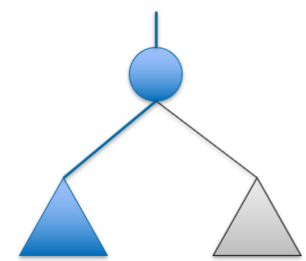
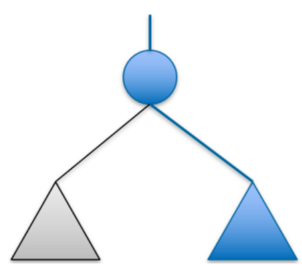
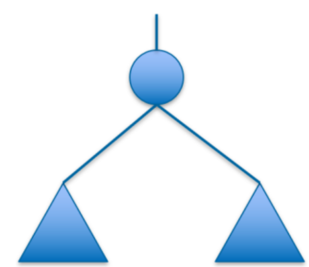
1. Hit a NIL node: the value does not exist
2. Equal: found the value!
3. Less: go to the left subtree
4. Greater: go to the right subtree

Delete Node: FIND



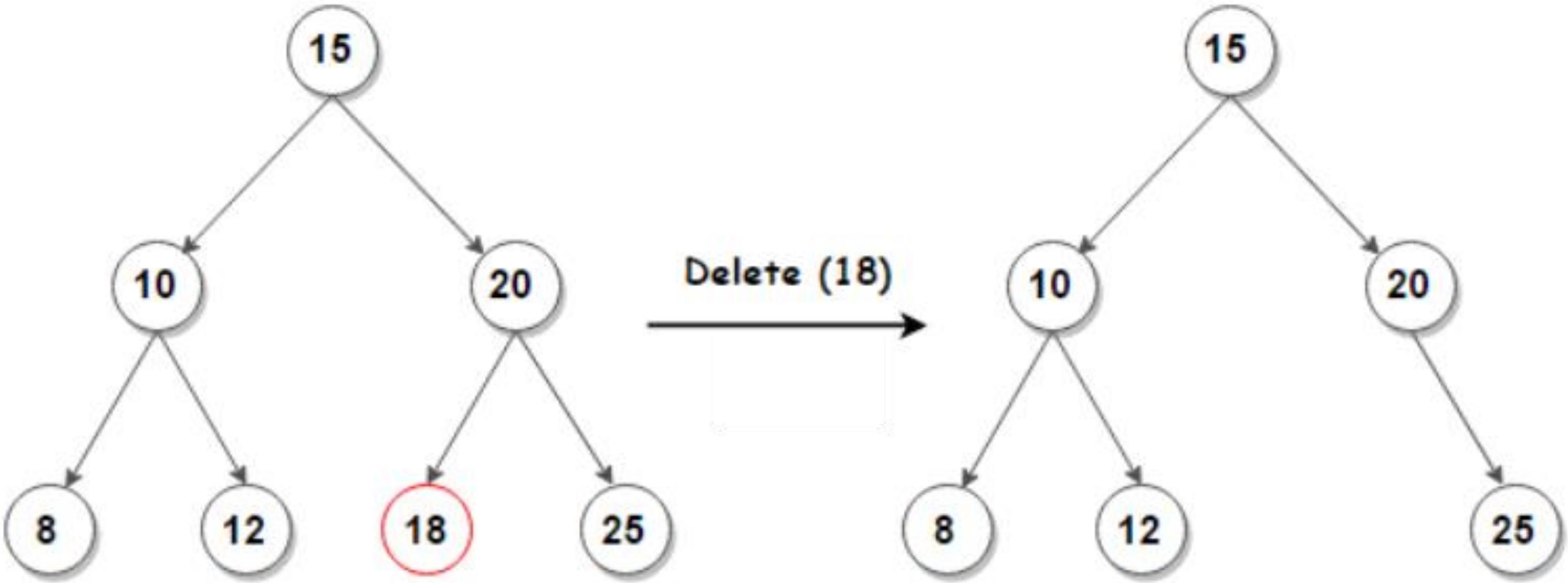
Delete Node: REPLACE

- In REPLACE phase, one of four cases below will happen; node to remove:

a leaf Case 1 	has left subtree but no right subtree Case 2 
has right subtree but no left subtree Case 3 	has both left and right subtree Case 4 

Delete Node: REPLACE

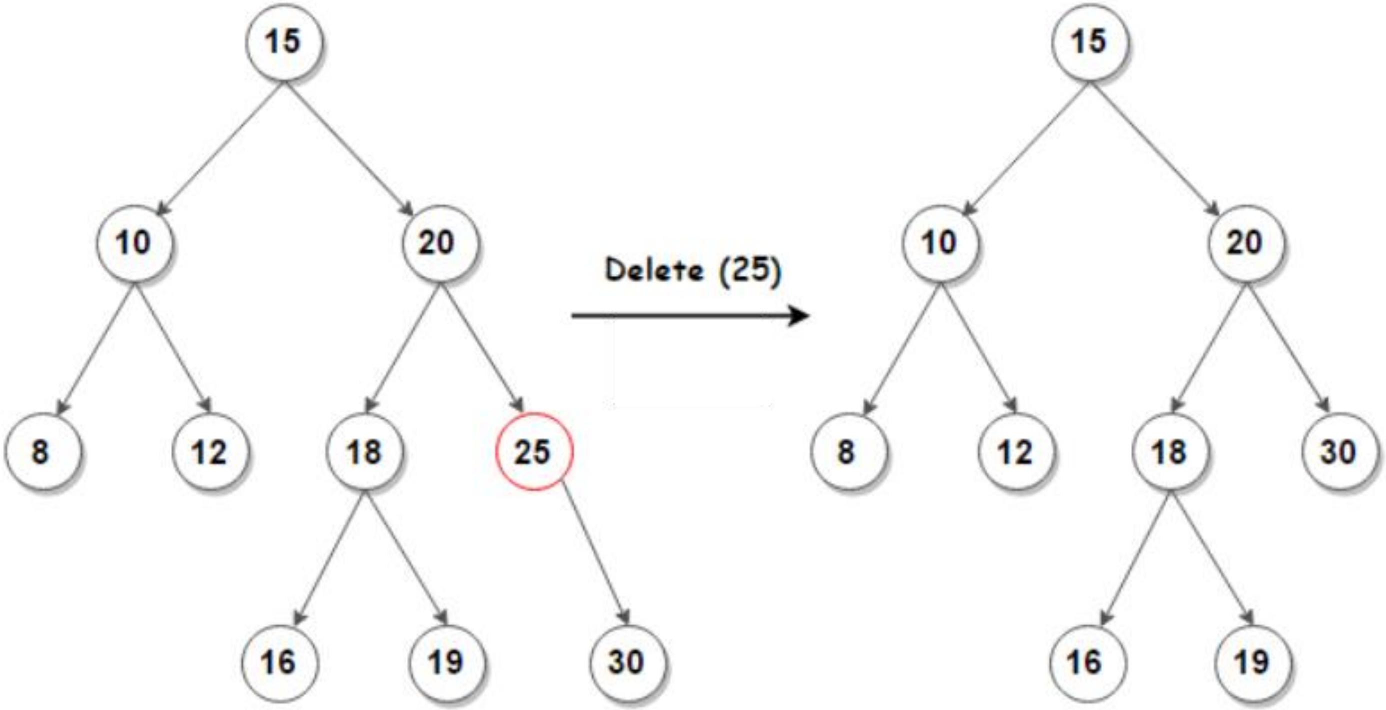
- **Case 1:** Deleting a node with no children: remove the node from the tree.



Ref: techiedelight.com

Delete Node: REPLACE

- **Case 2 & 3:** Deleting a node with one child: remove the node and replace it with its child.



Ref: techiedelight.com

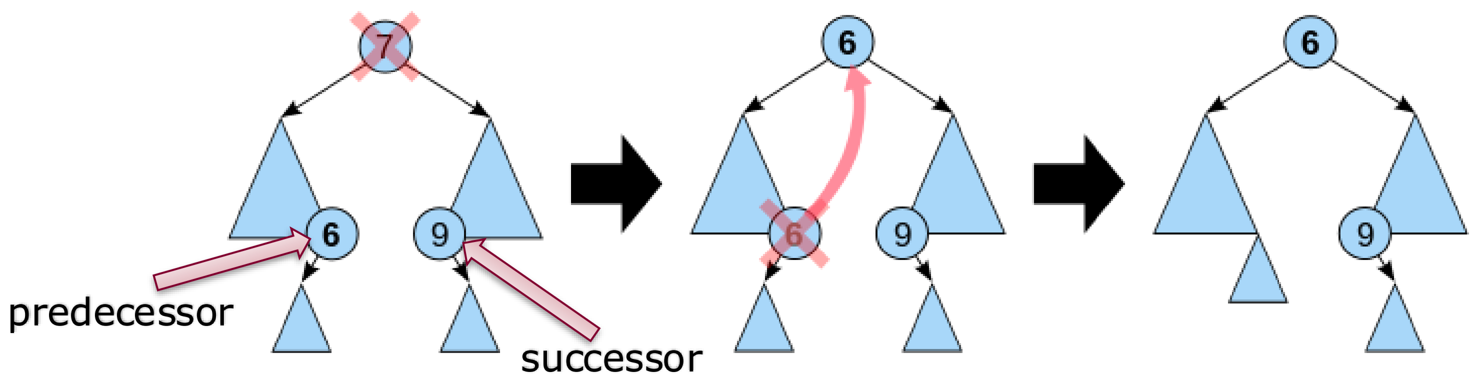
Delete Node: REPLACE

- **Case 4:** Deleting a node with two children:
  - call the node to be deleted X.
  - Do not delete X. Instead, choose either its inorder successor node or its inorder predecessor node, R.
  - Copy the value of R to X, then recursively call delete on R until reaching case 1, 2, or 3.

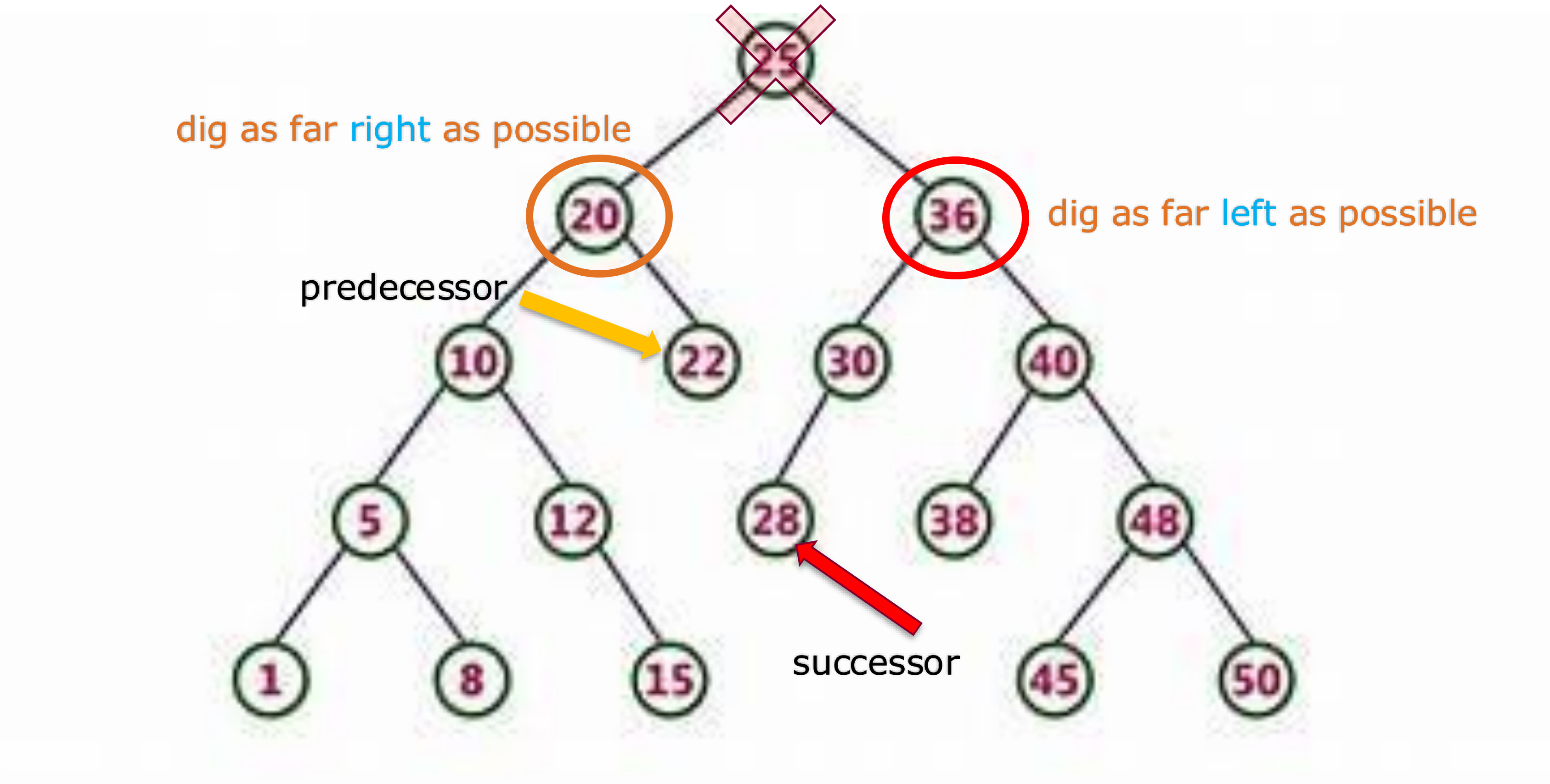
Ref: techiedelight.com

Inorder Successor vs. Predecessor Node

- Which node should replace the node we wish to remove?



BST : Delete Node



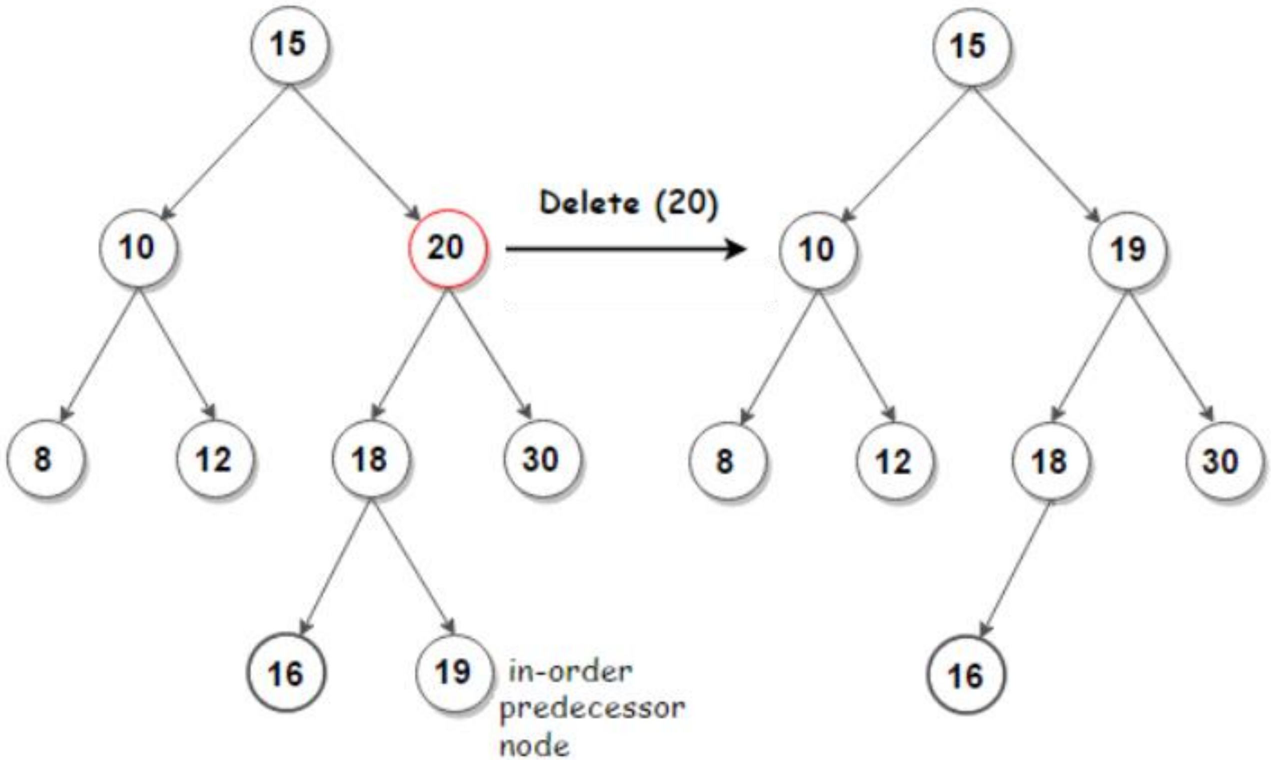
Delete Node: REPLACE

- **Case 4:** Deleting a node with two children:
  - If we choose the **inorder predecessor** of a node, as **the left subtree is not NIL** (our present case is a node with 2 children), then its inorder predecessor is a node with **the greatest value in its left subtree**, which will have at a maximum of 1 left subtree, so deleting it would fall in case 1 or case 2.
  - If we choose the **inorder successor** of a node, as **the right subtree is not NIL**, then its inorder successor is a node with **the least value in its left subtree**, which will have at a maximum of 1 right subtree, so deleting it would fall in case 1 or case 3.

Ref: techiedelight.com

Delete Node: REPLACE

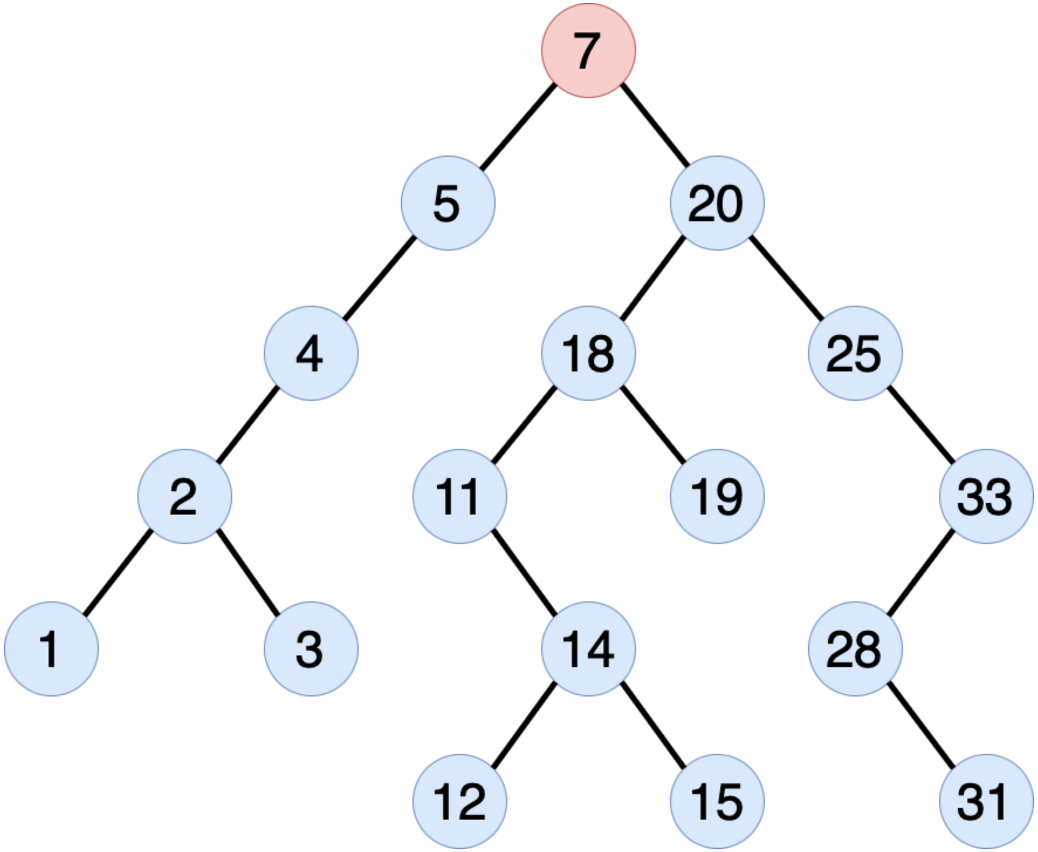
- **Case 4:** Deleting a node with two children:



Ref: techiedelight.com

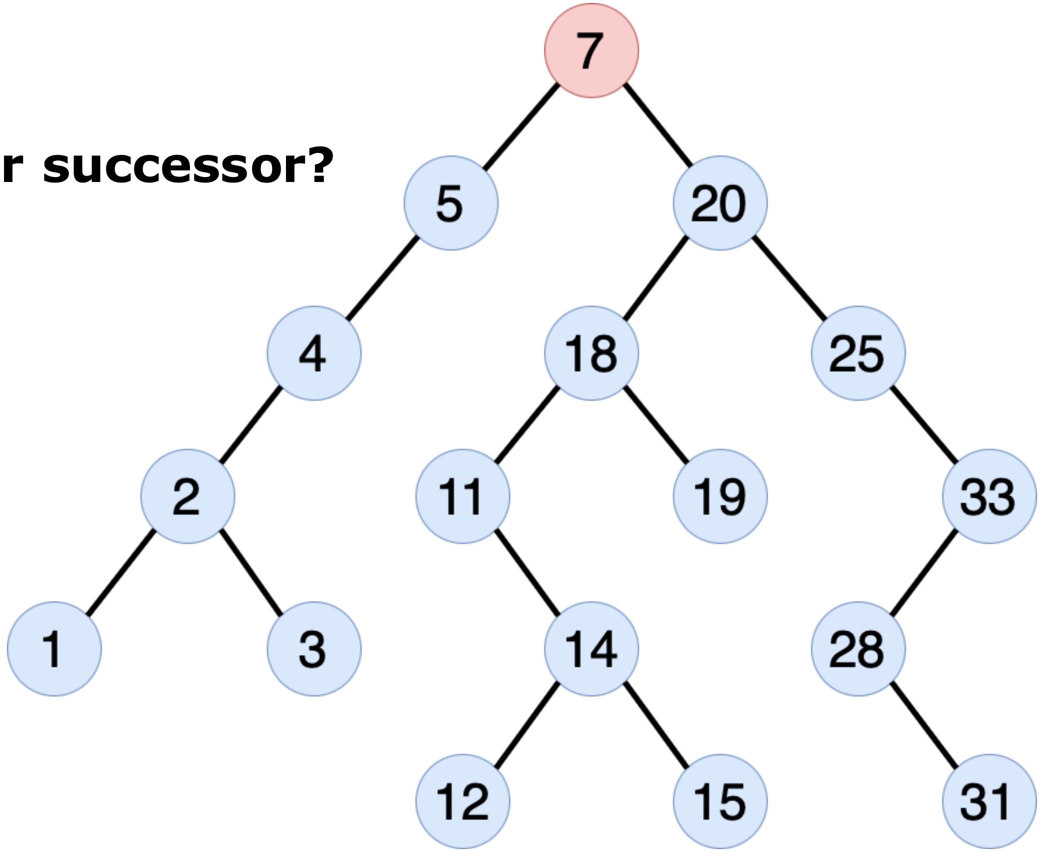
More Example
Case 4

Let’s remove 7



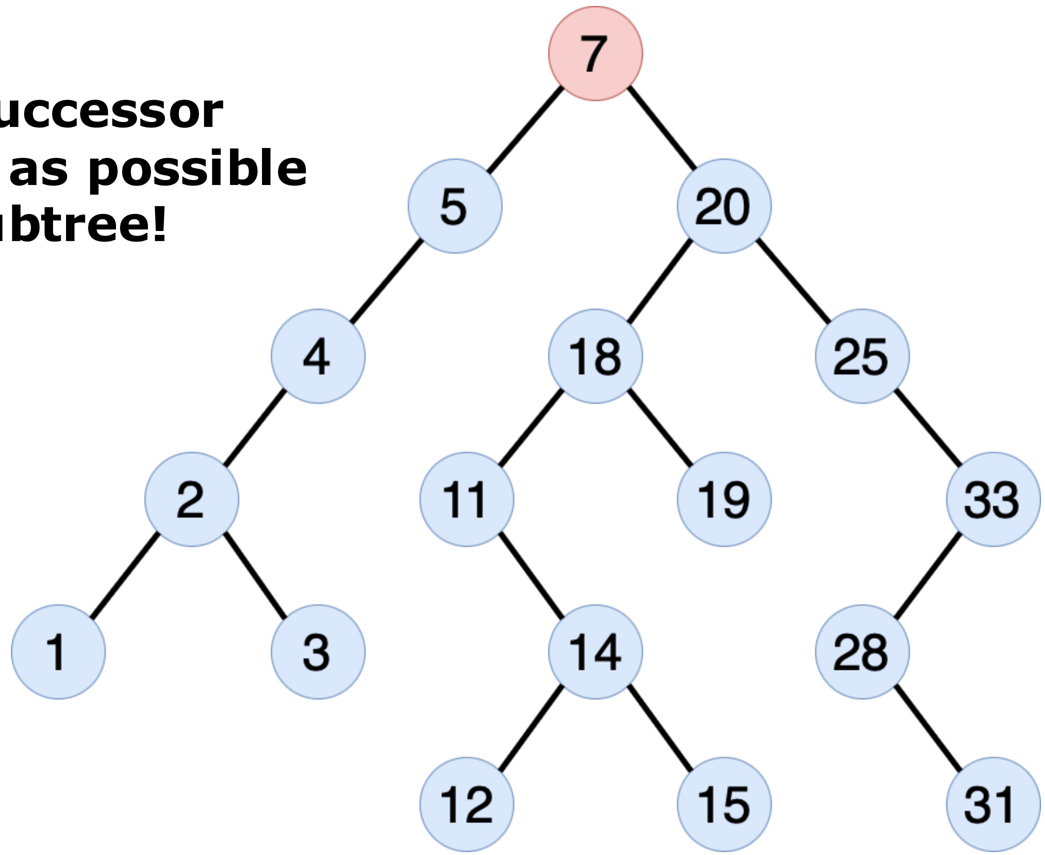
Let’s remove 7

inorder
predecessor or successor?

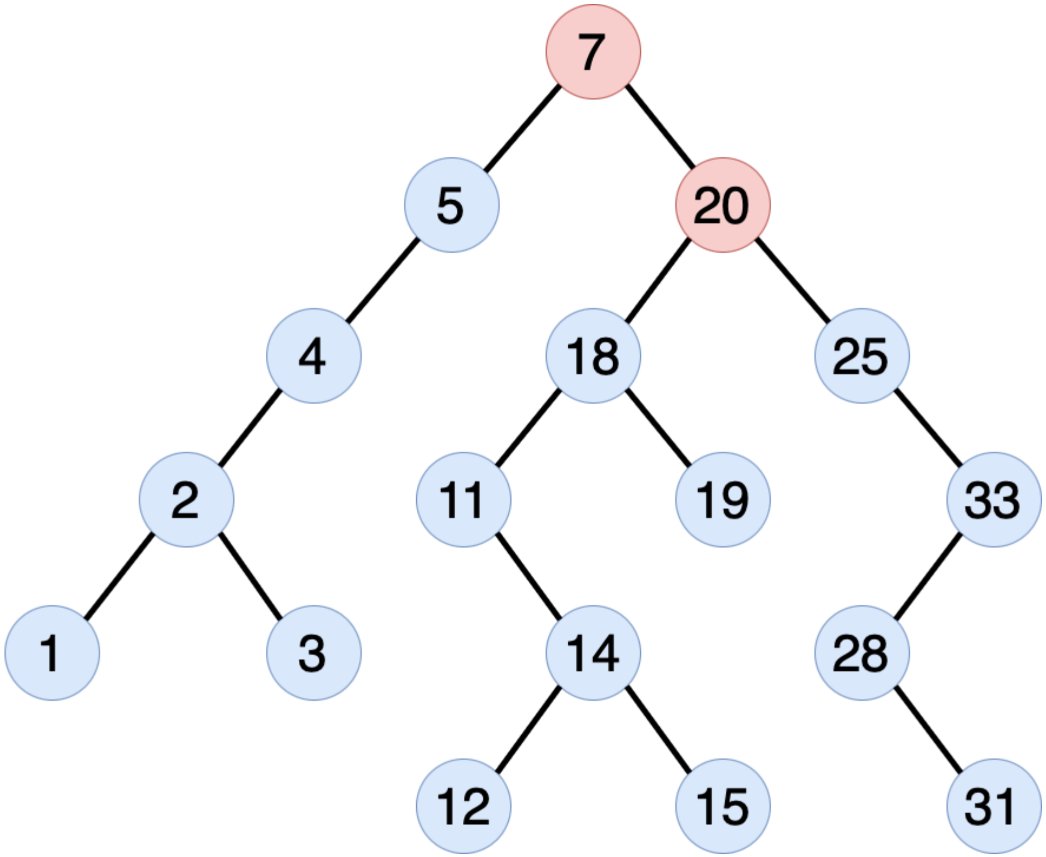


Let’s remove 7

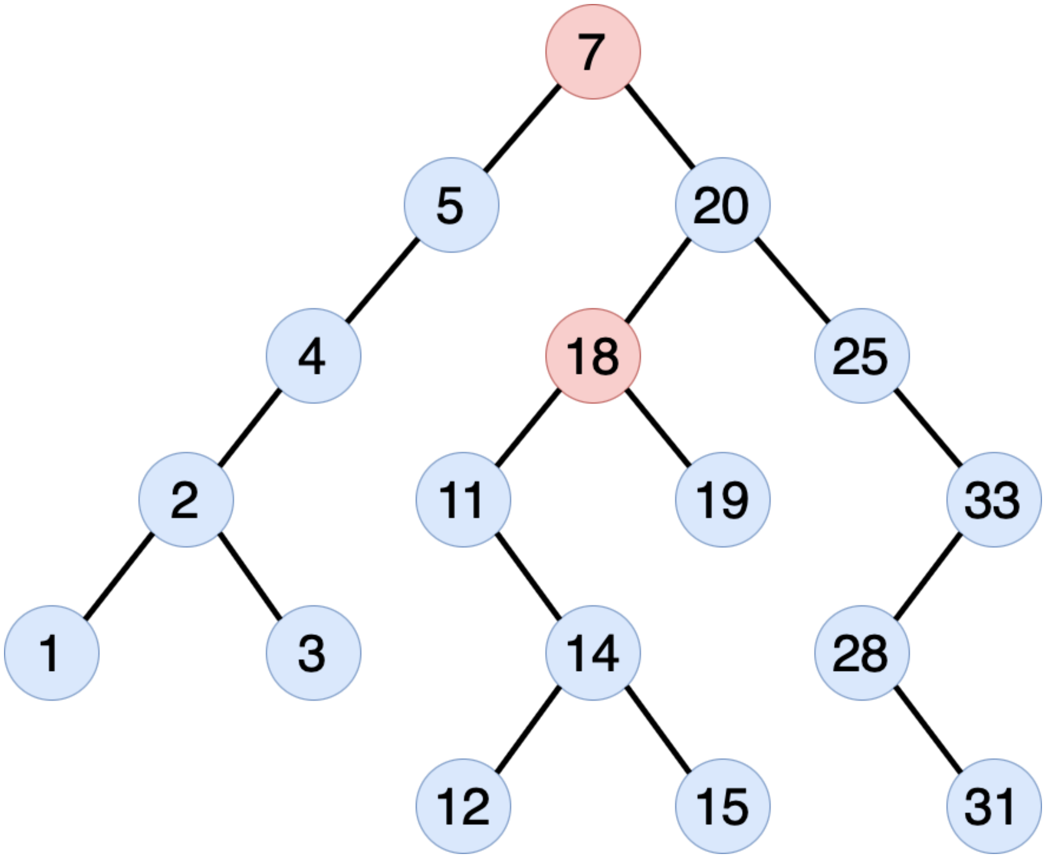
Let’s do the successor
Dig as far left as possible
in the right subtree!



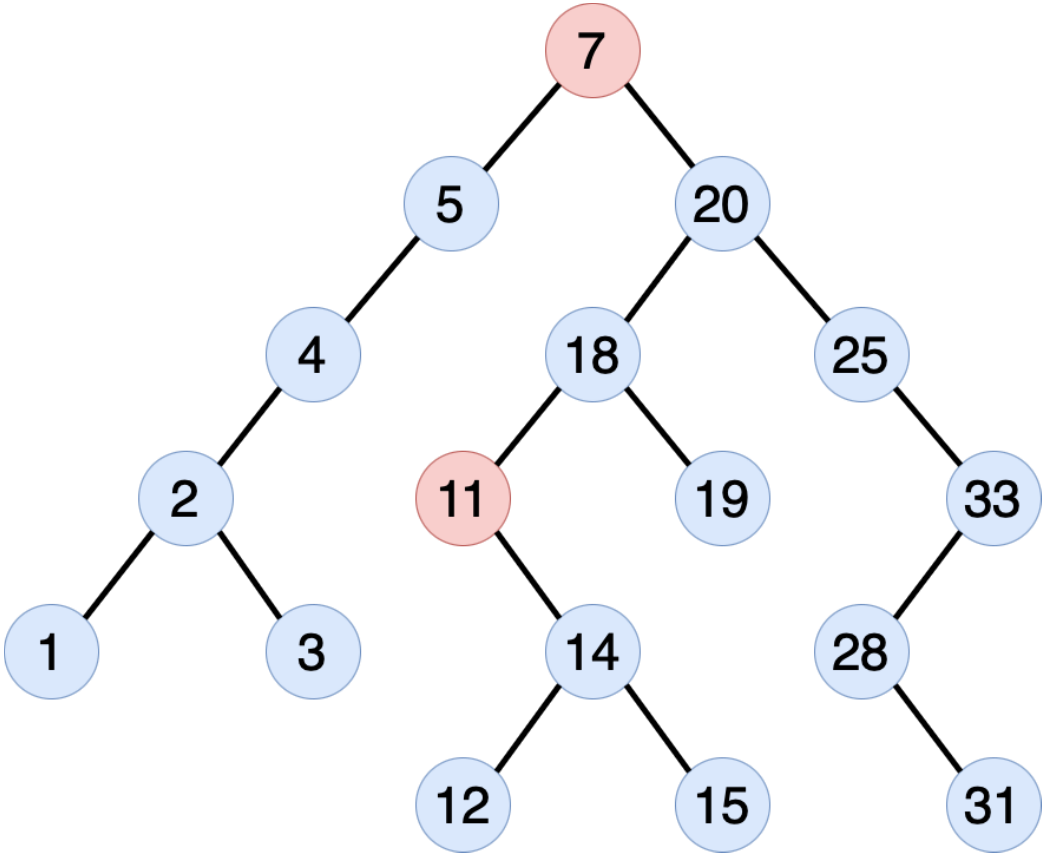
Let’s remove 7



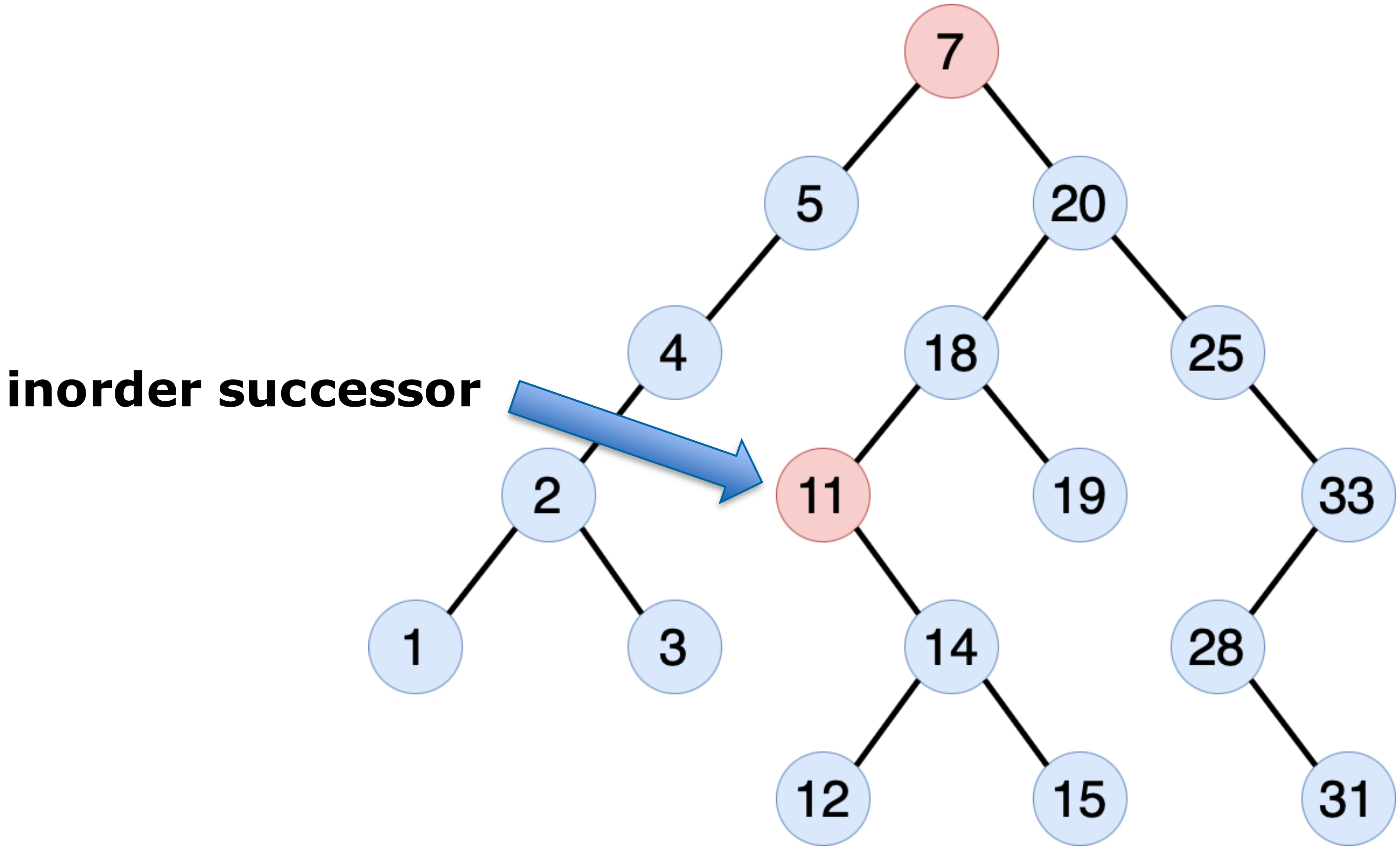
Let’s remove 7



Let’s remove 7

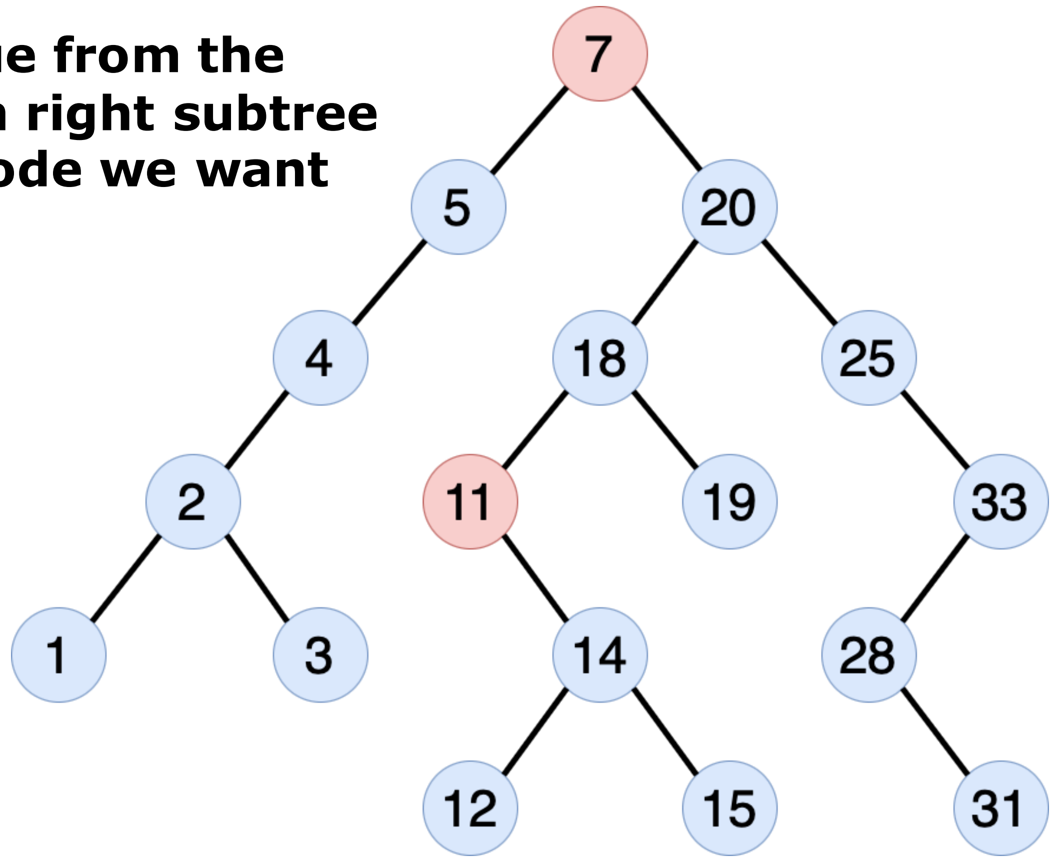


Let's remove 7



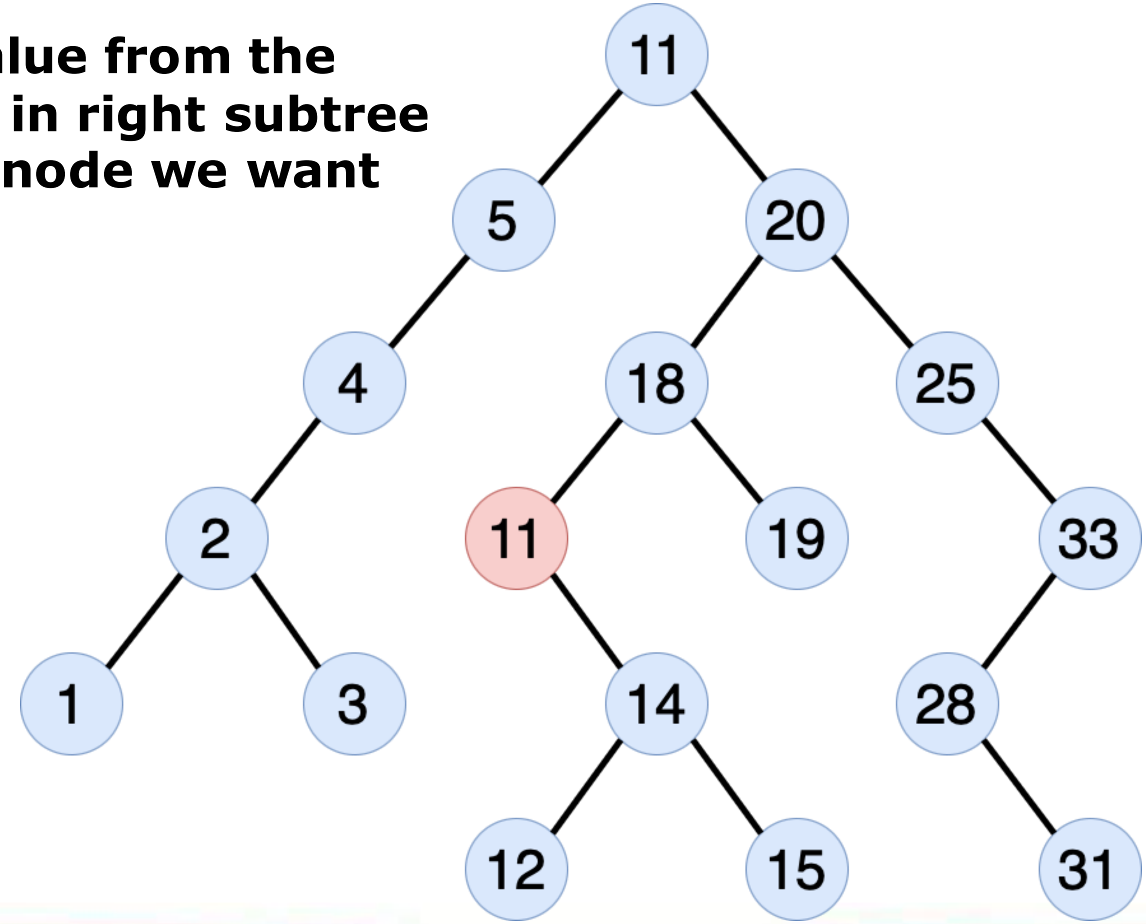
Let’s remove 7

Copy the value from the node found in right subtree (11) to the node we want to remove



Let’s remove 7

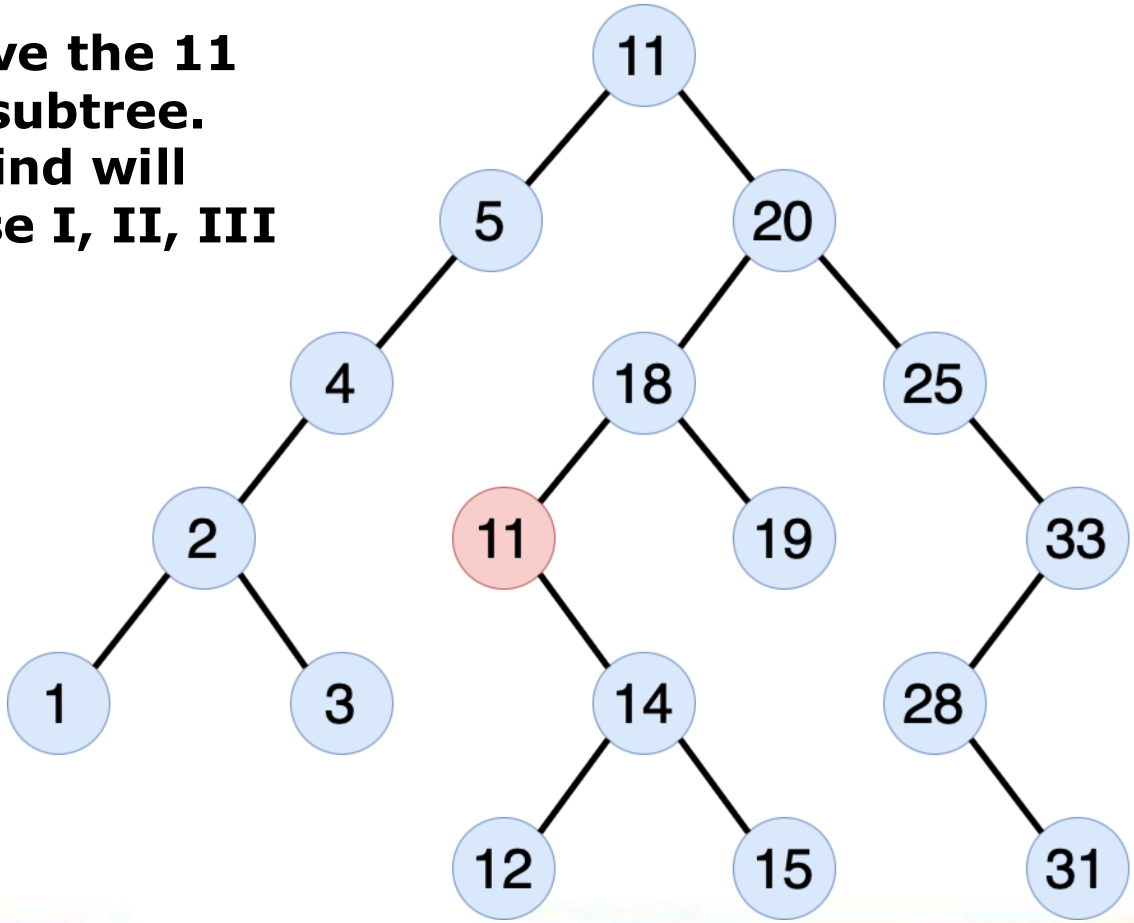
Copy the value from the node found in right subtree (11) to the node we want to remove



Copy the value from the node found in right subtree (11) to the node we want to remove

Let’s remove 7

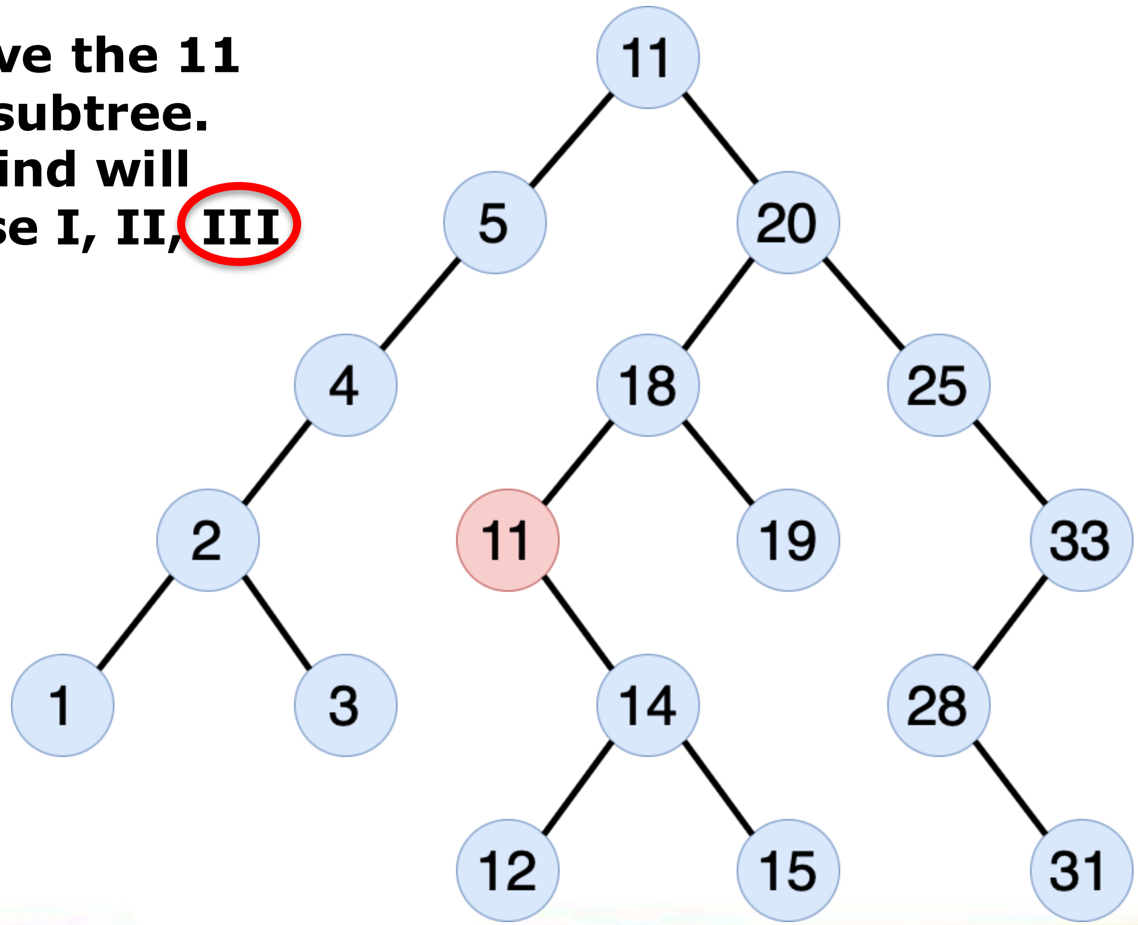
Now we have to remove the 11 we found in the right subtree. Luckily, the node we find will always be either a Case I, II, III removal



Copy the value from the node found in right subtree (11) to the node we want to remove

Let’s remove 7

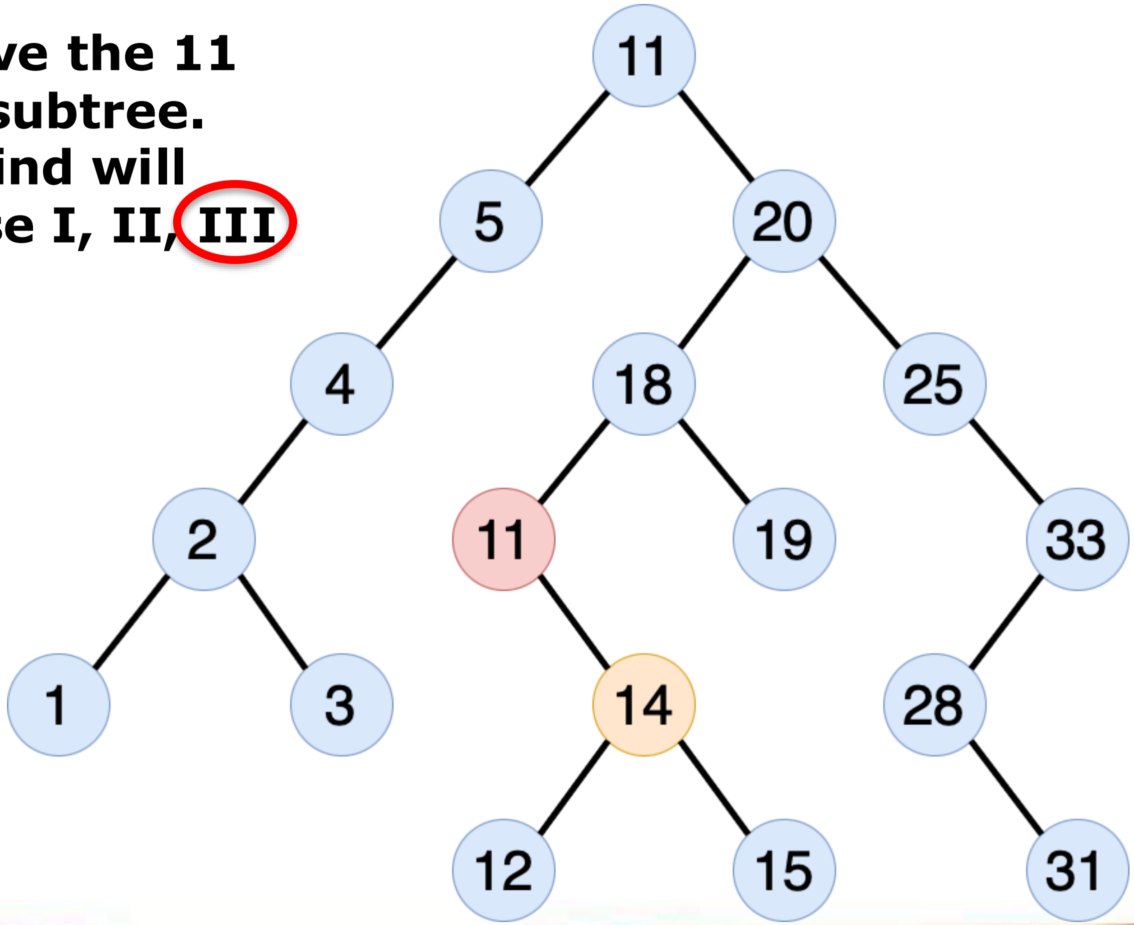
Now we have to remove the 11 we found in the right subtree. Luckily, the node we find will always be either a Case I, II, III removal



Copy the value from the node found in right subtree (11) to the node we want to remove

Let's remove 7

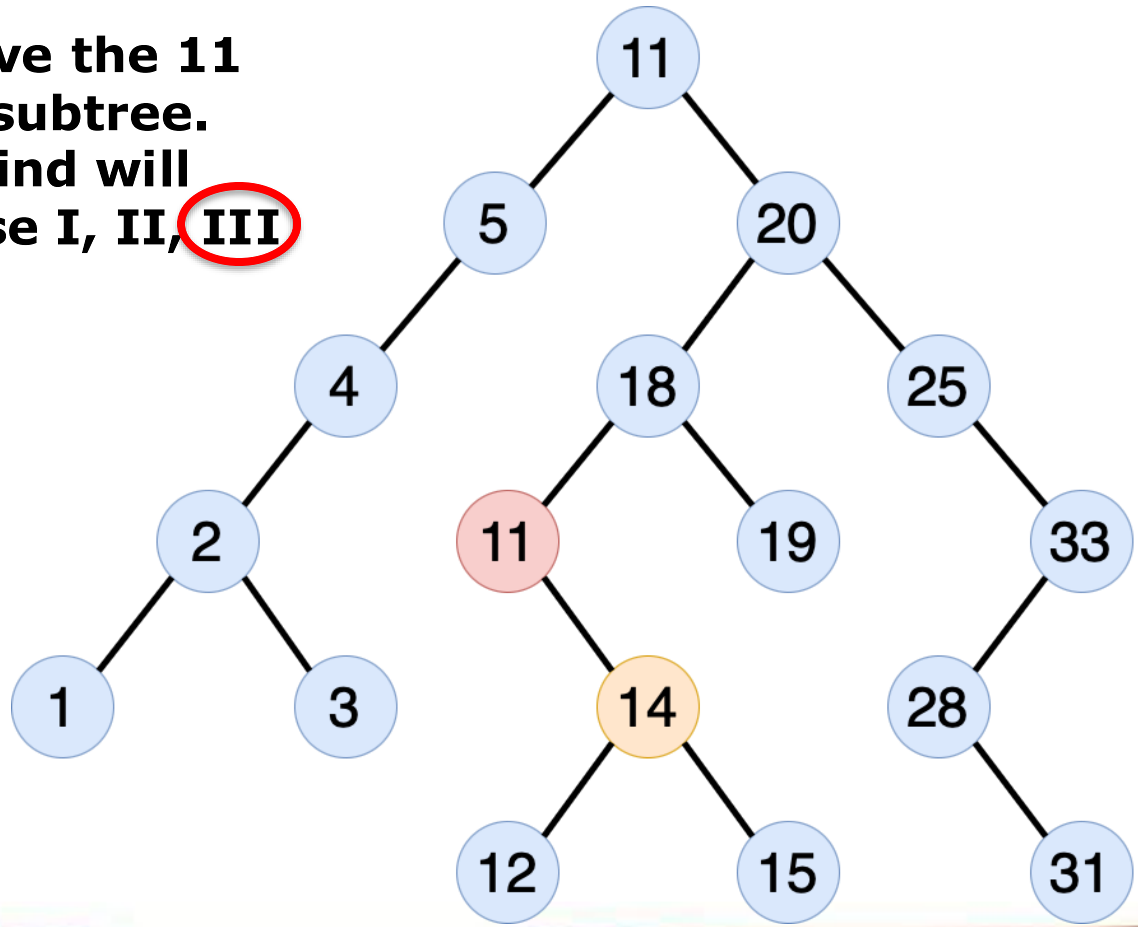
Now we have to remove the 11 we found in the right subtree. Luckily, the node we find will always be either a Case I, II, III removal



Copy the value from the node found in right subtree (11) to the node we want to remove

Let’s remove 7

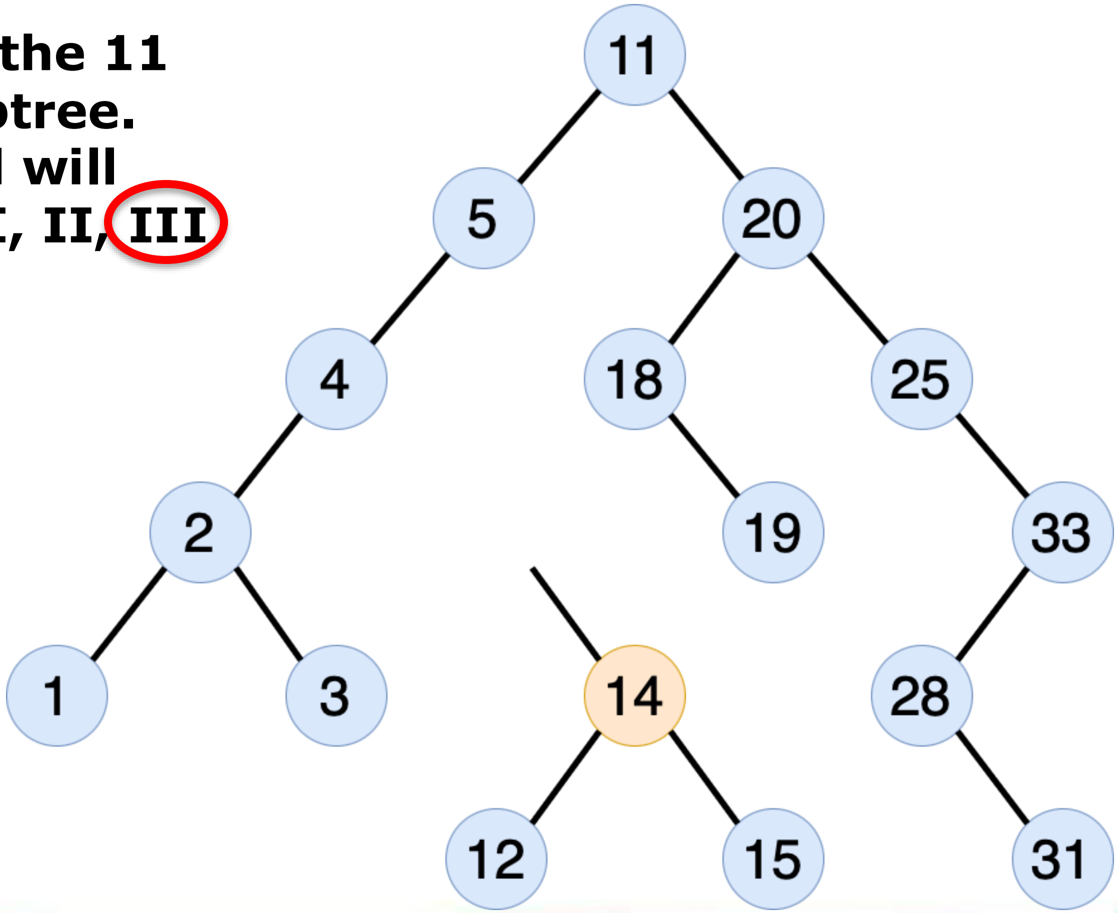
Now we have to remove the 11 we found in the right subtree. Luckily, the node we find will always be either a Case I, II, III removal



Copy the value from the node found in right subtree (11) to the node we want to remove

Let’s remove 7

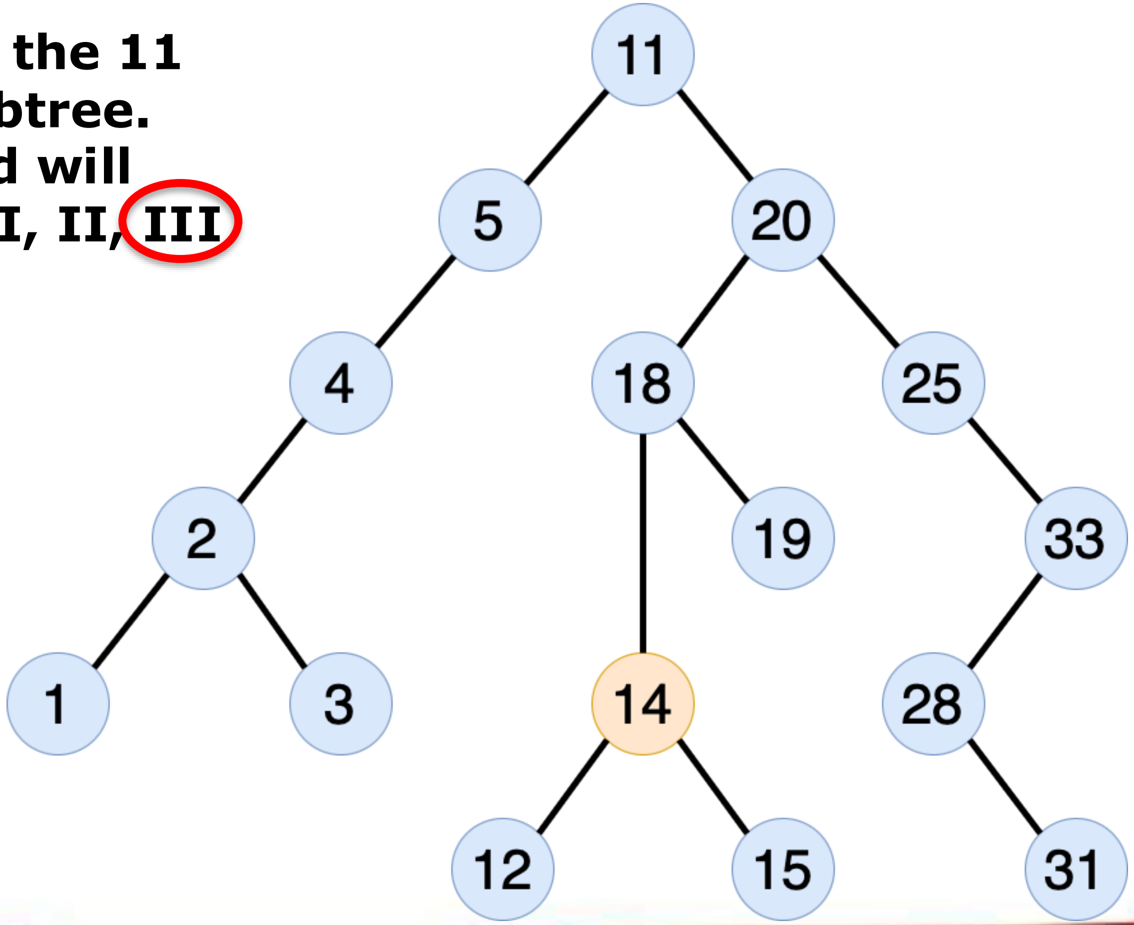
Now we have to remove the 11 we found in the right subtree. Luckily, the node we find will always be either a Case I, II, III removal



Copy the value from the node found in right subtree (11) to the node we want to remove

Let’s remove 7

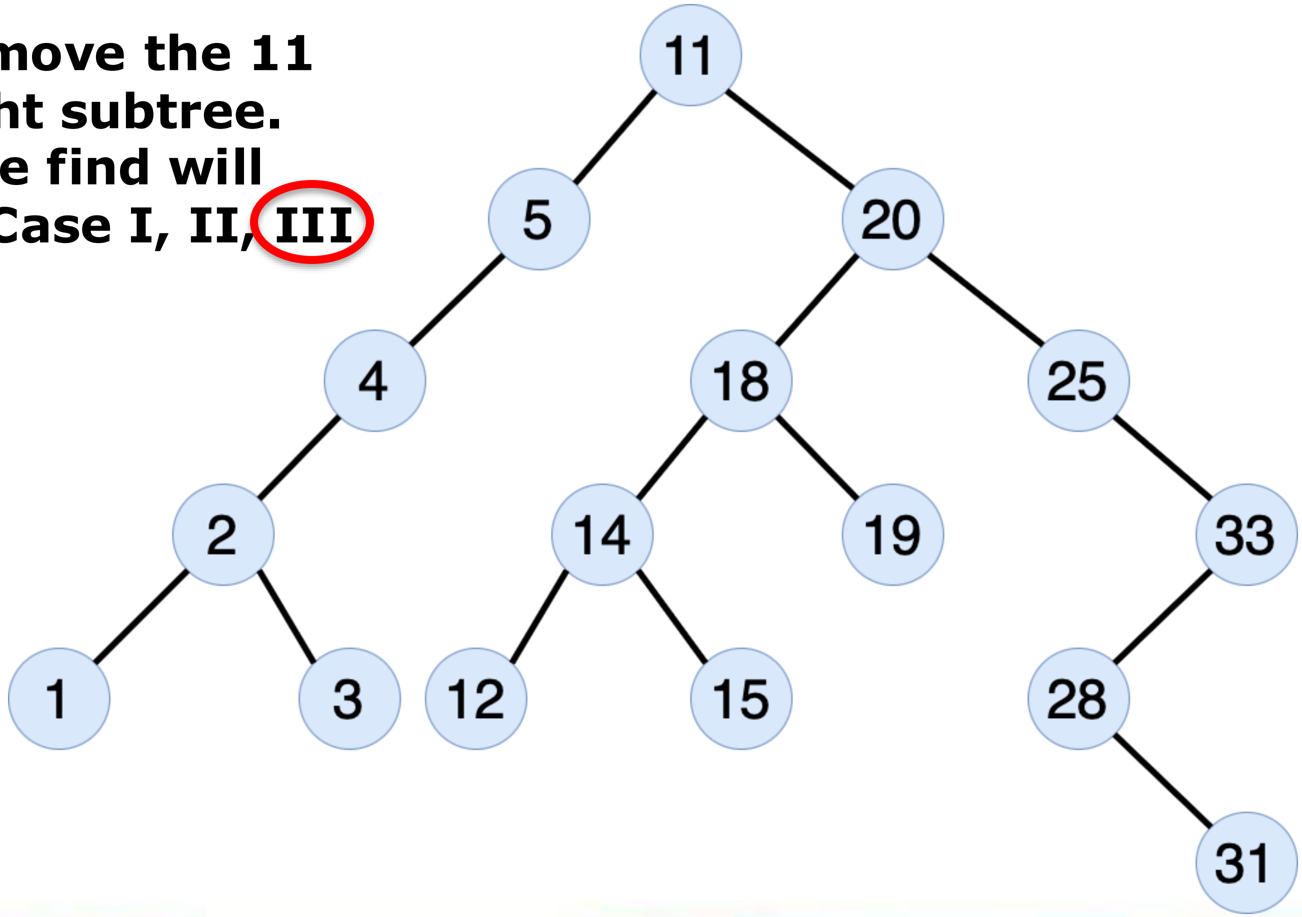
Now we have to remove the 11 we found in the right subtree. Luckily, the node we find will always be either a Case I, II, III removal



Copy the value from the node found in right subtree (11) to the node we want to remove

Let’s remove 7

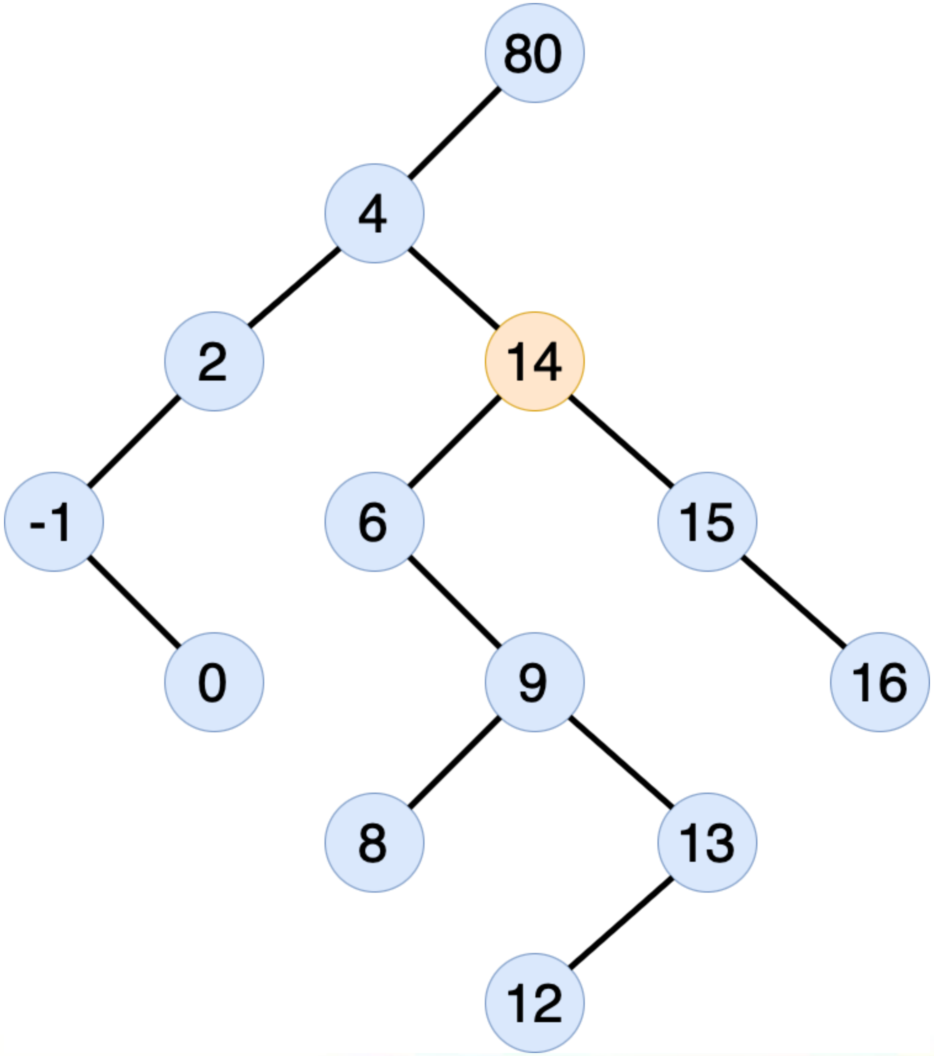
Now we have to remove the 11 we found in the right subtree. Luckily, the node we find will always be either a Case I, II, III removal



Copy the value from the node found in right subtree (11) to the node we want to remove

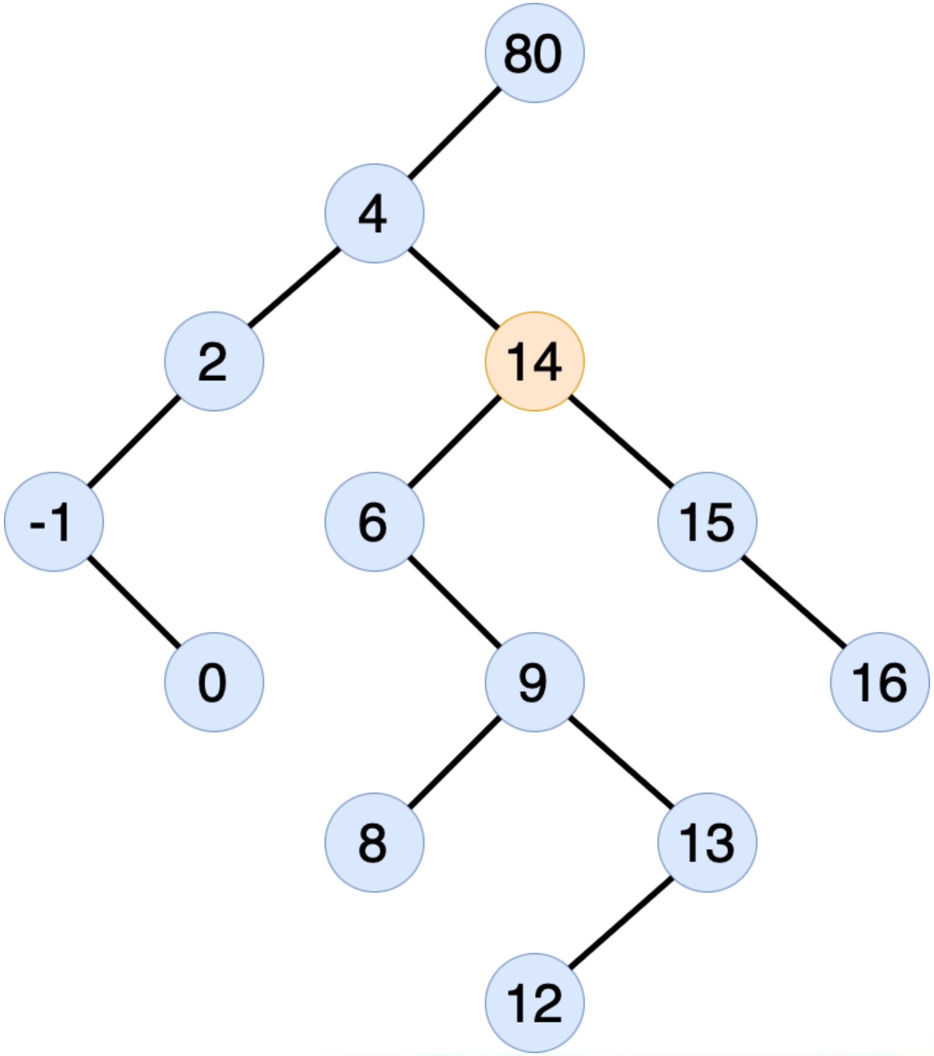
More Example
Case 4

Remove 14



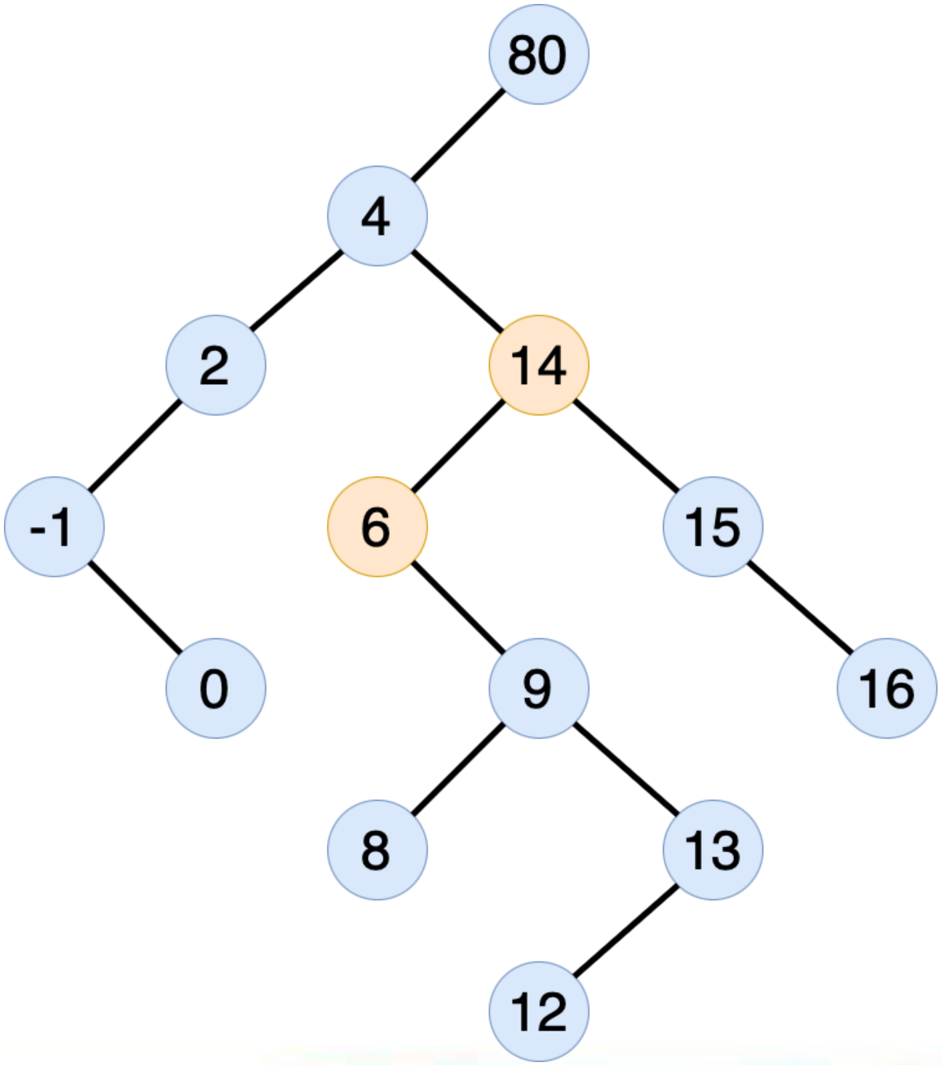
Remove 14

- Let’s do the predecessor this time.
- Dig as far right as possible in the left subtree.



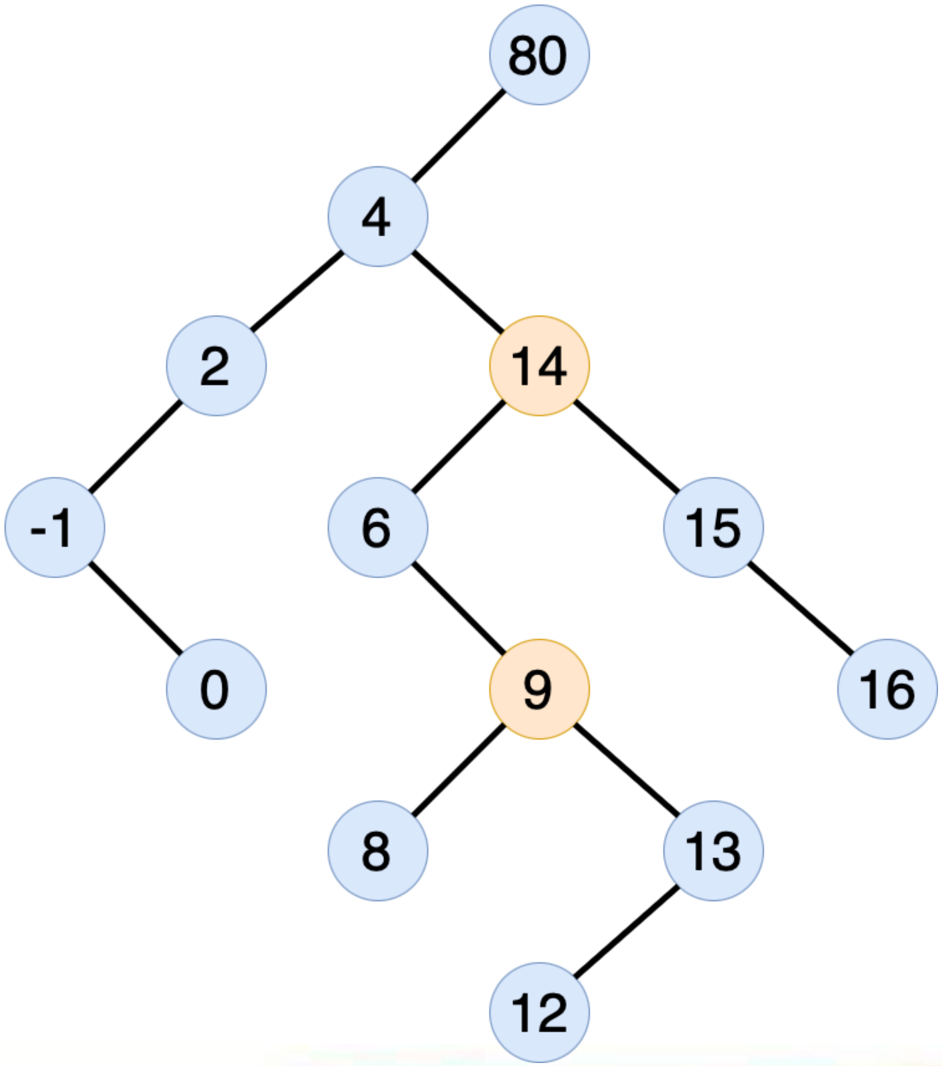
Remove 14

- Let's do the predecessor this time.
- Dig as far right as possible in the left subtree.



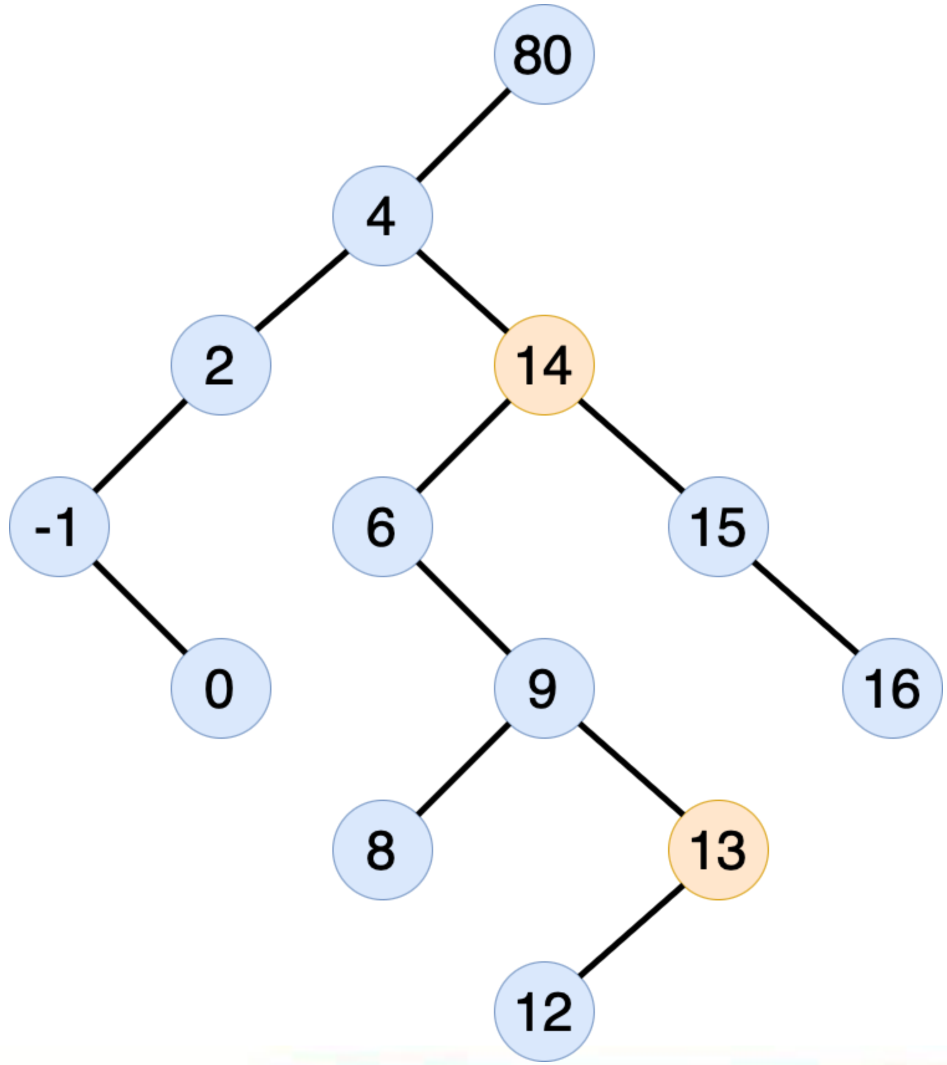
Remove 14

- Let’s do the predecessor this time.
- Dig as far right as possible in the left subtree.



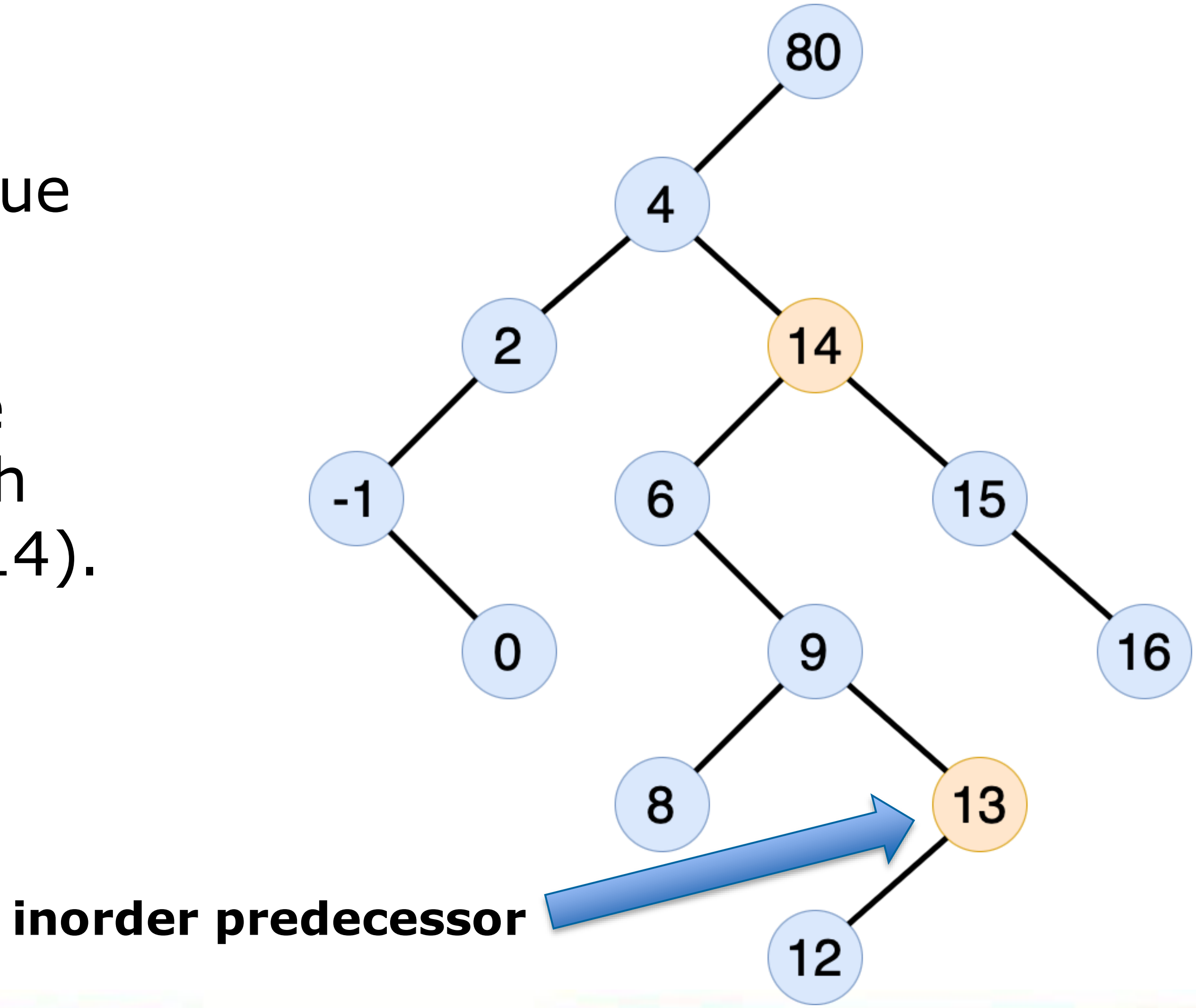
Remove 14

- Let’s do the predecessor this time.
- Dig as far right as possible in the left subtree.



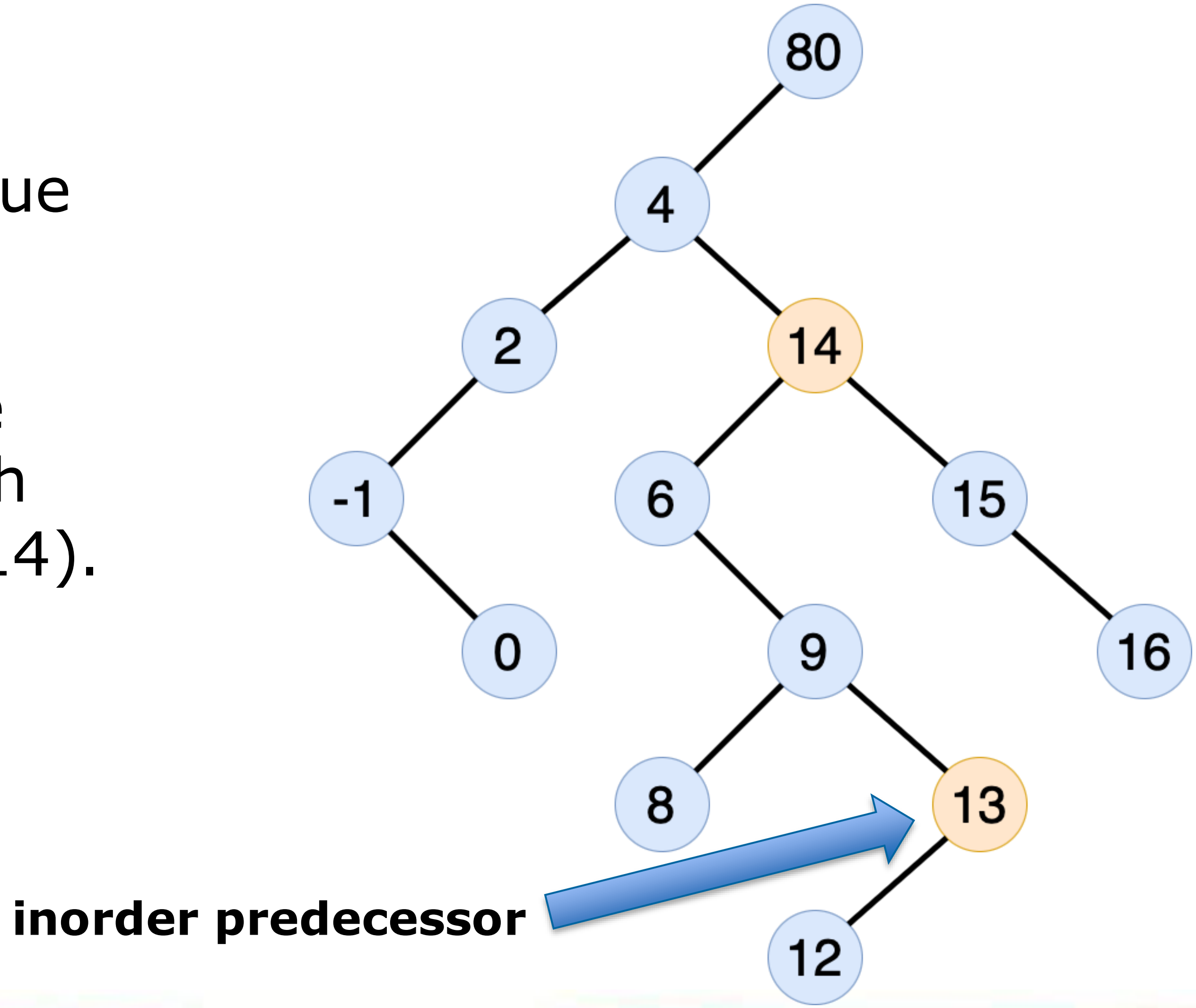
Remove 14

- Copy the value found in the predecessor (13) into the node we wish to remove (14).



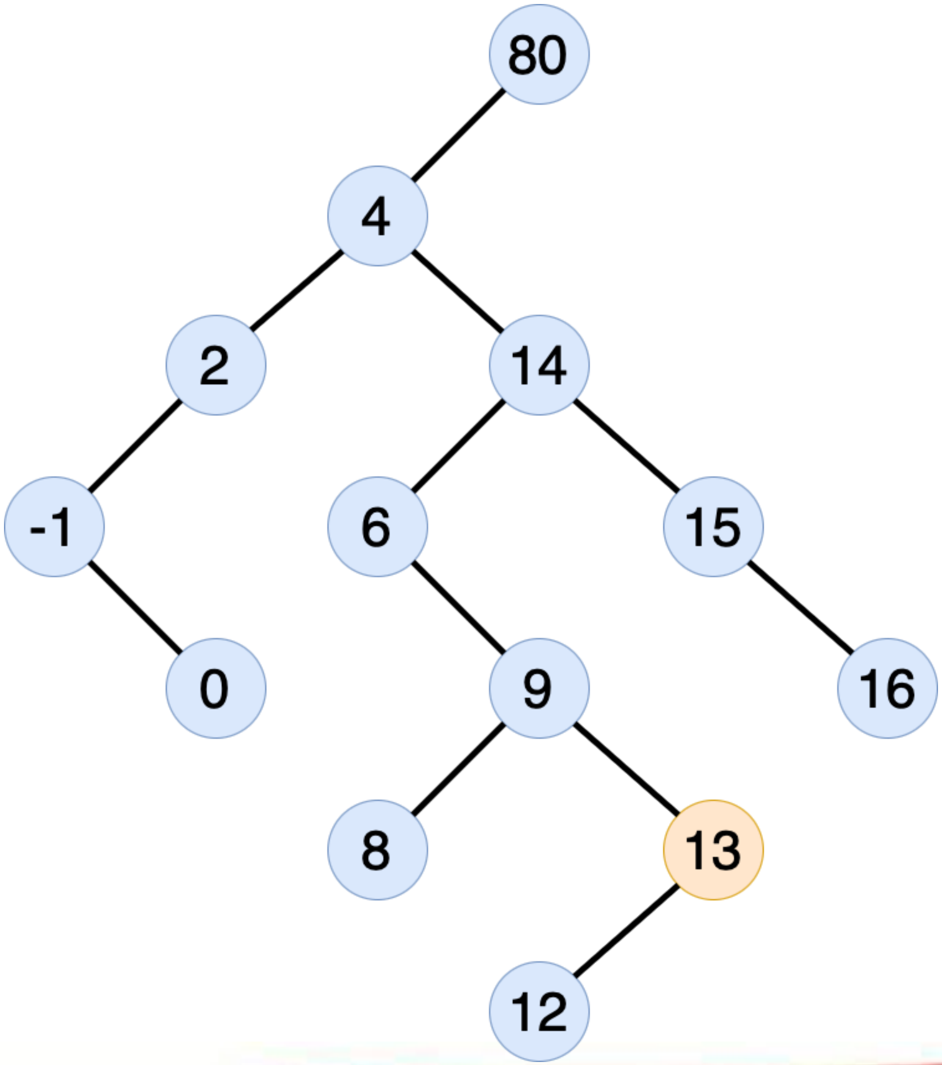
Remove 14

- Copy the value found in the predecessor (13) into the node we wish to remove (14).



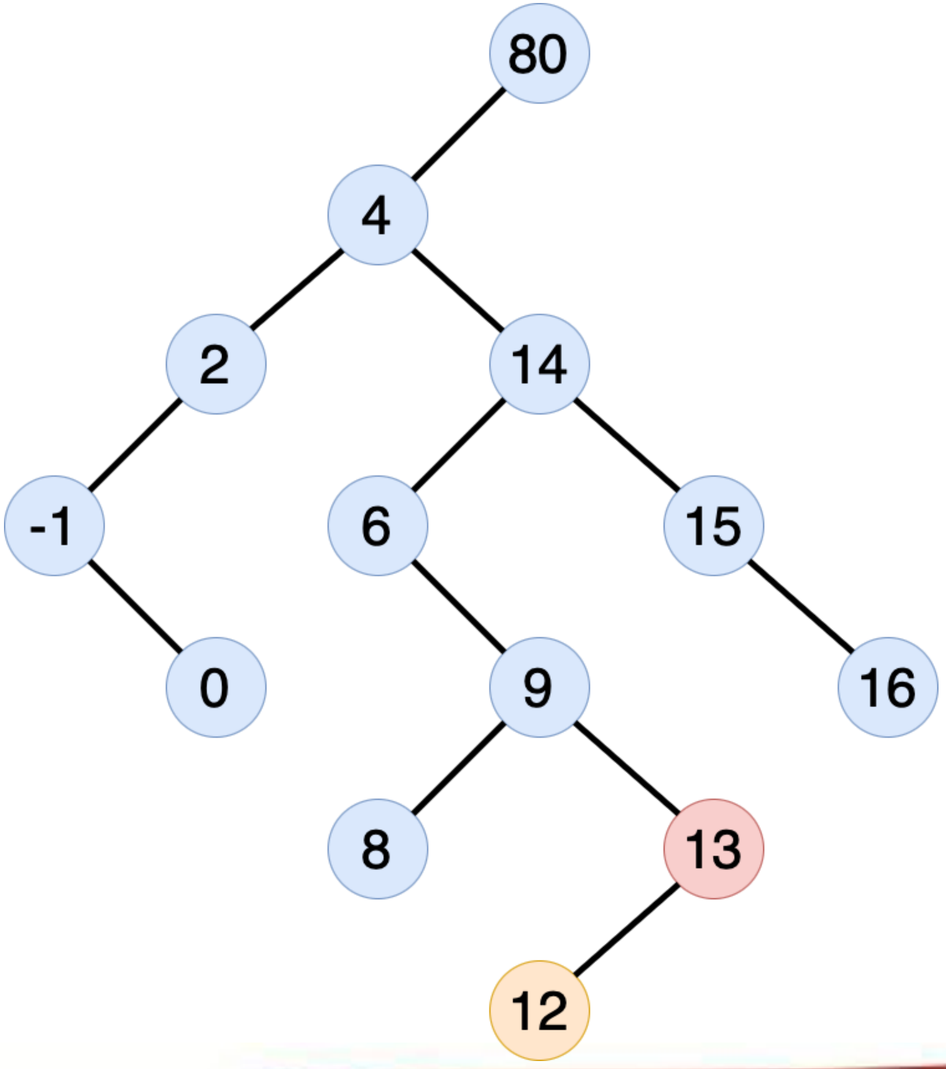
Remove 14

- Remove the predecessor of from the tree.



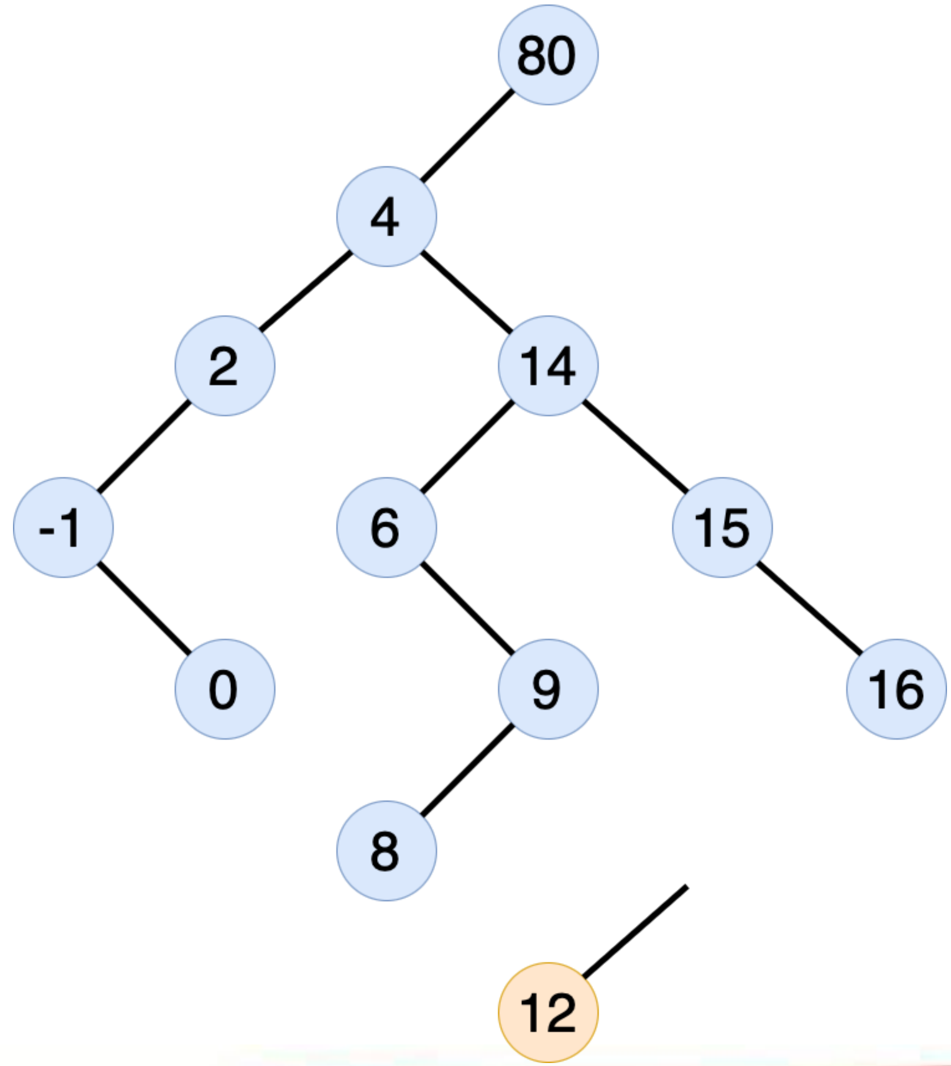
Remove 14

- Case 2:
  Deleting a node
  with left
  subtree only.



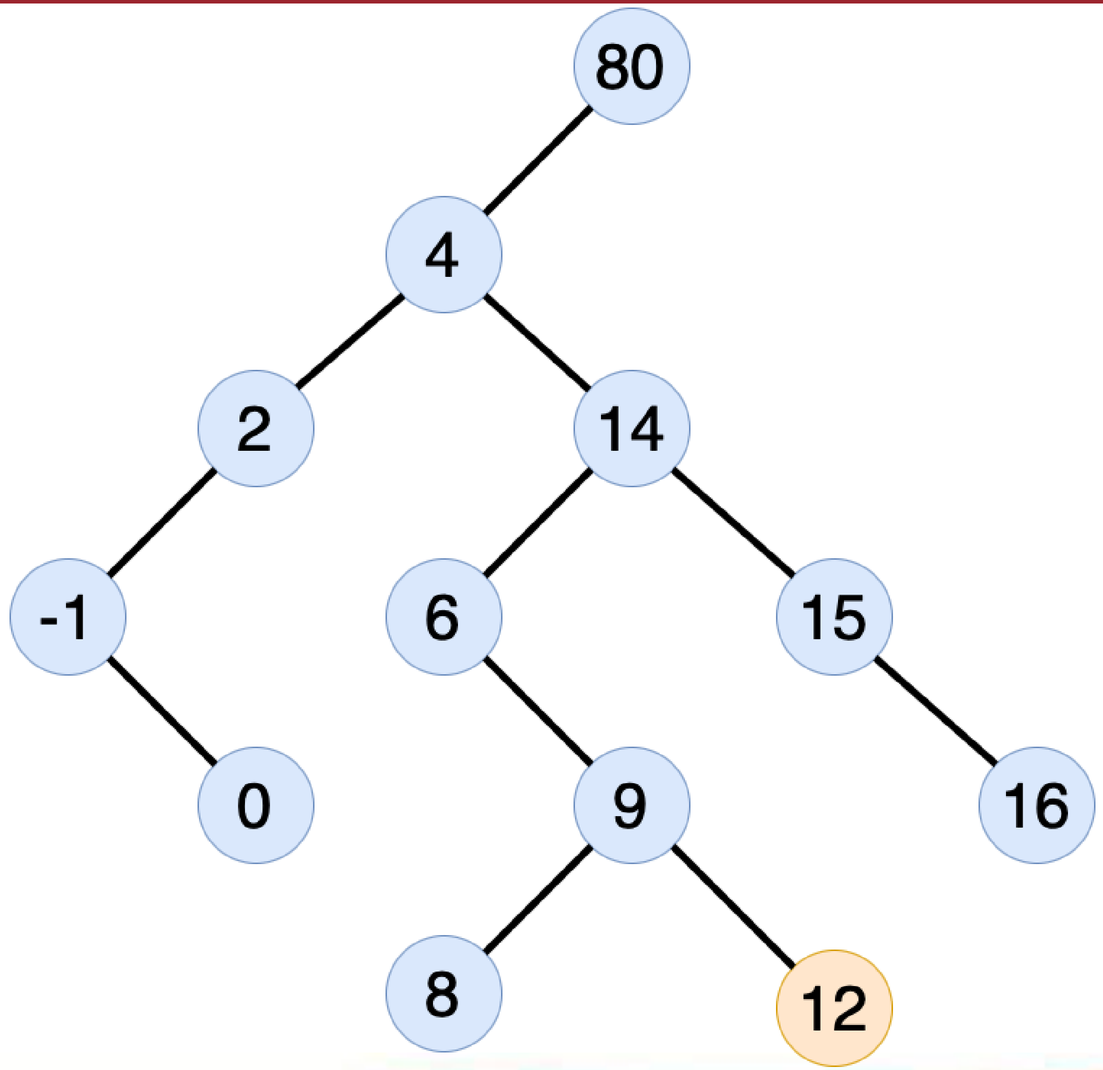
Remove 14

- Case 2:
  Deleting a node with left subtree only.



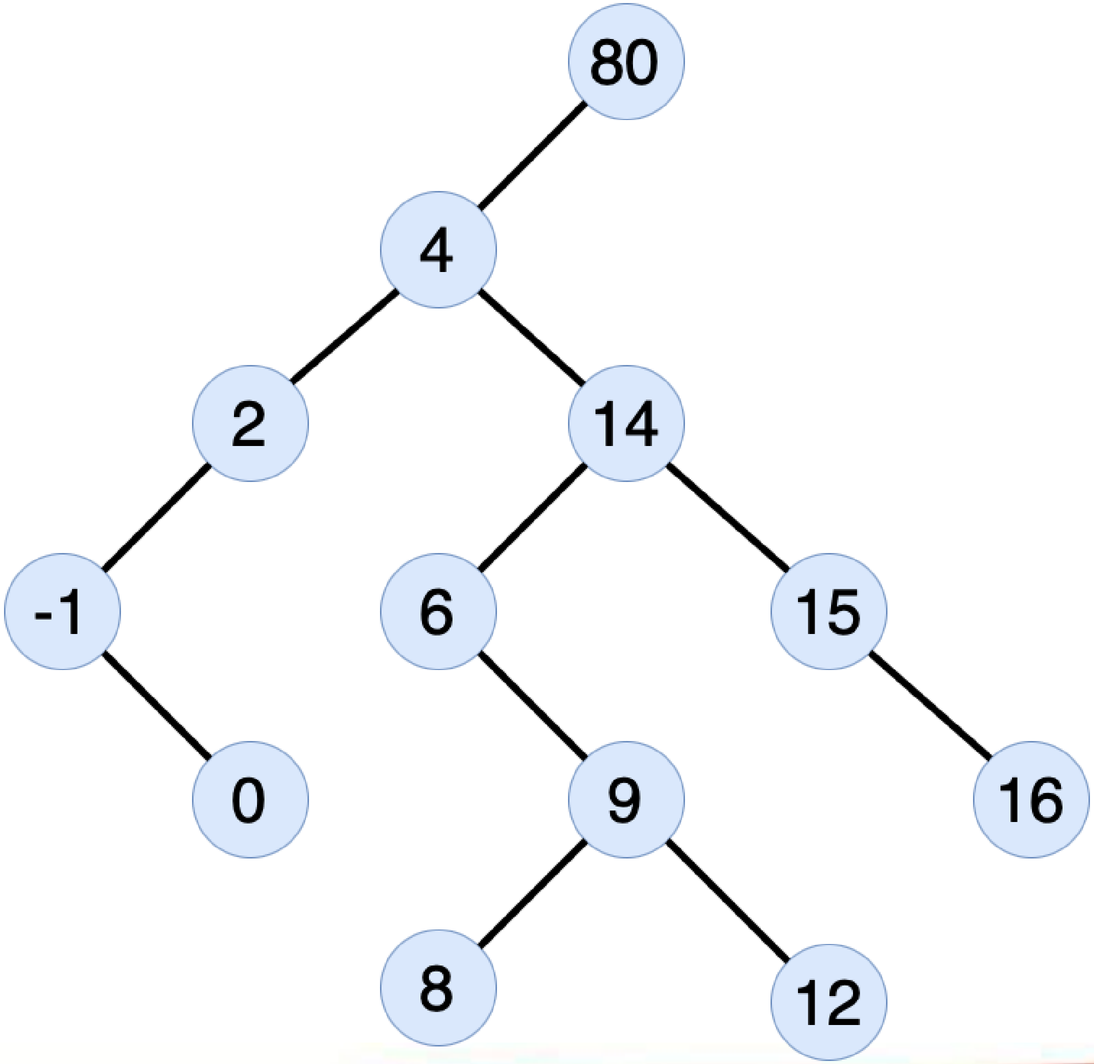
Remove 14

- Case 2:
  Deleting a node with left subtree only.



Remove 14

- Case 2: Deleting a node with left subtree only.



Binary Search Tree : Delete Node

```
function deleteBST( root : BinTree, x : Infotype ) → BinTree
dictionary

algorithm
  if root == NIL then return root endif
  else if ………… { node to remove is smaller/larger than the root }
      FIND Phase
  else  { aha, got you! Prepare to be removed! }
      REPLACE Phase
  endif
endfunction
```

FIND Phase

Binary Search Tree : Delete Node

```
function deleteBST( root : BinTree, x : Infotype ) → BinTree
dictionary

algorithm
  if root == NIL then return root endif
  else if x < root→info then  { node to remove is smaller than the root, go left }
    root→left = deleteBST( root→left, x )
    return root
  else if x > root→info then { node to remove is larger than the root, go right }
    root→right = deleteBST( root→right, x )
    return root
  else  { aha, got you! Prepare to be removed! }

      REPLACE Phase

  endif
endfunction
```

Binary Search Tree : Delete Node

REPLACE Phase

Case 1
Delete a leaf

Case 2
Delete a node
with left subtree only

Case 3
Delete a node
with right subtree only

Case 4
Delete a node
with left snd right subtree

REPLACE Phase

Binary Search Tree : Delete Node

```
else   { aha, got you! Prepare to be removed! }
    if root→left == NIL and root->right == NIL then { case 1: delete a leaf. }
      deallocate( root )
      root = NIL
      return root
    else if root→right == NIL then { case 2: delete node with left subtree only. }
      temp = root
      root = root→left
      deallocate( temp )
      return root
    else if root→left == NIL then { case 3: delete node with right subtree only. }
      temp = root
      root = root→right
      deallocate( temp )
      return root
    else { case 4: delete node with left and right subtree. }
      temp = findMin( root→right ) { use inorder successor. }
      root→info = temp→info
      root→right = deleteBST( root→right, temp→info )
      return root
    endif
endif
```

Binary Search Tree : Delete Node

```
else  { aha, got you! Prepare to be removed! }
    if root→left == NIL and root->right == NIL then { case 1: delete a leaf. }
      deallocate( root )
      root = NIL
    else if root→right == NIL then { case 2: delete node with left subtree only. }
      temp = root
      root = root→left
      deallocate( temp )
    else if root→left == NIL then { case 3: delete node with right subtree only. }
      temp = root
      root = root→right
      deallocate( temp )
    else { case 4: delete node with left and right subtree. }
      temp = findMin( root→right ) { use inorder successor. }
      root→info = temp→info
      root→right = deleteBST( root→right, temp→info )
    endif
    return root
endif
```

Binary Search Tree : Delete Node

```
function deleteBST( root : BinTree, x : Infotype ) → BinTree
dictionary
   temp : Address
   procedure deallocate( Address )
   function findMin( BinTree ) → Address
algorithm
   if root == NIL then return root endif
   else if ………… { node to remove is smaller/larger than the root }

         FIND Phase

   else  { aha, got you! Prepare to be removed! }

         REPLACE Phase

   endif
   return root
endfunction
```

Binary Search Tree : Delete Node

```
function deleteBST( root : BinTree, x : Infotype ) → BinTree
dictionary
   temp : Address
   procedure deallocate( Address )
   function findMin( BinTree ) → Address
algorithm
   if root != NIL then
      if ………… { node to remove is smaller/larger than the root }

            FIND Phase

      else  { aha, got you! Prepare to be removed! }

            REPLACE Phase

      endif
   endif
   return root
endfunction
```

Binary Search Tree : Find Minimum

```
function findMin( root : BinTree ) → Address

endfunction
```

Question?

Exercise

1. Draw all possible binary search trees for the data elements 5, 9, and 12.
2. Create a binary search tree using the following data entered as a sequential set:

```
14 23 7 10 33 56 80 66 70
```

3. Create a binary search tree using the following data entered as a sequential set:

```
7 10 14 23 33 56 66 70 80
```

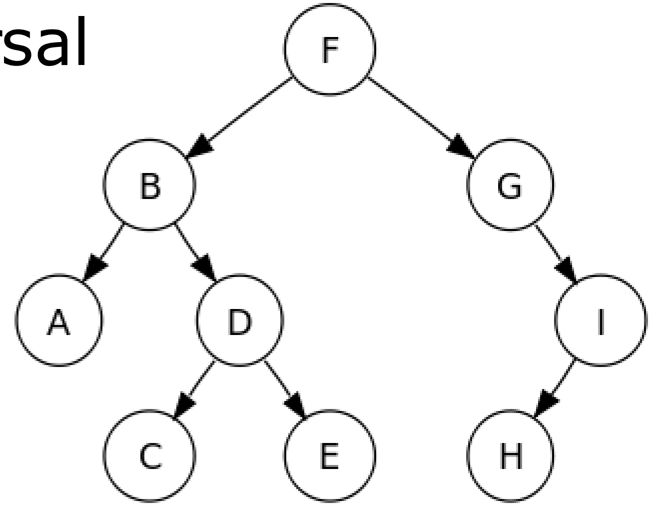
4. Create a binary search tree using the following data entered as a sequential set:

```
80 70 66 56 33 23 14 10 7
```


Reference:
Data Structures: A Pseudocode Approach with C, Second Edition by Richard F. Gilberg & Behrouz A. Forouzan (2005)

Traversal on Binary Tree

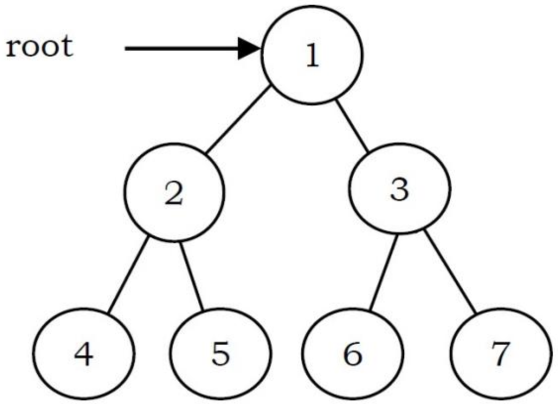
- Dept First Search (DFS) traversal
  - Preorder
  - Inorder
  - Postorder
- Breadth First Search (BFS) traversal
  - Level-order



Preorder Traversal

- This is the simplest traversal to understand.
  - In preorder traversal, each node is processed before (pre) either of its subtrees.
- However, even though each node is processed before the subtrees, it still requires that some information must be maintained while moving down the tree.

Ref.
Data Structures and Algorithms by Narasimha Karumanchi (2017)



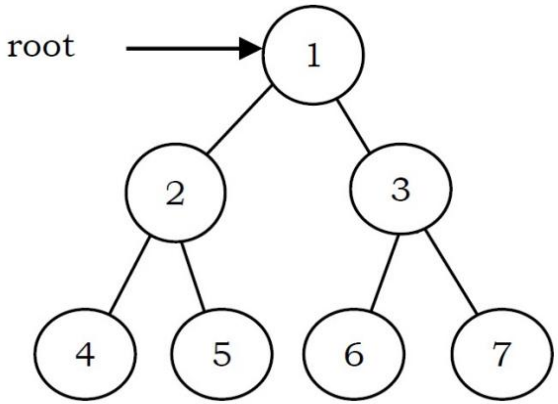
Preorder Traversal

- In the example above, 1 is processed first, then the left subtree, and this is followed by the right subtree.
- Therefore, processing must return to the right subtree after finishing the processing of the left subtree.
- To move to the right subtree after processing the left subtree, we must maintain the root information. The obvious ADT for such information is a stack. Because of its LIFO structure, it is
- possible to get the information about the right subtrees back in the reverse order.

Ref.
Data Structures and Algorithms by Narasimha Karumanchi (2017)

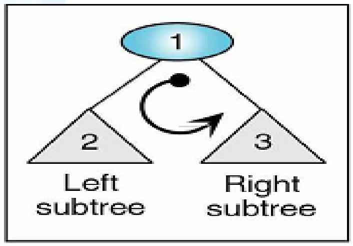
Preorder Traversal

- Preorder traversal is defined as follows:
  - Visit the root.
  - Traverse the left subtree in Preorder.
  - Traverse the right subtree in Preorder.
- Root → Left → Right
- The nodes of tree
  would be visited in the order:
  1 2 4 5 3 6 7

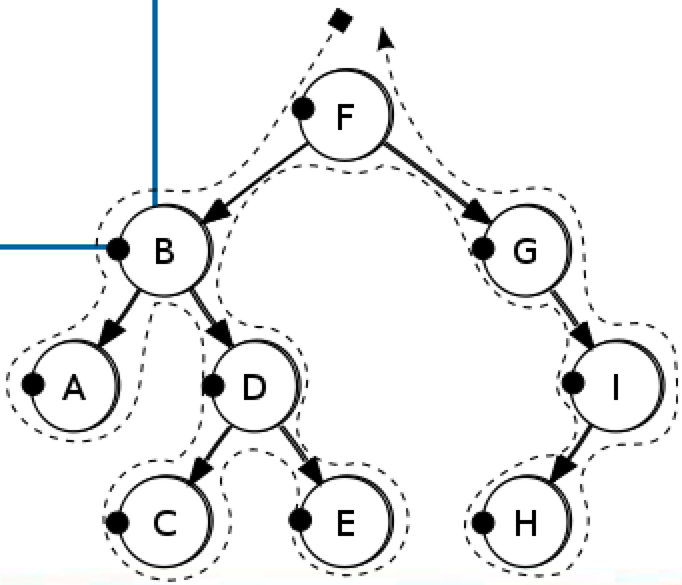


Preorder Traversal

```
procedure preOrder( in root : BinTree )
algorithm
  if root != NIL then
    output( root→info )
    preOrder( root→left )
    preOrder( root→right )
endprocedure
```

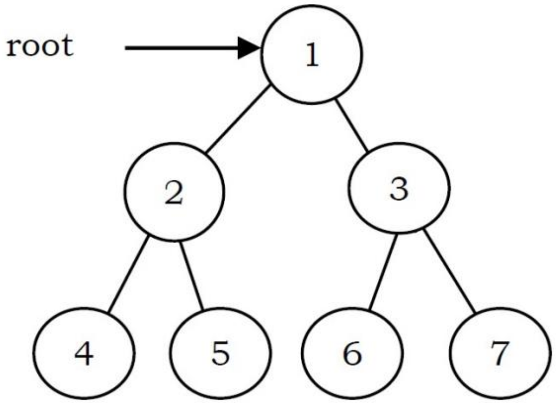


(a) Preorder traversal



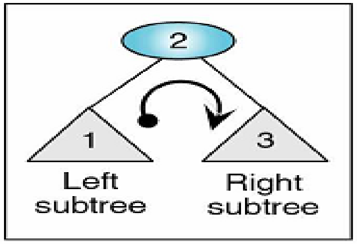
Inorder Traversal

- In Inorder Traversal the root is visited between the subtrees.
- Inorder traversal is defined as follows:
  - Traverse the left subtree in Inorder.
  - Visit the root.
  - Traverse the right subtree in Inorder.
- Left → Root → Right
- The nodes of tree would be visited in the order:
  4 2 5 1 6 3 7

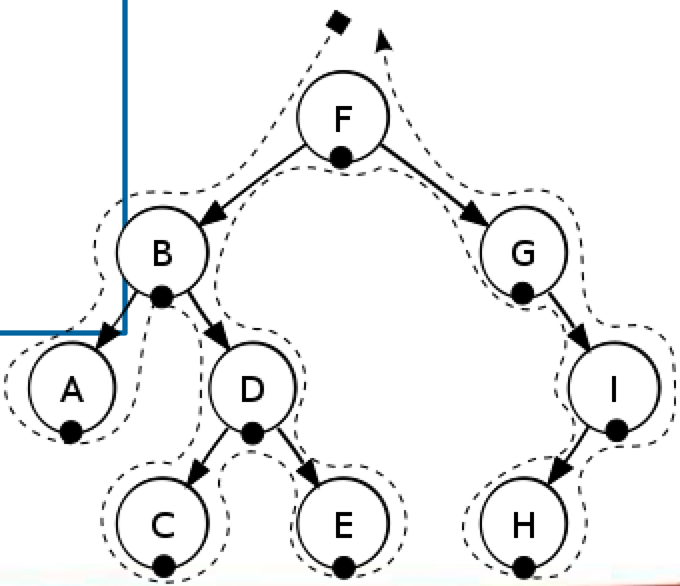


Inorder Traversal

```
procedure inOrder( in root : BinTree )
algorithm
  if root != NIL then
    inOrder( root→left )
    output( root→info )
    inOrder( root→right )
  endif
endprocedure
```

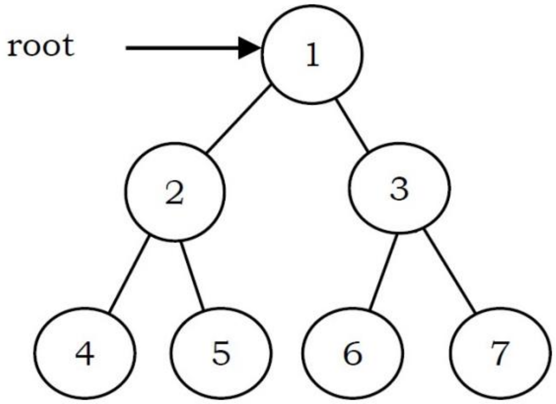


(b) Inorder traversal



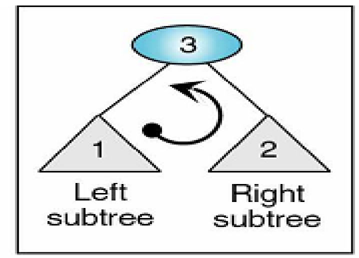
Postorder Traversal

- In postorder traversal, the root is visited after both subtrees.
- Postorder traversal is defined as follows:
  - Traverse the left subtree in Postorder.
  - Traverse the right subtree in Postorder.
  - Visit the root.
- Left → right → Root
- The nodes of the tree would be visited in the order: 4 5 2 6 7 3 1

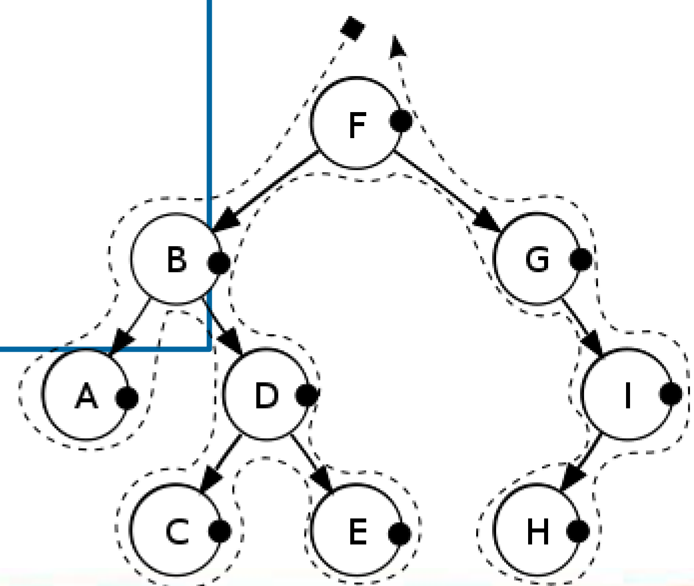


Postorder Traversal

```
procedure postOrder( in root : BinTree )
algorithm
  if root != NIL then
    postOrder( root→left )
    postOrder( root→right )
    output( root→info )
  endif
endprocedure
```



(c) Postorder traversal



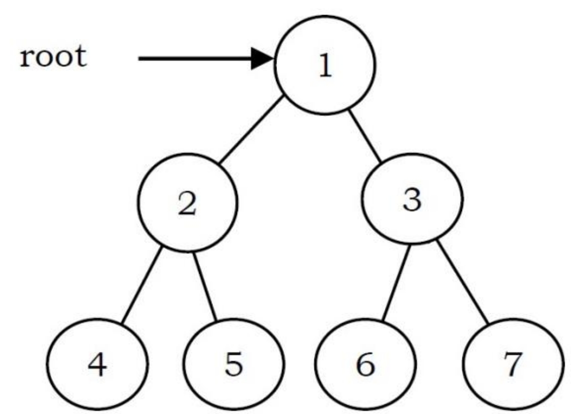
Challenge

Can you write non-recursive preorder, inorder, and postorder traversal?

Hint: use stack!

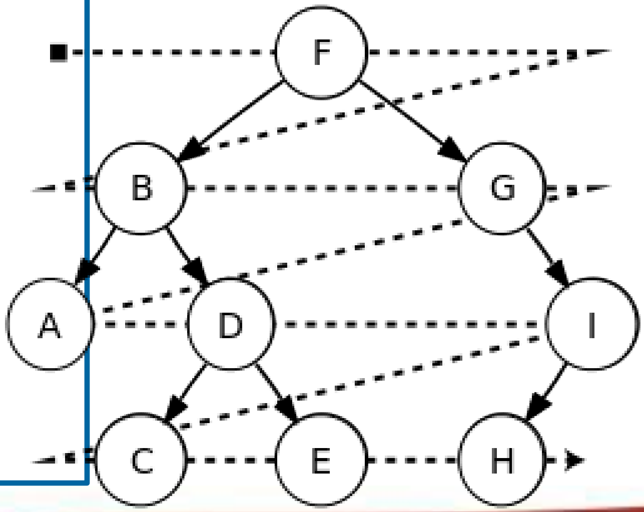
Level-order Traversal

- Level order traversal is defined as follows:
  - Visit the root.
  - While traversing level (, keep all the elements at level (+ 1 in queue.
  - Go to the next level and visit all the nodes at that level.
  - Repeat this until all levels are completed.
- The nodes of the tree are visited in the order:
  1 2 3 4 5 6 7

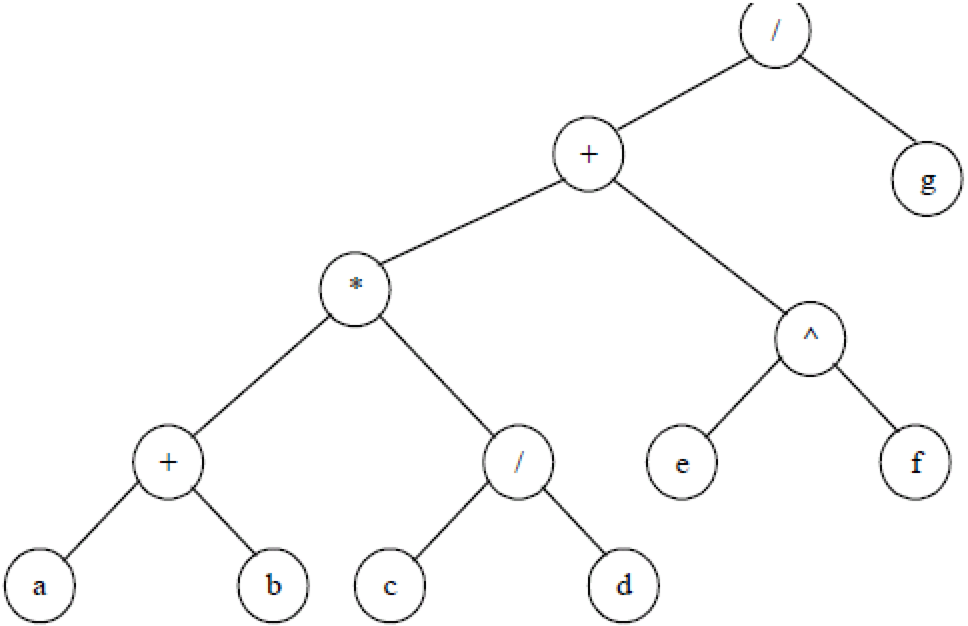


Level-order Traversal

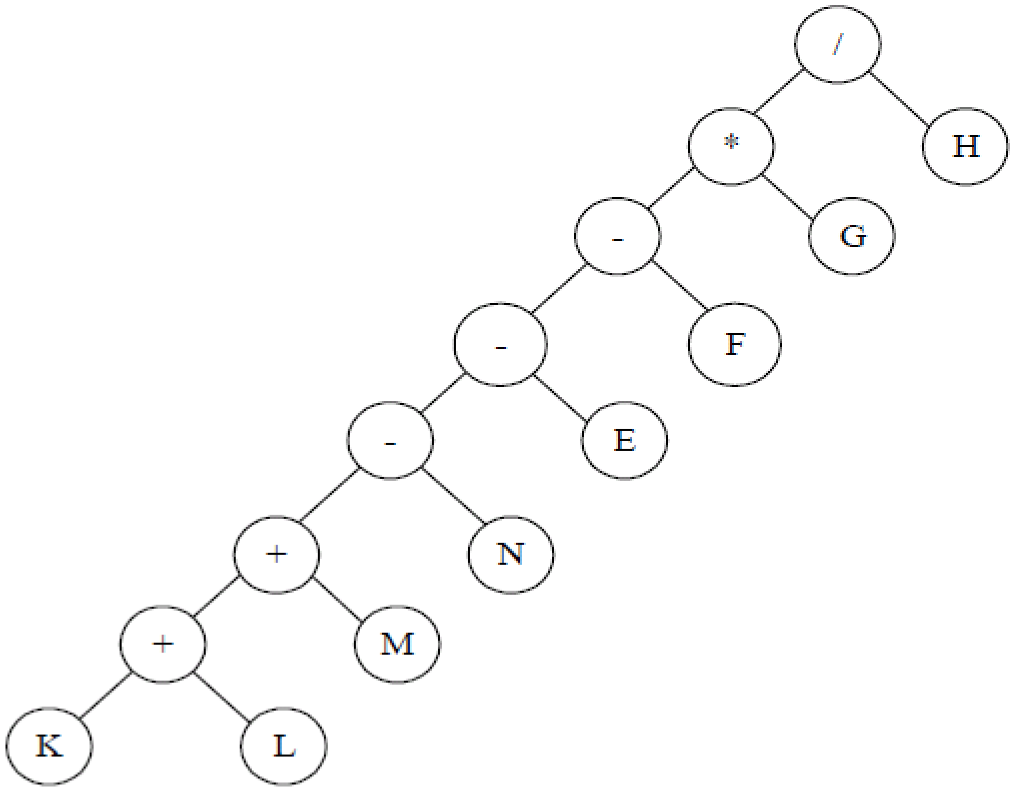
```
procedure levelOrder( in root : BinTree )
dictionary
  Q : Queue
  temp : Address
algorithm
  enqueue( Q, root )
  while not isEmpty(Q) do
    temp = dequeue( Q )
    output( temp-->info )
    if temp-->left != NIL do
      enqueue( Q, temp-->left )
    endif
    if temp-->right != NIL do
      enqueue( Q, temp-->right )
    endif
  endwhile
endprocedure
```



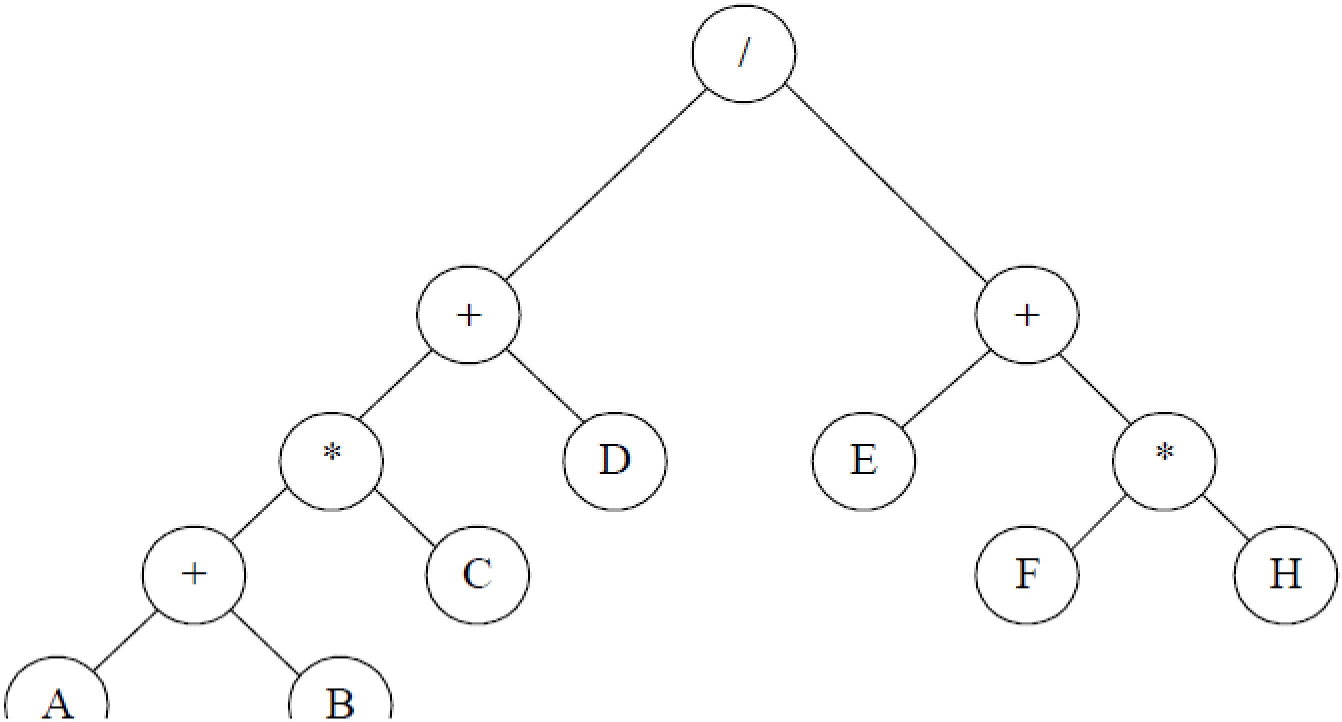
Exercise – write the traversal - 1



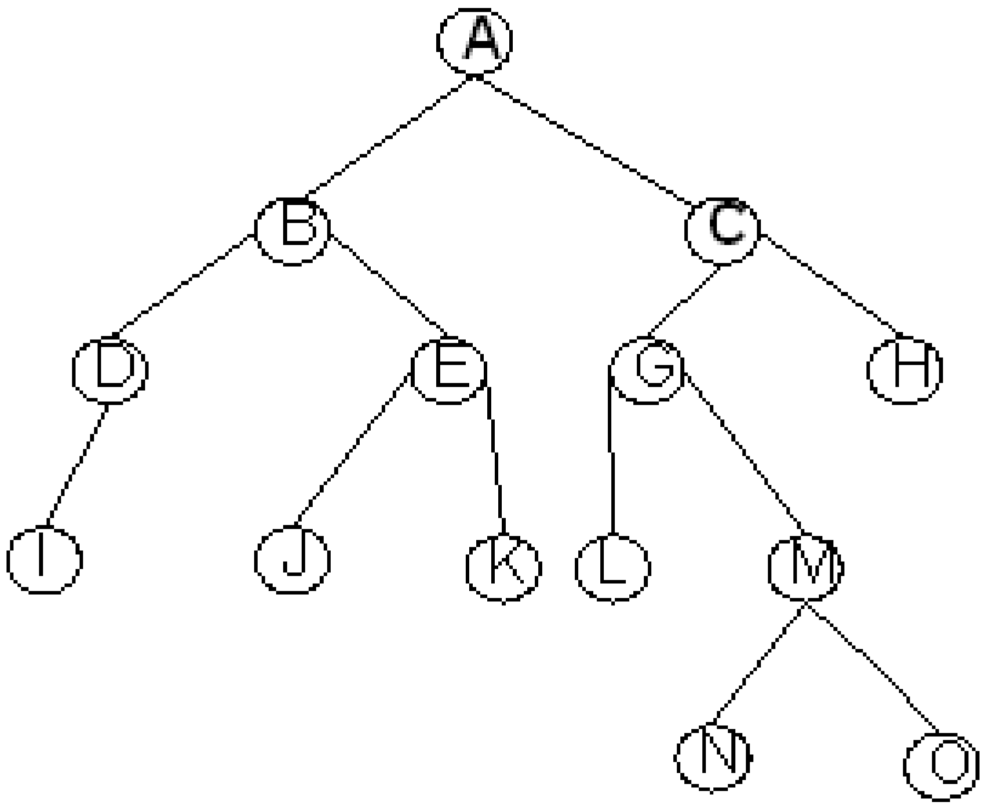
Exercise – write the traversal - 2



Exercise – write the traversal - 3



Exercise – write the traversal - 4



Exercise – Create the Tree

- Assume there is ONE tree, which if traversed by inorder resulting : EACKFHDBG, and when traversed by preorder resulting : FAEKCDHGB
  - Draw the tree that satisfy the condition above
- Find a binary tree whose preorder and inorder traversals create the same result.

Question?

More Exercise

You can find more exercises on tree in (Gilberg & Forouzan, 2005, p. 292), (Weiss, 2014, p. 201), (Drozdek, 2013, p. 298), or else.

References

Weiss, M. A. (2014). Data Structures and Algorithm Analysis in C++.

Gilberg, R. F. & Forouzan, A. (2005). Data Structures: A Pseudocode Approach with C, 2nd Ed.

Drozdek, A. (2013). Data Structures and Algorithms.

Karumanchi, N. (2017). Data Structures and Algorithms.

THANK YOU