UCSC Silicon Valley Extension

Advanced C Programming

Trees and Binary search trees

Instructor: Radhika Grover

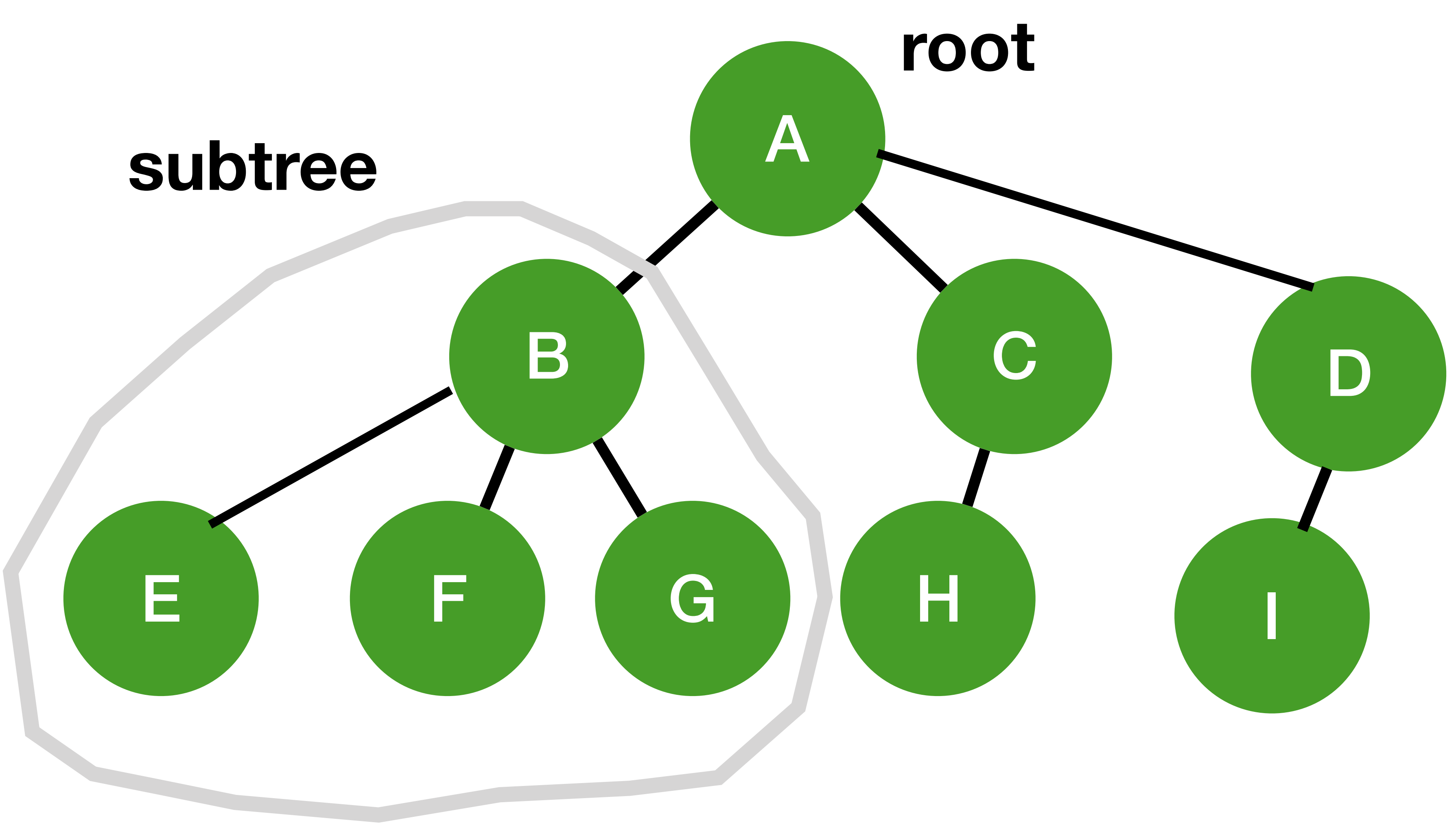
"*Trees sprout up just about everywhere in computer science*" - Donald Knuth

Overview

- Tree
- Definitions
- Binary search trees traversal (preorder, postorder, inorder, level-order)
- Binary search tree insertion and deletion
- AVL trees

Tree

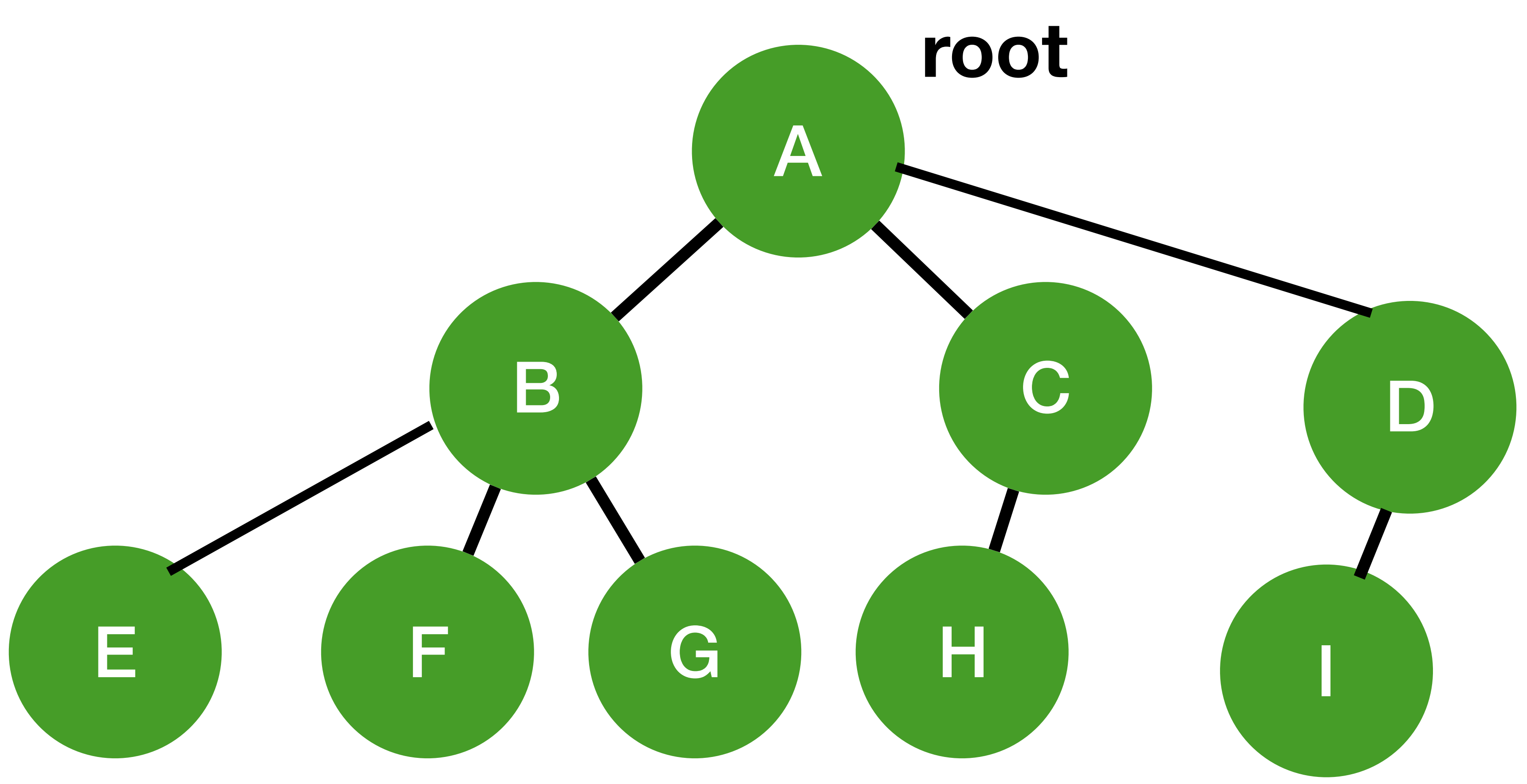
- Collection of nodes
- Has zero or more subtrees where the root of each subtree is connected to the root.
- Root of each subtree node is a child of the root, and root is its parent (recursive definition).
- Leaf nodes do not have any children.



Examples: B is the *parent* of E, F, G
E is a *child* of B
leaf nodes: E, F, G, H, I

Path

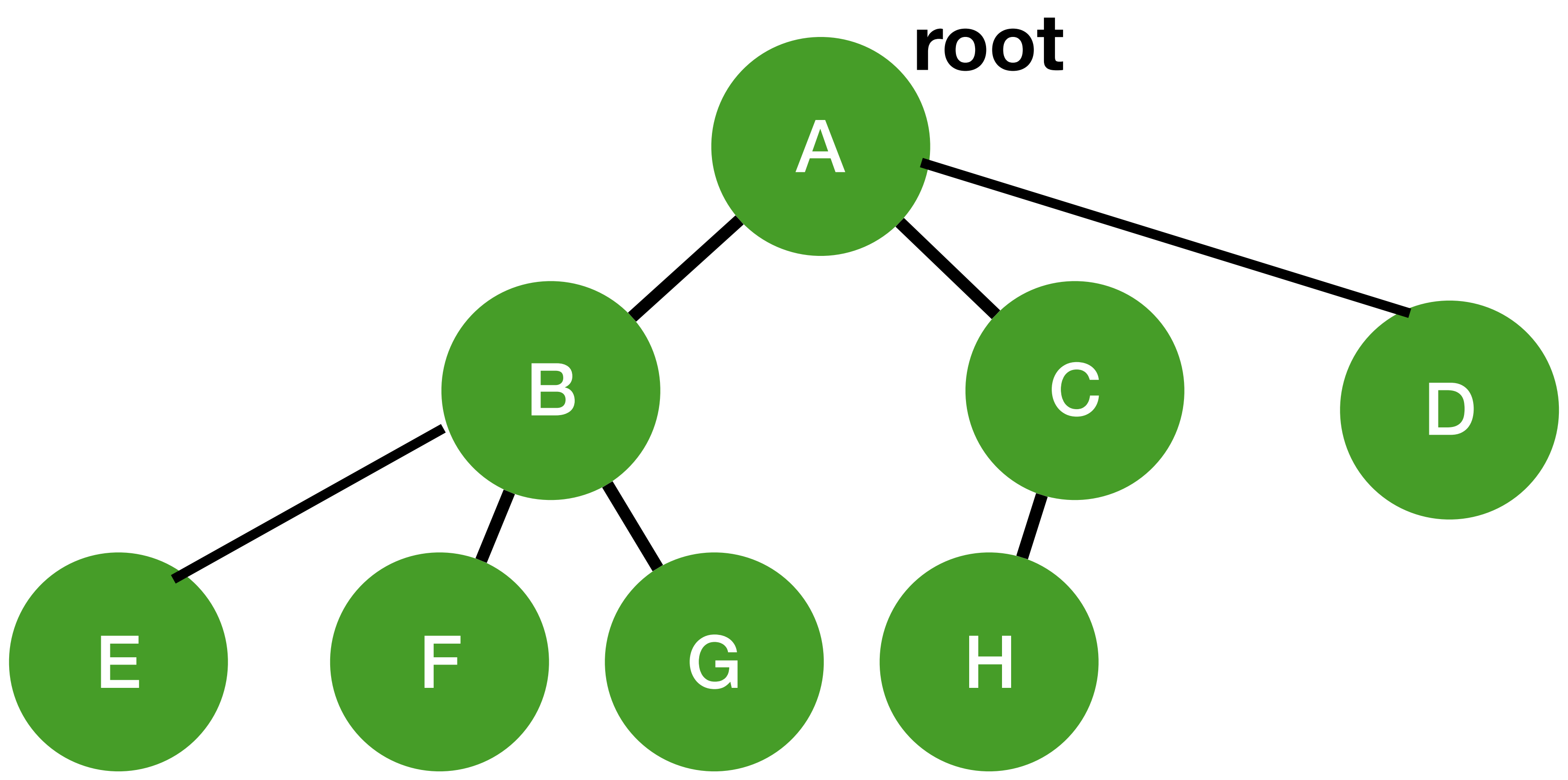
- Sequence of nodes $n_1, n_2, \ldots, n_k$ where n_i is the parent of n_{i+1} for $1 \leq i < k$.
- In a tree, there is exactly one path from root to each node.



Examples: A-B-G, C-H, A-D-I

Depth and height

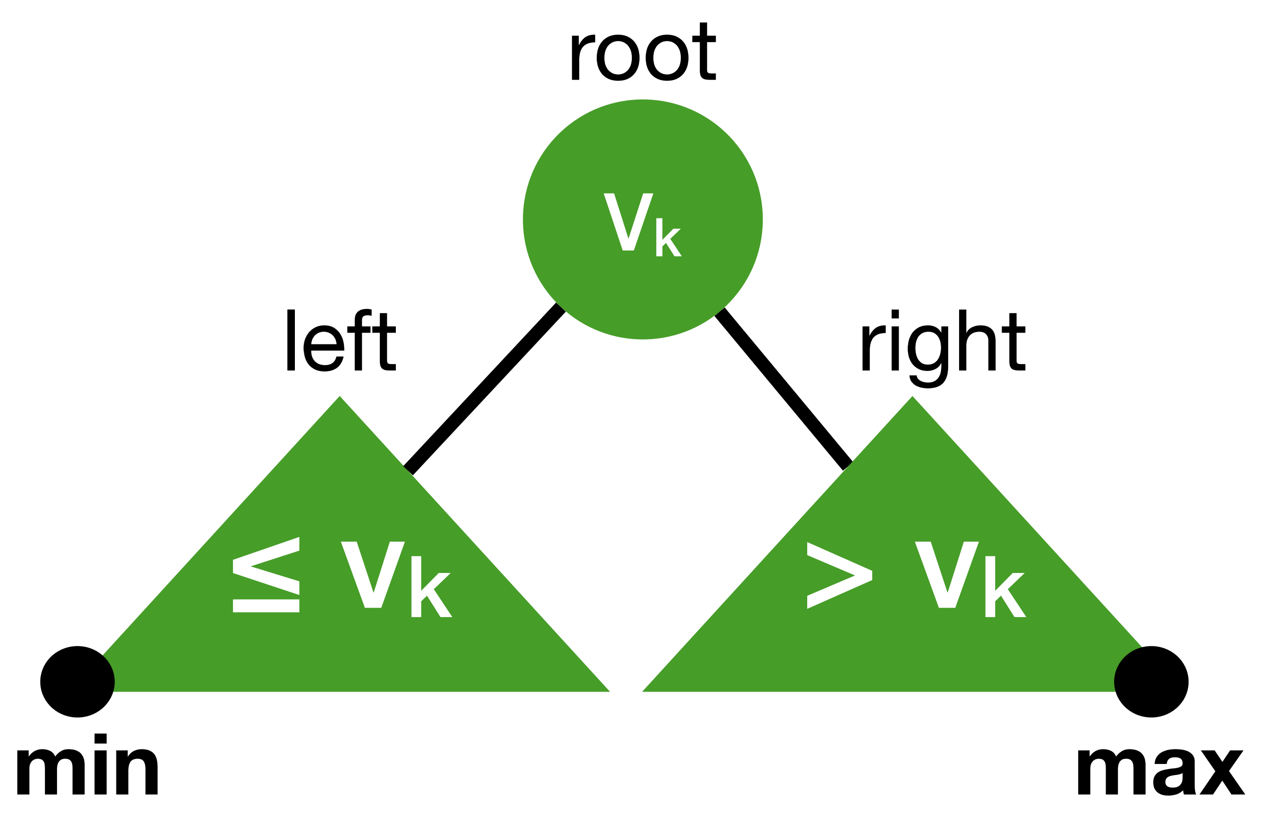
- *Depth of a node* is the number of edges on the path to a node from the root.
- *Height of a node* is the number of edges on the *longest* path from that node to a leaf.
- *Height of a tree* is the height of the root node.

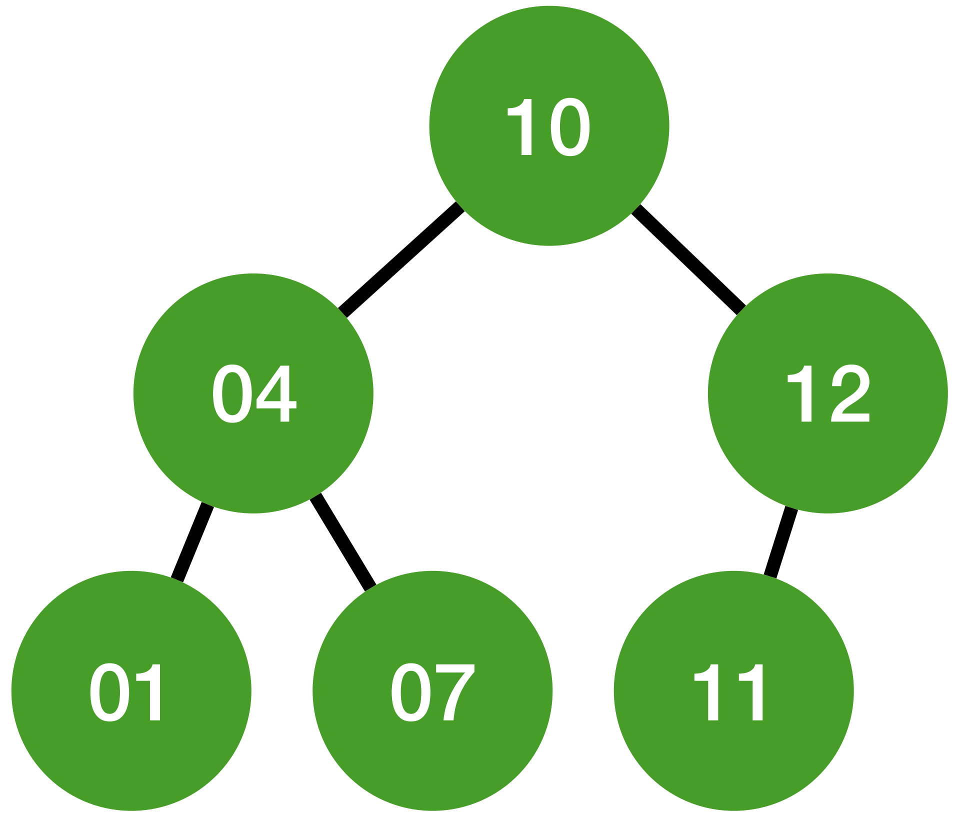


Examples: Depth of A = 0
Depth of G = 2
Height of C = 1
Height of A = tree height = 2

Binary search tree

- Type of binary tree with a special property:
  - The key at *each* root node is:
    - greater than or equal to the keys of all nodes in its left subtree.
    - less than the keys of all nodes in its right subtree.





Binary search tree

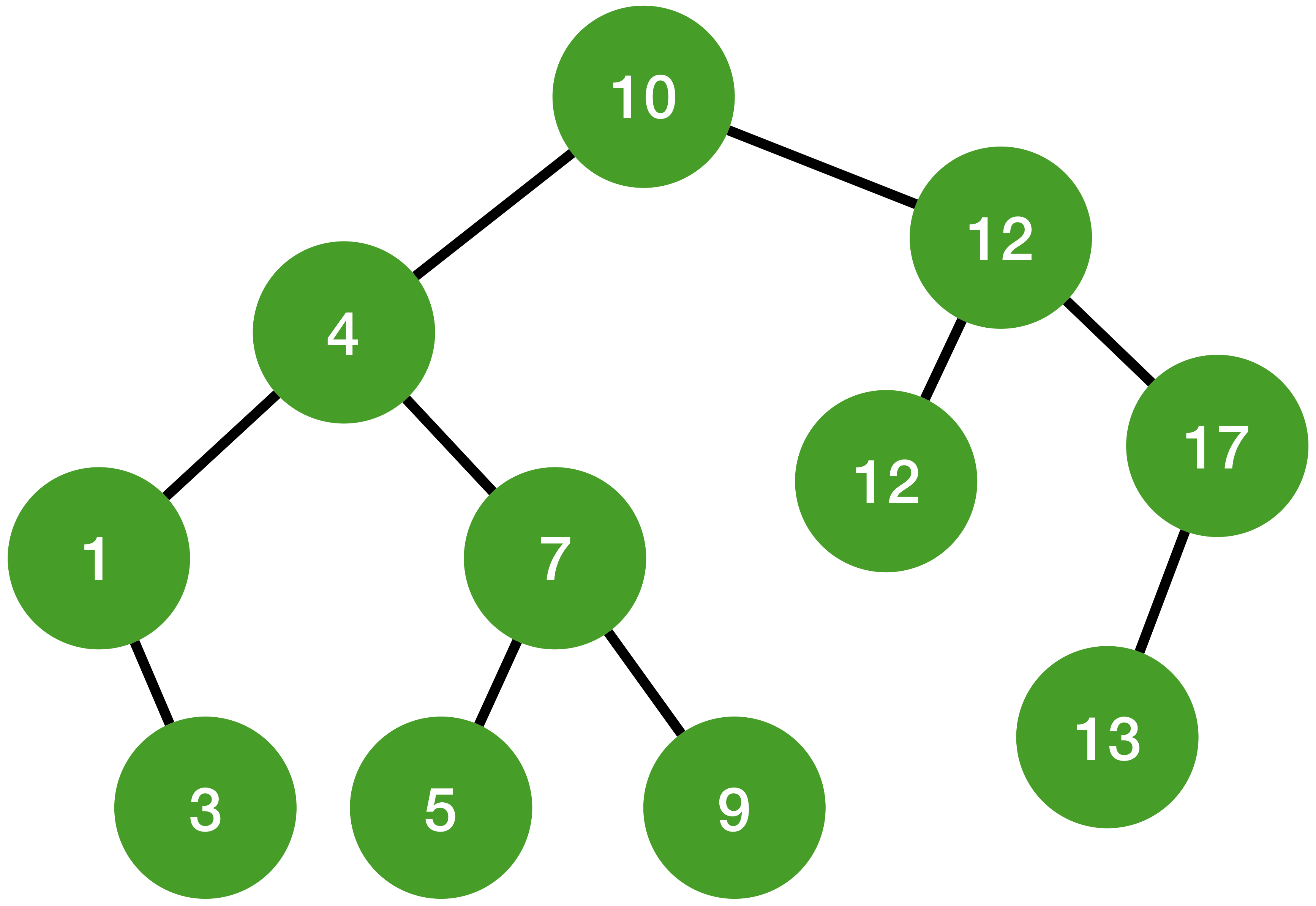
Supports operations such as :

- Search
- Tree traversal - preorder, inorder, level-order and postorder
- Insert
- Delete

Binary search tree - time complexity

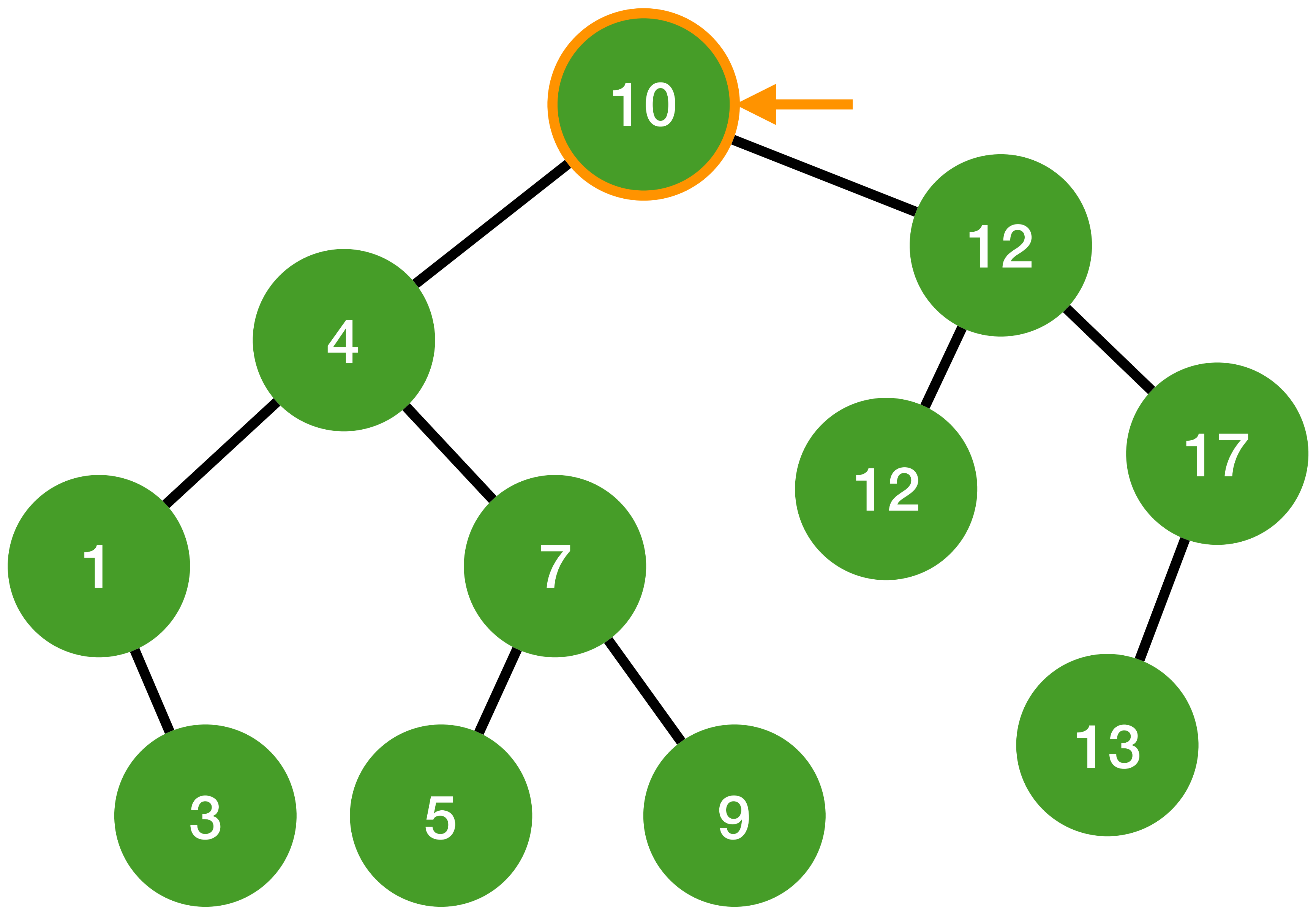
	Average case	Worst case
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Binary search tree - Search



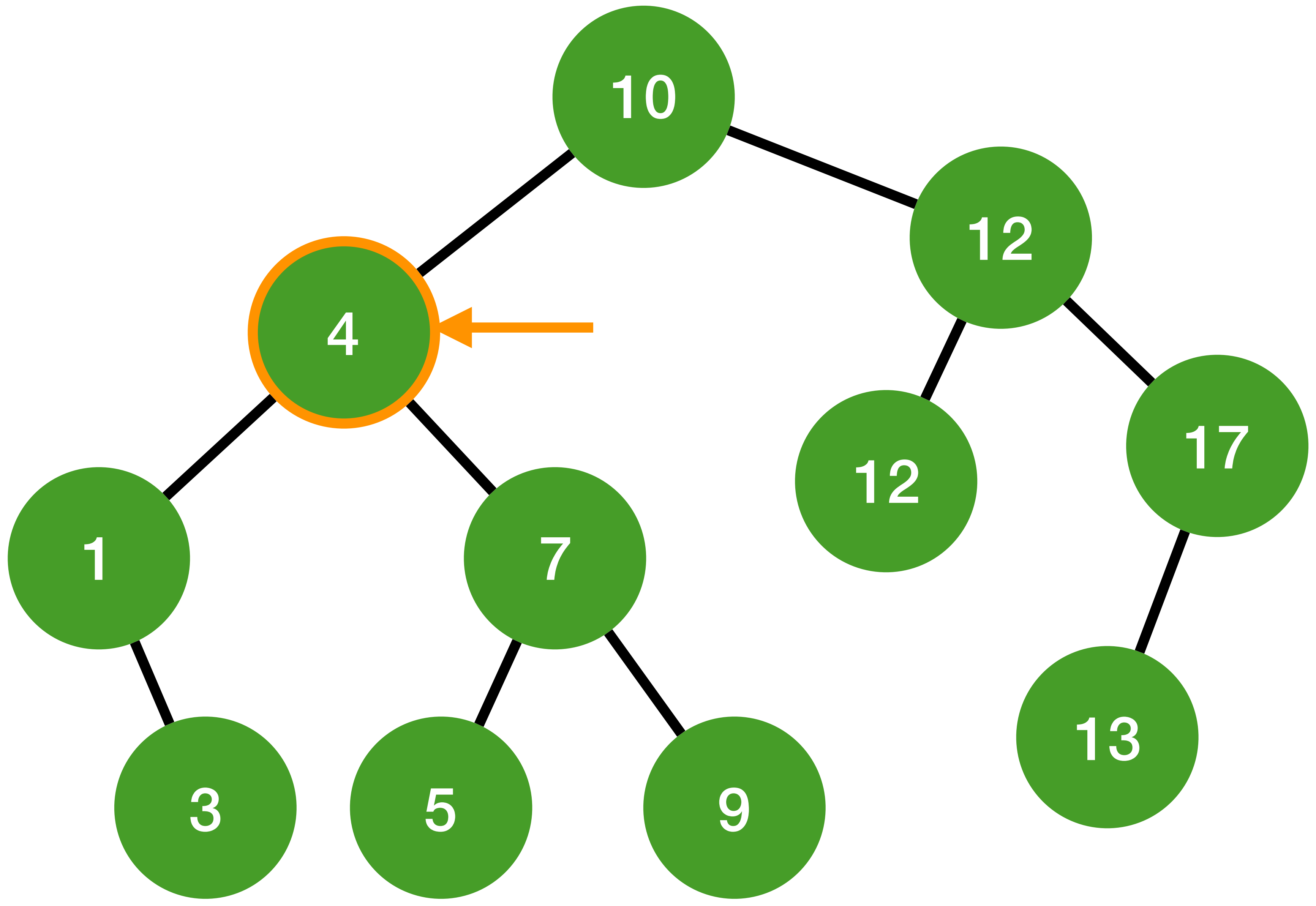
Search for key 5

Binary search tree - Search



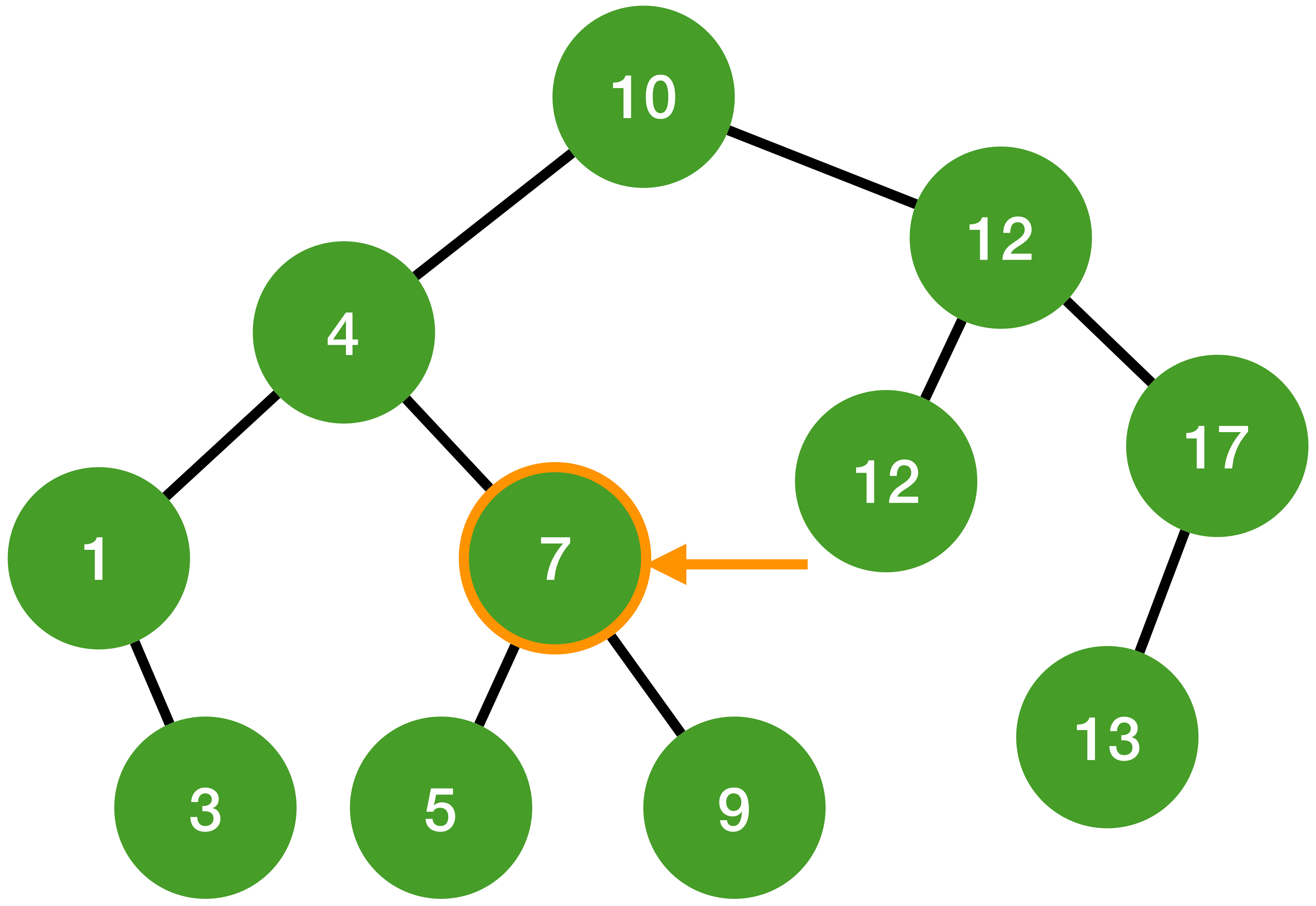
5 < 10 ? yes => search left

Binary search tree - Search



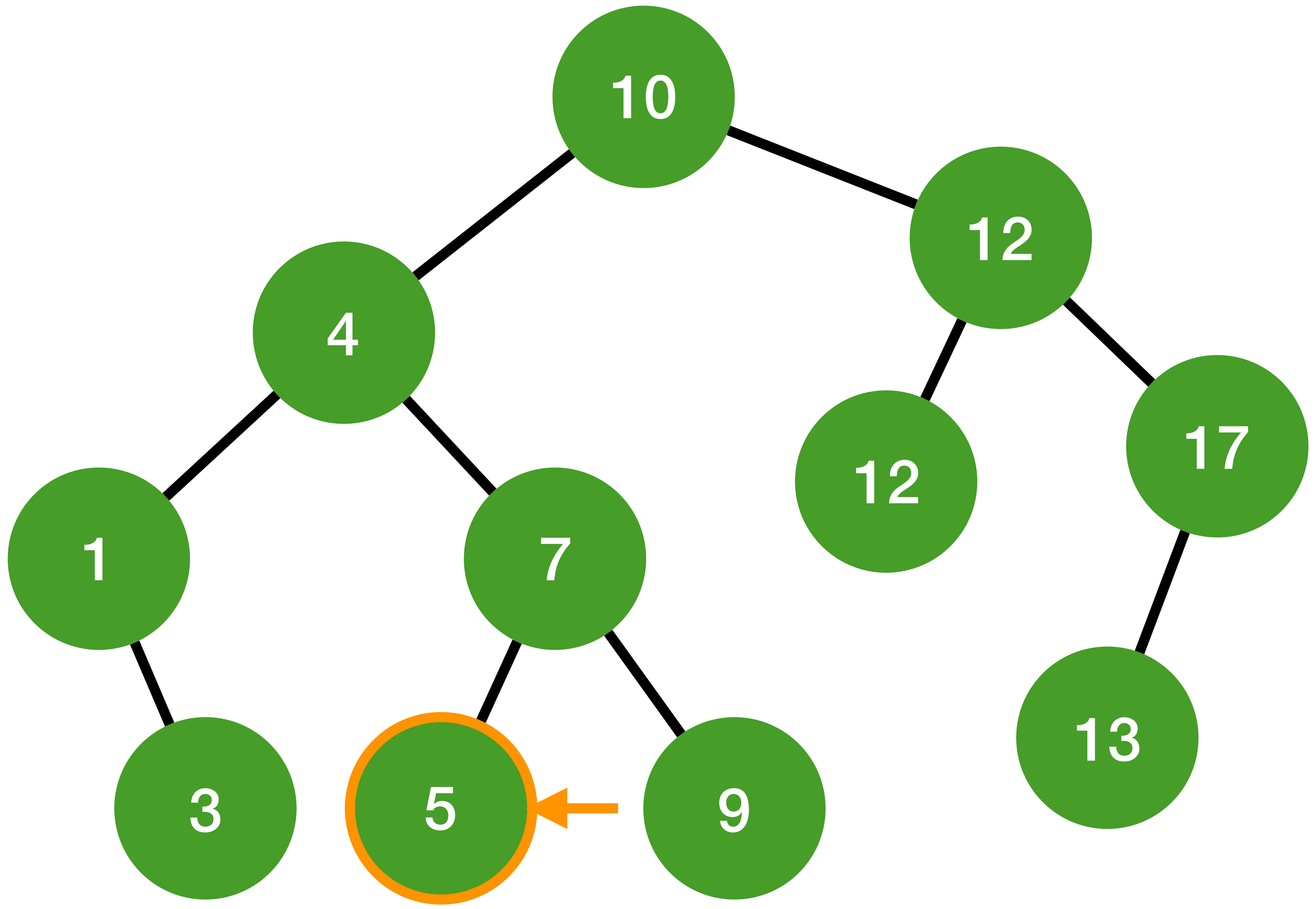
5 < 4 ? No => search right

Binary search tree - Search



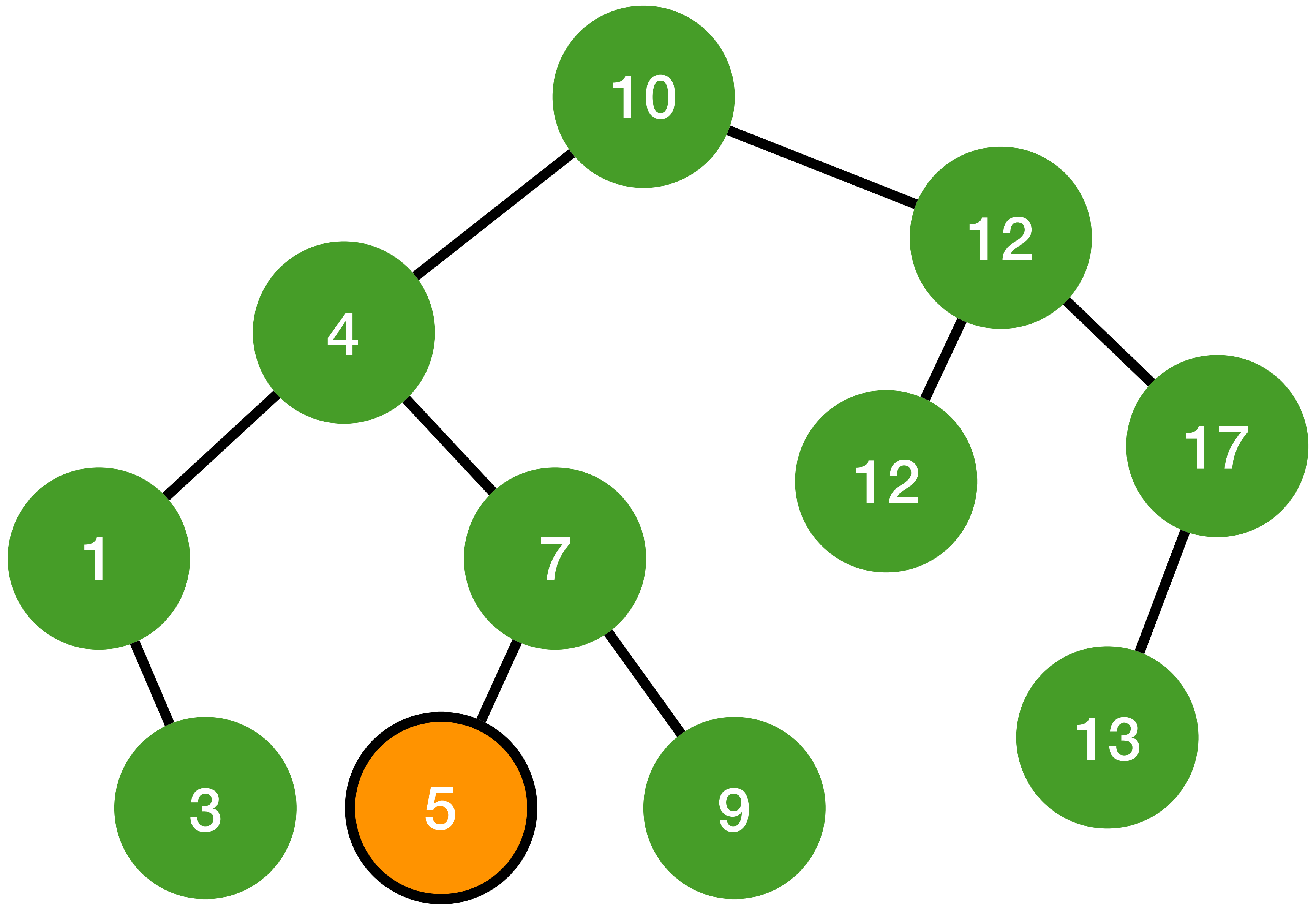
5 < 7 ? yes => search left

Binary search tree - Search



Key == 5

Binary search tree - Search



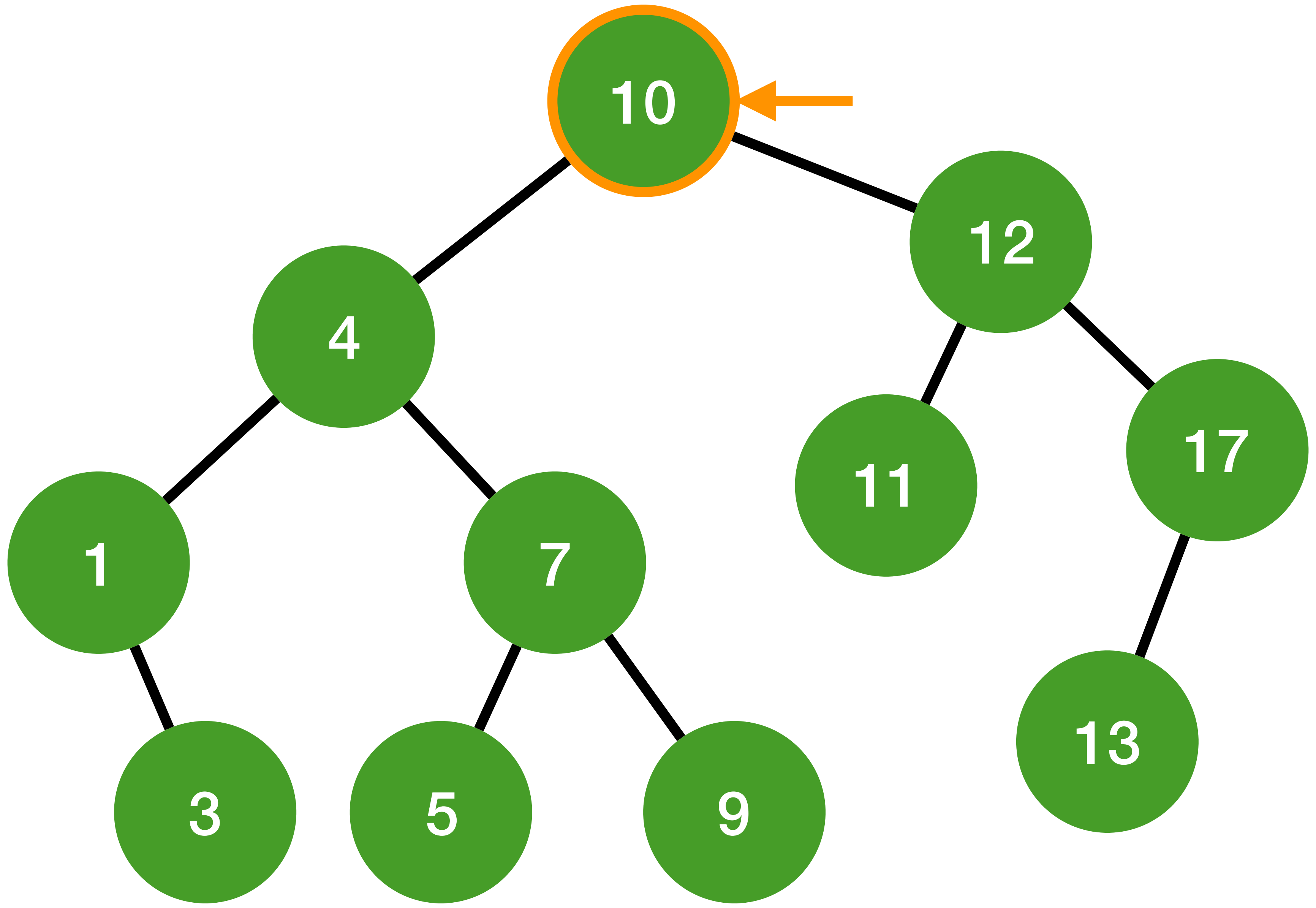
Binary search tree - Search algorithm

```
SearchTree(x, k){
  while(x != NULL and k!= x.key){
     if (k < x.key)
        x = x.left;
     else
        x = x.right;
  }
  return x;
}
```


Binary search tree traversal

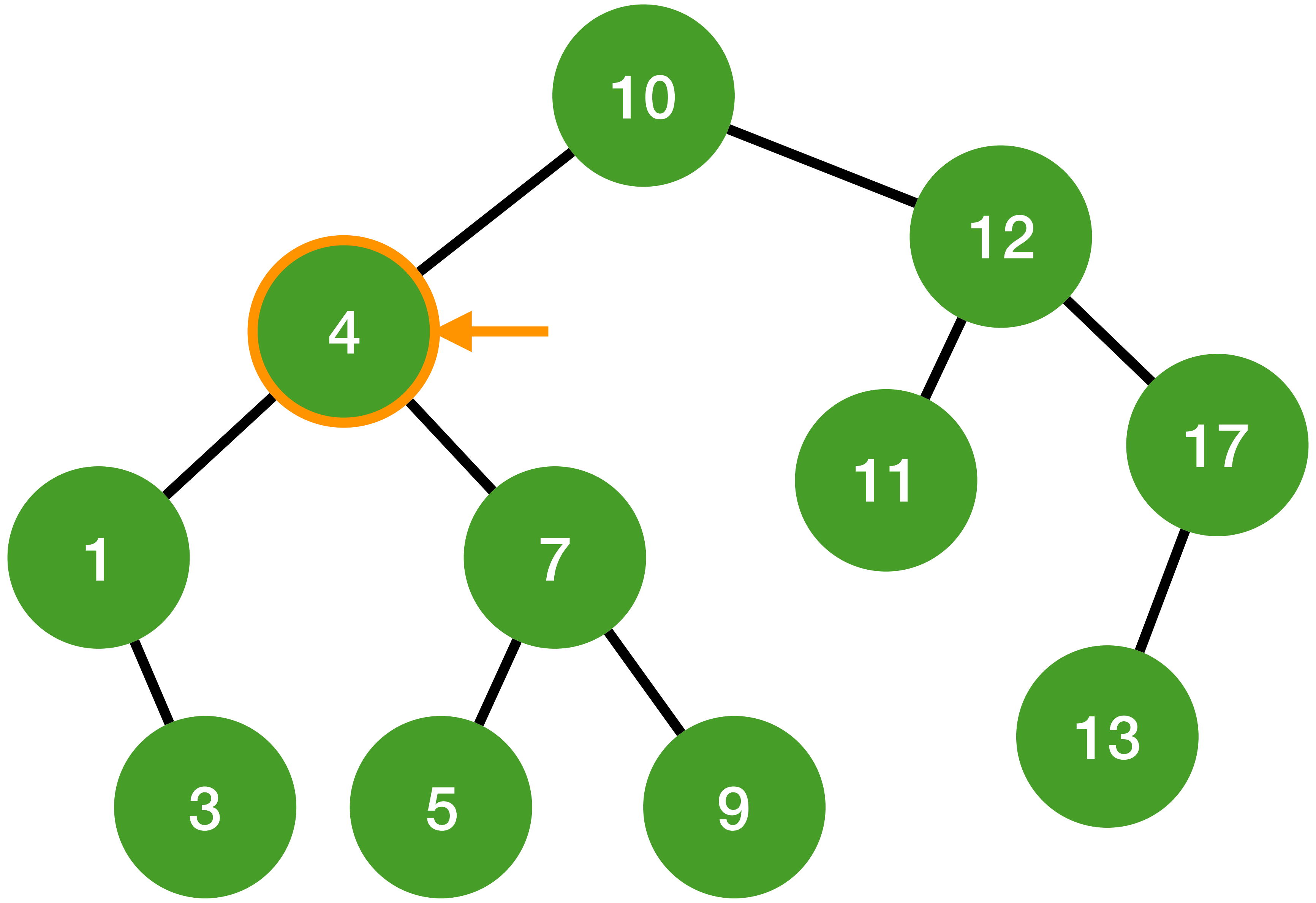
- Inorder: Left, **root**, right
- Preorder: **root**, left, right
- Postorder: left, right, **root**

Binary search tree traversal - In order



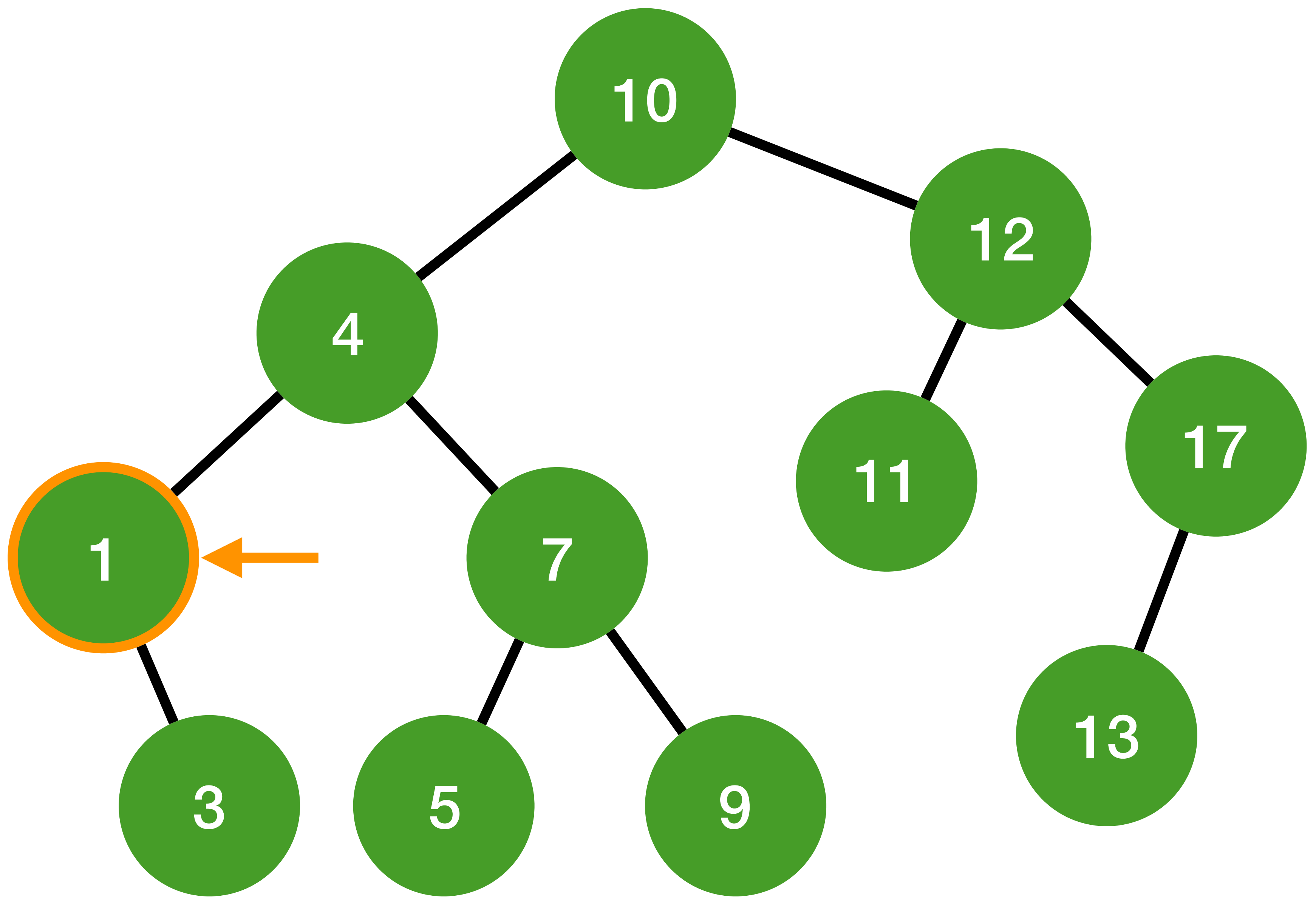
In order =

Binary search tree traversal - In order



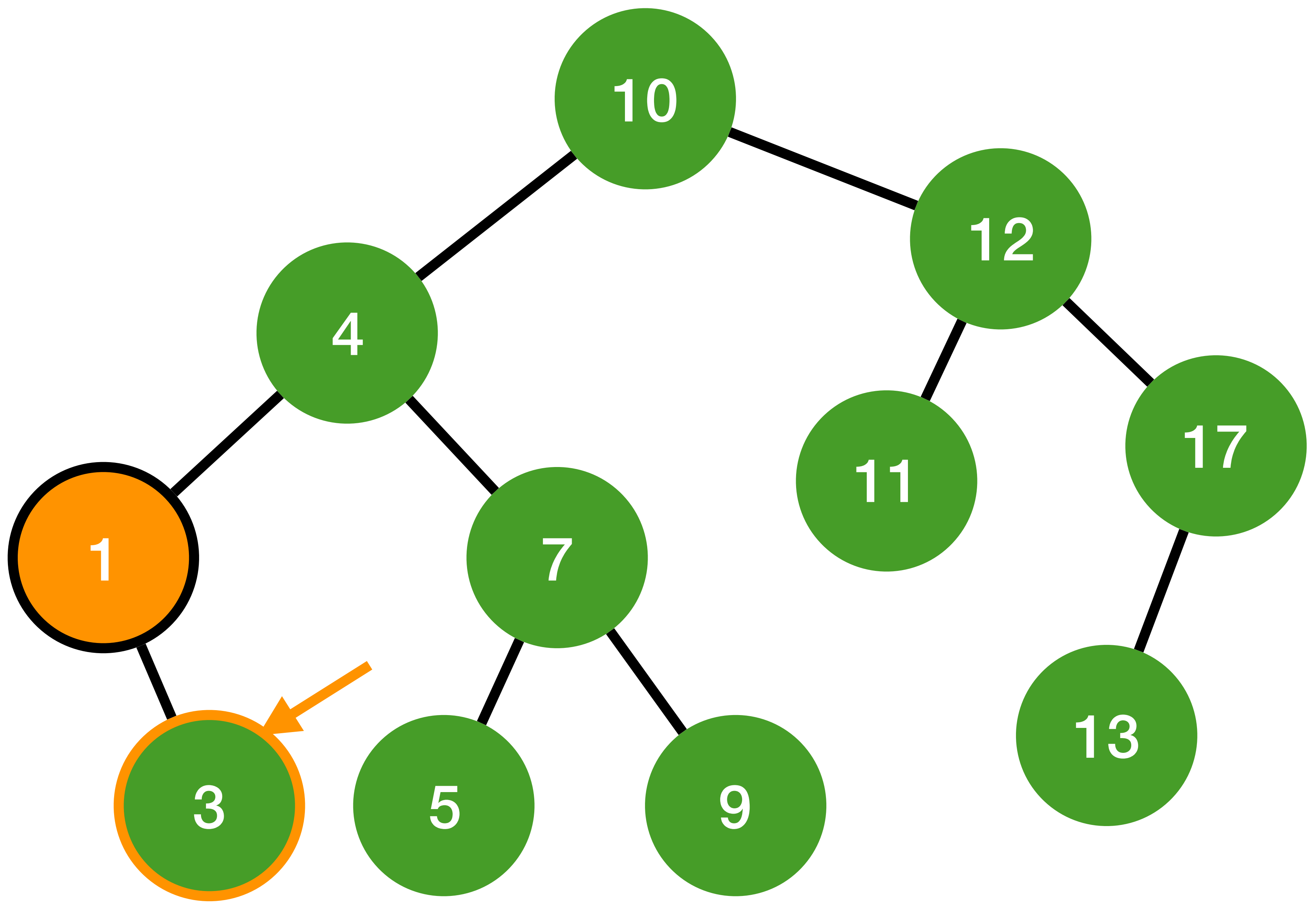
In order =

Binary search tree traversal - In order



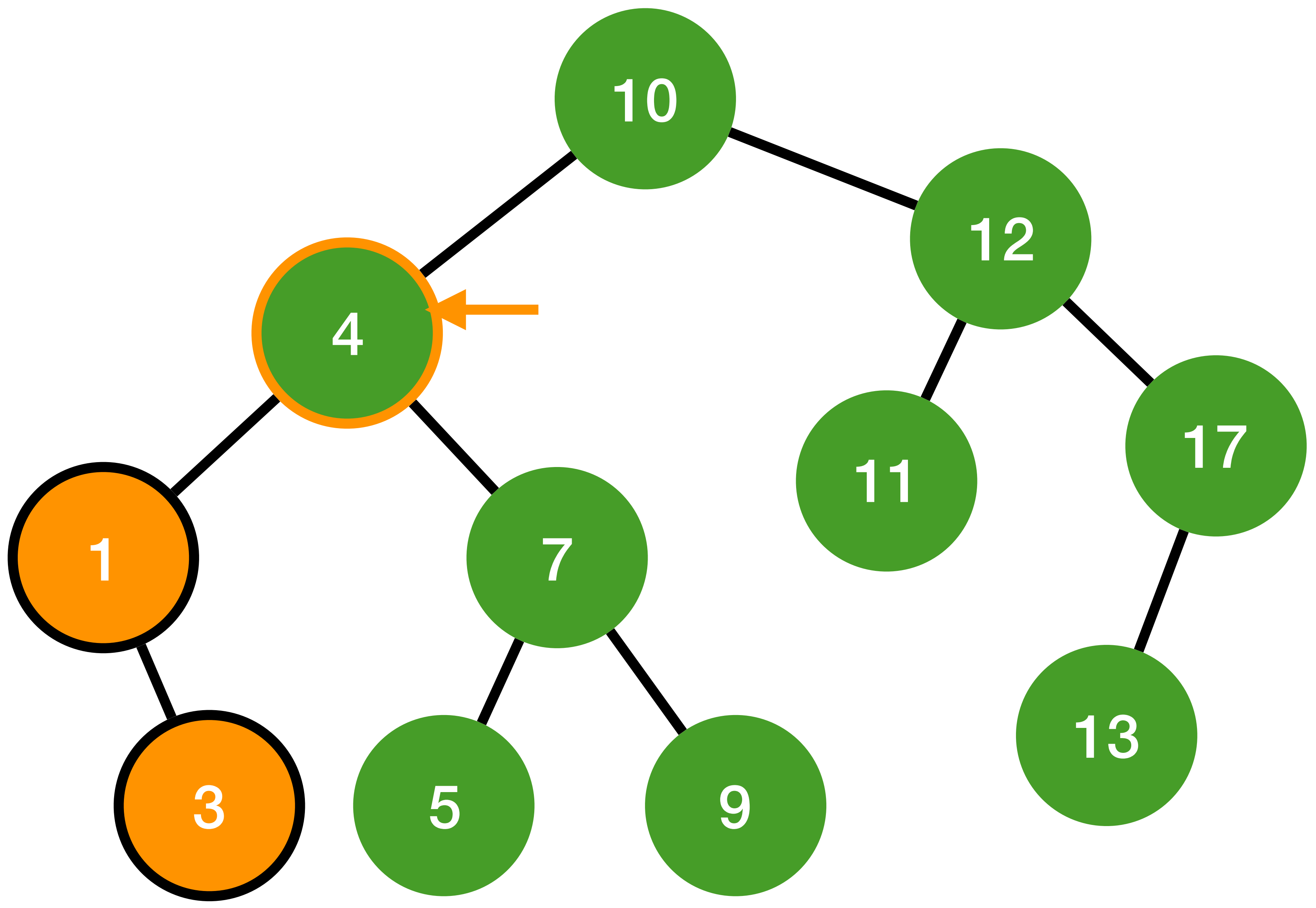
In order =

Binary search tree traversal - In order



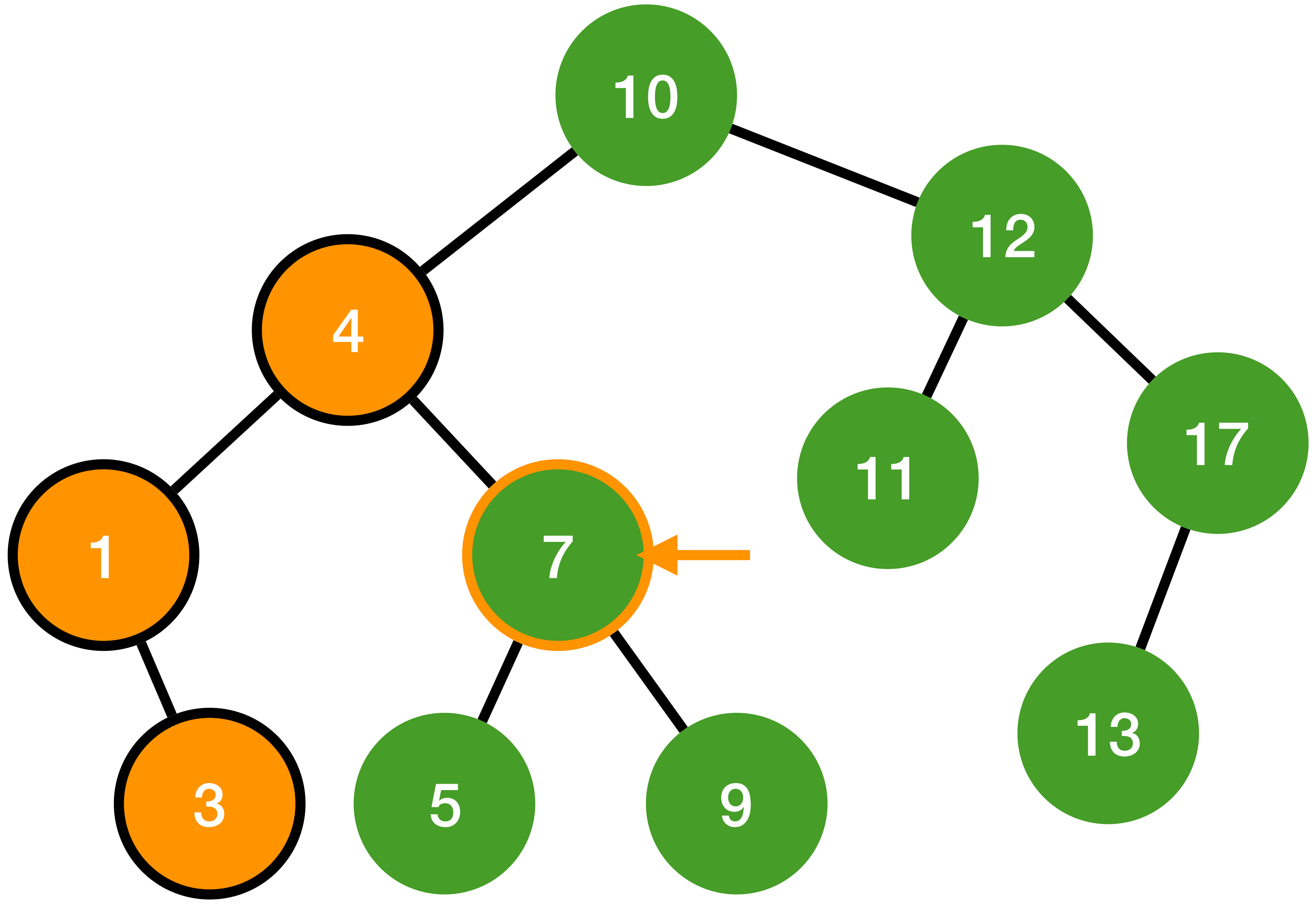
In order = 1

Binary search tree traversal - In order



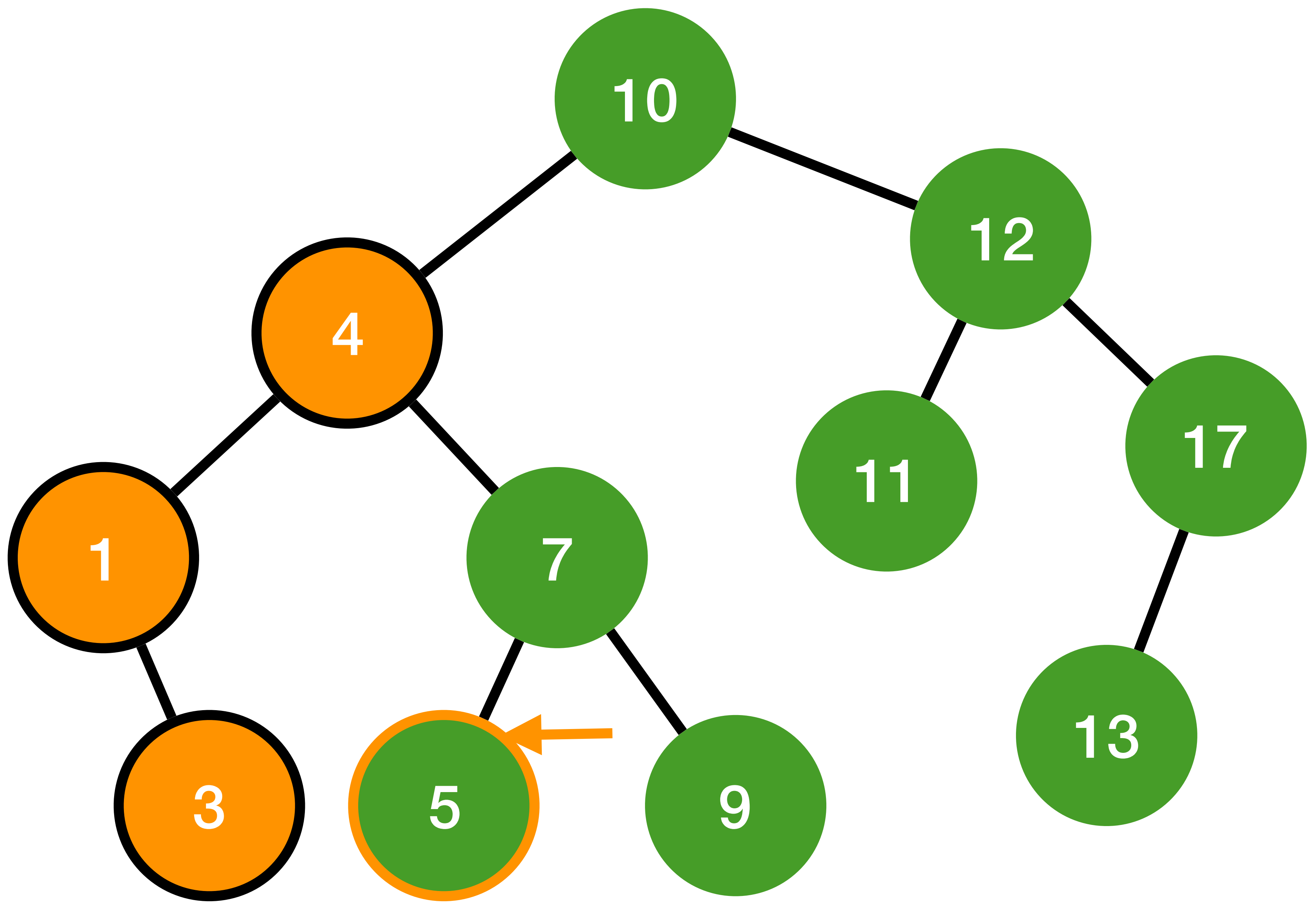
In order =	1	3

Binary search tree traversal - In order



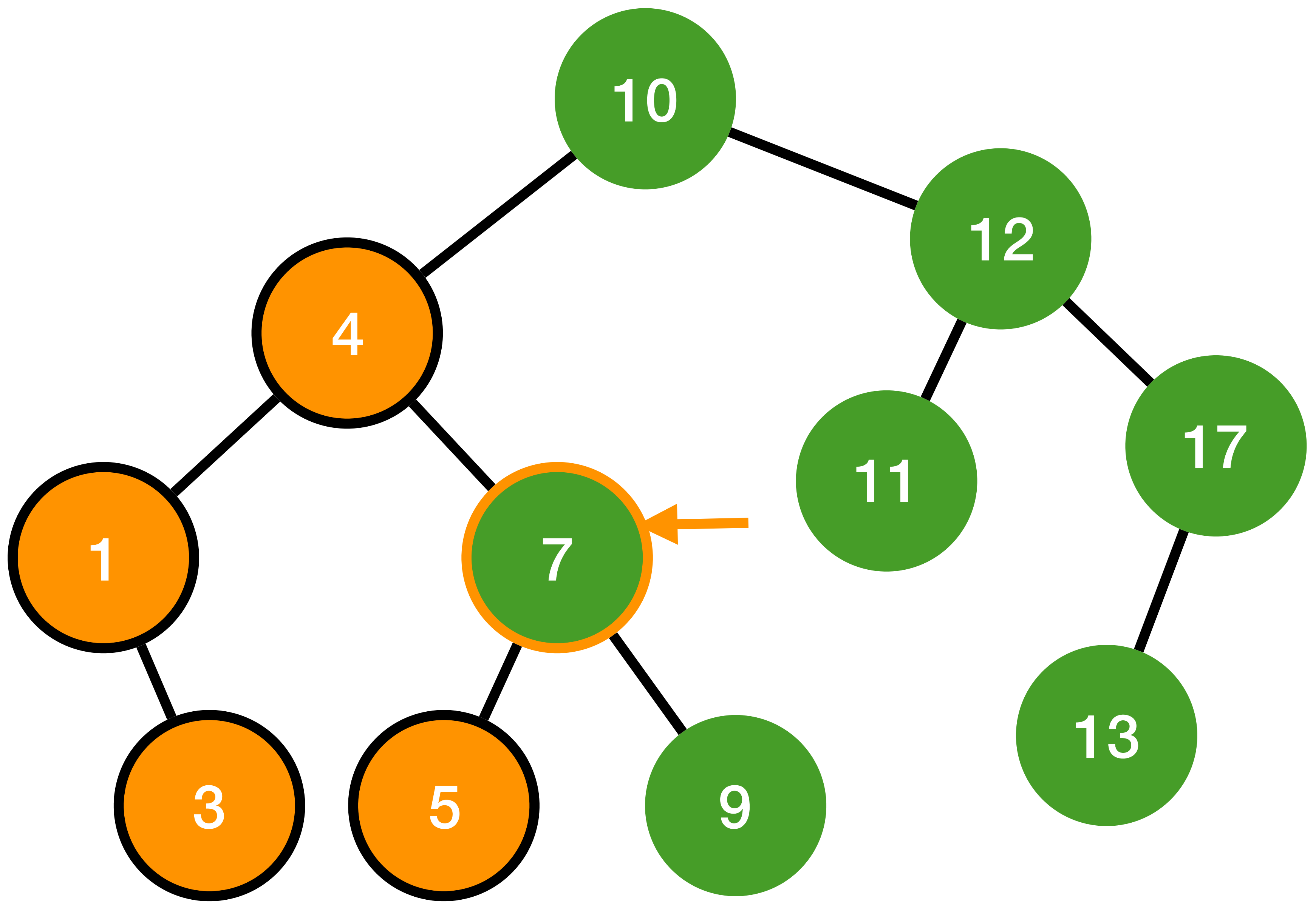
In order =			
	1	3	4

Binary search tree traversal - In order



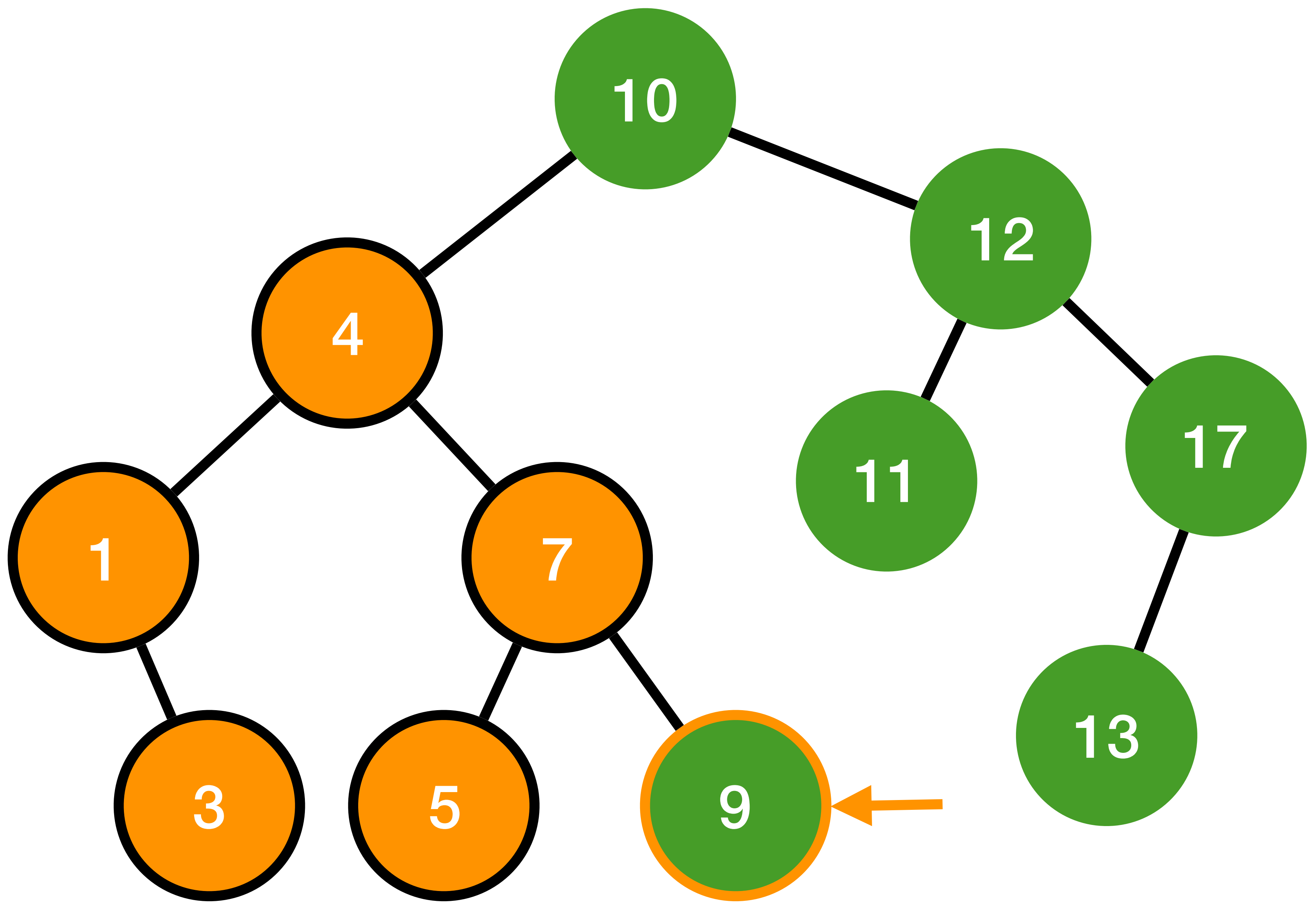
In order =	1	3	4

Binary search tree traversal - In order



In order =				
	1	3	4	5

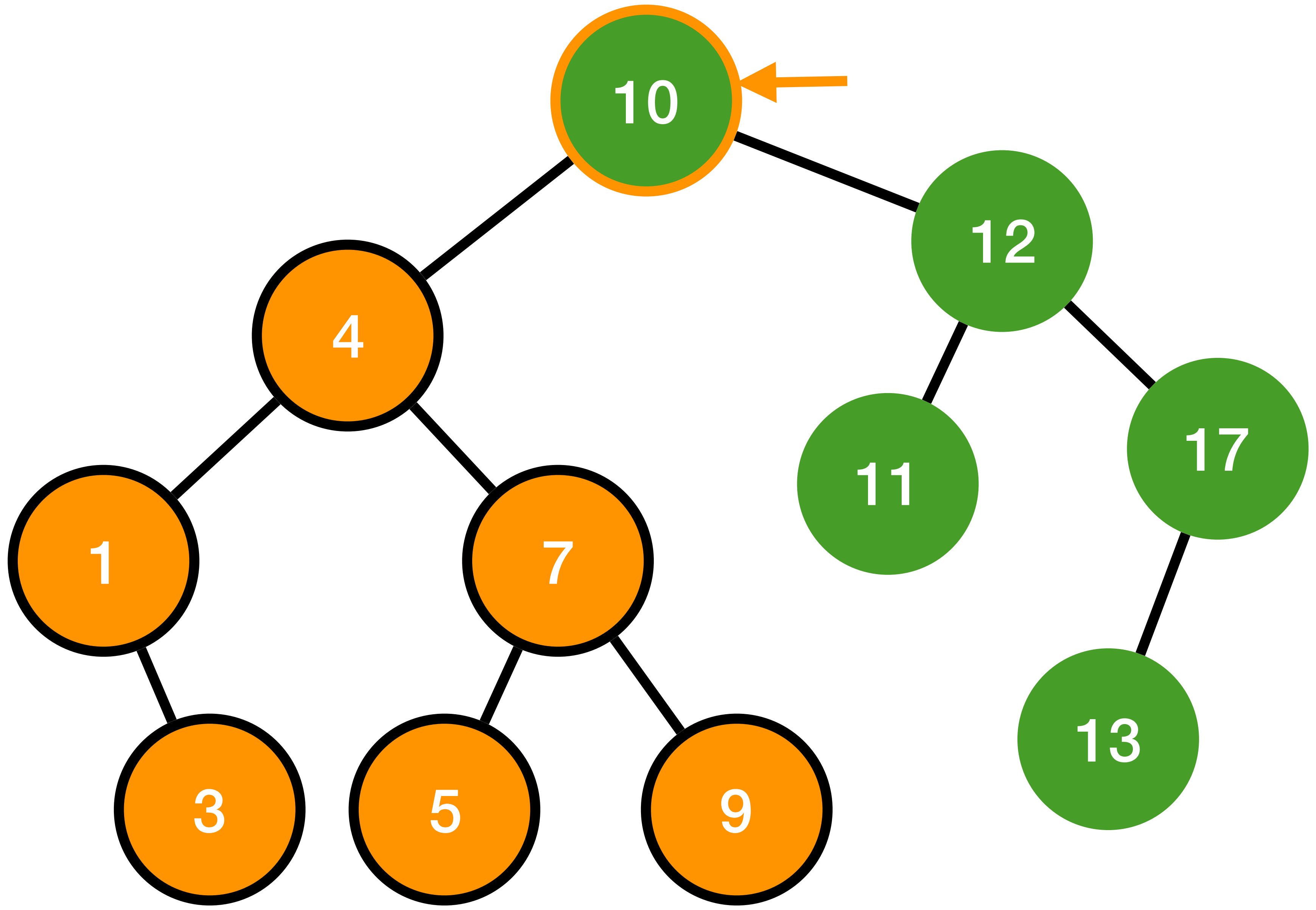
Binary search tree traversal - In order



In order =

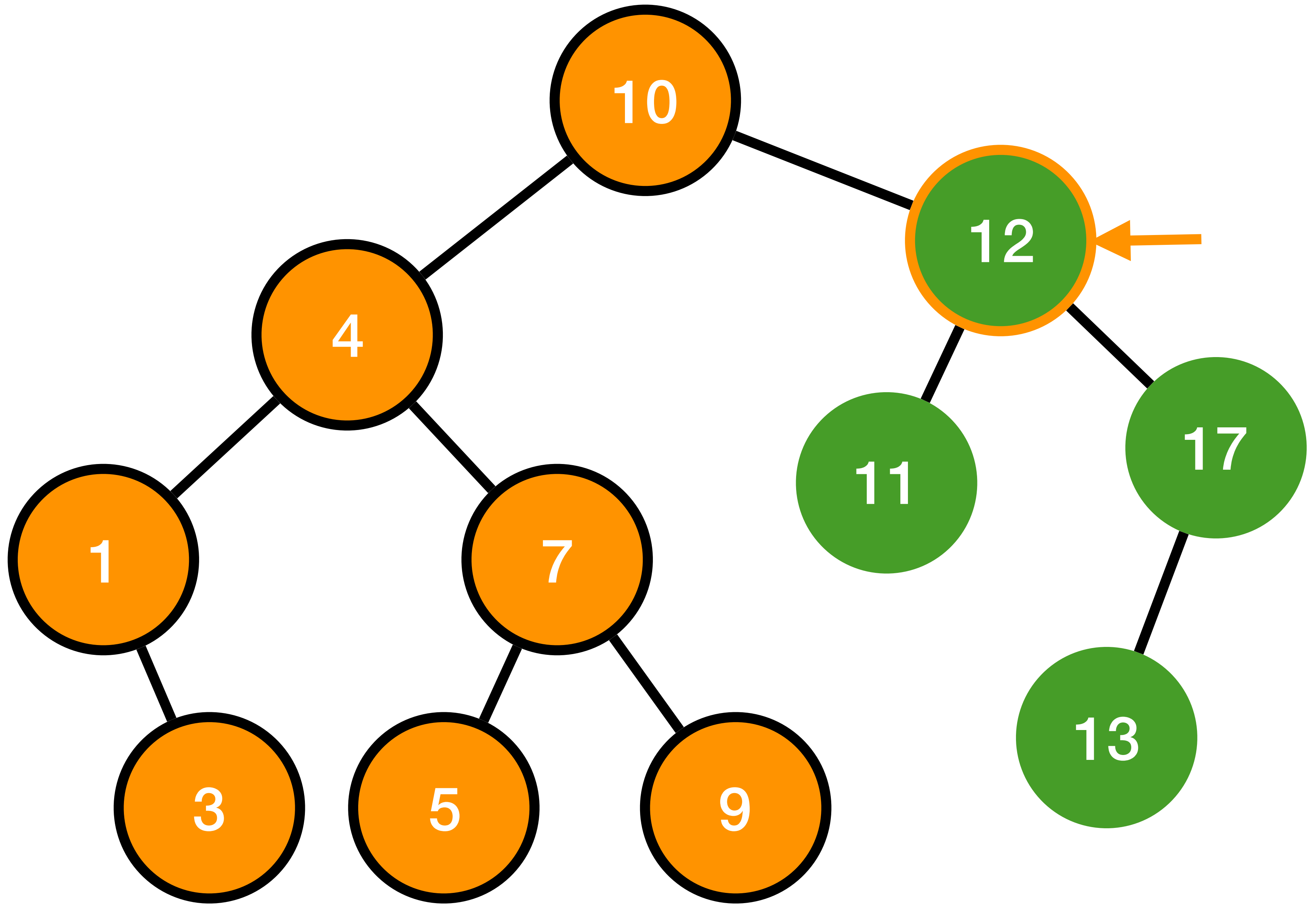
1	3	4	5	7

Binary search tree traversal - In order



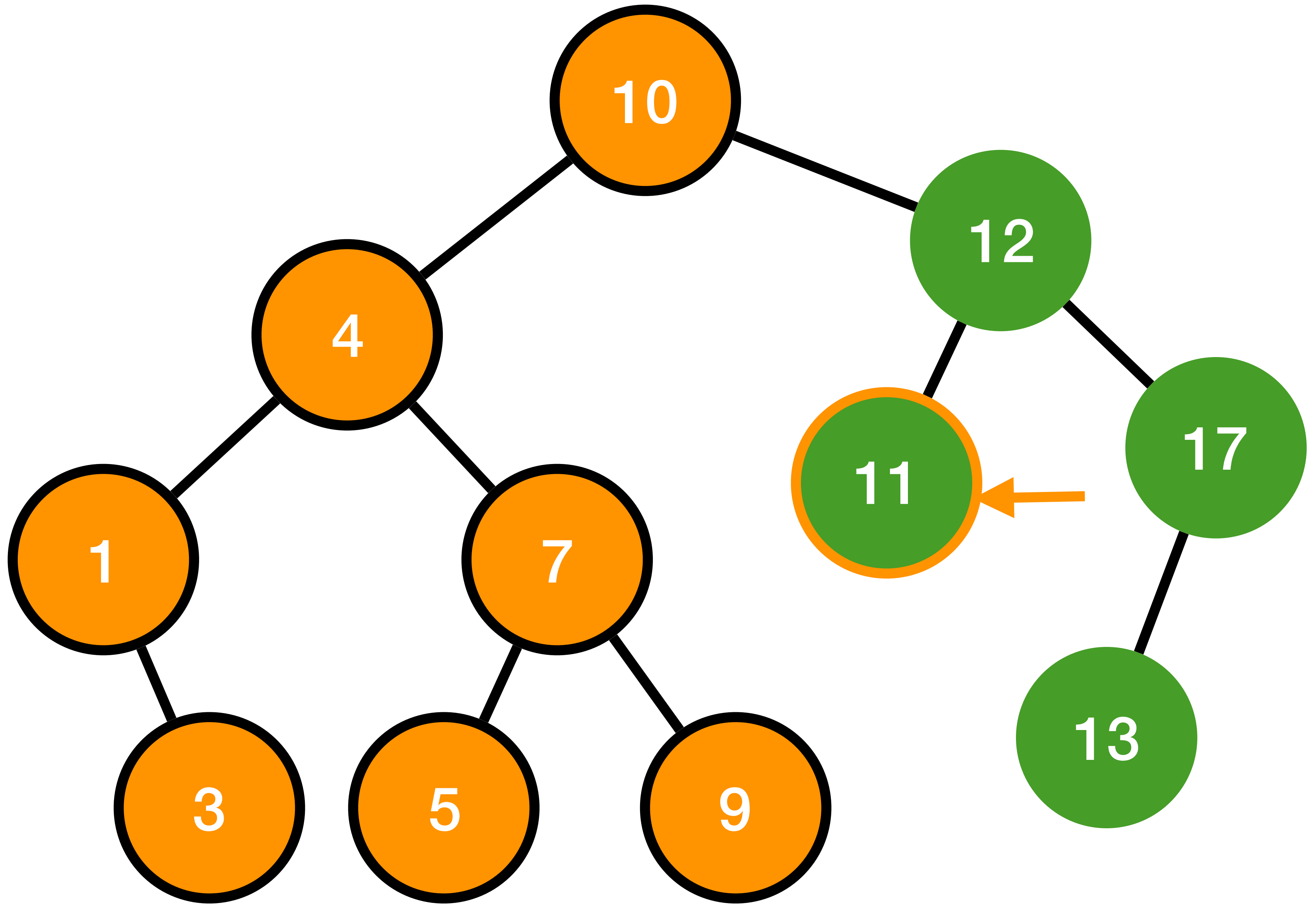
In order =	1	3	4	5	7	9

Binary search tree traversal - In order



In order =	1	3	4	5	7	9	10

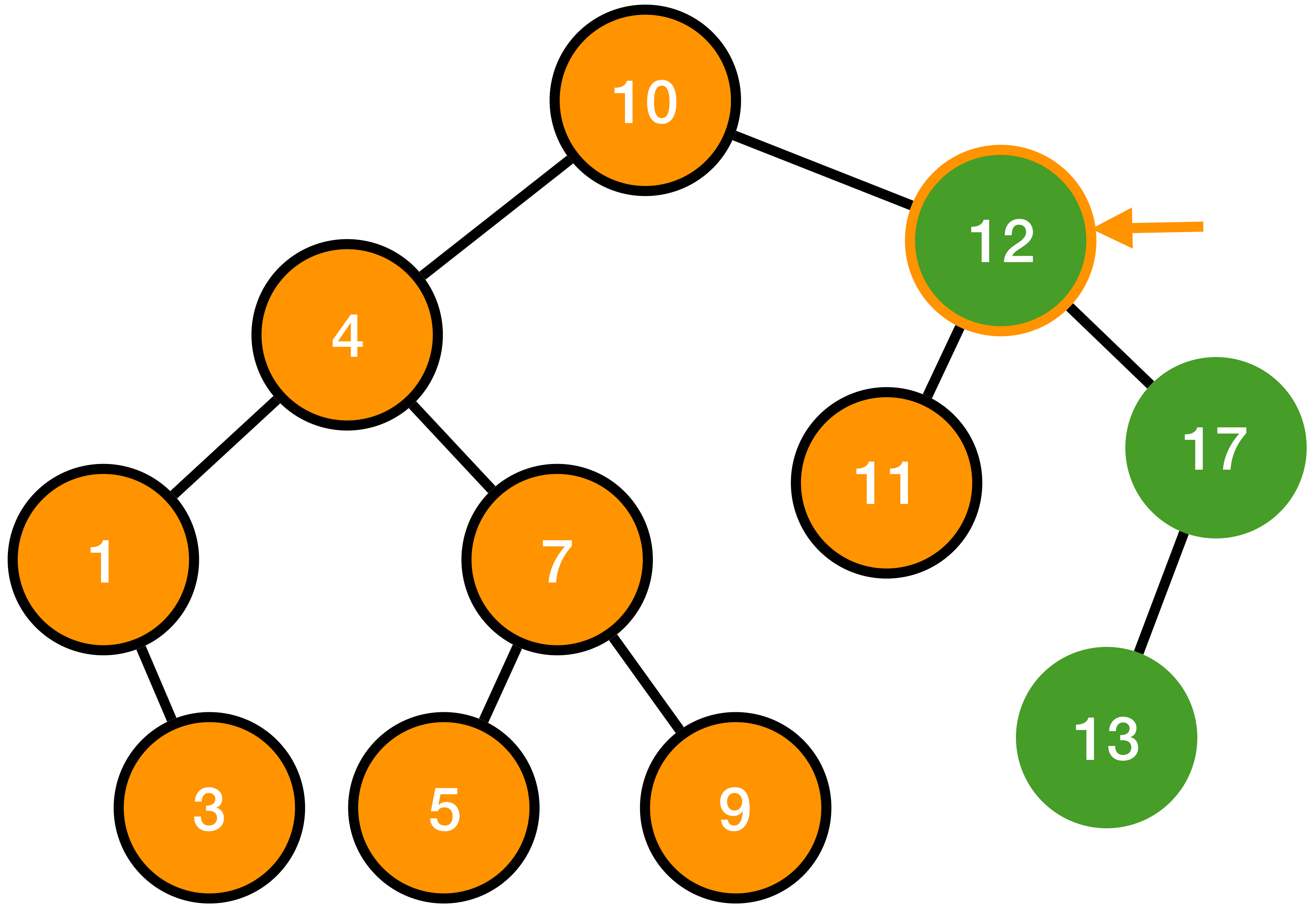
Binary search tree traversal - In order



In order =

1	3	4	5	7	9	10

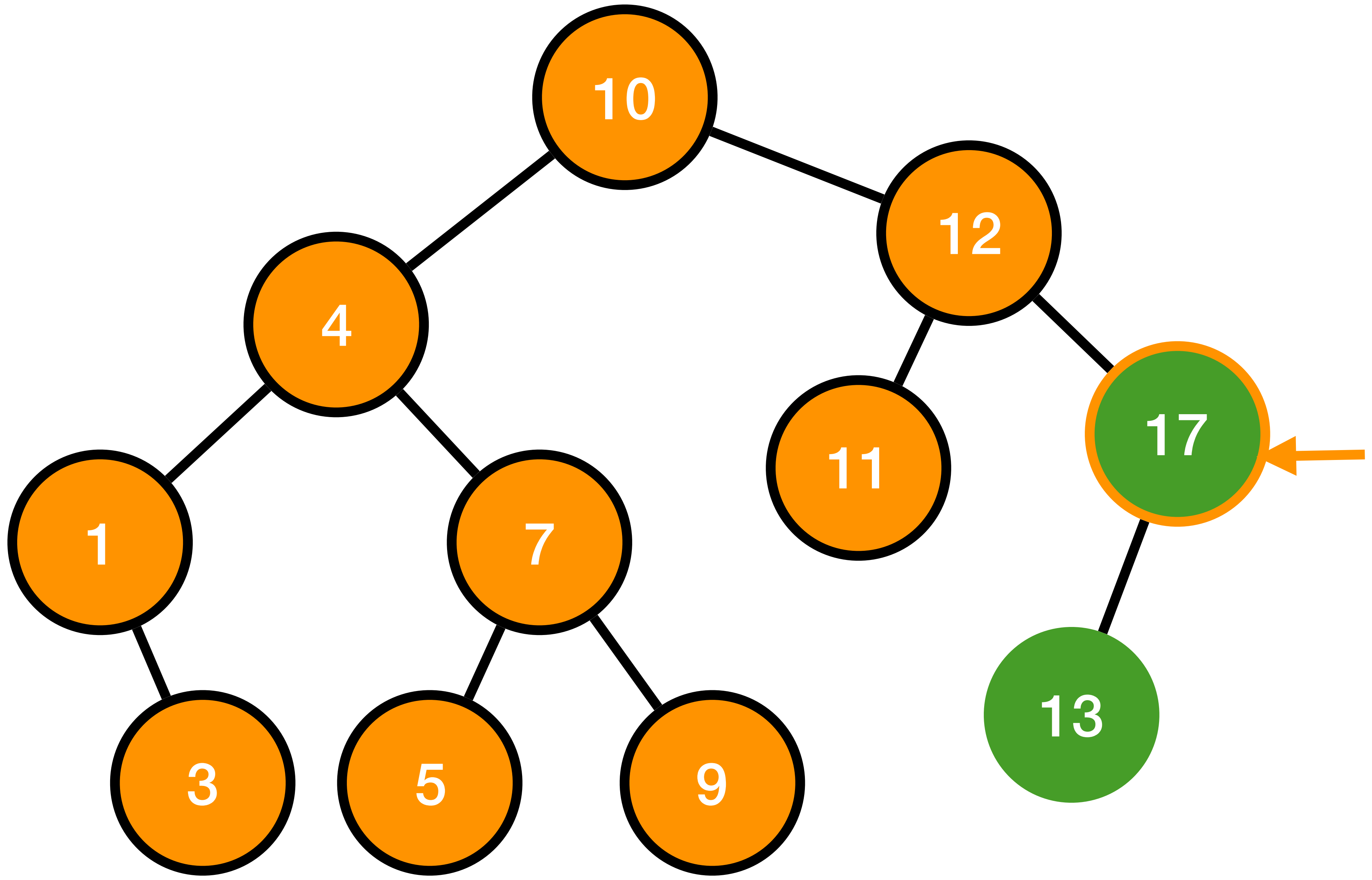
Binary search tree traversal - In order



In order =

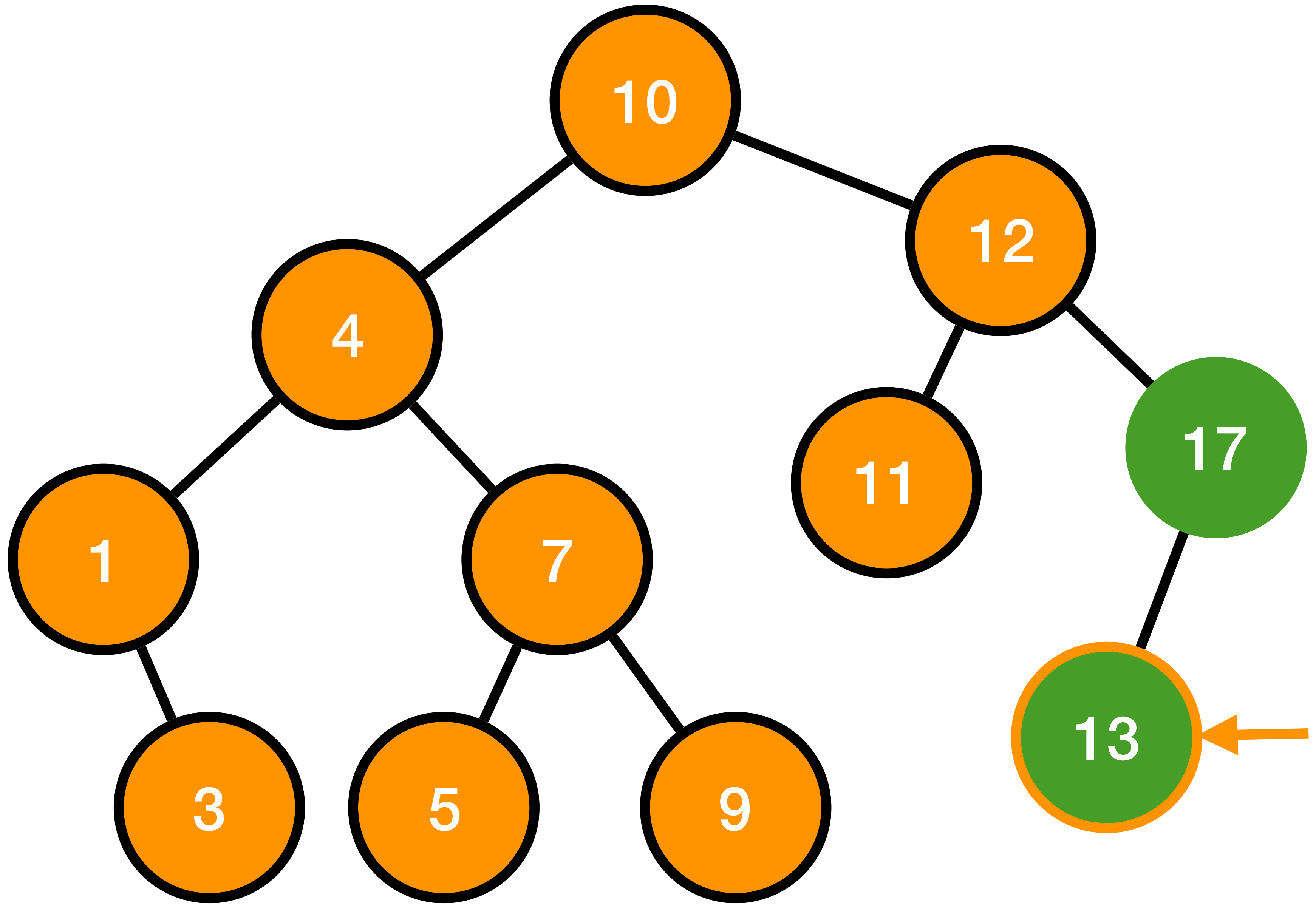
1	3	4	5	7	9	10	11

Binary search tree traversal - In order



In order =	1	3	4	5	7	9	10	11	12

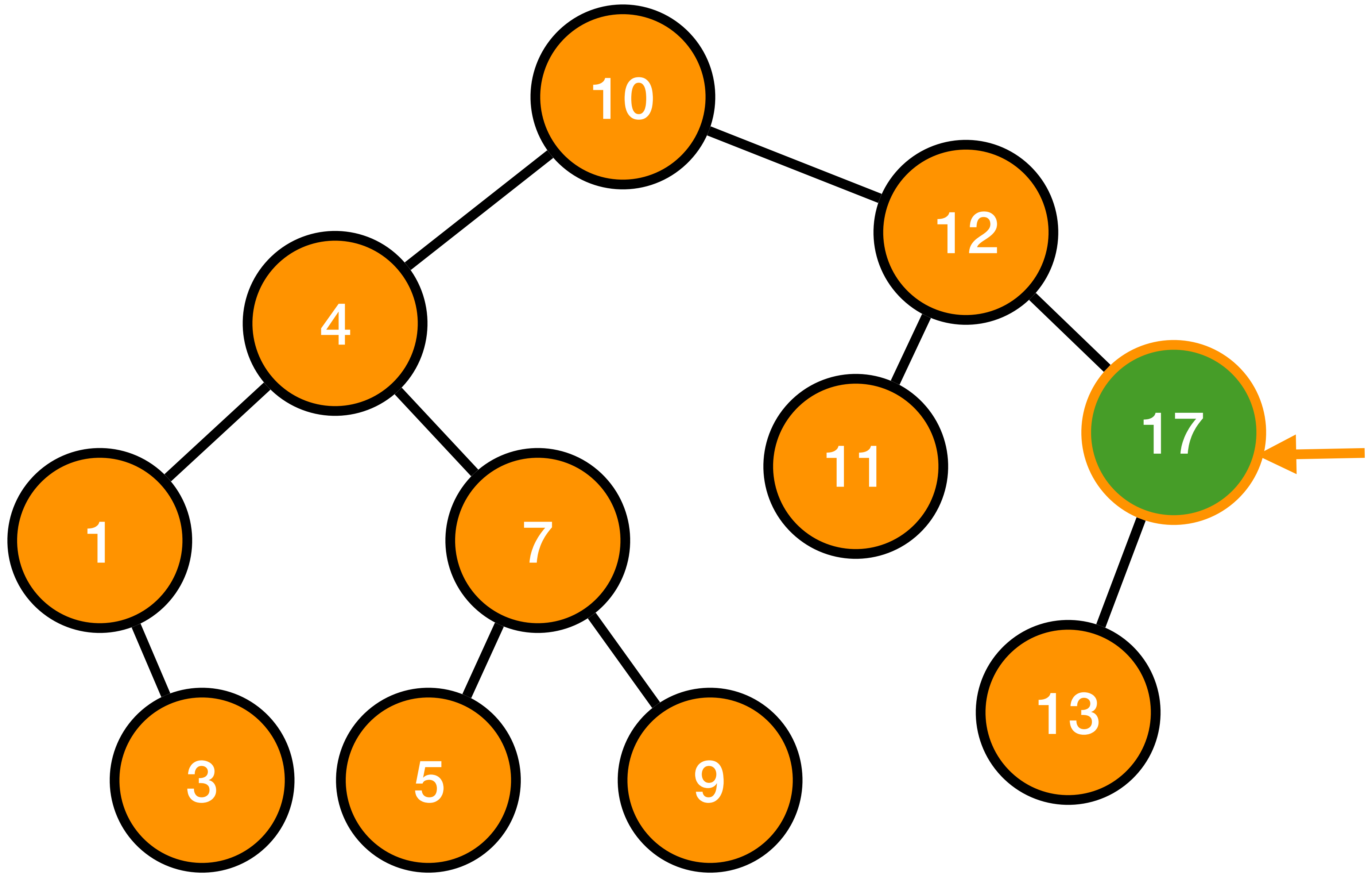
Binary search tree traversal - In order



In order =

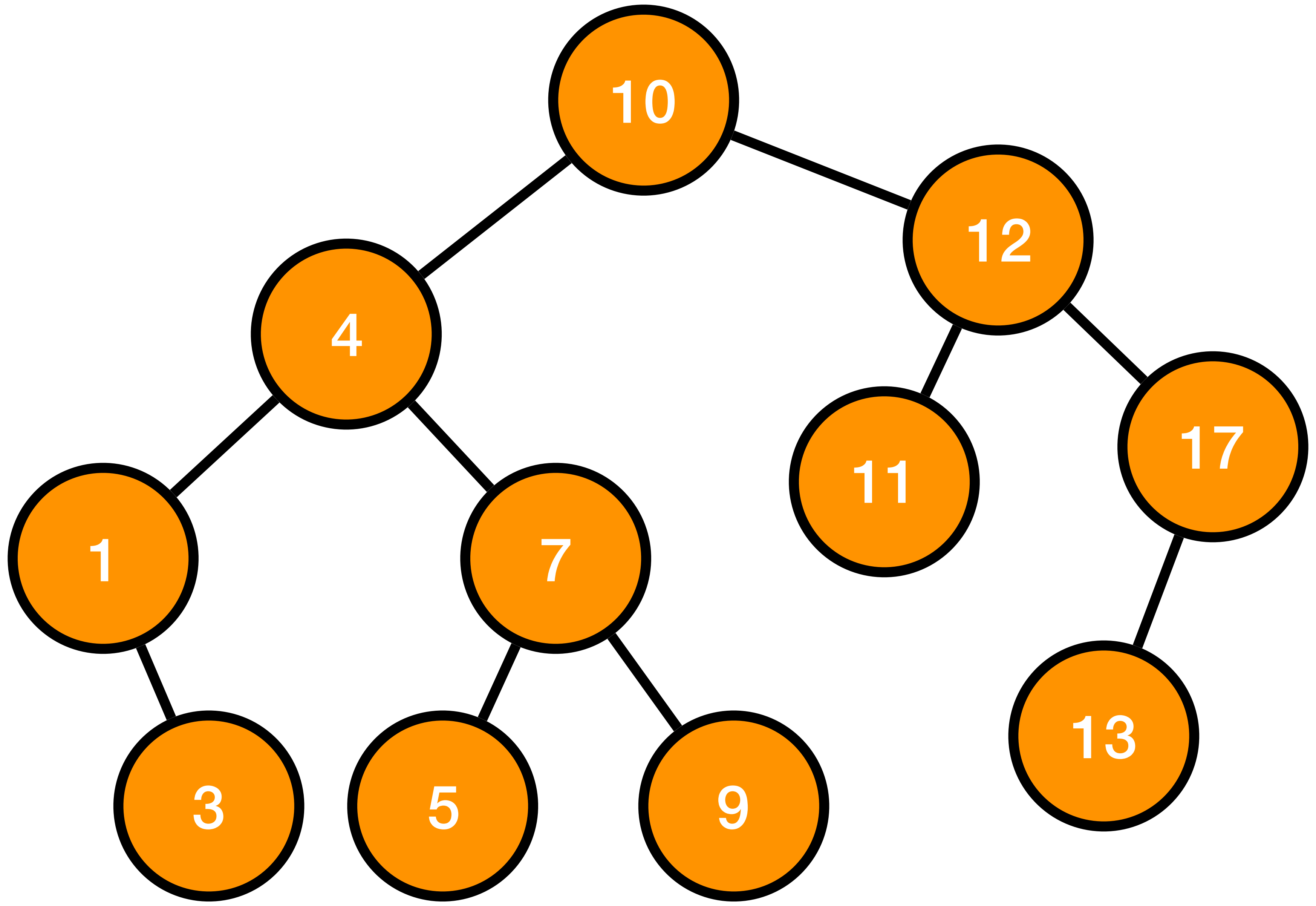
1	3	4	5	7	9	10	11	12

Binary search tree traversal - In order



In order =	1	3	4	5	7	9	10	11	12	13

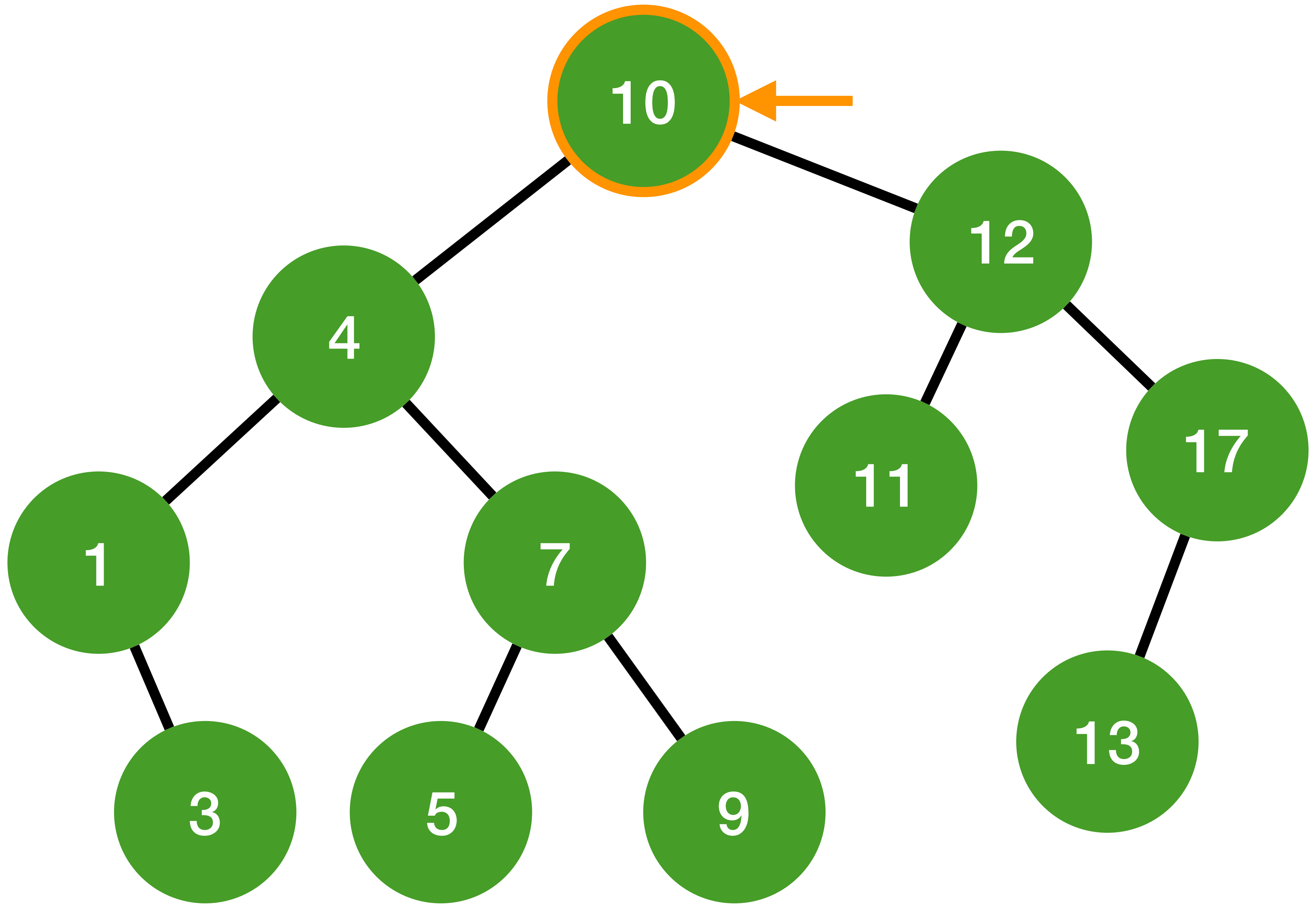
Binary search tree traversal - In order



In order =

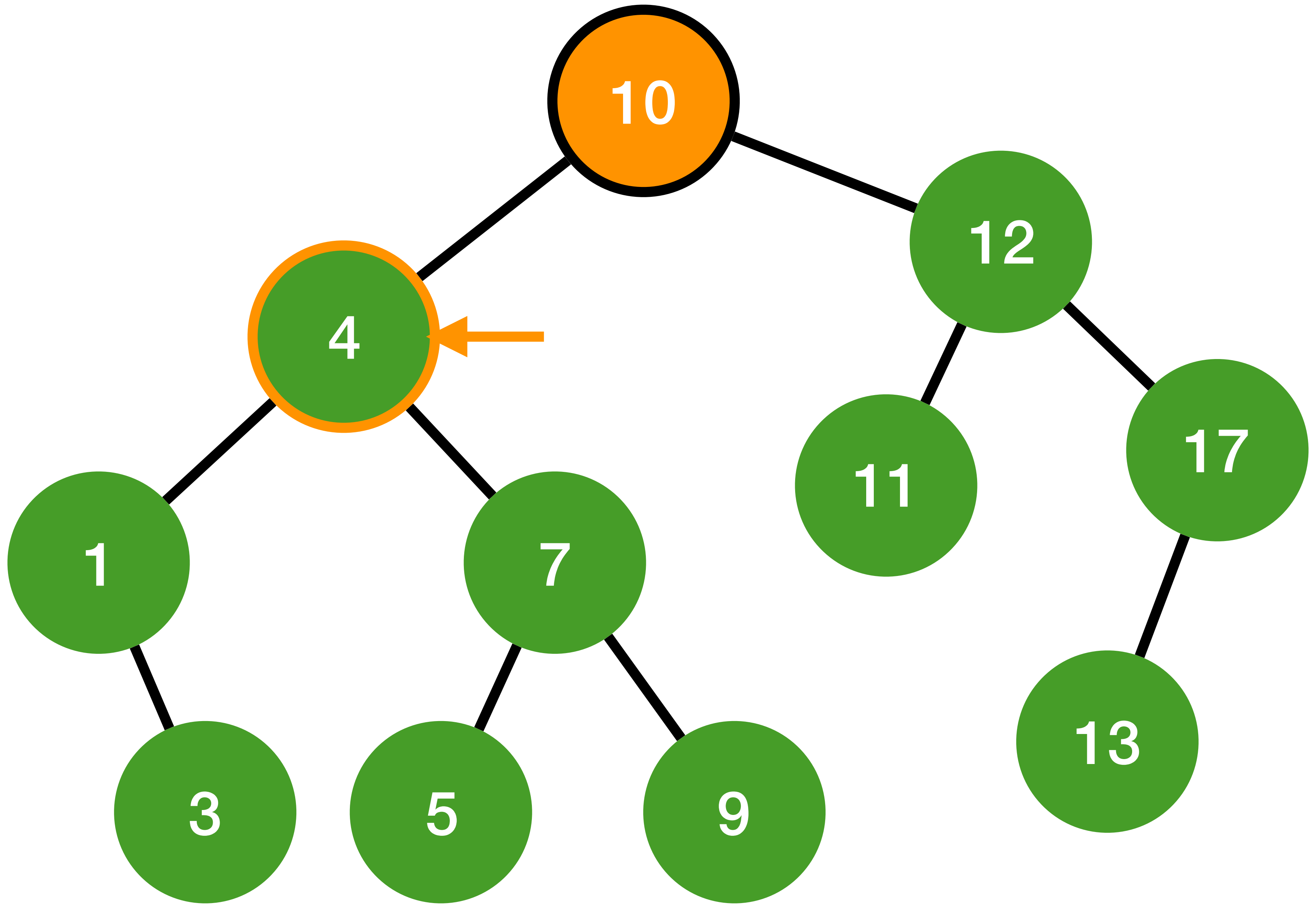
1	3	4	5	7	9	10	11	12	13	17

Binary search tree traversal - Pre order



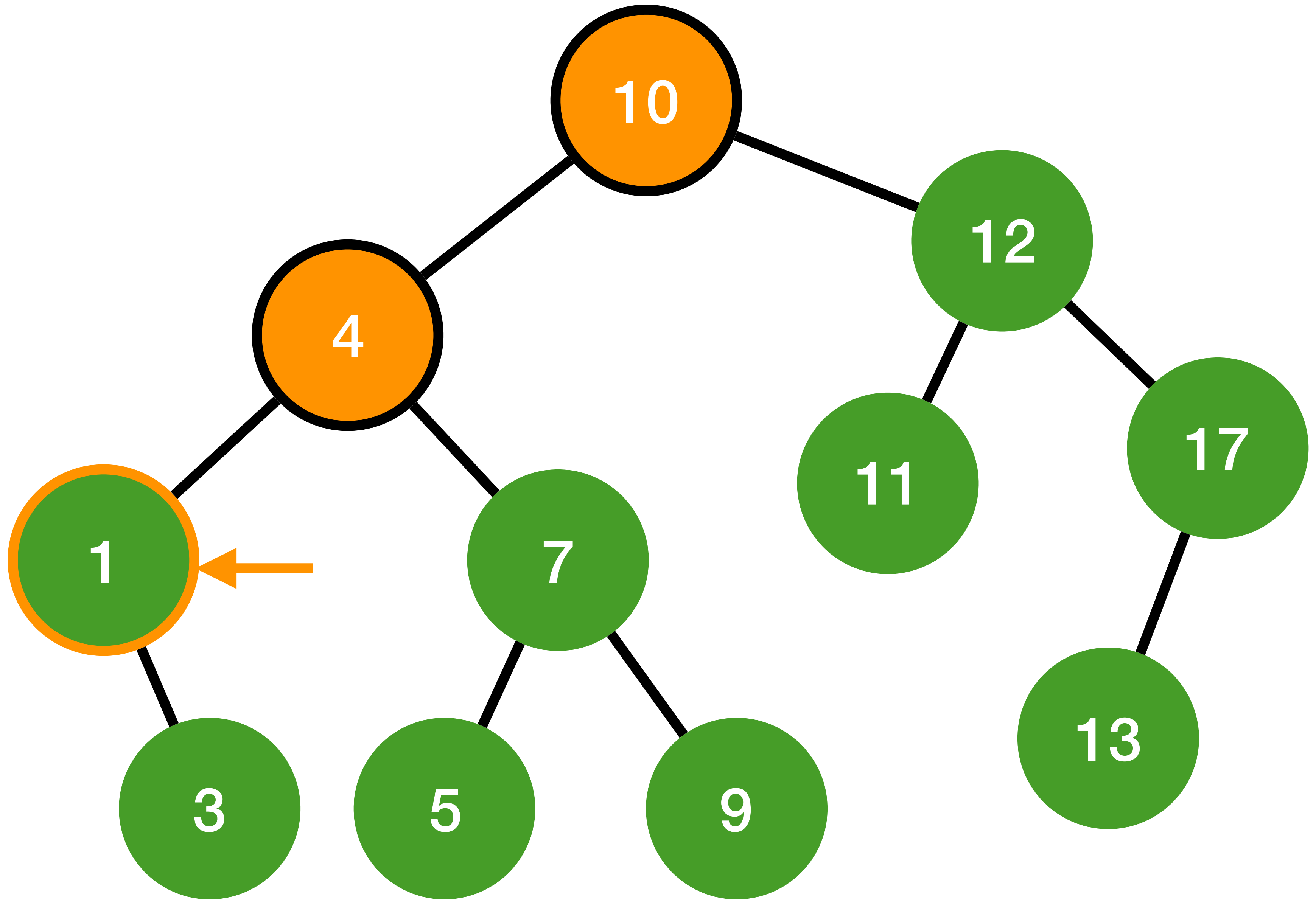
Pre order =

Binary search tree traversal - Pre order



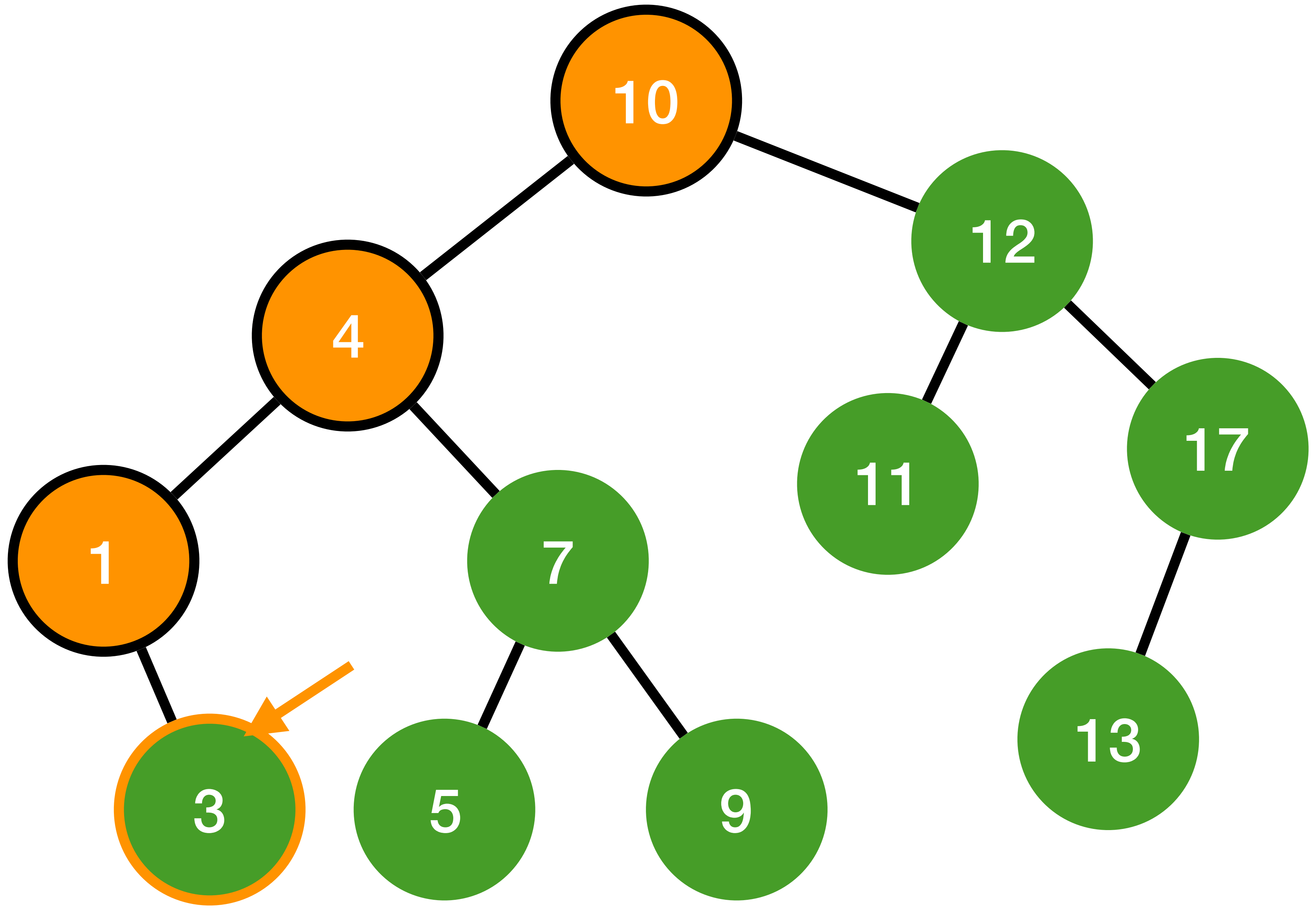
Pre order = 10

Binary search tree traversal - Pre order



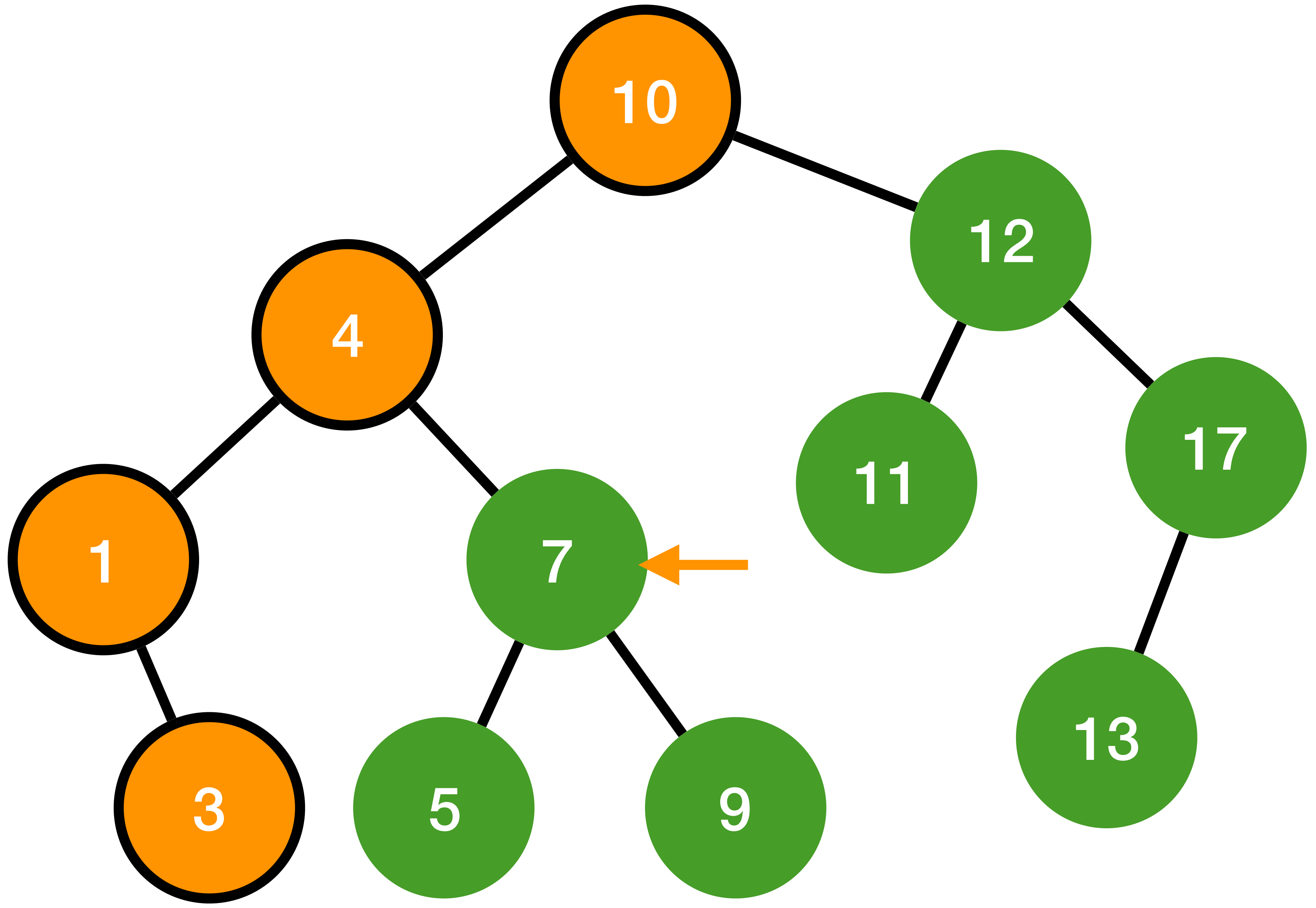
Pre order =	10	4

Binary search tree traversal - Pre order



Pre order =	10	4	1

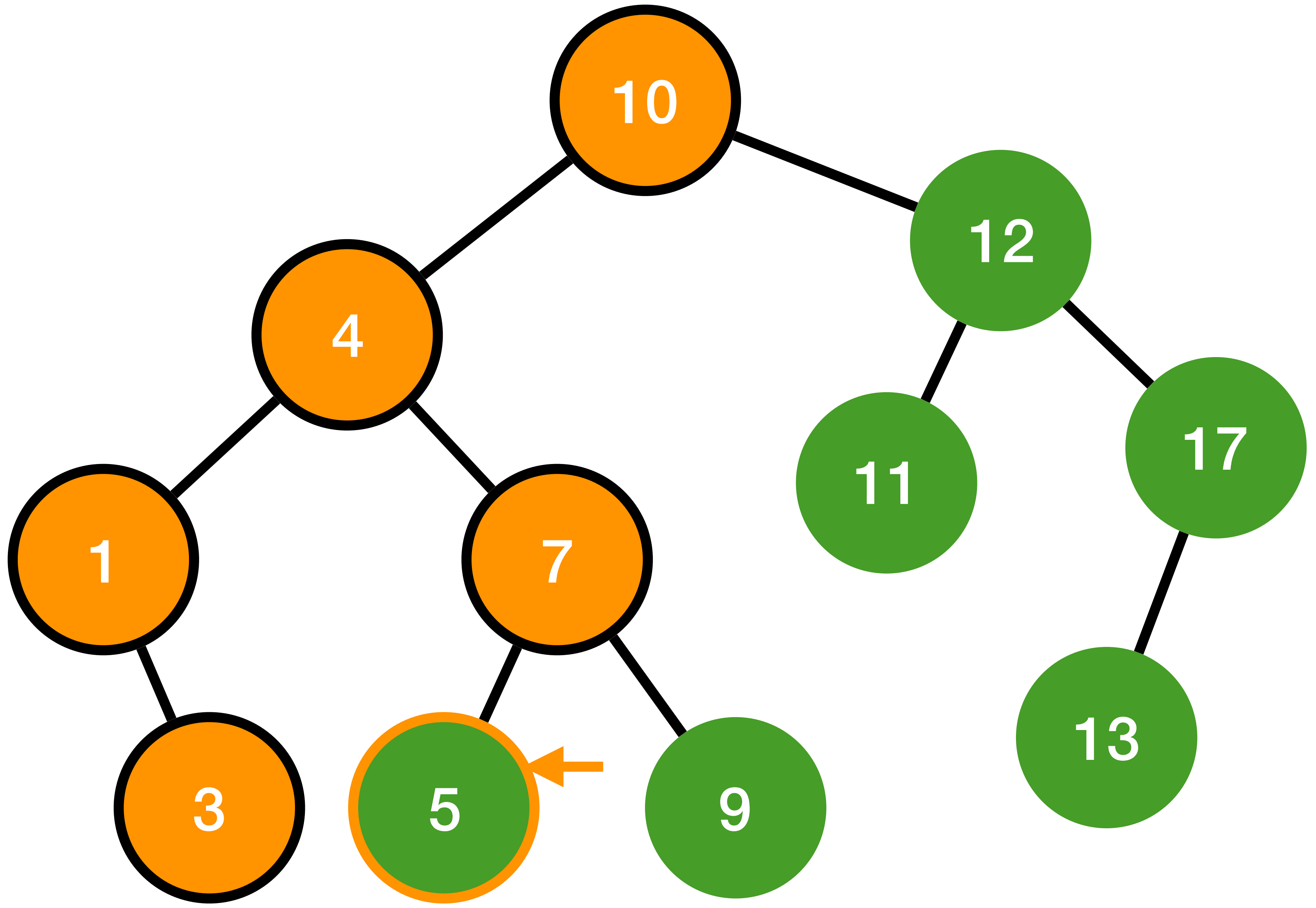
Binary search tree traversal - Pre order



Pre order =

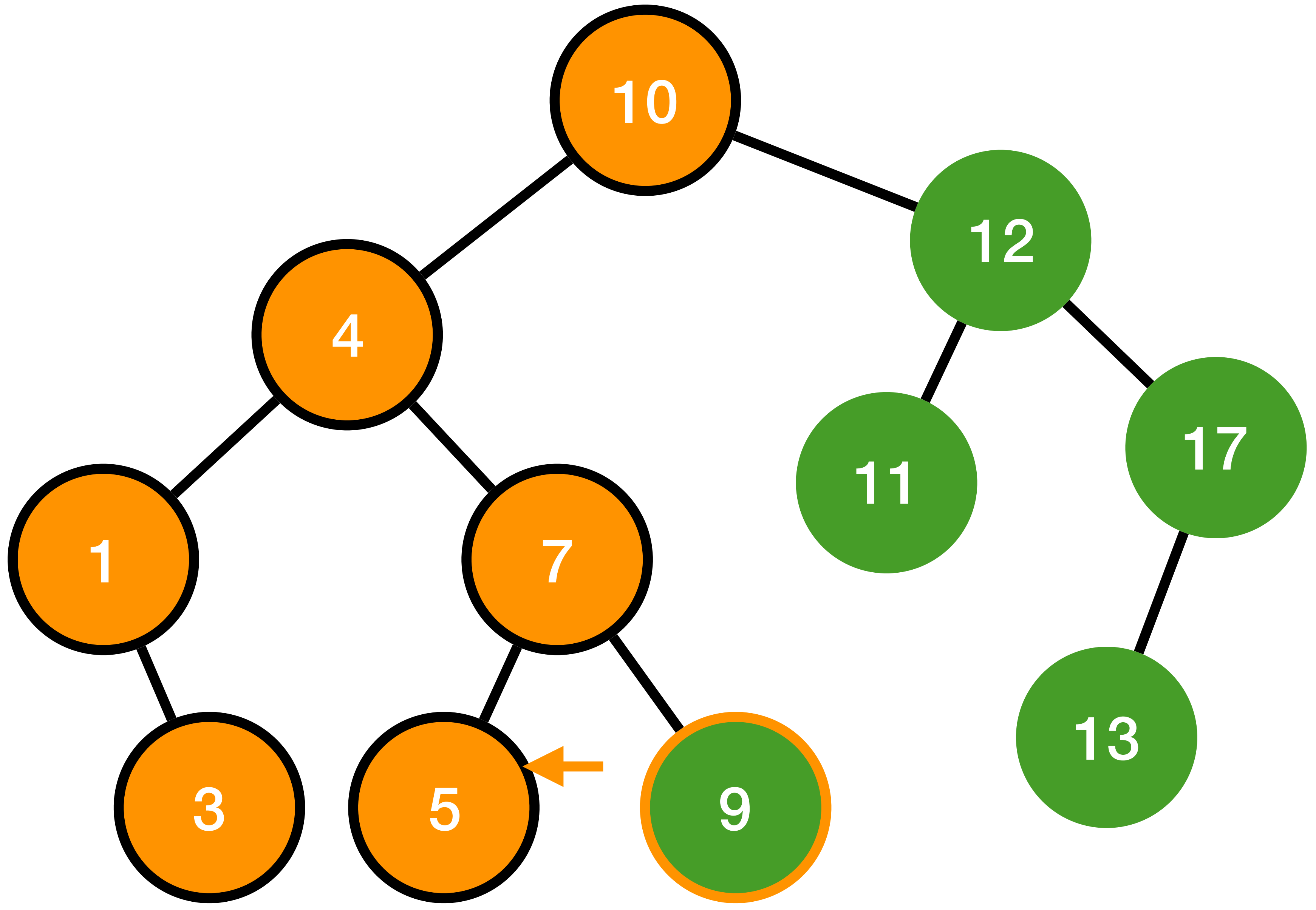
10	4	1	3

Binary search tree traversal - Pre order



Pre order =	10	4	1	3	7

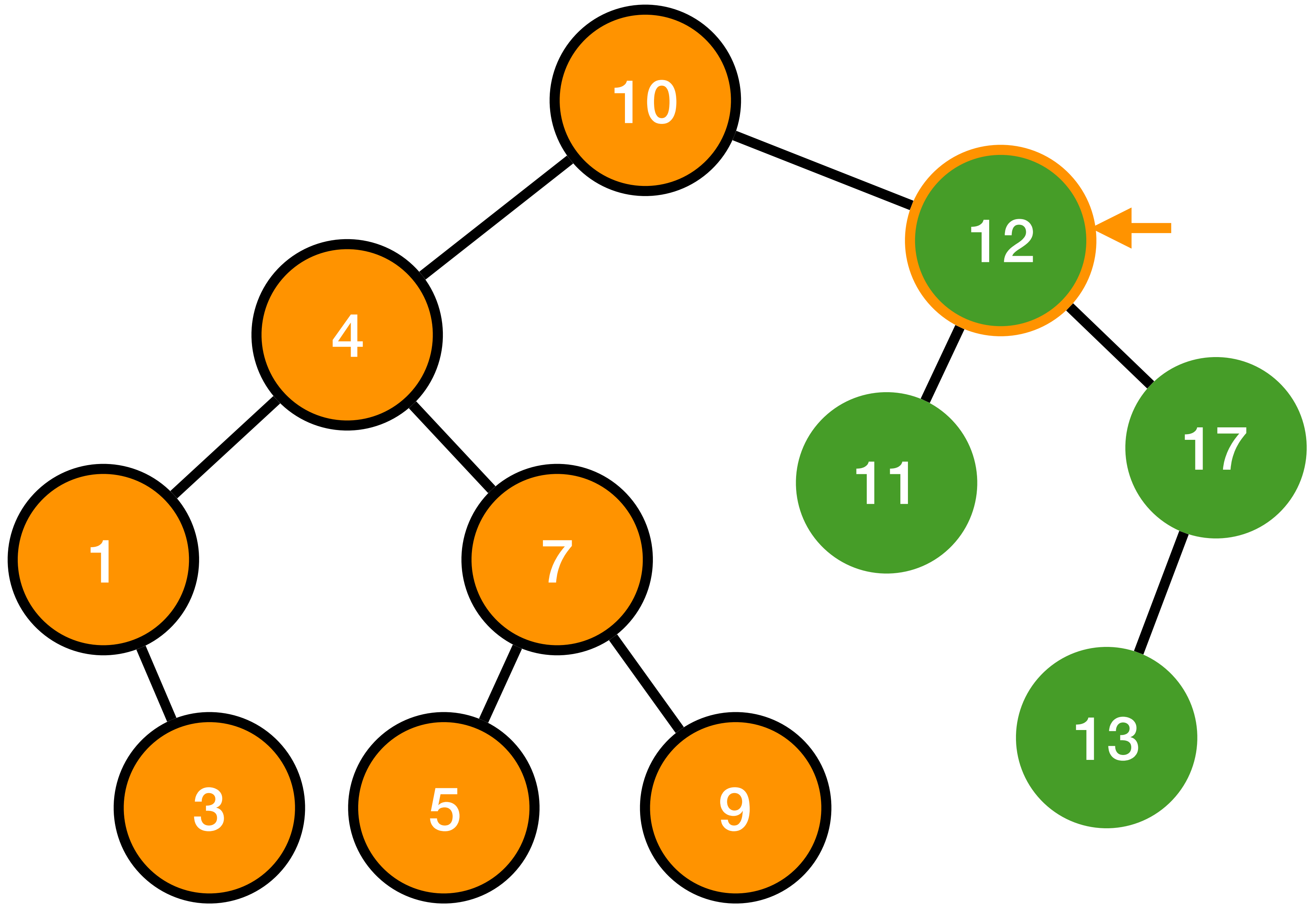
Binary search tree traversal - Pre order



Pre order =

10	4	1	3	7	5

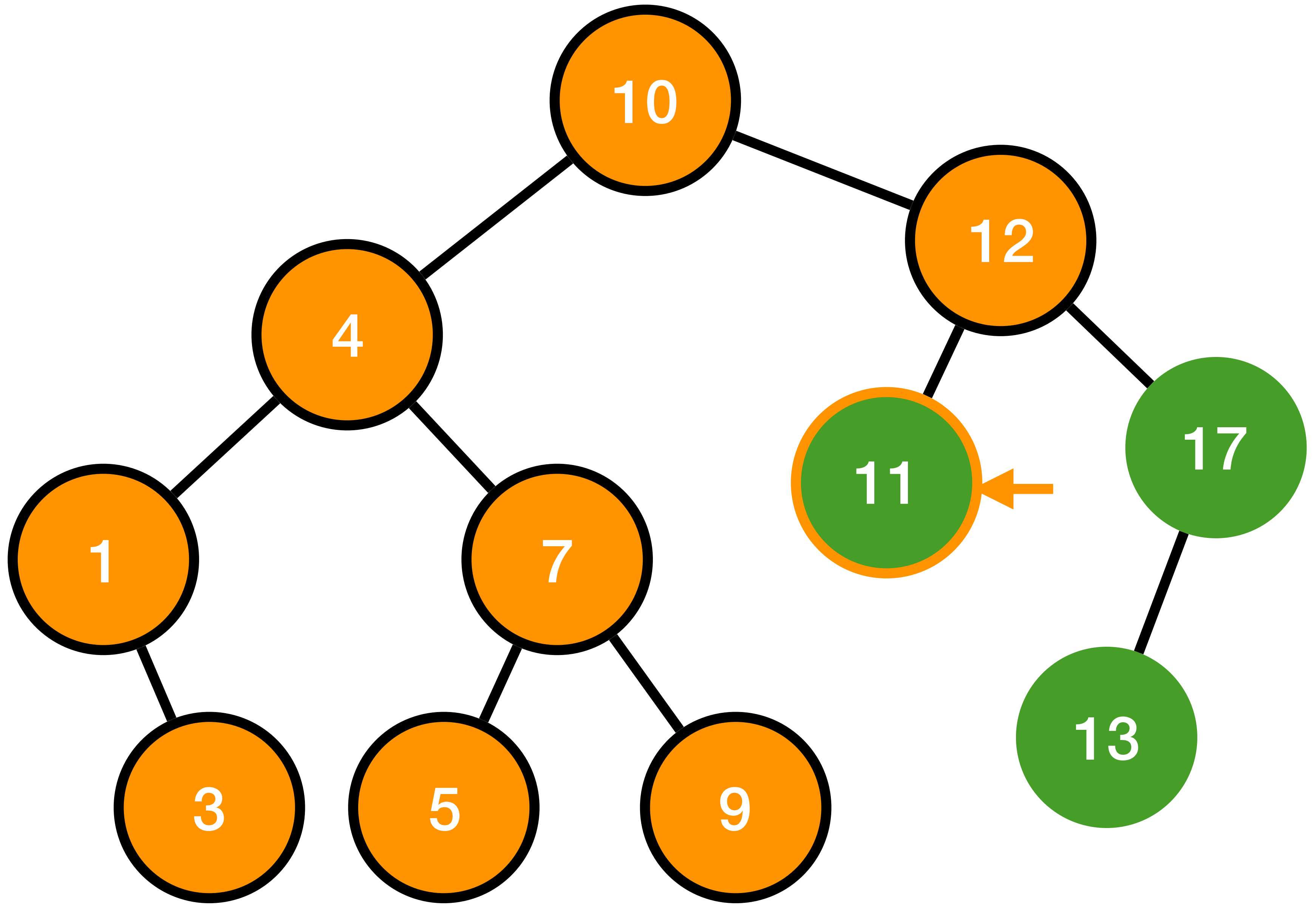
Binary search tree traversal - Pre order



Pre order =

10	4	1	3	7	5	9

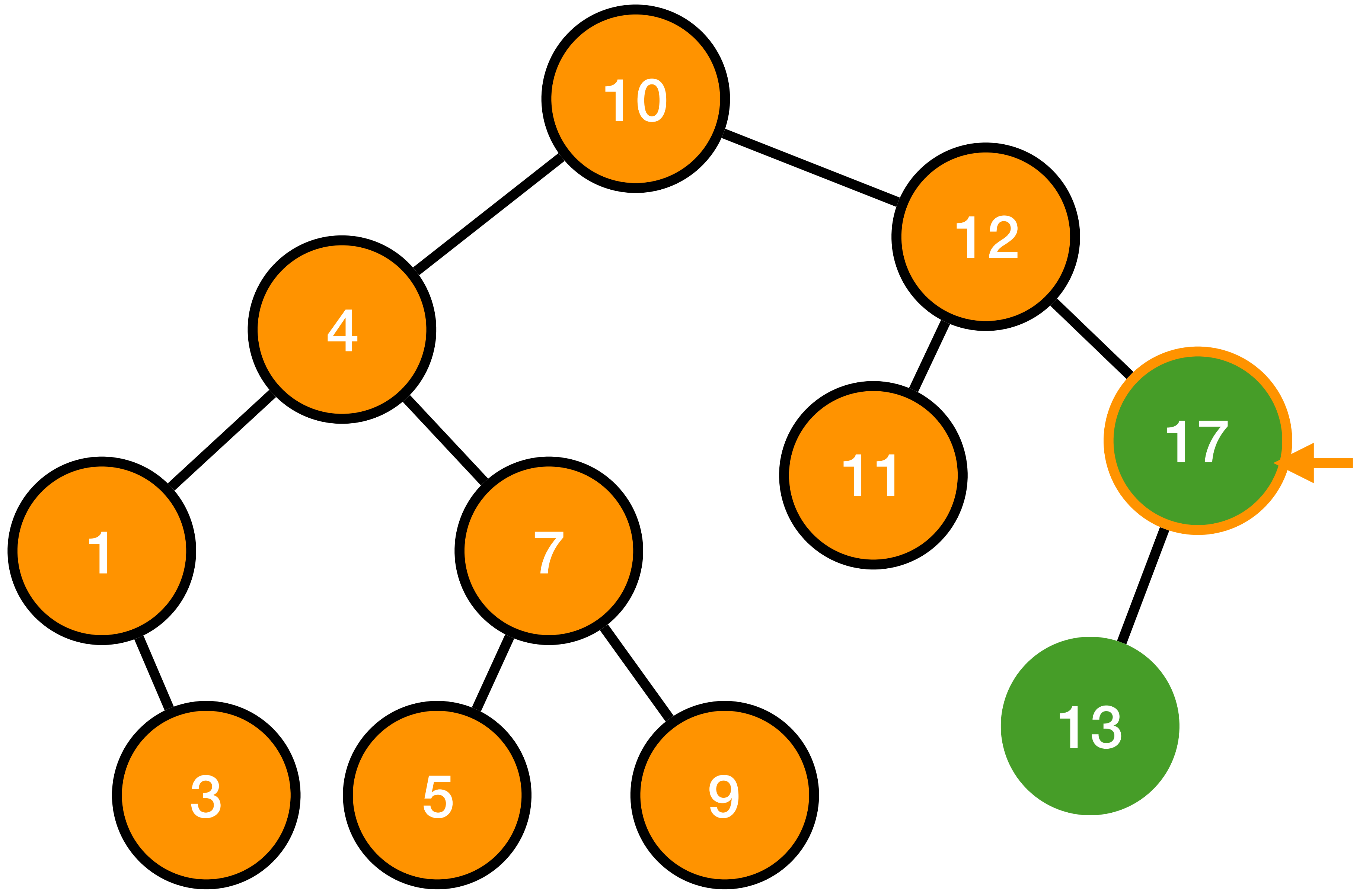
Binary search tree traversal - Pre order



Pre order =

10	4	1	3	7	5	9	12

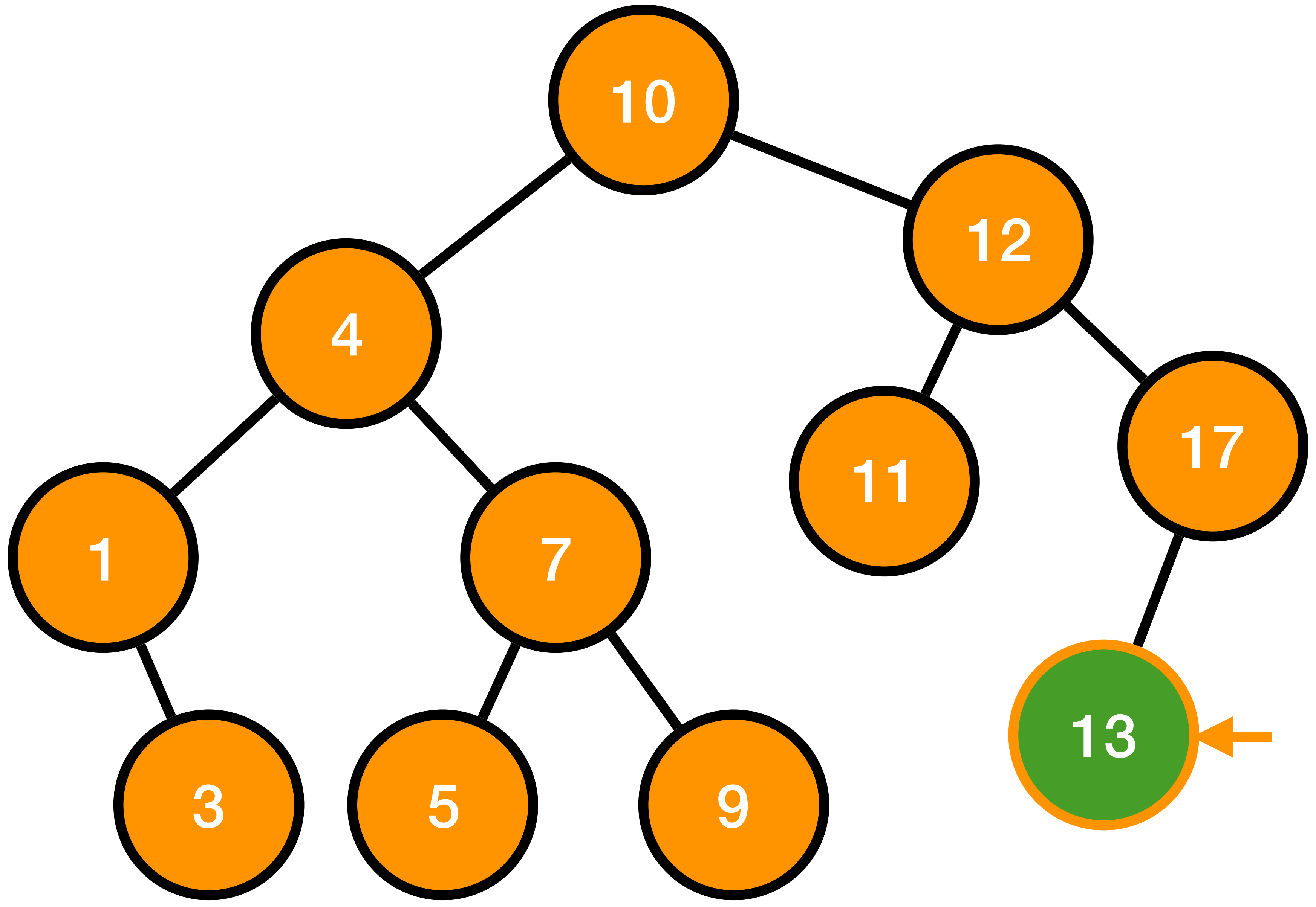
Binary search tree traversal - Pre order



Pre order =

10	4	1	3	7	5	9	12	11

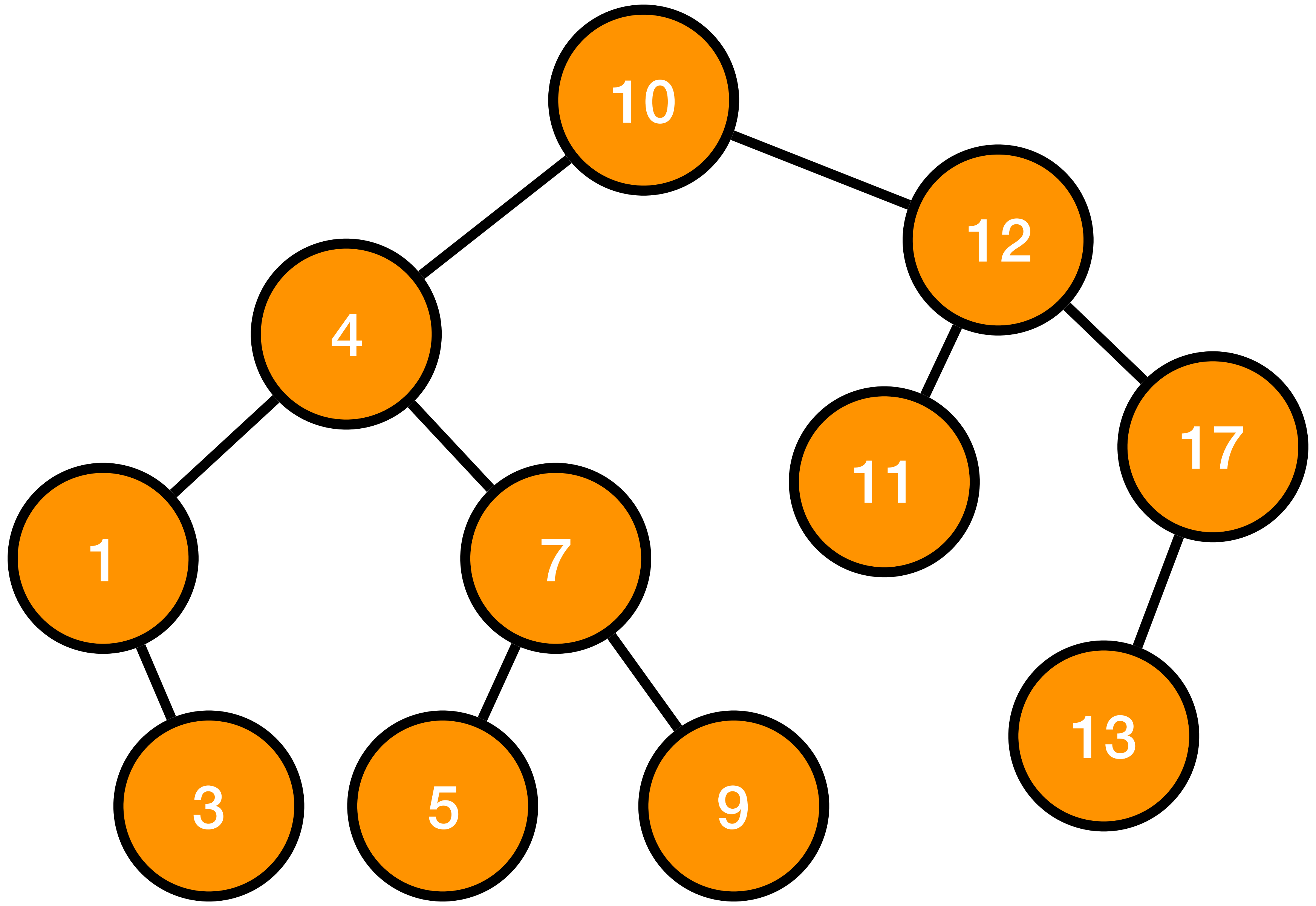
Binary search tree traversal - Pre order



Pre order =

10	4	1	3	7	5	9	12	11	17

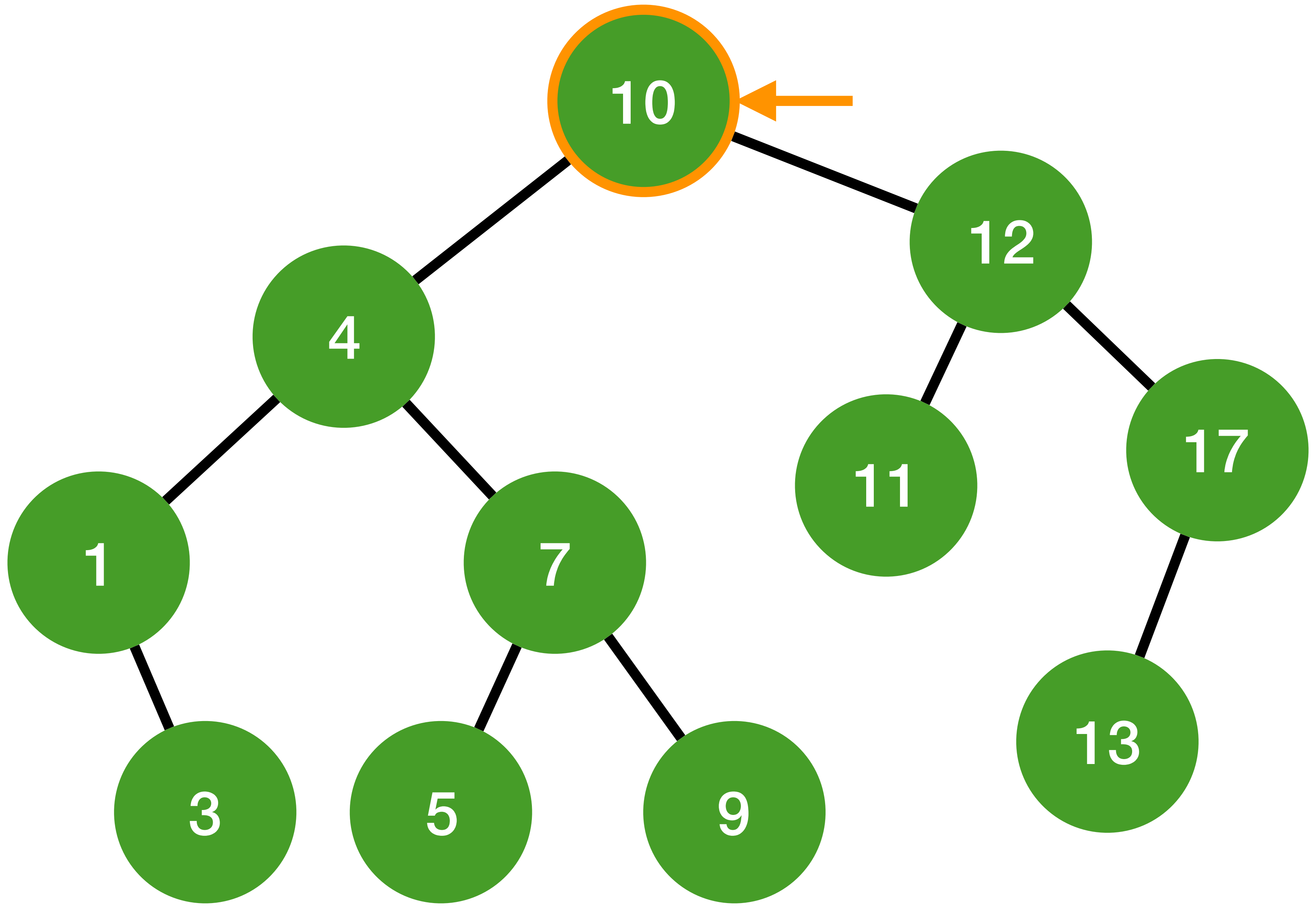
Binary search tree traversal - Pre order



Pre order =

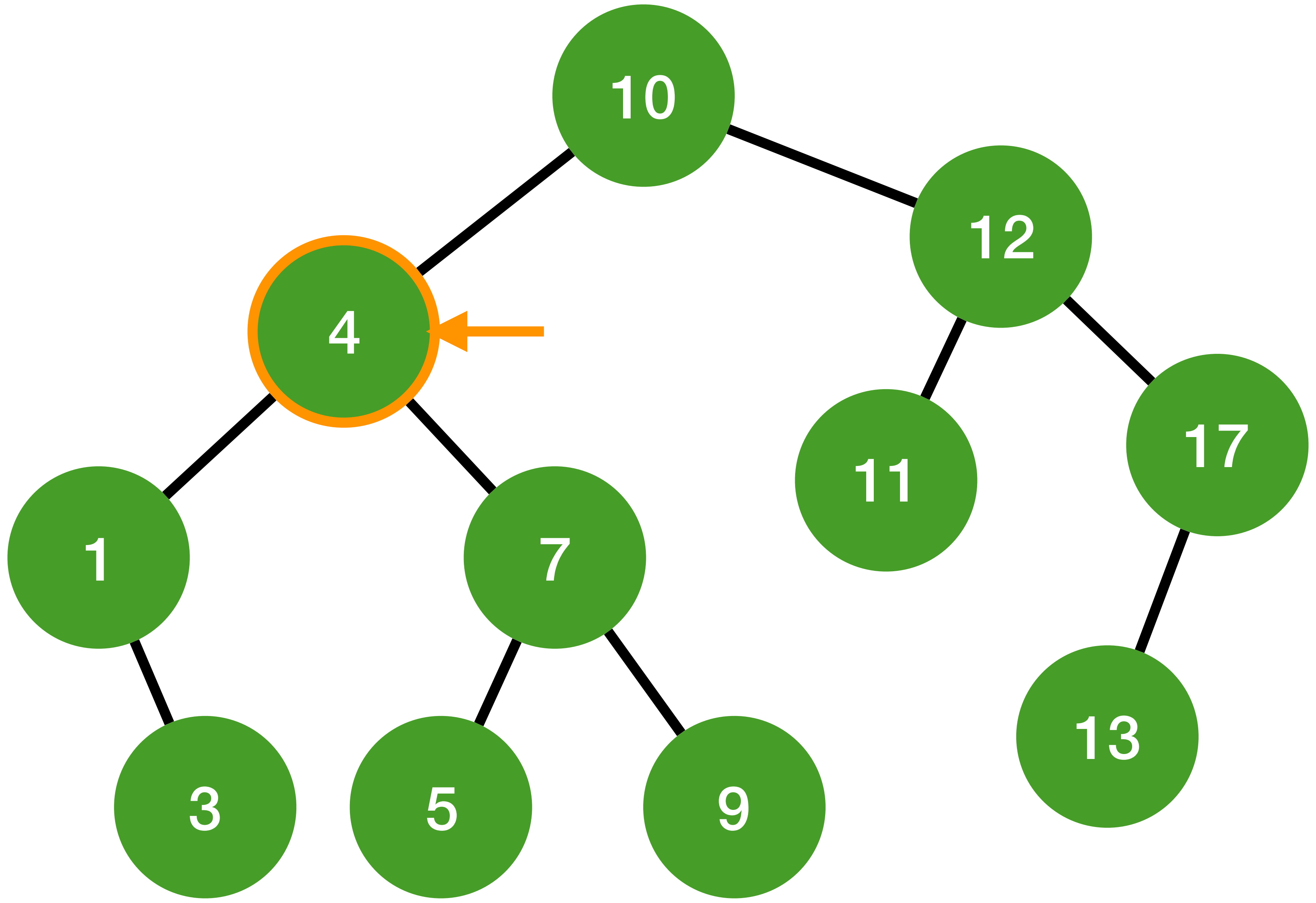
10	4	1	3	7	5	9	12	11	17	13

Binary search tree traversal - Post order



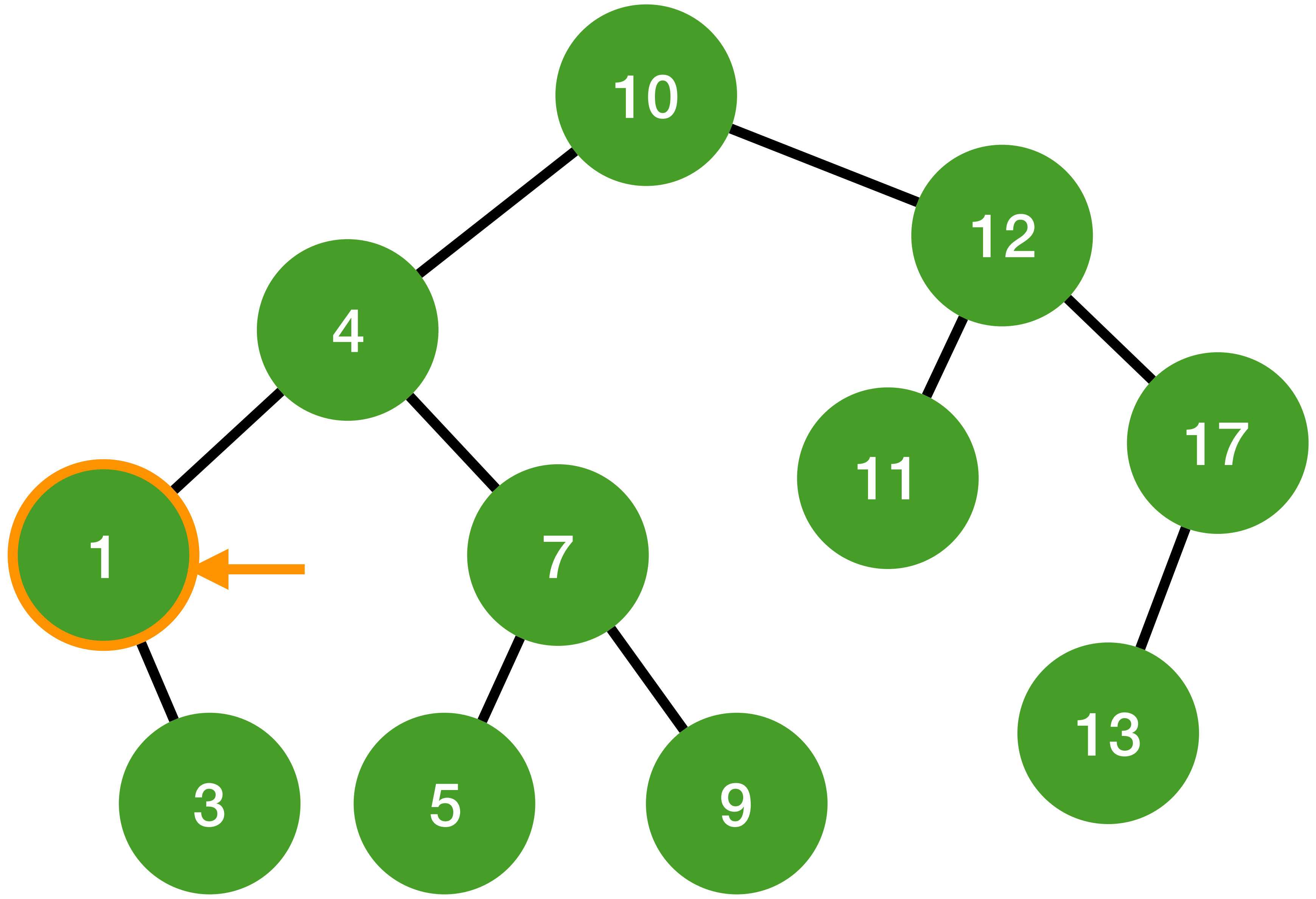
Post order =

Binary search tree traversal - Post order



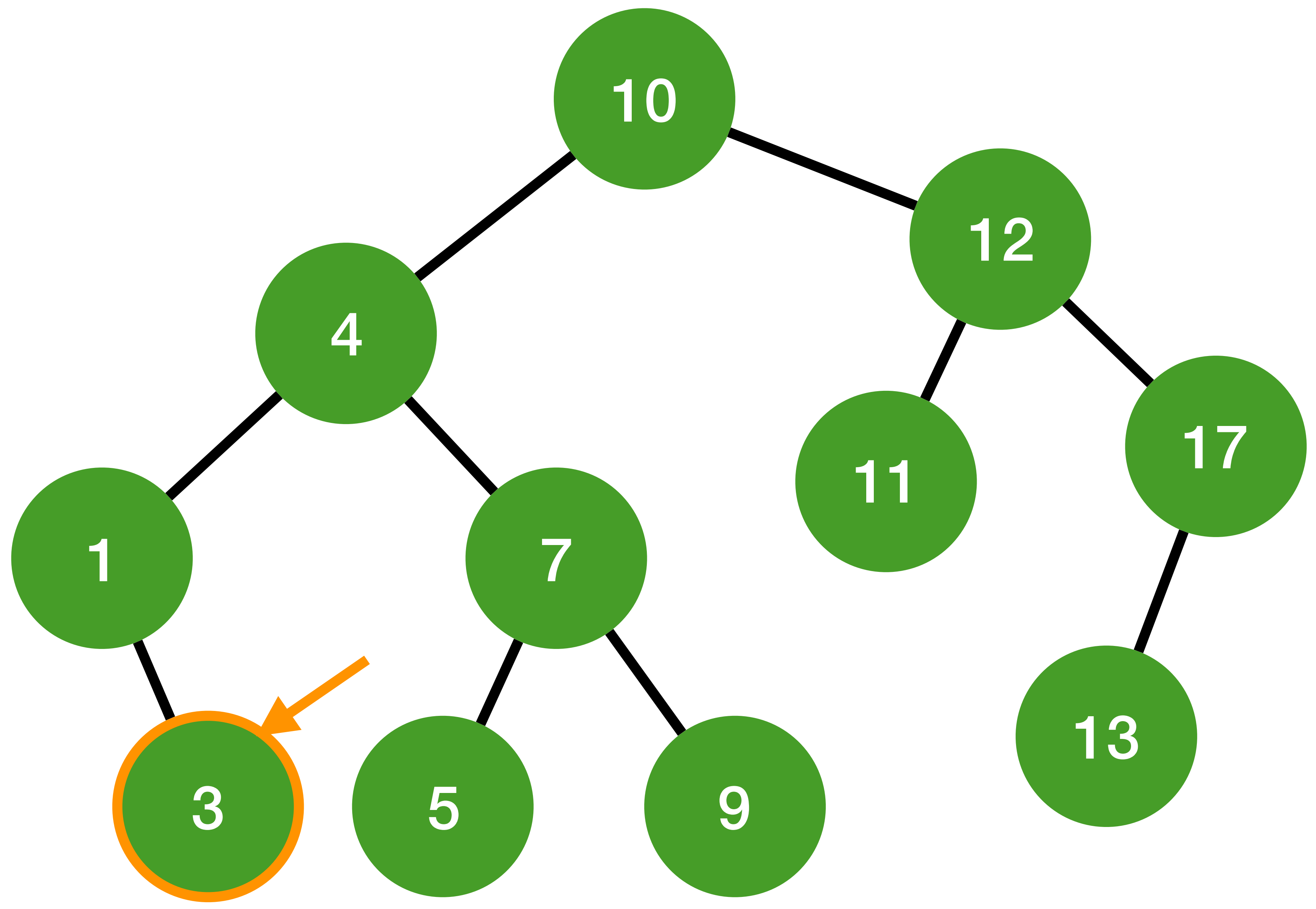
Post order =

Binary search tree traversal - Post order



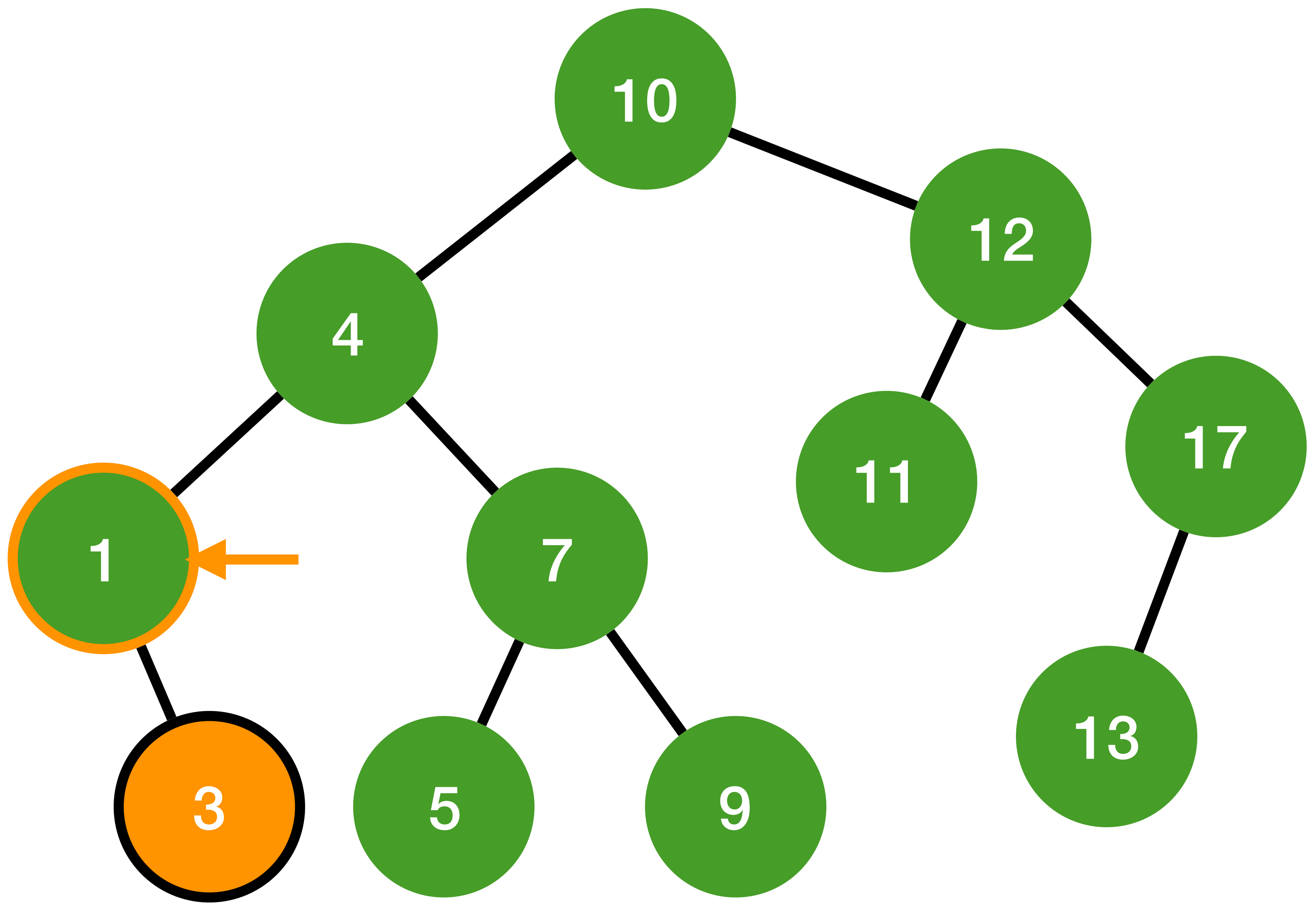
Post order =

Binary search tree traversal - Post order



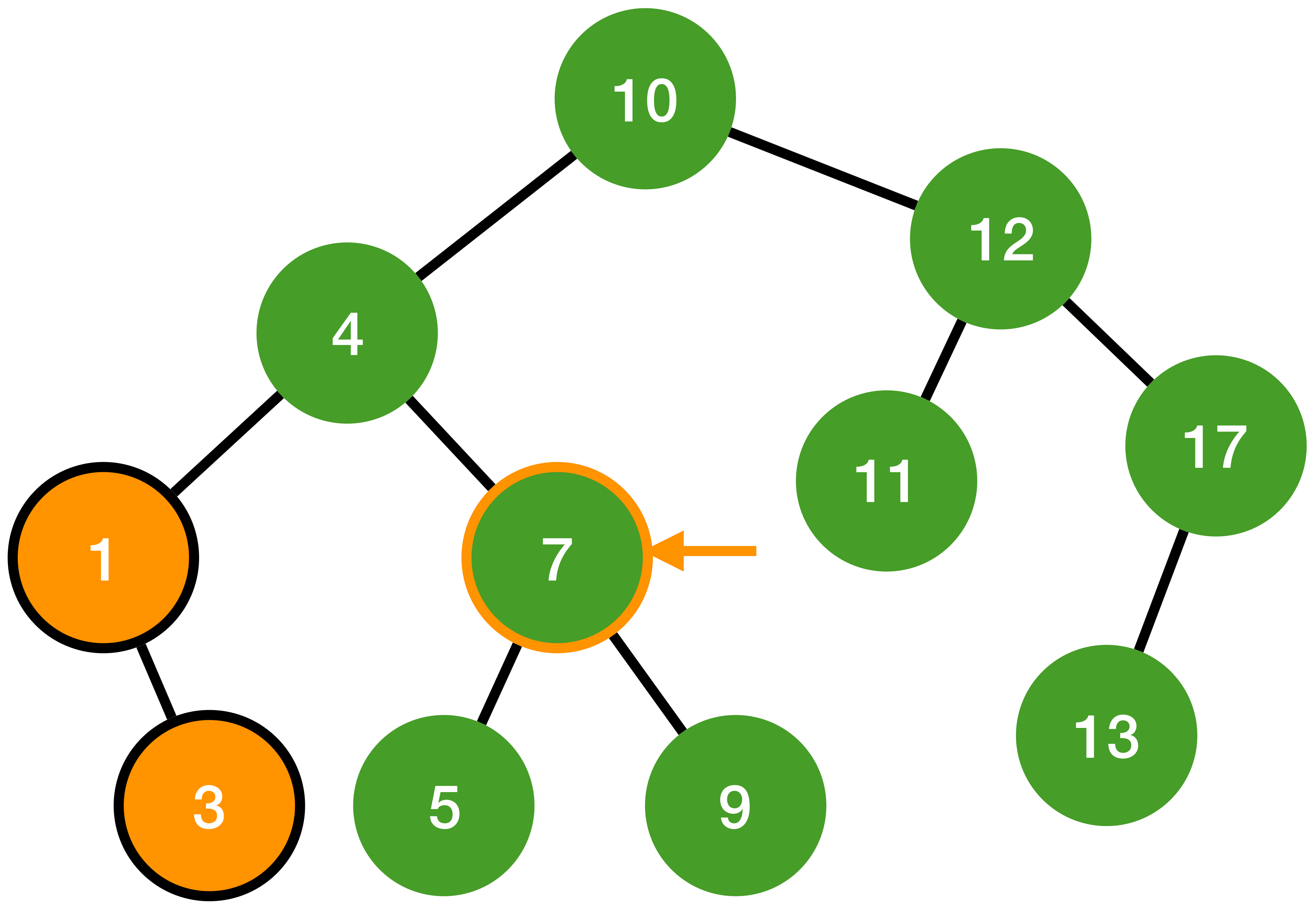
Post order =

Binary search tree traversal - Post order



Post order = 3

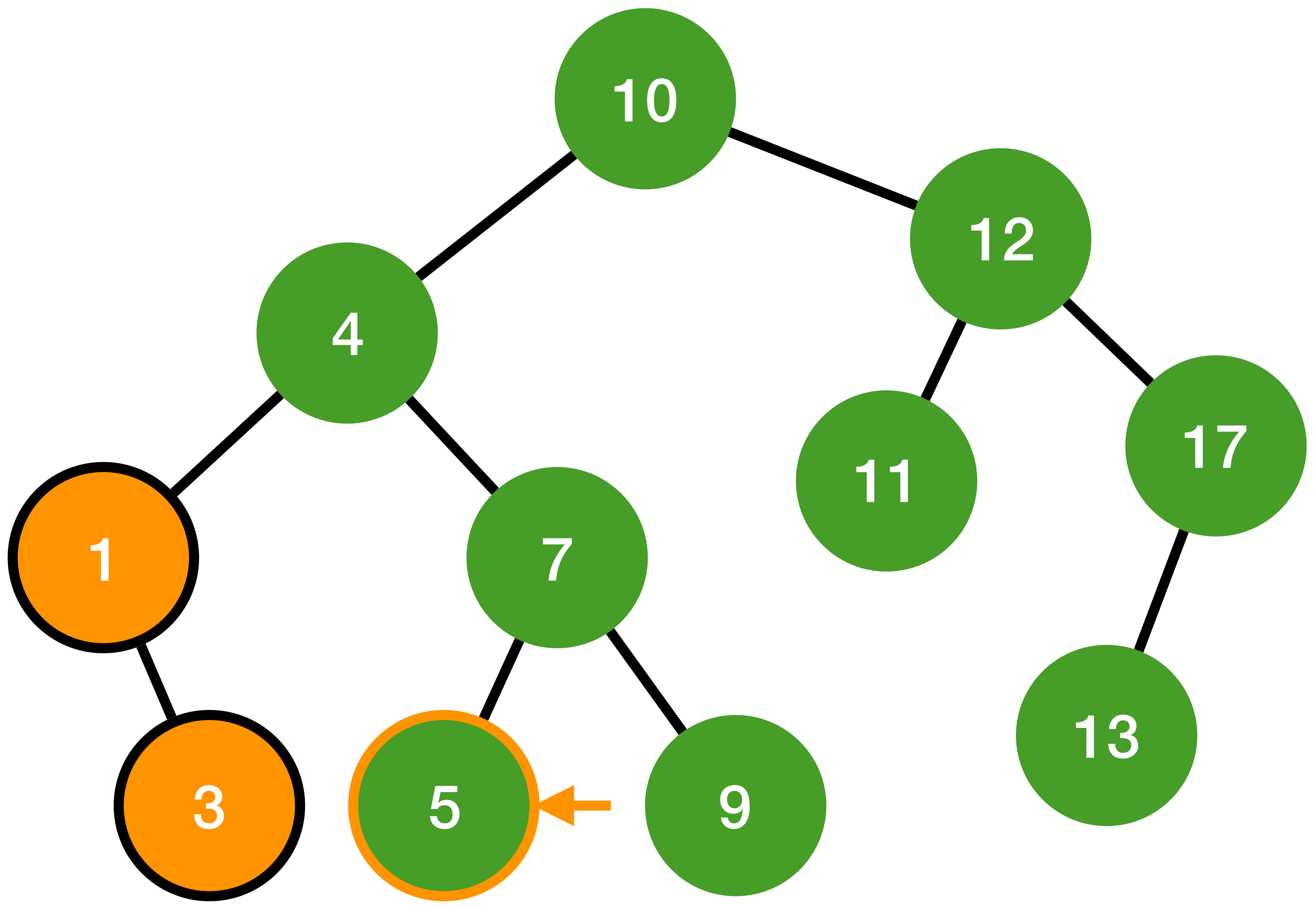
Binary search tree traversal - Post order



Post order =

3	1

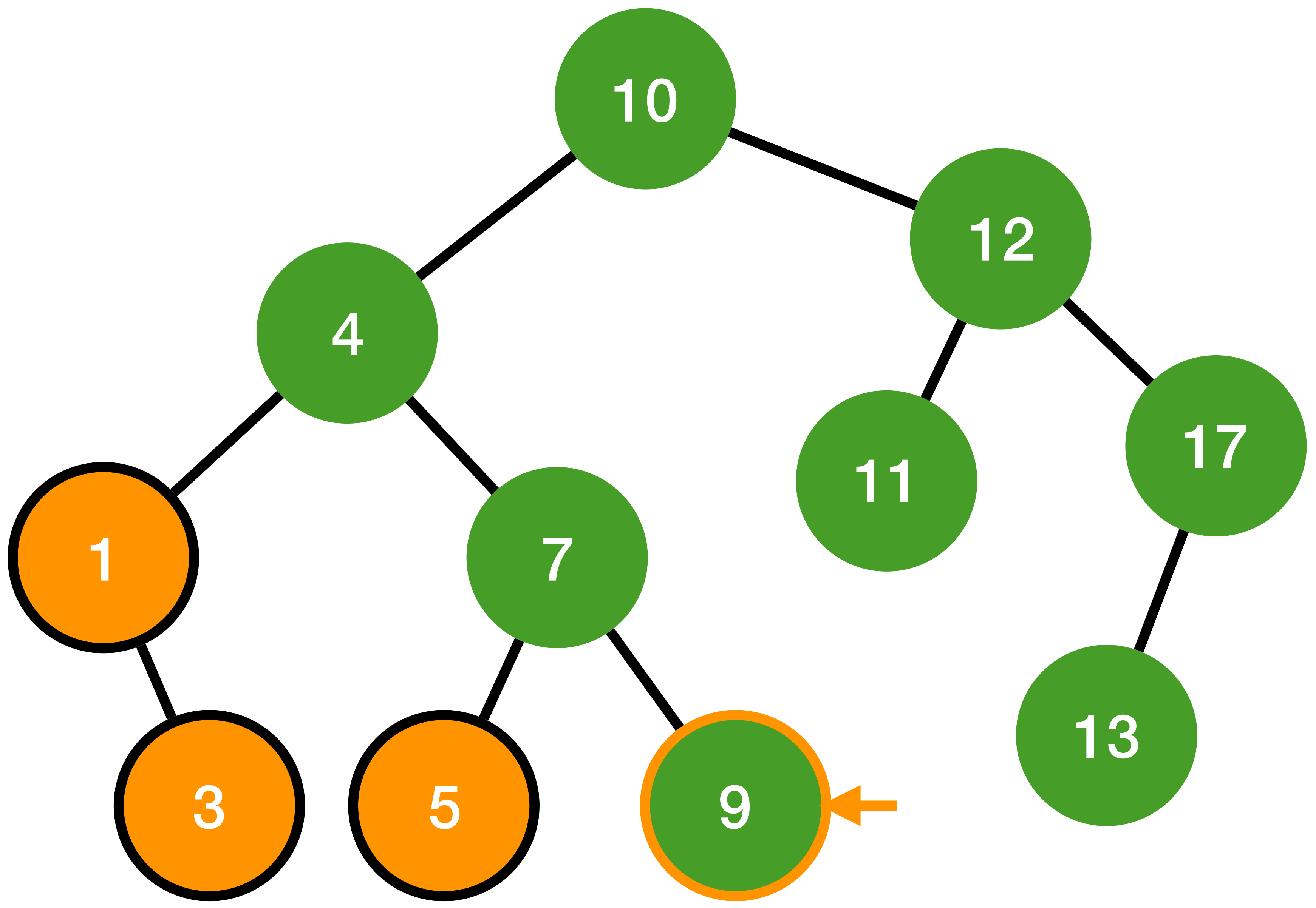
Binary search tree traversal - Post order



Post order =

3	1

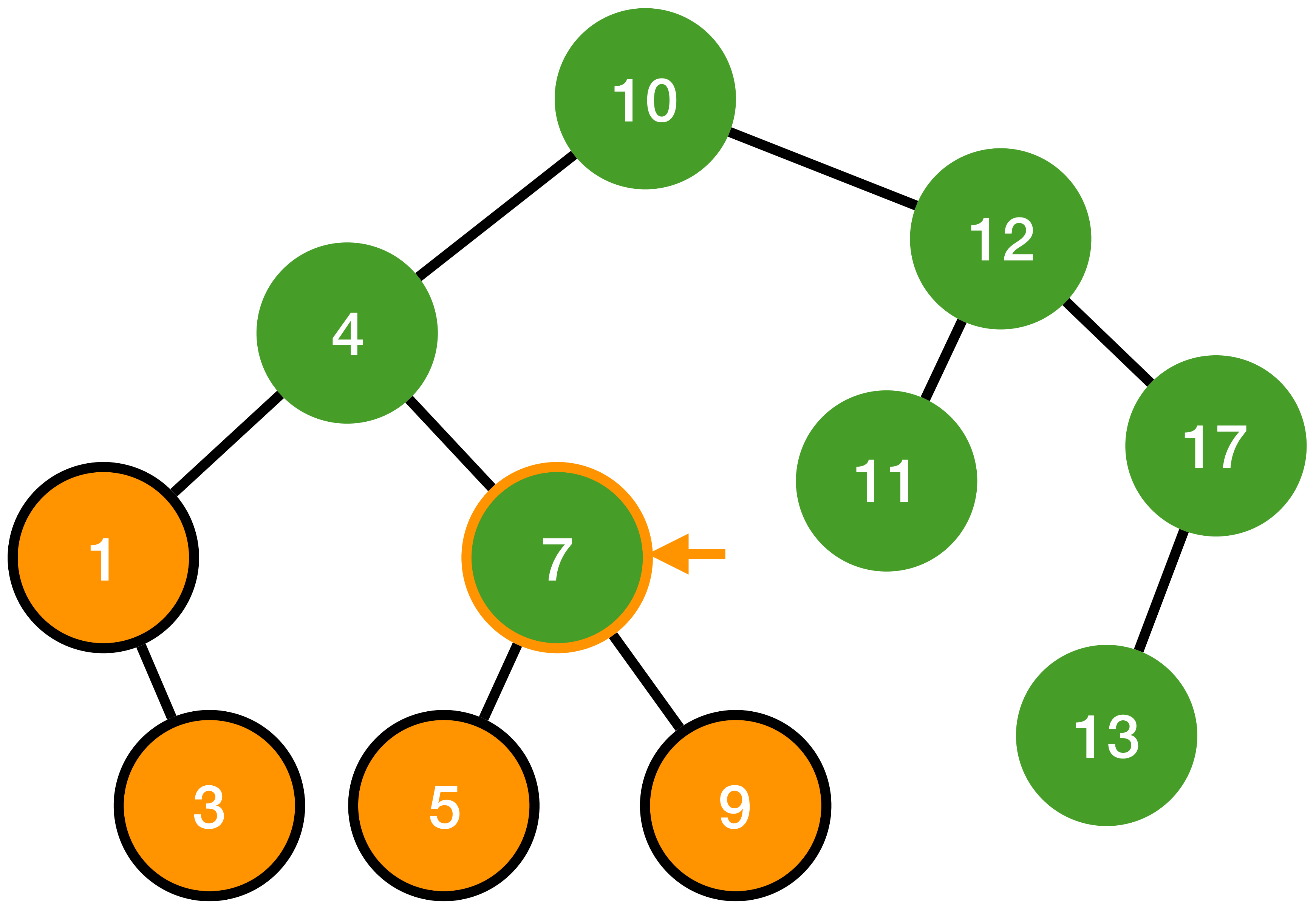
Binary search tree traversal - Post order



3	1	5

Post order =

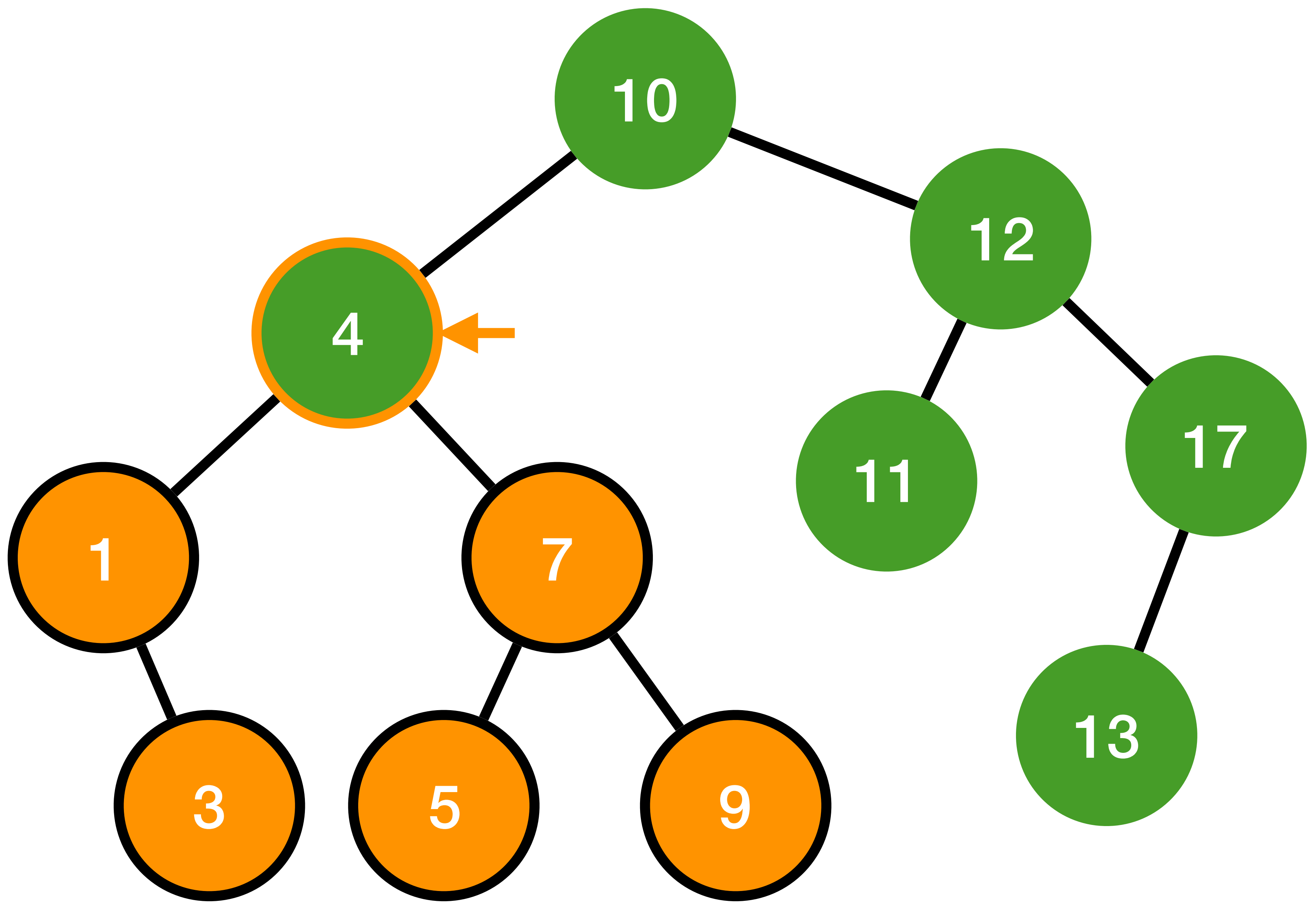
Binary search tree traversal - Post order



Post order =

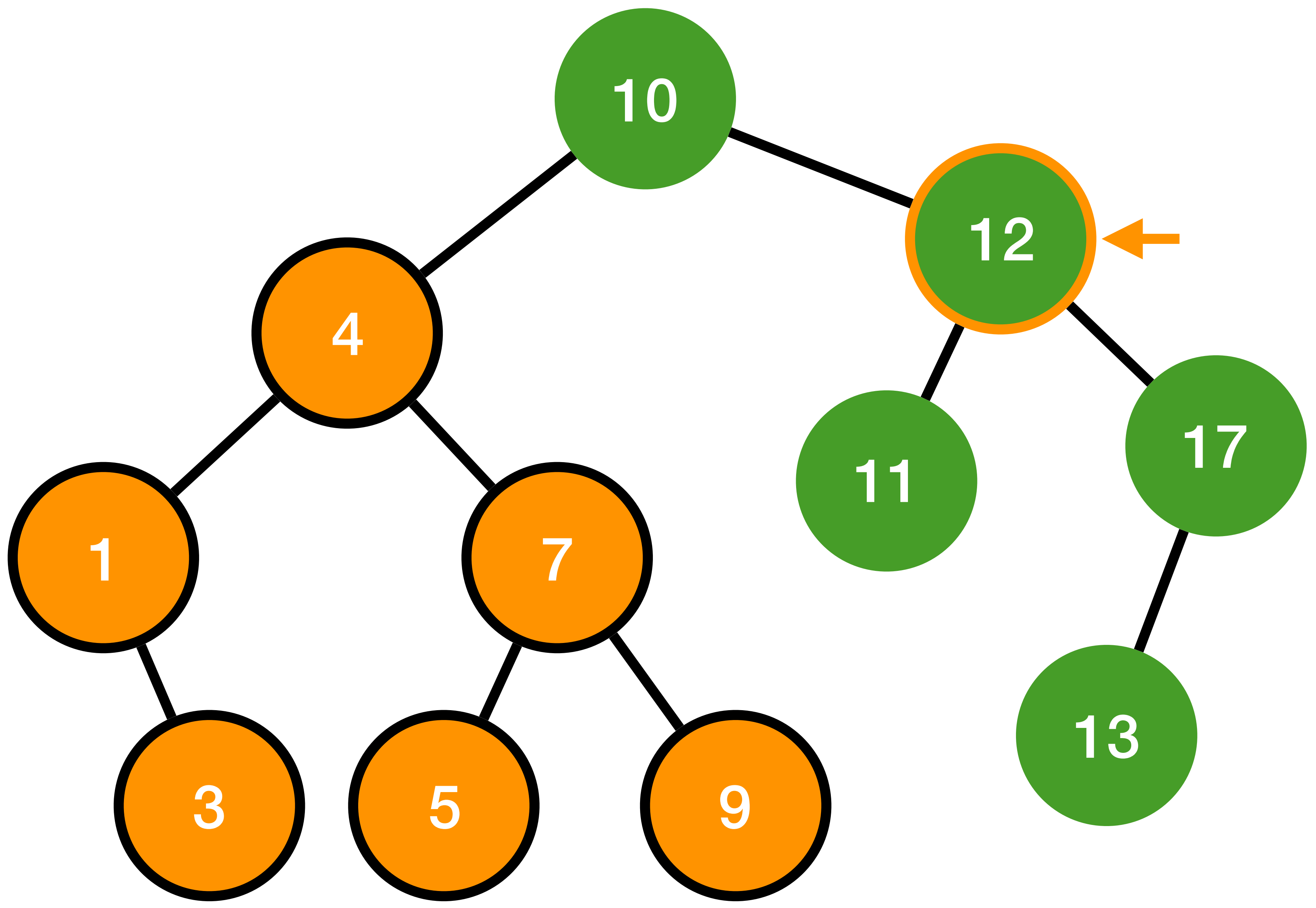
3	1	5	9

Binary search tree traversal - Post order



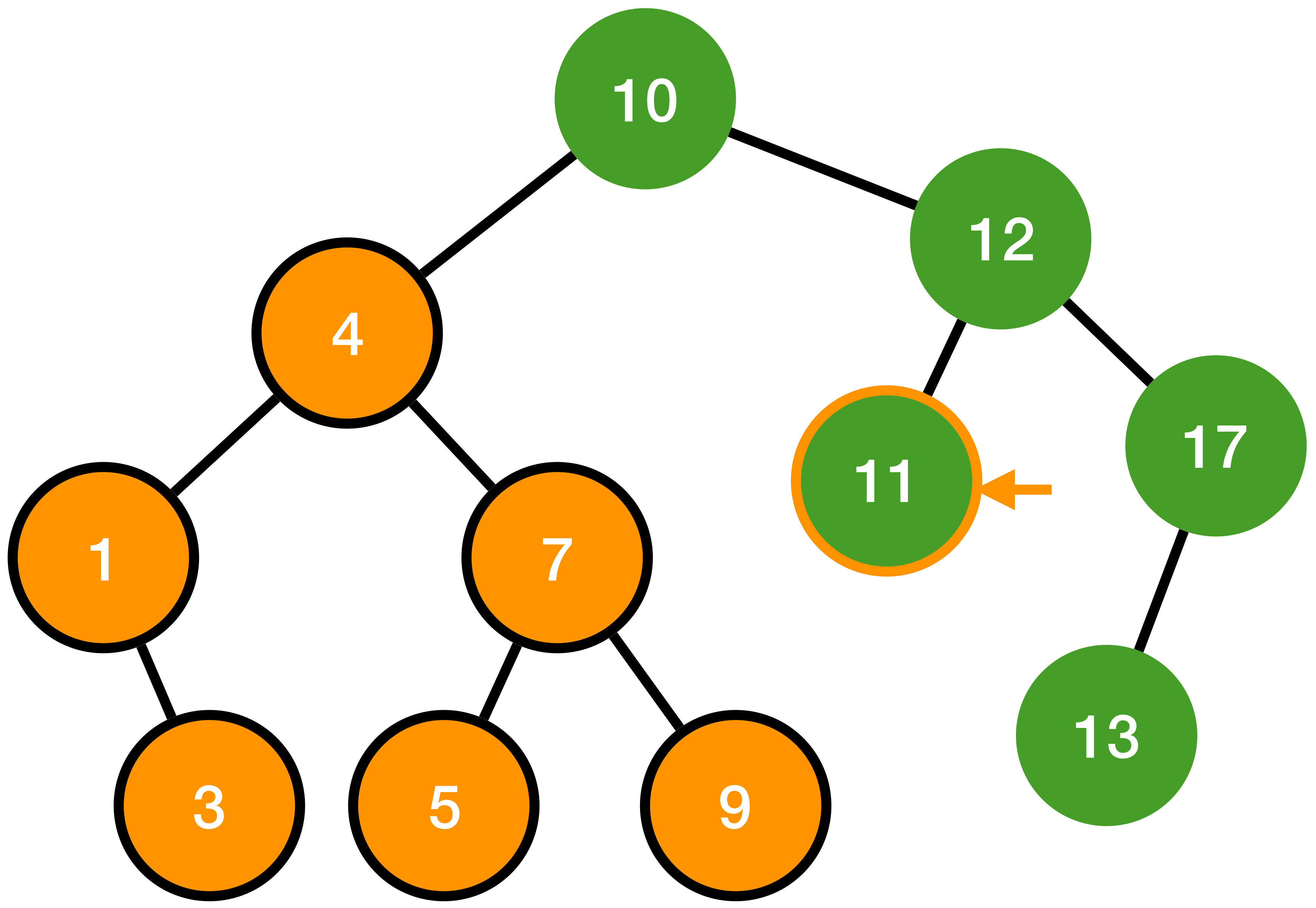
Post order =	3	1	5	9	7

Binary search tree traversal - Post order



Post order =					
3	1	5	9	7	4

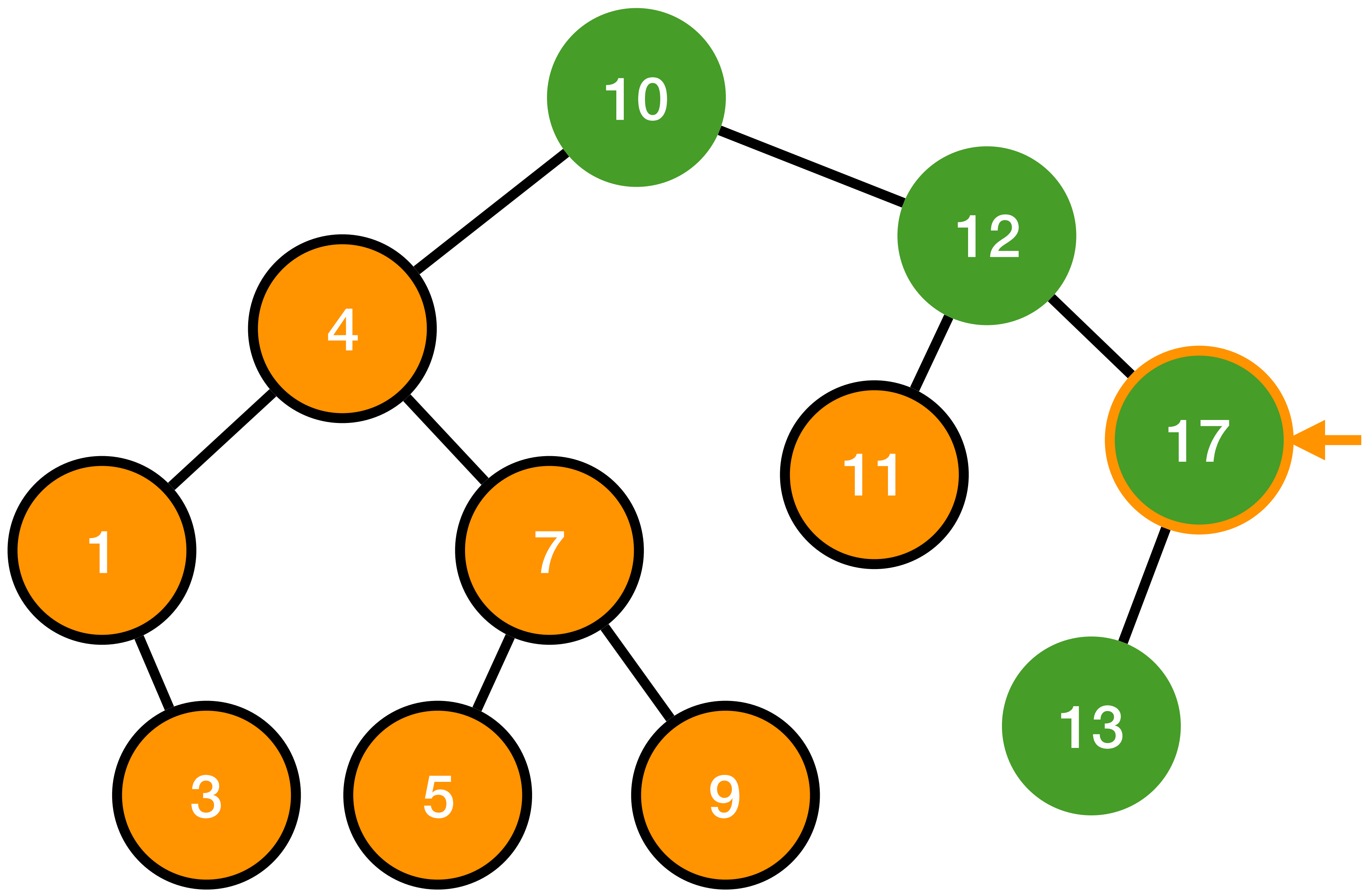
Binary search tree traversal - Post order



Post order =

3	1	5	9	7	4

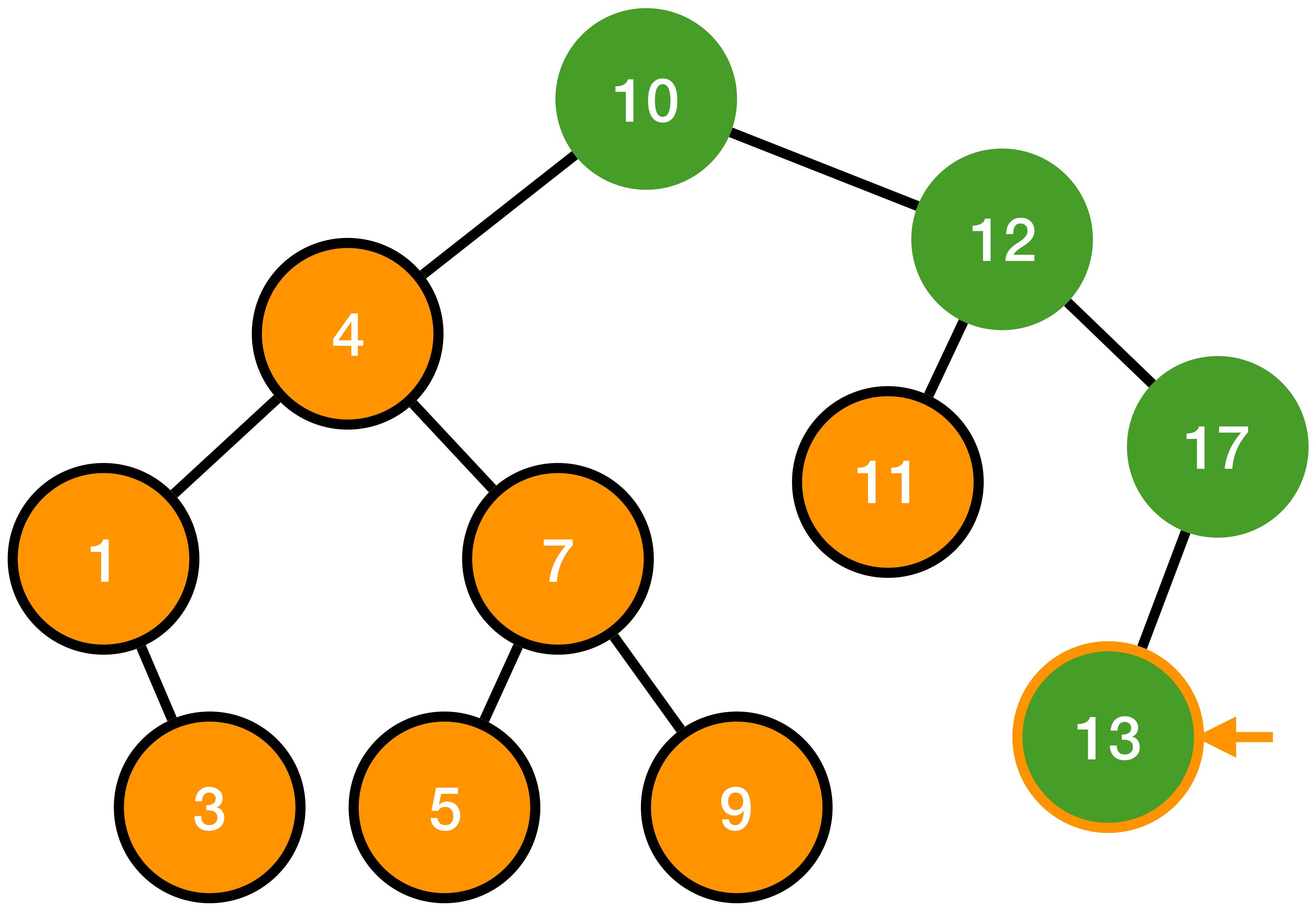
Binary search tree traversal - Post order



Post order =

3	1	5	9	7	4	11

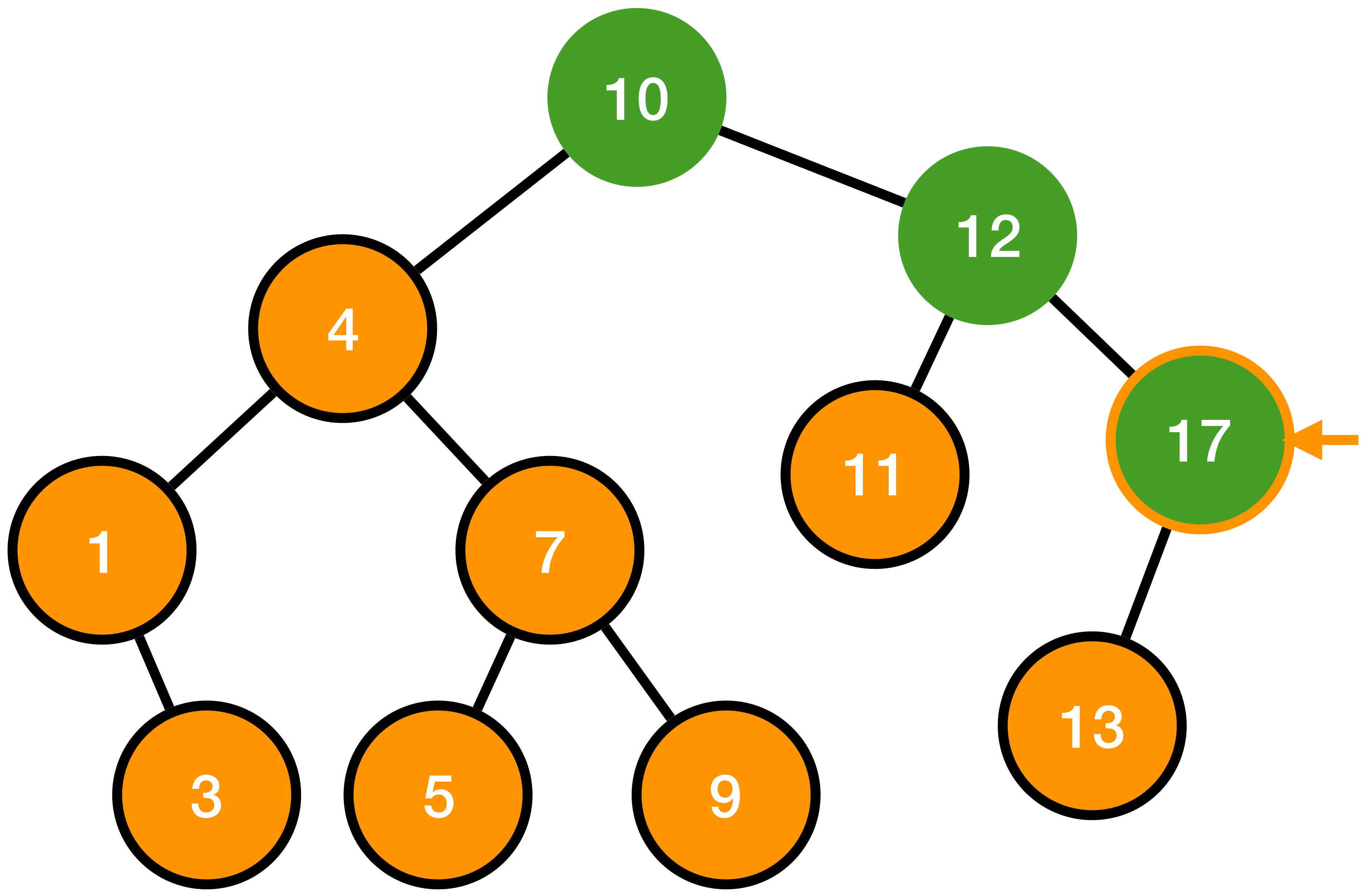
Binary search tree traversal - Post order



Post order =

3	1	5	9	7	4	11

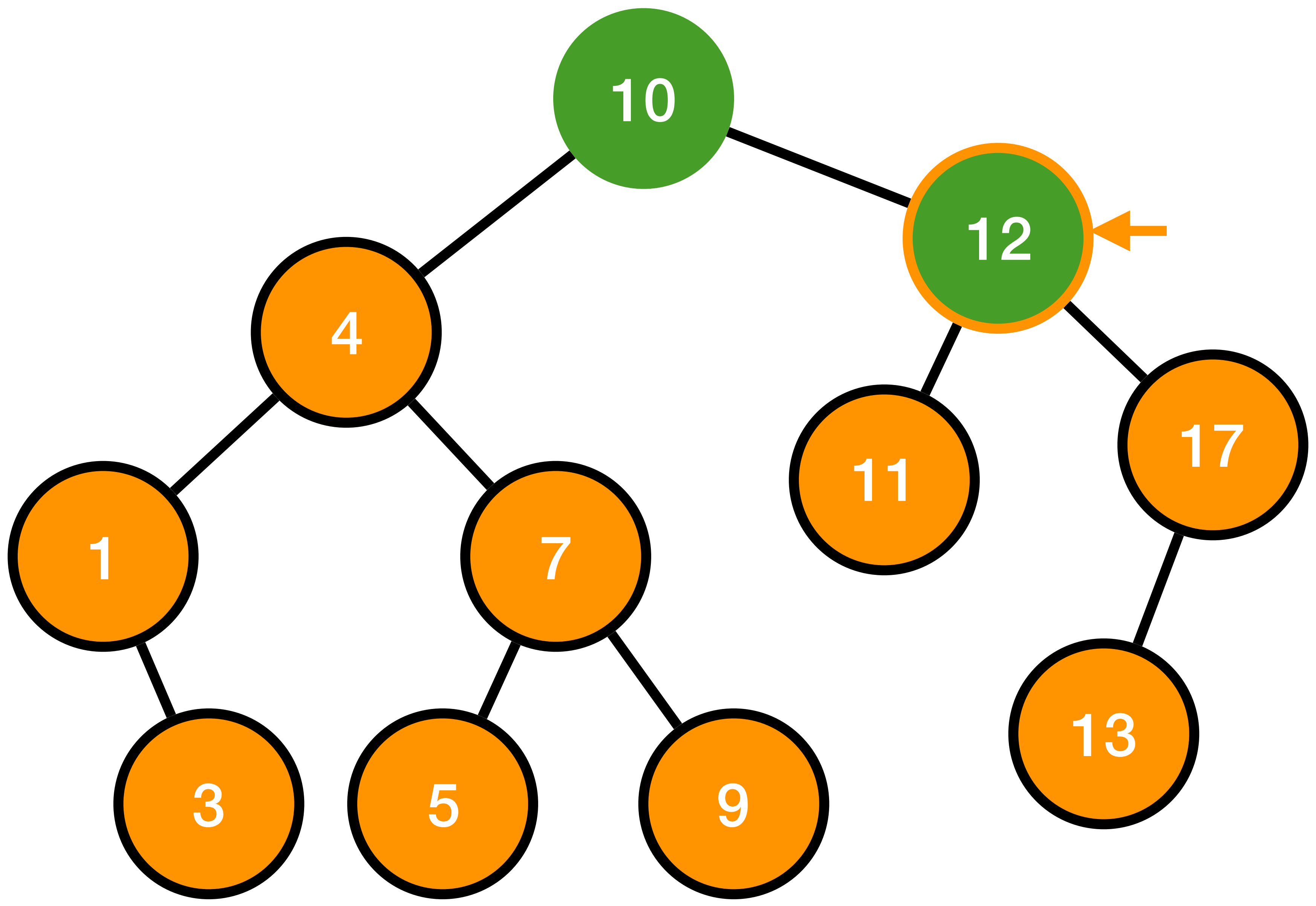
Binary search tree traversal - Post order



Post order =

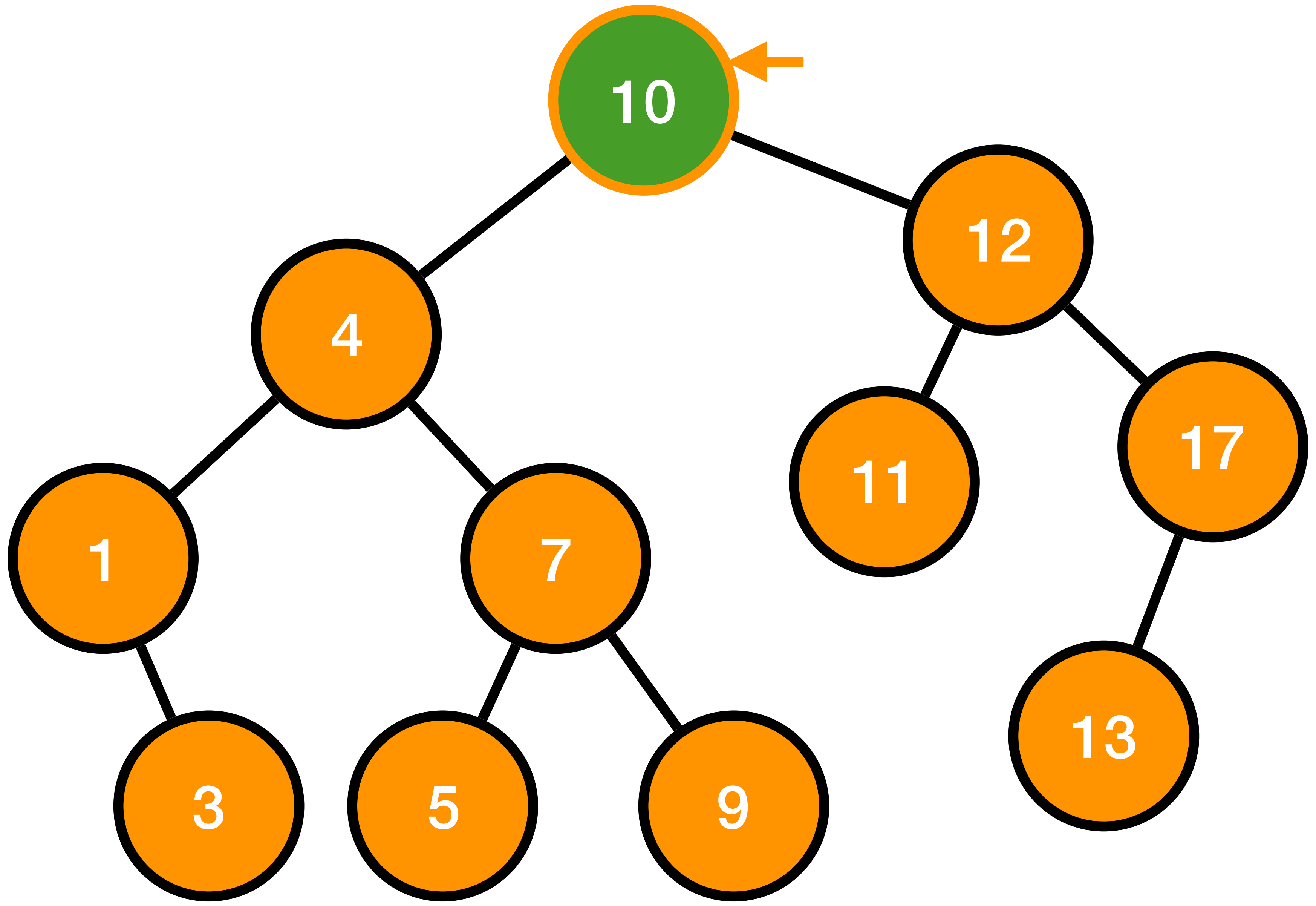
3	1	5	9	7	4	11	13

Binary search tree traversal - Post order



Post order =	3	1	5	9	7	4	11	13	17

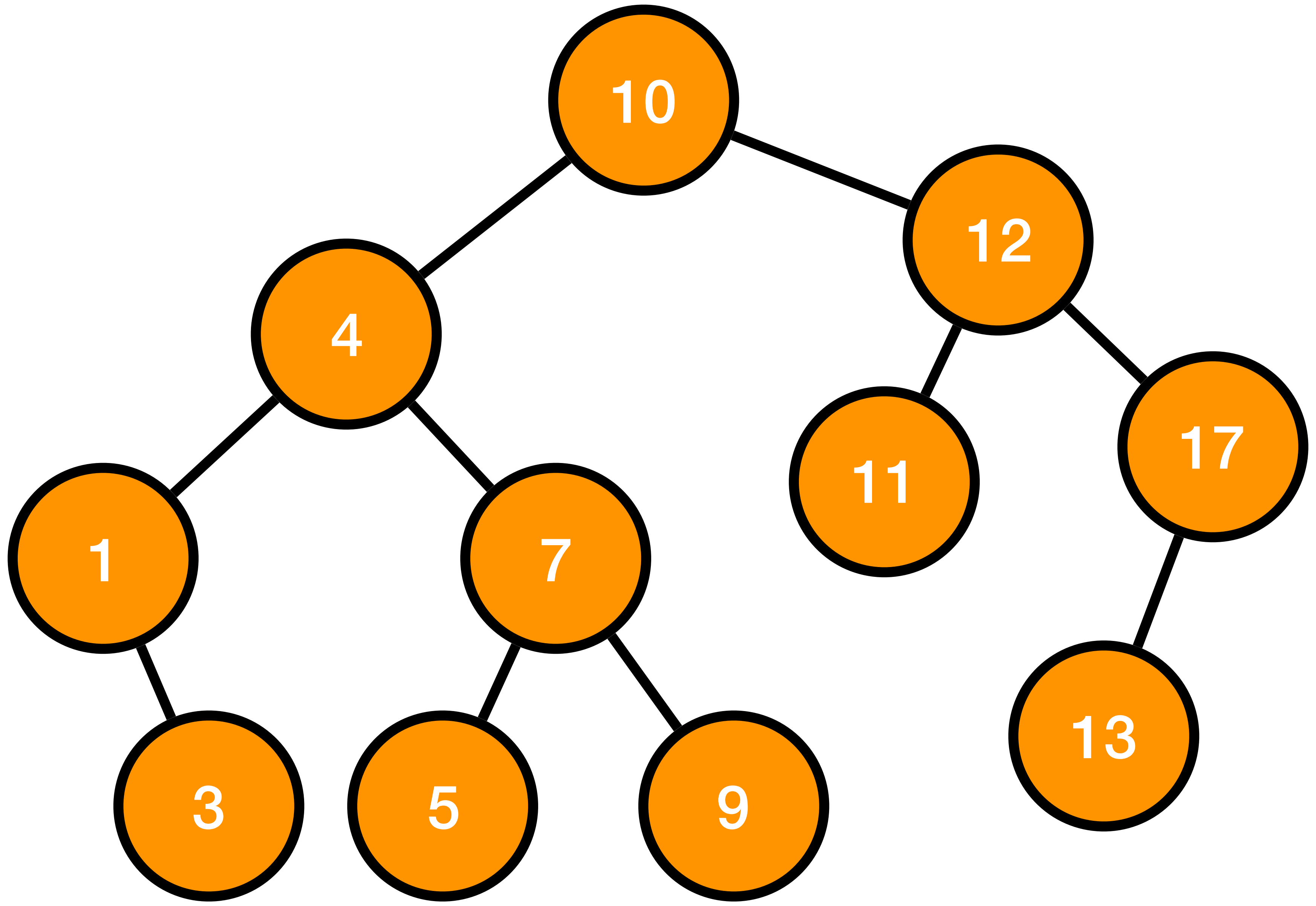
Binary search tree traversal - Post order



Post order =

3	1	5	9	7	4	11	13	17	12

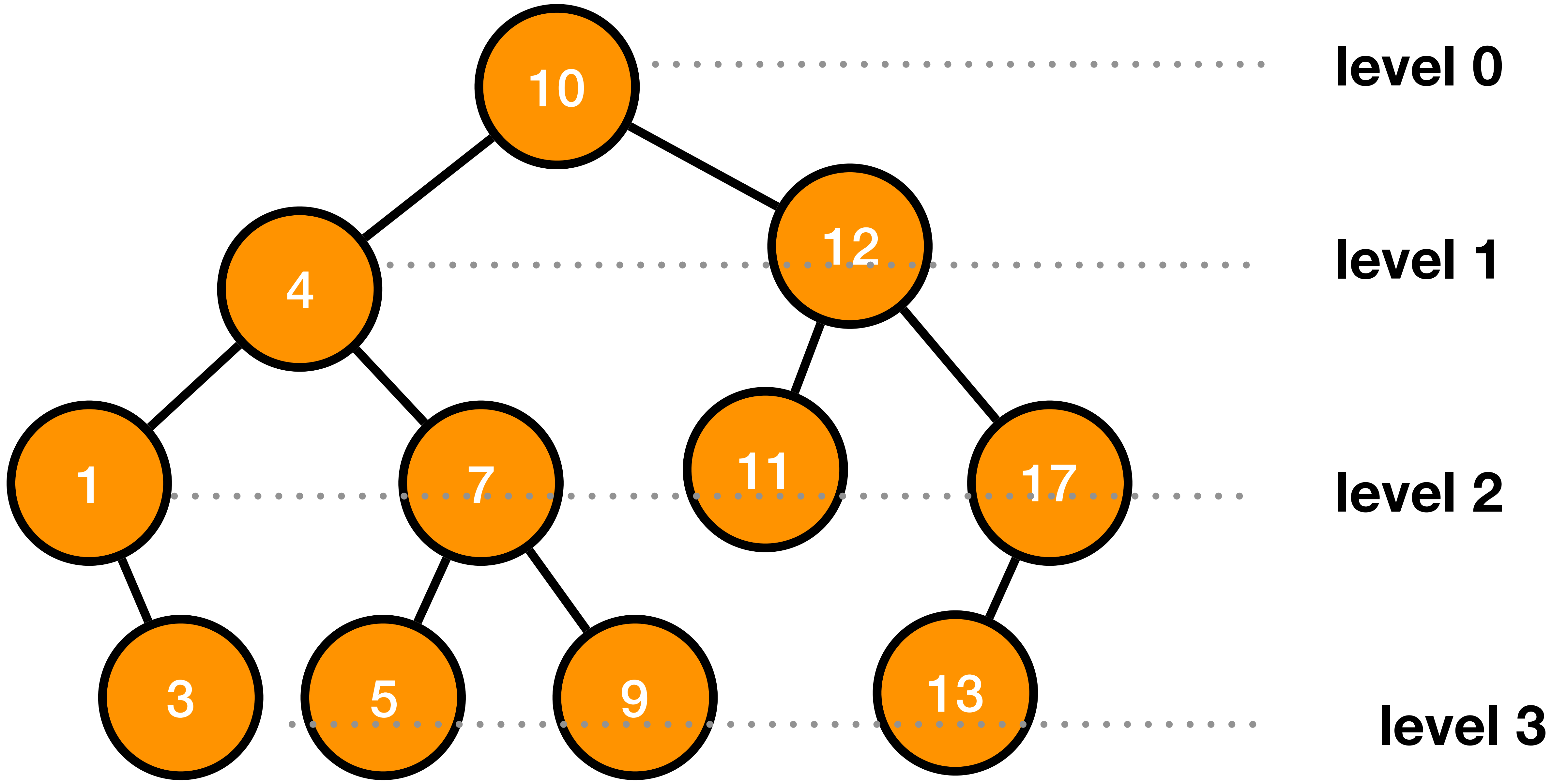
Binary search tree traversal - Post order



Post order =	3	1	5	9	7	4	11	13	17	12	10

Binary search tree - level-order

Print nodes at level *i* before the nodes at level *i*+1



Level-order =

10	4	12	1	7	11	17	3	5	9	13

A queue is used to implement level-order traversal.

in order traversal C code

```c
// inorder traversal using recursion
void inOrder(TreeNode *bst) {

  if (bst != NULL) {
    inOrder(bst->left);
    printf("%d \n", bst->data.value);
    inOrder(bst->right);
  }
}
```

```c
// inorder traversal without recursion
void inOrder_non_recursive(TreeNode *tree) {
  Stack *stack = createStack(10);
  push(stack, tree);
  while (!isEmpty(stack)) {
    while(tree->left != NULL) {
      tree = tree->left;
      push(stack, tree);
    }

    tree = peek(stack);
    pop(stack);
    printf("Popped stack %d \n", tree->data.value);

    if (tree->right != NULL) {
      tree = tree->right;
      push(stack, tree);
    }

  }
}
```

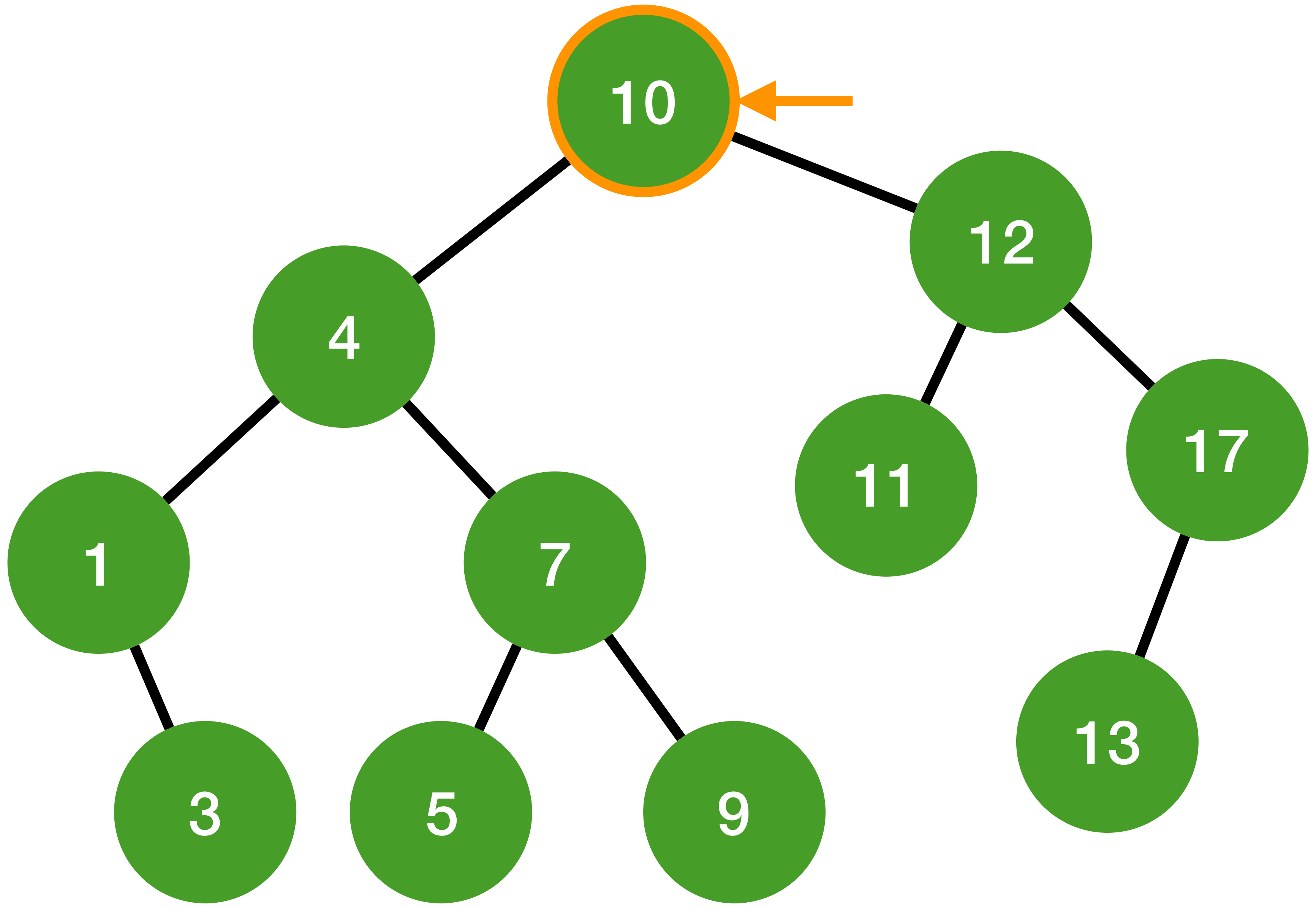

Binary search tree traversal : pre-order algorithm

```
preOrder(x){
  if (x != NULL) {
    print x.key;
    preOrder(x.left);
    preOrder(x.right);
  }
}
```

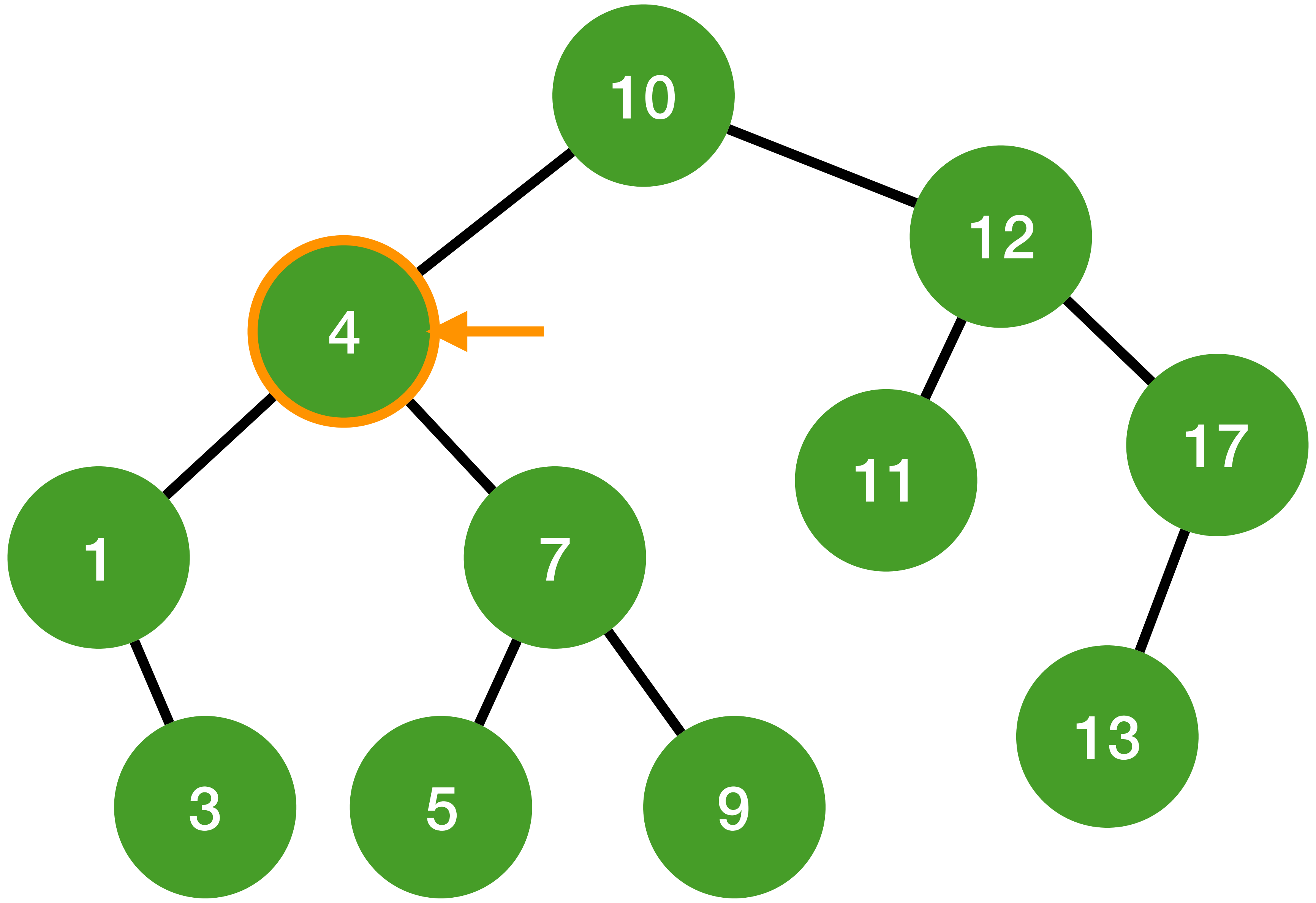

Binary search tree traversal : post order algorithm

```
postOrder(x){
  if (x != NULL) {
    postOrder(x.left);
    postOrder(x.right);
    print x.key;
  }
}
```

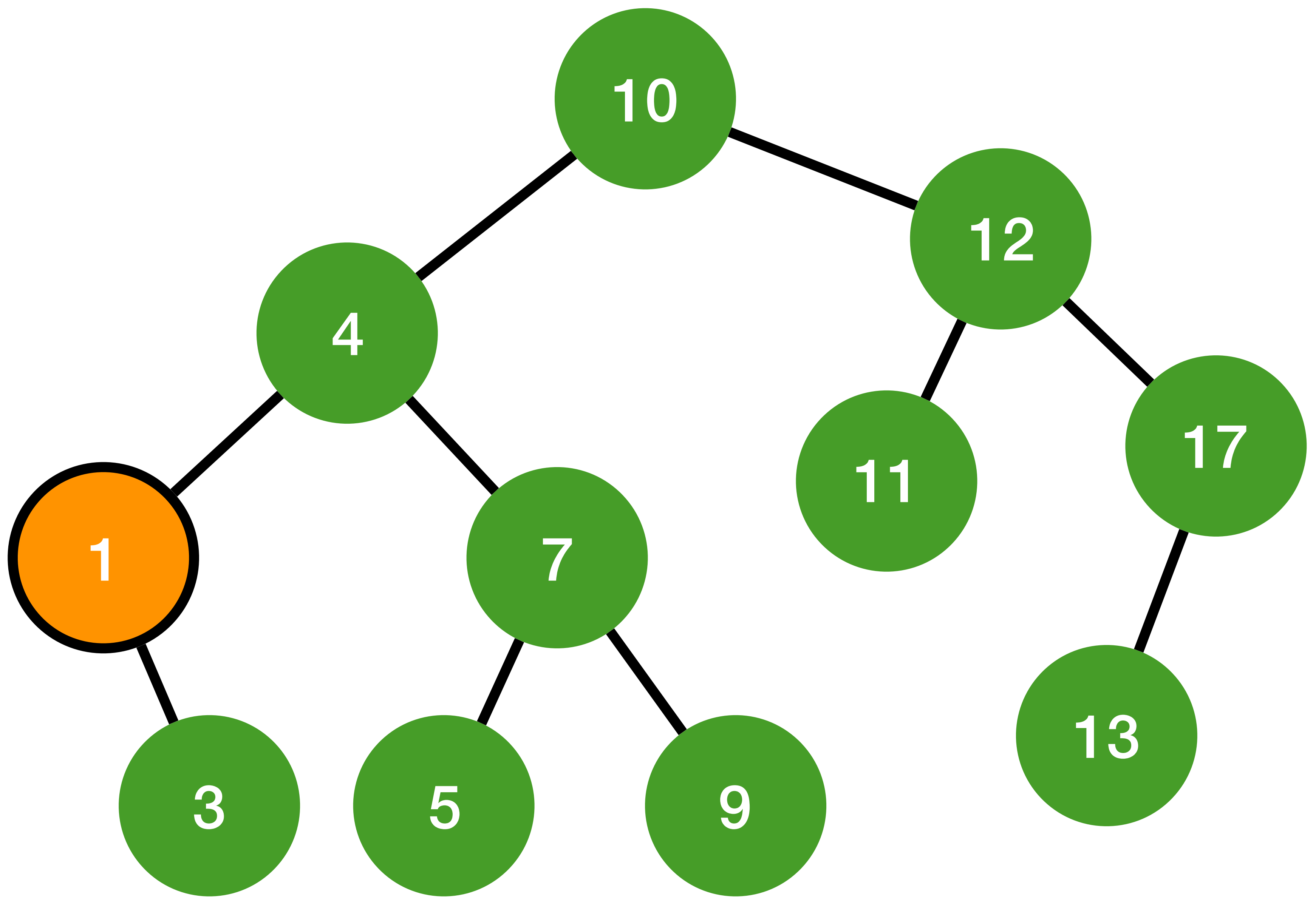

Binary search tree - minimum value



Binary search tree - minimum value



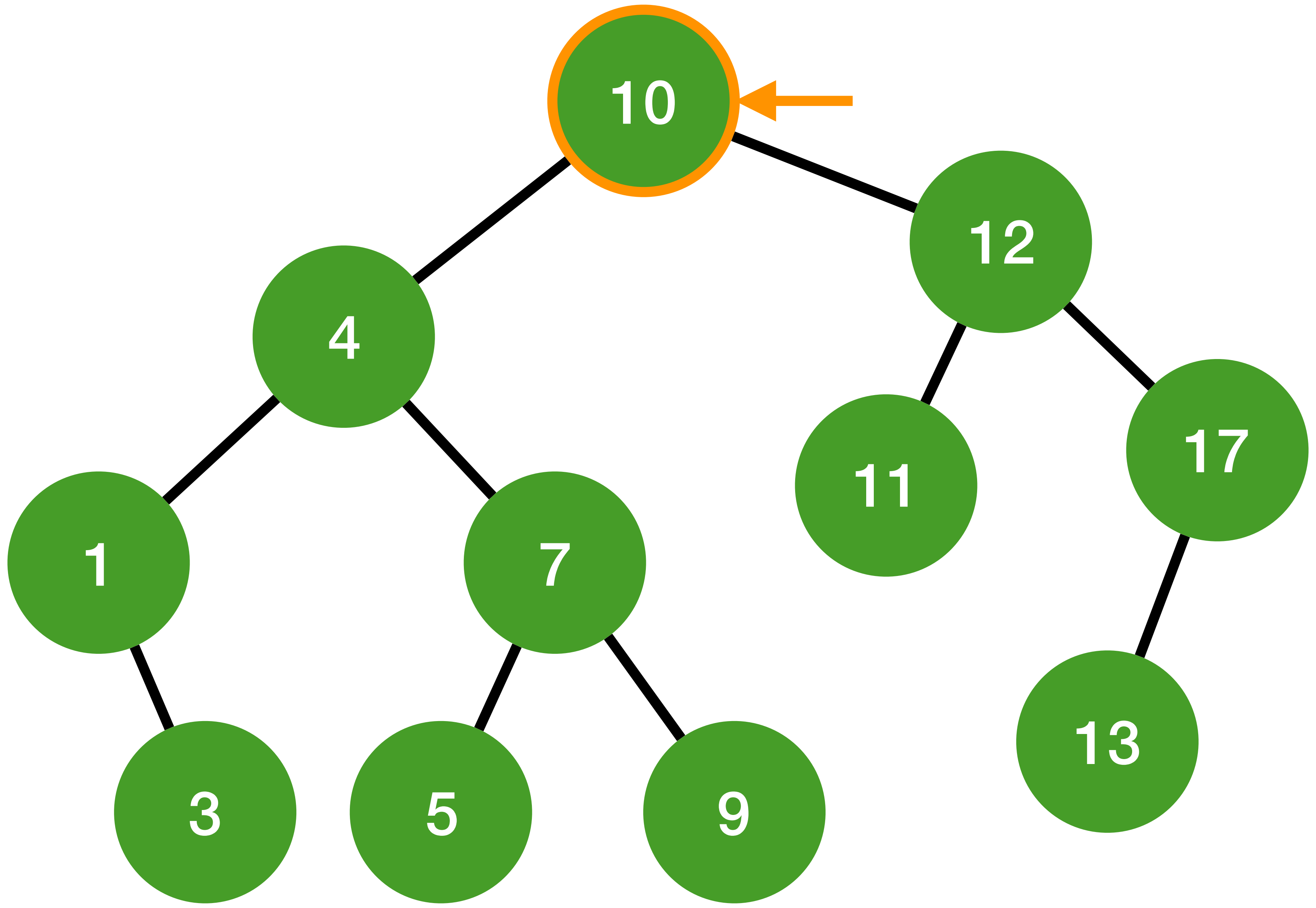
Binary search tree - minimum value



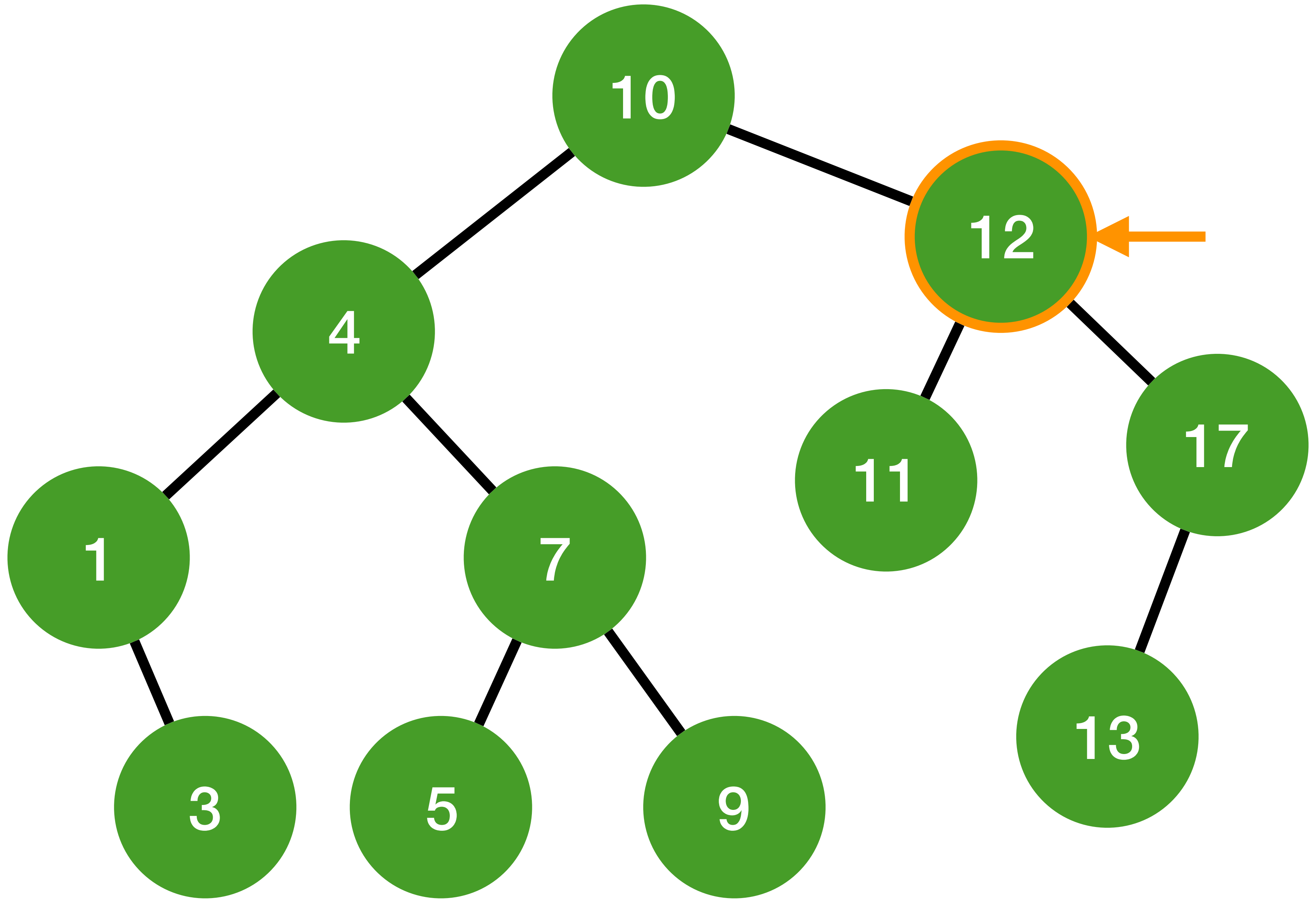
Binary search tree - minimum value algorithm

```
BinaryTreeMin(x){
  while (x.left != NIL)
    x = x.left;
  return x;
}
```

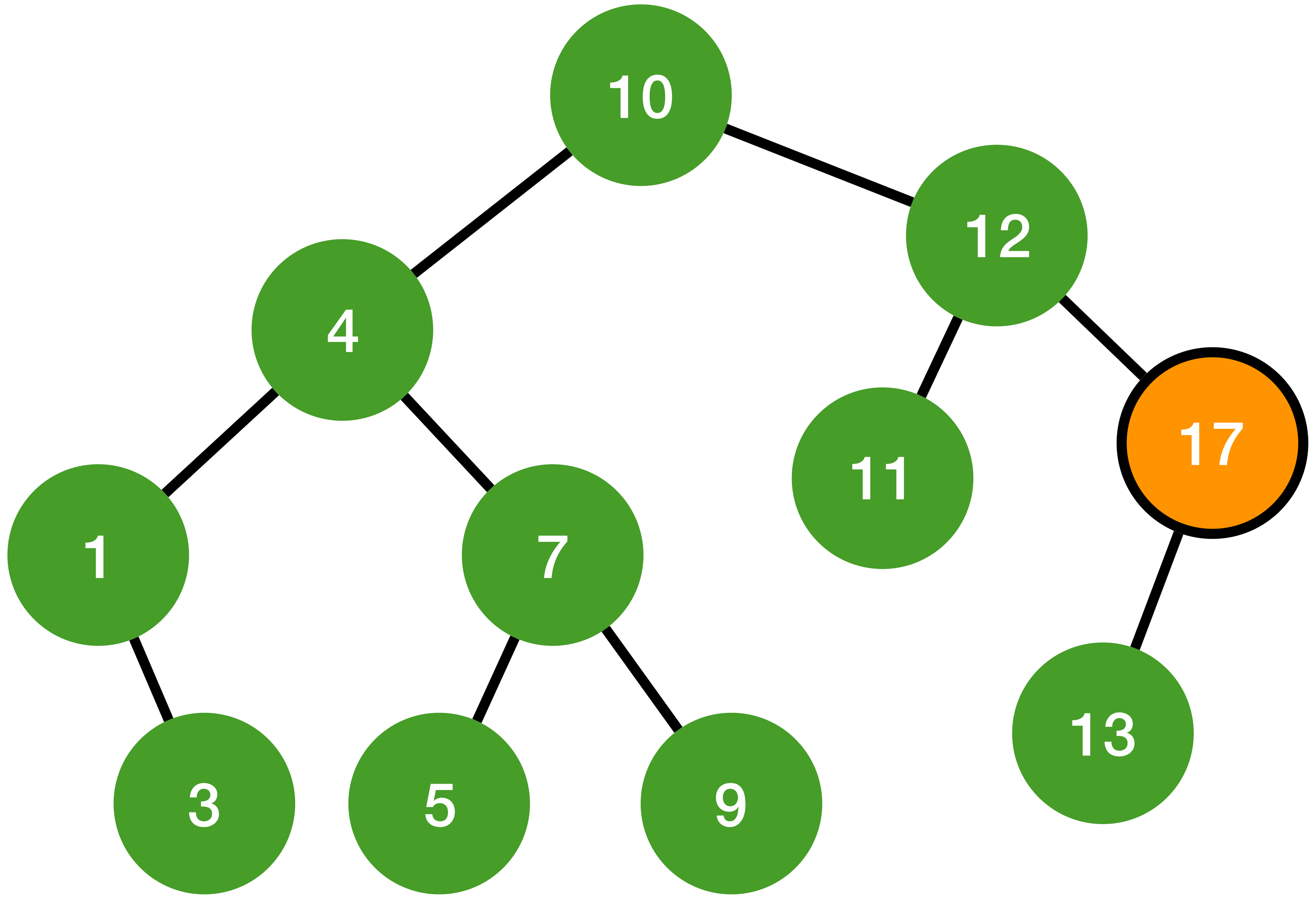

Binary search tree - maximum value



Binary search tree - maximum value



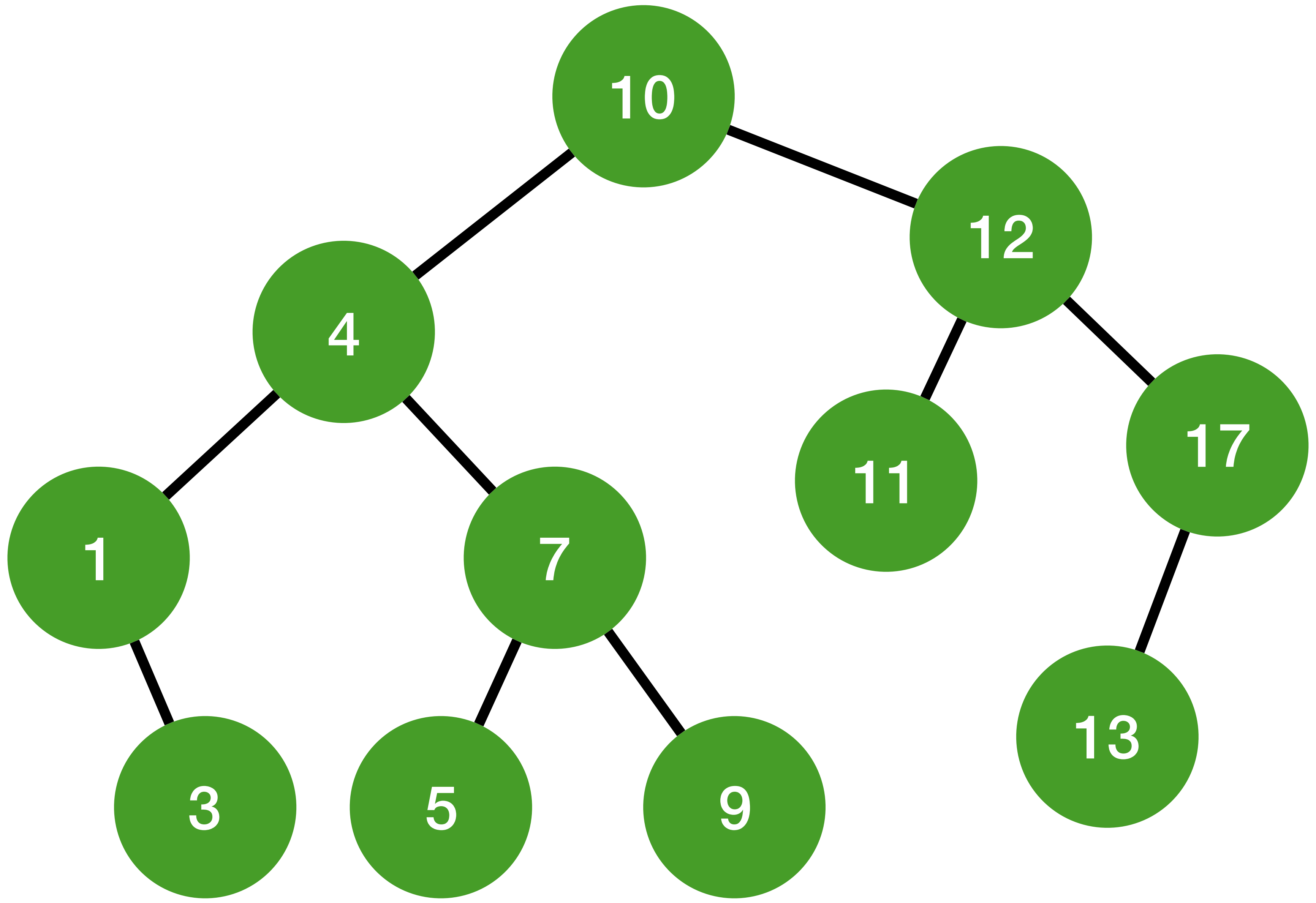
Binary search tree - maximum value



Binary search tree - minimum value algorithm

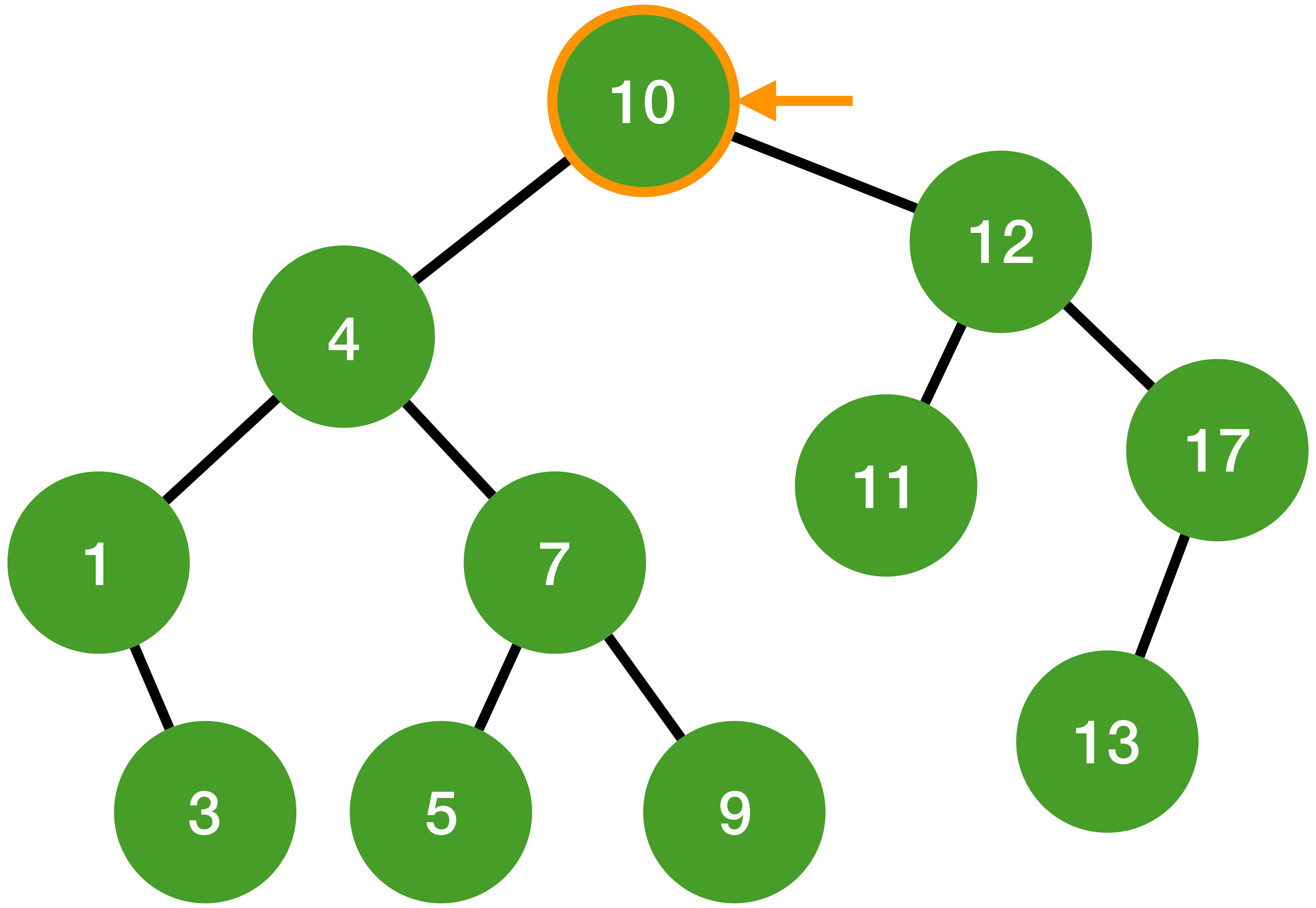
```
BinaryTreeMax(x){
  while (x.right != NULL)
    x = x.right;

  return x;
}
```


Binary search tree - insertion



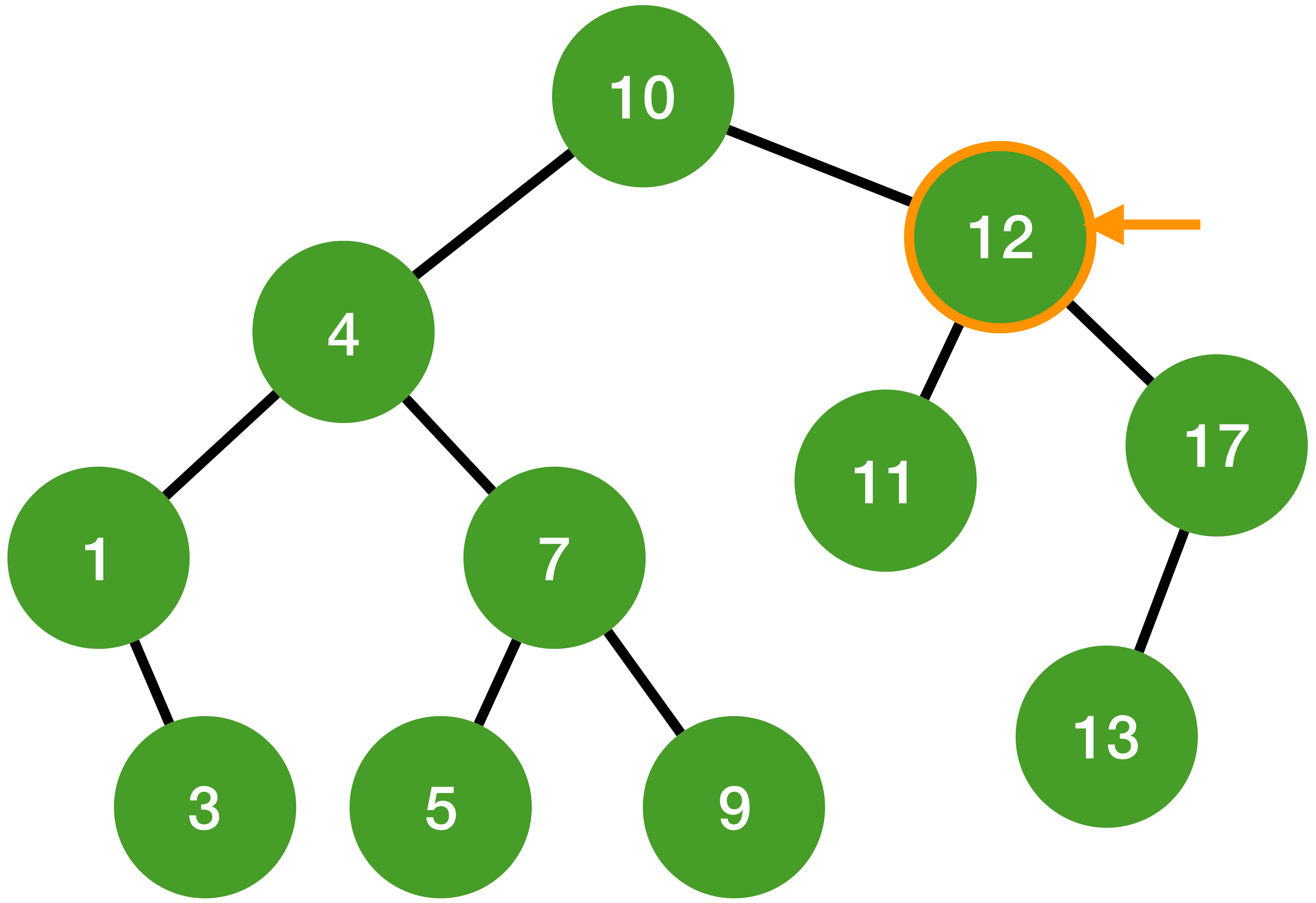
insert item with key 15

Binary search tree - insertion



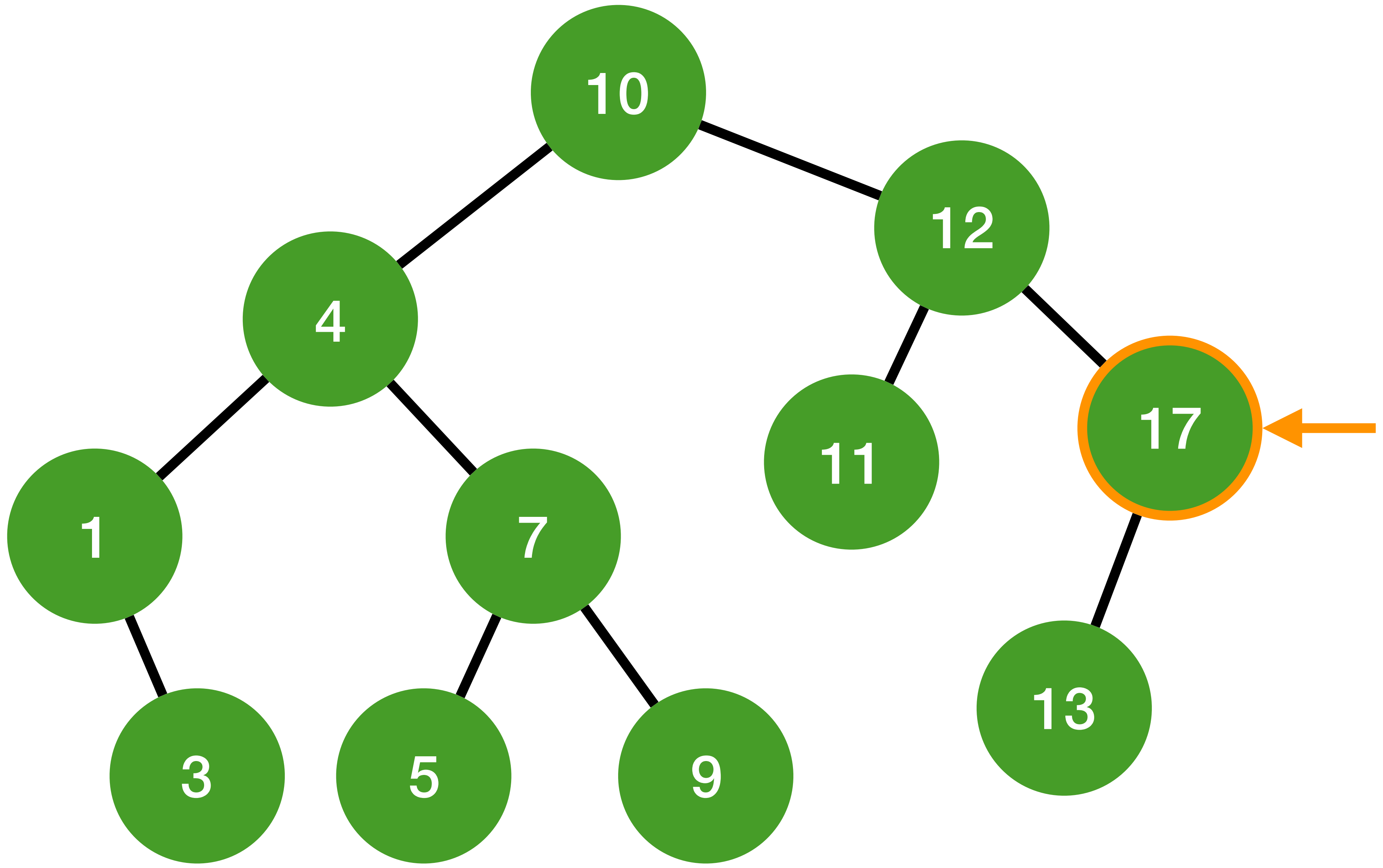
15 > 10 => right

Binary search tree - insertion



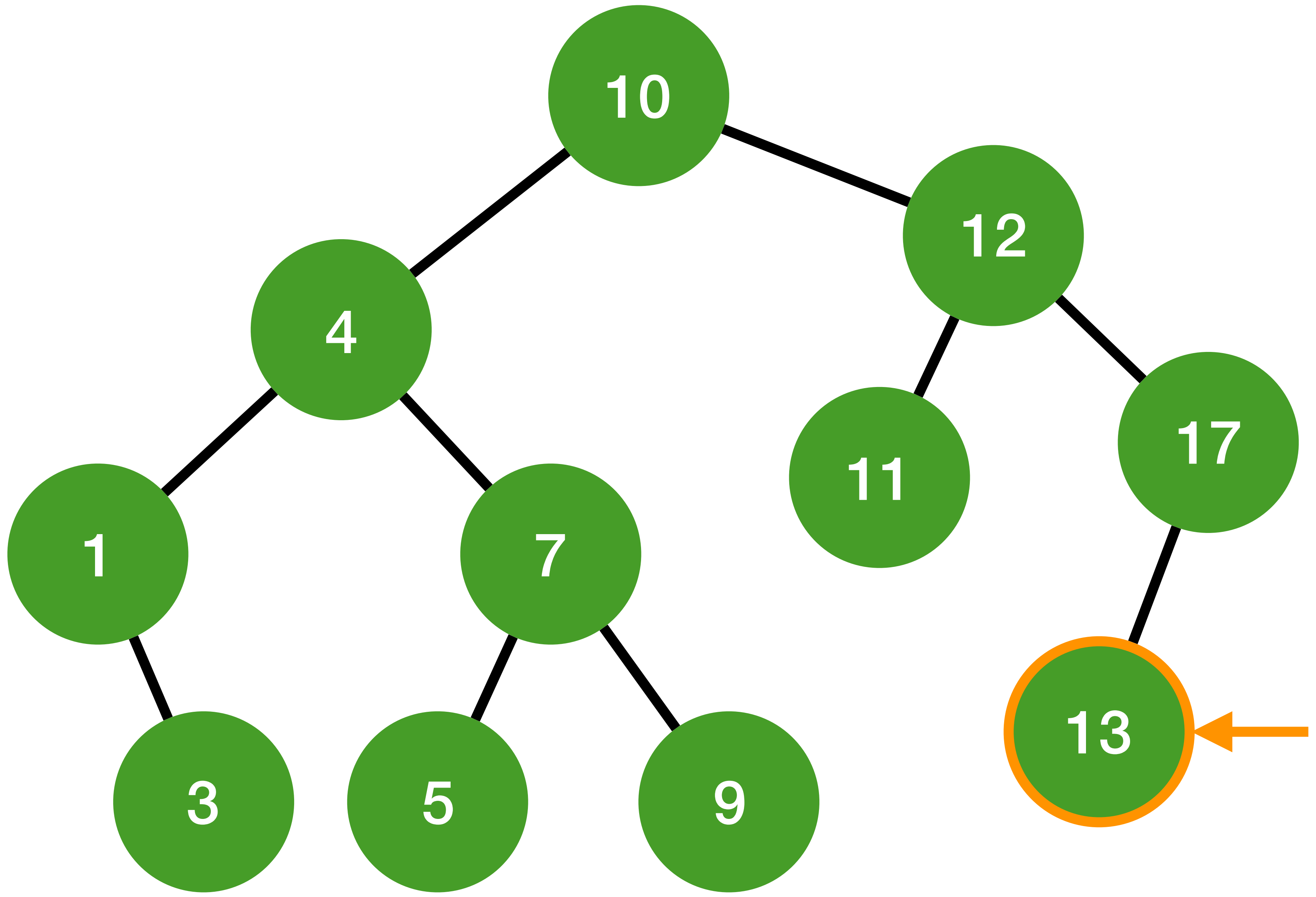
15 > 12 => right

Binary search tree - insertion



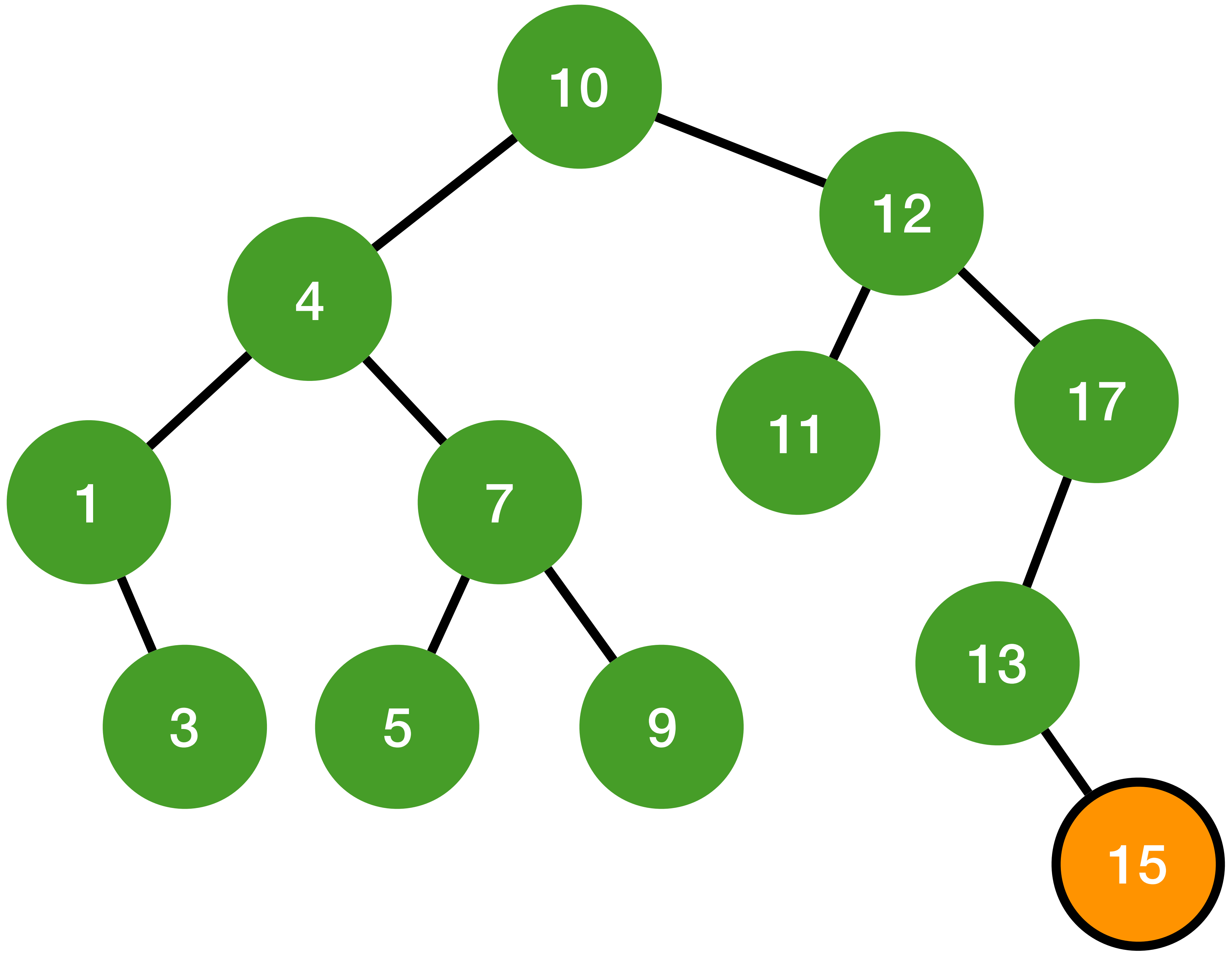
15 < 17 => left

Binary search tree - insertion



15 > 13 => right

Binary search tree - insertion



Binary search tree - insertion algorithm

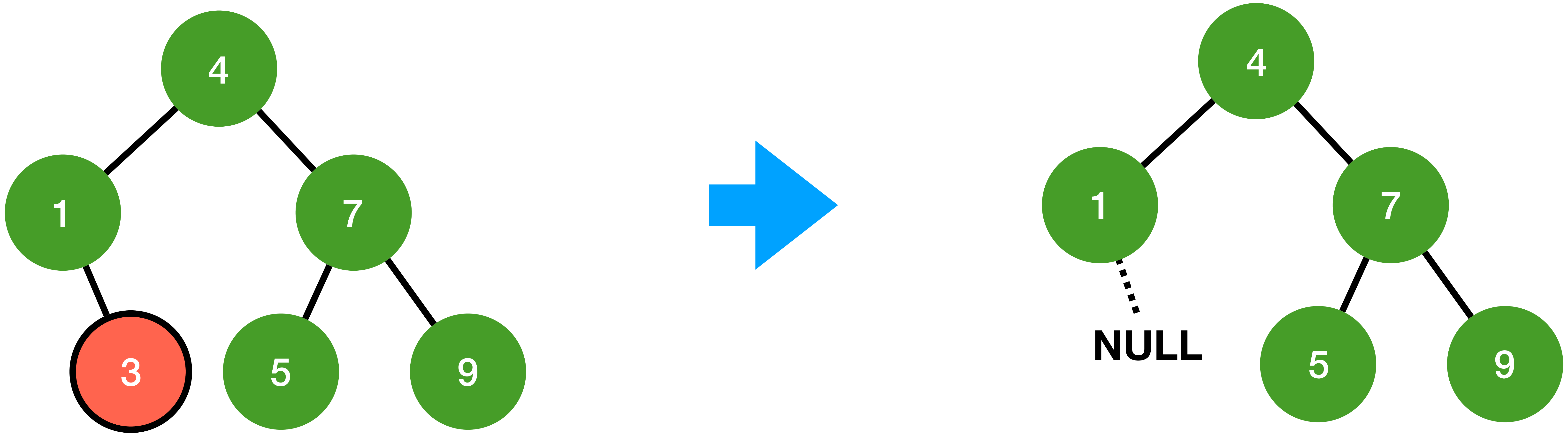
Insert *item* into the tree pointed to by *bst*:

```
TreeNode * insert_non_recursive(TreeNodeData item, TreeNode *bst) {

  TreeNode *temp = bst;
  TreeNode *ptr = bst;

  // find the position where to insert and save in temp
  while (ptr != NULL) {
    temp = ptr;
    if (ptr->data.value < item.value)
      ptr = ptr->right;
    else
      ptr = ptr->left;
  }
```


Binary search tree - insertion algorithm

```
    TreeNode *newNode = (TreeNode *) calloc(sizeof(TreeNode), 1);

    if (newNode == NULL) {
      printf("Error, node could not be allocated");
      exit (EXIT_FAILURE);
    }

    newNode->data = item;
    newNode->left = NULL;
    newNode->right = NULL;
    if ( temp == NULL) // bst is NULL
      bst = newNode;
    // temp points to position to insert
    else if (temp->data.value < item.value)
      temp->right = newNode;
    else
      temp->left = newNode;

    return bst;
}
```

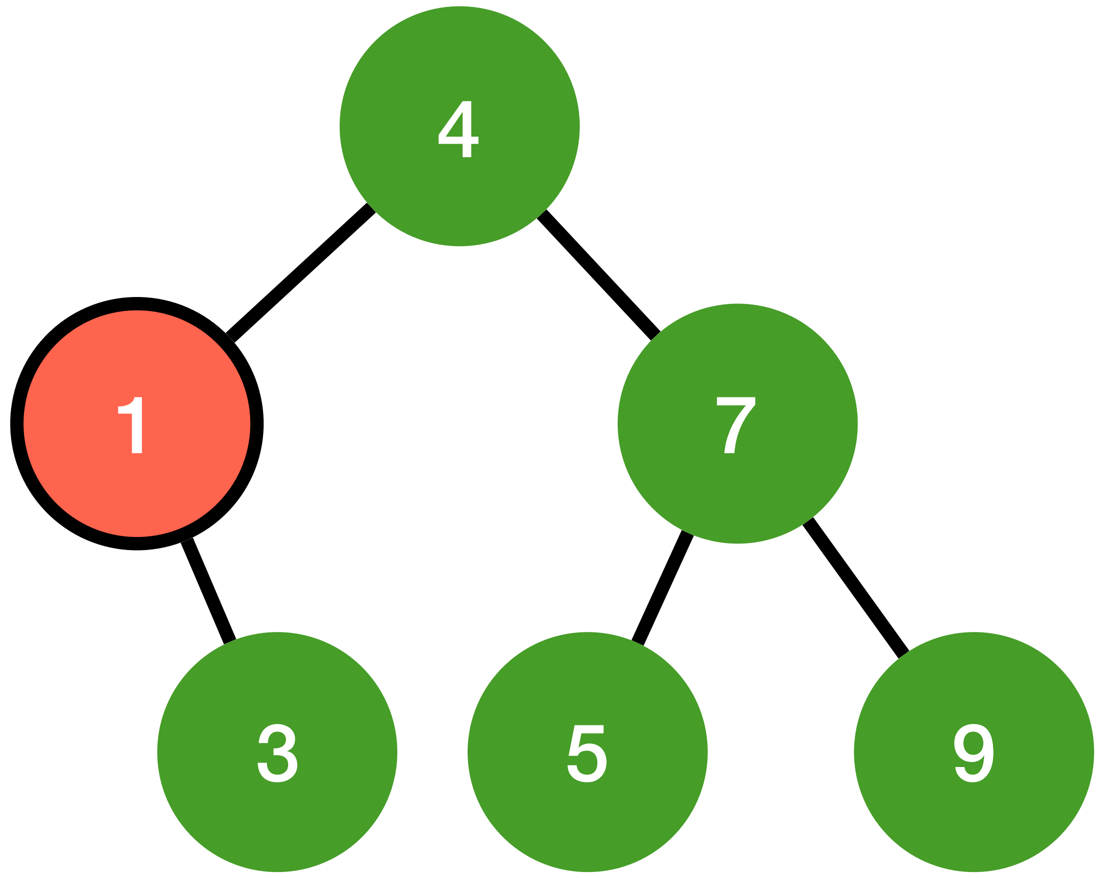

Binary search tree - deletion



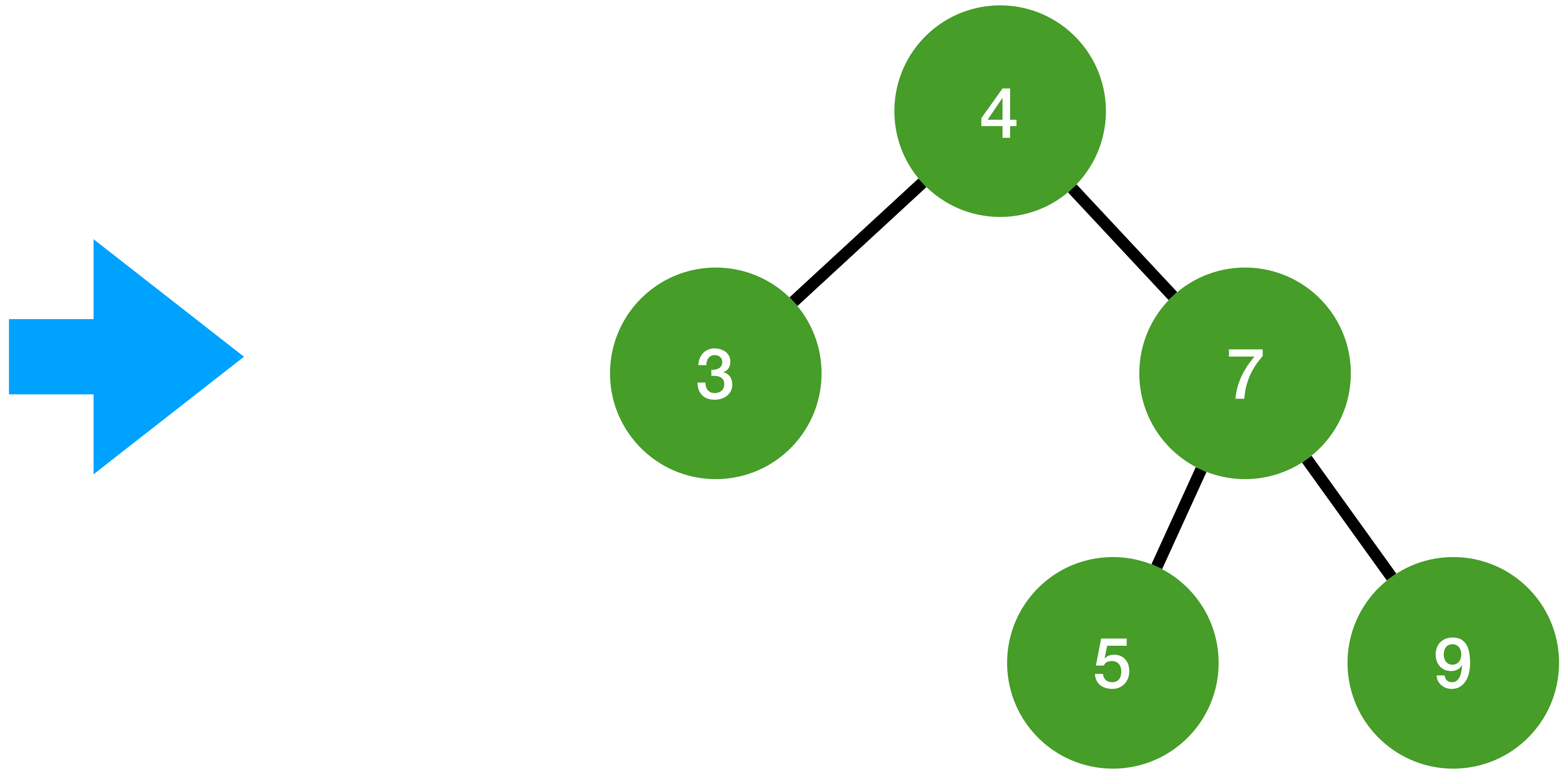
Remove node 3 has no children

remove node 3 and
replace node 3 with NULL as its child

Binary search tree - deletion

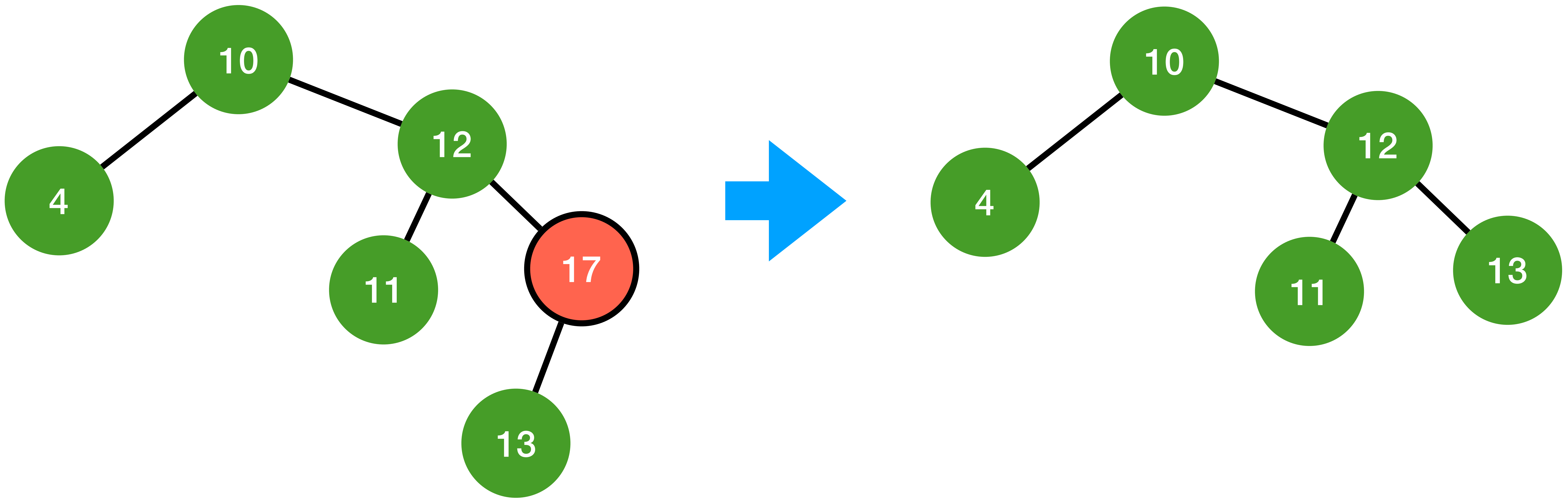


Remove node 1 (with one right child)



Replace node 1 with its right child node 3

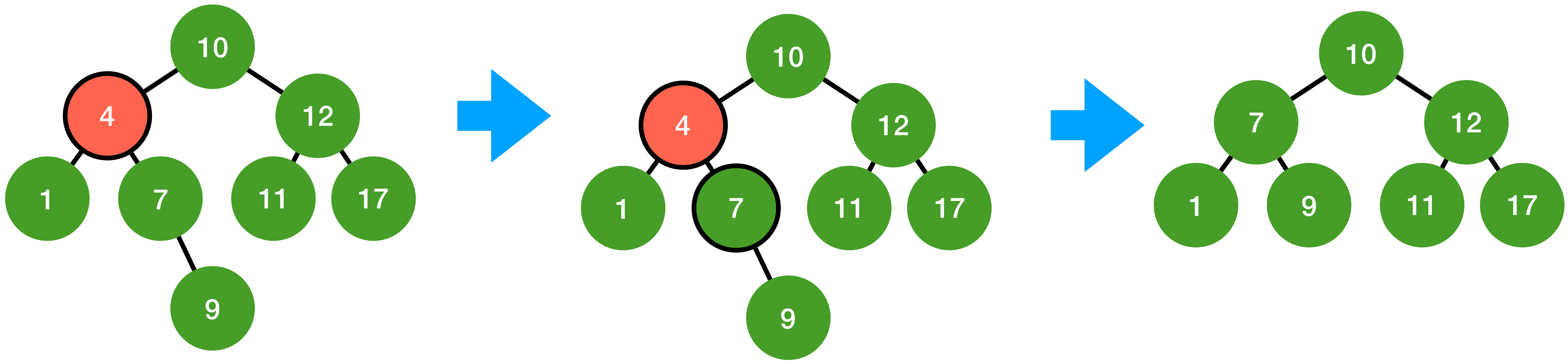
Binary search tree - deletion



Remove node 17 has one left child

Replace node 17 with its left child node

Binary search tree - deletion



Remove node 4 (with two children)

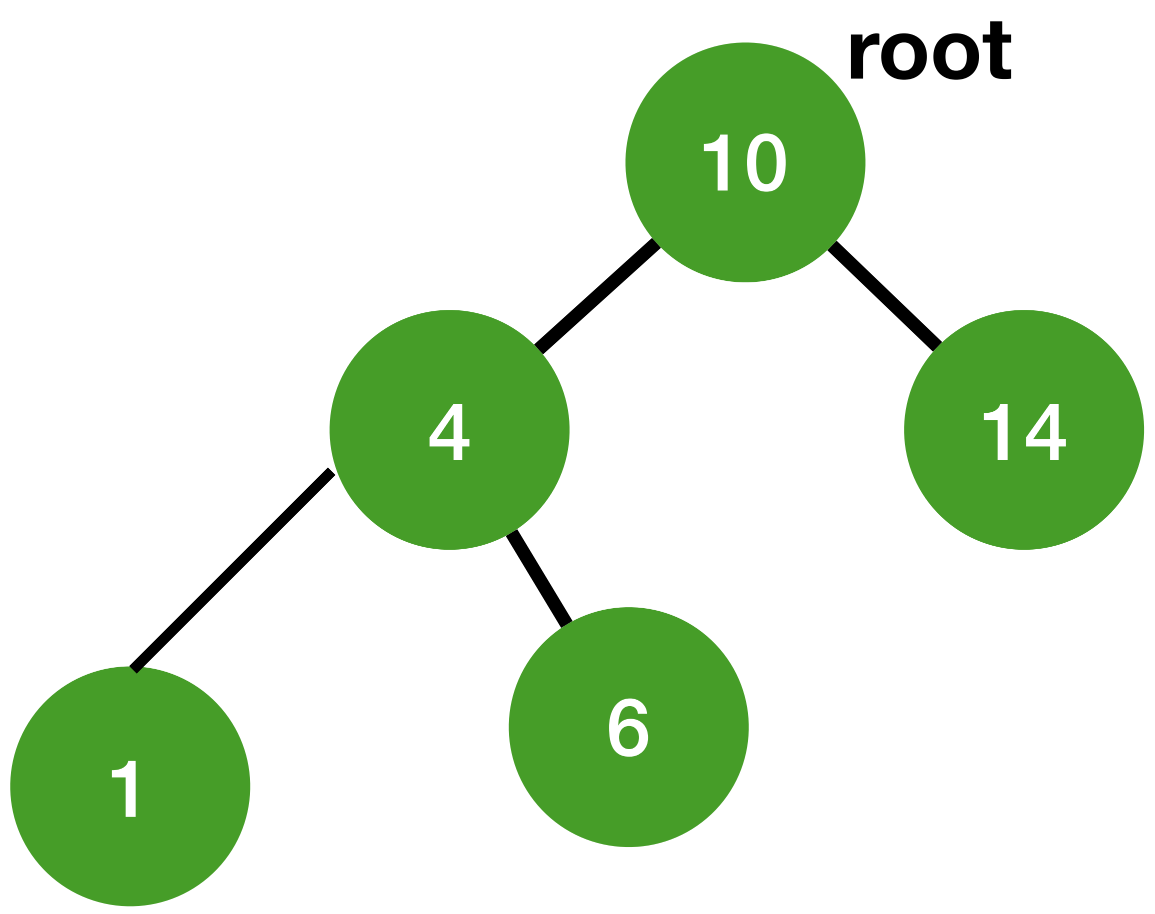
Right child node 7 with one child is its successor

node 7 takes node 4's position

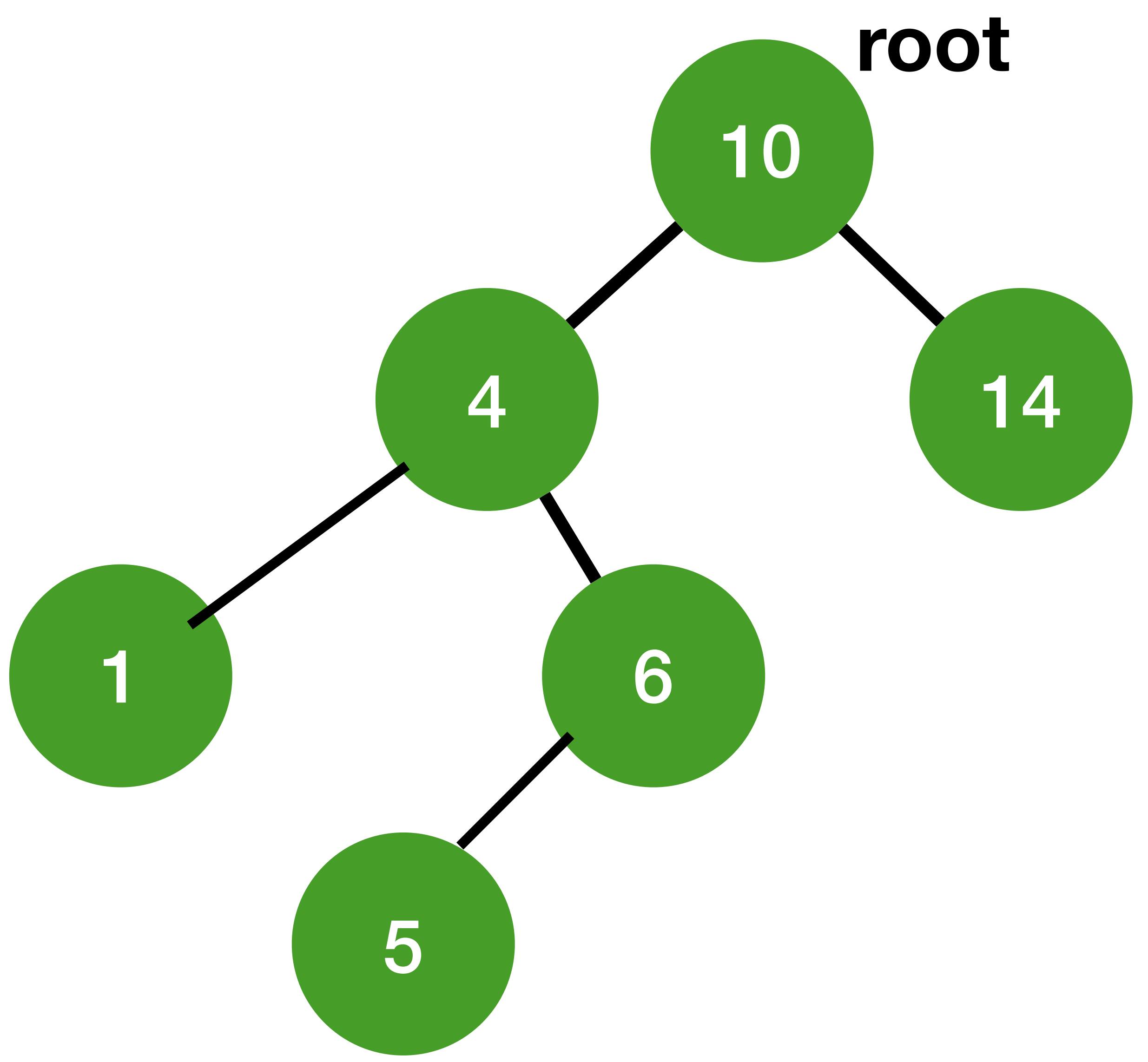
AVL Tree

- Binary search trees can become unbalanced.
- AVL tree is binary search tree with a balance condition:
  - for every node in tree, height of its left and right subtrees can differ by at most 1
  - Ensure that the depth of the tree is $O(\log n)$

AVL vs non-AVL tree



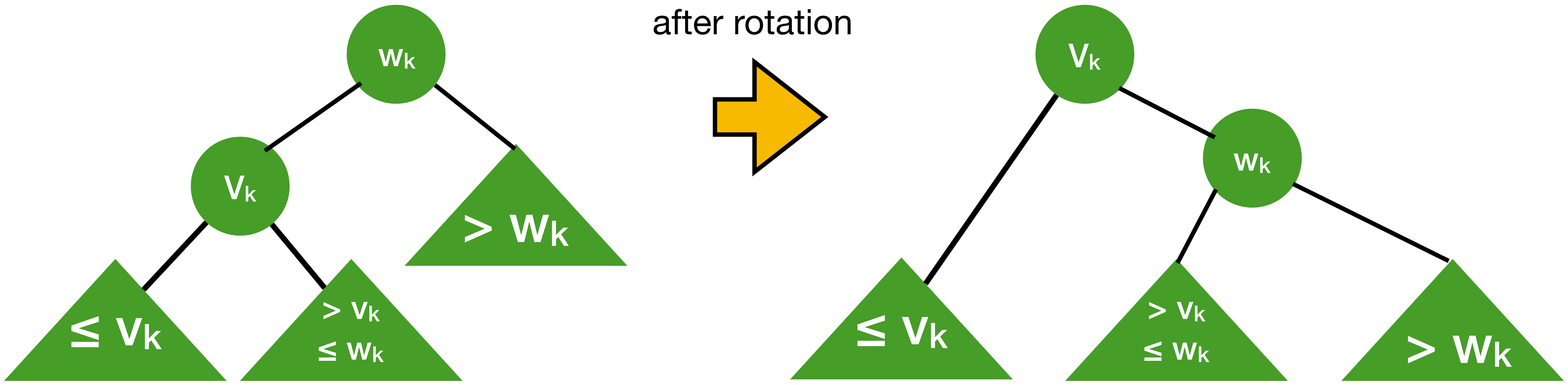
A binary search tree that is an AVL tree



A binary search tree that is not an AVL tree

Single Rotations

- Single rotations can be done at any node to convert a non AVL binary search tree to an AVL tree.



Further Reading

- Binary and AVL search trees in recommended texts
- Cormen et al. Introduction to Algorithms. Binary Search Tree
- https://en.wikipedia.org/wiki/Tree_traversal

