Tree Sort

Group No:24

Introduction & History:

Tree sort is invented by Robert W. Floyd in 1964.Tree sort algorithm does sort operation on binary search tree nodes. At first, it creates a binary search tree with all inputs and then in orderly traverses the tree so that the elements come out in sorted order. It follows the divide and conquers paradigm. It is equivalent to quicksort as both recursively partition the element based on a pivot. The best and average time complexity of tree sort is O(nlogn) and worst case is O(n^2). The worst-case appears when the algorithm operates on an already sorted set, or almost sorted inputs set. But the worst case can be optimized from O(n^2) to O(nlogn)by using a self-balancing binary search tree. Tree sort algorithm needs separate heap memory allocation for making binary search tree that generate significant performance. The space complexity of tree sort is O(n) as every individual element of n set represents a node.

Original Pseudocode:

```
begin Integer i, 5; array m(I:2 X n - II;
for i :w I step 1 until n do min + f - I) :- path (UNSORTED
  III, a + I - I);
for i      a - I step - I until I do mill          minimum (mi2 X
  as(2 X i +II);
for j :le I step I until* do
  begin SORTED ui :m left half (mil)); i        nighthay (HOD;
    miff ;. infinity;
  for i i+ 2 while > 0 do mill:- minimum frni2 X IL m12 X
    i+I))
  end
end TREESORT
```

Easy Representation of Original Pseudocode

Algorithm 1: *insert(Node node, Key value)*: Inserts a new node in the BST

```
Data: node: The input node object
      value: The value of node key
Result: Returns the inserted node object
if node = null then
    return Node(value);
end
else if value <= node.value then
    node.left ← insert(node.left, value);
end
else
    node.right ← insert(node.right, value);
end
return node;
```

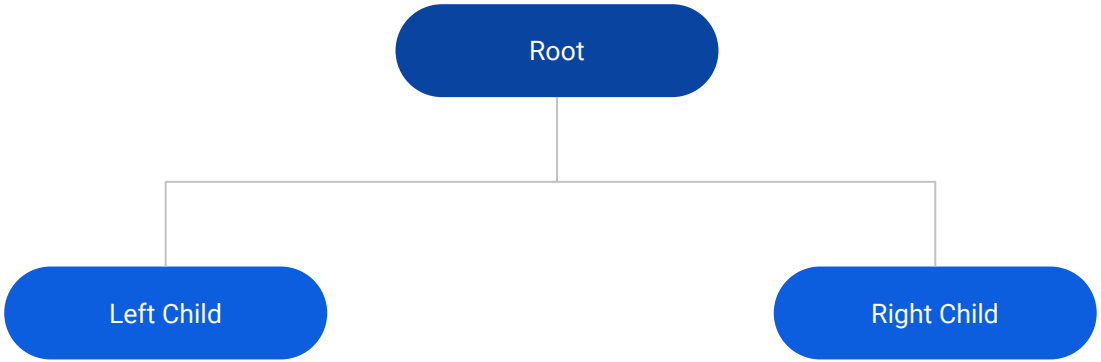
Algorithm 2: *traverseInOrder(Node node)*: Traverses and prints the BST in order

```
Data: node: The input node object
Result: Prints the final BST in ascending order
if node != null then
    traverseInOrder(node.left);
    print(node.value);
    traverseInOrder(node.right);
end
```

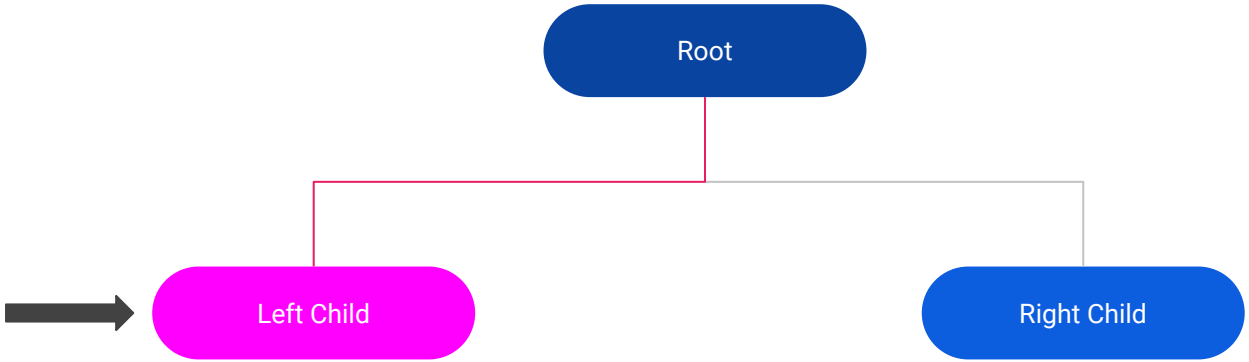
In Order Traverse:

This is a bst traversing technique. It just maintains a sequence. At first traverse the left childs of root then returns in the root and then traverse the right childs of root and returned back in the root. It is done recursively.

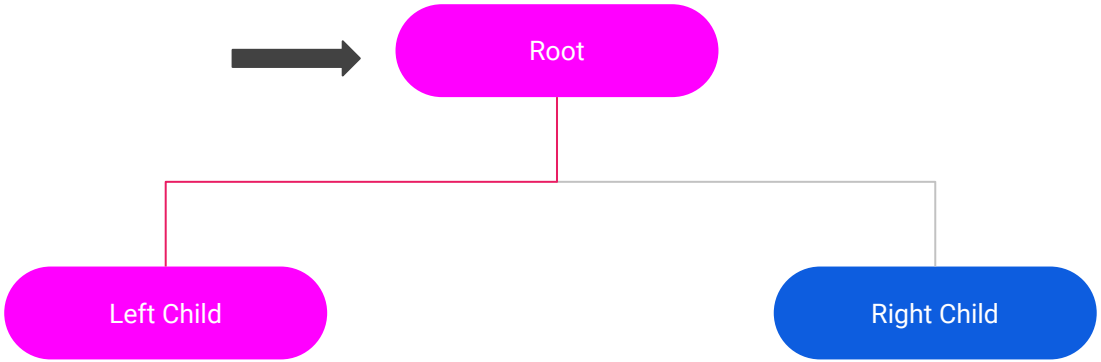
Simulation of In order Traverse:



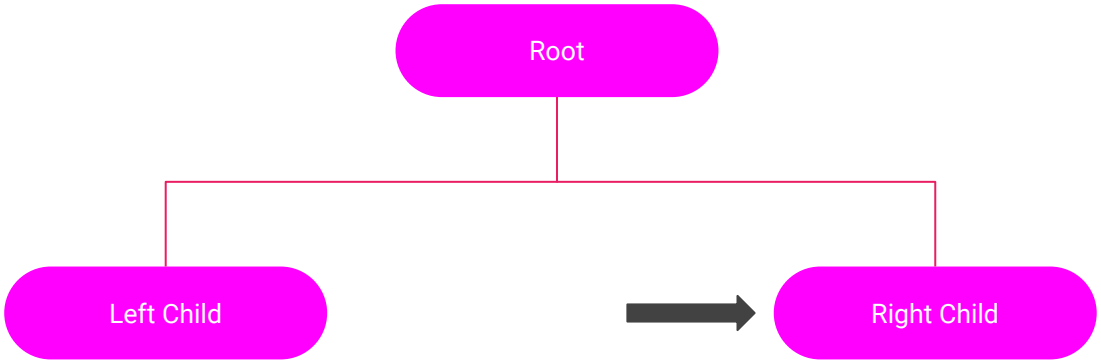
Simulation of In order Traverse:



Simulation of In order Traverse:



Simulation of In order Traverse:

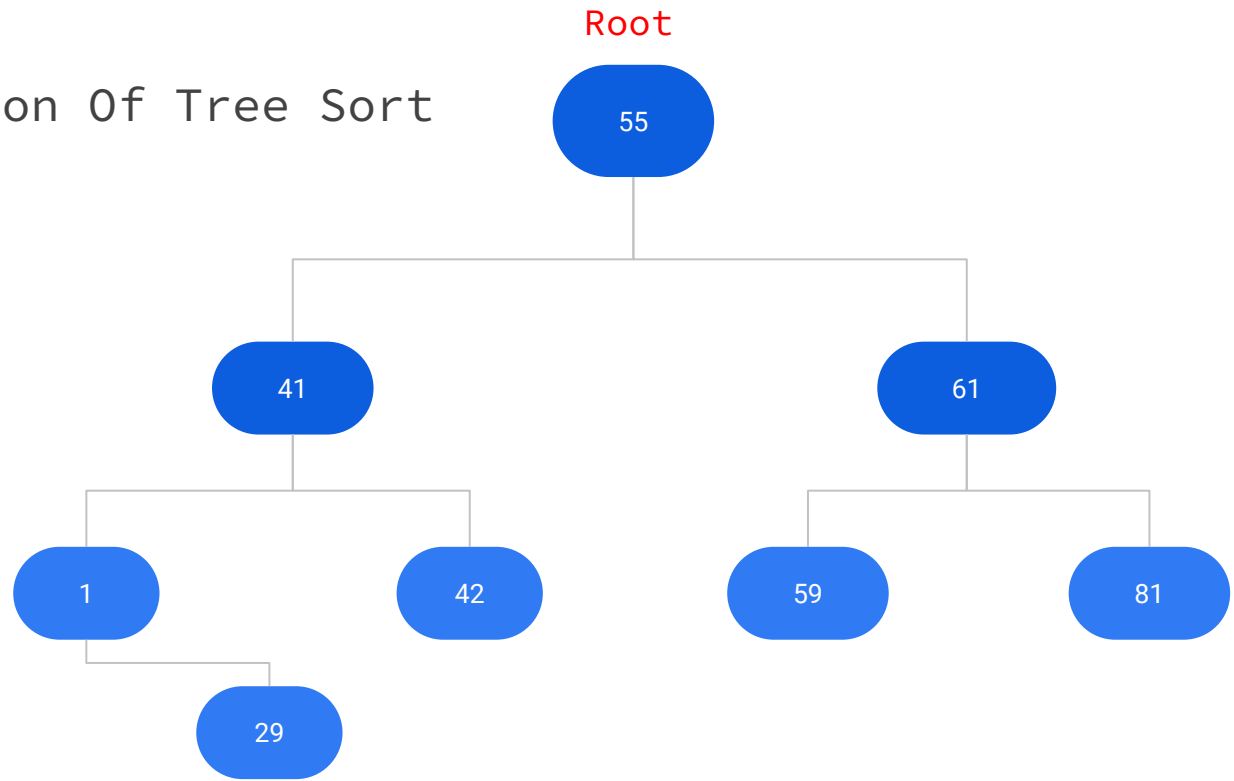


Implementation in C++:

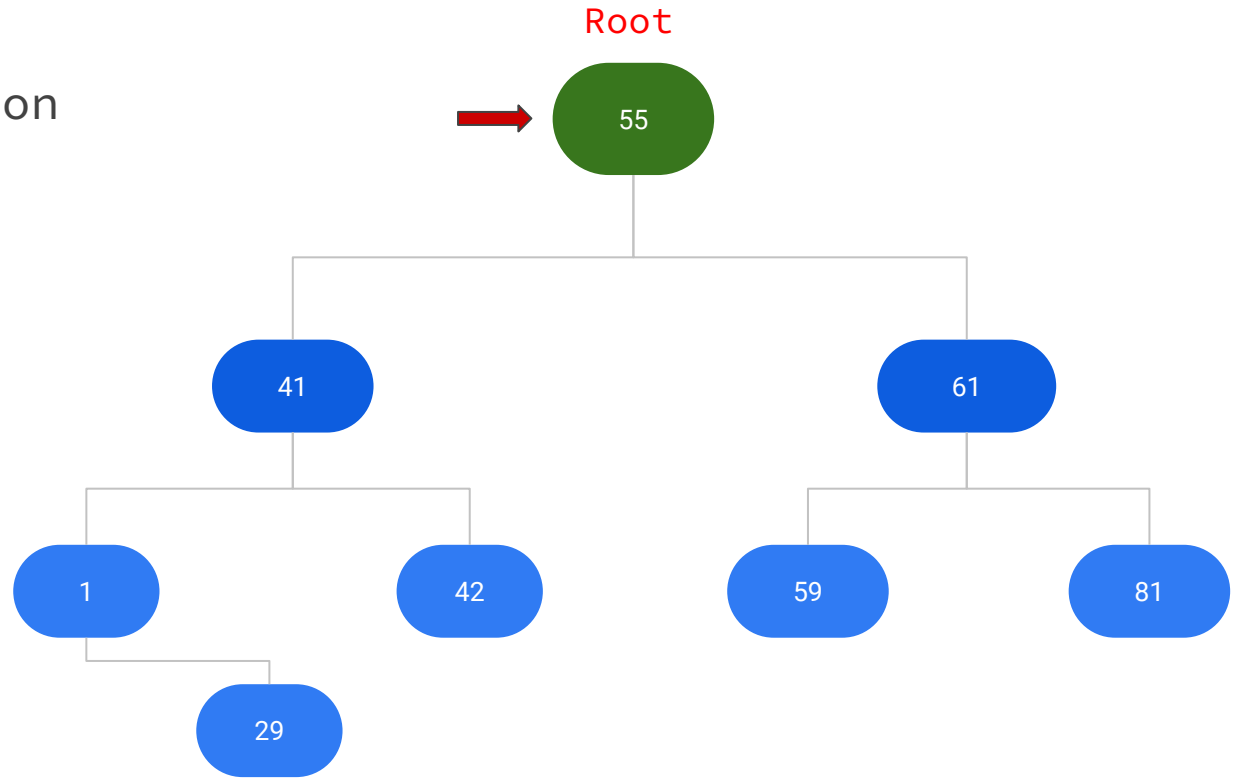
```
struct node{
    int data = 0;
    node *right = nullptr;
    node *left = nullptr;
};
   void tree_sort(node *root)
{
    if (root == nullptr)
    {
       return;
    }

    tree_sort(root->left);
    cout << root->data << " ";
    tree_sort(root->right);
}
```

```
void insert(node *&root, int x){
    if (root == nullptr)
    { root = new node;
     root->data = x;
    }
    else if (x > root->data)
    {
       insert(root->right, x);
    }
else if (x < root->data)
    {
       insert(root->left, x);
    }
}
```

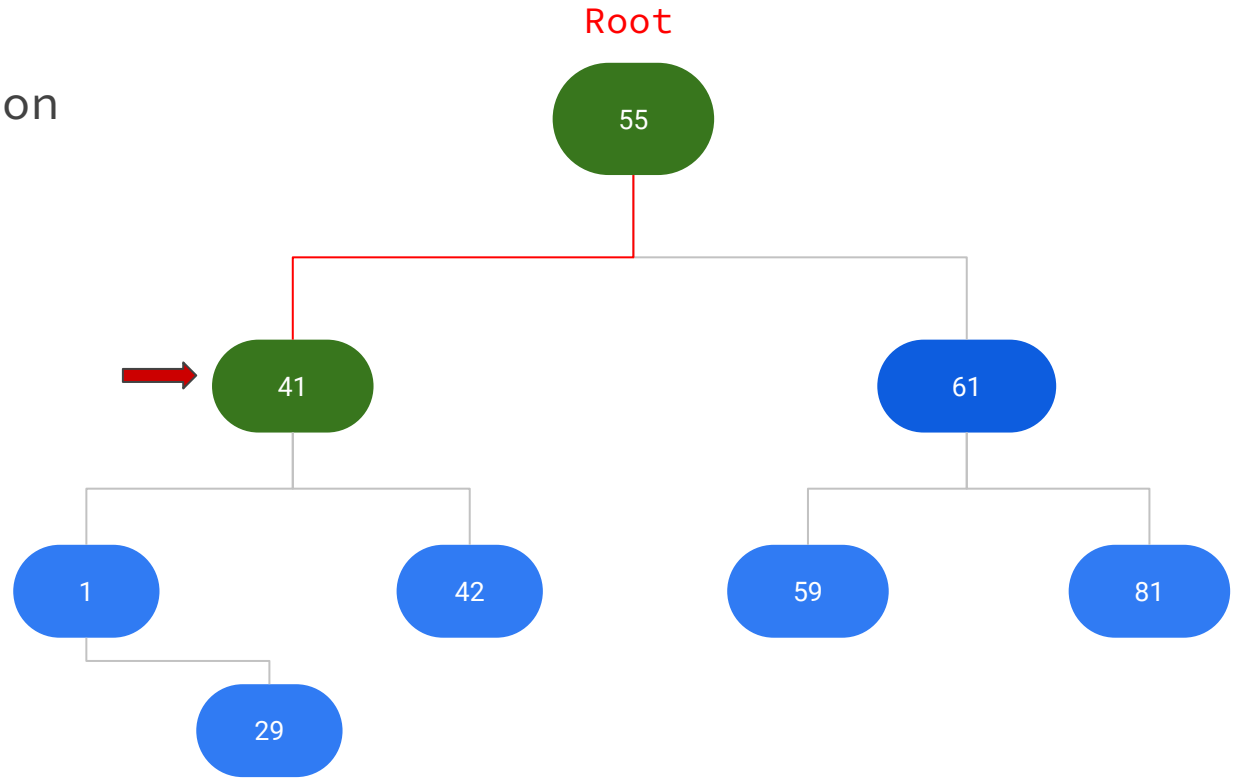
Simulation Of Tree Sort



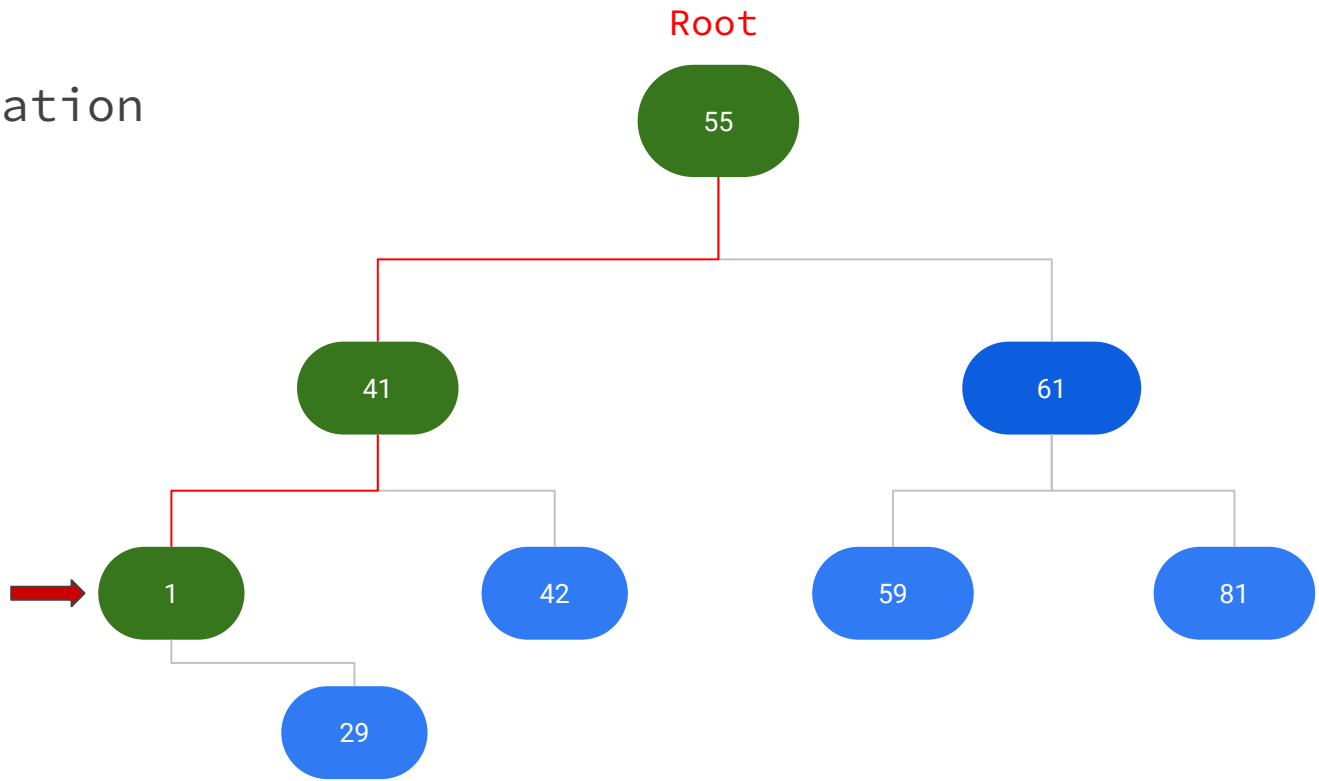
Simulation



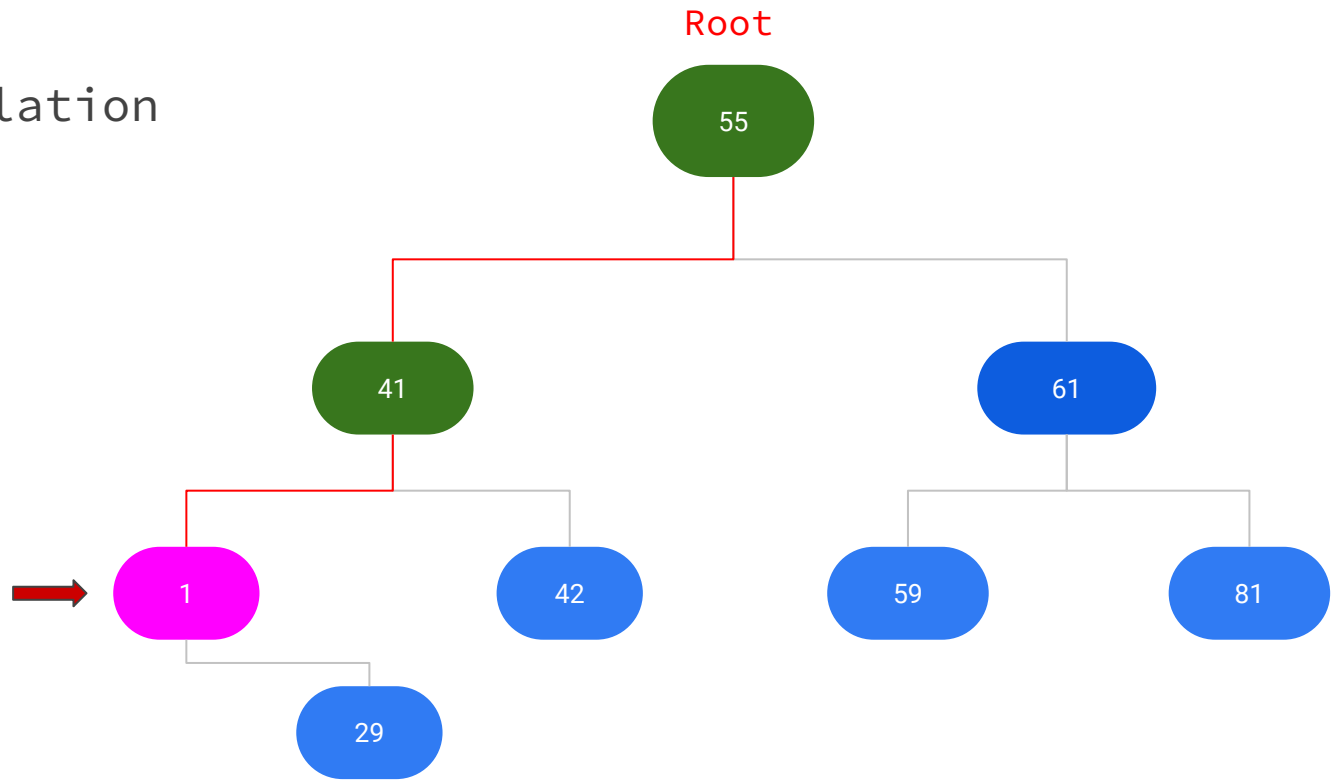
Simulation



Simulation

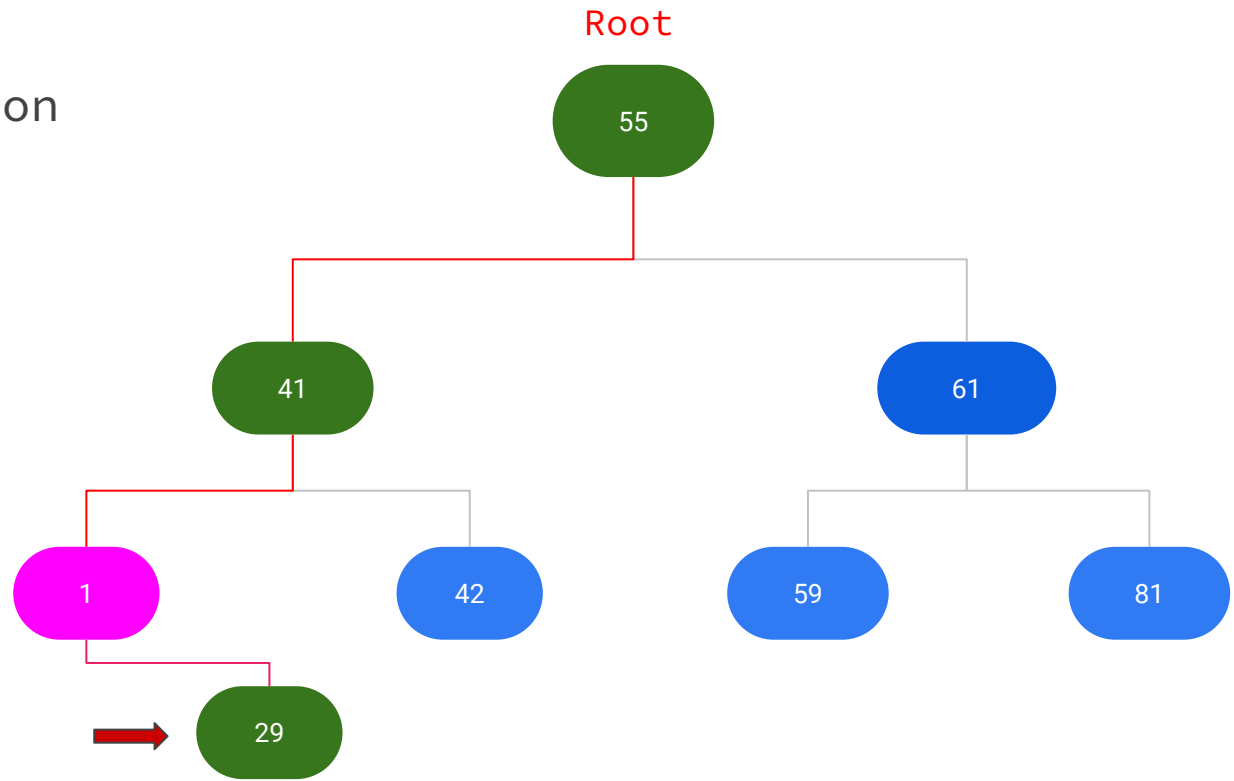


Simulation

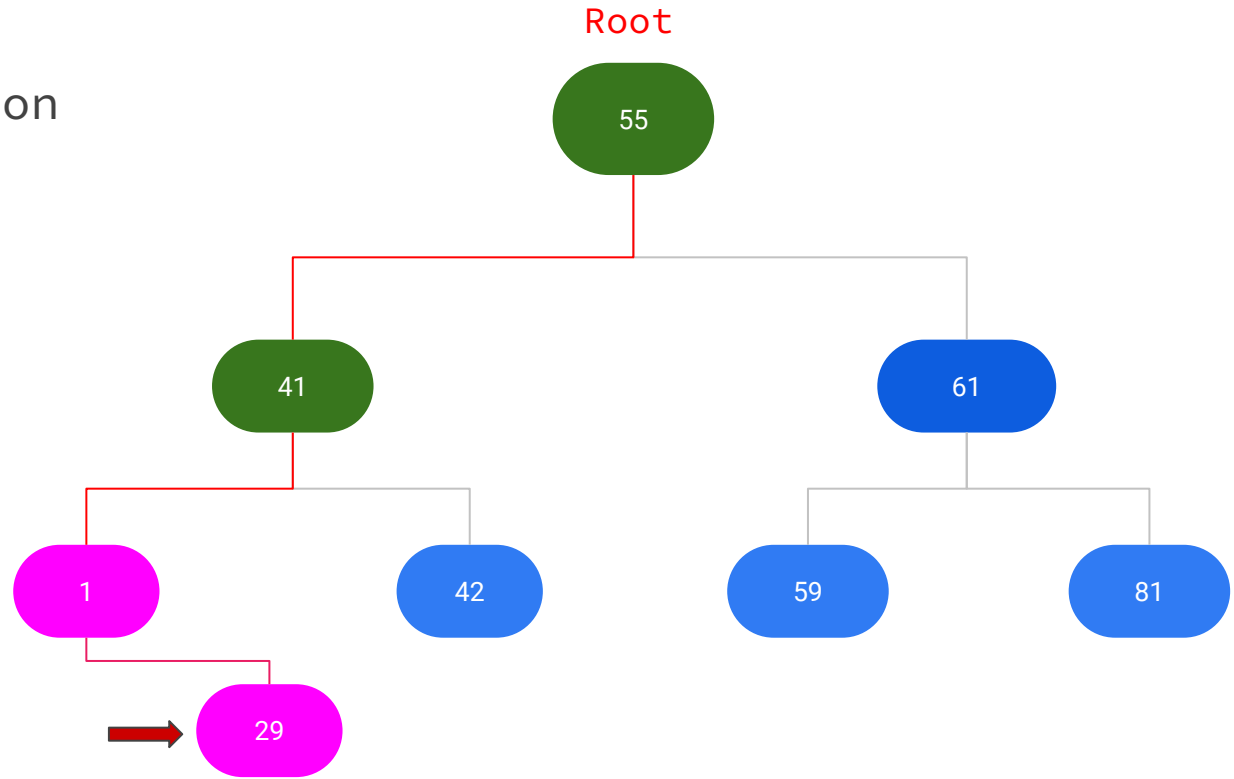


1

Simulation

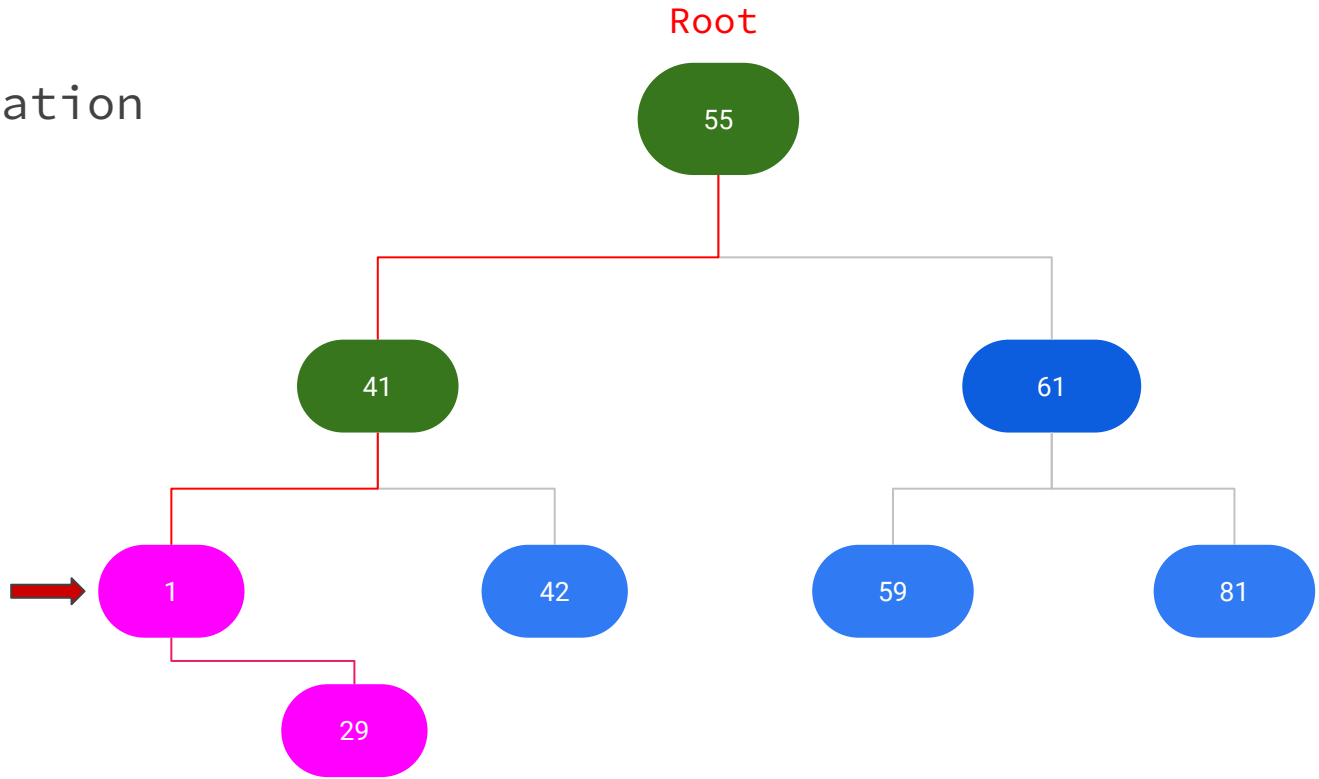


Simulation



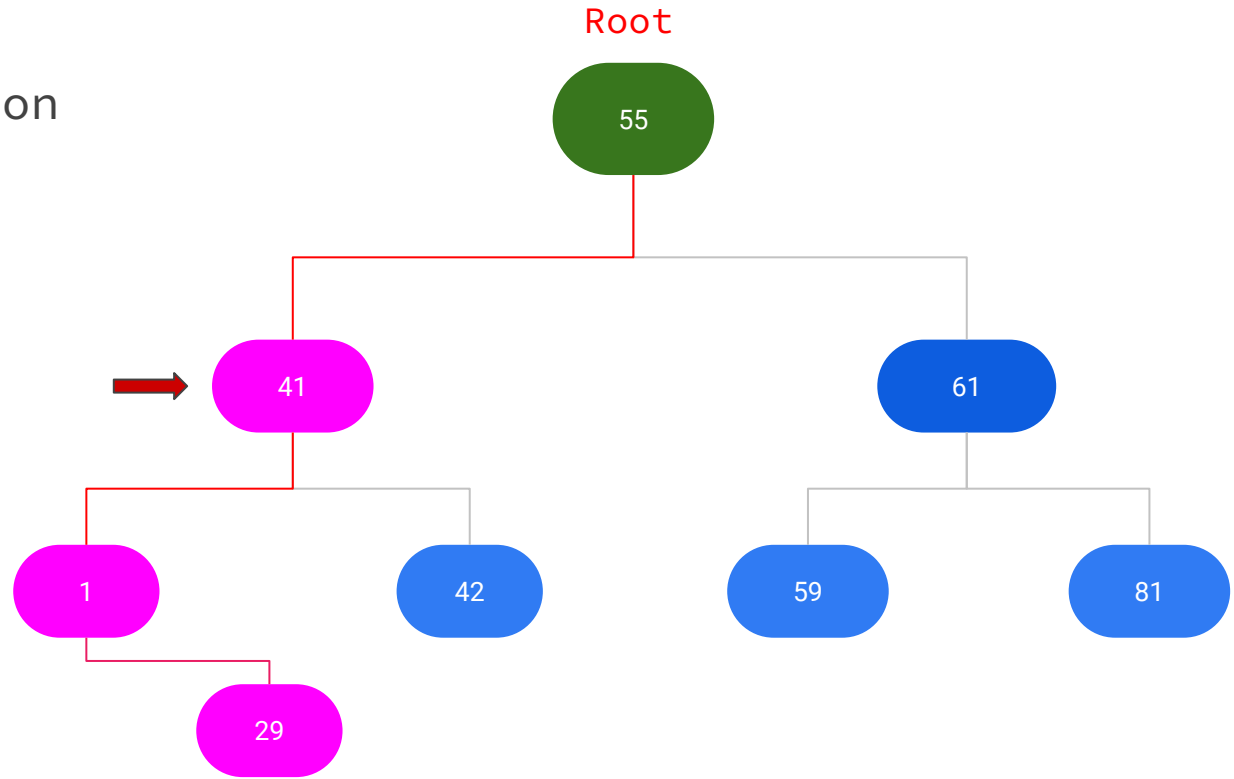
1 29

Simulation



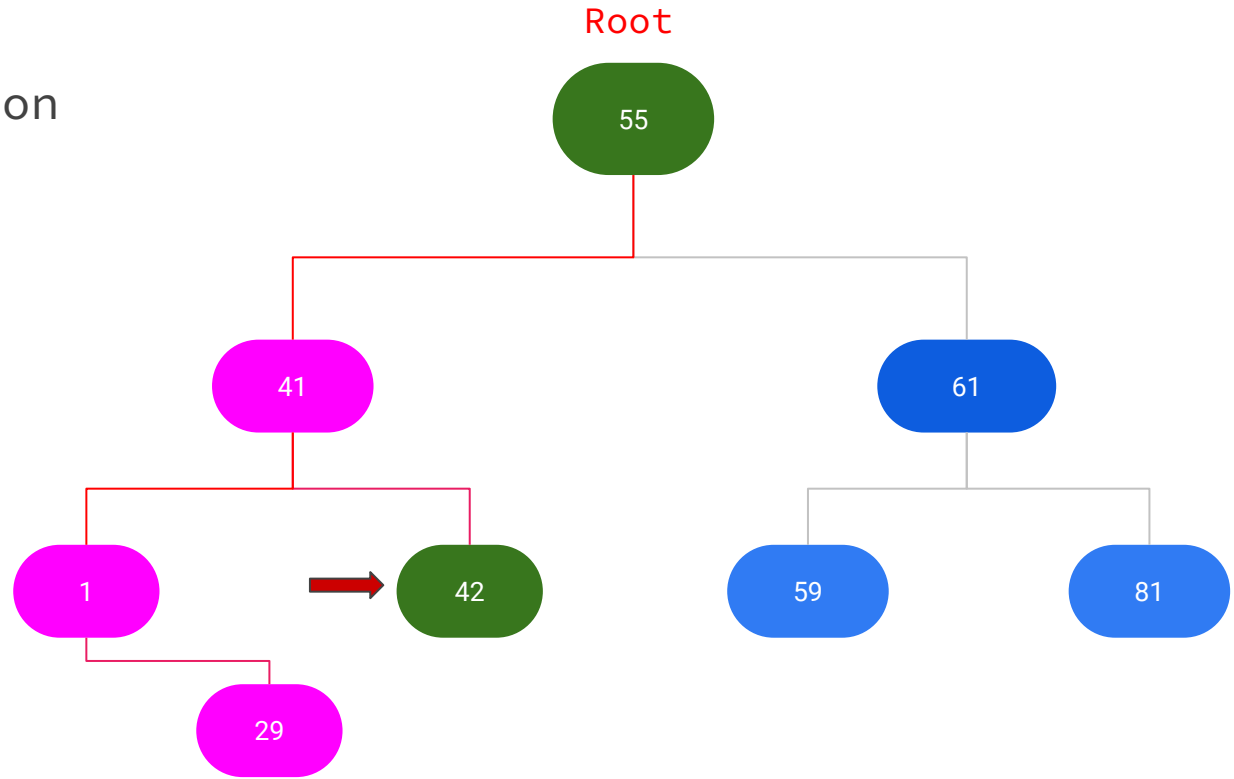
1 29

Simulation



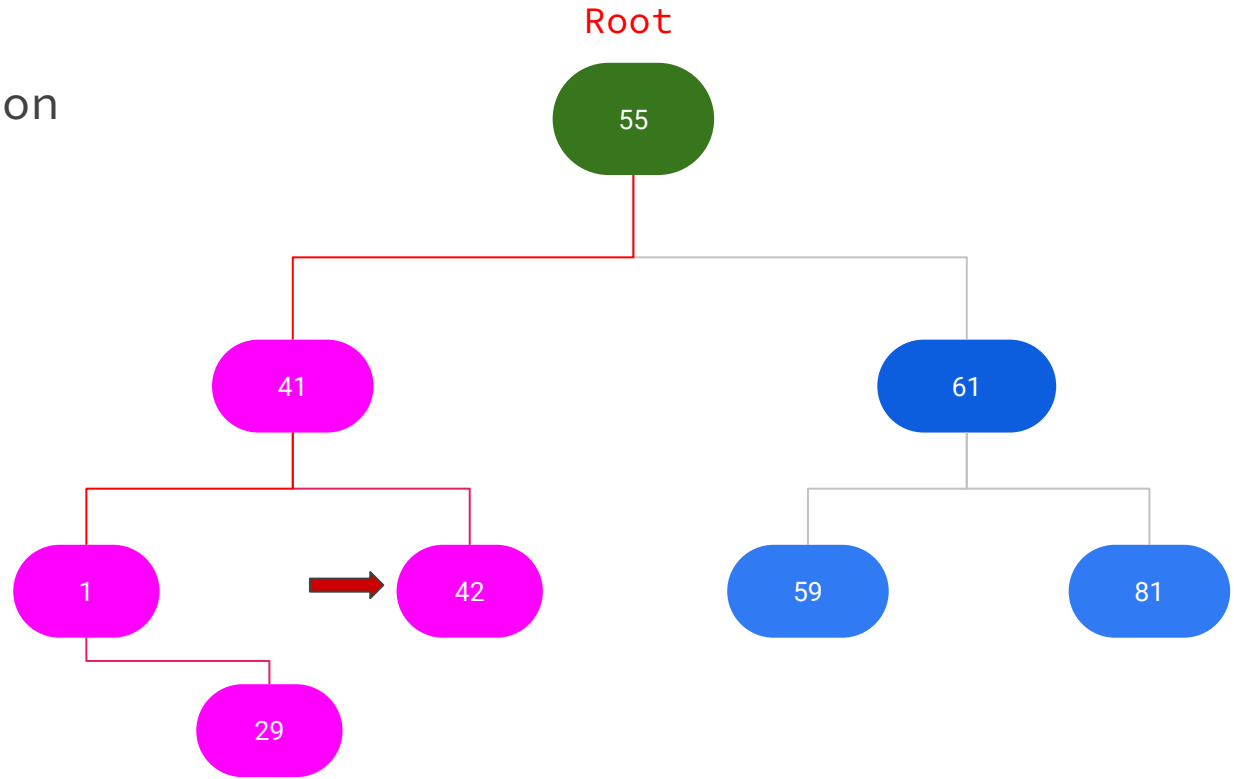
1 29 41

Simulation



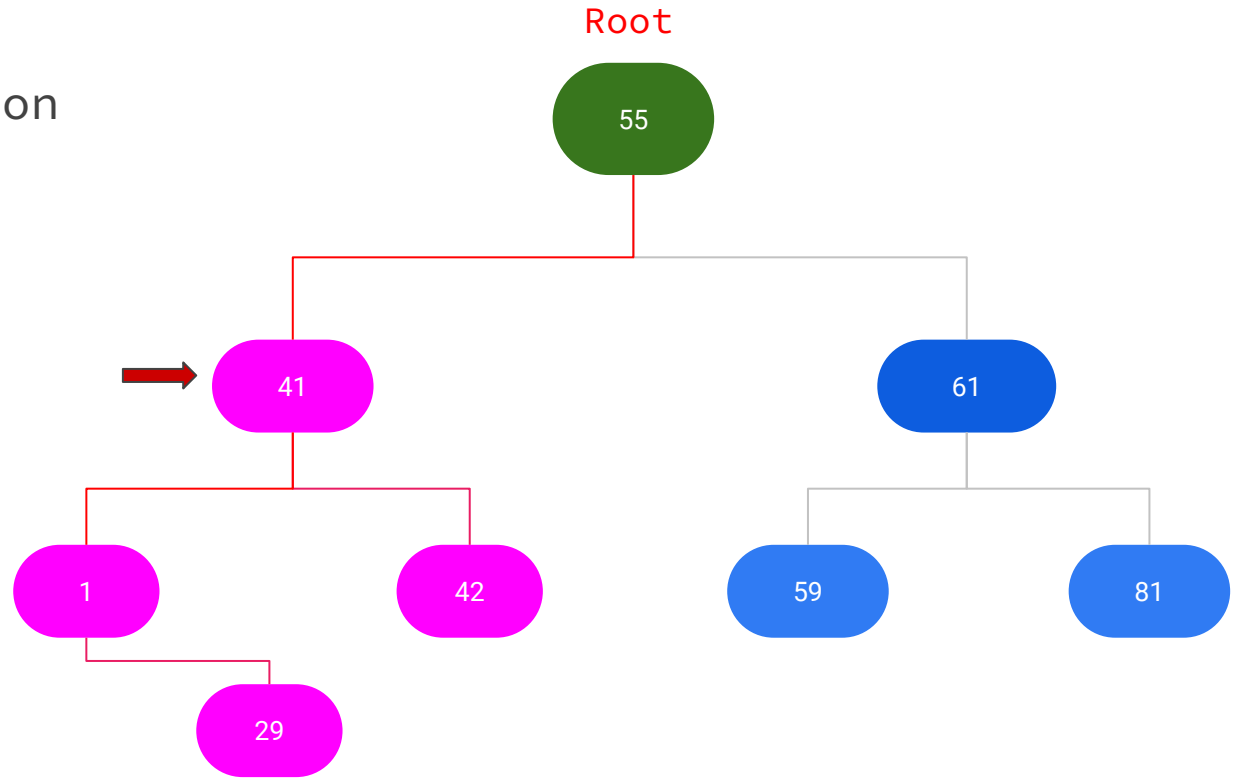
1 29 41

Simulation



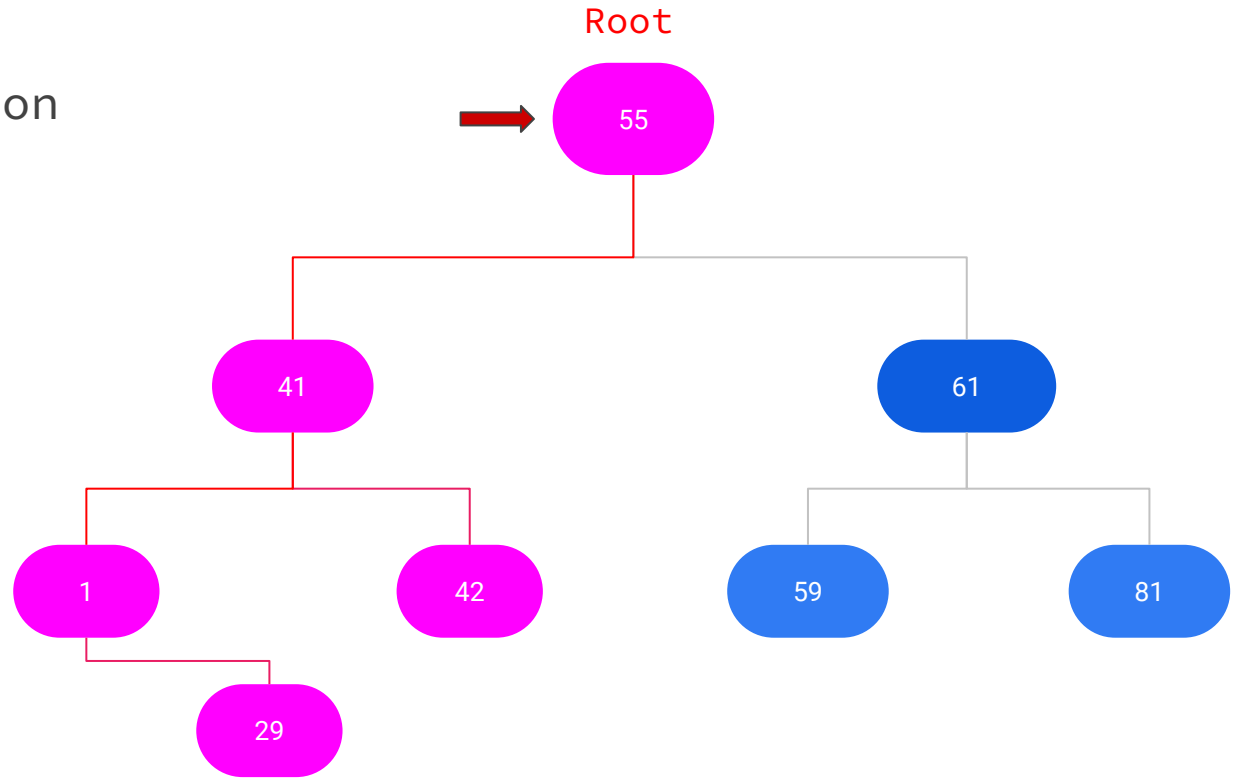
1 29 41 42

Simulation



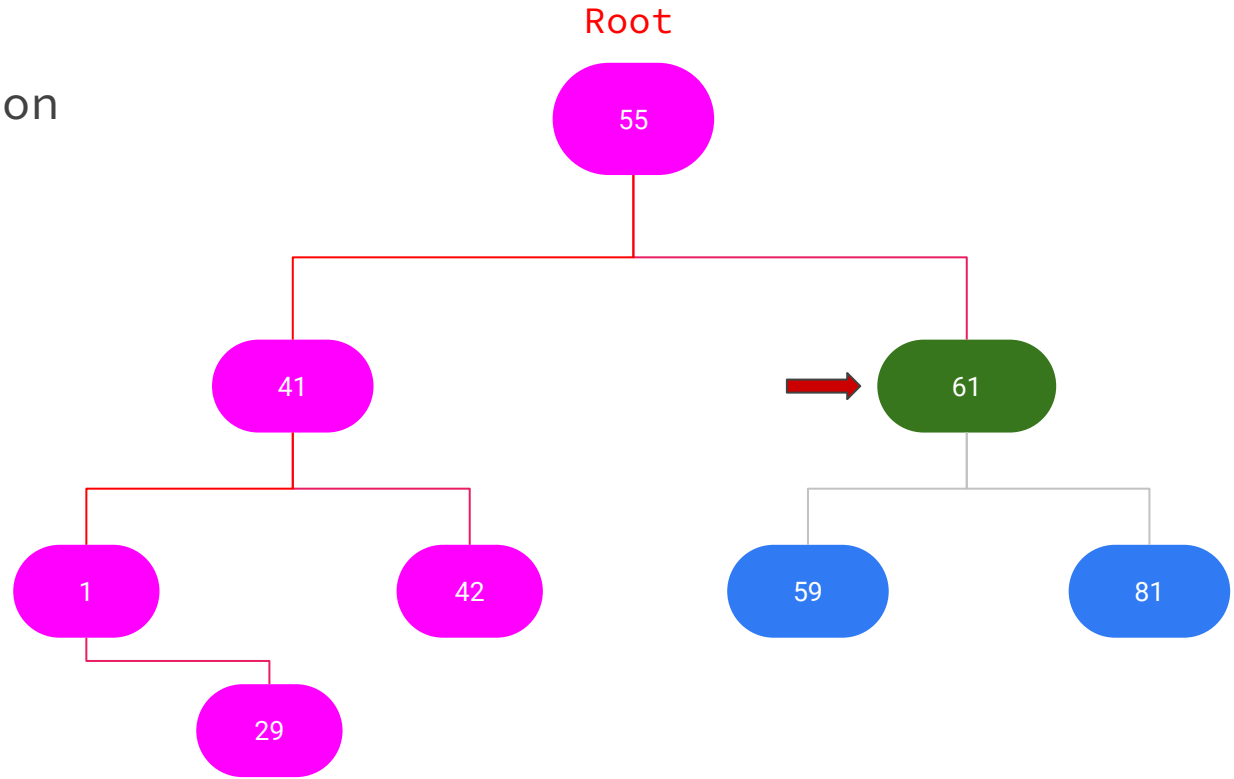
1 29 41 42

Simulation



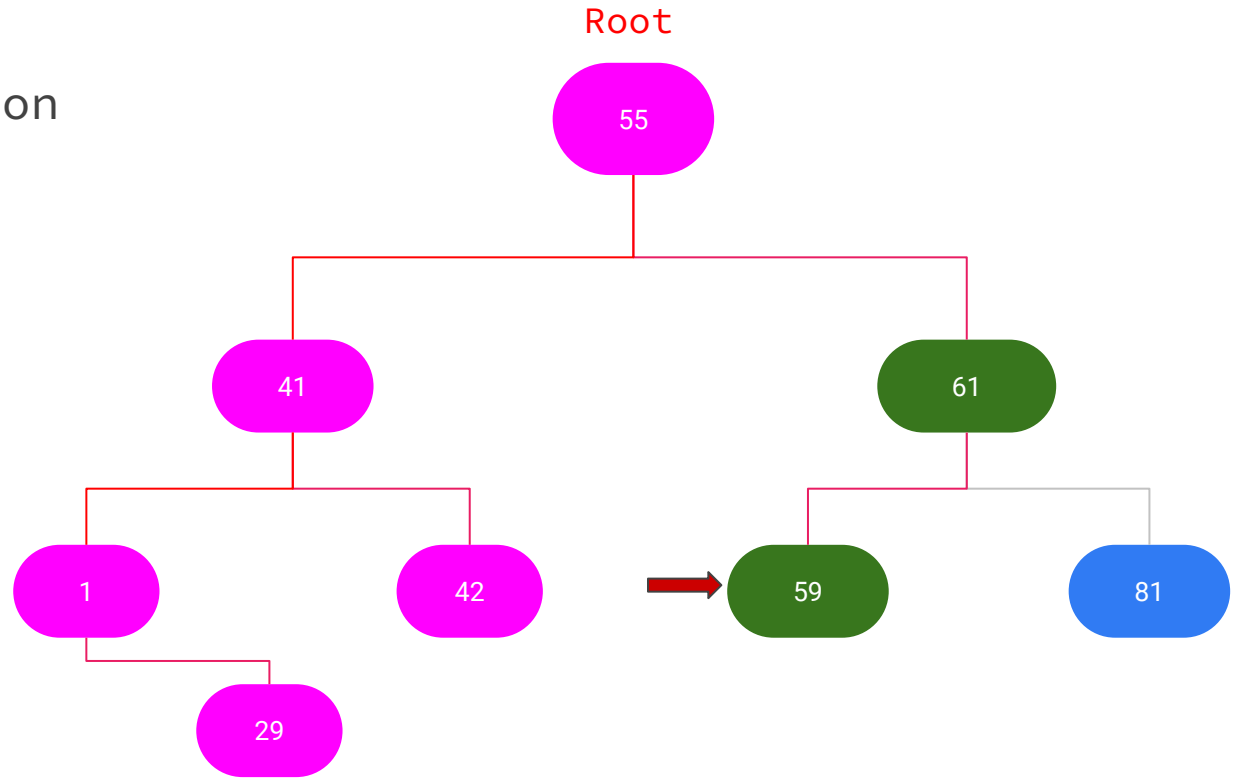
1 29 41 42 55

Simulation



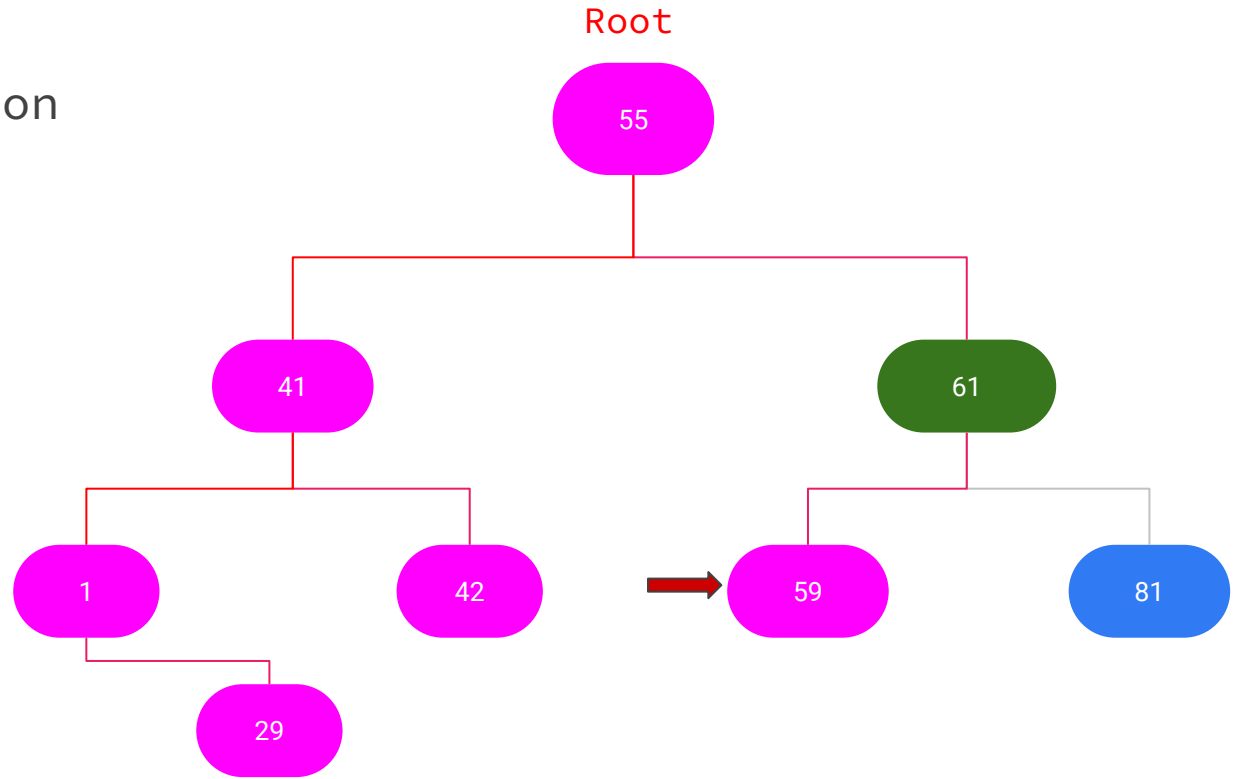
1 29 41 42 55

Simulation



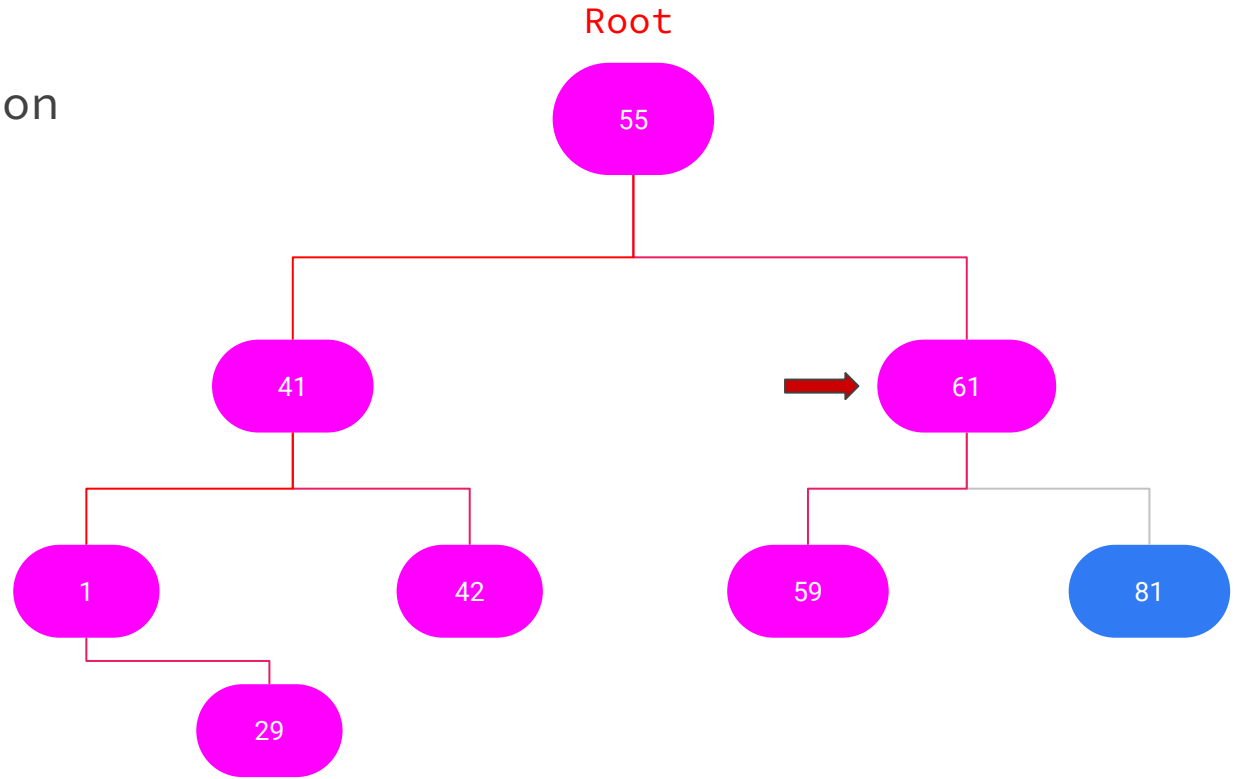
1 29 41 42 55

Simulation



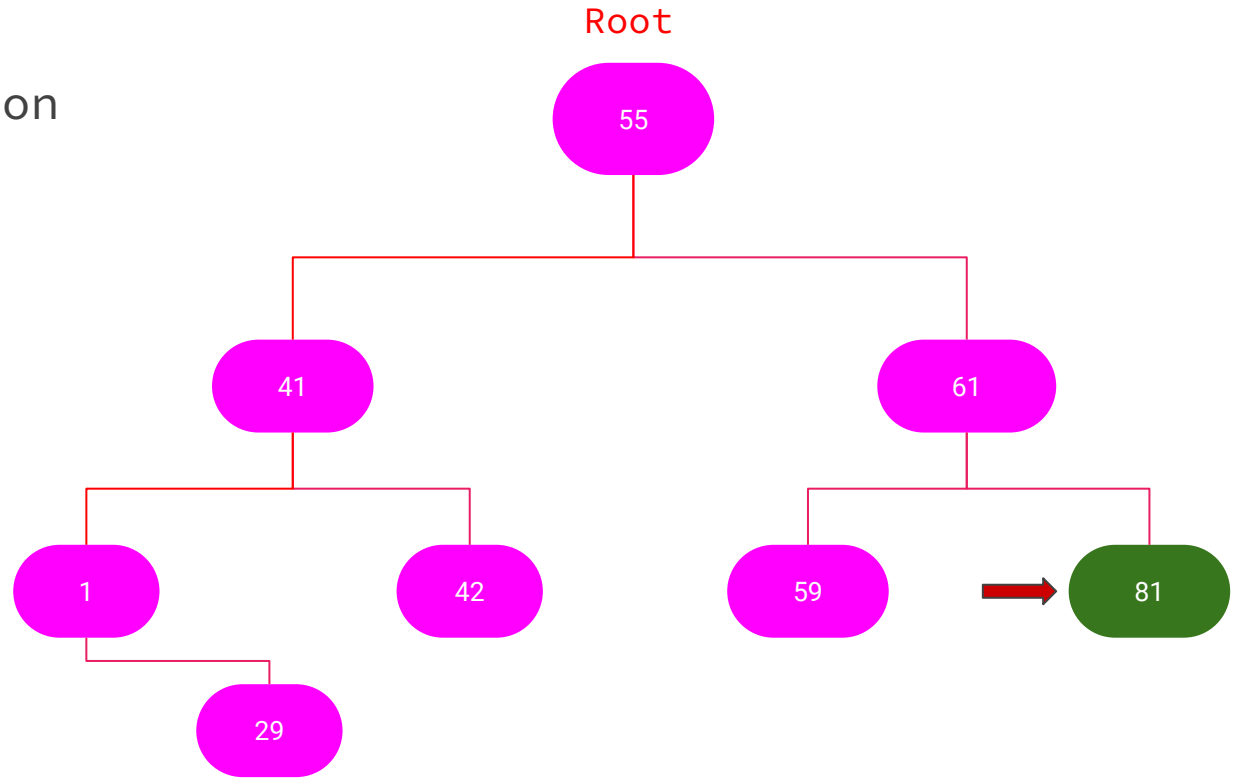
1 29 41 42 55 59

Simulation



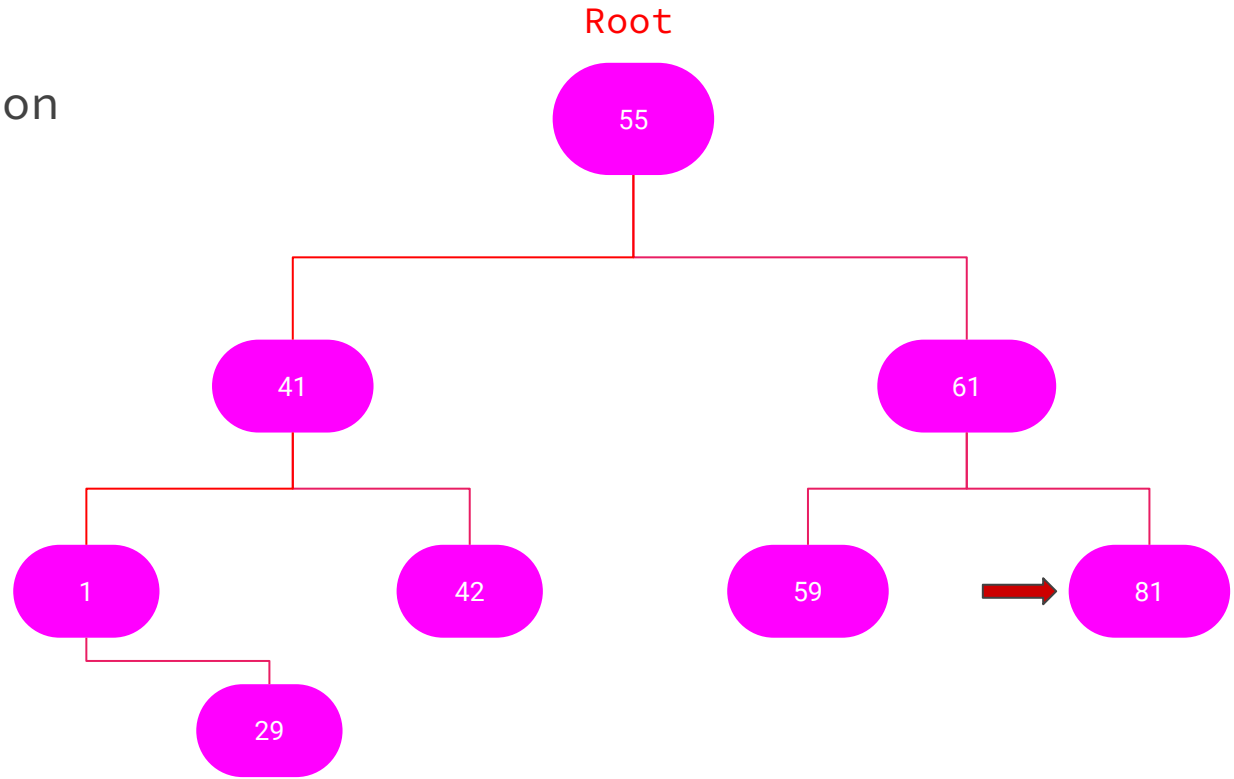
1 29 41 42 55 59 61

Simulation



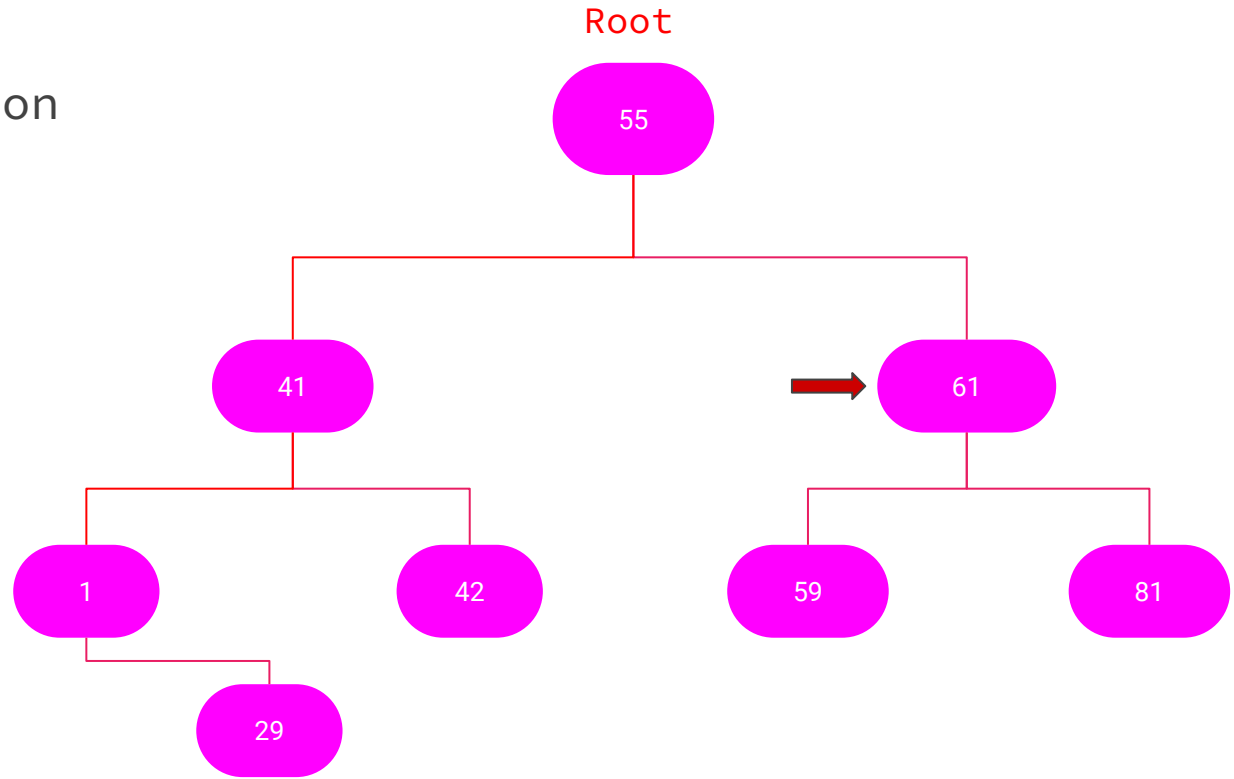
1 29 41 42 55 59 61

Simulation



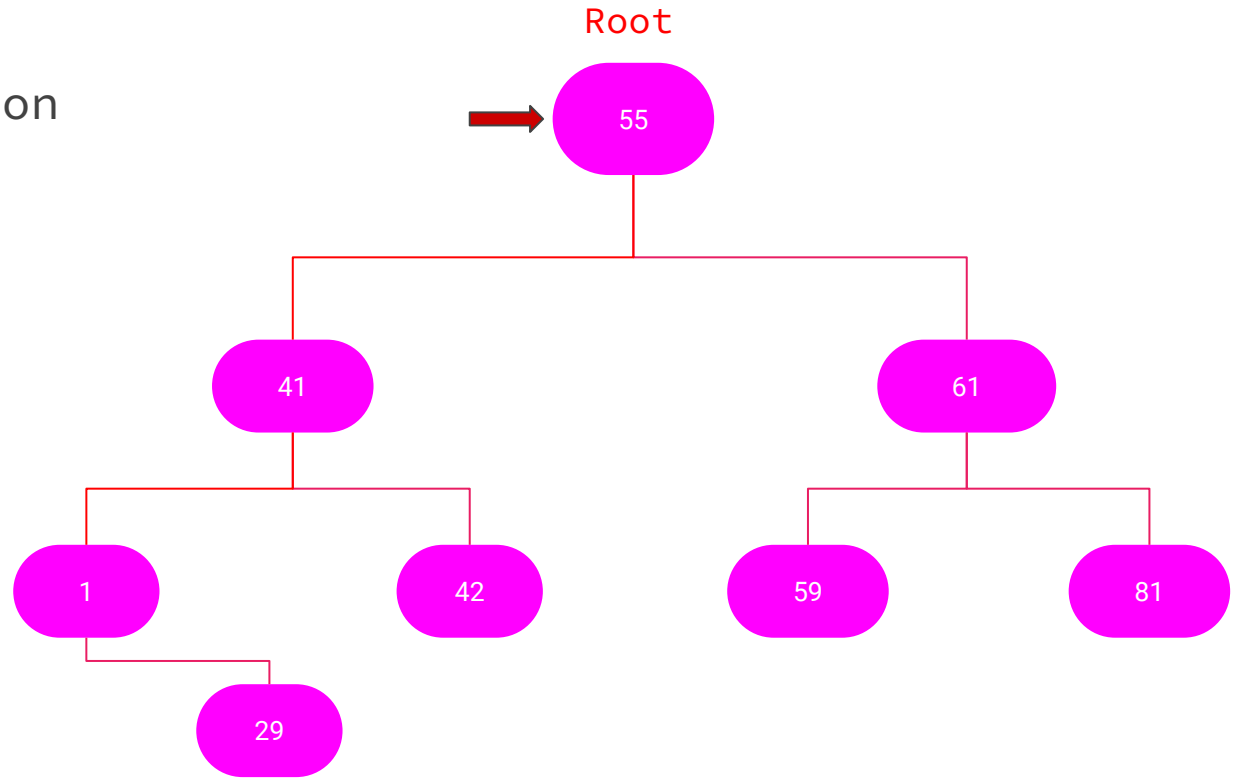
1 29 41 42 55 59 61 81

Simulation



1 29 41 42 55 59 61 81

Simulation



1 29 41 42 55 59 61 81

Complexity Analysis:

Tree sort algorithm requires a binary search tree to sort the integer datas. It just does 2 operation one is tree insertion and other is in order traversal. In order traversal time complexity is always 0(n) cause it is just visit all the nodes and print the values in sorted manner. But tree sort time complexity is fully dependent on bst insertion operation. Insertion operation varies on types of bst.

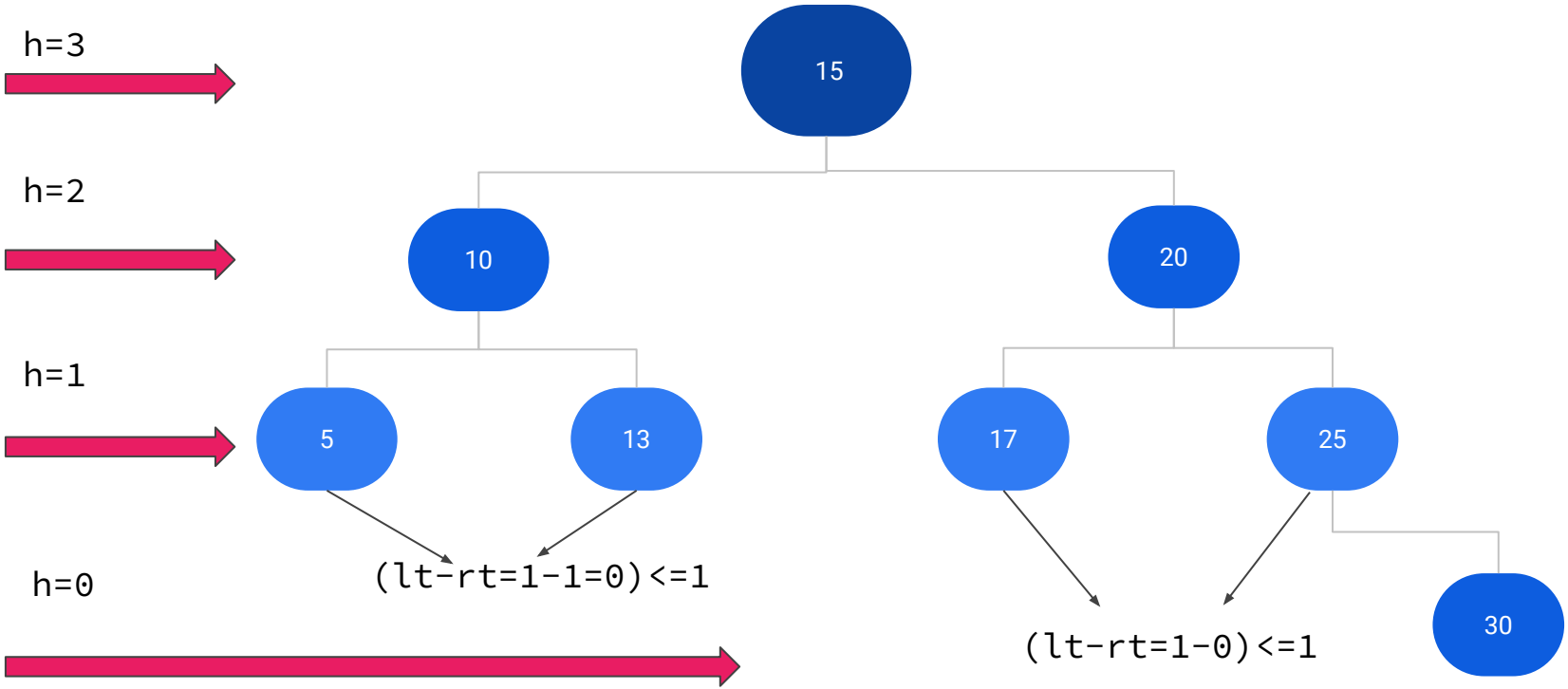
Best Case:

Tree sort best case would get when bst is balanced binary tree. If bst is a balanced binary tree then insertion operation time complexity for 1 elements is logn then for n elements it’s nlogn.

Balanced Binary Tree:

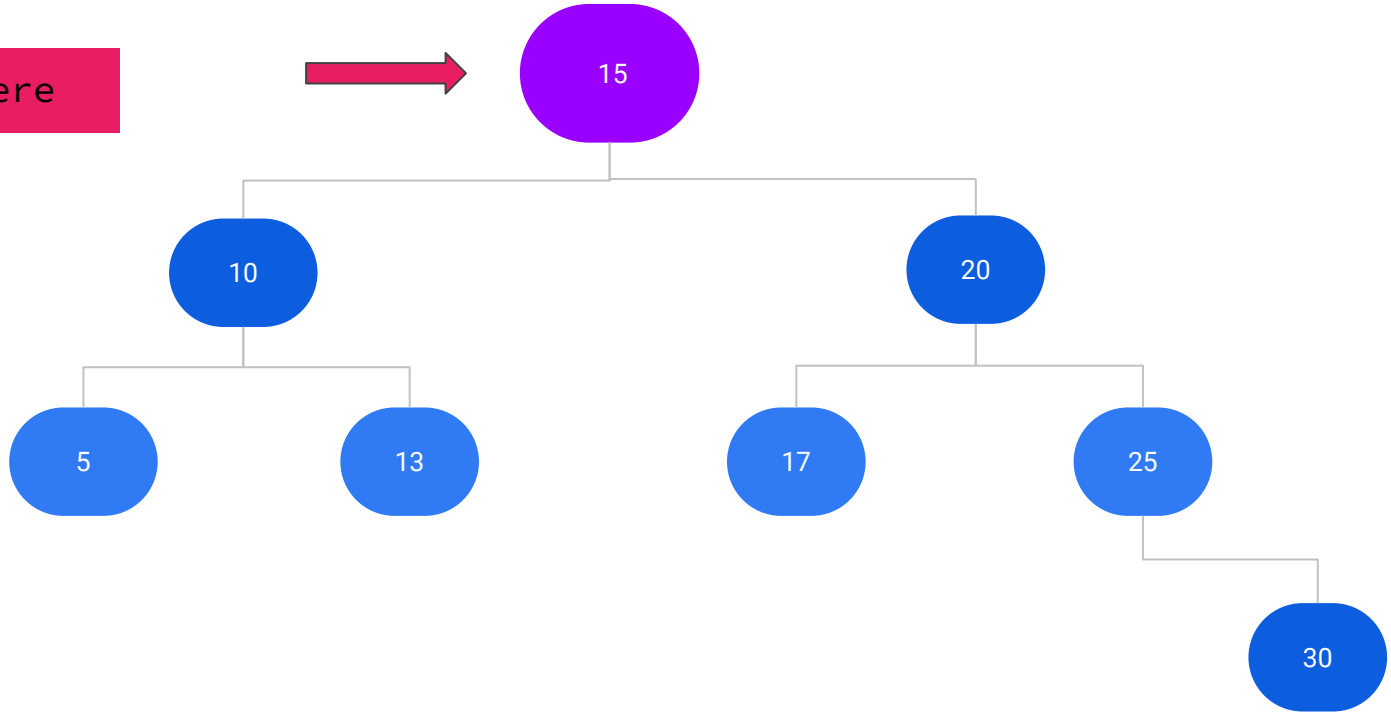
A balanced binary tree also called as a height balanced tree is a binary tree structure in which the left and right subtrees of every node differ in height by no more than 1.

Balanced Binary Tree:

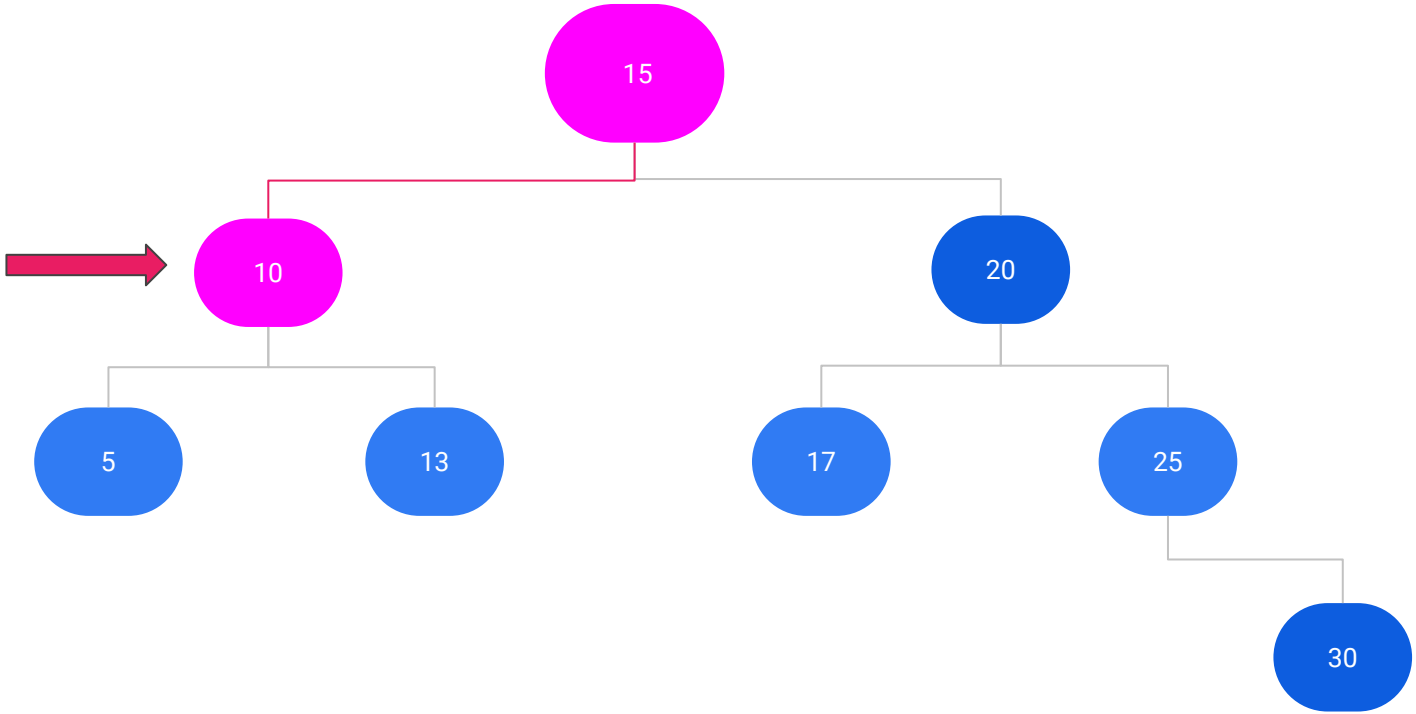


Insertion in Balanced Binary Tree:

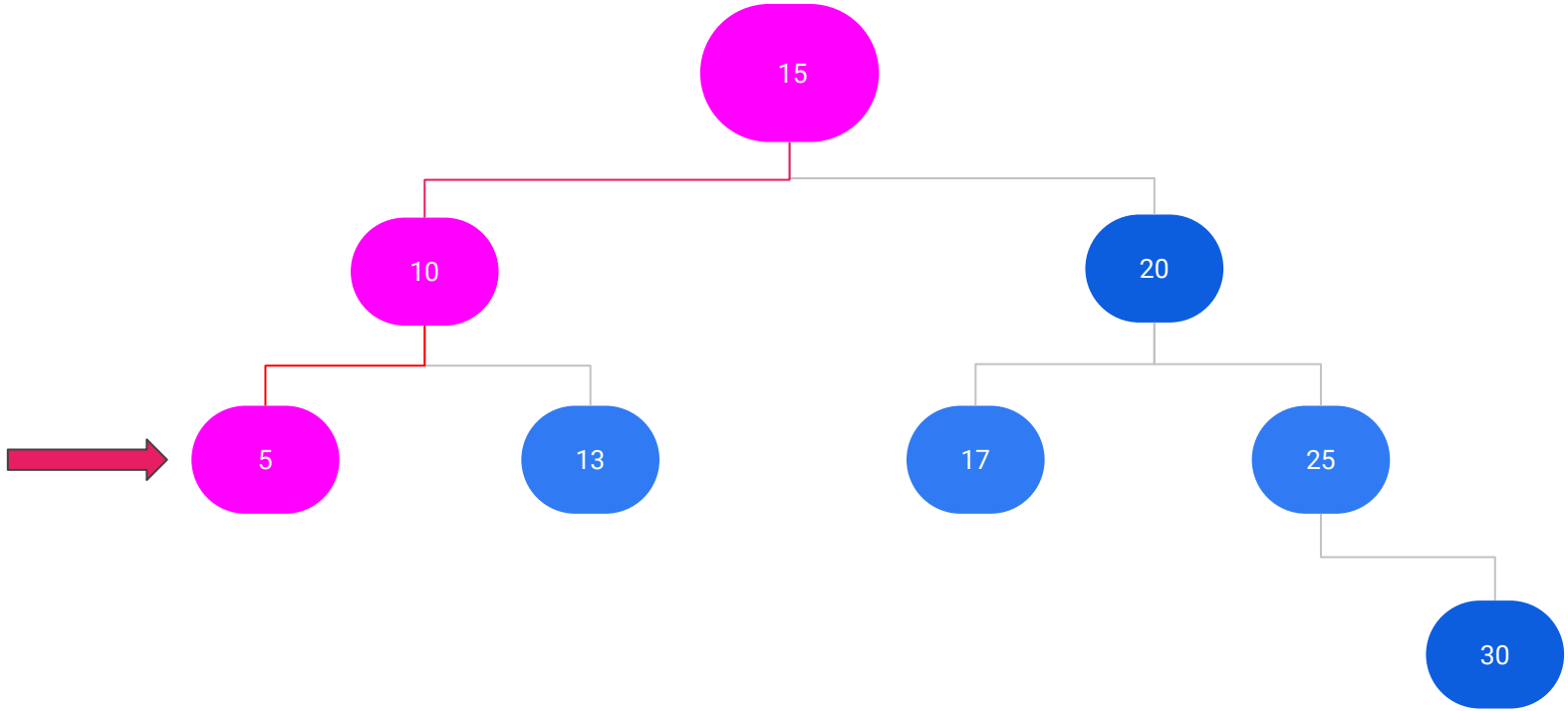
Insert 3 here



Insertion in Balanced Binary Tree:

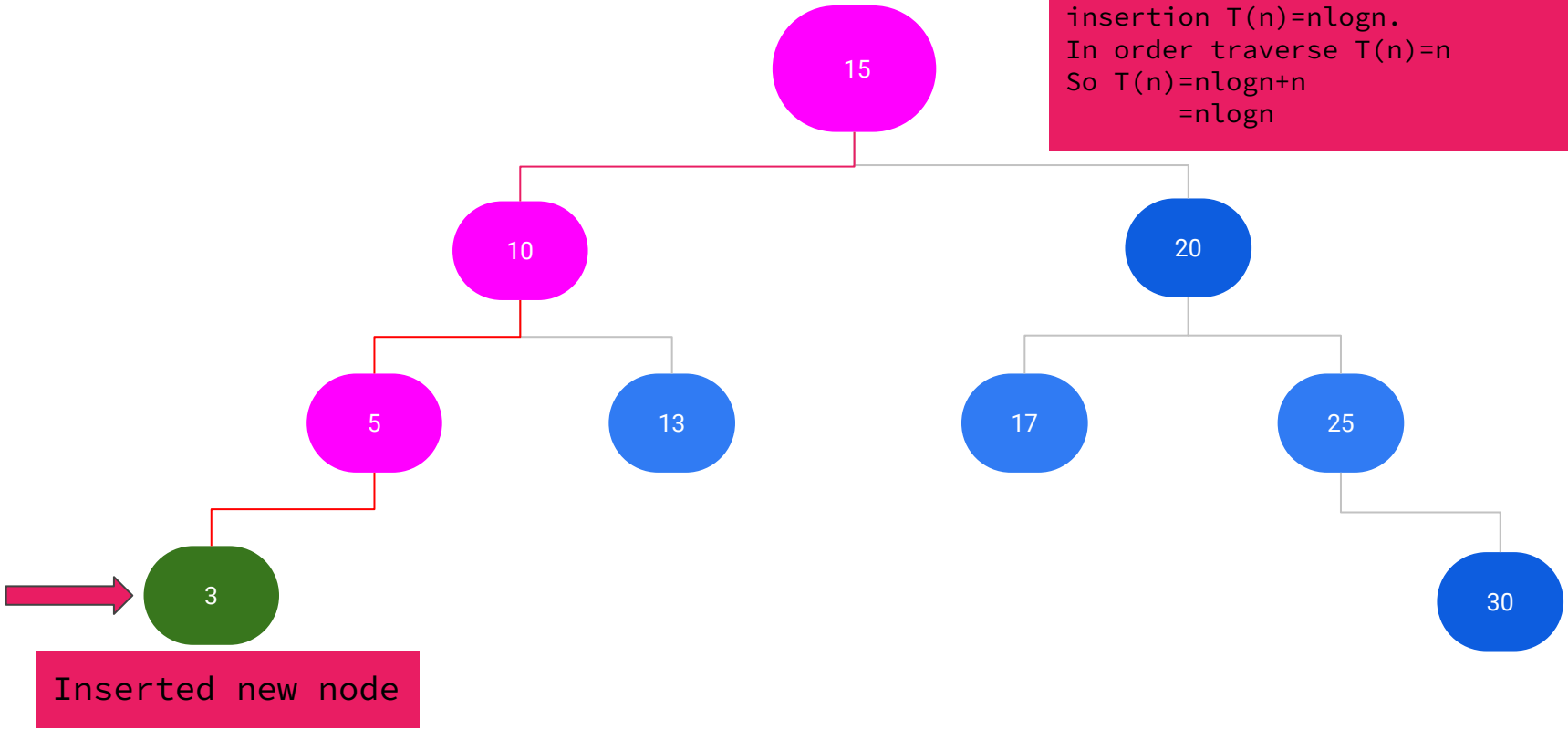


Insertion in Balanced Binary Tree:



Insertion in Balanced Binary Tree:

Intuitive analysis:
Here, for inserting 3 needs to traverse just half of the tree. The height of the tree is logn.Complexity for this insertion is logn. So for n elements insertion T(n)=nlogn.
In order traverse T(n)=n
So T(n)=nlogn+n
=nlogn



Best Case Continued(Substitution Method):

$$T(n) = 2T(n/2) + cn$$

$$= 4T(n/4) + 2cn$$

$$= 8T(n/8) + 3cn$$

$$= 2^k T(n/2^k) + kcn$$

Best Case Continued(Substitution Method) Cont'd:

$n/2^k=1$

$n=2^k$

$\log_2 n=\log_2 2^k$

$k=\log_2 n$

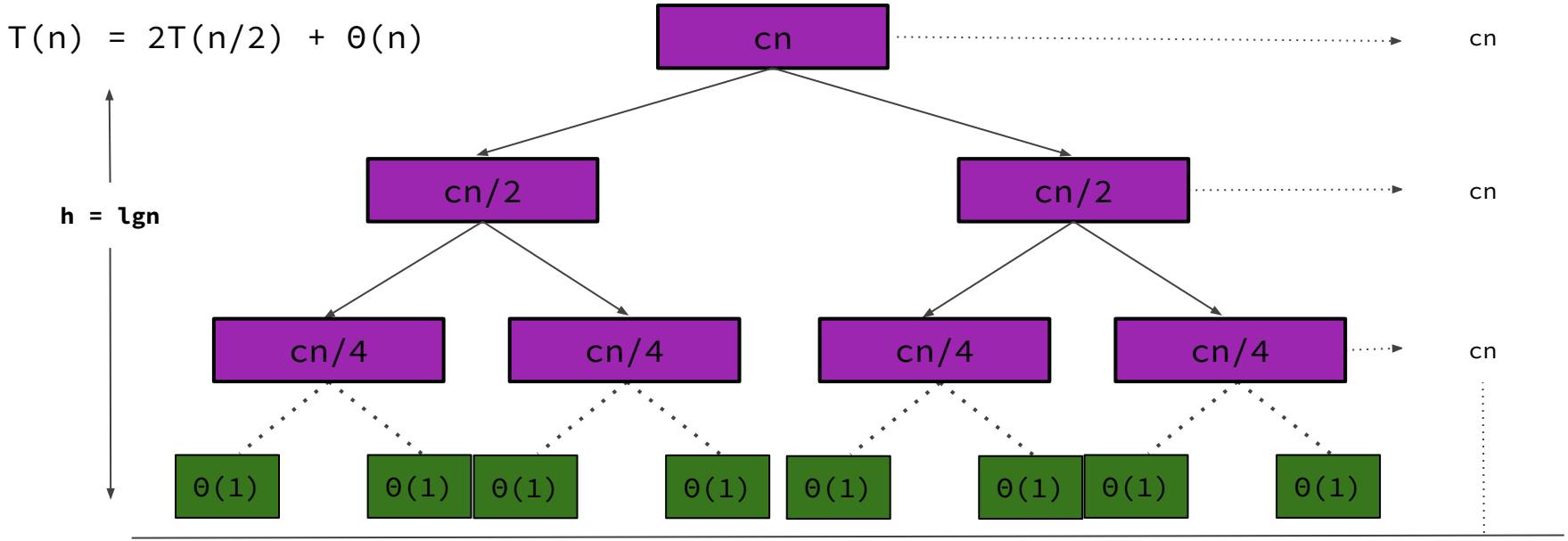
$=2^{\log n}(n/n)+ cn\log_2 n$

$=2\log_2 n+ n\log_2 n$

$= n\log_2 n$

Best complexity of tree sort is $0(n\log_2 n)$.

Best Case Continued(Recursion Tree) :



T(n) = lgn * cn
= **θ(nlgn)**

Best complexity of tree sort is θ(nlog$_2$n).

Best Case Continued(Master Theorem) :

———

T(n) = 2T(n/2) + 0(n)

Here,

a=2, b=2 and f(n)=n

So,

$n^{logb\_a} = n^{log2\_2} = n^1 = n$

n=n

Thus, case 2

So, $T(n) = \Theta(nlg^{(0+1)}n) = \Theta(nlg^1n) =$ **0(nlgn)**

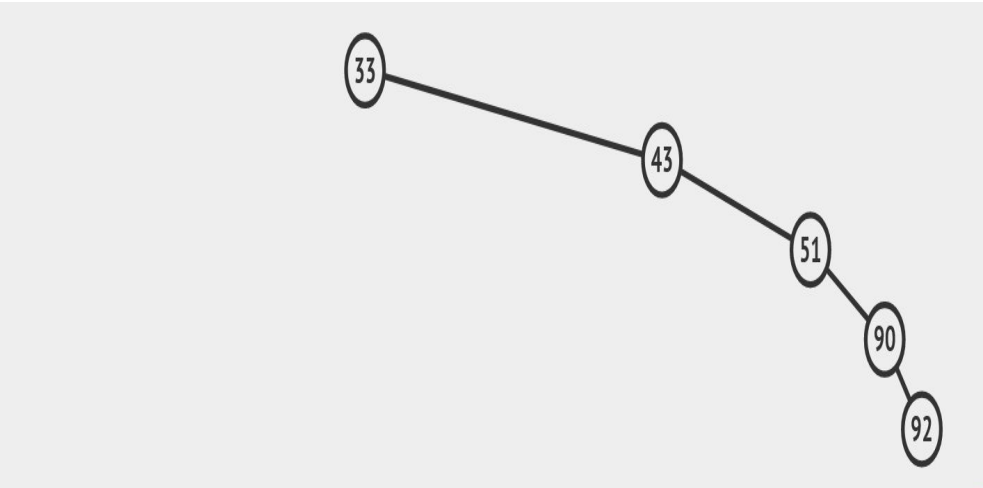
Best complexity of tree sort is 0(nlog$_2$n).

Worst Case:

Worst case of tree sort would get when binary tree is unbalanced. The worst-case appears when the algorithm operates on an already sorted set or almost sorted inputs set. Sorted input sets create left skewed tree or right skewed tree that are unbalanced tree or height imbalanced tree.The worst case can be optimized from O(n^2) to O(nlogn)by using a self-balancing binary search tree then time complexity of worst case reduced exponential n^2 to nlogn and then the best,worst and average cases of tree sort is $\Theta(n\log_2 n)$.

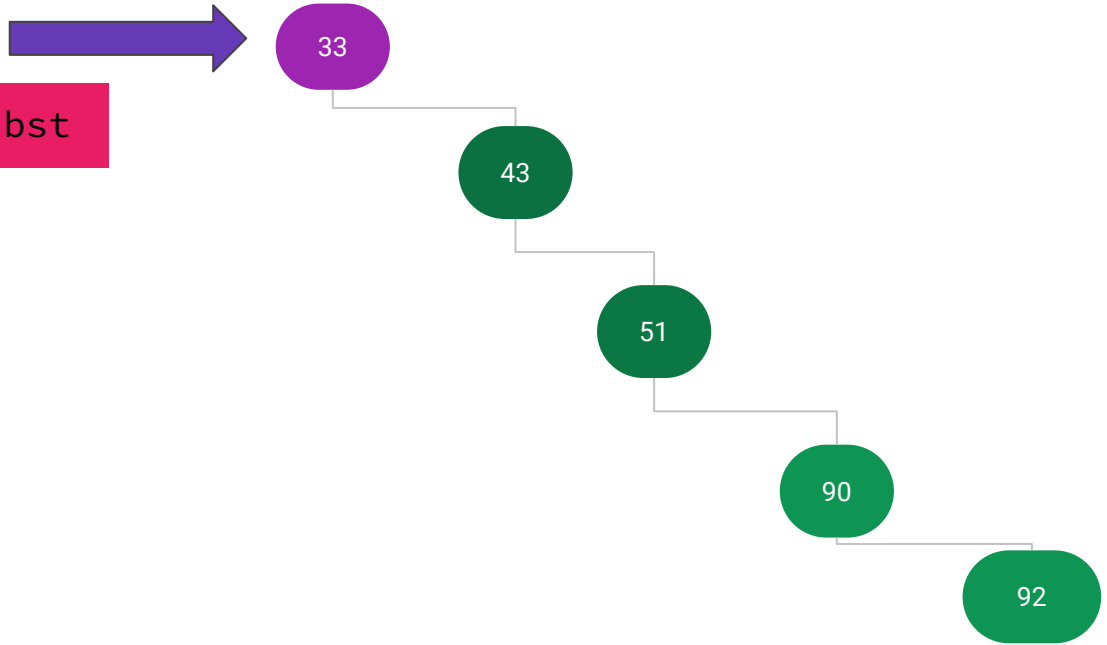
Worst Case Analysis:

It’s a right skewed binary search tree. The height of this tree is n. If now adds one element 95 in bst the insertion operation has to check all elements in bst that’s why time complexity will be n^2.

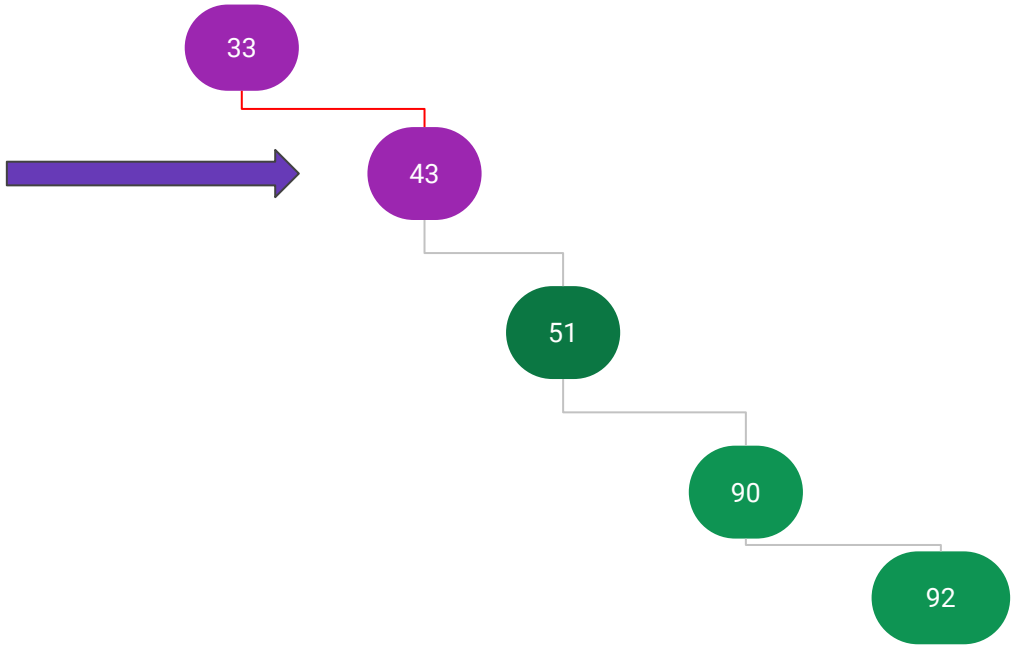


Worst Case Analysis Cont'd:

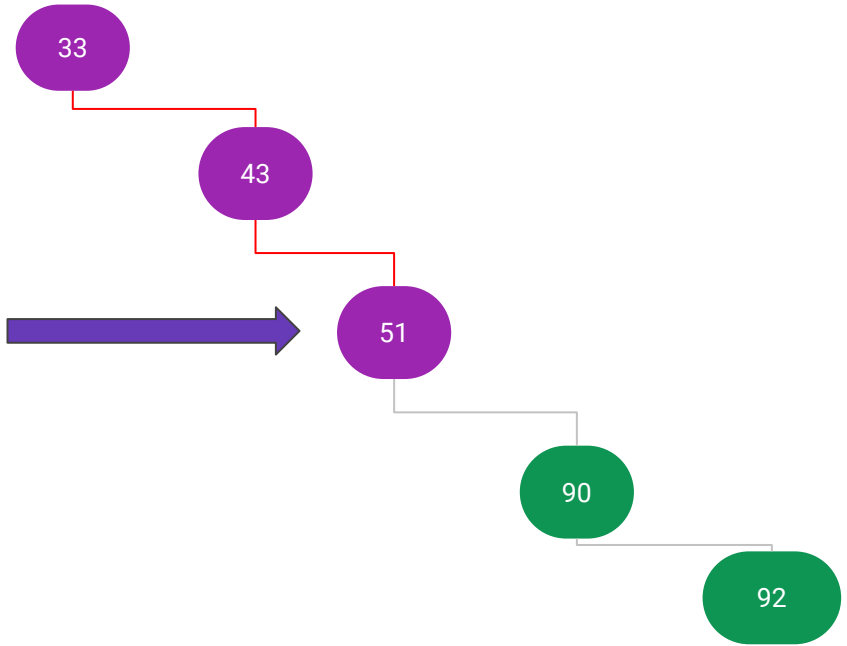
`Insert 95 in this bst`



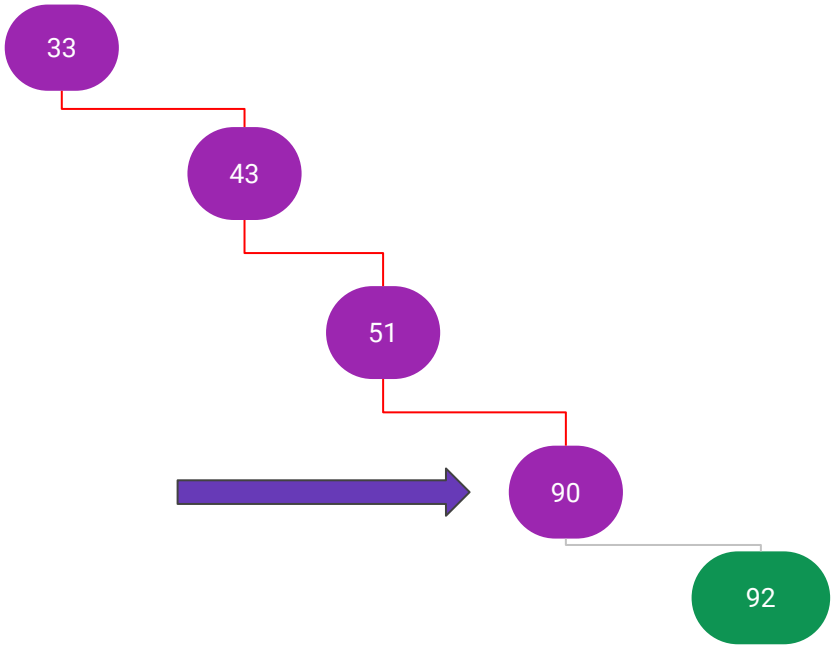
Worst Case Analysis Cont'd:



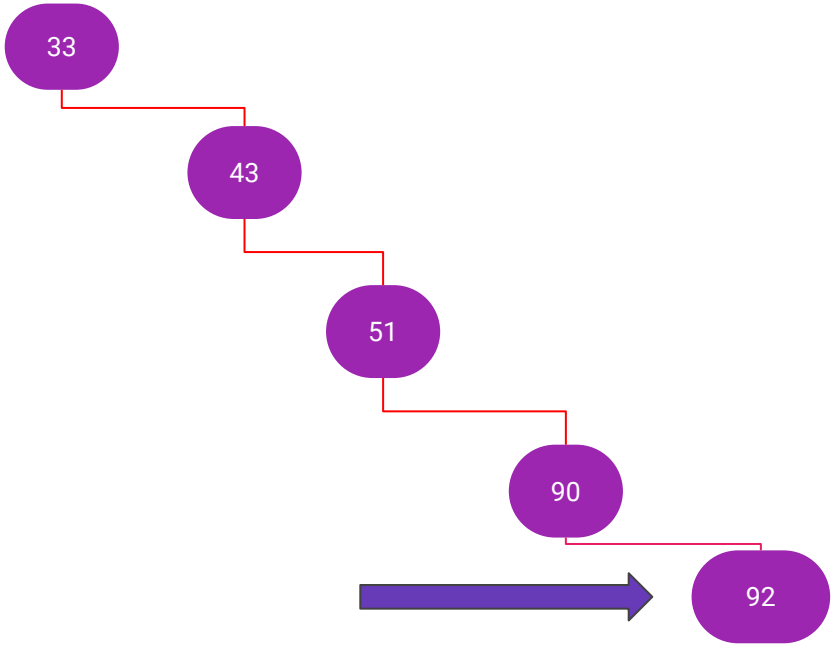
Worst Case Analysis Cont'd:



Worst Case Analysis Cont'd:

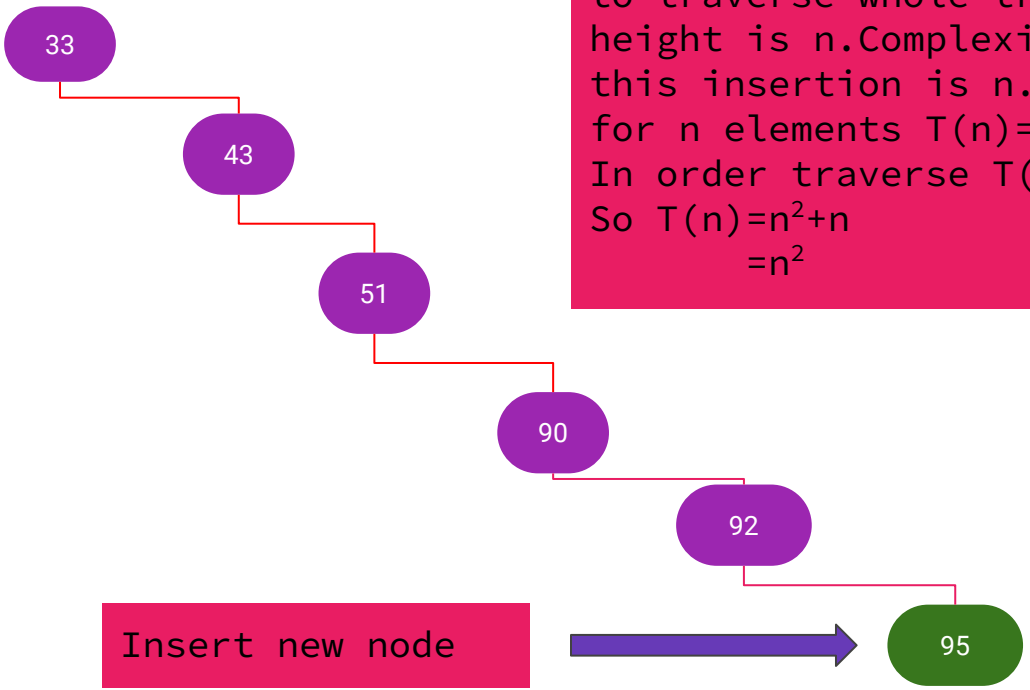


Worst Case Analysis Cont'd:



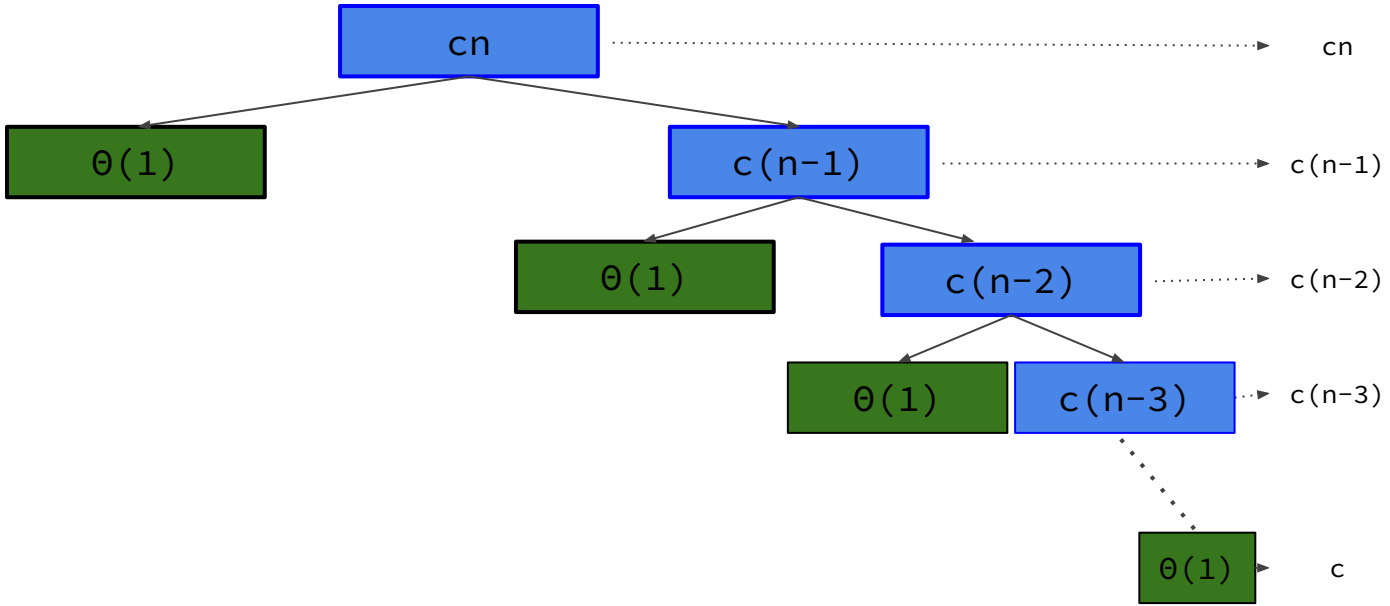
Worst Case Analysis Cont'd:

Intuitive Analysis:
Here for inserting 95 has to traverse whole tree.The height is n.Complexity for this insertion is n. So for n elements $T(n)=n^2$.
In order traverse $T(n)=n$
So $T(n)=n^2+n$
$=n^2$



Worst Case (Recursive Tree Method):

$T(n) = T(n-1) + \Theta(n)$



$$T(n) = c(n + (n-1) + (n-2) + ... + 1)$$
$$= c(n(n+1)/2)$$
$$= \mathbf{\Theta(n^2)}$$

Average Case:

Average case comes when the bst is mixed of balanced and unbalanced subtrees.

- In the average case, INSERT produces a mix of “BALANCED” and “UNBALANCED” tree splits.
- In a recursion tree for an average-case execution of INSERT, the good and bad splits are distributed randomly throughout the tree.

Suppose, for the sake of intuition, that the good and bad splits alternate levels in the tree, and that the balanced splits are best-case splits and the unbalanced splits are worst-case splits.

Average Case Analysis:

———

Unbalanced, UB(n) = B(n-1) + 0(n)

Balanced, B(n) = 2UB(n/2) + 0(n)

So now,

$$
\begin{aligned}
B(n) &= 2UB(n/2) + 0(n) \\
&= 2(B(n/2 - 1) + 0(n/2)) + 0(n) \\
&= 2B(n/2 - 1) + 0(n) \\
&= 0(\mathbf{nlgn}) \\
&= \text{Balanced}
\end{aligned}
$$

Average time complexity of tree sort is $0(n\log_2 n)$ that same as best time complexity.

Pros & Cons:

It is as fast as quick sort and in a linked list tree sort can make changes very easily. Tree sort algorithm needs separate heap memory allocation for making binary search tree that generate significant fast performance.

It takes a lot of time when the elements of an array is already sorted then it takes more time than other sorting algorithm. In worst case the running time is O(n^2)

Attributes of Tree Sort:

- **Tree sort is a adaptive sorting algorithm.**

  Input set has impact on time complexity of tree sort. When Input set is fully sorted or almost sorted then it's time complexity is n^2 while other cases give time complexity in nlogn.

- **Tree sort is a online sorting algorithm.**

  Tree sort is an online sorting algorithm that builds a binary search tree from the elements input to be sorted, and then traverses the tree, in-order.

- **Tree sort is only applicable on positive and negative integers.**