ECE 220: Computer Systems & Programming

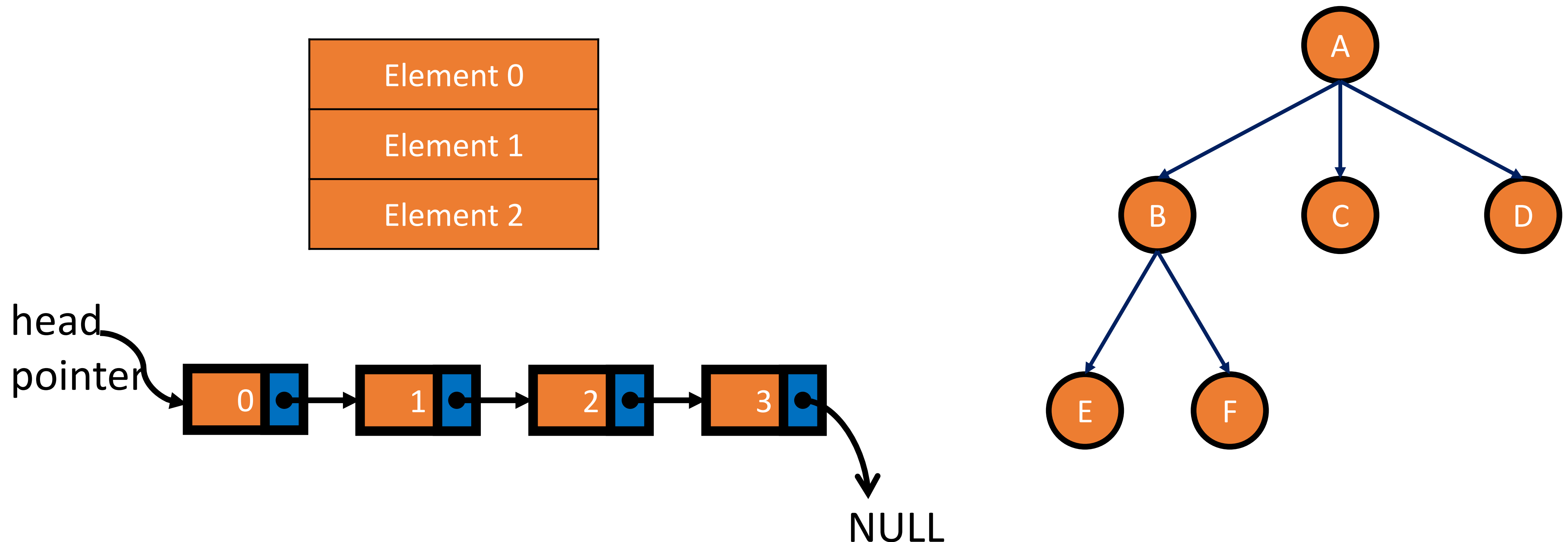
Lecture 19: Trees, Traversal and Search

Thomas Moon

April 2, 2024

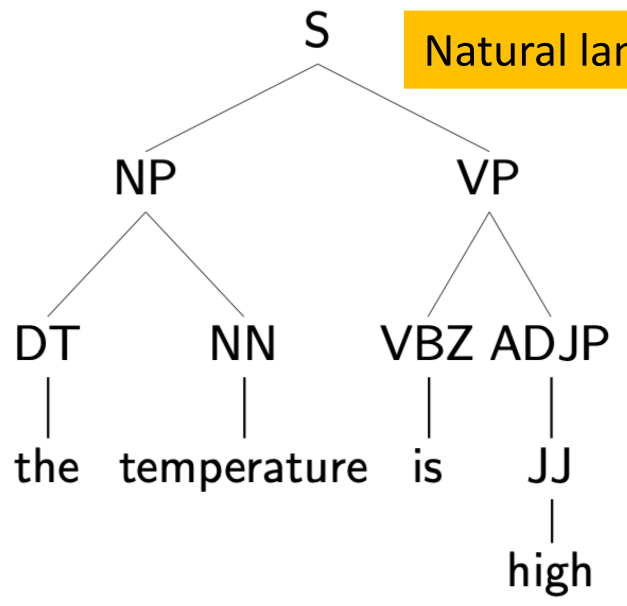
Tree Data Structure

- Array, Linked List, Stack, Queue: Linear data structures
- Tree: Hierarchical Non-linear data structure, a collection of nodes connected by links (or edges)

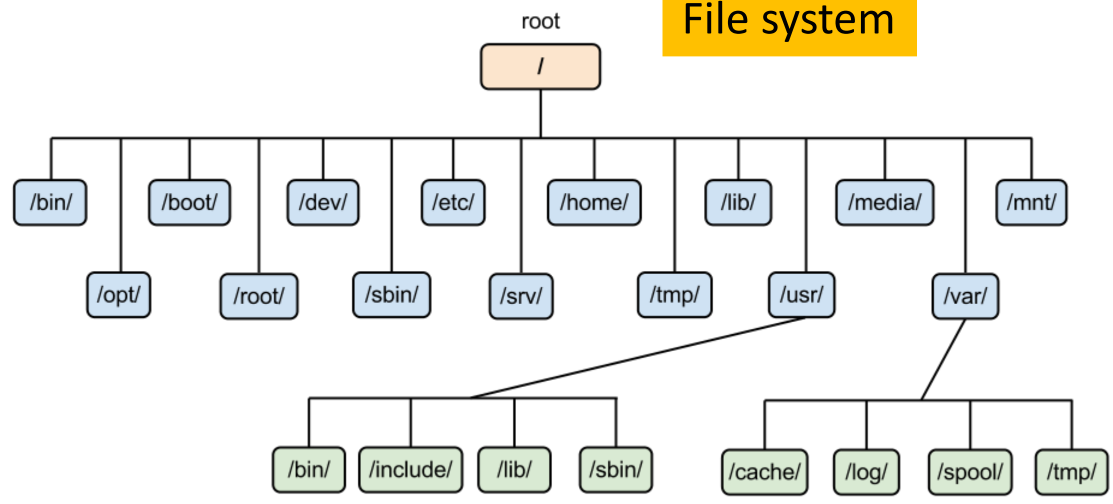


Application of Tree

Natural language processing



File system

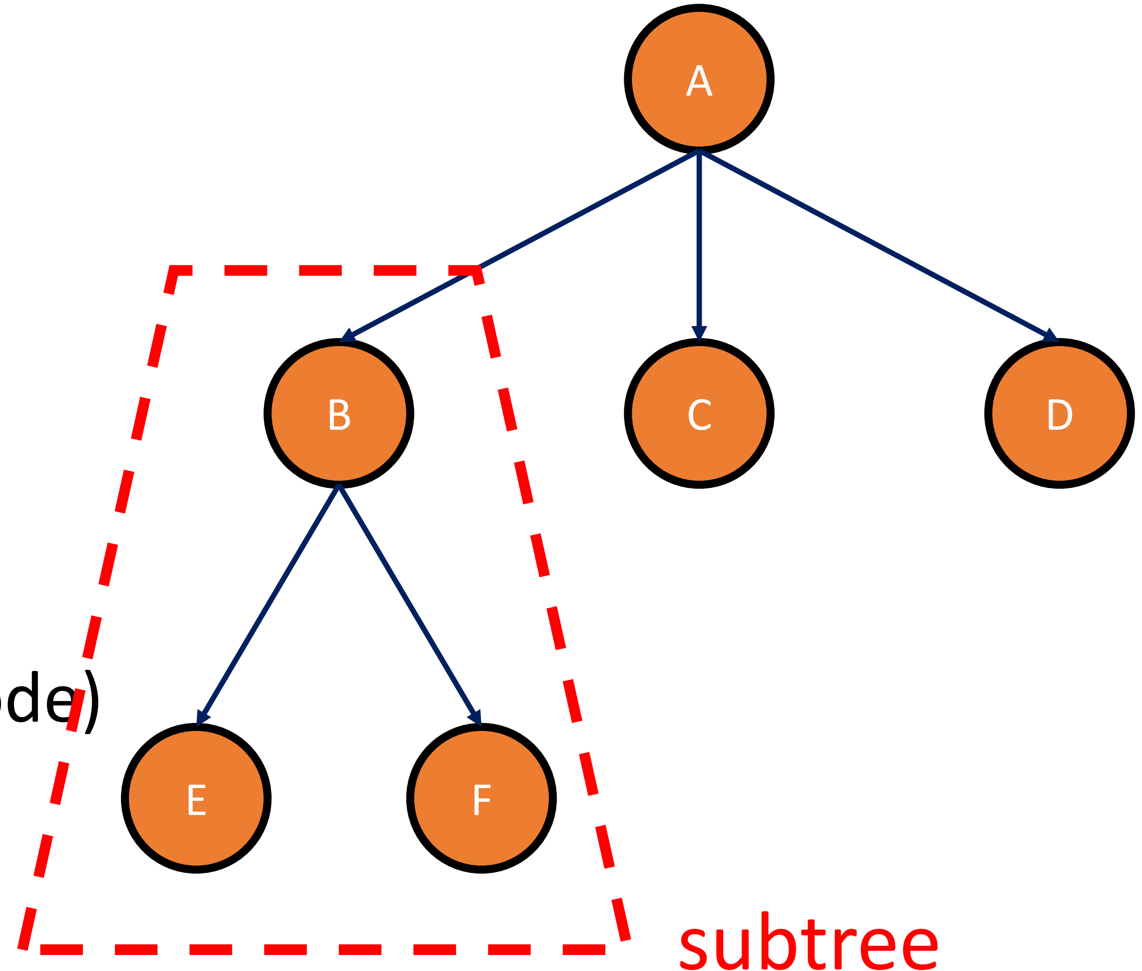


Computer langauge

statement sequence
while
return
condition
body
compare op: ≠
variable name: b
constant value: 0
branch
variable name: a
condition
if-body
else-body
compare op: >
assign
assign
variable name: a
variable name: b
variable name: a
bin op op: −
variable name: b
bin op op: −
variable name: a
variable name: b
variable name: b
variable name: a

Terminology of Tree

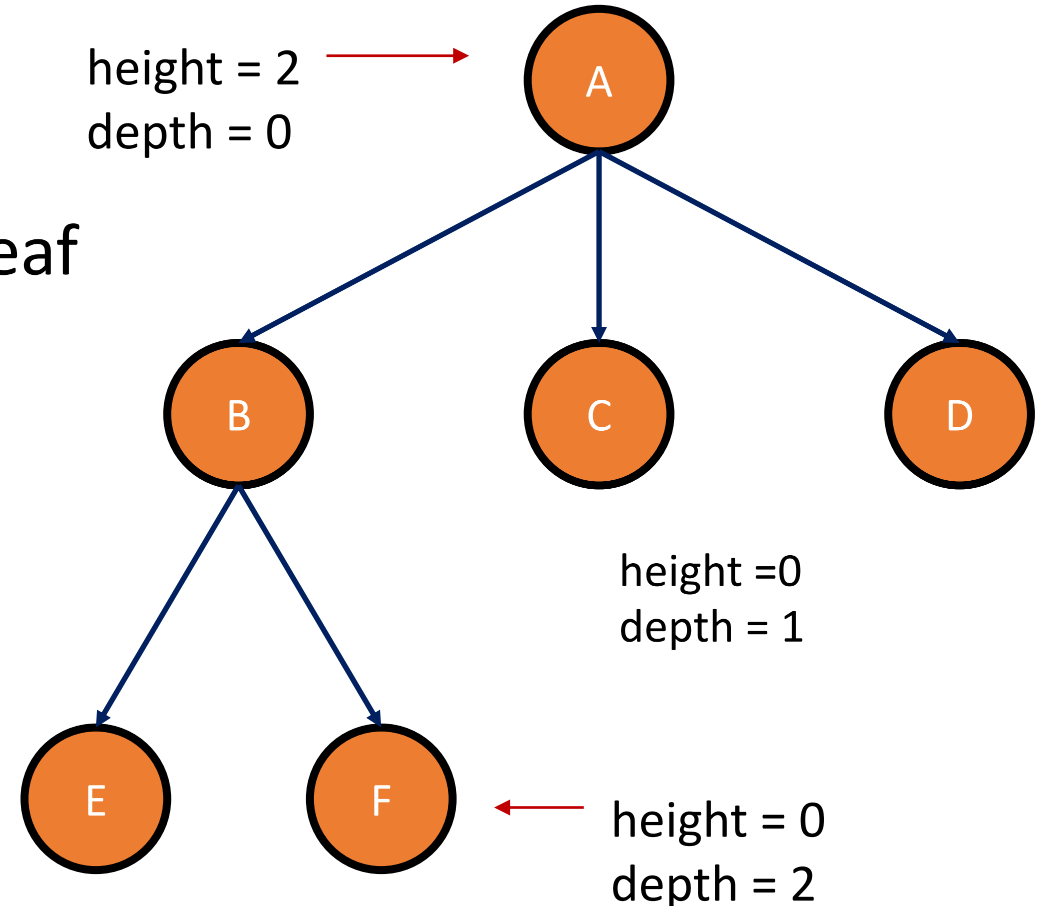
- root node (topmost node)
  A
- child nodes of A:
  B, C, D
- parent node of F:
  B
- inner nodes (has child node(s))
  A, B
- leaf nodes (does not have child node)
  E, F, C, D
- sibling nodes of D:
  B, C



Terminology of Tree

- length of a path = number of edges

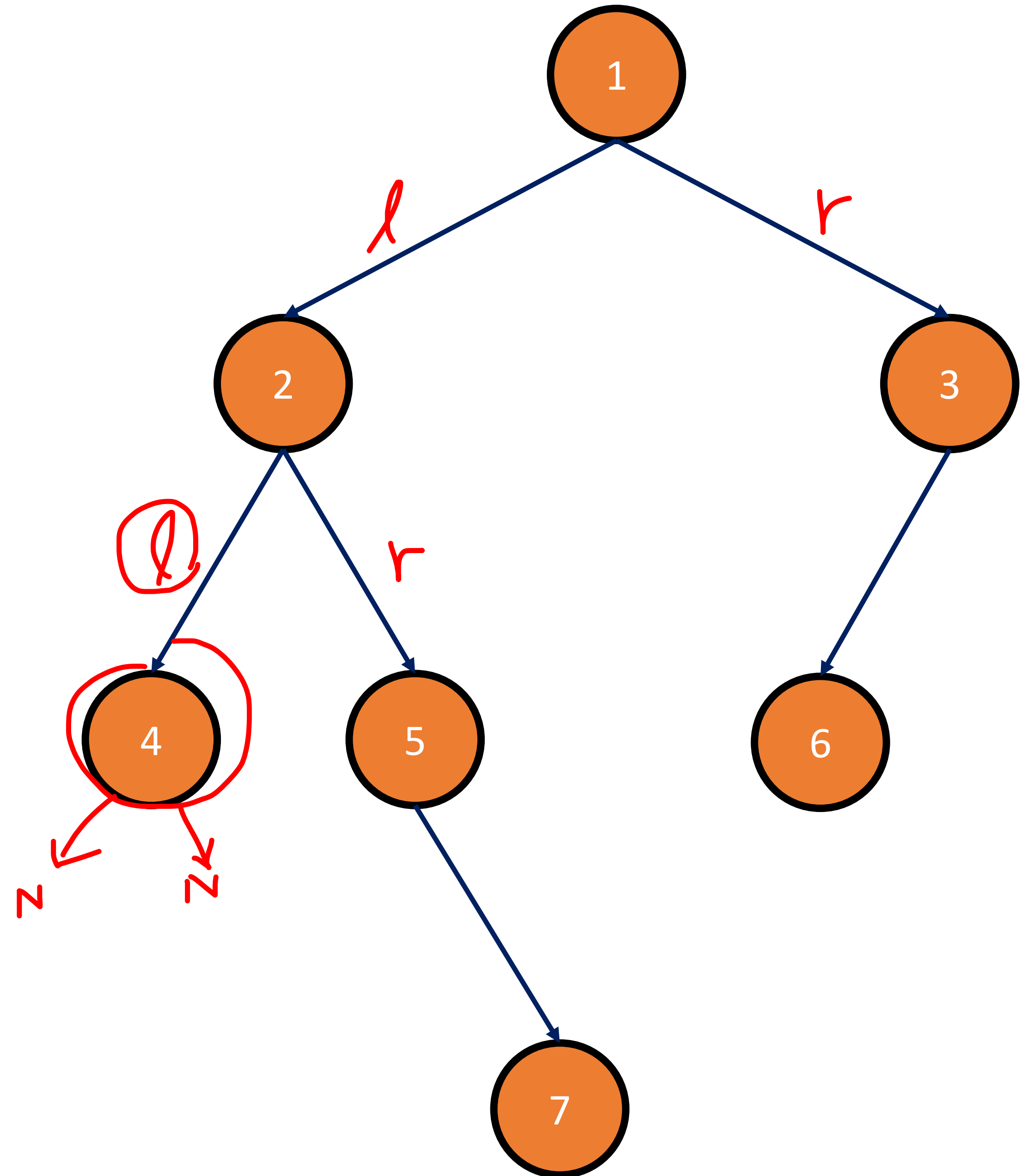
  length of path A-B-E = 2
- Height of a node = length of the longest path from the node to a leaf
- Depth of a node = length of path from root to the node
- Height of tree = height of root



Binary Tree

- Each node has at most 2 child nodes.

```
typedef struct nodeTag t_node;

struct nodeTag
{
    int data;
    t_node *left;
    t_node *right;
};
```

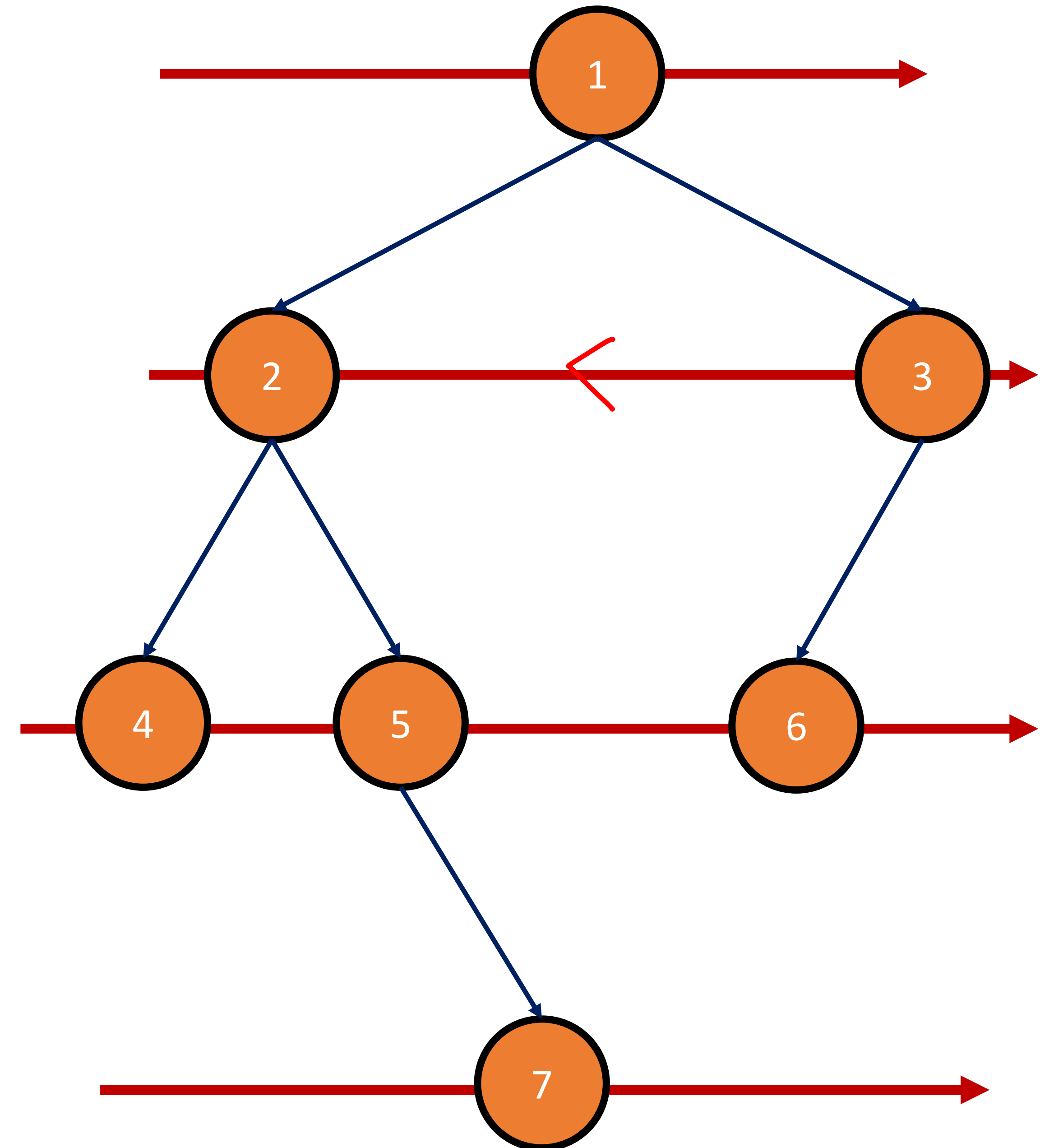


Tree Traverse – BFS

- <u>Breadth-First Search </u>(level order)

  1 2 3 4 5 6 7

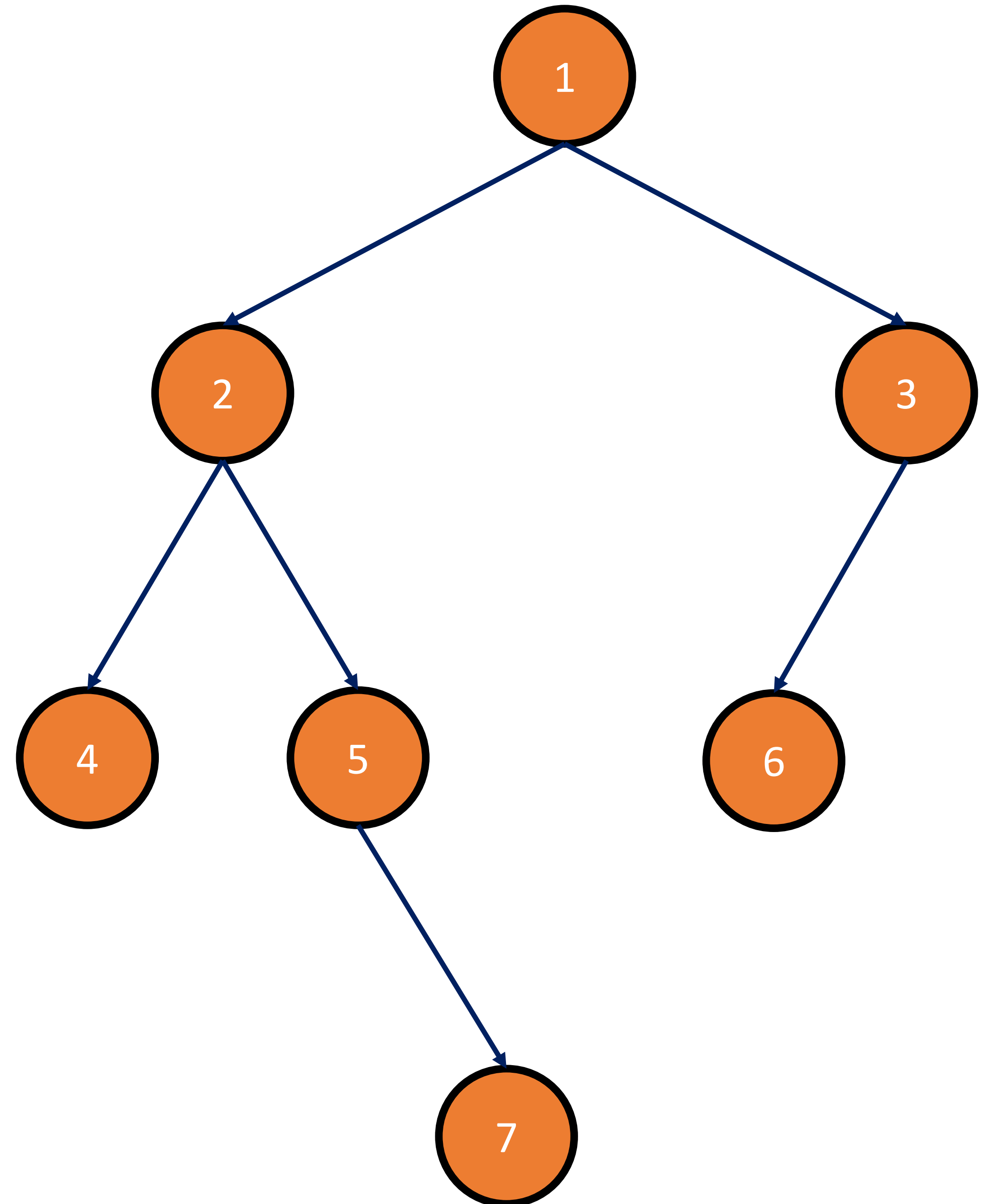
Traverse through all the children of a node, then visit the grandchildren.



Tree Traverse –DFS

- Depth-First Search

1. Pre-order: root, left, right
   1 2 4 5 7 3 6
2. In-order: left, root, right
   4 2 5 7 1 6 3
3. Post-order: left, right, root
   4 7 5 2 6 3 1



Tree Traverse –DFS

- Depth-First Search

1. Pre-order: root, left, right

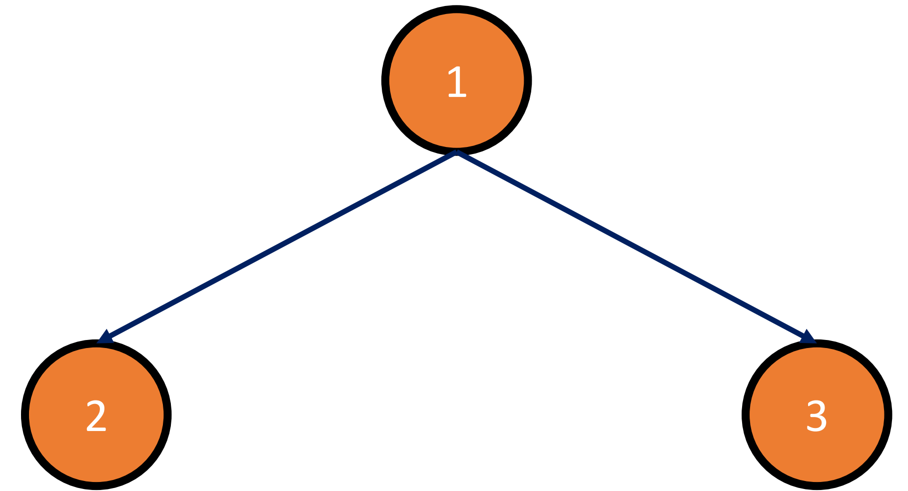
   1 2 3

2. In-order: left, root, right

   2 1 3

3. Post-order: left, right, root

   2 3 1



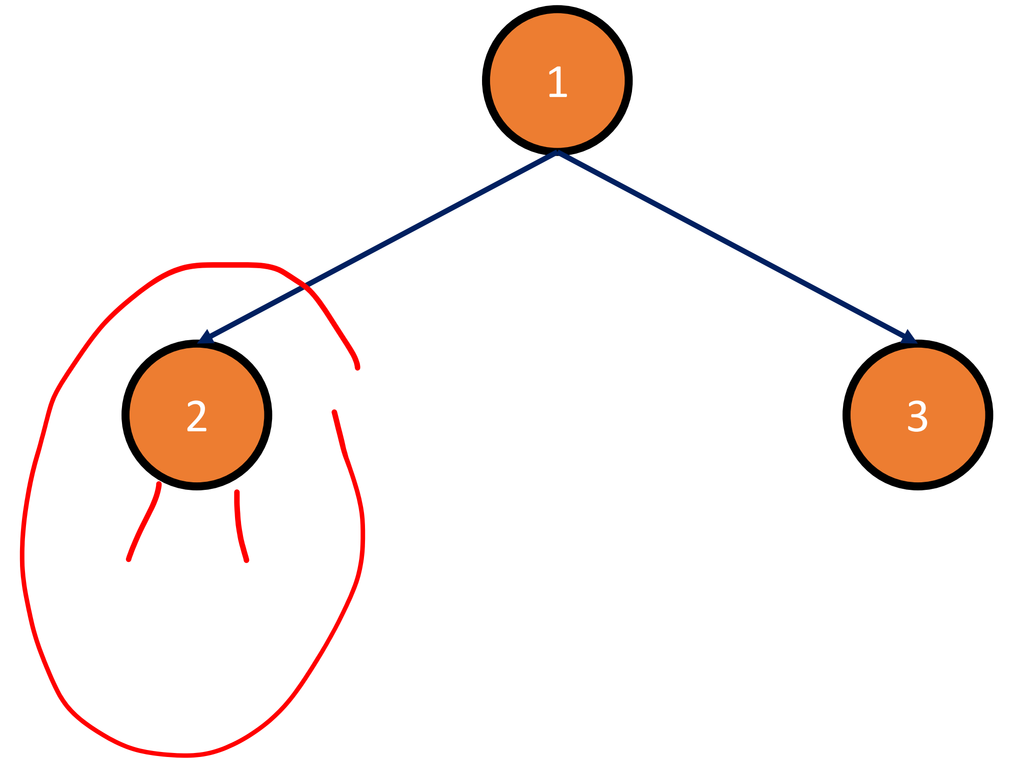
Tree Traverse –DFS

- Depth-First Search

1. Pre-order: root, left, right

For each node, read the data of the node, then visit the left subtree and then the right subtree.

1 2 3

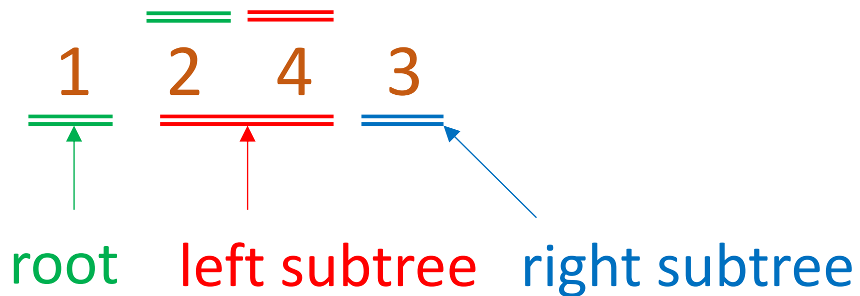


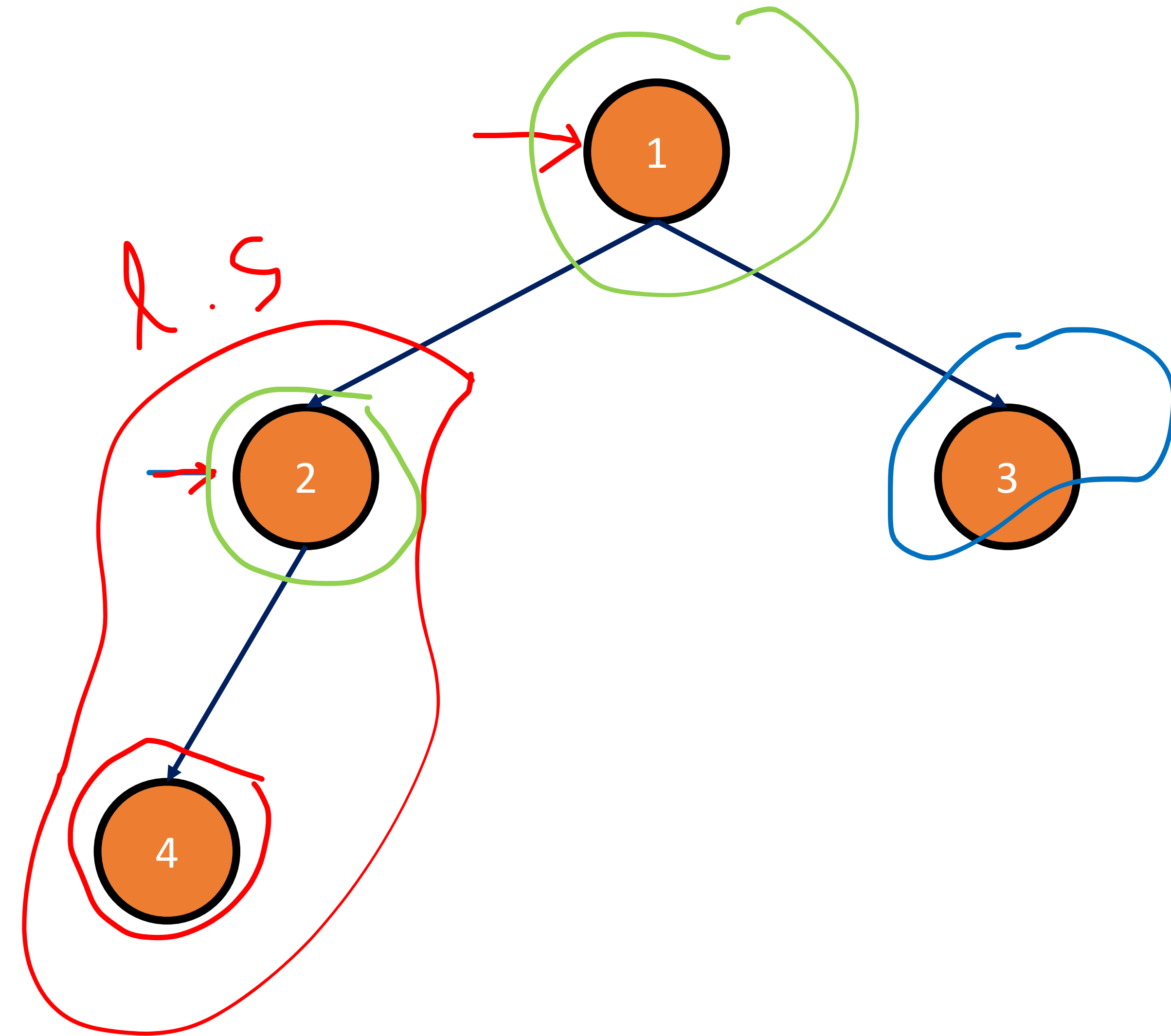
Tree Traverse –DFS

- <u>Depth-First Search</u>

1. Pre-order: root, left, right

For each node, read the data of the node, then visit the left subtree and then the right subtree.



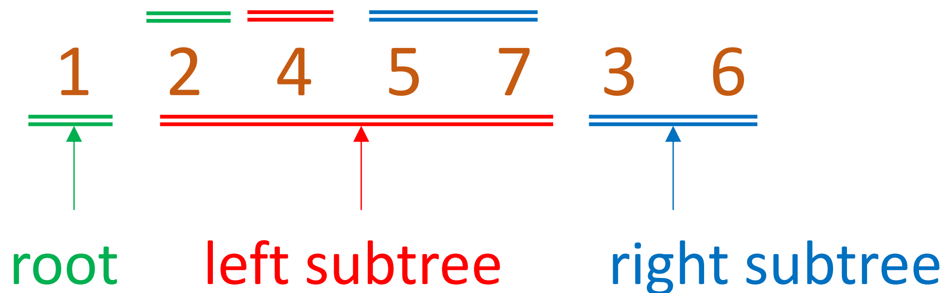


Tree Traverse –DFS

- Depth-First Search

1. Pre-order: root, left, right

For each node, read the data of the node, then visit the left subtree and then the right subtree.

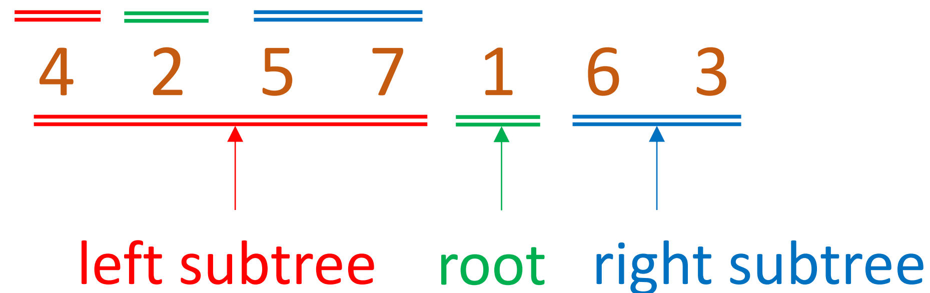


Tree Traverse –DFS

- Depth-First Search

2. In-order: left, root, right

For each node, visit the left subtree, then read the data of the node, then visit the right subtree.

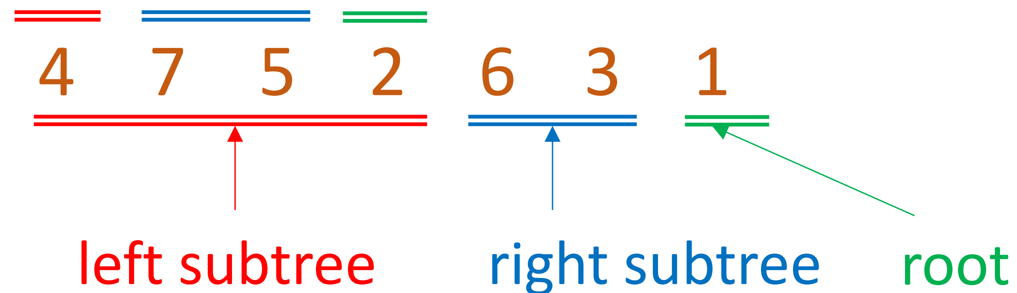


Tree Traverse –DFS

- Depth-First Search

3. Post-order: left, right, root

For each node, visit the left subtree, then visit the right subtree, then read the data of the node.



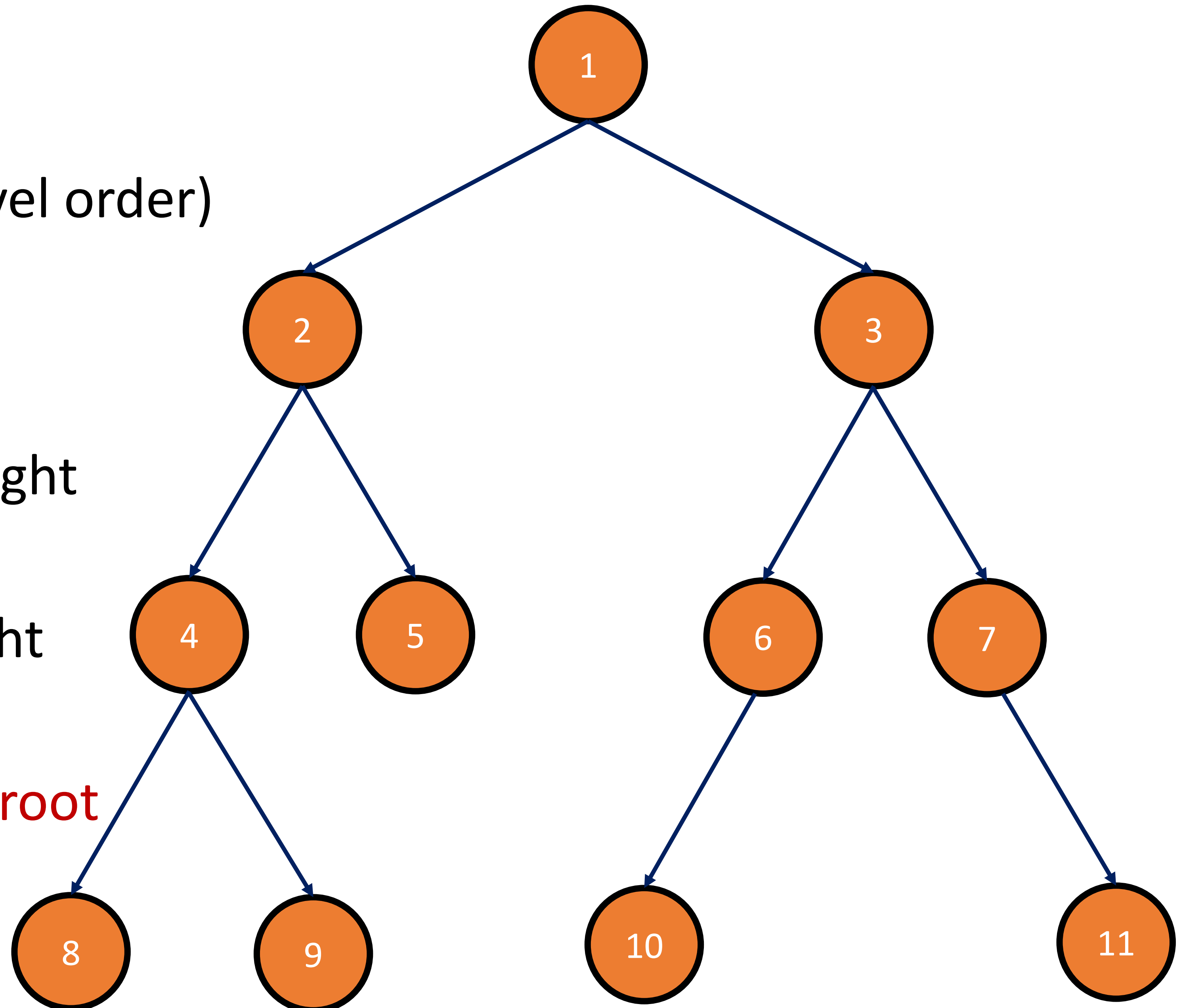
Practice

- Breadth-First Search (level order)

1 2 3 4 5 6 7 8 9 10 11

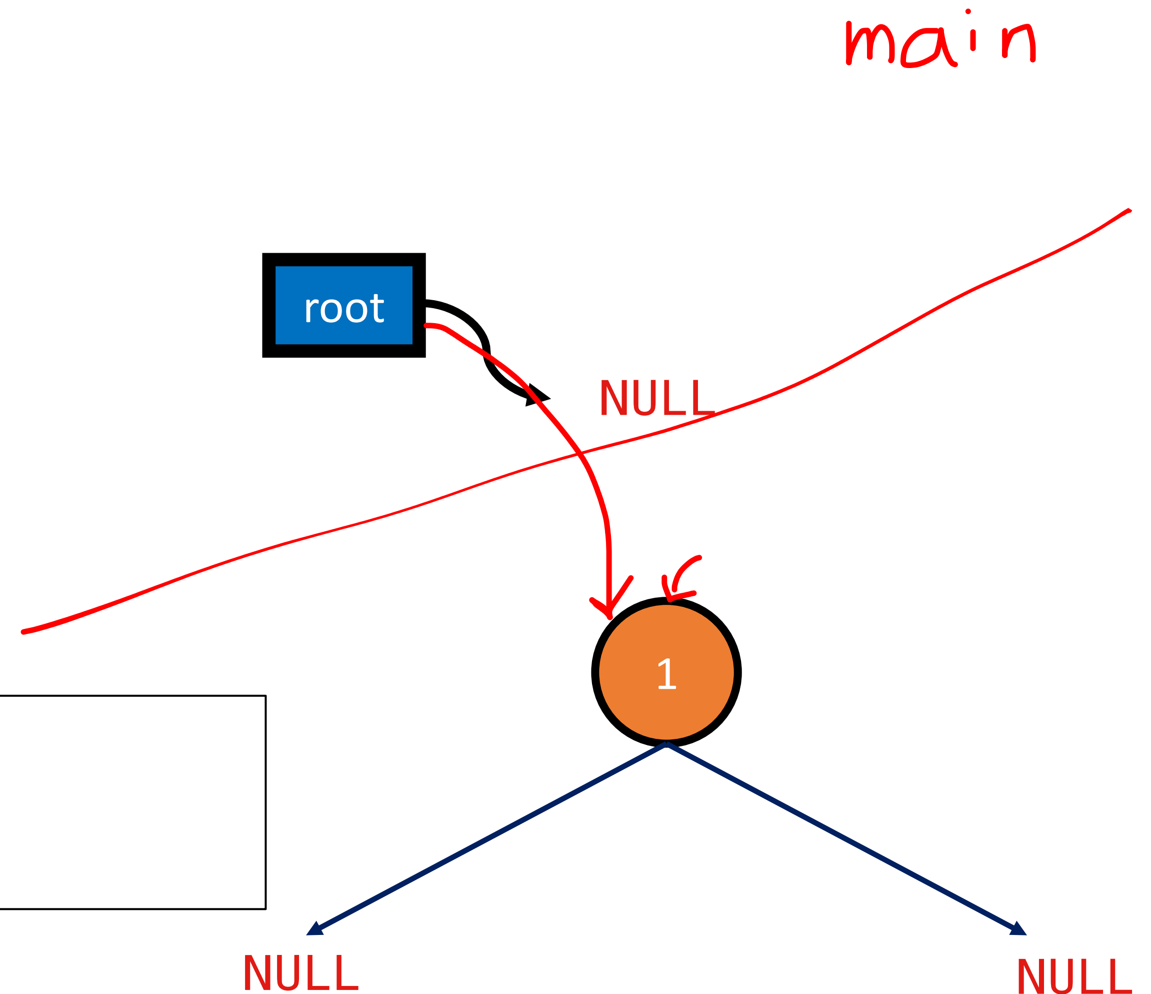
- Depth-First Search

1. Pre-order: root, left, right
   1 2 4 8 9 5 3 6 10 7 11
2. In-order: left, root, right
   8 4 9 2 5 1 10 6 3 7 11
3. Post-order: left, right, root
   8 9 4 5 2 10 6 11 7 3 1



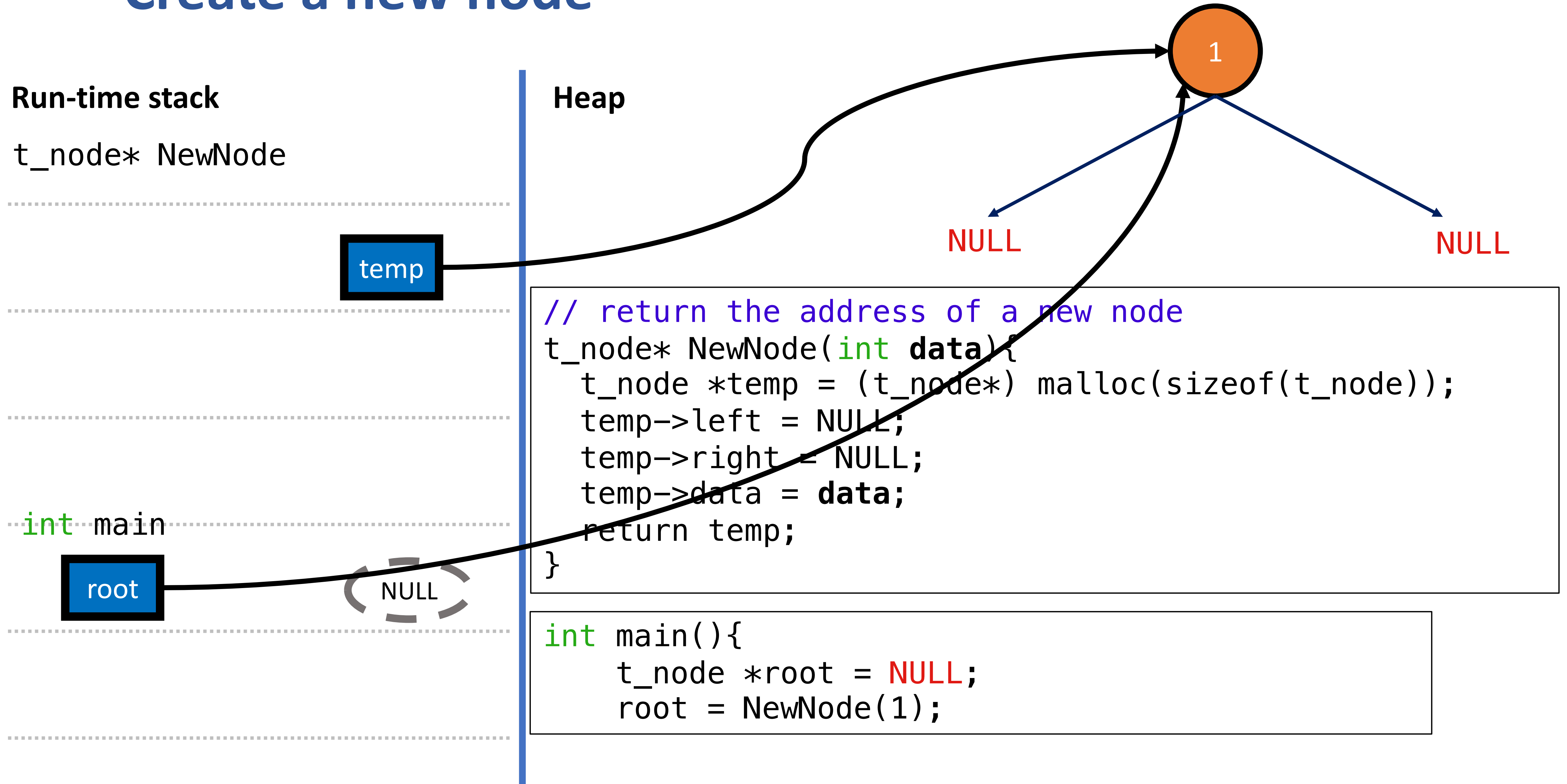
Tree Structure

```
typedef struct nodeTag t_node;
struct nodeTag
{
    int data;
    t_node *left;
    t_node *right;
};


int main(){
    t_node *root = NULL;
    root = NewNode(1);
```




```
// return the address of a new node
t_node* NewNode(int data){
}
```

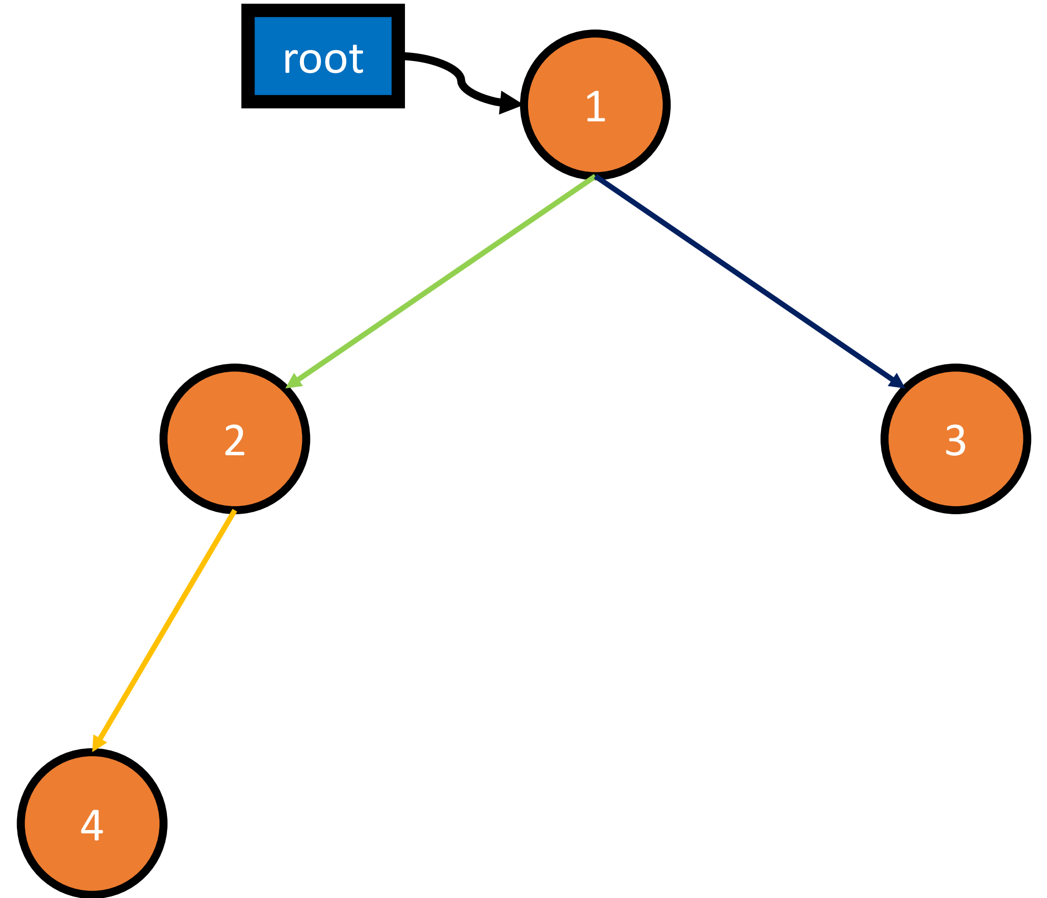
Create a new node



Tree Structure

Construct the tree by using NewNode

```
int main(){
    t_node *root = NULL;


    root = NewNode(1);
    root->left = NewNode(2);
    root->left->left = NewNode(4);

    root->right = NewNode(3);
```



Implement DFS by Recursive

- Depth-First Search

2. In-order: left, root, right

For each node, visit the left subtree, then print the data of the node, then visit the right subtree.

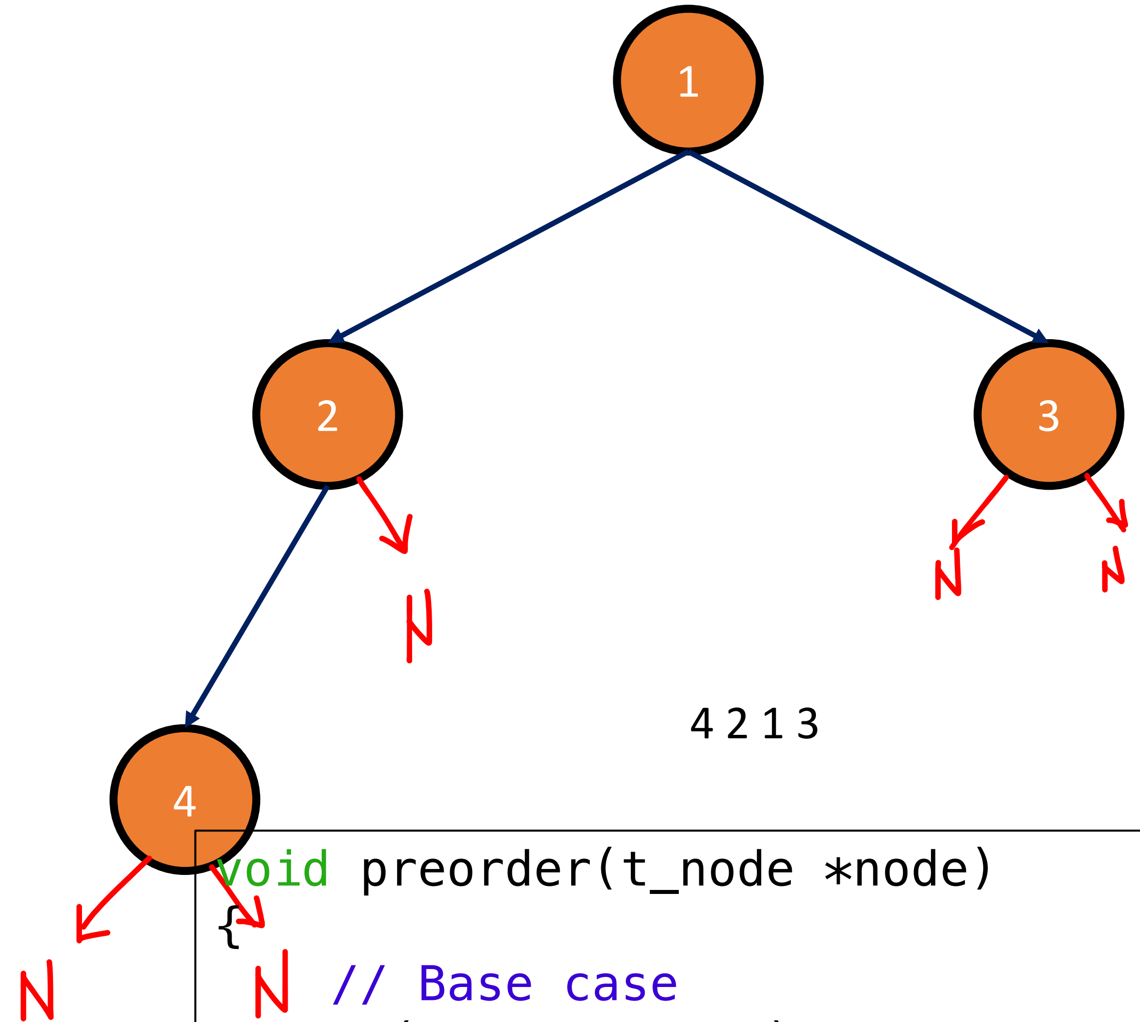
- Do the same thing until we visit all the nodes.
- Same actions in slightly different order for “pre-order” and “post-order”.

```
void inorder(t_node *node)
{
    // Base case
    if(node _______________)
        return;
    // Recursive case

}
```

Tree Structure

```
int main(){
    t_node *root = NULL;
    …

    inorder(root);
```

```
void inorder(t_node *node)
{
    // Base case
    if(node == NULL)
        return;
    // Recursive case
    inorder(node->left);
    printf("%d ", node->data);
    inorder(node->right);
}
```

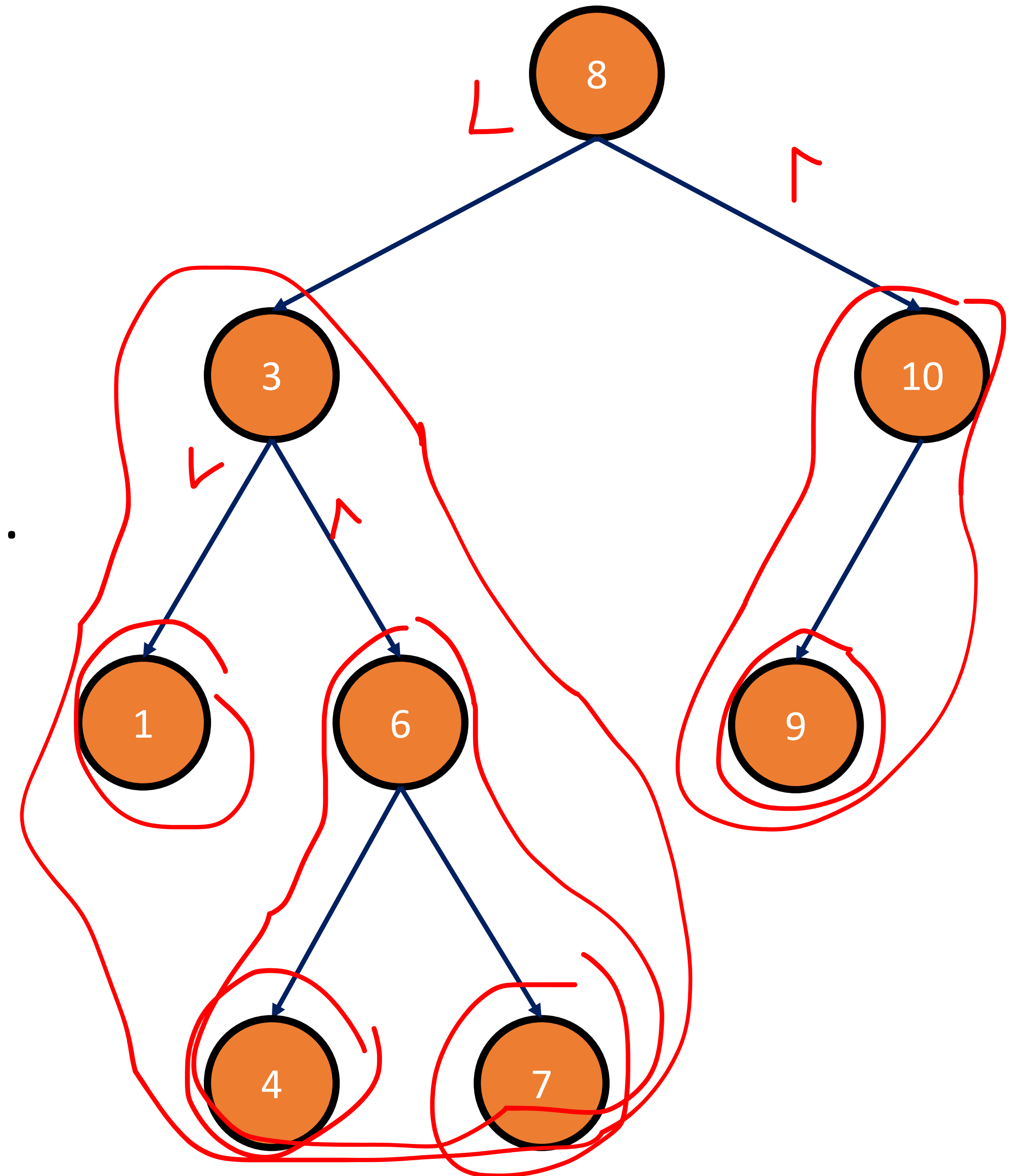



```
void preorder(t_node *node)
{
    // Base case
    if(node == NULL)
        return;
    // Recursive case
    printf("%d ", node->data);
    preorder(node->left);
    preorder(node->right);
}
```

Binary Search Tree (BST)

BST is an *ordered (sorted)* binary tree.

- Data of nodes on the left subtree is smaller than the data of parent node.
- Data of nodes on the right subtree is larger than the data of parent node.
- Both left and right subtree must also be BST.
- Data in each node is unique.

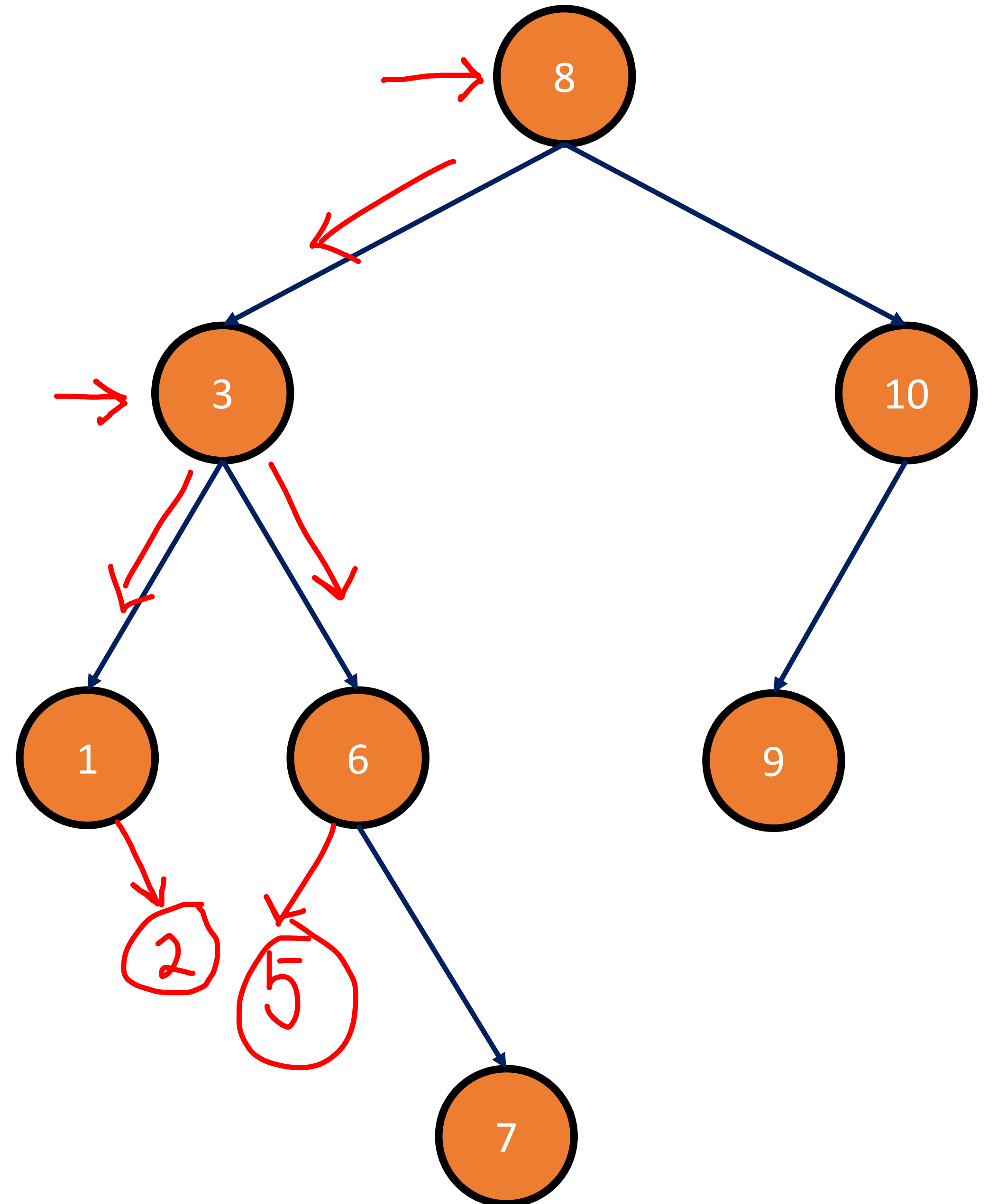


Insert a New Node to a BST

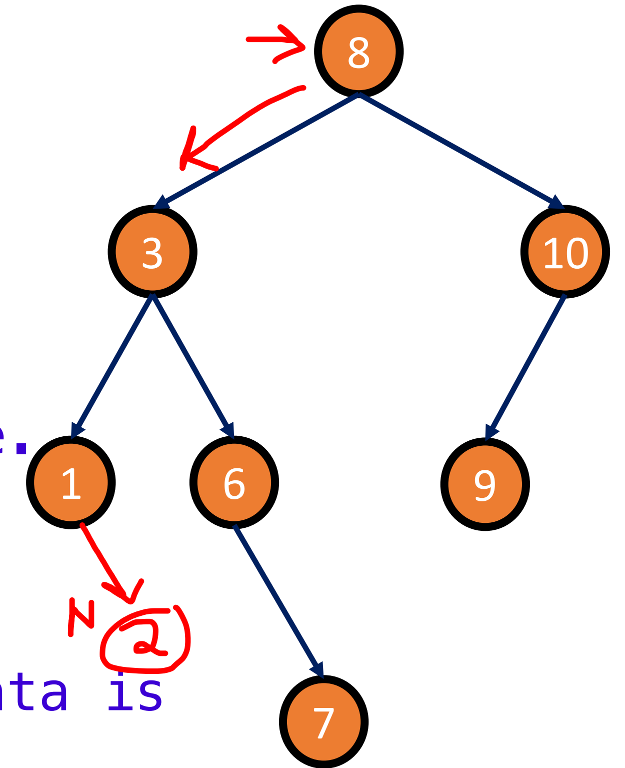
- A new node is always inserted at leaf nodes.

- Where should a new node with ‘2’ be inserted in this BST?

5?

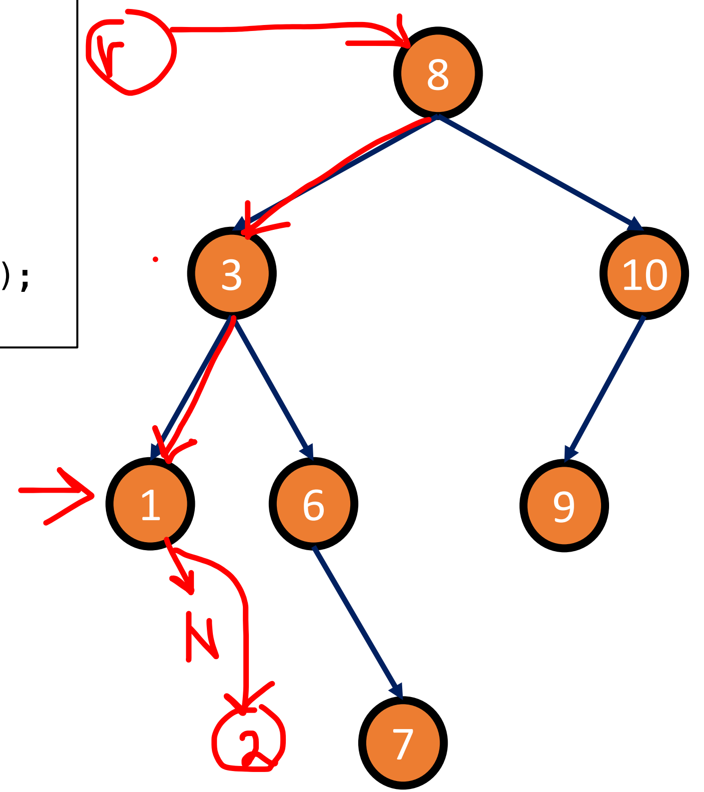


BST Insertion by Recursion




```
t_node* BSTInsert(t_node *node, int data){
    //base case : Found the right place to insert the node.
    if(node ==NULL){
        return NewNode(data);
    }
    // recursive case: Traverse either to the left (new data is
            smaller) or the right (new data is larger)
    else{
        if(data < node->data)
            node->left = BSTInsert(node->left , data);
        else
            node->right = BSTInsert(node->right , data);
        // return the unchanged node
        return node;
    }
}
```

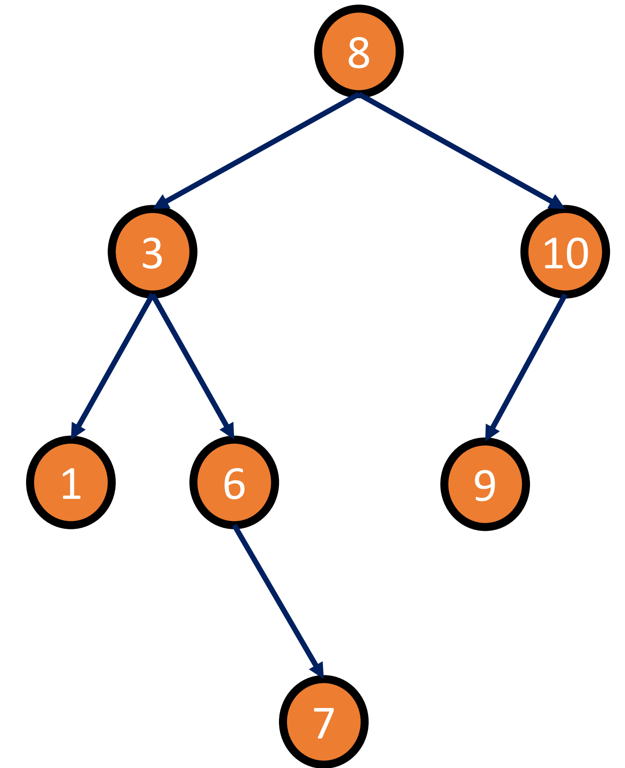
```
t_node* BSTInsert(t_node *node, int data){
    if(node ==NULL)
        return NewNode(data);
    else{
        if(data < node->data)
            node->left = BSTInsert(node->left , data);
        else
            node->right = BSTInsert(node->right , data);
    return node;
```




```
                8
root= BSTInsert(root, 2);
                    return root;
                                   3
    root->left = BSTInsert(root->left, 2);
                            return root->left;
                                                1
        root->left->left = BSTInsert(root->left->left, 2);
                                  return root->left->left;
                                                           NULL
            root->left->left->right = BSTInsert(root->left->left->right, 2);

                                            return NewNode(2);
```

BST Search

```
t_node* BSTSearch(t_node *node, int key){
    // base case
    // 1. no match
    if(___________)
        return NULL;
    // 2. yes match
    if(node->data == key)
        return node;

    // recursive case
    if(key < node->data)
        return BSTSearch(___________, key);
    else
        return BSTSearch(___________, key);
}
```



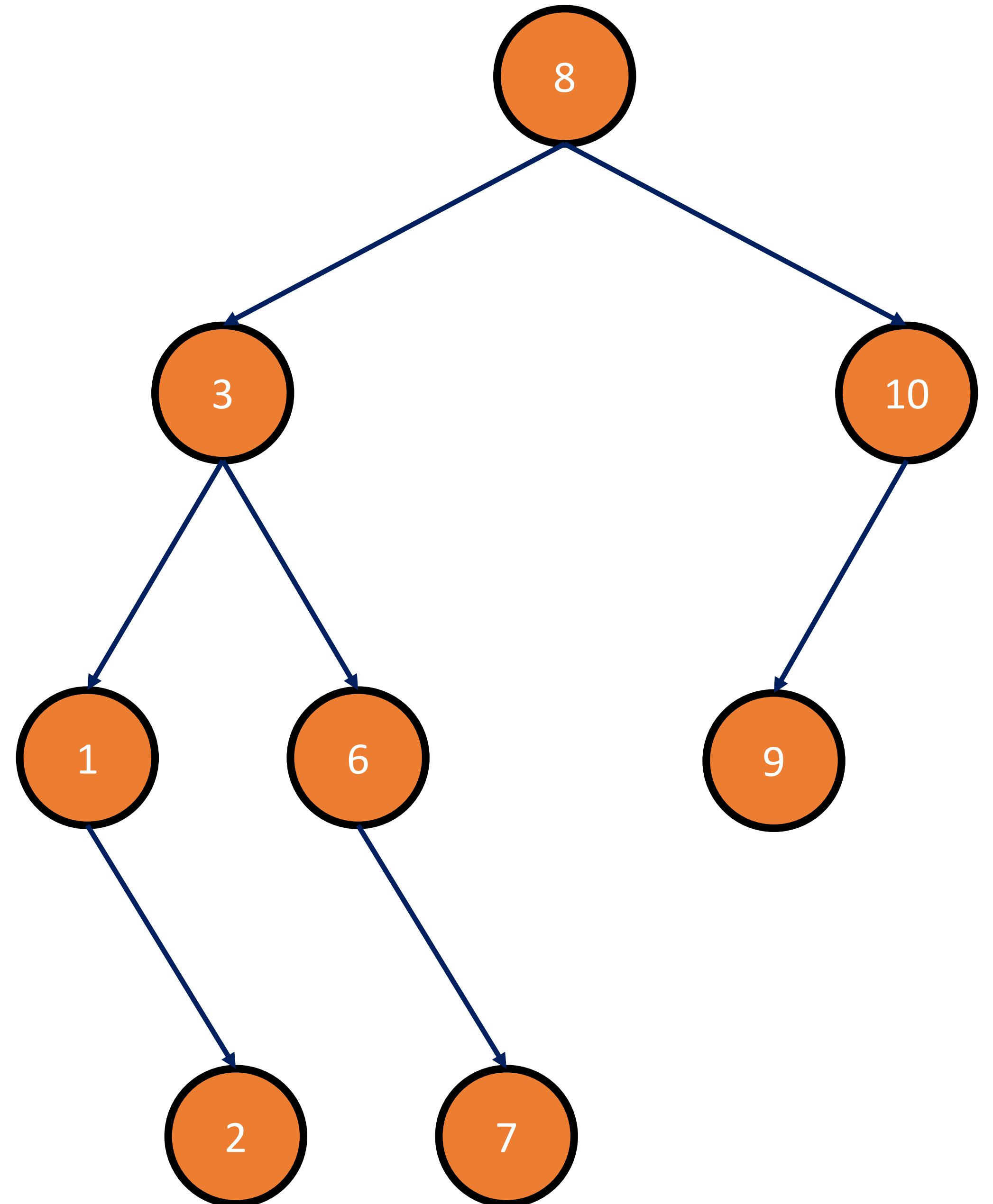
Find Min & Max in BST

- How to find the minimum in BST?

=>Traverse left! (until your left is NULL)

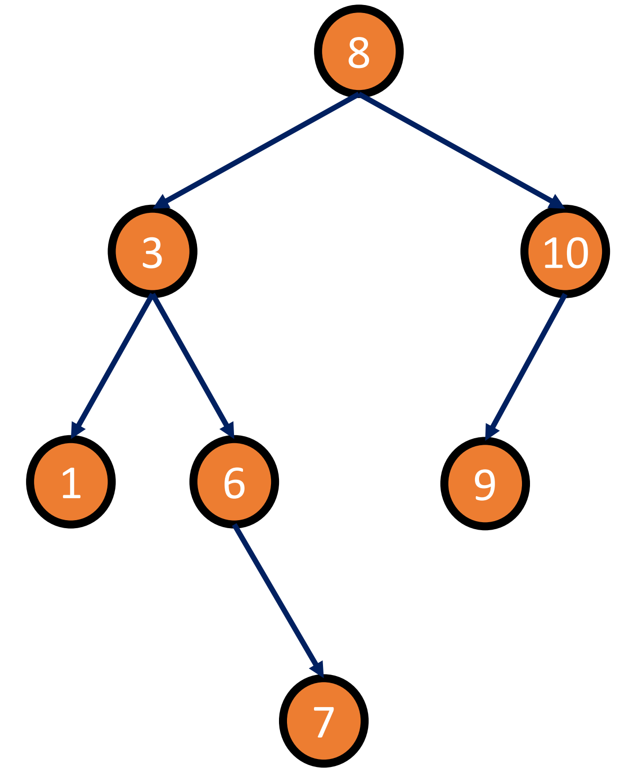
- How to find the maximum in BST?

=>Traverse right! (until your right is NULL)



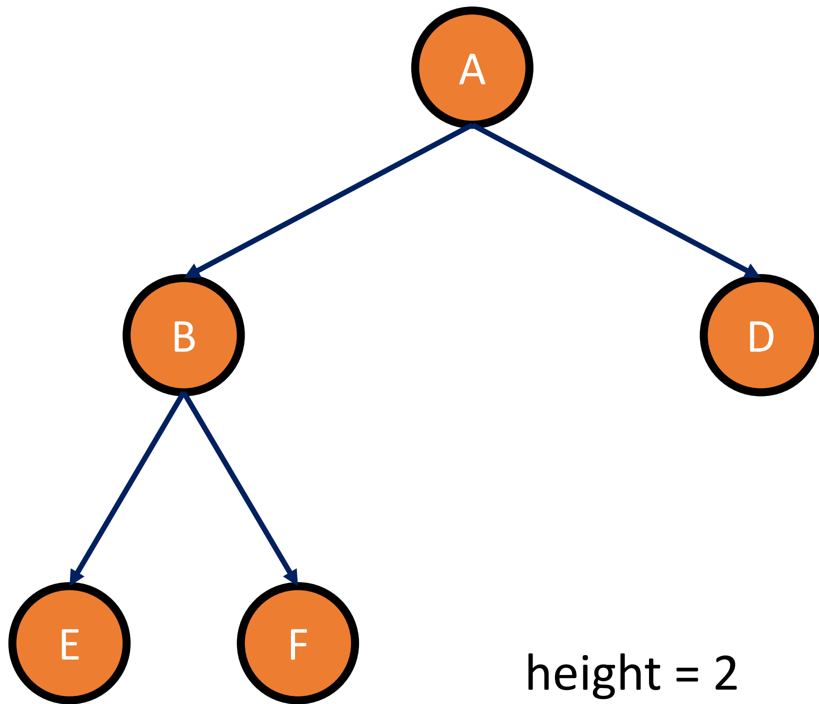
Find Minimum & Maximum in BST

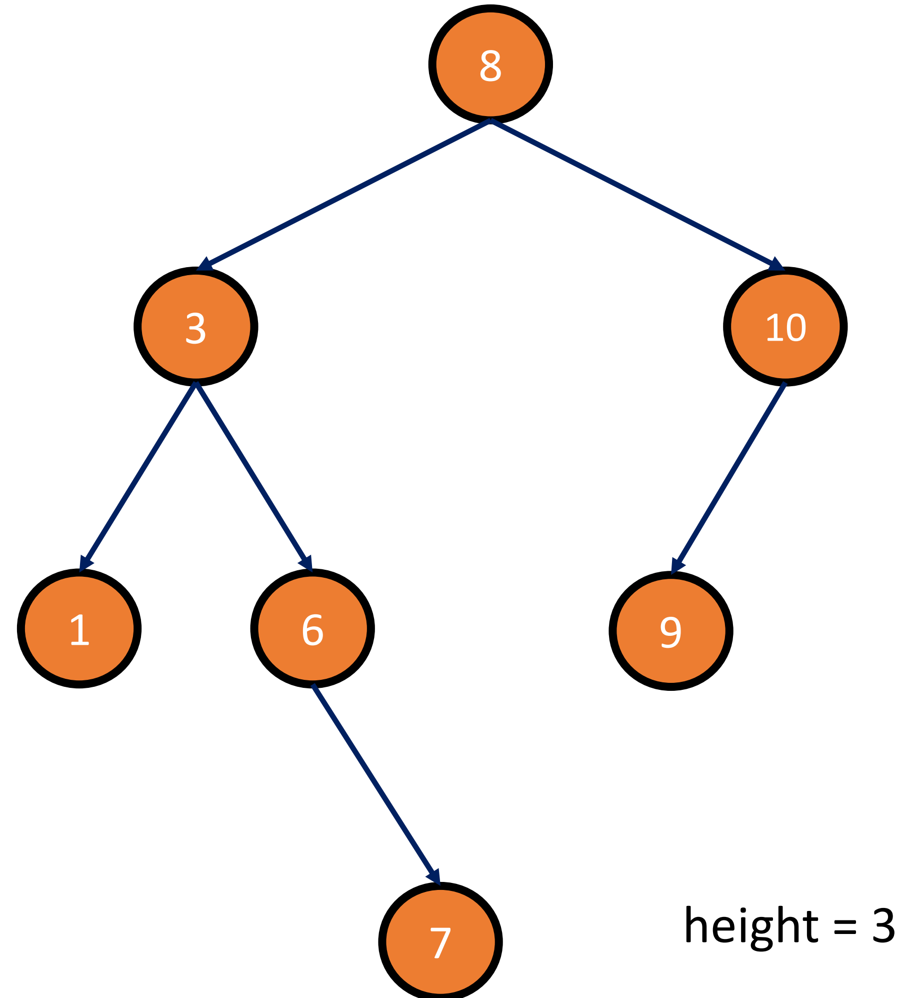
```
t_node* FindMin(t_node *node){
    // base case: Reached to the left-most nodes
    if(_________________)
        return node;
    // recursive case: Traverse to the left node
    else
        return FindMin(_____________);
}

t_node* FindMax(t_node *node){
    // base case: Reached to the right-most nodes
    if(_________________)
        return node;
    // recursive case: Traverse to the right node
    else
        return FindMax(_____________);
}
```



Calculate Height of Tree

- Height of a node = length of the longest path from the node to a leaf
- Height of tree = height of root





Calculate Height of Tree

```
getHeight(node 10) = max(getHeight(node 9), -1) + 1
                   = getHeight(node 9) + 1
```

```
int getHeight(t_node *node)
```

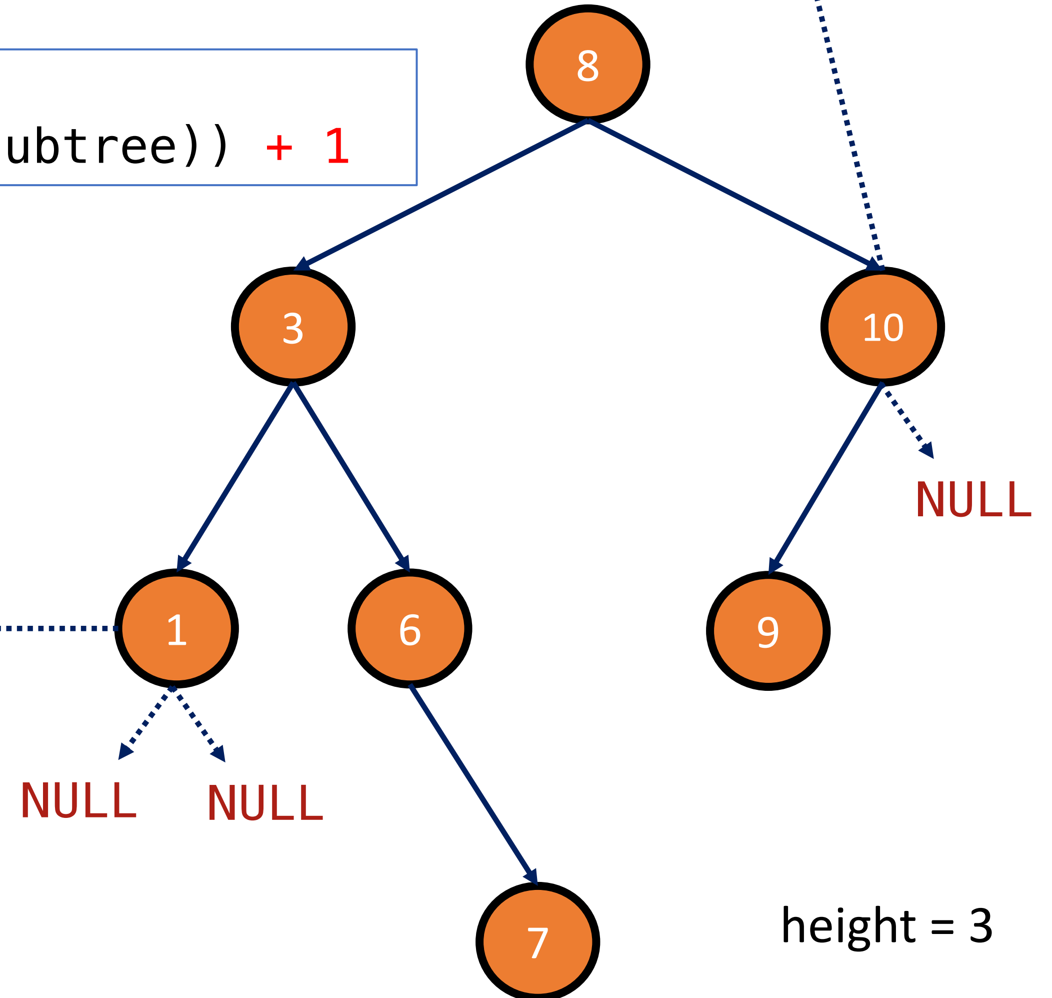
recursive case

```
getHeight(node) =
max(getHeight(left subtree), getHeight(right subtree)) + 1
```

base case

```
if(node == NULL)
   return -1;
```

```
getHeight(node 1) = max(-1, -1) + 1 = 0
```



Calculate Height of Tree

```
int getHeight(t_node *node){
    int lh, rh;
    // base case: Reached to NULL
    if(node == NULL)
        return -1;
    // recursive case: Calculate the height of the left-subtree and
      the right-subtree, and take the bigger one.
    else{
        lh = _______________________;
        rh = _______________________;
        if(lh>rh)
            return lh + 1;
        else
            return rh + 1;
    }
}
```

Delete a Tree

- Free every single node in a tree

→Traverse the tree (and delete each node)

→But which order?

- Breadth-First Search (level order)

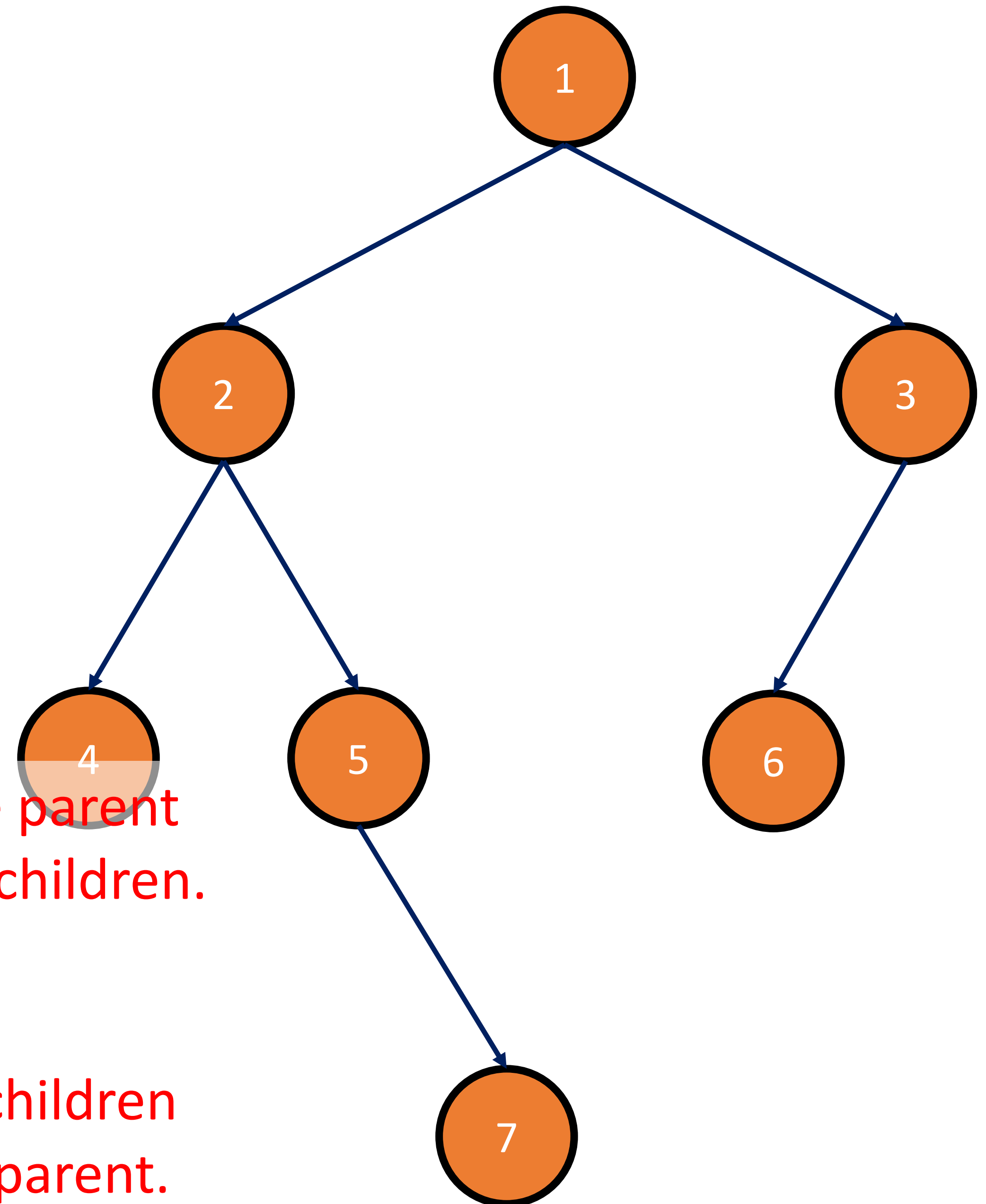
  1 2 3 4 5 6 7

- Depth-First Search

1. Pre-order: root, left, right
   1 2 4 5 7 3 6
2. In-order: left, root, right
   4 2 5 7 1 6 3
3. Post-order: left, right, root
   4 7 5 2 6 3 1

Delete the parent before its children.

Delete all children then their parent.



postorder & DeleteTree

```c
void postorder(t_node *node)
{
    // Base case
    if(node ==NULL)
        return;
    // Recursive case
    else{
        postorder(node->left);
        postorder(node->right);
        printf("%d ", node->data);
    }
}
```

```c
void DeleteTree(t_node *node)
{
    // Base case
    if(node ==NULL)
        return;
    // Recursive case
    else{
        DeleteTree(node->left);
        DeleteTree(node->left);
        __________;
    }
}
```