Data Structure and Algorithms

Session-24

Dr. Subhra Rani Patra
SCOPE, VIT Chennai

Time & Space Complexity-
'Pre-Order Traversal' of Binary Tree(Linked-List implementation):

preorderTraversal(root) -- T(n)

if (root equals null) -- O(1)

return error message --- O(1)

else -- O(1)

print root --- O(1)

preorderTraversal (root.left) --- T(n/2)

preorderTraversal(root.right) -- T(n/2)

Time Complexity – O(n)

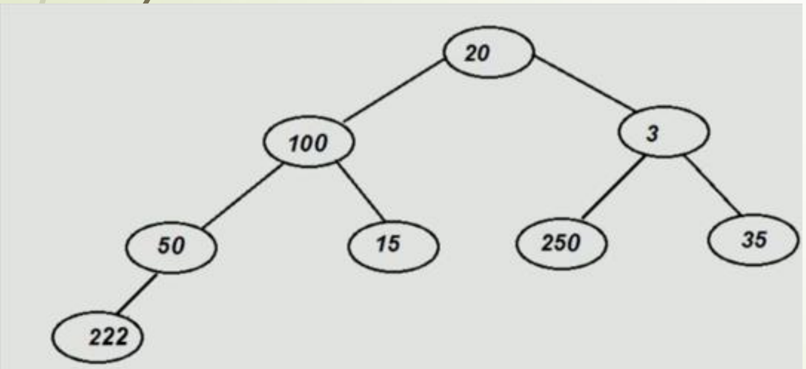
Space Complexity – O(n)

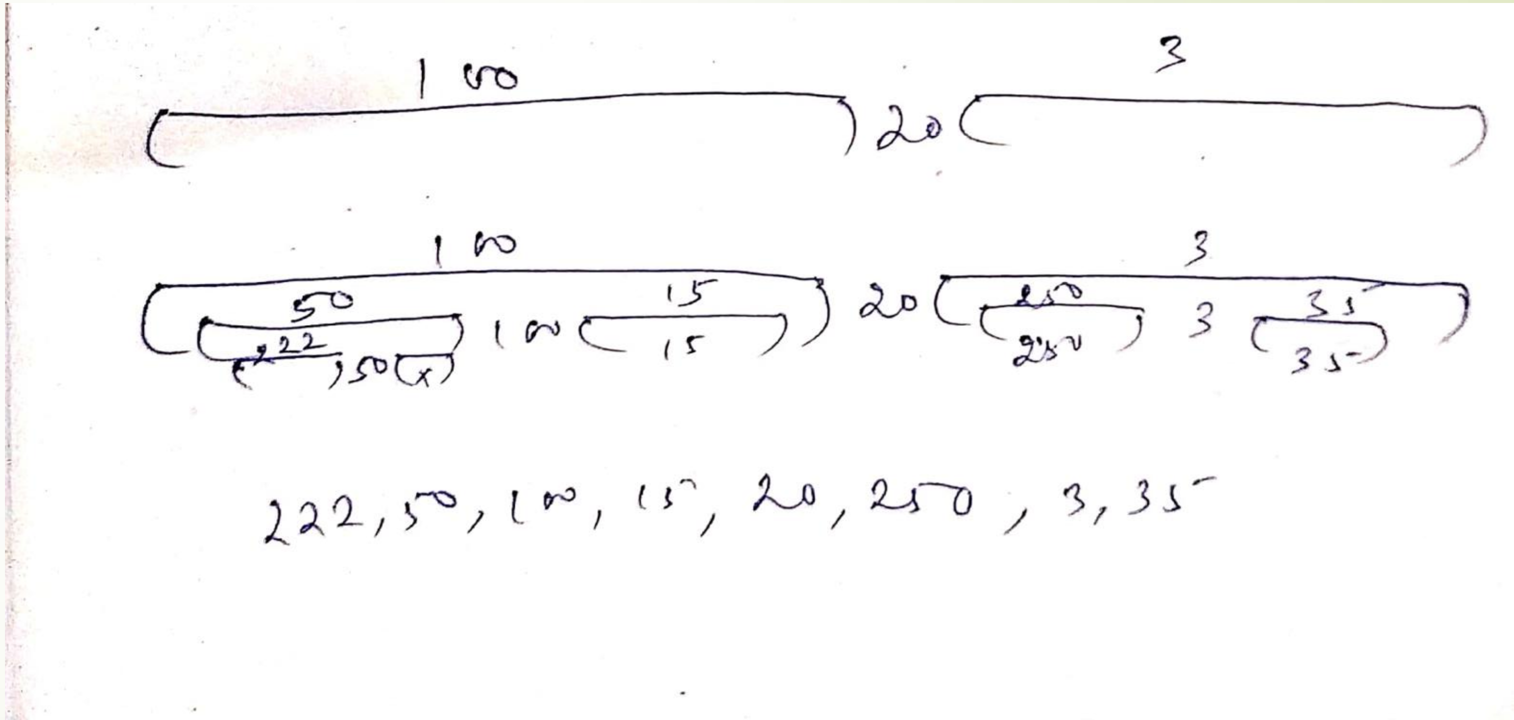
Algorithm-
'In-Order Traversal' of Binary Tree(Linked-List implementation):

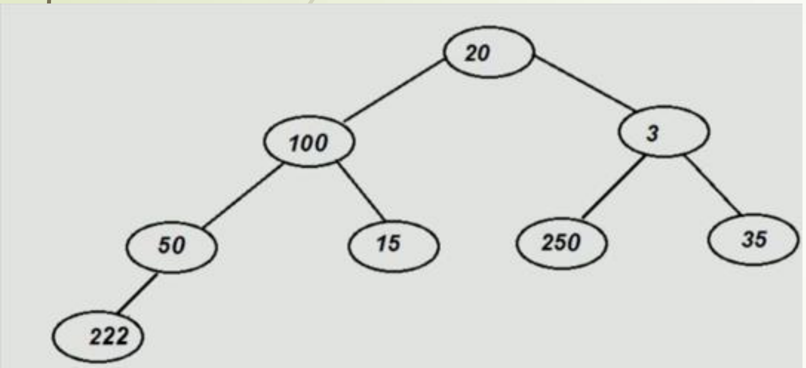
Left Subtree

Root

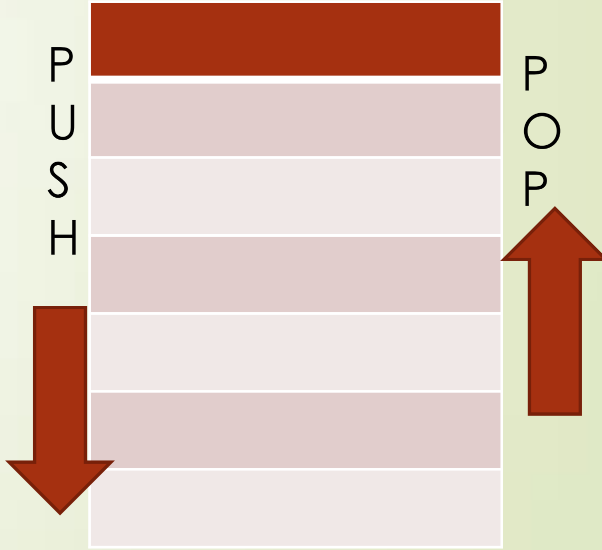
Right Subtree








```
inOrderTraversal (root)
  if (root equals null)
    return
  else
     inOrderTraversal(root.left)
     print root
     inOrderTraversal(root.right)
```



Time & Space Complexity-
'In-Order Traversal' of Binary Tree(Linked-List implementation):

inOrderTraversal (root) -- T(n)

if (root equals null) -- O(1)

return -- O(1)

else -- O(1)

inOrderTraversal(root.left) --- T(n/2)

print root --- O(1)

inOrderTraversal(root.right) -- T(n/2)

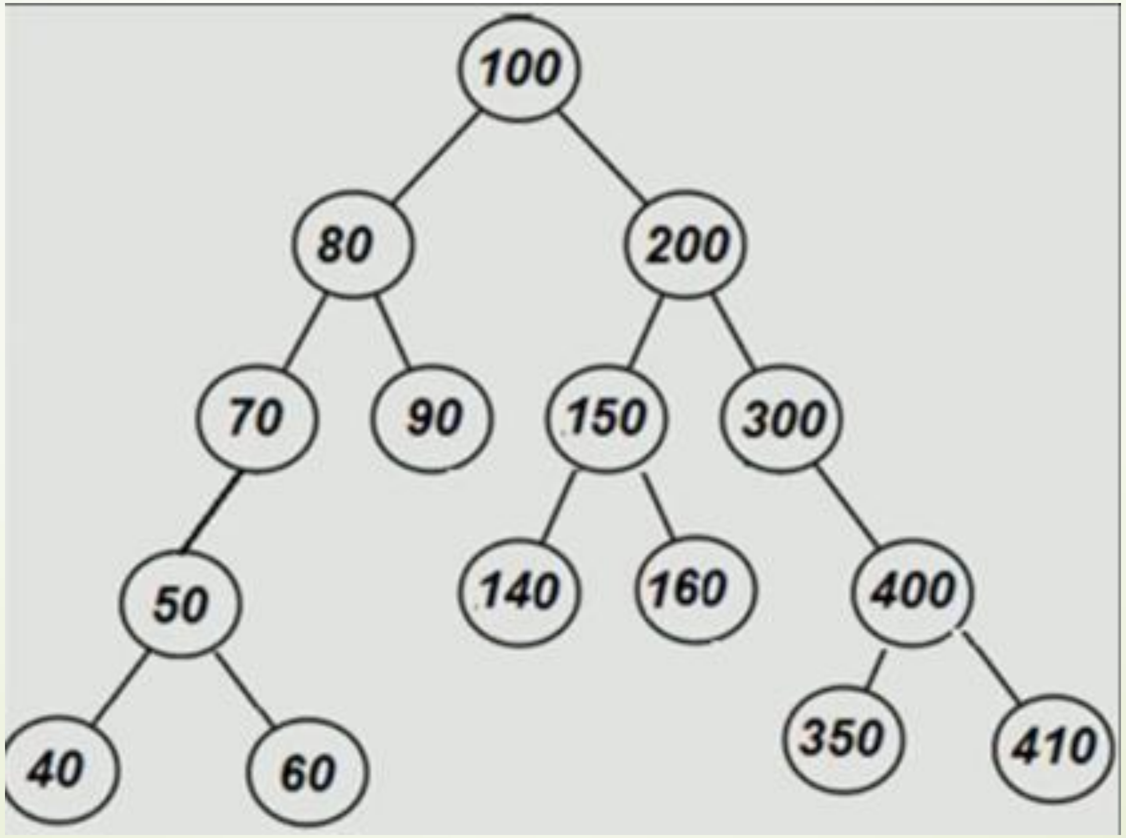
Time Complexity – O(n)

Space Complexity – O(n)

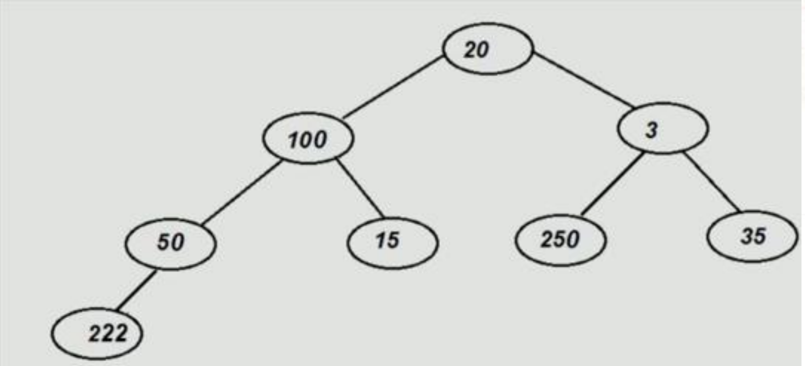
***__Predecessor__**: Predecessor of a node is the immediate previous node in Inorder traversal of the Binary Tree.*

***__Successor__**: Successor of a node is the immediate next node in Inorder traversal of the Binary Tree.*

__Inorder traversal__ - 40, 50, 60, 70, 80, 90, 100, 140, 150, 160, 200, 300, 350, 400, 410



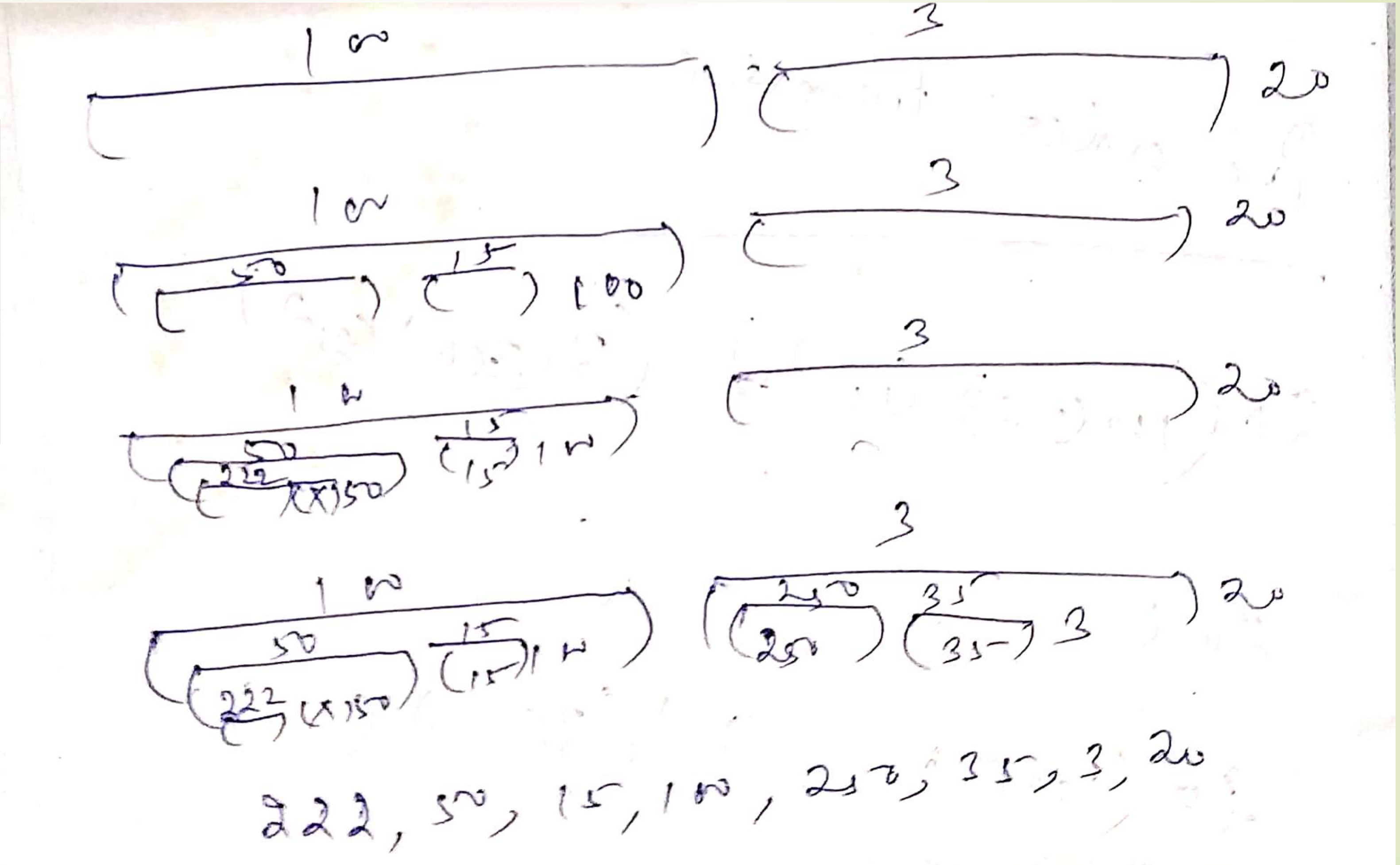
Algorithm-
‘Post-Order Traversal’ of Binary Tree(Linked-List implementation):



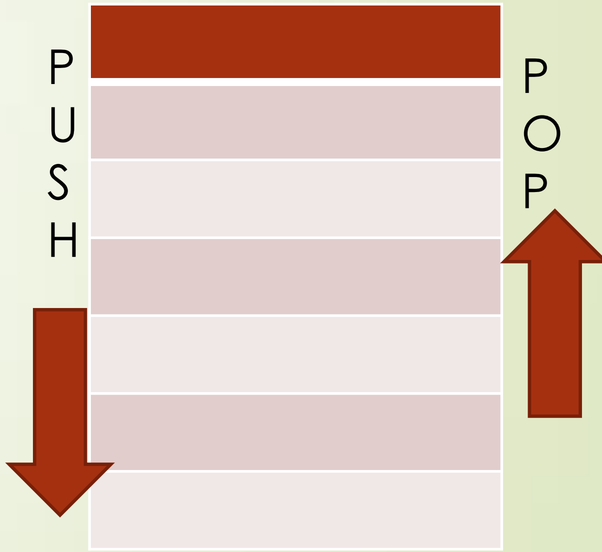
Left Subtree

Right Subtree

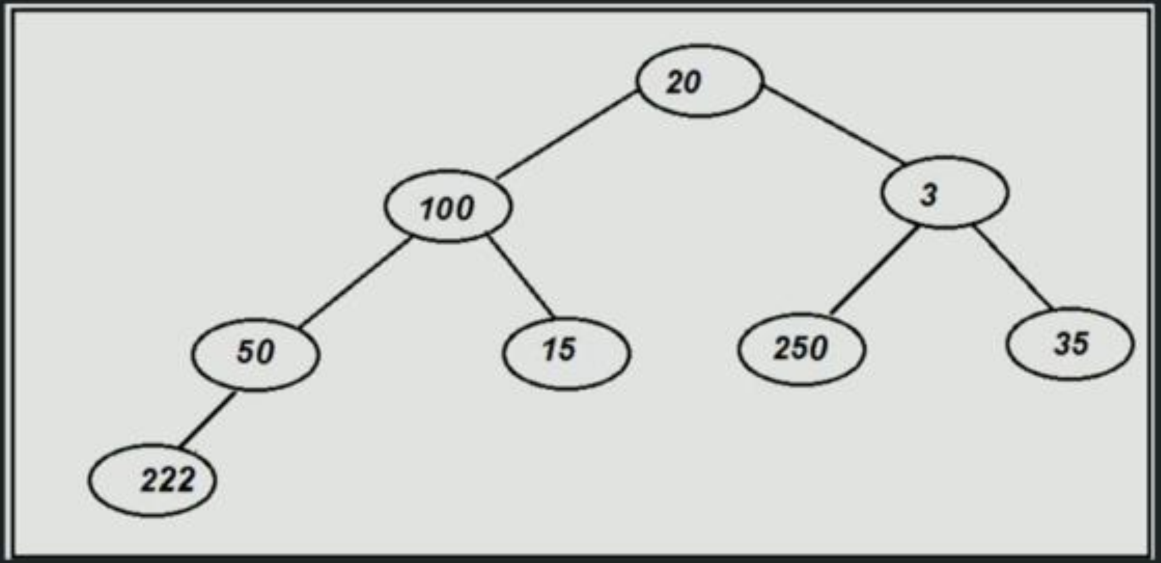
Root



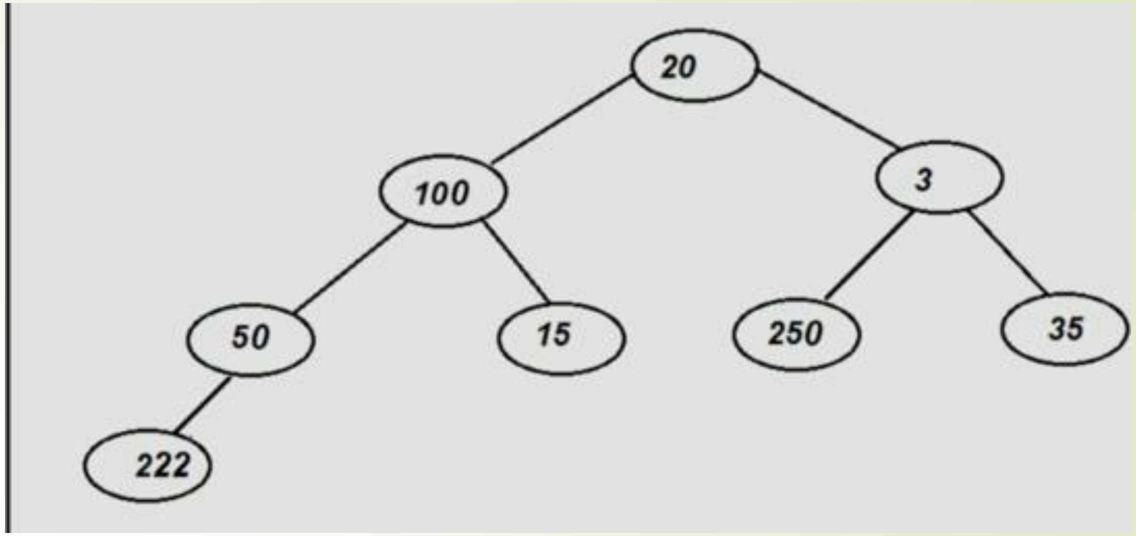
```
postOrderTraversal(root)
  if (root equals null)
      return
  else
      postOrderTraversal(root.left)
      postOrderTraversal(root.right)
      print root
```

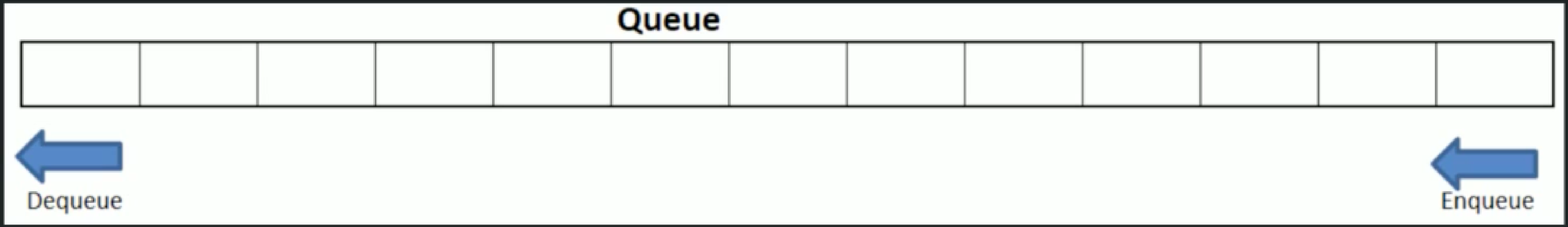


Algorithm-
'Level Order Traversal' of Binary Tree(Linked-List implementation):



```
levelOrderTraversal(root)
    Create a Queue(Q)
    enqueue(root)
    while (Queue is not empty)
        enqueue() the child of first element
        dequeue() and print
```





20,100,3,50,15,250,25,222

Queue

Time & Space Complexity- 'Level Order Traversal' of Binary Tree(Linked-List implementation):

levelOrderTraversal(root)

create a Queue(Q) -- O(1)

enqueue(root) -- O(1)

while (Queue is not empty) -- O(n)

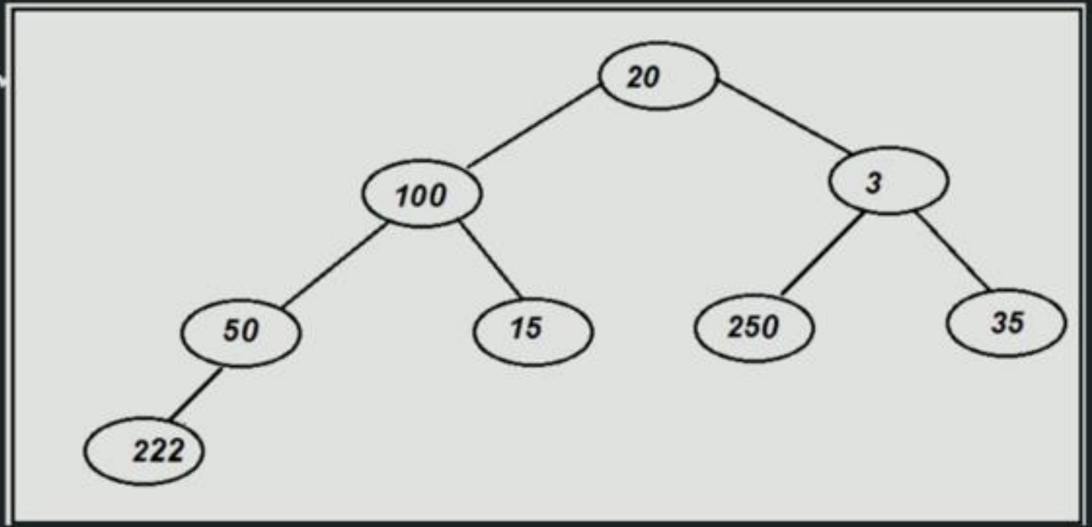
enqueue() the child of first element ------------------------------------- O(1)

dequeue() and print -- O(1)

Time Complexity – O(n)

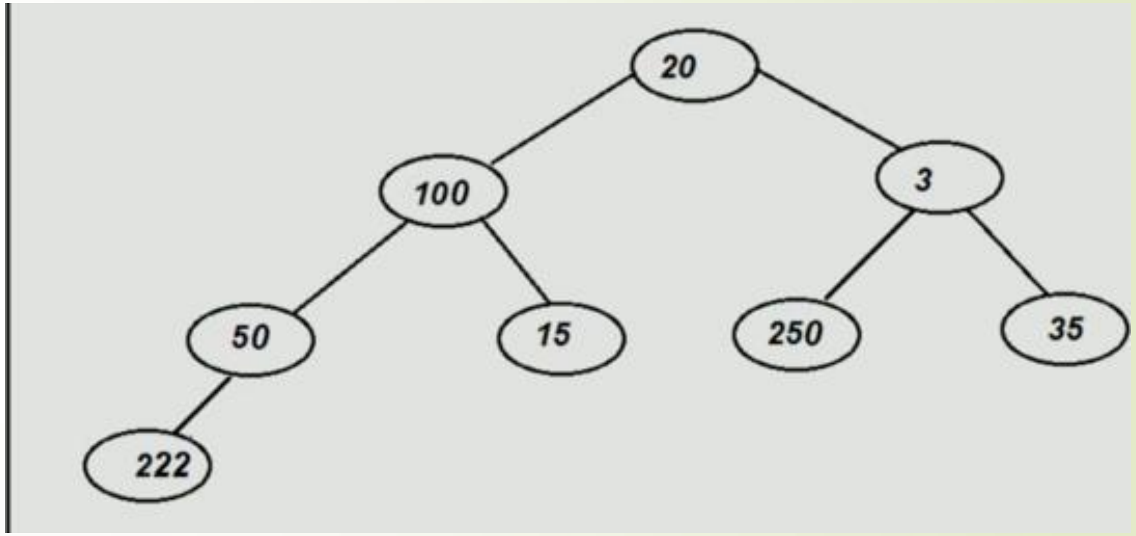
Space Complexity – O(n)

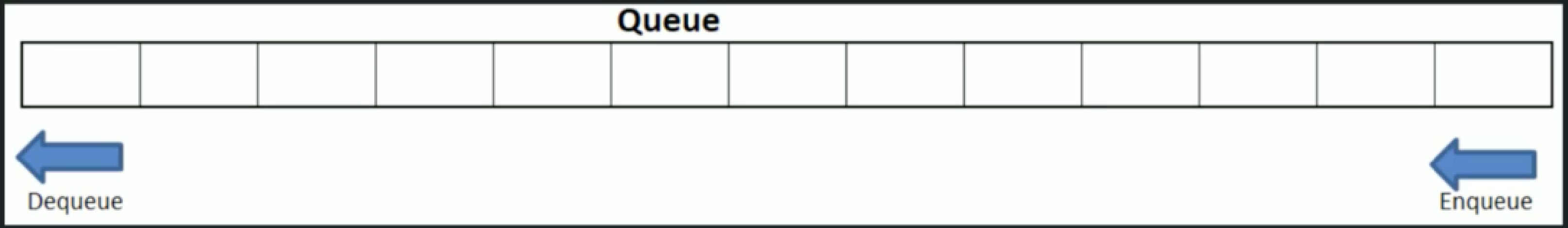
Searching a node in Binary Tree(Linked-List implementation):



Algorithm-
Searching a node in Binary Tree(Linked-List implementation):

```
searchForGivenValue (value)
 if root == null
   return error message
 else
   do a level order traversal
     if value found
       return success message
   return unsuccessful message
```





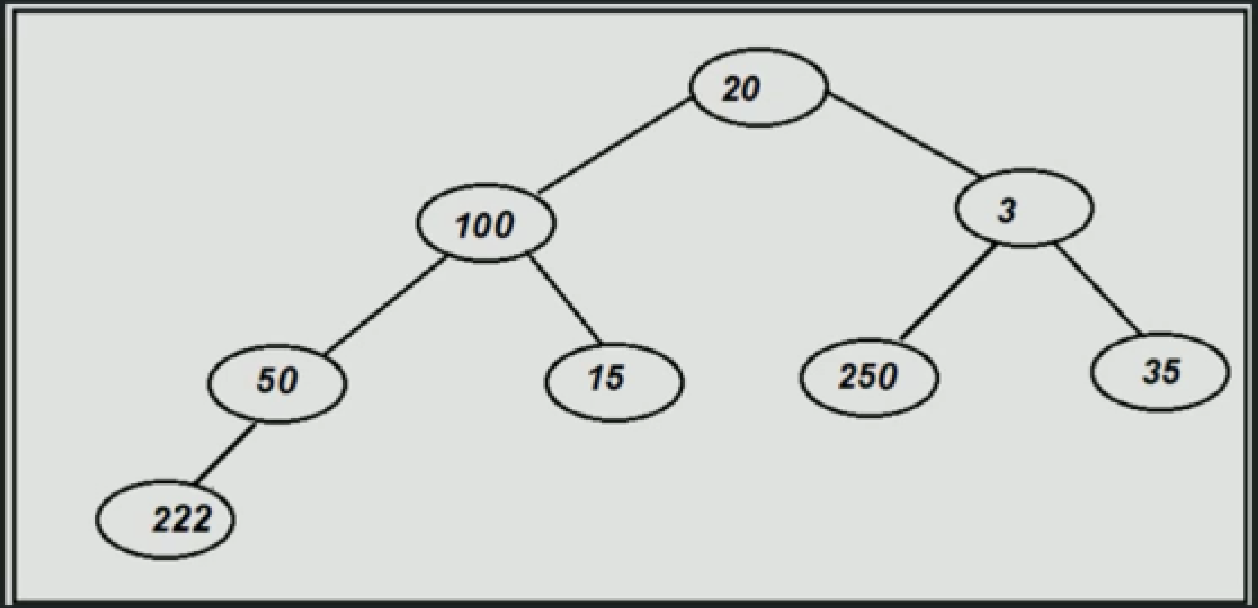
Time & Space Complexity- Searching a node in Binary Tree(Linked-List implementation):

```
searchForGivenValue (value)
  if root = null ------------------------------------------------ O(1)
    return error message ---------------------------------------- O(1)
  else ---------------------------------------------------------- O(1)
    do a level order traversal ---------------------------------- O(n)
      if value found  ------------------------------------------- O(1)
        return success message ---------------------------------- O(1)
    return  unsuccessful message -------------------------------- O(1)
```

Time Complexity – O(n)

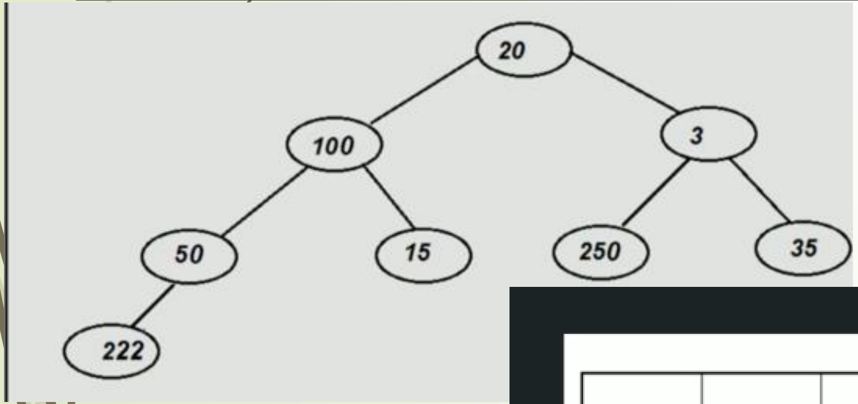
Space Complexity – O(n)

Insertion of node in Binary Tree (Linked-List implementation):



Algorithm-
Insertion of node in Binary Tree (Linked-List implementation):

- ✓ Insertion:
  - ✓ When the root is blank
  - ✓ Insert at first vacant child



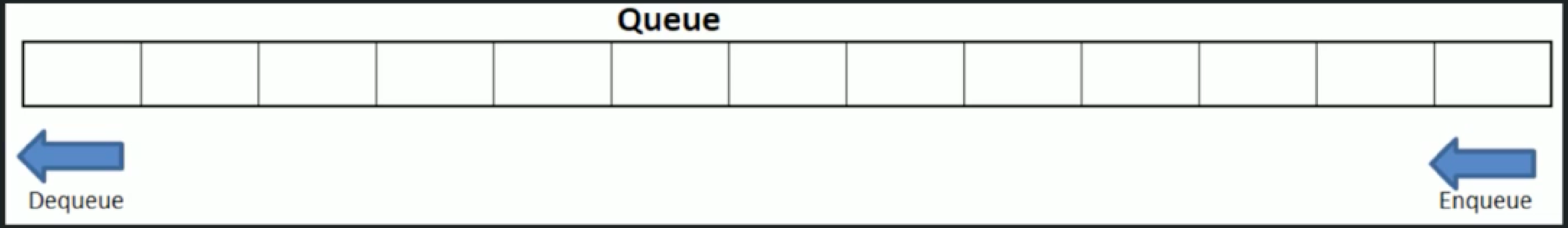

```
insertNodeInBinaryTree():

    if root is blank
        insert new node at root

    else

        do a level order traversal and find the first blank space

        insert in that blank place
```



Time & Space Complexity- Insertion of node in Binary Tree (Linked-List implementation):

insertNodeInBinaryTree():

if root is blank -- O(1)

insert new node at root --- O(1)

else --- O(1)

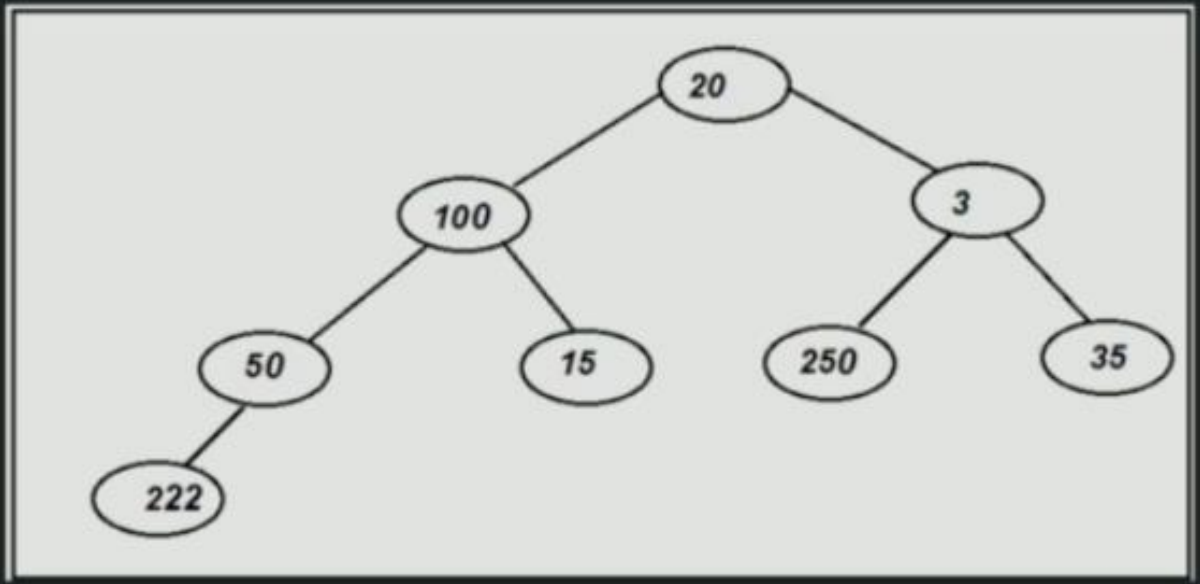
do a level order traversal and find the first blank space ------------------------------ O(n)

insert in that blank place -- O(1)

Time Complexity – O(n)

Space Complexity – O(n)

Deletion of node from Binary Tree(Linked-List implementation):



- ✓ Deletion:
  - ✓ When the value to be deleted is not existing in the tree
  - ✓ When the value to be deleted exists in the tree

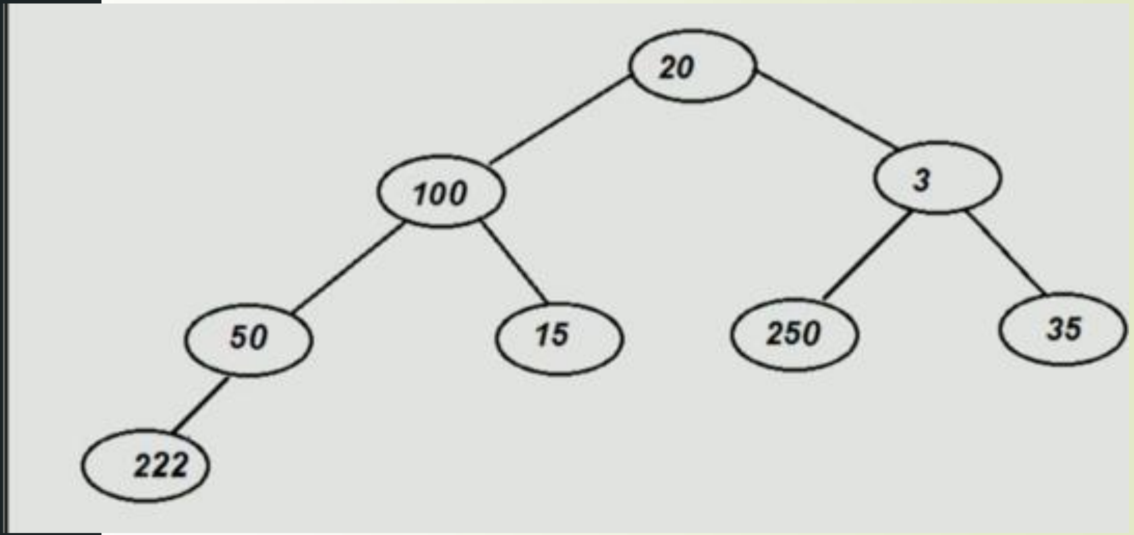
deleteNodeFromBinaryTree()

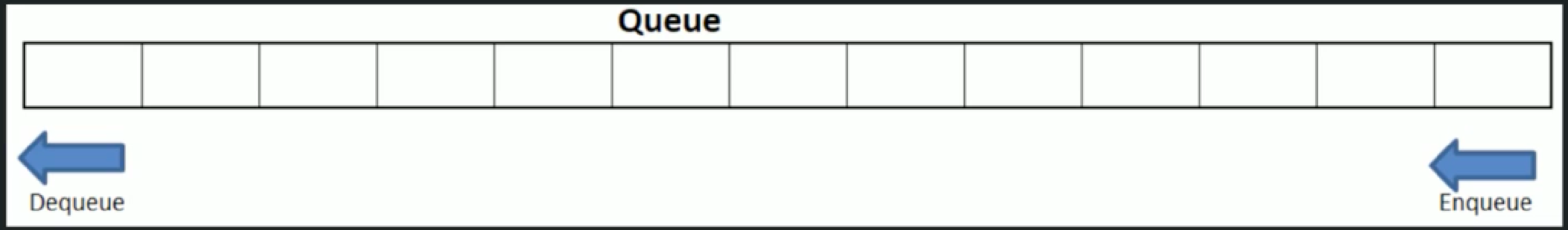
search for the node to be deleted

find deepest node in the tree

copy deepest node's data in current node

delete deepest node





Time & Space Complexity- Deletion of node from Binary Tree(Linked-List implementation):

deleteNodeFromBinaryTree()

search for the node to be deleted -- O(n)

find deepest node in the tree -- O(n)

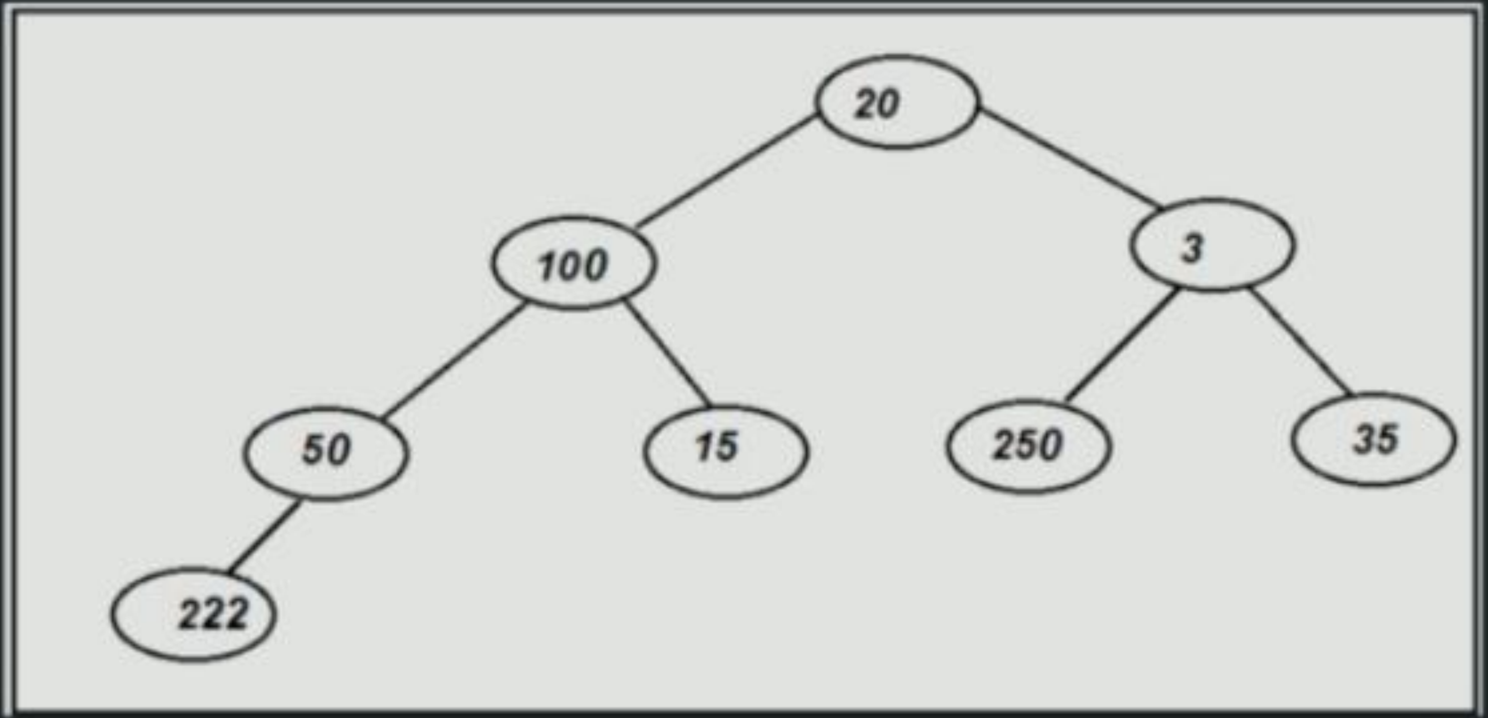
copy deepest node's data in current node ------------------------------ O(1)

delete deepest node -- O(1)

Time Complexity – O(n)

Space Complexity – O(n)

Delete Binary Tree (Linked-List implementation):



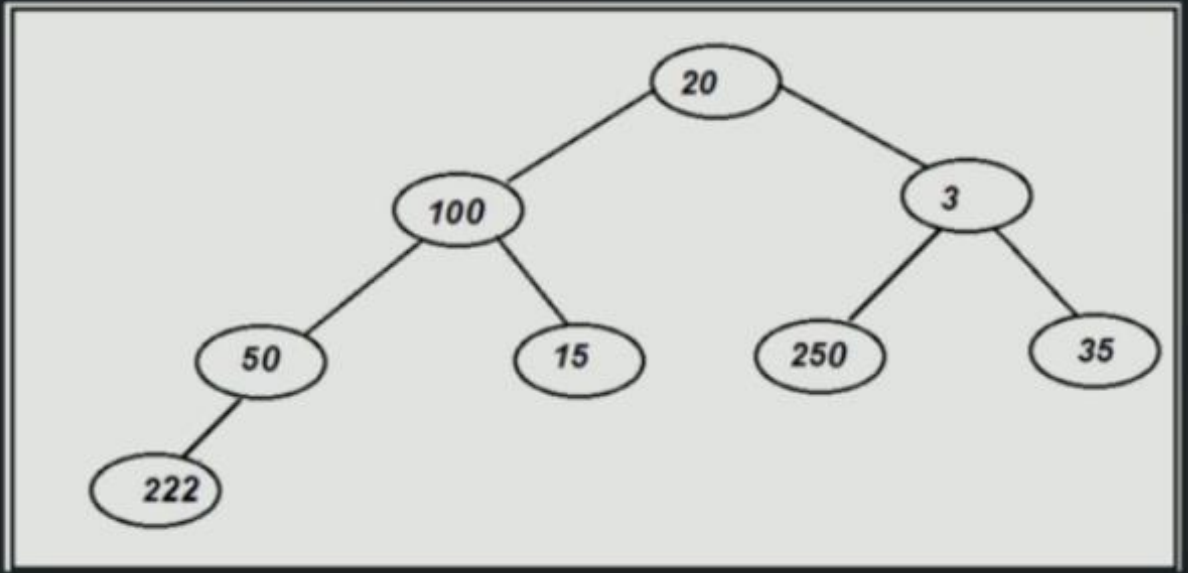
Algorithm-
Delete Binary Tree (Linked-List implementation):

```
DeleteBinaryTree()
    root = null
```

Binary Tree (Array implementation)

Binary Tree - Array Representation:

✓How does tree looks like at logical level ?



✓How does tree looks when implemented via Array:

Cell#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Value		20	100	3	50	15	250	35	222									

Left Child – cell [2x]

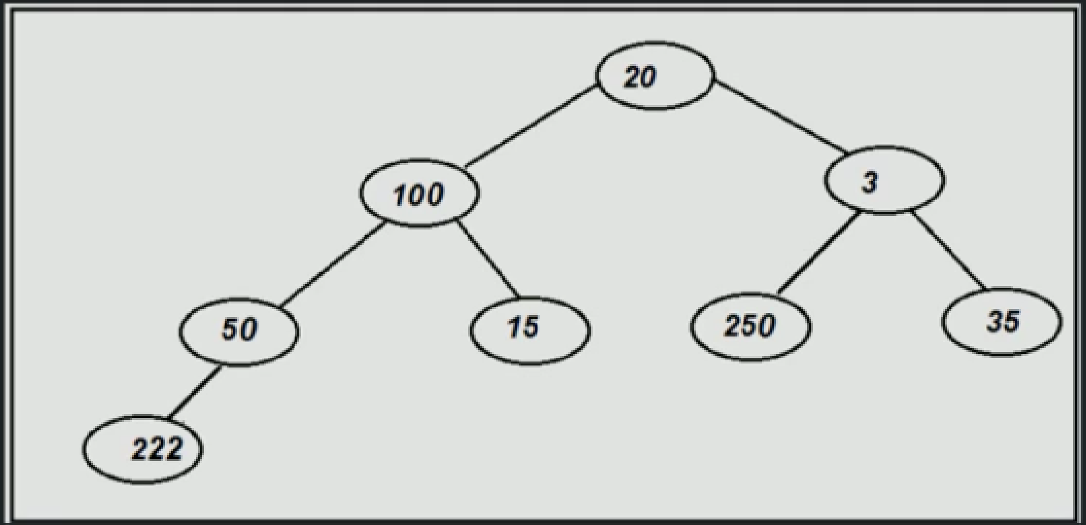
Right Child – cell [2x + 1]

Algorithm – Creation of Binary Tree(Array Implementation):

```
createBinaryTree()
    create a blank array of 'size'
    update lastUsedIndex to 0
```

Cell #	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Value																		

Insertion of node in Binary Tree (Array Implementation):



Cell#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Value		20	100	3	50	15	250	35	222									

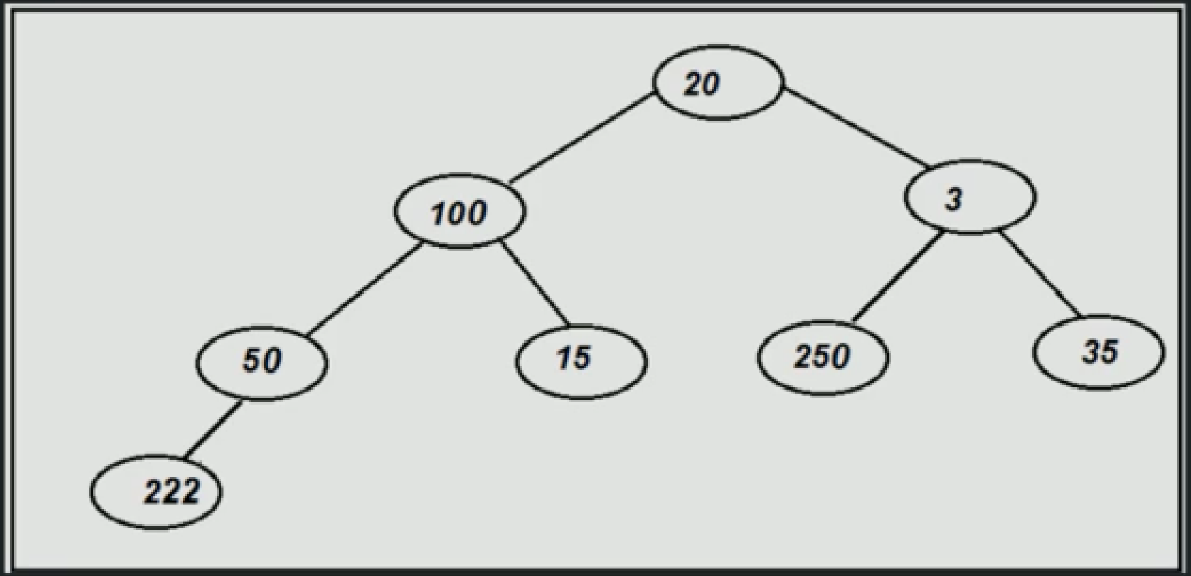
Algorithm - Insertion of node in Binary Tree(Array Implementation):

- ✓ *Insertion:*
  - ✓ *If array is full, return error message*
  - ✓ *Insert at first vacant cell in array*

```
insertValueinBinaryTree()
    if Tree is full
        return error message
    else
        insert value in first unused cell of array
        update lastUsedIndex
```

Cell#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Value		20	100	3	50	15	250	35	222									

Searching a node in Binary Tree(Array Implementation):



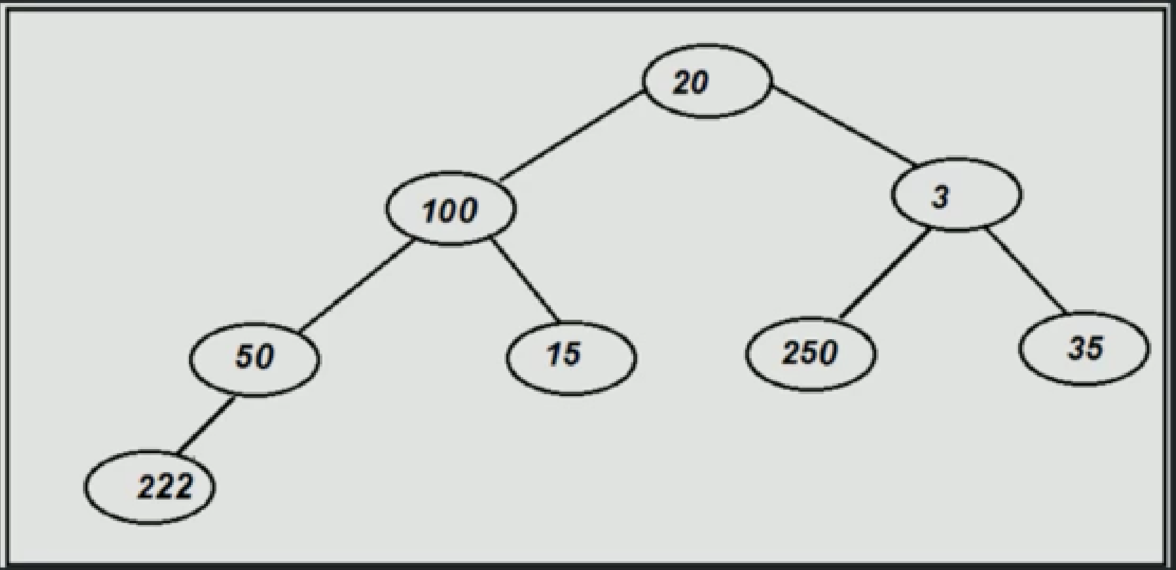
Cell#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Value		20	100	3	50	15	250	35	222									

Algorithm - Searching a node in Binary Tree(Array Implementation):

- ✓ Search:
  - ✓ When the value to be searched does not exists in the tree
  - ✓ When the value to be searched exists in the tree

```
searchValueInBinaryTree()
    traverse the entire array from 1 to lastUsedIndex
    if value is found
        return success message
    return error message
```

Traversing all nodes of Binary Tree(Array Implementation):



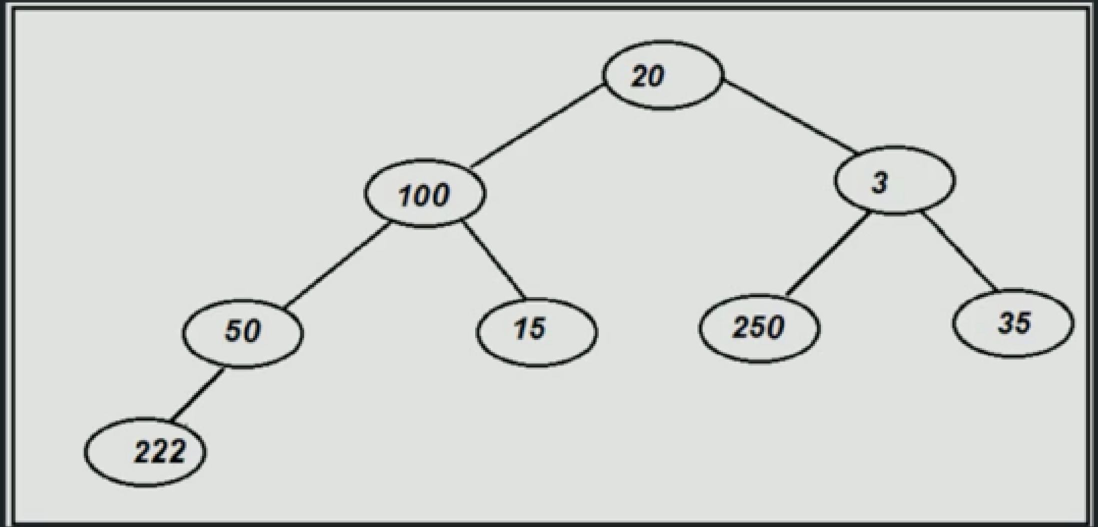

```
InorderTraversal (index)
    if index > lastUsedIndex
        return
    else
        InorderTraversal(index*2)
        print current index.value
        InorderTraversal(index*2+1)
```

Cell#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Value		20	100	3	50	15	250	35	222									

- ✓Depth First Search:
  - ✓ PreOrder Traversal
  - ✓ InOrder Traversal
  - ✓ PostOrder Traversal
- ✓Breadth First Search:
  - ✓ LevelOrder Traversal

Algorithm(Pre-Order Traversal) of Binary Tree(Array Implementation):

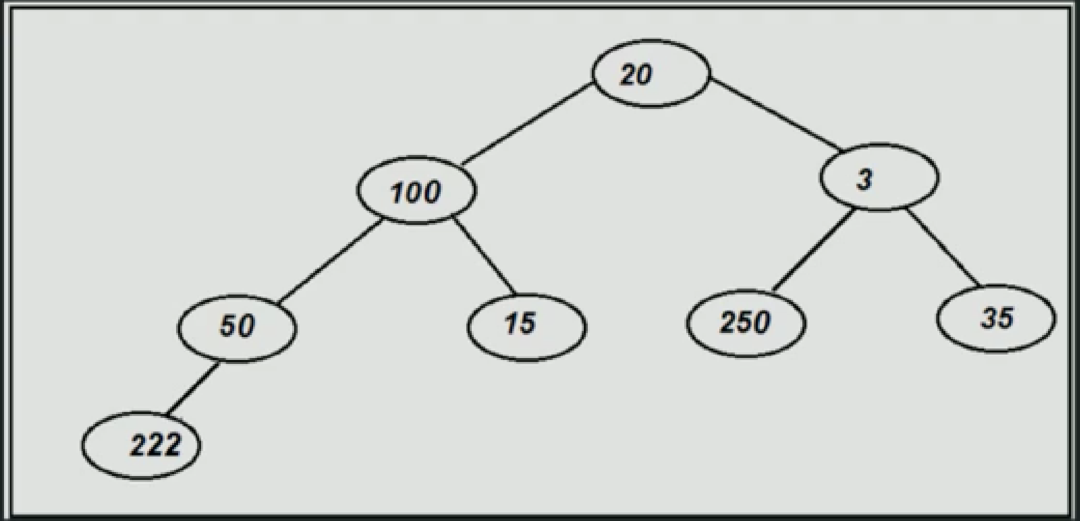
```
preOrderTraversal(index)
  if index > lastUsedIndex
    return
  else
    print current index.value
    preOrderTraversal(index*2)
    preOrderTraversal(index*2+1)
```



Cell#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Value		20	100	3	50	15	250	35	222									

Algorithm(Post-Order Traversal) of Binary Tree(Array Implementation):

```
postOrderTraversal(index)
    if index > lastUsedIndex
        return
    else
        postOrderTraversal(index*2)
        postOrderTraversal(index*2+1)
        print current index.value
```



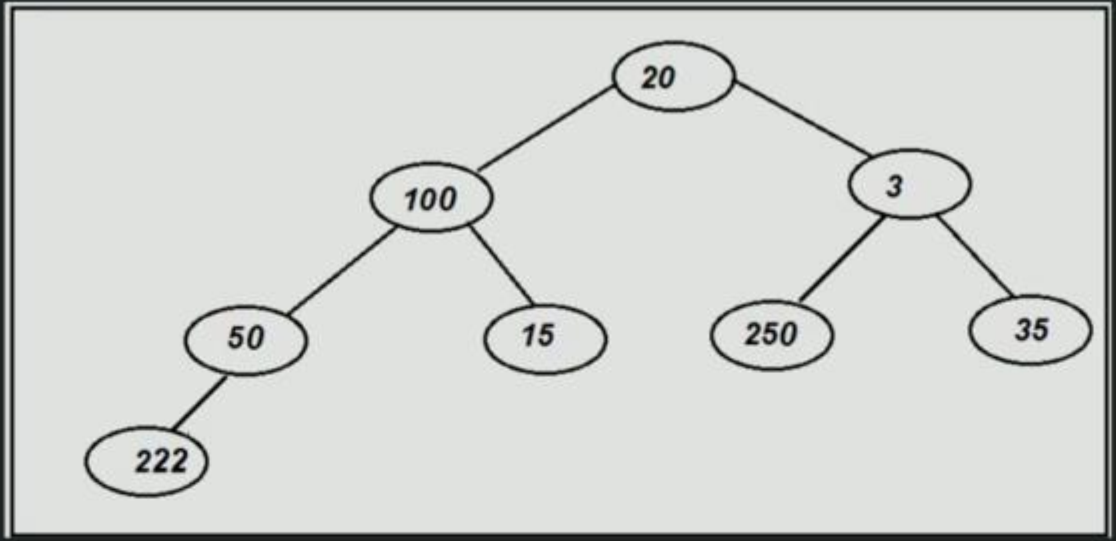
Cell#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Value		20	100	3	50	15	250	35	222									

Algorithm(Breadth First / Level Order Traversal) of Binary Tree(Array Implementation):

levelOrderTraversal()

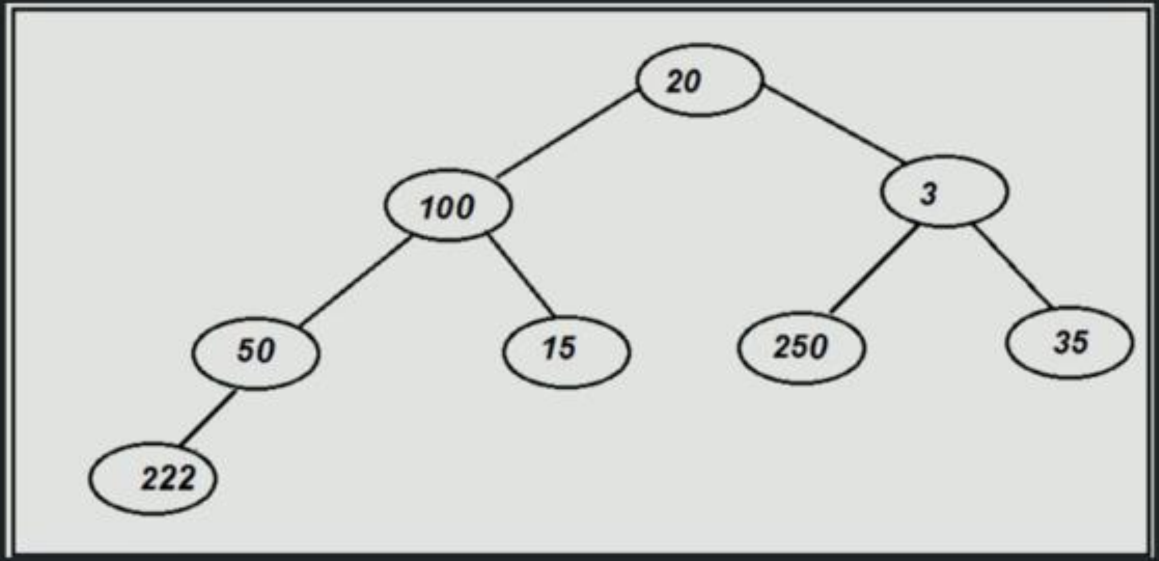
loop: 1 to lastUsedIndex

print current index.value



Cell#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Value		20	100	3	50	15	250	35	222									

Deletion of node from Binary Tree(Array Implementation):



Cell#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Value		20	100	3	50	15	250	35	222									

Algorithm - Deletion of node from Binary Tree(Array Implementation):

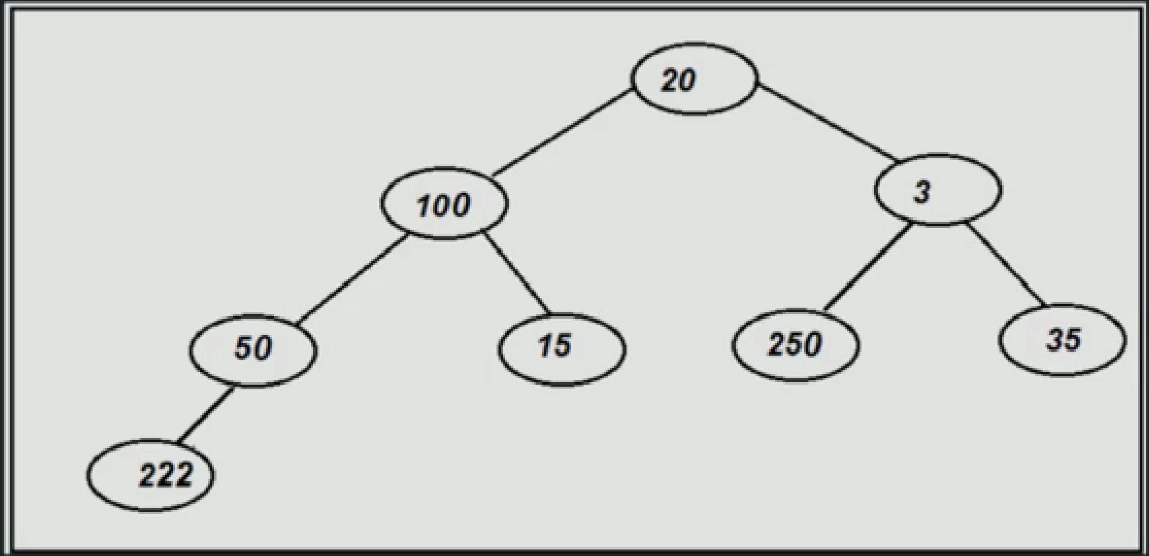
deleteNodeFromBinaryTree()

search for desired value in array

if value found

replace this cell's value with last cell and update lastUsedIndex

return error message



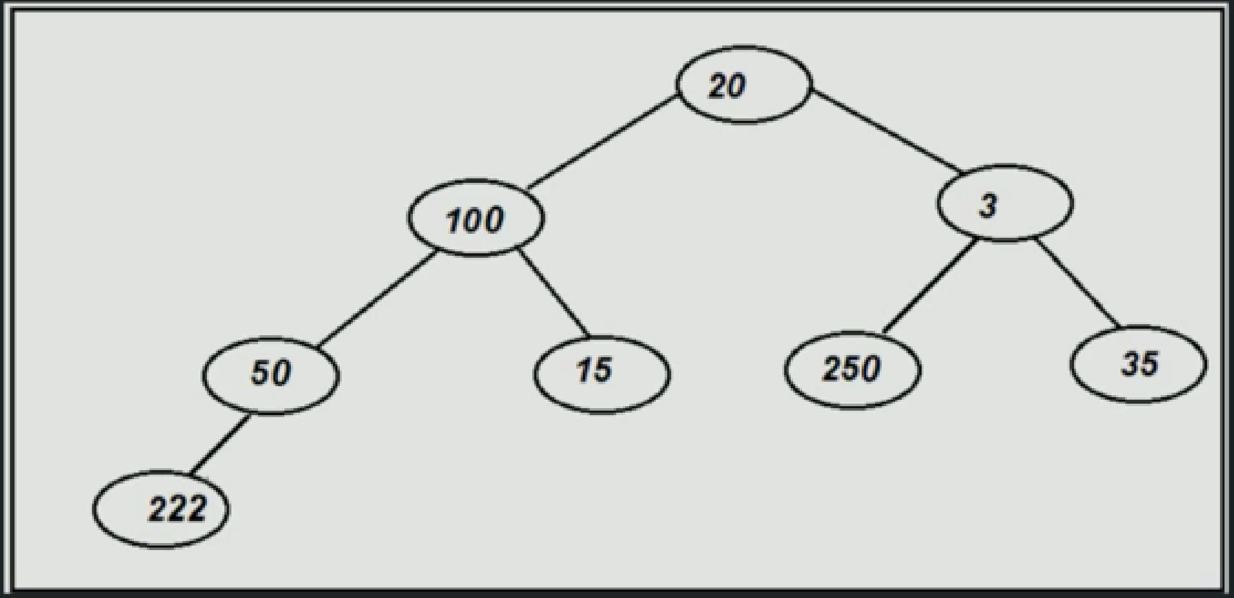
✓ Deletion:

- ✓ When the value to be deleted is not existing in the tree
- ✓ When the value to be deleted is exists in the tree

Algorithm - Delete Binary Tree (Array implementation):

deleteBinaryTree()

set array as null



Cell#	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Value		20	100	3	50	15	250	35	222									

Thank you