

# **COSC 2436: Binary Search Trees (BST)**

# What is a BST?

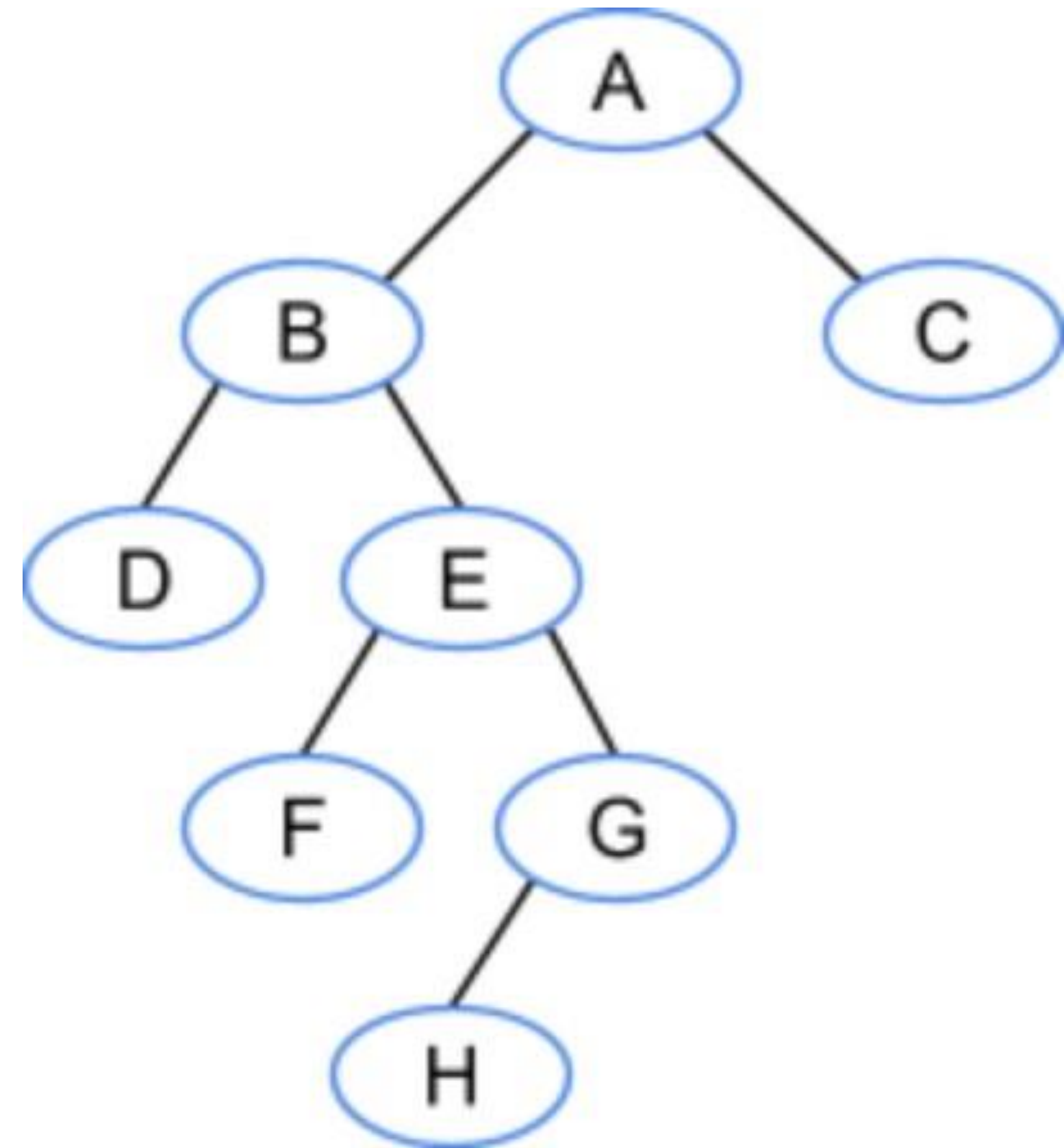
- A BST is a type of binary tree
- The left nodes' values are less than the value of the parent node
- The right nodes' values are greater than the value of the parent node
- Best Case Time Complexity:  **$O(\log n)$**
- Worst Case Time Complexity:  **$O(n)$**

# BST: Terms to know

- Edge - the link from a parent node to a child node
- Depth - the number of edges on the path from the root to the node (the root node has depth 0)
- Level - all the nodes that have the same depth
- Height - the largest depth of any node (tree with one node has height 0)

# BST: Terms to know (review)

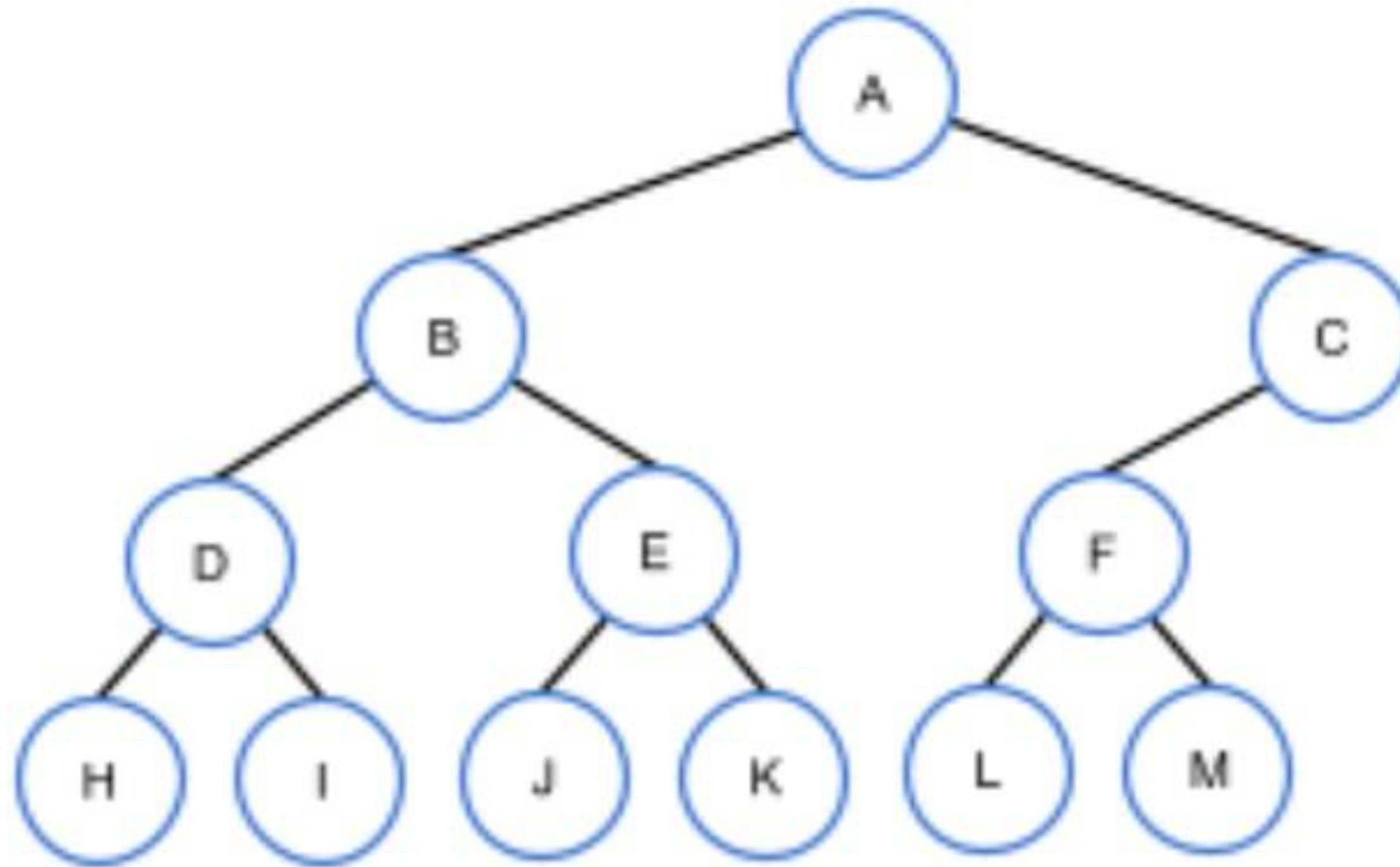
- Node *E* has depth 2
- Node *A* has depth 0
- Node *G* has depth 3
- Nodes *B* and *C* form level 1
- The tree's height is 4



# BST: full, complete, perfect

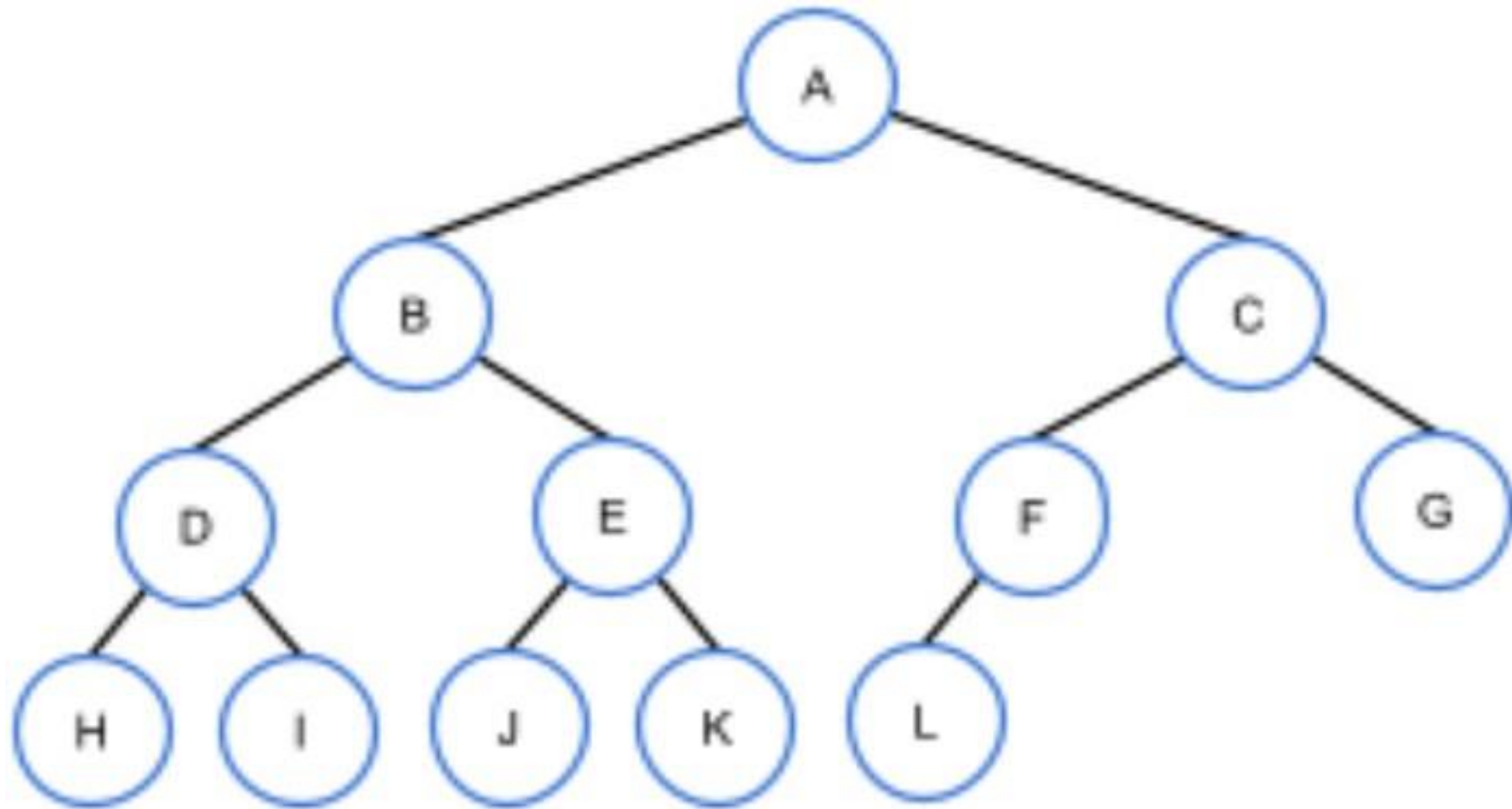
- A **full** tree means that every node contains 0 or 2 children.
- A **complete** tree means that all levels contain all possible nodes.  
\*note that for the last level must have all the nodes as far left as possible for it to be complete.
- A **perfect** tree means that all internal nodes have 2 children and all leaf nodes are at the same level

# not full, not complete, not perfect



Not full, not complete, not perfect

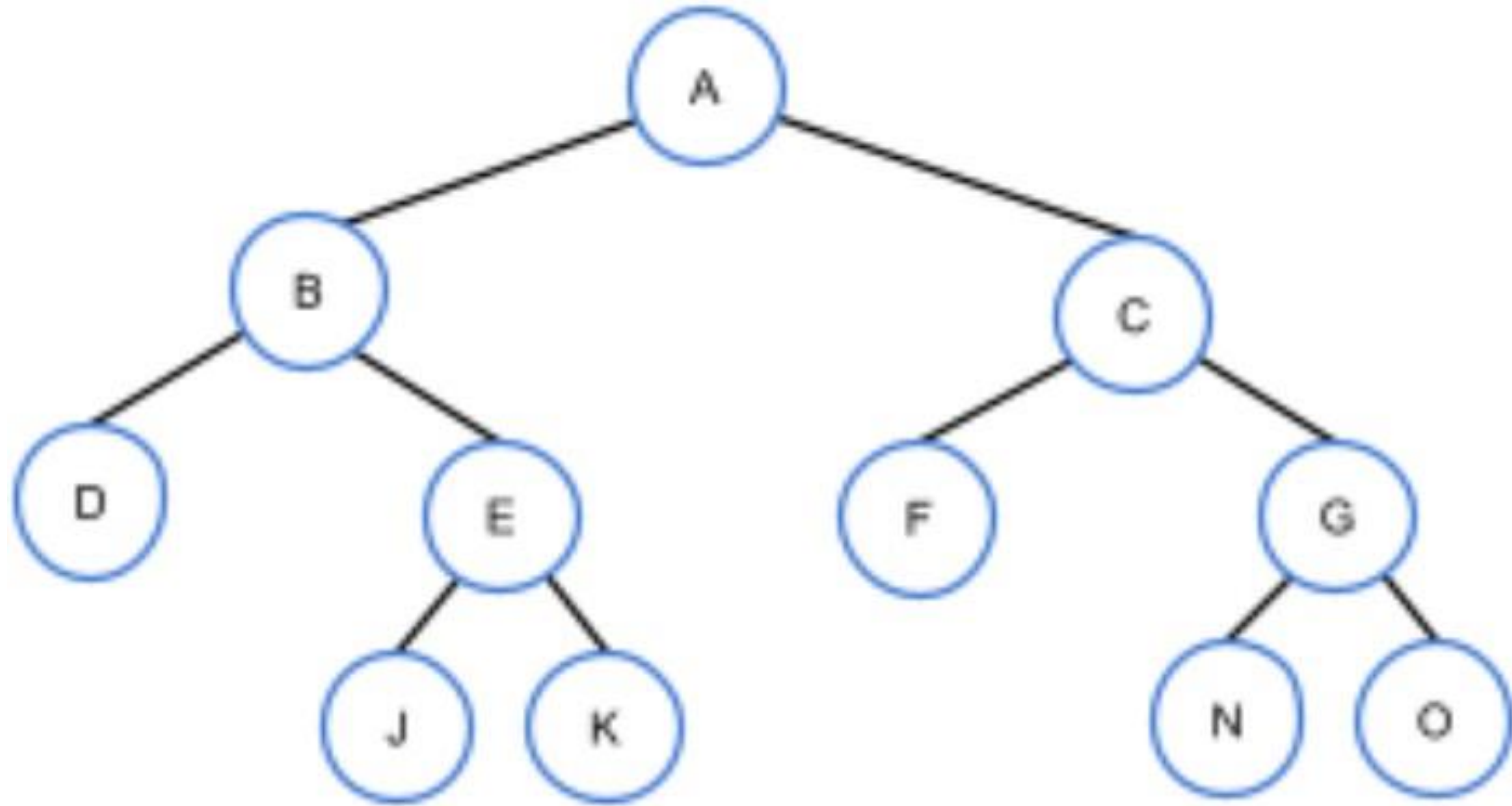
# not full, complete, not perfect



Not full, complete, not perfect



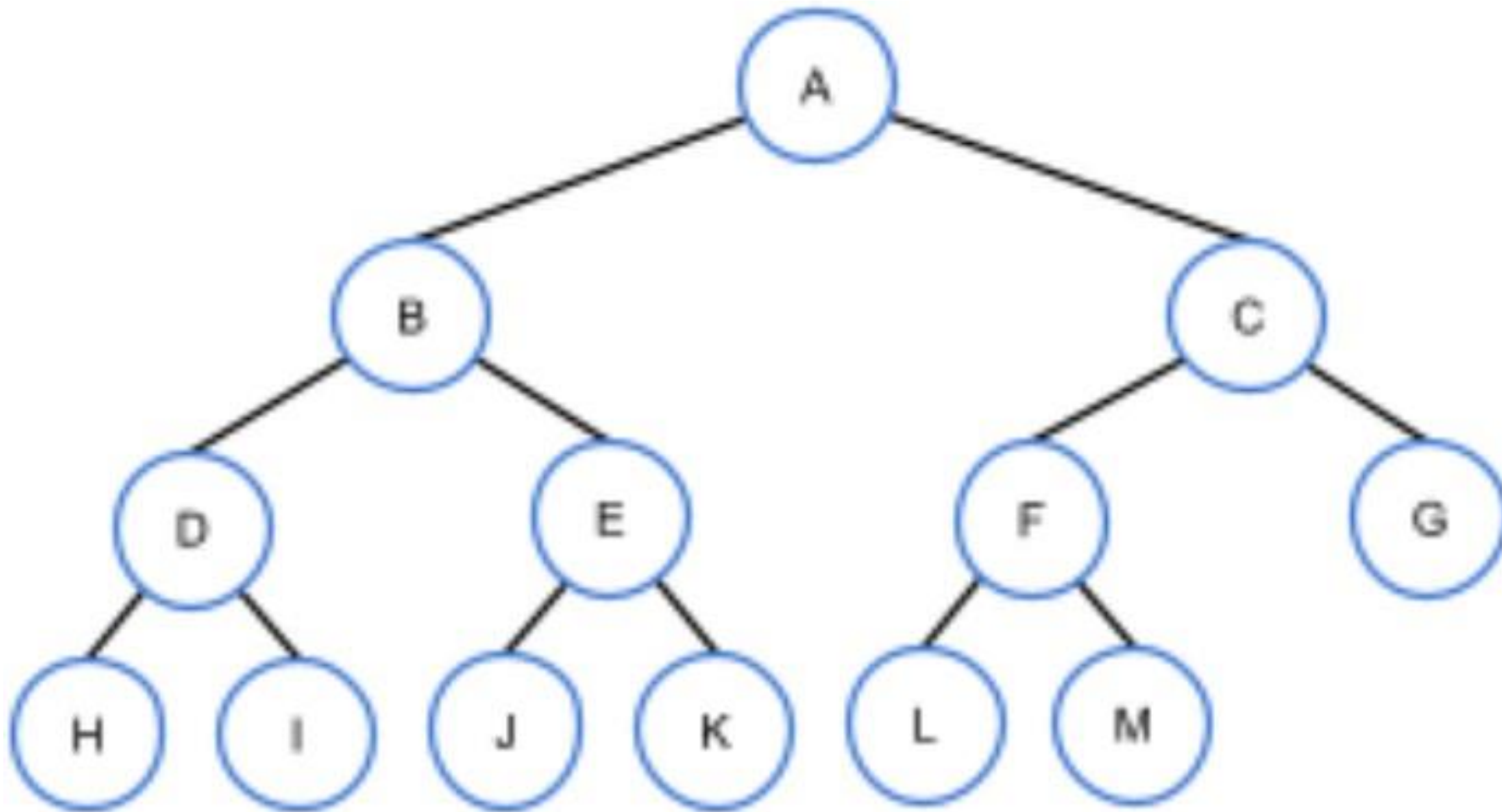
# full, not complete, not perfect



Full, not complete, not perfect

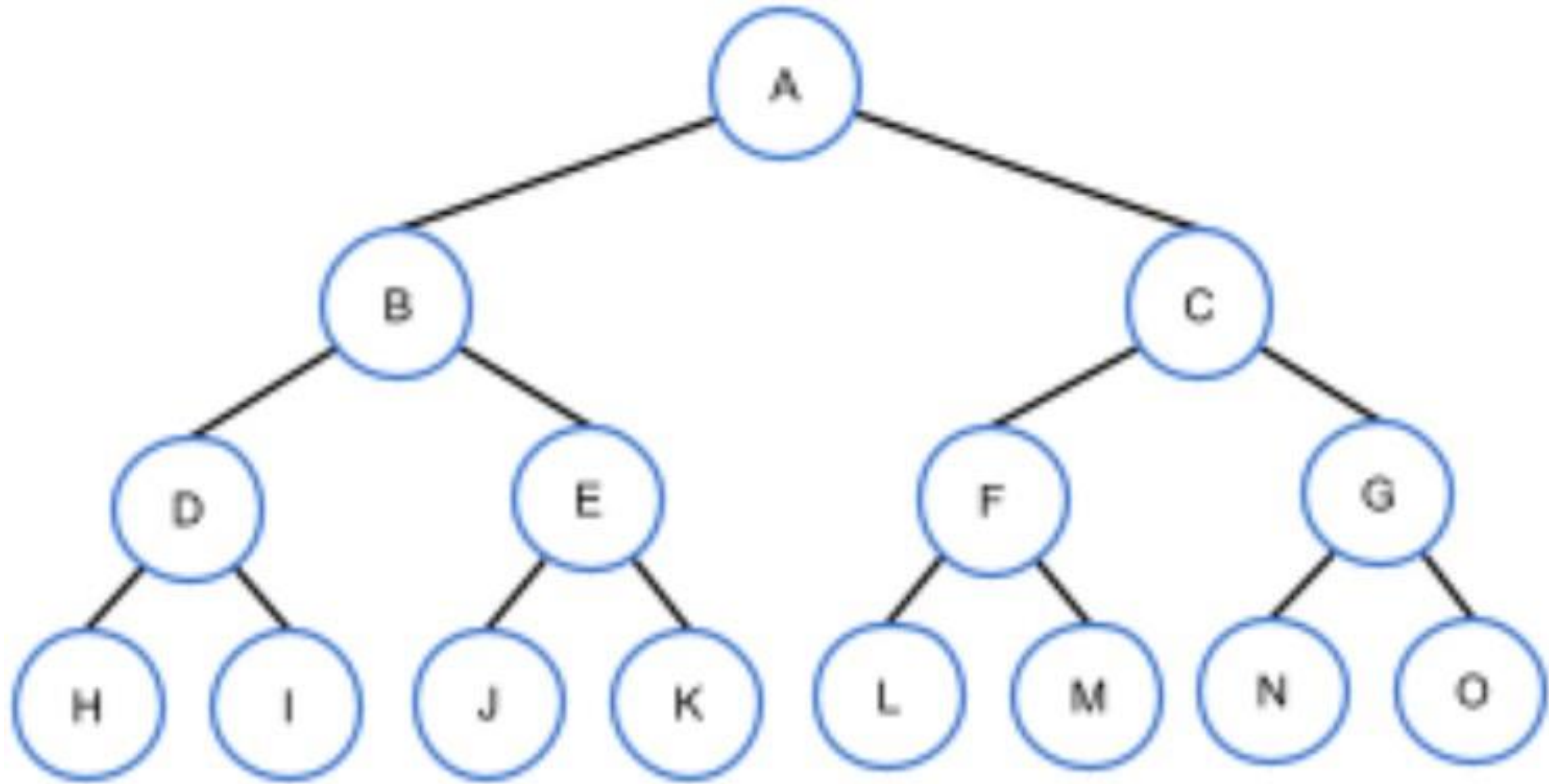


# full, complete, not perfect



Full, complete, not perfect

# full, complete, perfect



Full, complete, perfect

# BST: Insert

Insert 4

Root is NULL so insert 4 at root

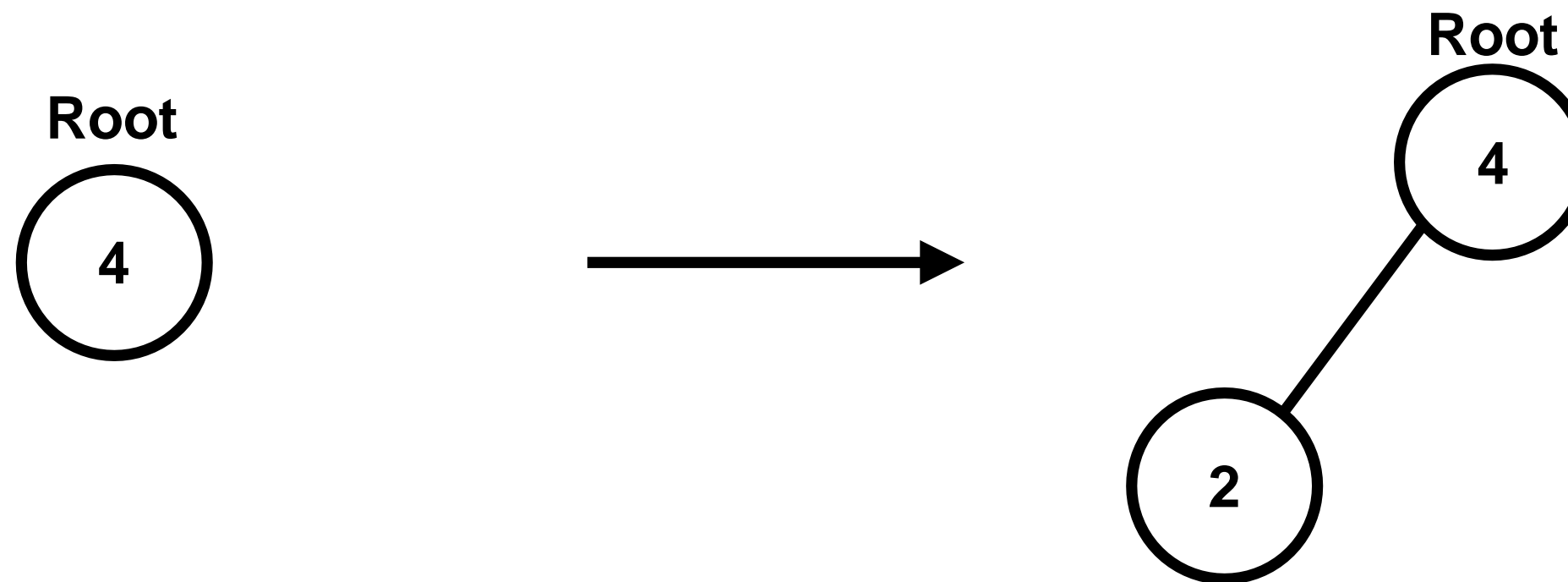


# BST: Insert

Insert 2

Root is not NULL

2 is less than 4 so go to left



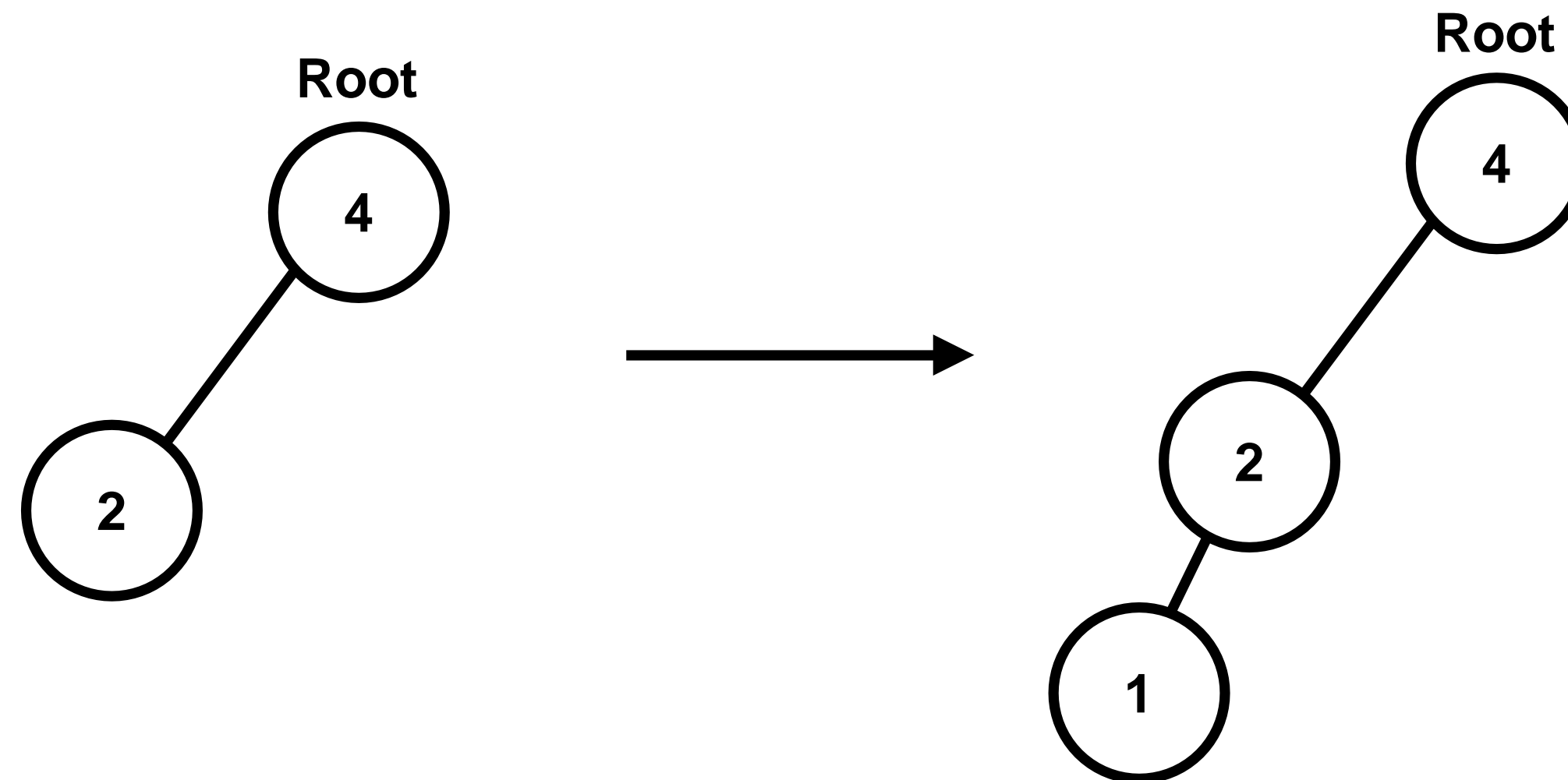
# BST: Insert

Insert 1

Root is not NULL

1 is less than 4 so go to left

1 is less than 2 so go to left

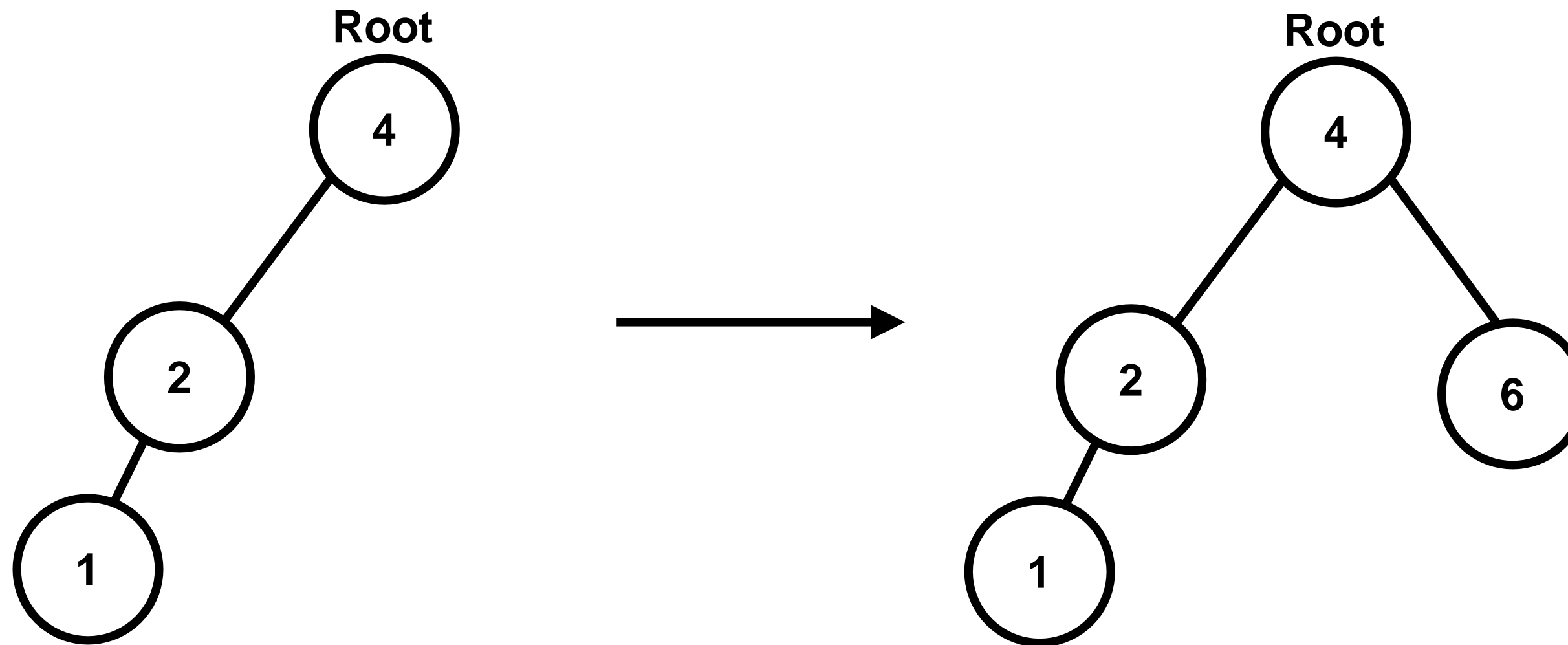


# BST: Insert

Insert 6

Root is not NULL

6 is greater than 4 so go to right



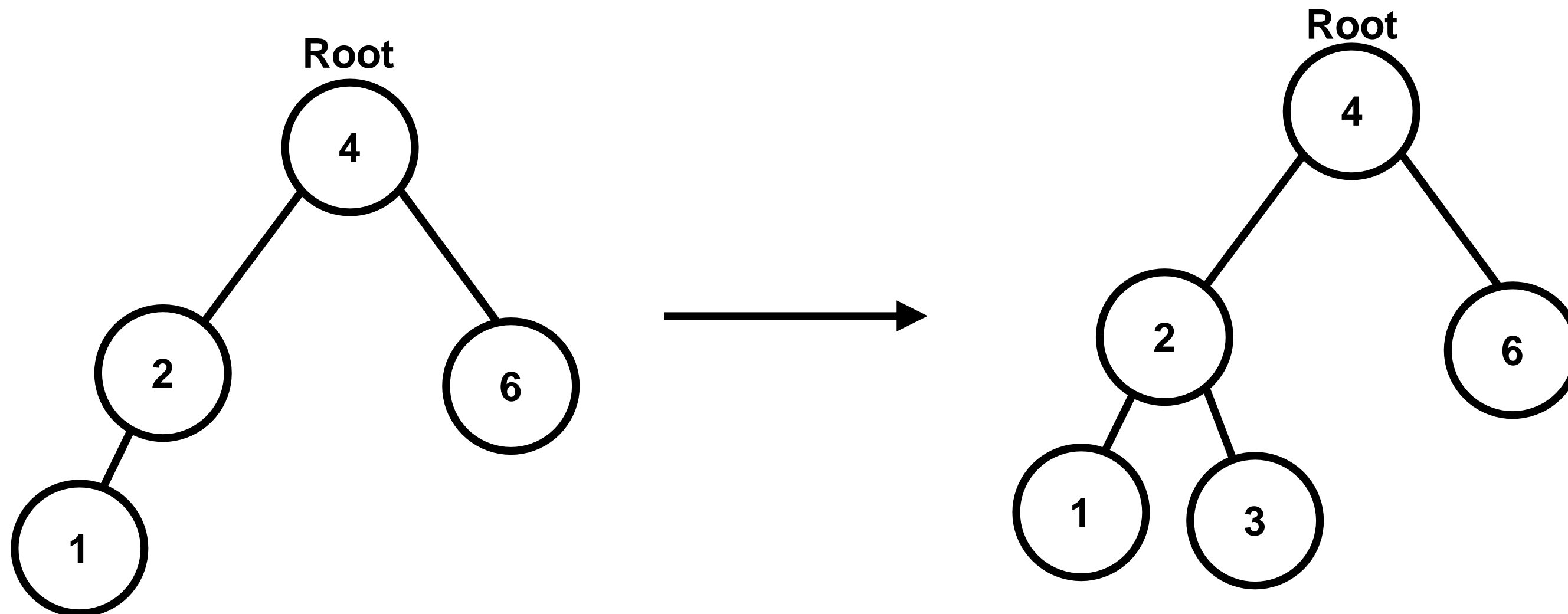
# BST: Insert

Insert 3

Root is not NULL

3 is less than 4 so go to left

3 is greater than 2 so go to right





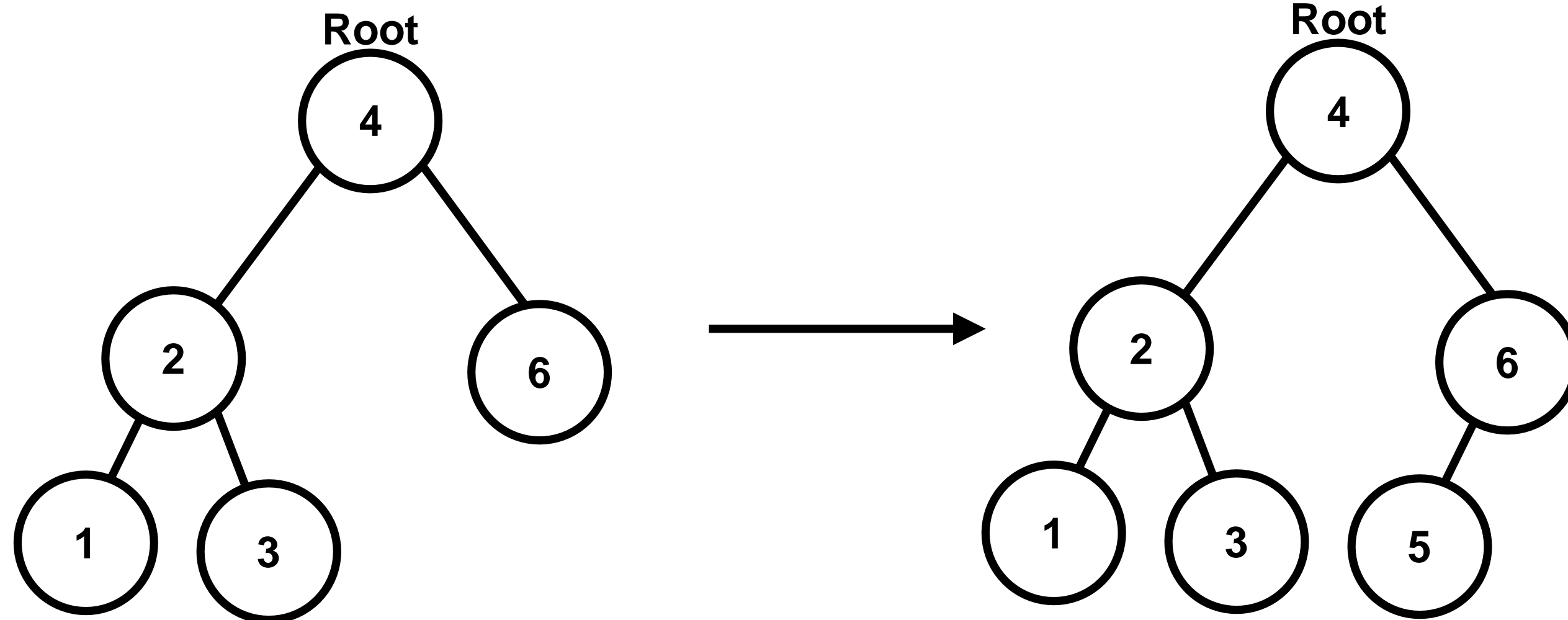
# BST: Insert

Insert 5

Root is not NULL

5 is greater than 4 so go to right

5 is less than 6 so go to left



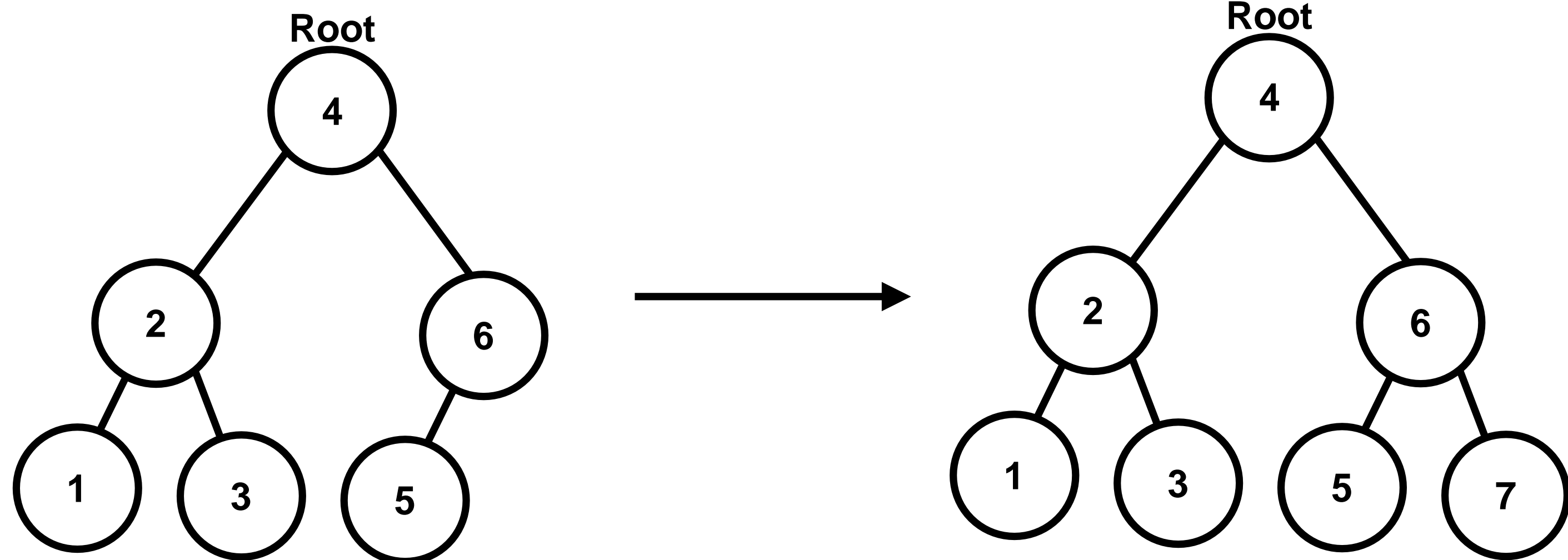
# BST: Insert

Insert 7

Root is not NULL

7 is greater than 4 so go to right

7 is greater than 6 so go to right



# Insertion Recursive

```
void insert(int d)
{
    if (root == nullptr)
    {
        root = new node(d);
        return;
    }
    insertR(root, d);
}

void insertR(node *r, int d)
{
    if (d < r->data)
    {
        if (r->left == nullptr)
            r->left = new node(d);
        else
            insertR(r->left, d);
    }
    else if (d > r->data)
    {
        if (r->right == nullptr)
            r->right = new node(d);
        else
            insertR(r->right, d);
    }
}
```

```
node *insert(node *node, int value) {
    if (node == nullptr) {
        return new node (value);
    }
    else if (value < node->data) {
        node->left = insert(node->left, value);
    }
    else if (value > node->data) {
        node->right = insert(node->right, value);
    }
    return node;
}
```

# Insertion Iterative

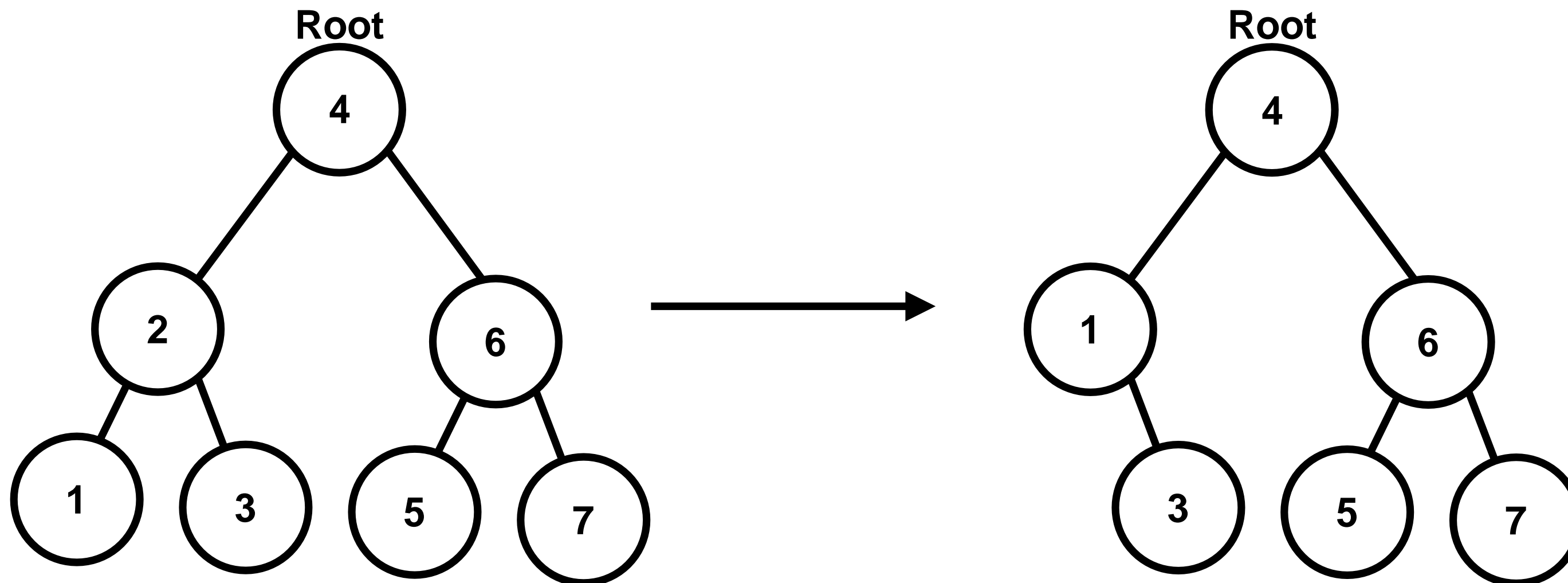
```
void insert(node*& root, int key)
{
    node *curr = root;
    node *parent = nullptr;

    if (root == nullptr)
    {
        root = new node(key);
        return;
    }
    while (curr != nullptr)
    {
        parent = curr;
        if (key < curr->data) {
            curr = curr->left;
        }
        else {
            curr = curr->right;
        }
    }
    if (key < parent->data) {
        parent->left = new node(key);
    }
    else {
        parent->right = new node(key);
    }
}
```

# BST: Remove

Remove 2

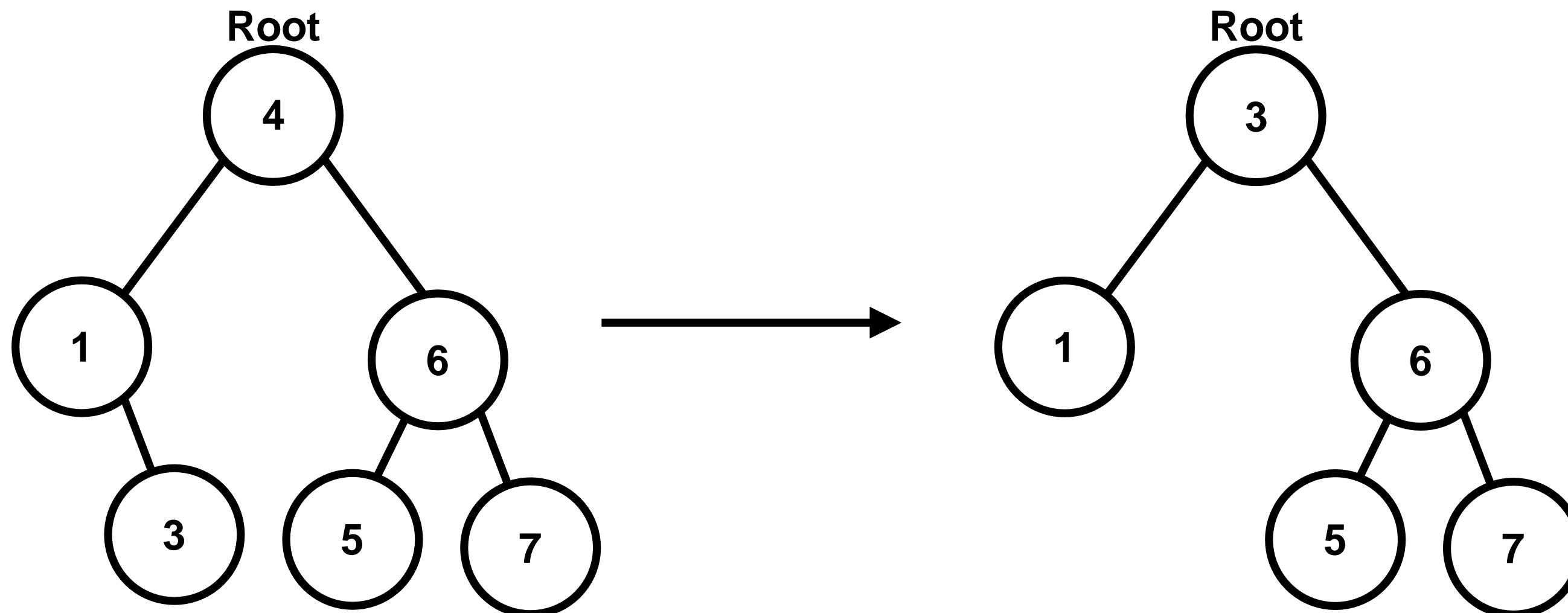
Replace 2 with in-order predecessor (which is 1)



# BST: Remove

Remove 4

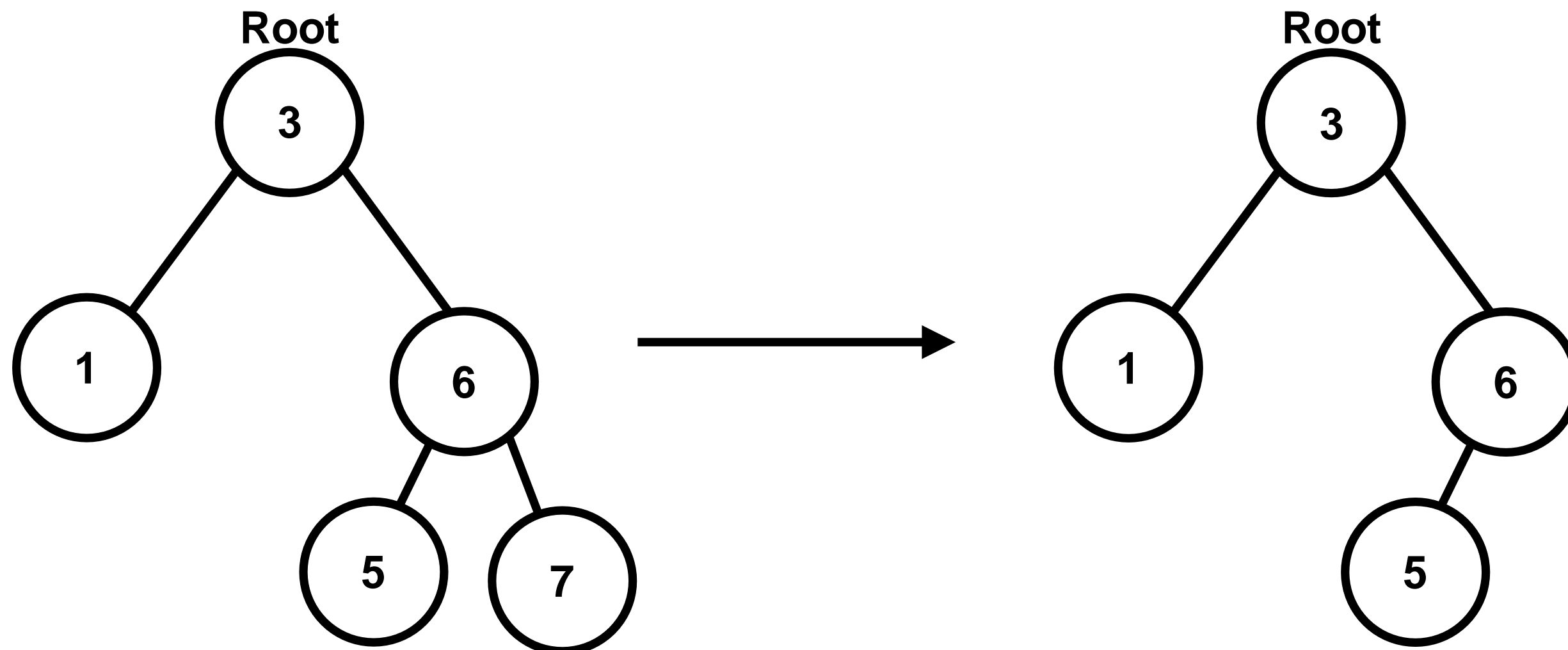
Replace 4 with in-order predecessor (which is 3)



# BST: Remove

Remove 7

7 is a leaf so nothing to replace

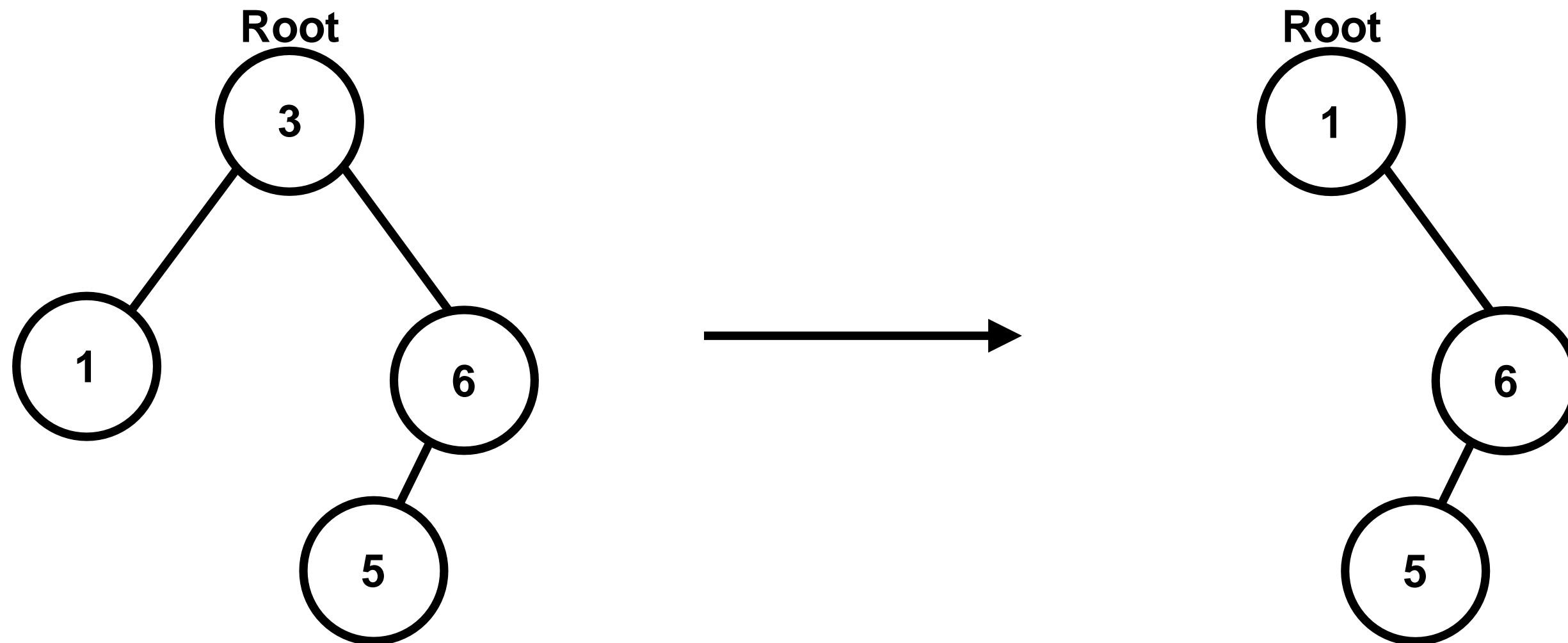




# BST: Remove

Remove 3

Replace 3 with in-order predecessor (which is 1)

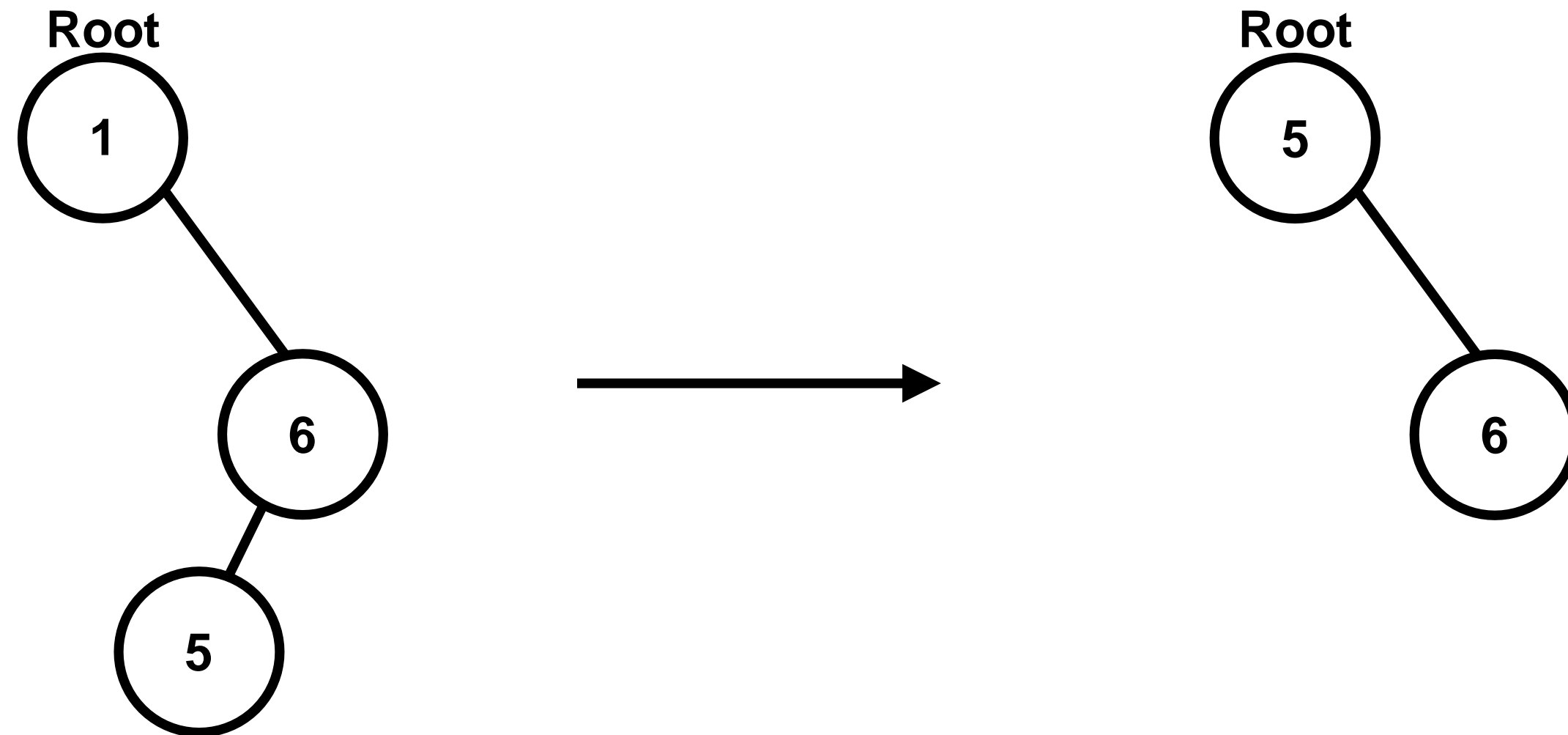


# BST: Remove

**Remove 1**

**Replace 1 with in-order predecessor**

**Since 1 has no in-order predecessor, replace 1 with in-order successor (which is 5)**

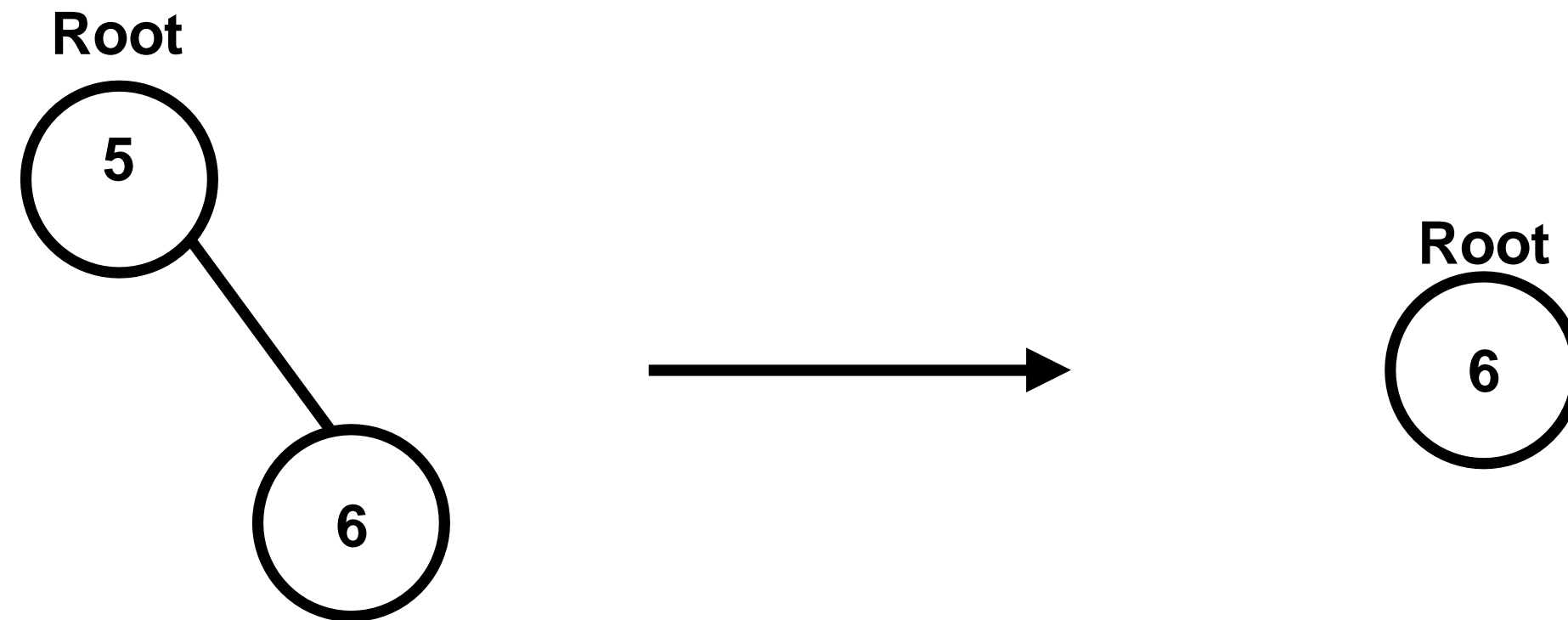


# BST: Remove

**Remove 5**

**Replace 5 with in-order predecessor**

**Since 5 has no in-order predecessor, replace 5 with in-order successor (which is 6)**



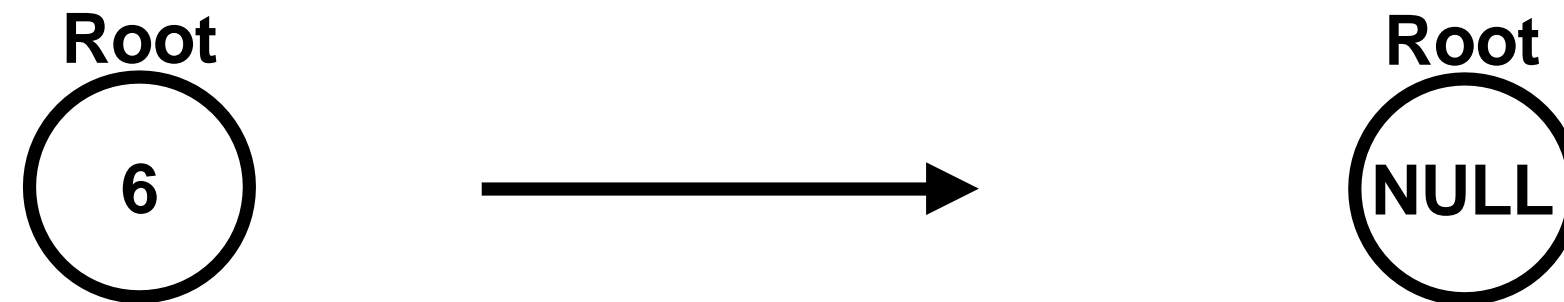
# BST: Remove

**Remove 6**

**Replace 6 with in-order predecessor**

**Since 6 has no in-order predecessor, replace 6 with in-order successor**

**Since 6 has no in-order predecessor set root to NULL**



# Deletion Recursion

```
node *predecessor(node *cur)
{
    if (cur == nullptr)
        return nullptr;
    cur = cur->left;
    while (cur->right != nullptr)
        cur = cur->right;

    return cur;
}
```

```
node *successor(node *cur)
{
    if (cur == nullptr)
        return nullptr;
    cur = cur->right;
    while (cur->left != nullptr)
        cur = cur->left;
    return cur;
}
```

```
node *remove(node *r, int d)
{
    if (r == nullptr)
        return r;
    else if (r->data < d)
        r->right = remove(r->right, d);
    else if (r->data > d)
        r->left = remove(r->left, d);
    else
    {
        if (r->left == nullptr && r->right == nullptr)
        {
            delete r;
            r = nullptr;
        }
        else if (r->left == nullptr)
        {
            node *temp = r;
            r = r->right;
            delete temp;
        }
        else if (r->right == nullptr)
        {
            node *temp = r;
            r = r->left;
            delete temp;
        }
        else
        {
            node *temp = predecessor(r);
            r->data = temp->data;
            r->left = remove(r->left, temp->data);
        }
    }
    root = r;
    return r;
}
```

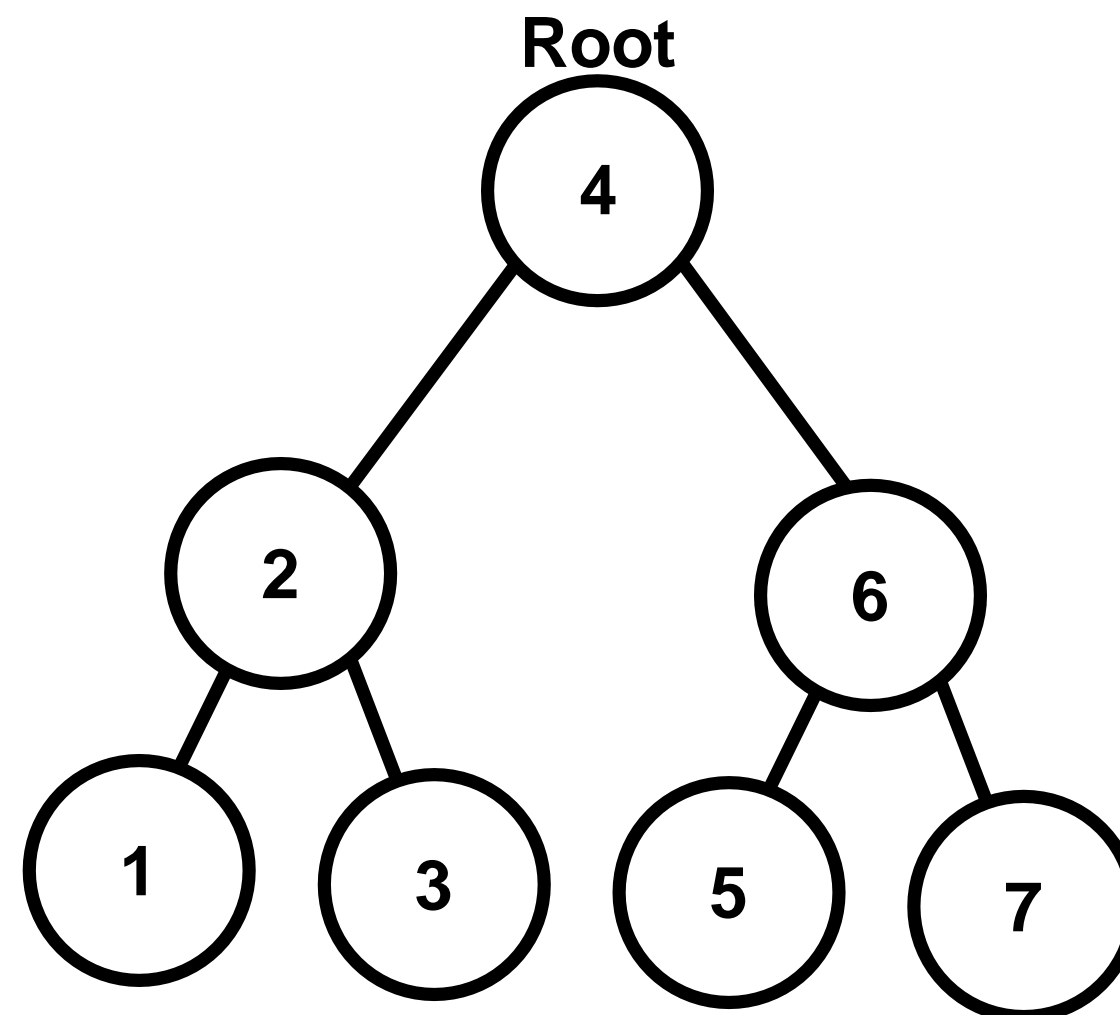
# BST: Level Order

Level Order:

4

2 6

1 3 5 7



# BST: Level Order

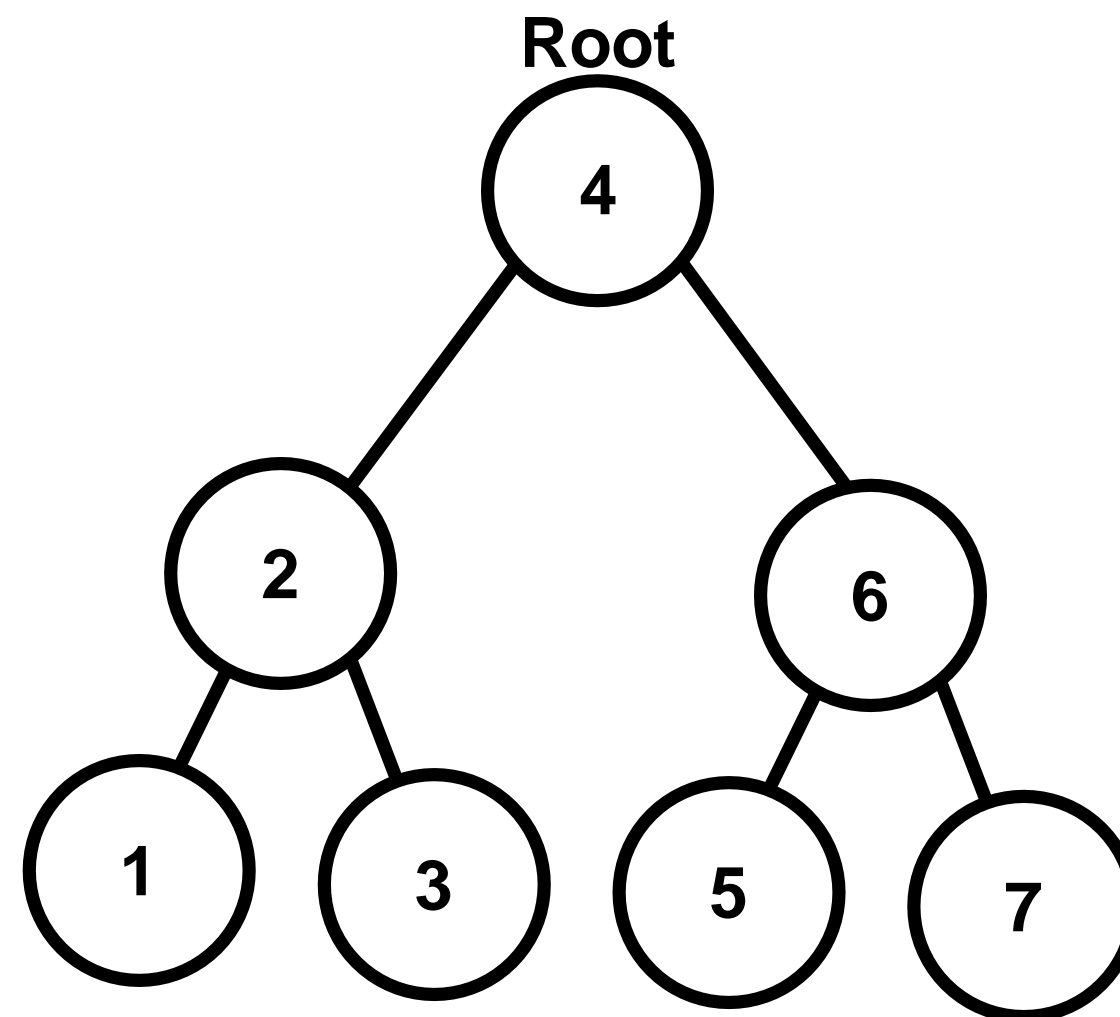
```
void levelorder(node *n){
    if(n == nullptr){
        cout << "BST is empty" << endl;
        return;
    }
    queue<node*> q;
    q.push(n);
    while(!q.empty()){
        int levelSize = q.size();
        for(int i = 0; i < levelSize; i++){
            node *cu = q.front();
            q.pop();
            cout << cu->value << " ";
            if(cu->left != nullptr)
                q.push(cu->left);
            if(cu->right != nullptr)
                q.push(cu->right);
        }
        cout << endl;
    }
}
```



# BST: Inorder

Inorder:

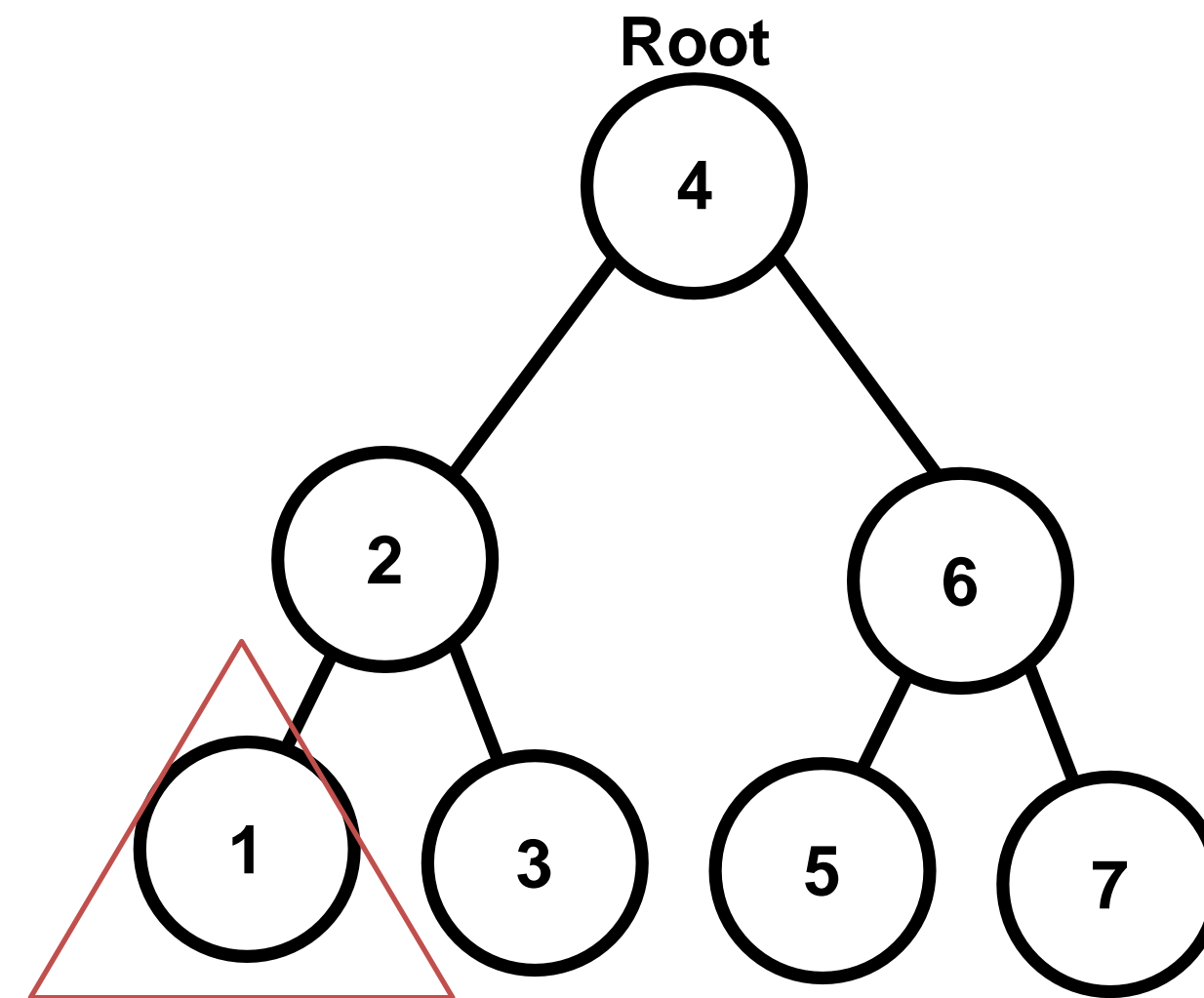
**1 2 3 4 5 6 7**



# BST: Inorder

Inorder:

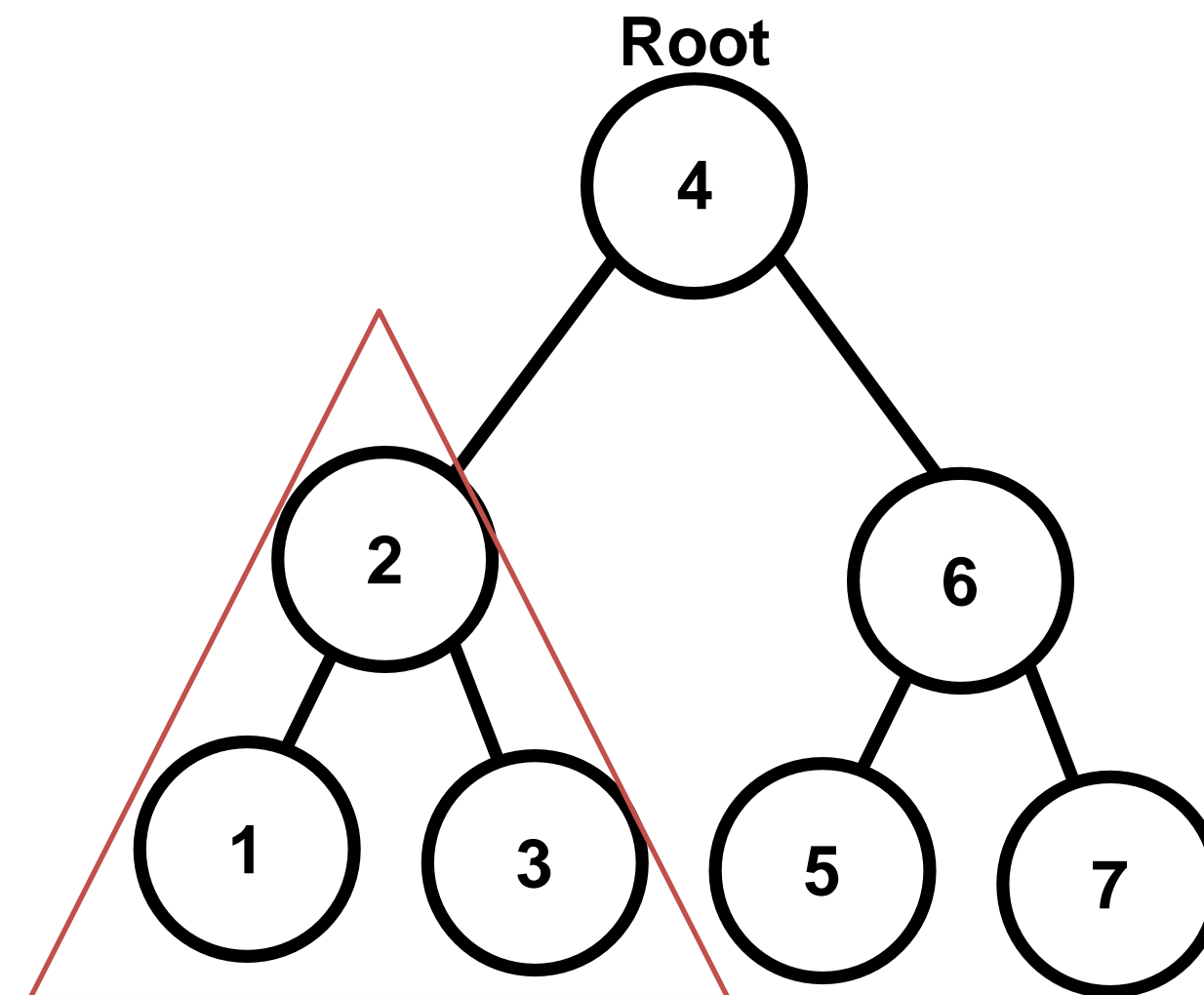
**1 2 3 4 5 6 7**



# BST: Inorder

Inorder:

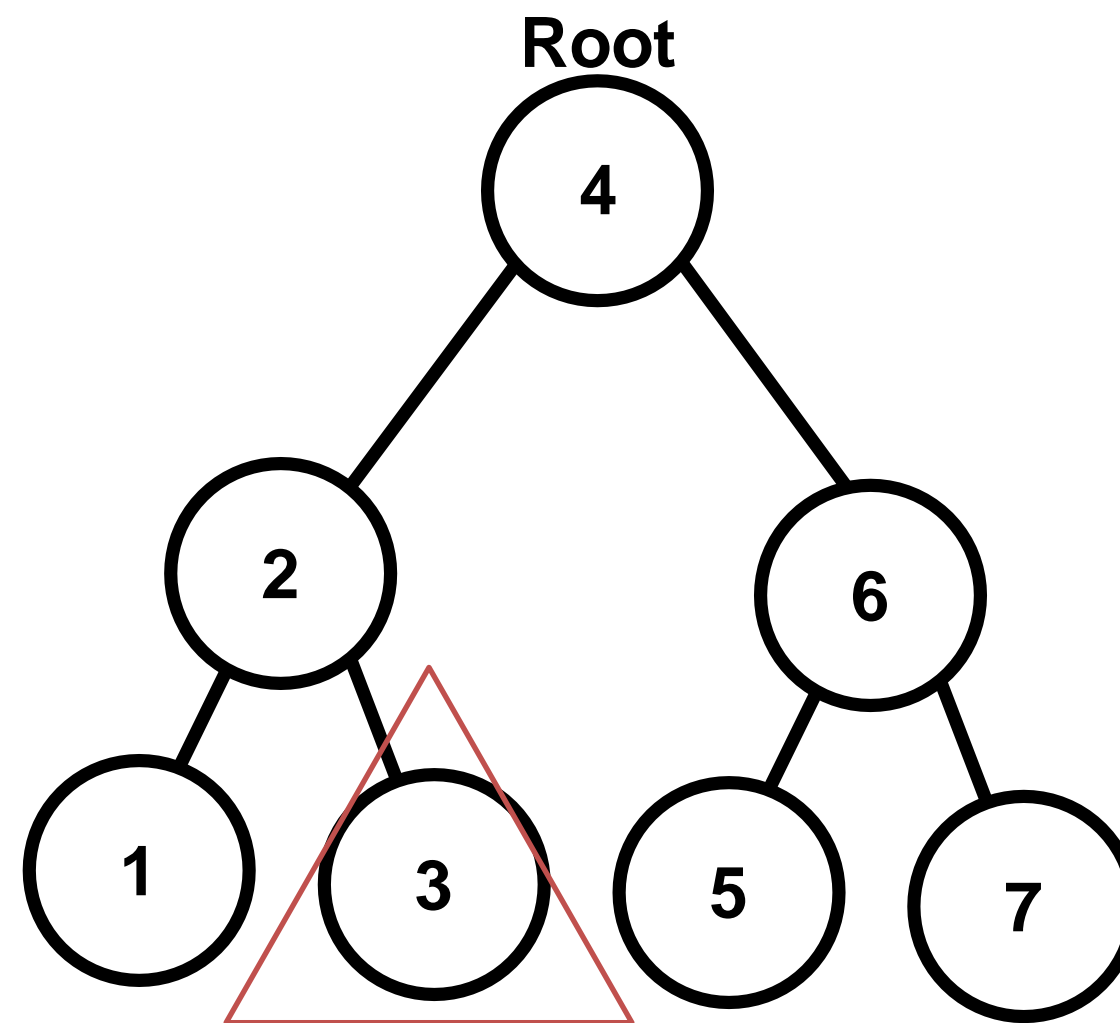
**1 2 3 4 5 6 7**



# BST: Inorder

Inorder:

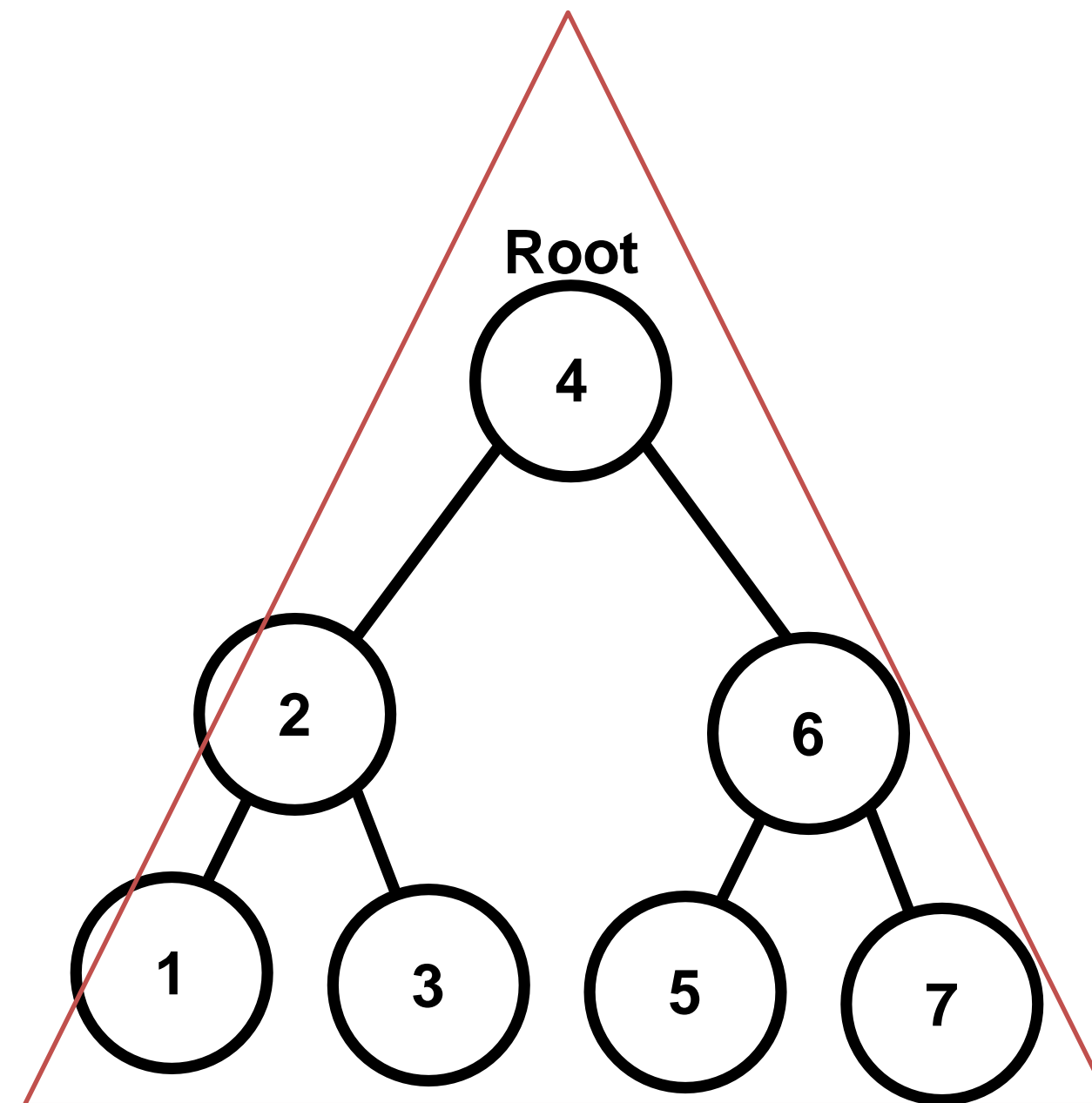
**1 2 3 4 5 6 7**



# BST: Inorder

Inorder:

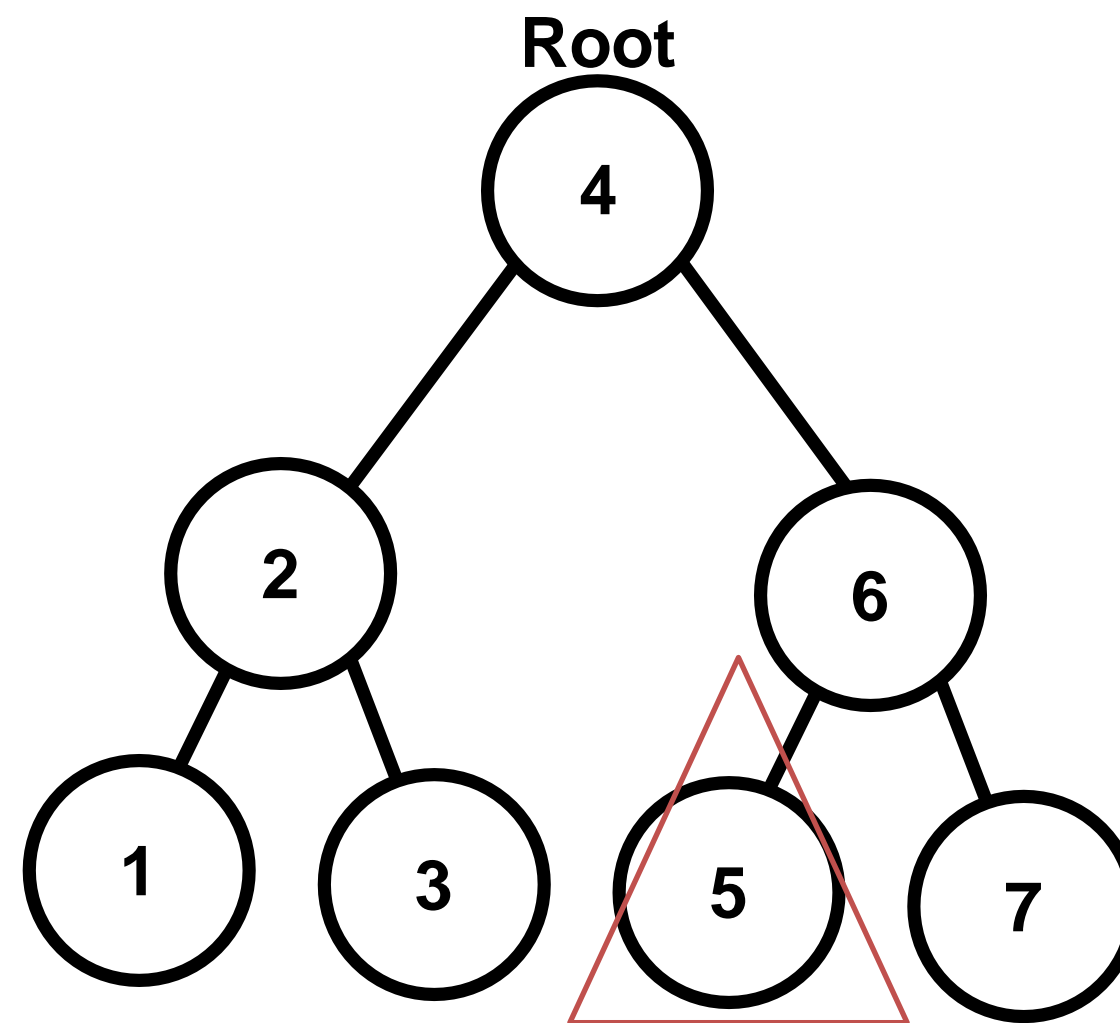
**1 2 3 4 5 6 7**



# BST: Inorder

Inorder:

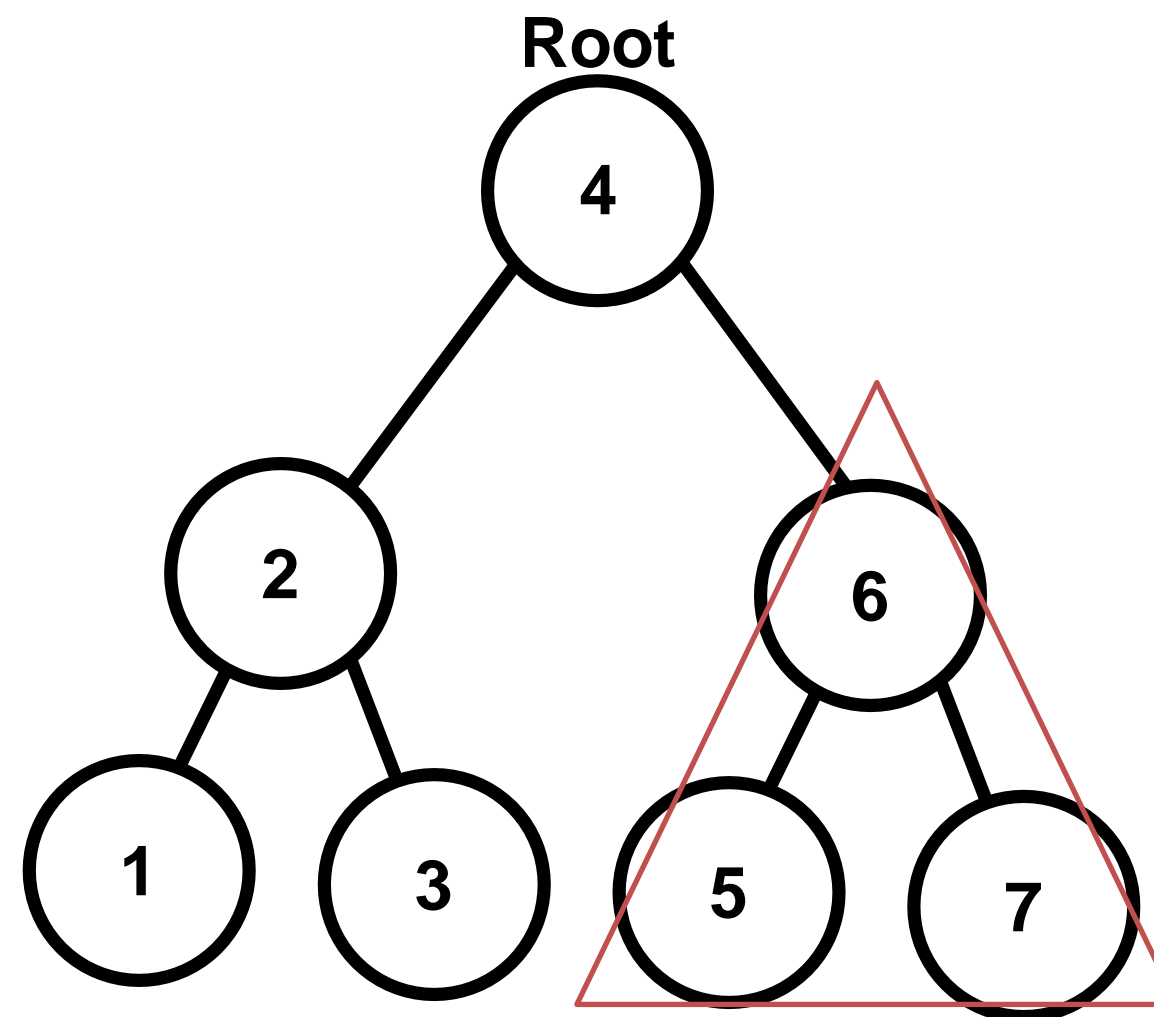
**1 2 3 4 5 6 7**



# BST: Inorder

Inorder:

**1 2 3 4 5 6 7**

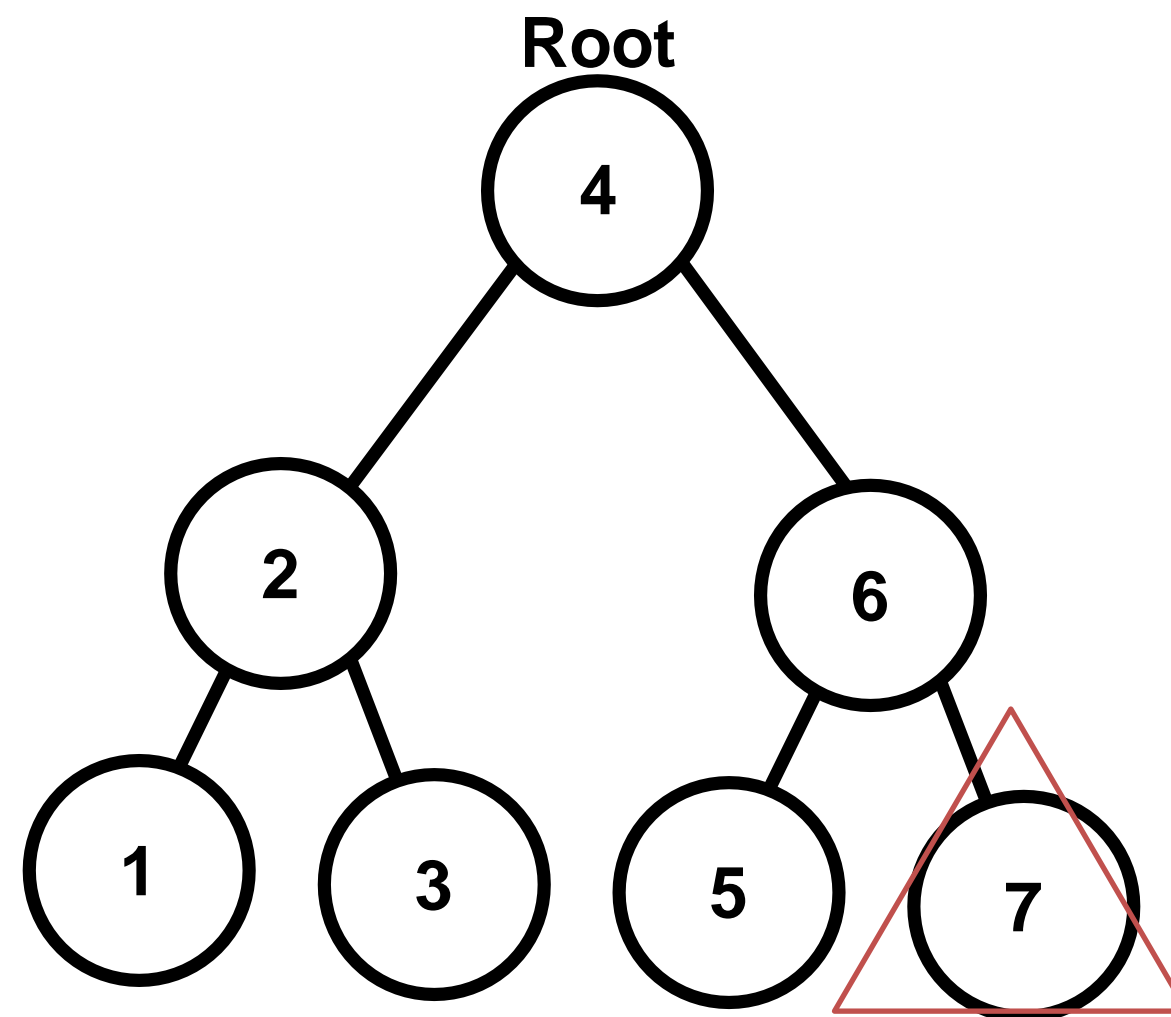




# BST: Inorder

Inorder:

**1 2 3 4 5 6 7**



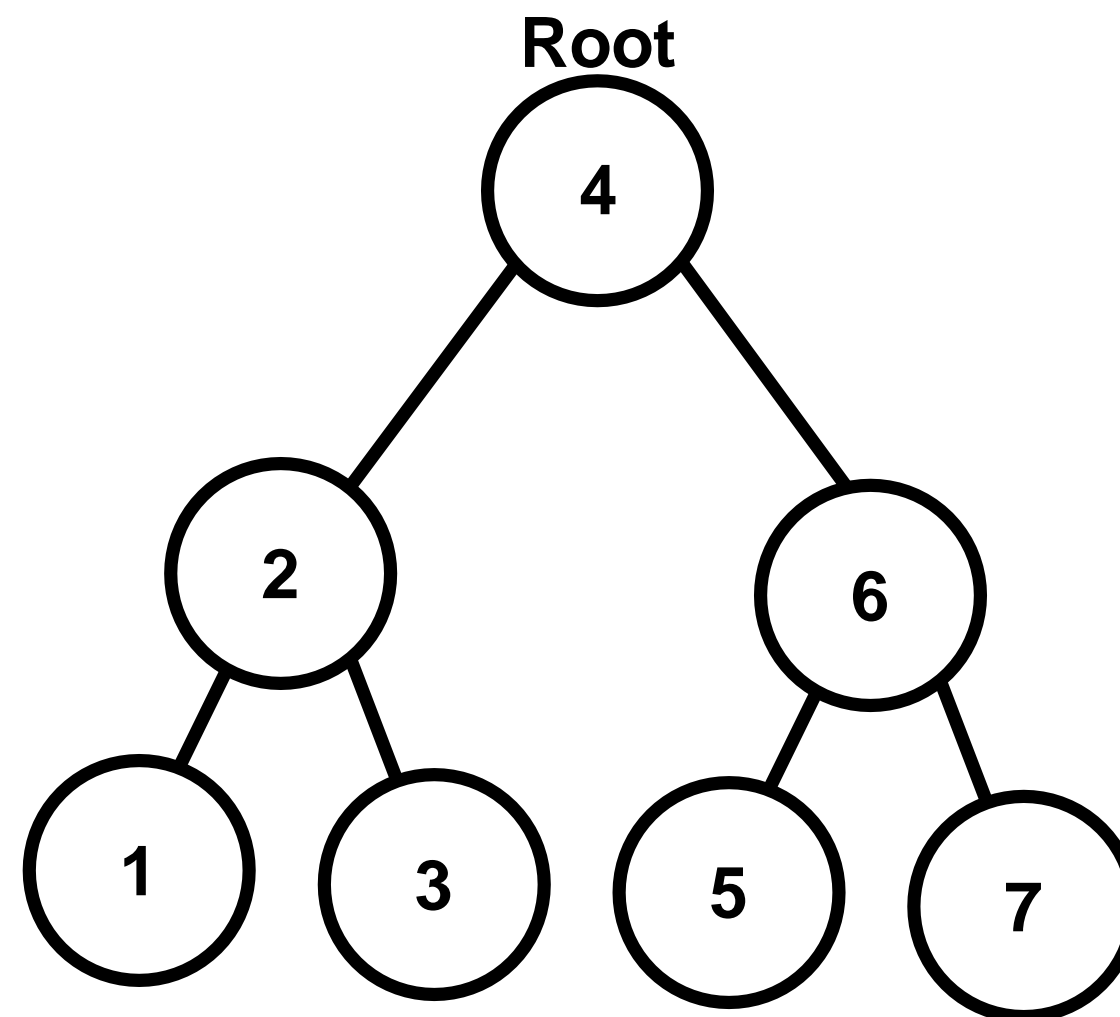
# BST: Inorder

```
void inorder(node *n){  
    if(n == nullptr)  
        return;  
    inorder(n->left);  
    cout << n->value << " ";  
    inorder(n->right);  
}
```

# BST: Preorder

**Preorder:**

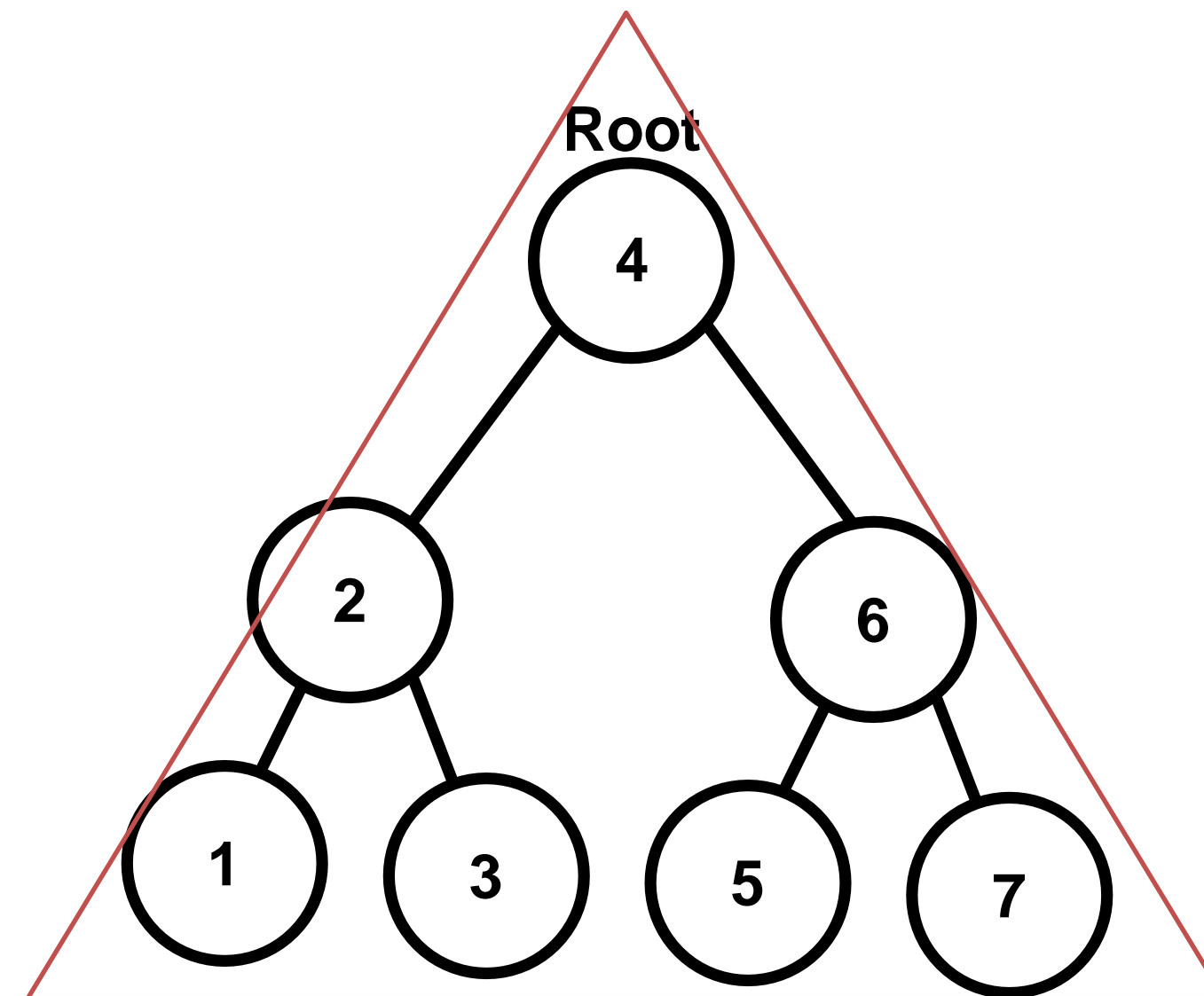
**4 2 1 3 6 5 7**



# BST: Preorder

Preorder:

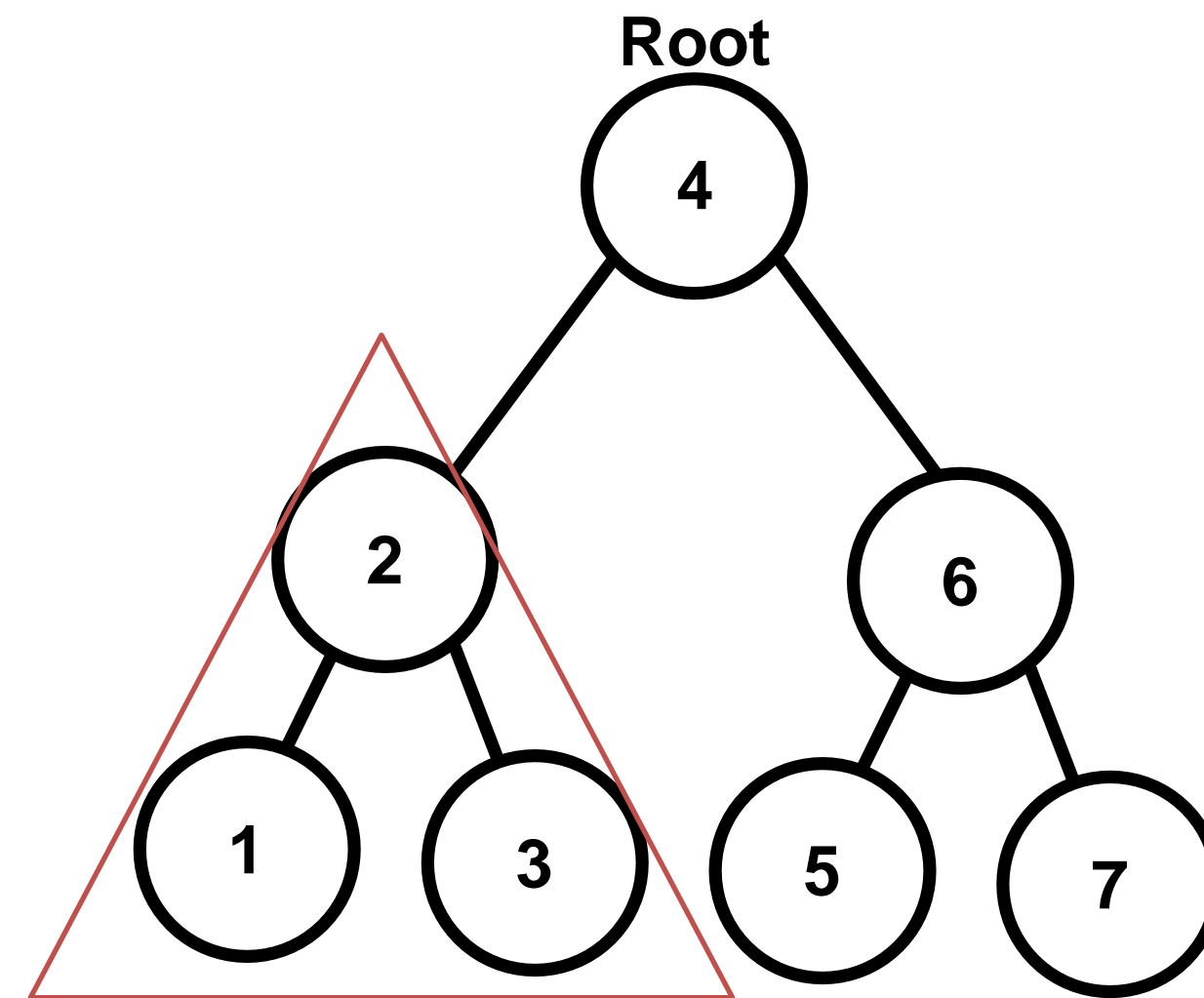
**4 2 1 3 6 5 7**



# BST: Preorder

Preorder:

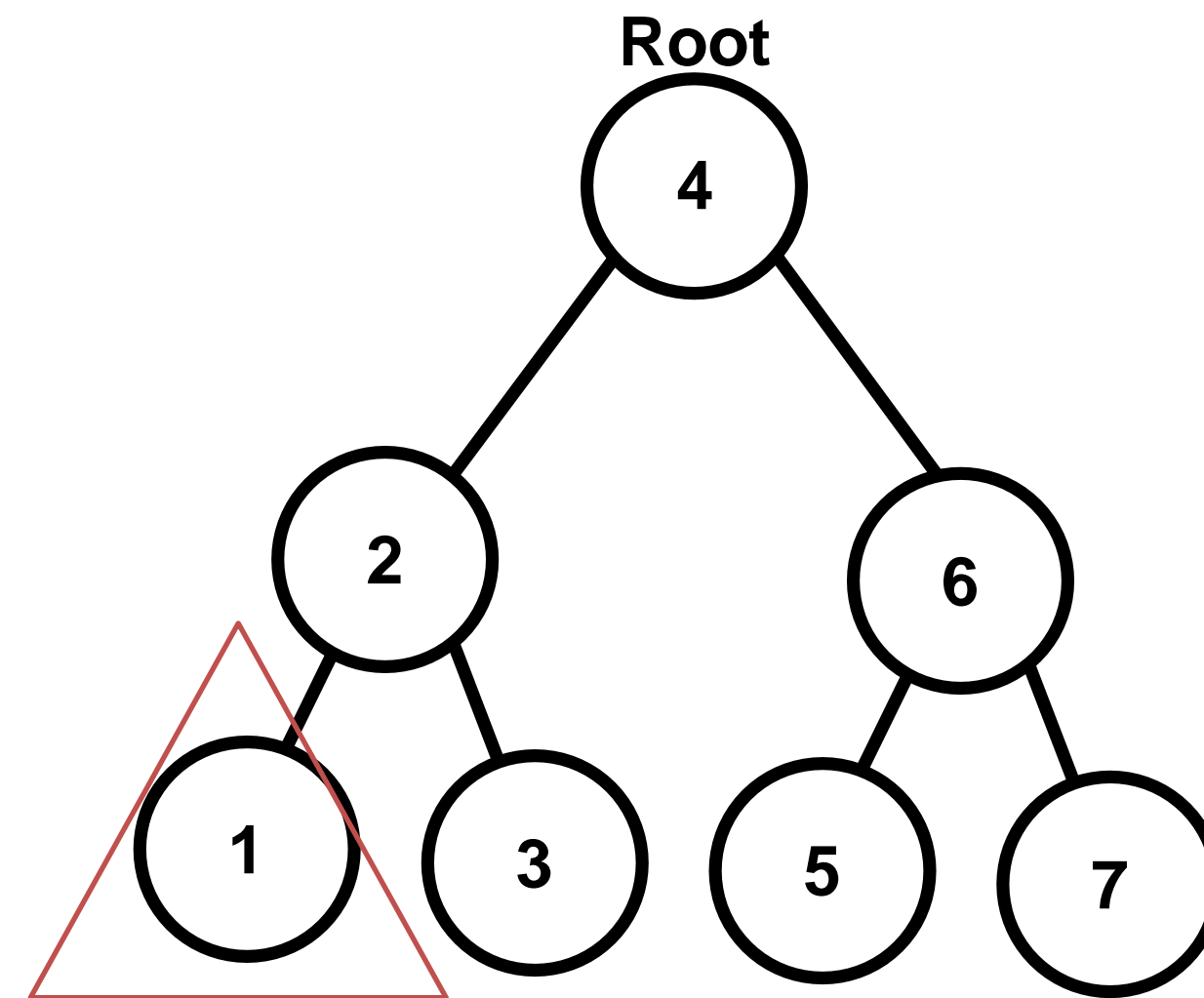
**4 2 1 3 6 5 7**



# BST: Preorder

Preorder:

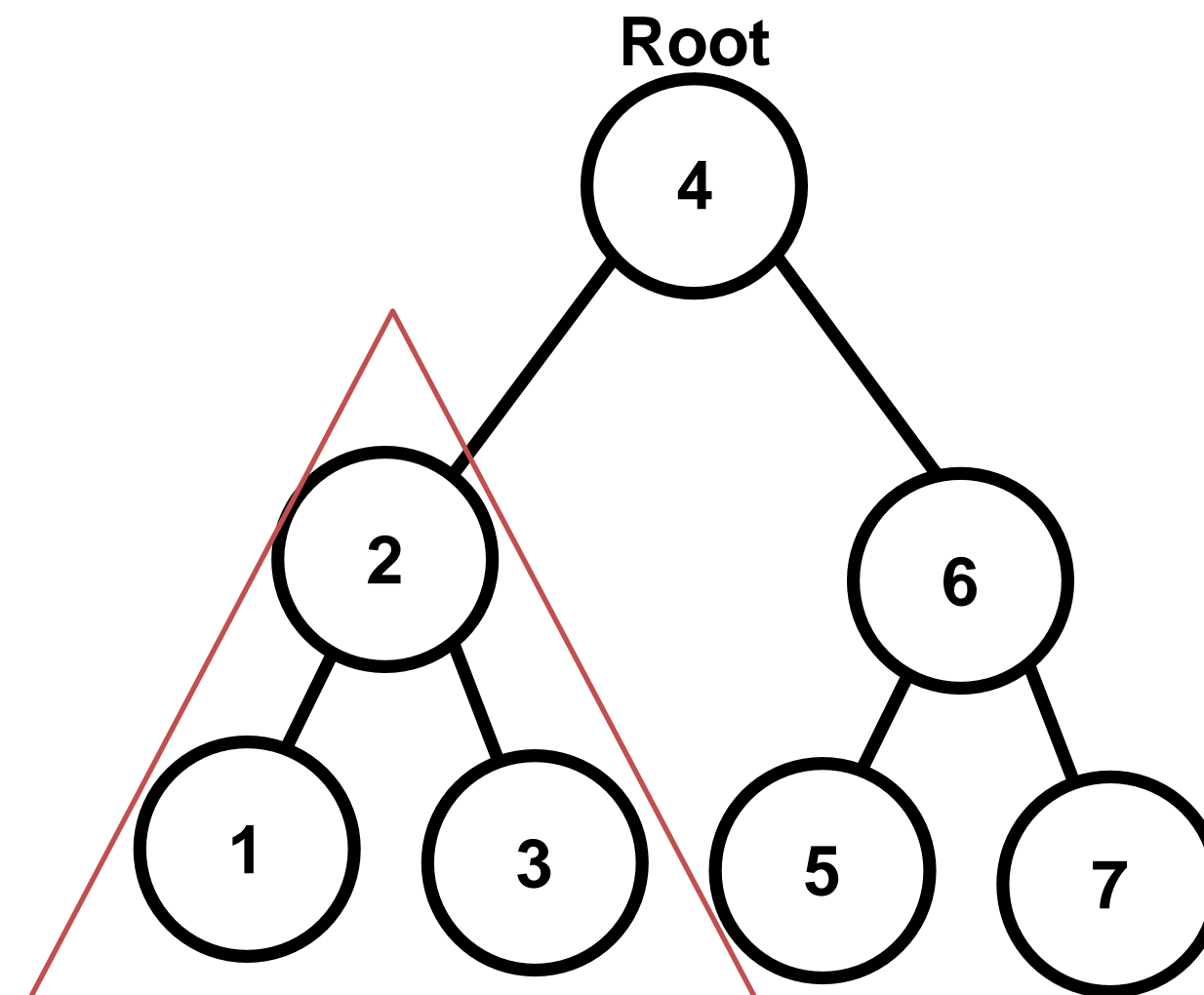
**4 2 1 3 6 5 7**



# BST: Preorder

Preorder:

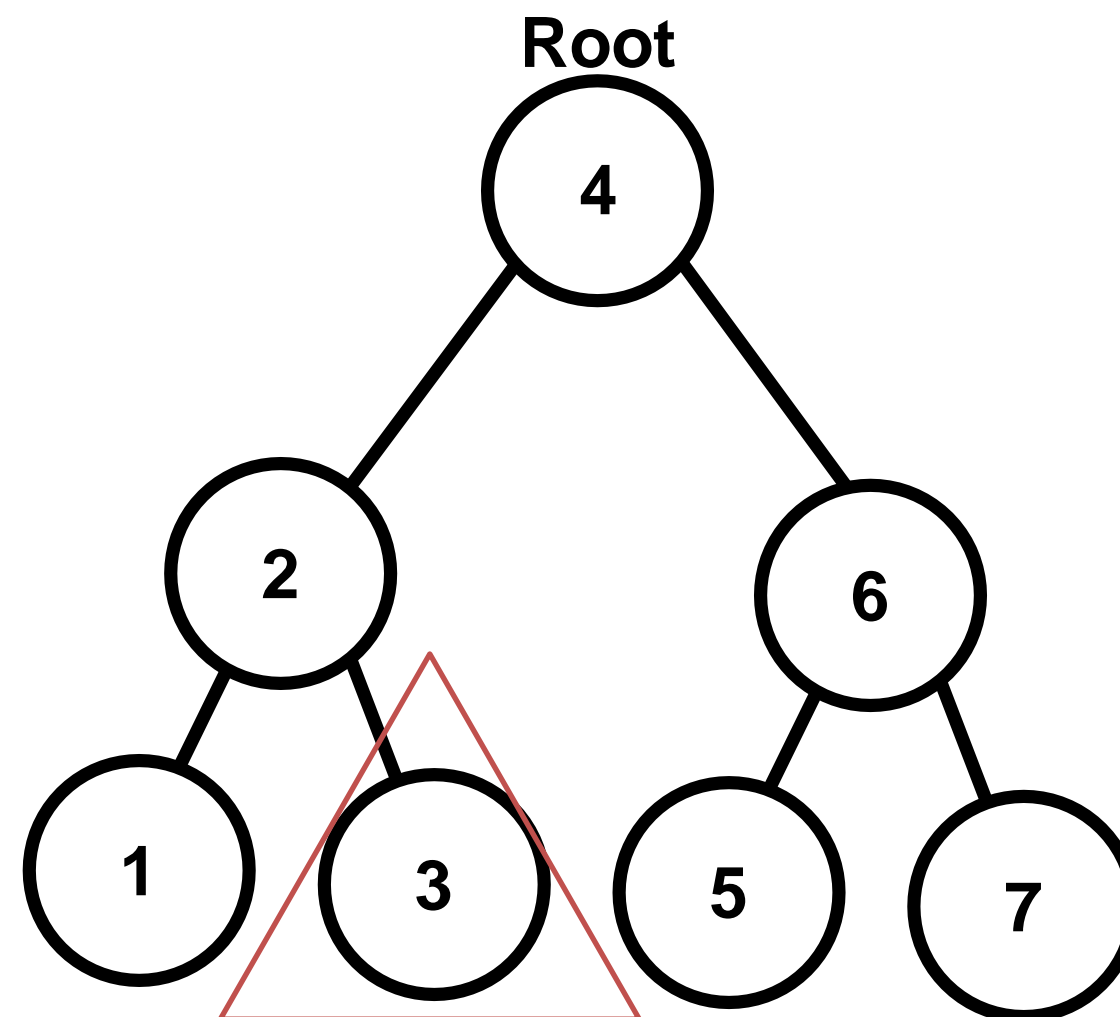
**4 2 1 3 6 5 7**



# BST: Preorder

Preorder:

**4 2 1 3 6 5 7**

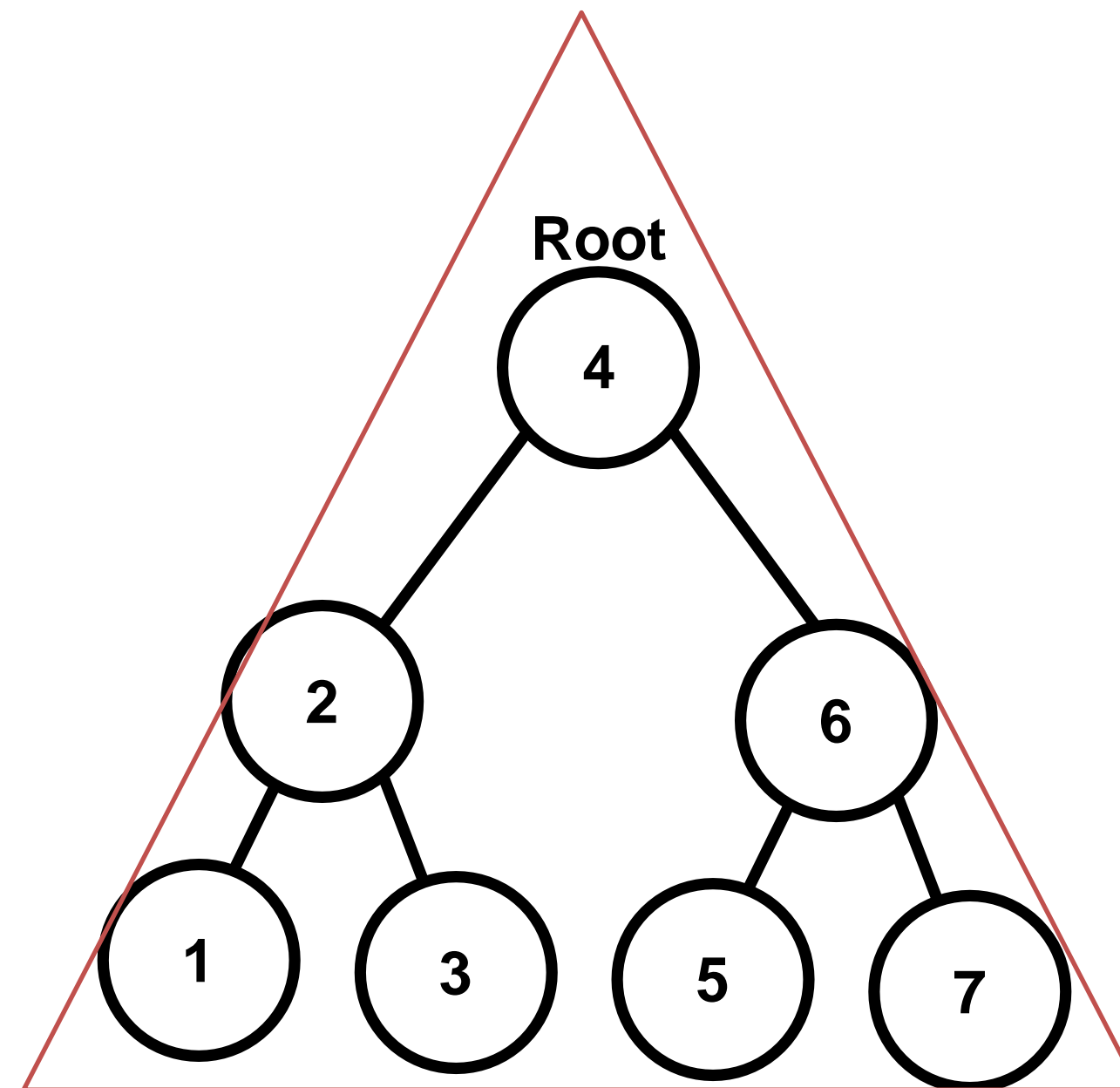




# BST: Preorder

Preorder:

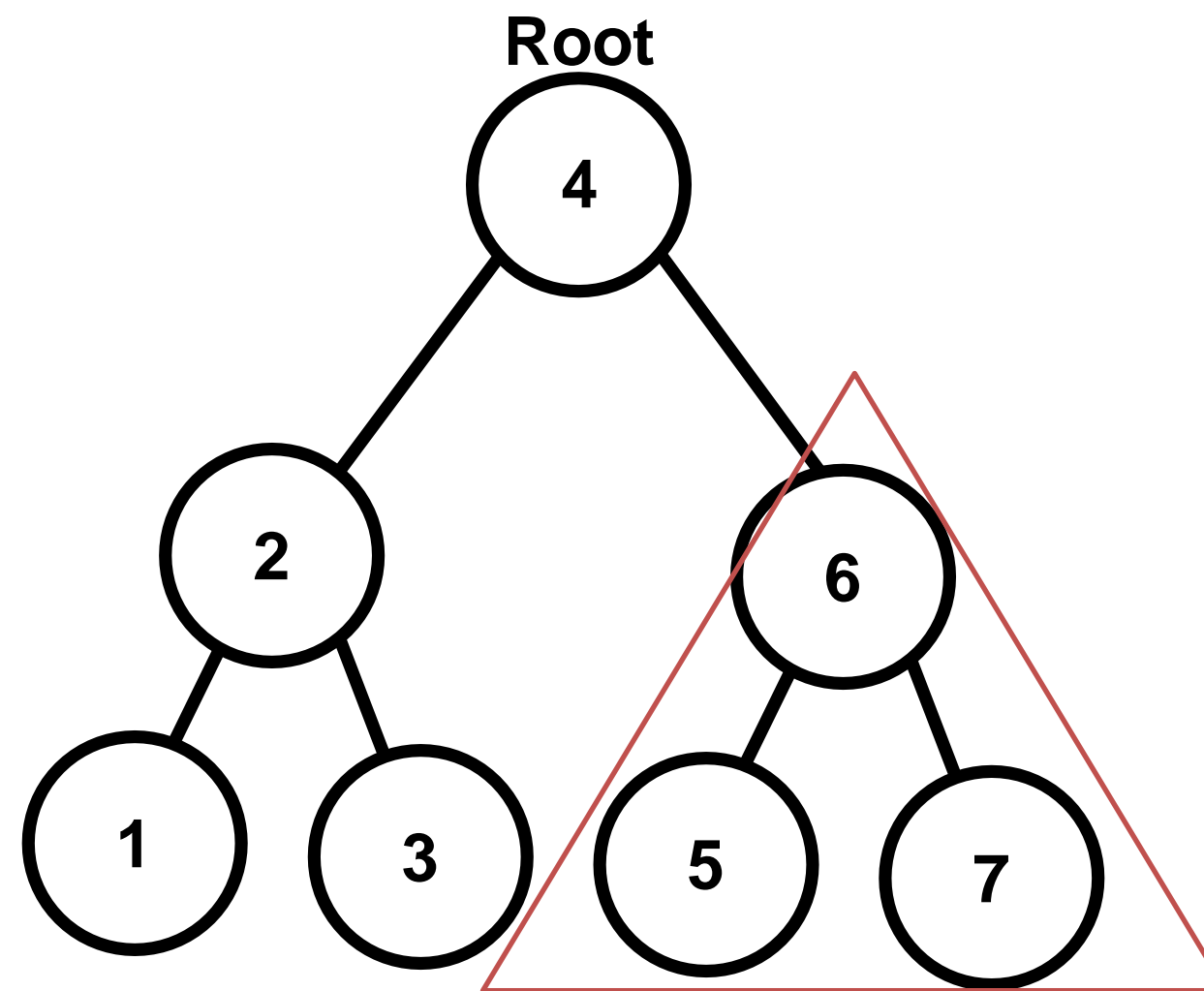
**4 2 1 3 6 5 7**



# BST: Preorder

Preorder:

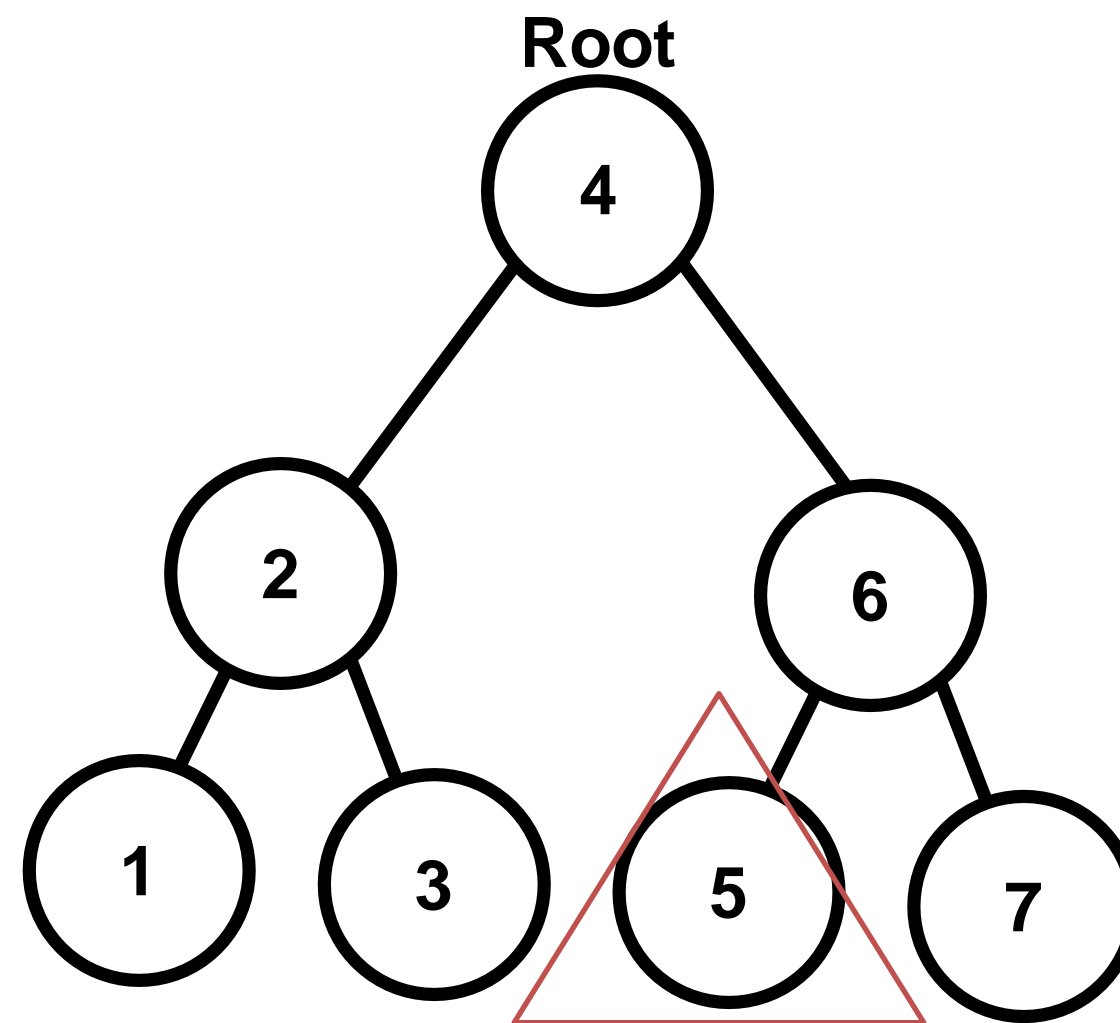
**4 2 1 3 6 5 7**



# BST: Preorder

Preorder:

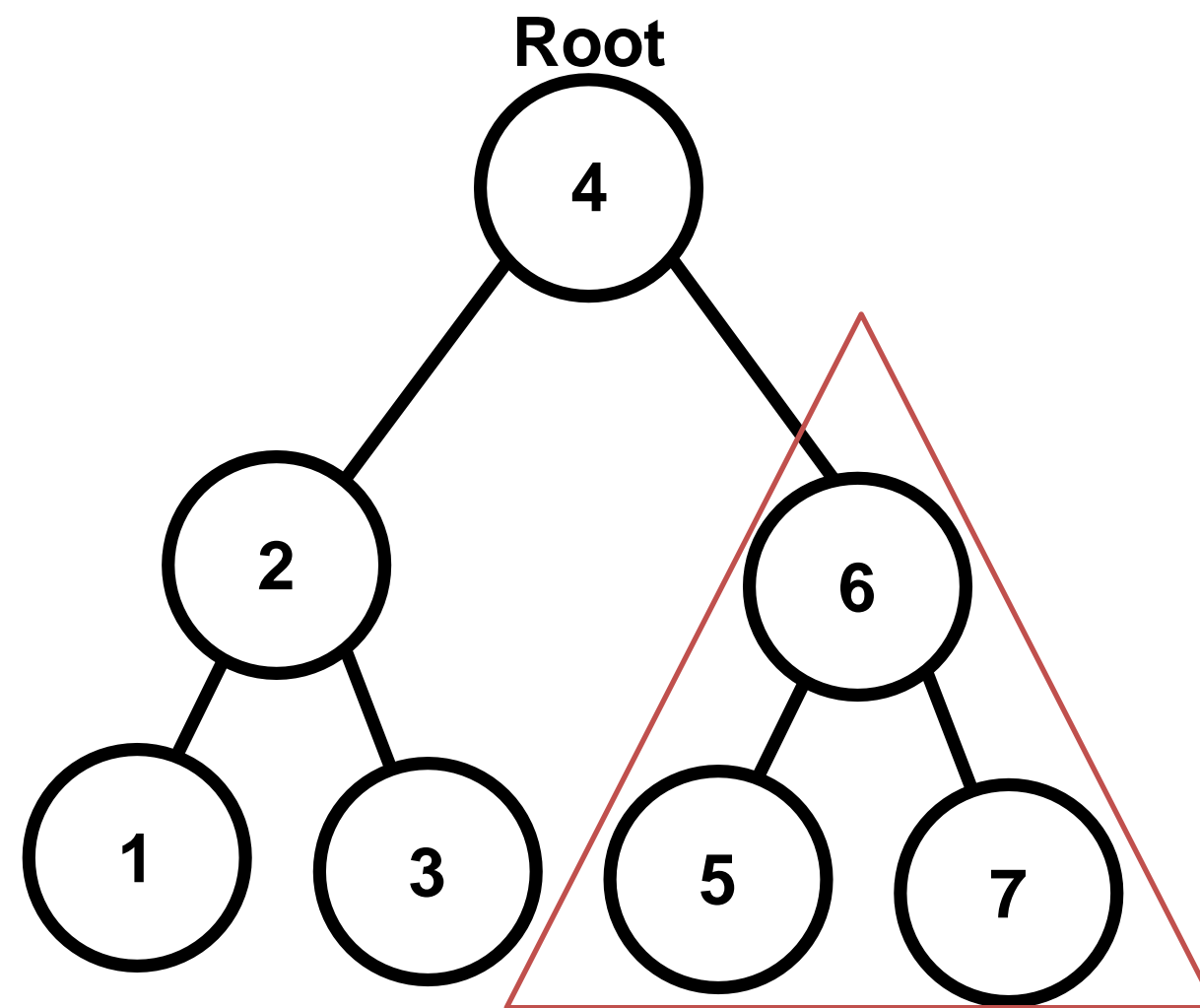
**4 2 1 3 6 5 7**



# BST: Preorder

Preorder:

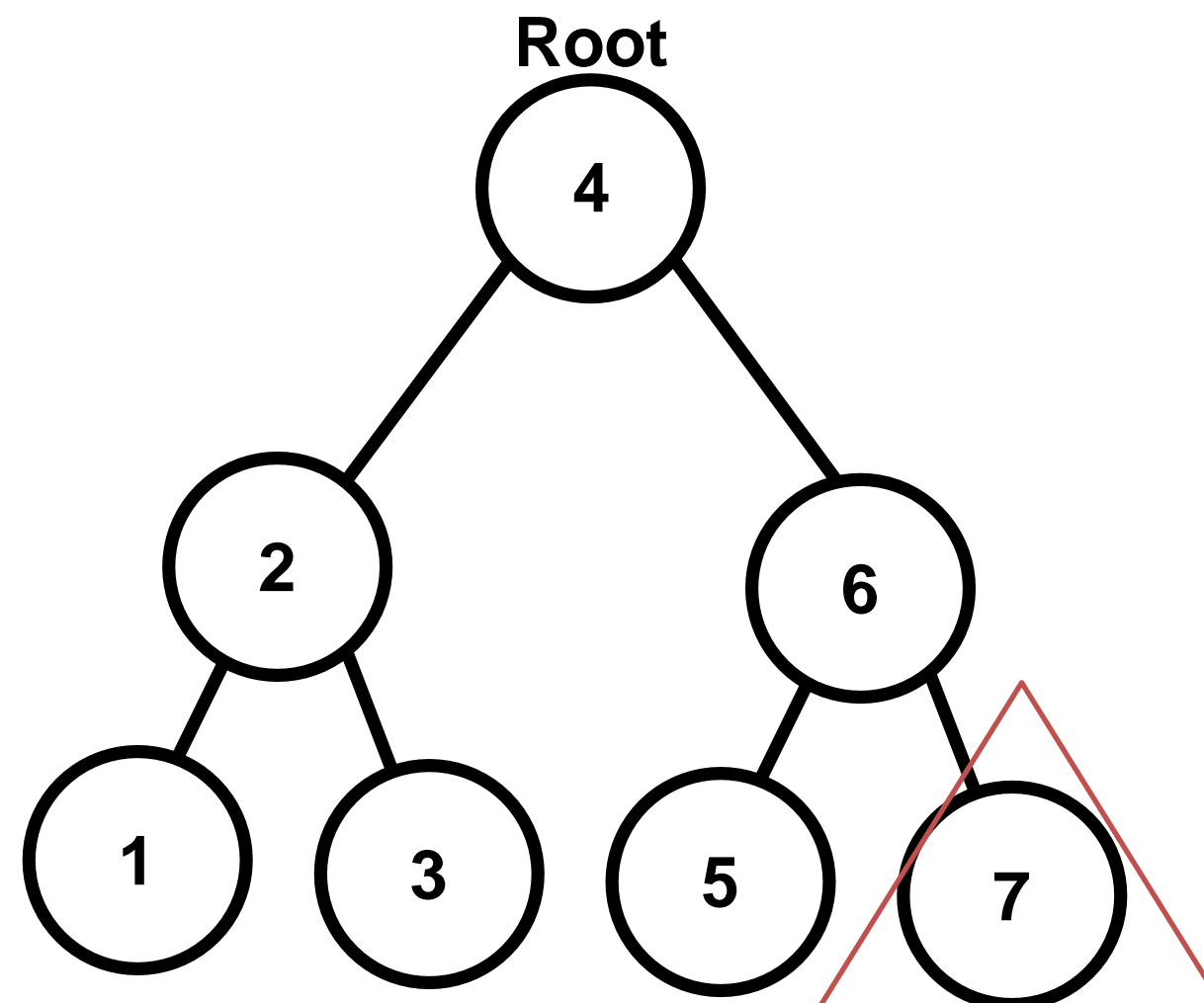
**4 2 1 3 6 5 7**



# BST: Preorder

Preorder:

**4 2 1 3 6 5 7**

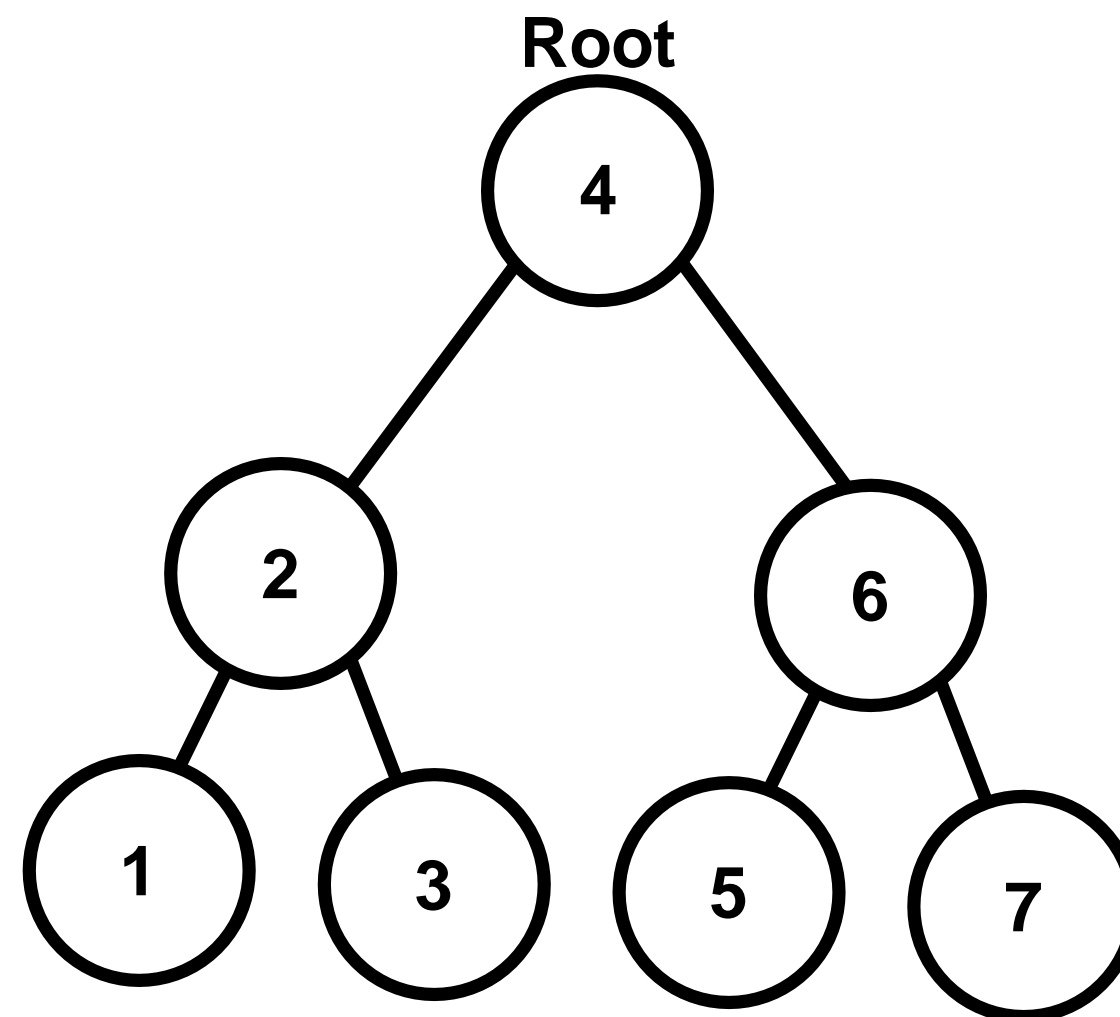


# BST: Preorder

```
void preorder(node *n){  
    if(n == nullptr)  
        return;  
    cout << n->value << " ";  
    preorder(n->left);  
    preorder(n->right);  
}
```

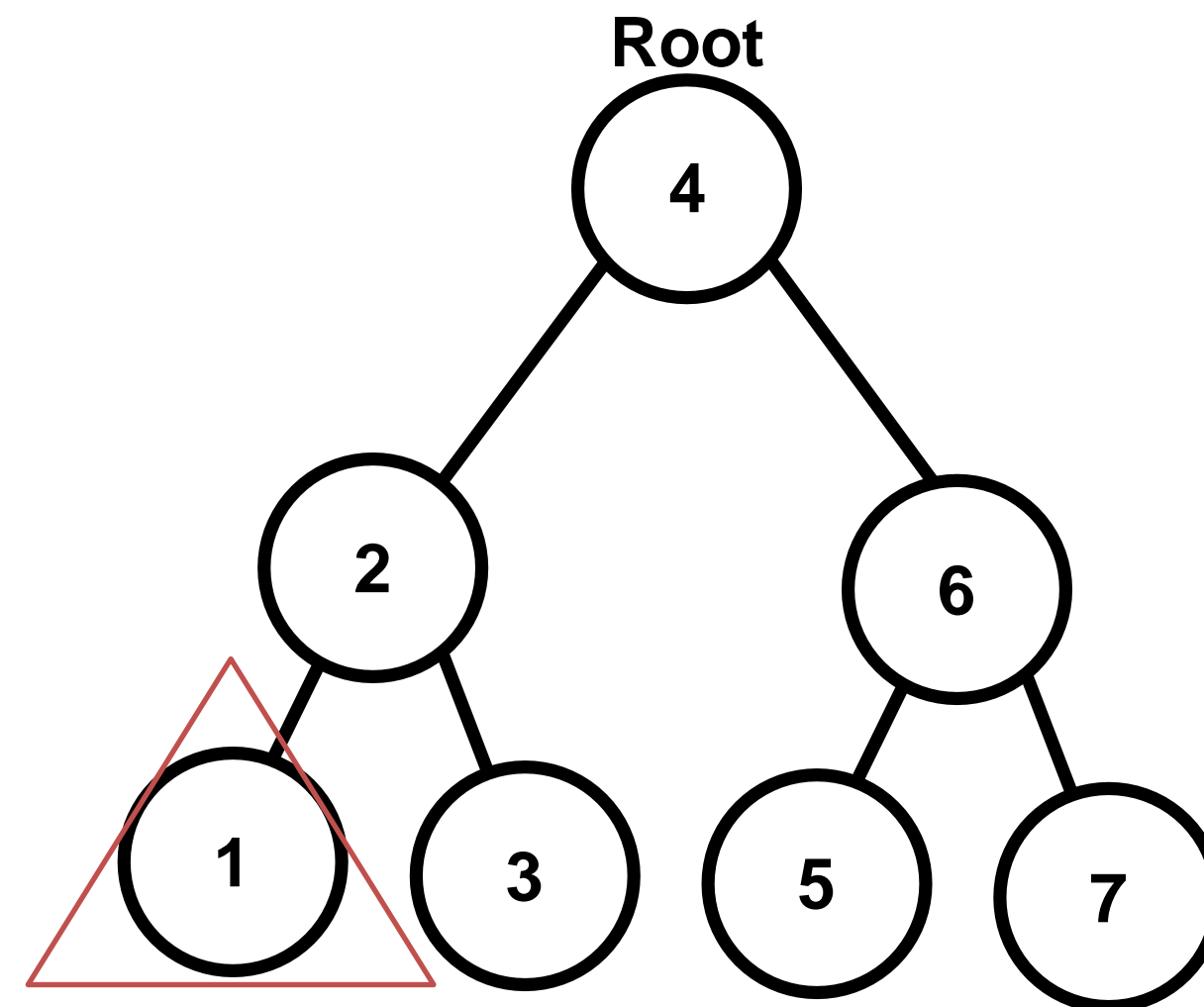
# BST: Postorder

Postorder:  
1 3 2 5 7 6 4



# BST: Postorder

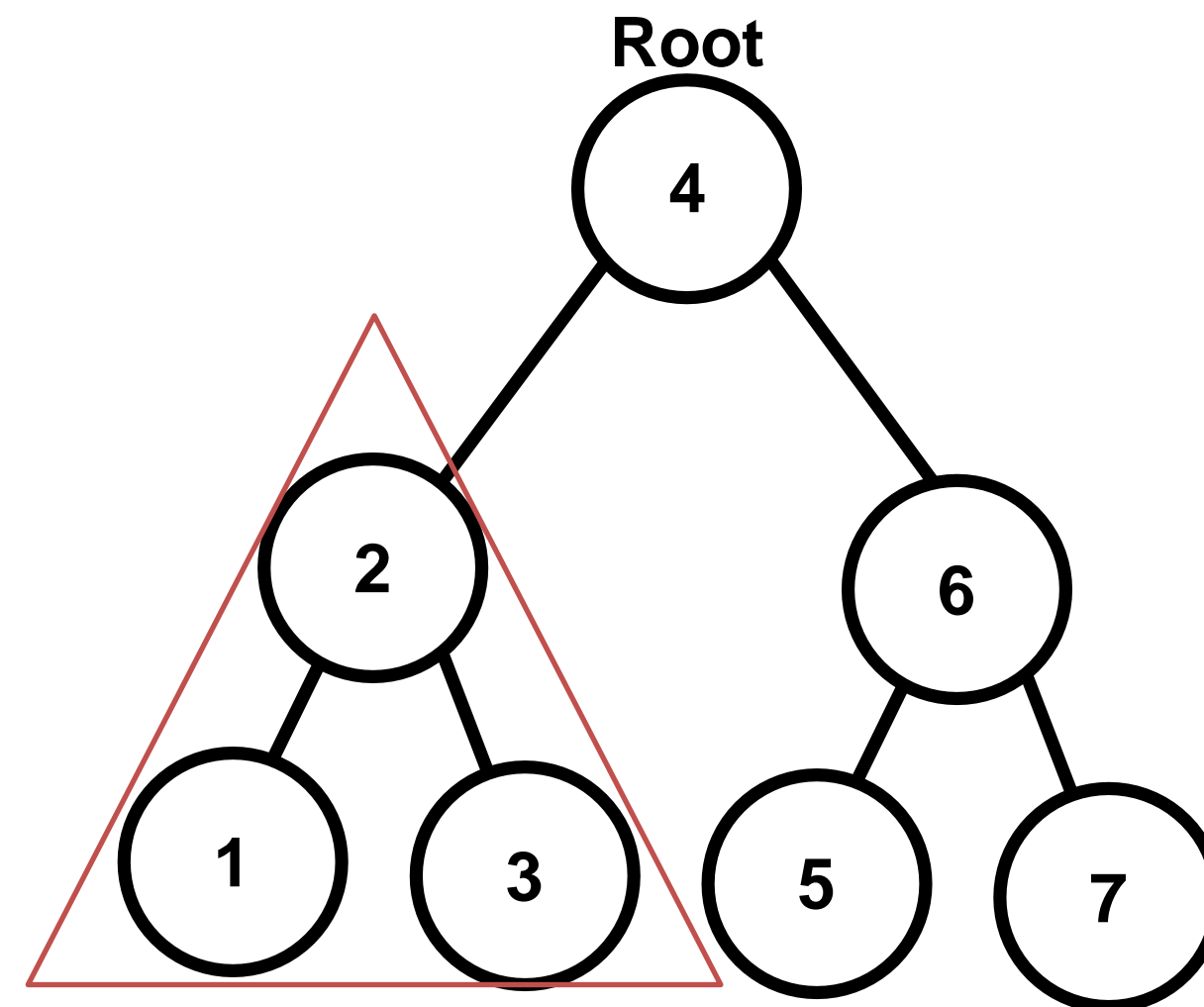
Postorder:  
1 3 2 5 7 6 4





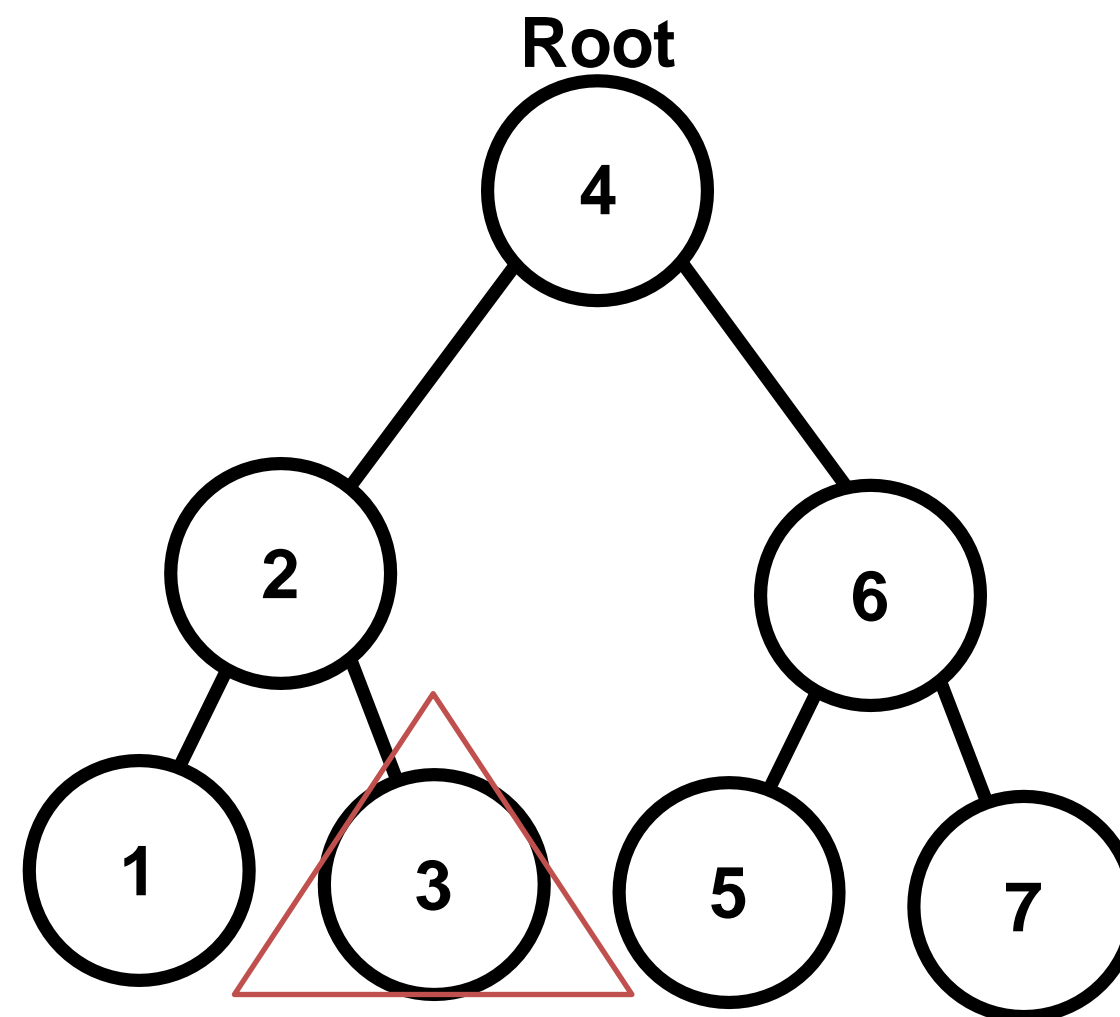
# BST: Postorder

Postorder:  
1 3 2 5 7 6 4



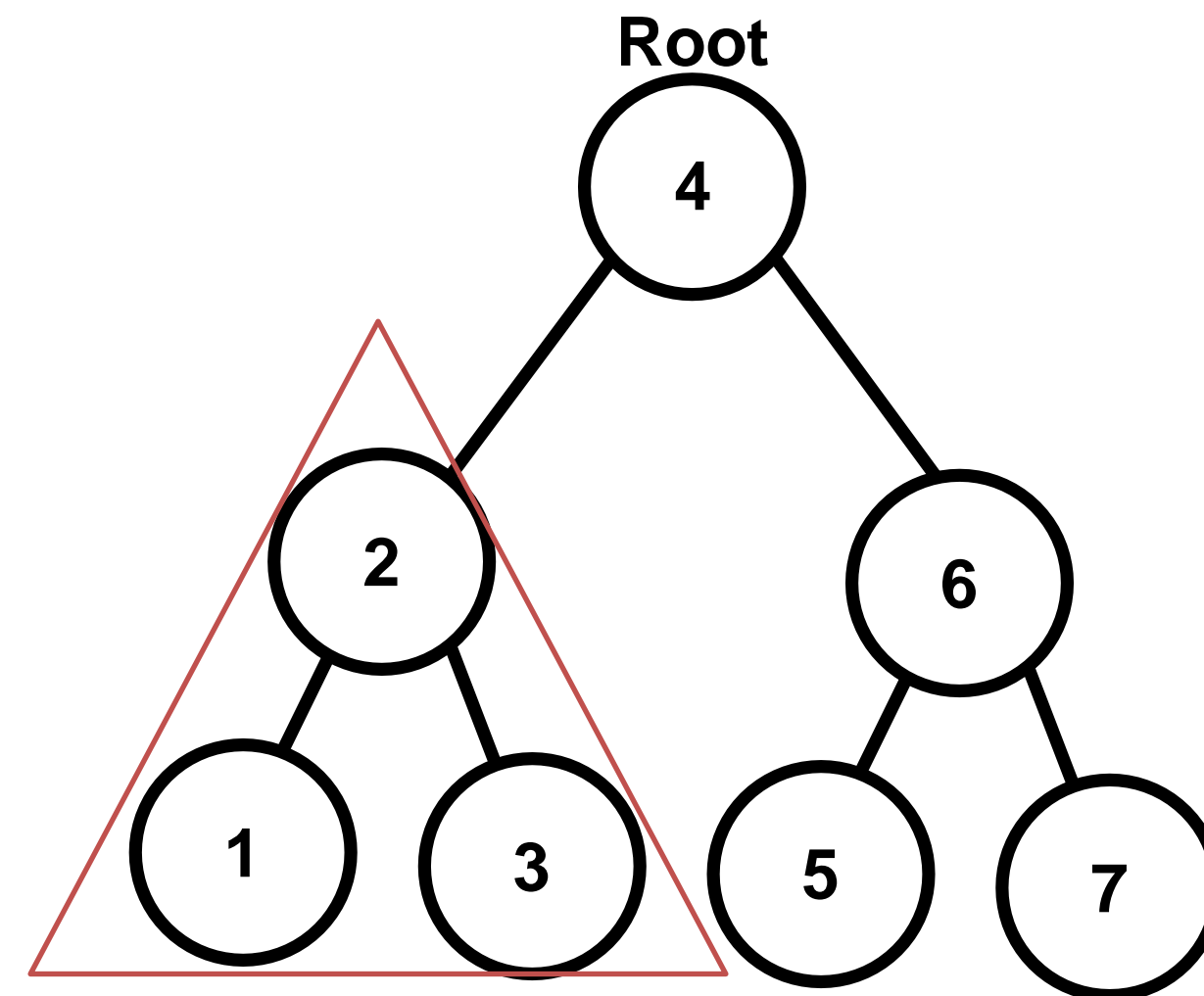
# BST: Postorder

Postorder:  
1 3 2 5 7 6 4



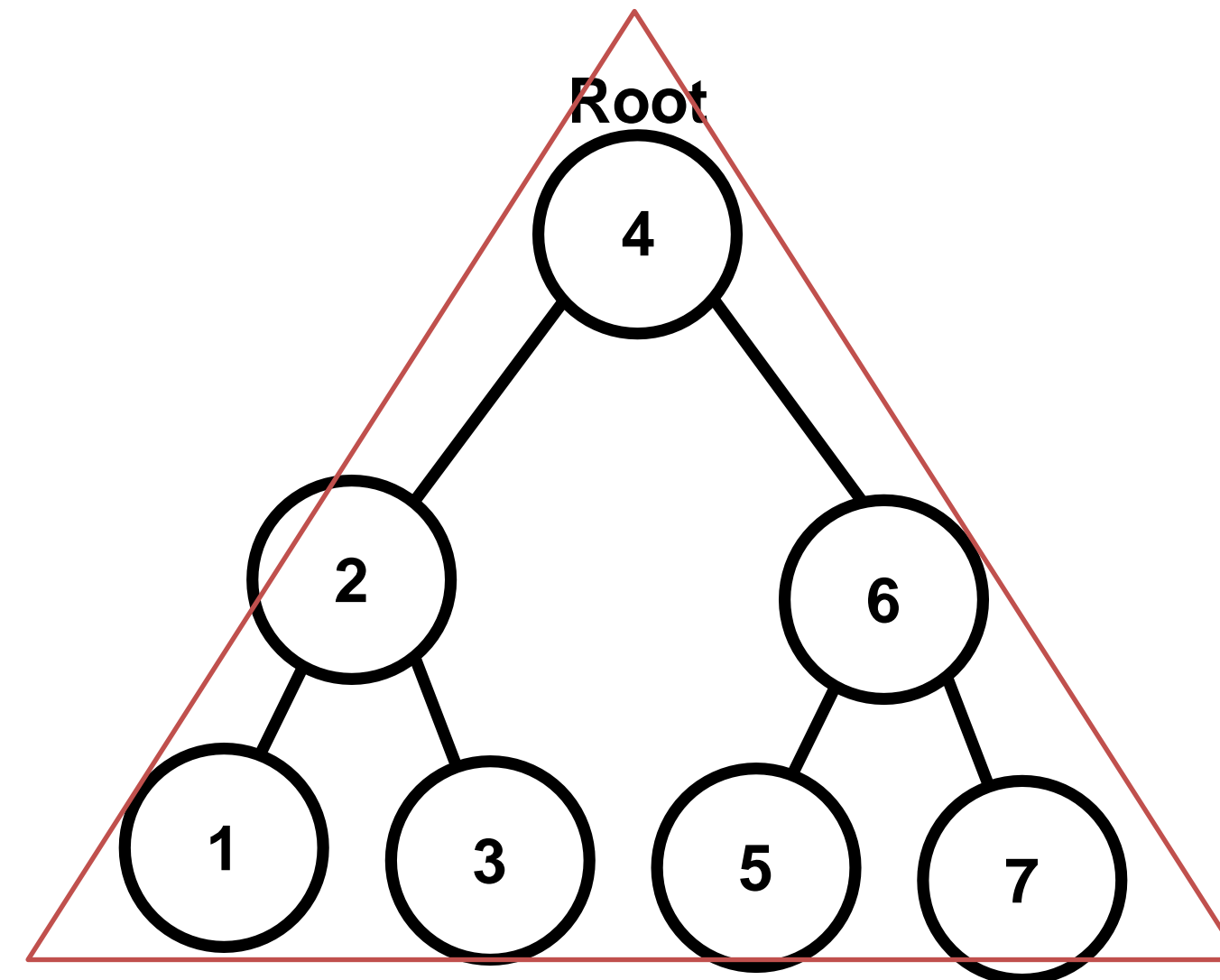
# BST: Postorder

Postorder:  
1 3 2 5 7 6 4



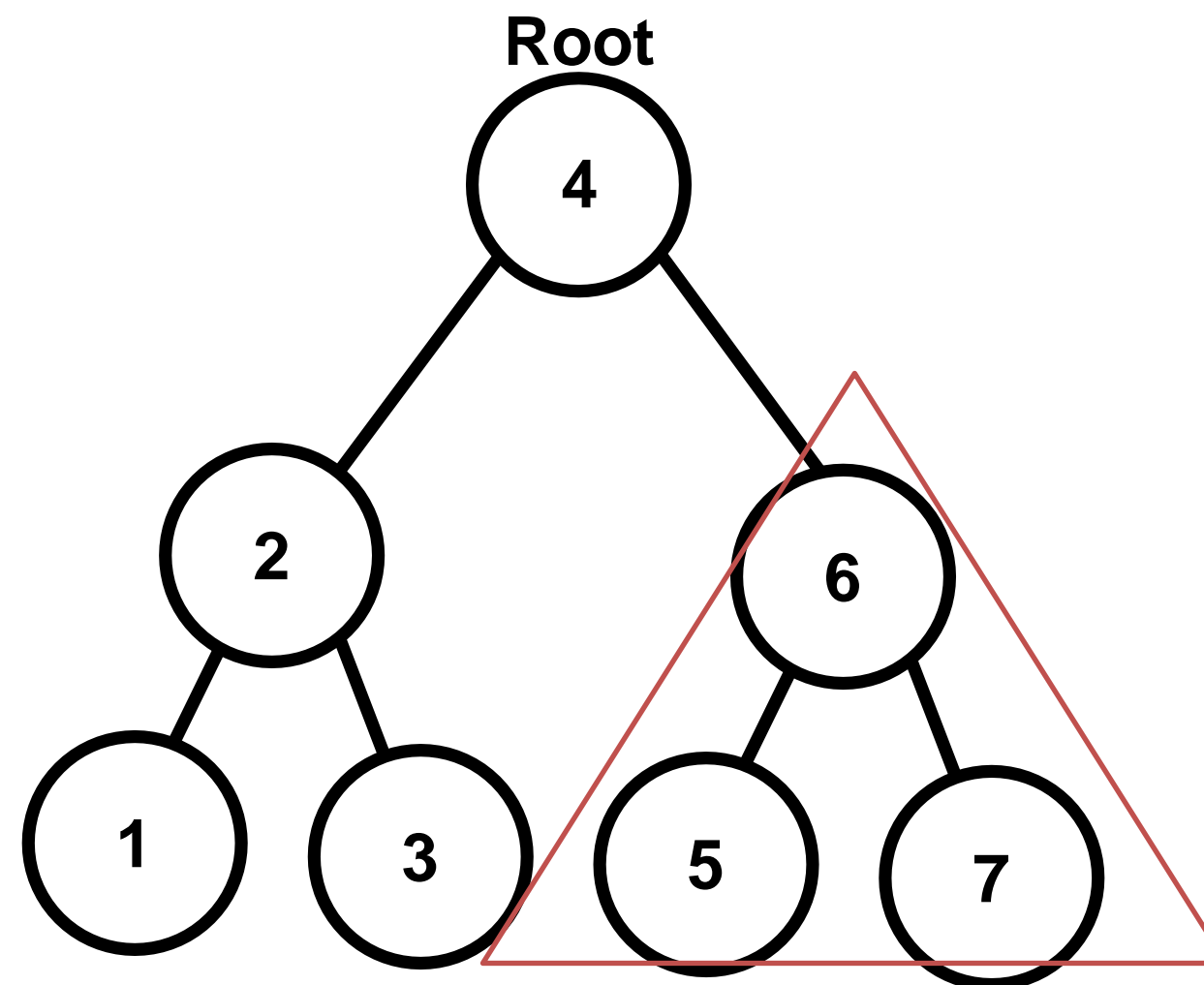
# BST: Postorder

Postorder:  
1 3 2 5 7 6 4



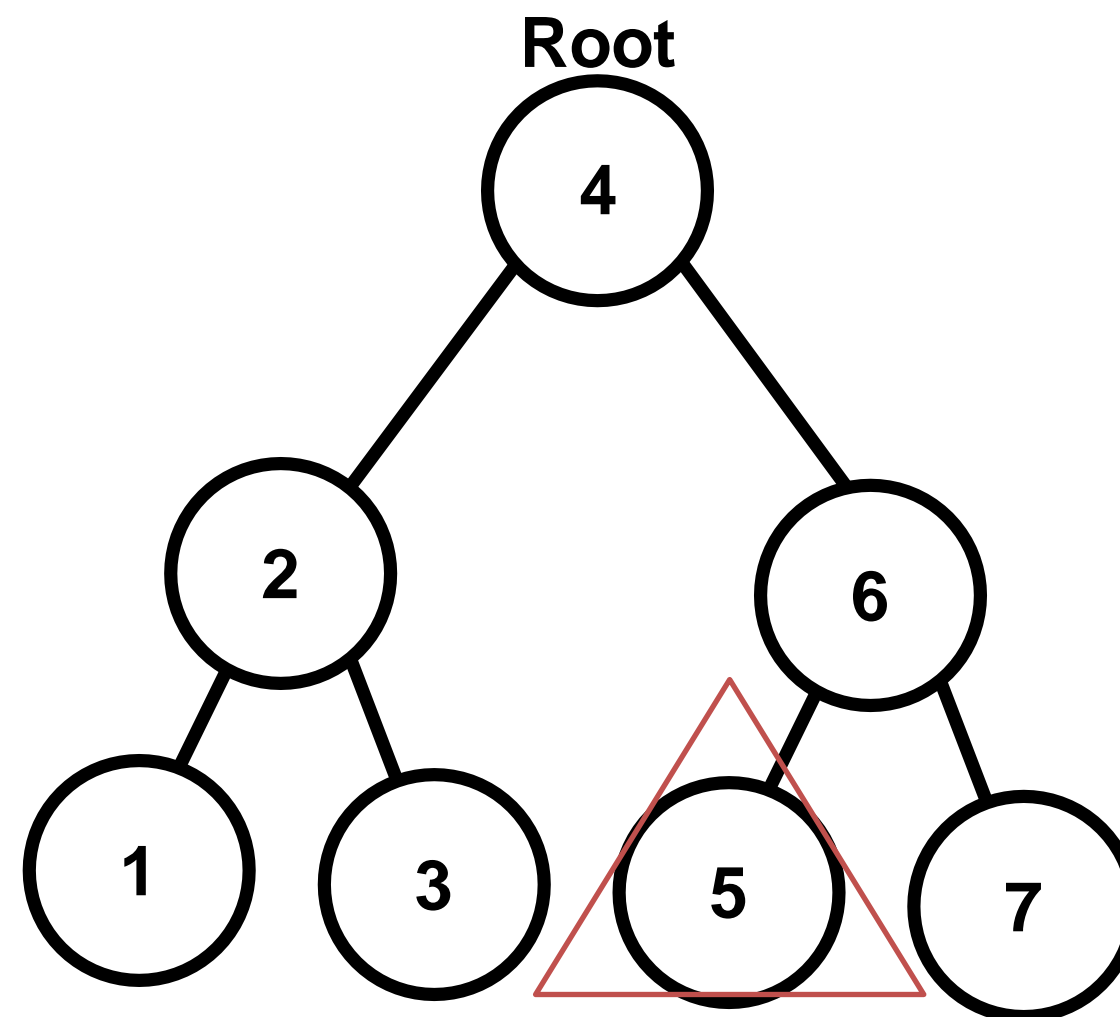
# BST: Postorder

Postorder:  
1 3 2 5 7 6 4



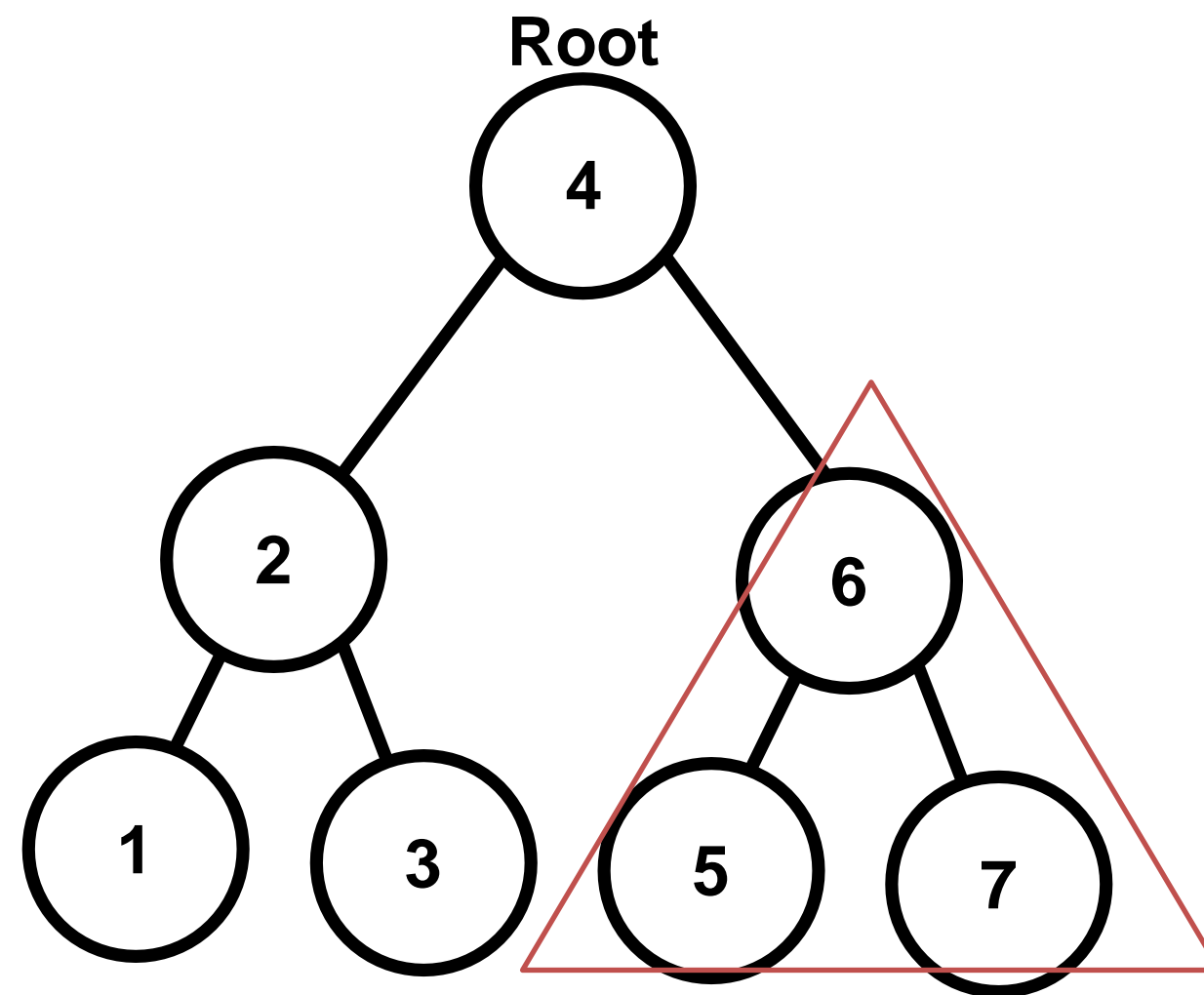
# BST: Postorder

Postorder:  
1 3 2 5 7 6 4



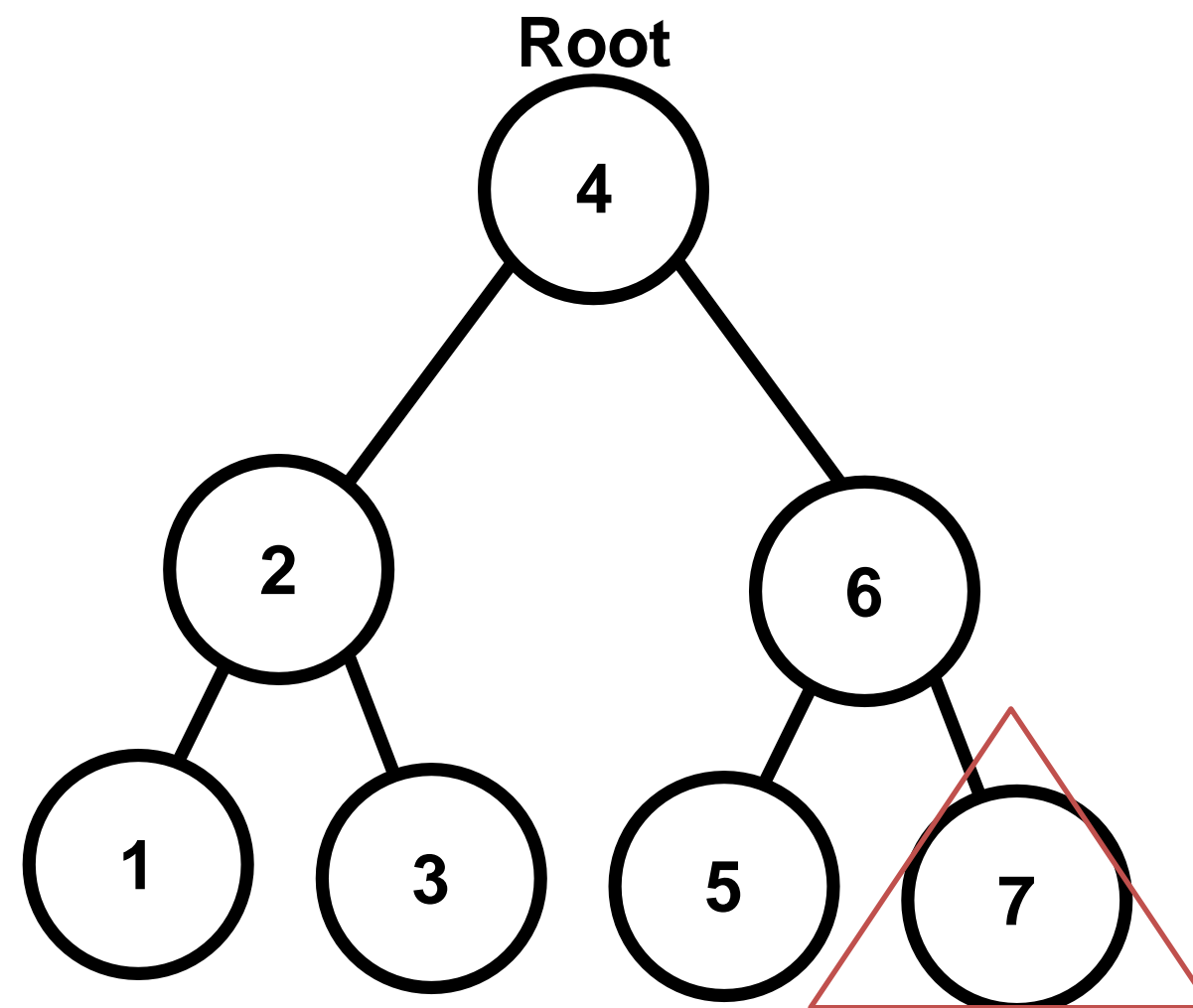
# BST: Postorder

Postorder:  
1 3 2 5 7 6 4



# BST: Postorder

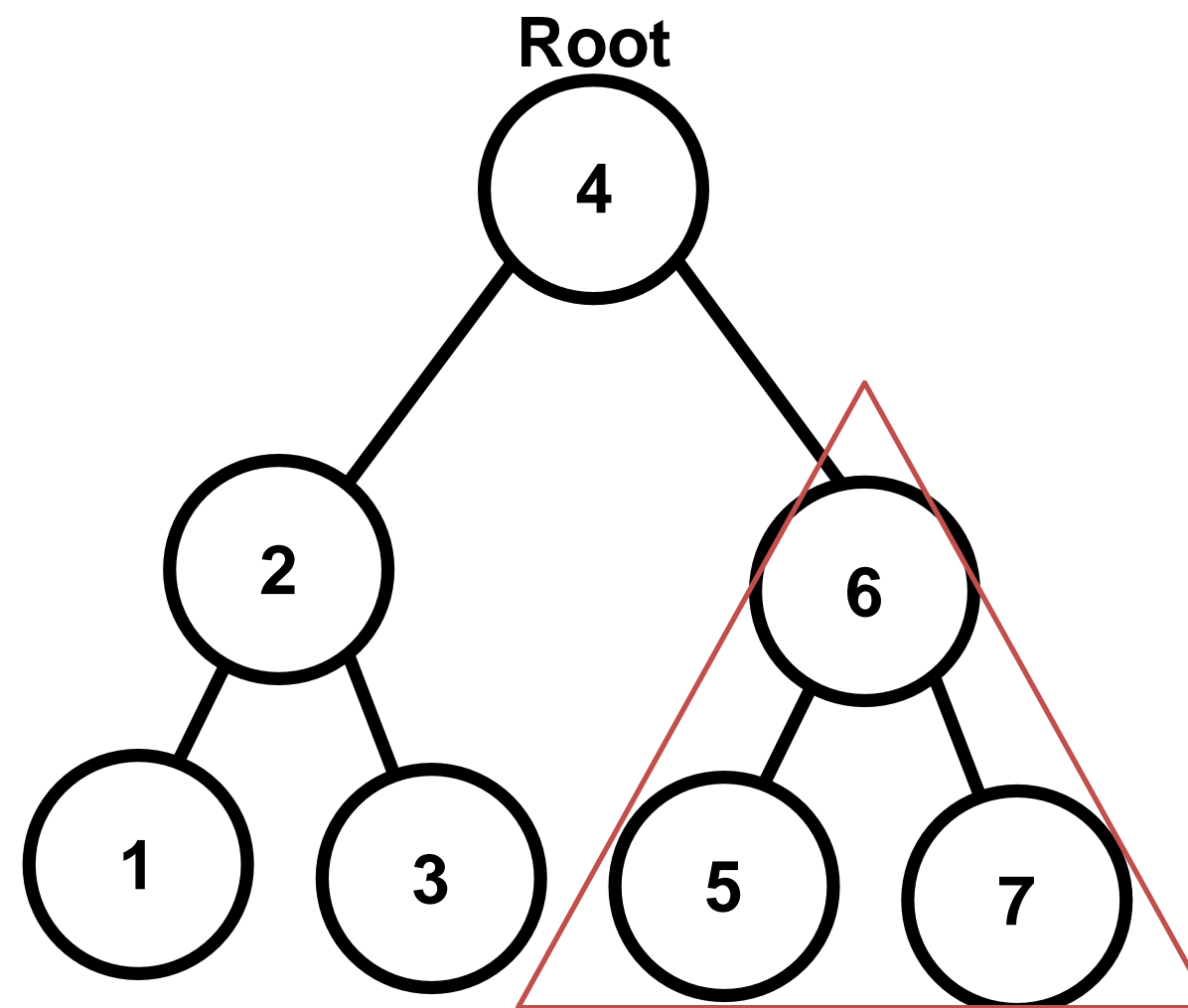
Postorder:  
1 3 2 5 7 6 4





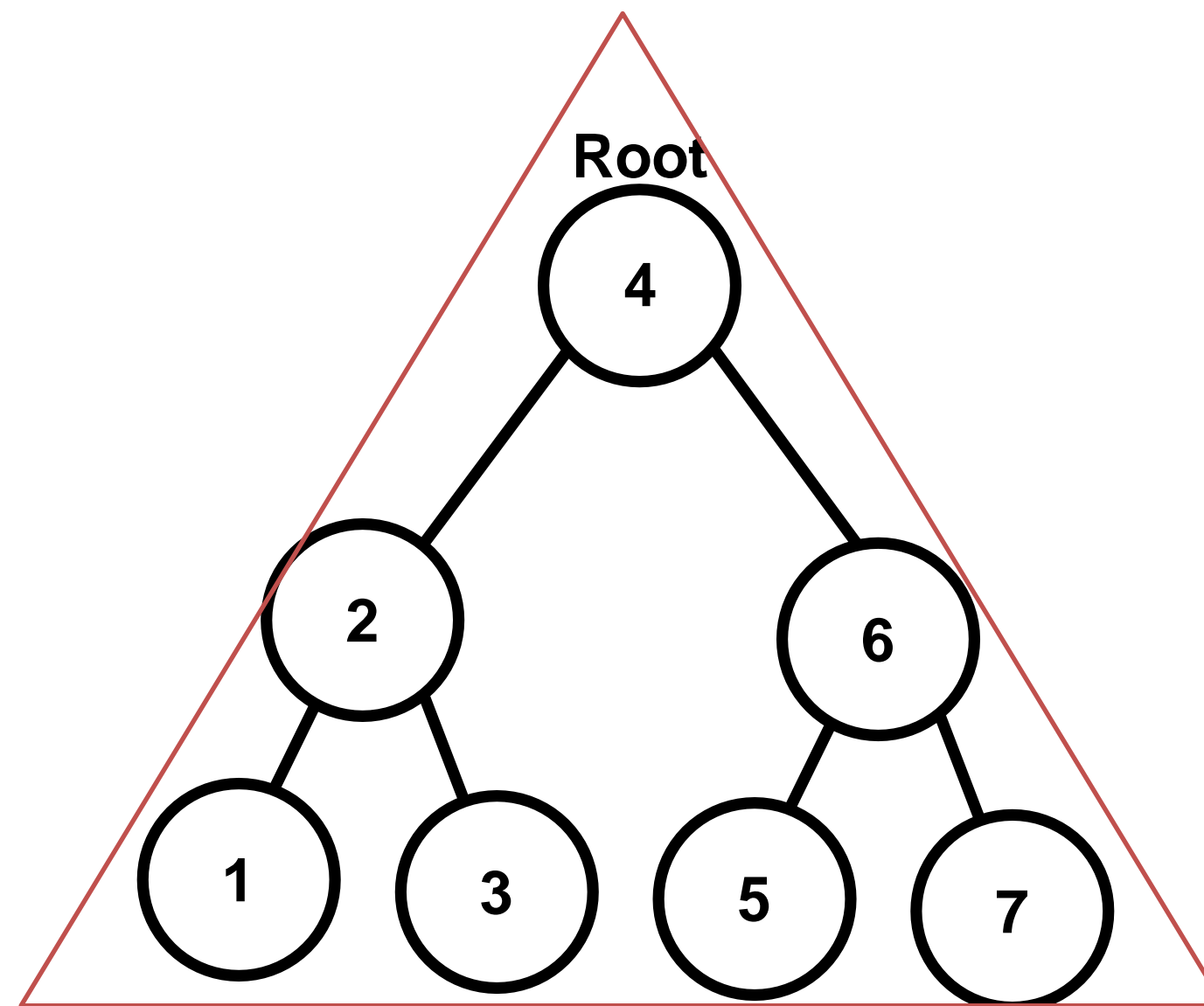
# BST: Postorder

Postorder:  
1 3 2 5 7 6 4



# BST: Postorder

Postorder:  
1 3 2 5 7 6 4



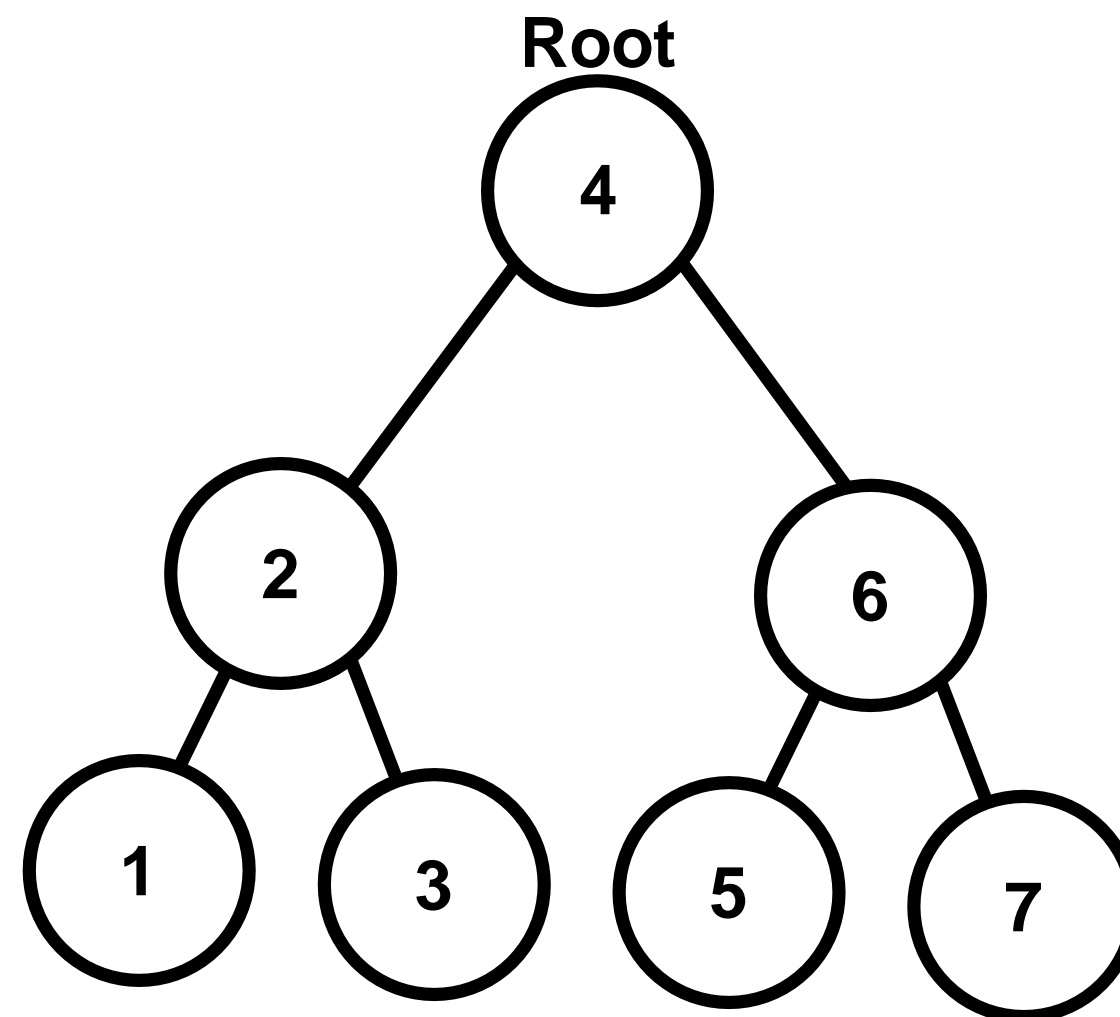
# BST: Postorder

```
void postorder(node *n){  
    if(n == nullptr)  
        return;  
    postorder(n->left);  
    postorder(n->right);  
    cout << n->value << " ";  
}
```

# BST: Reverse Order

Reverse Order:

**7 6 5 4 3 2 1**

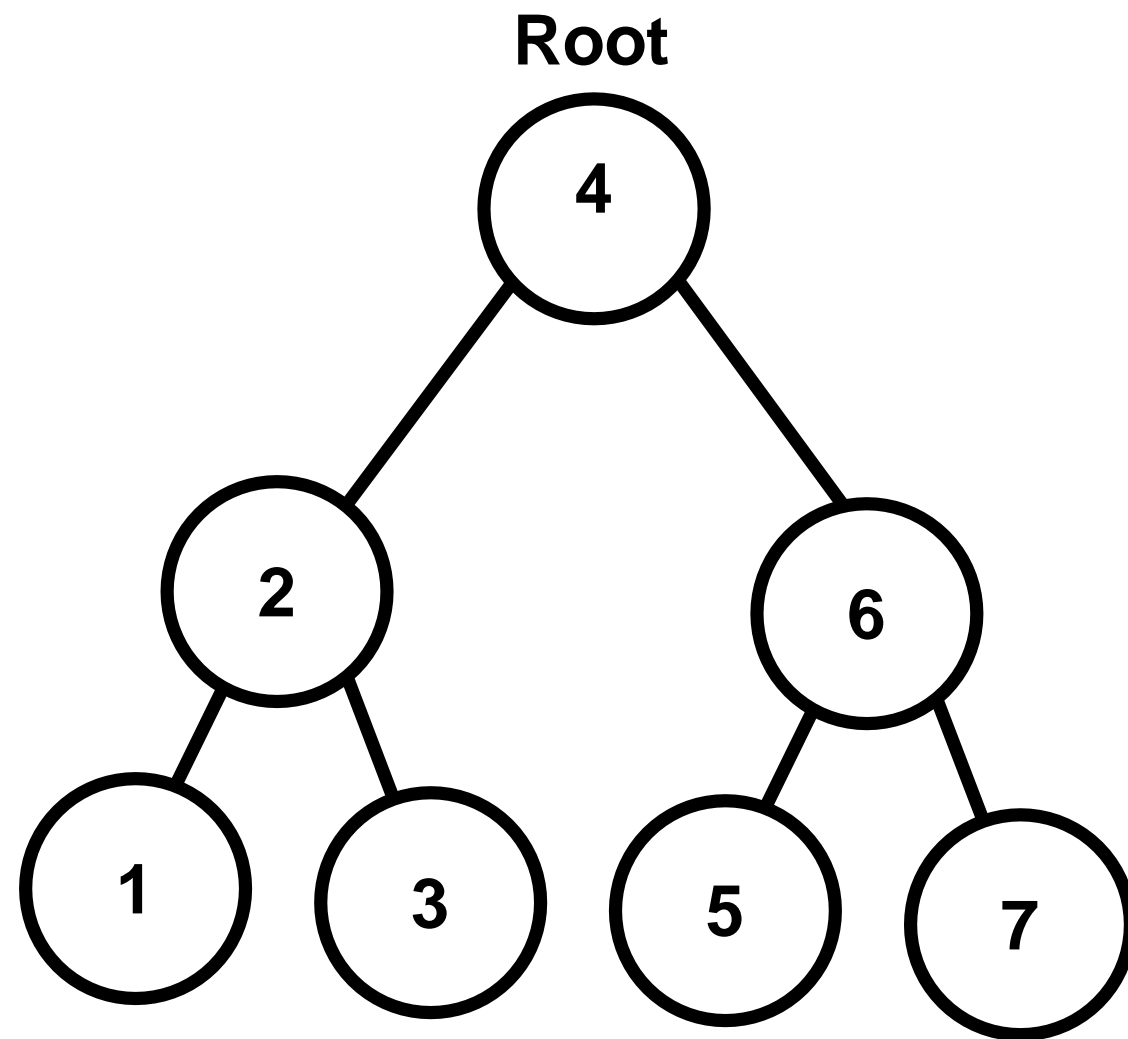


# BST: Reverse Order

```
void reverseorder(node *n){  
    if(n == nullptr)  
        return;  
    reverseorder(n->right);  
    cout << n->value << " ";  
    reverseorder(n->left);  
}
```

# BST: isBST()

Write a function that returns **true** if a tree is a BST and **false** otherwise.



**True**

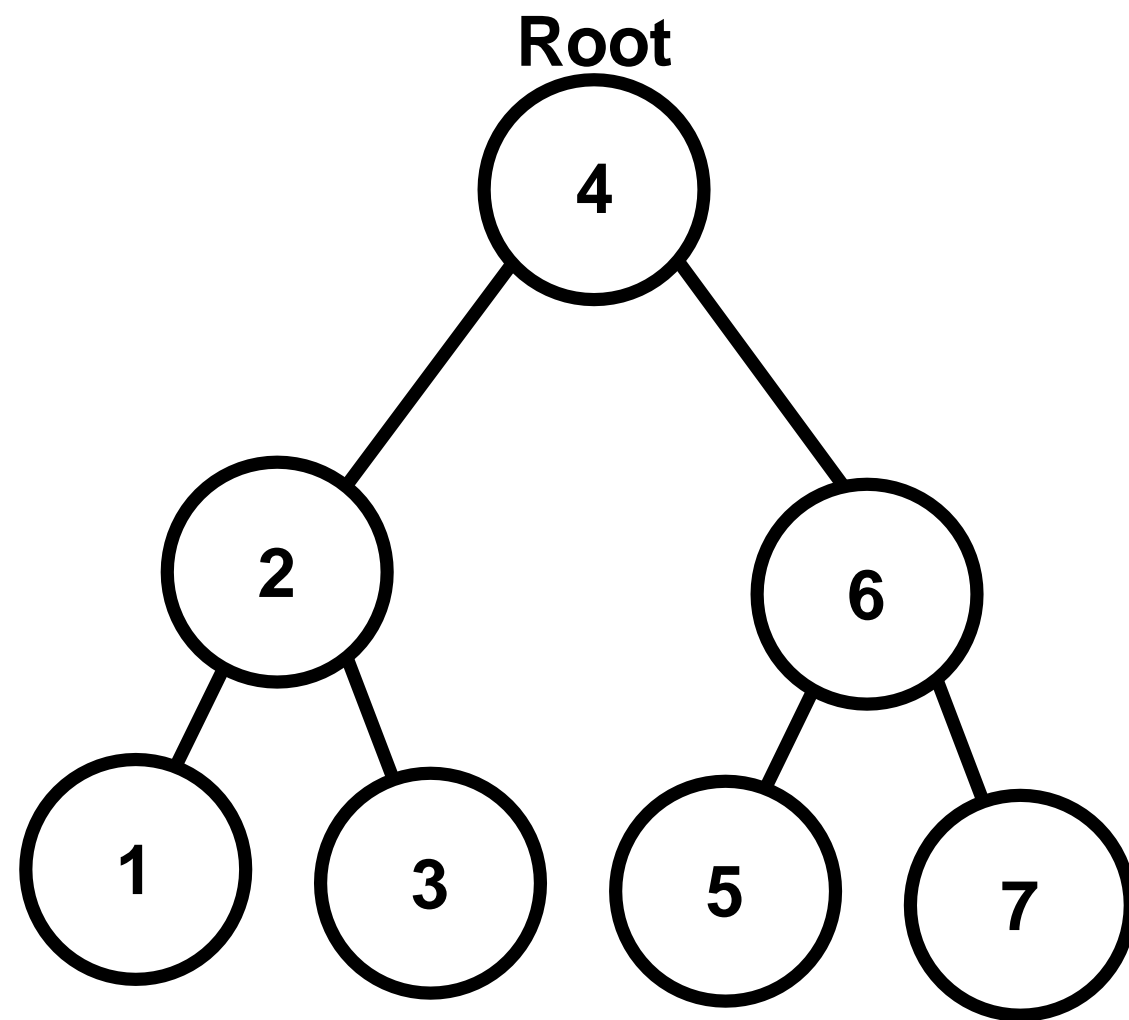


# BST: isBST()

```
165 ▼ bool isBST(node *n){  
166     if(n == nullptr)  
167         return true;  
168     else if(n->left != nullptr && n->left->value > n->value)  
169         return false;  
170     else if(n->right != nullptr && n->right->value < n->value)  
171         return false;  
172     else  
173         return (isBST(n->left) && isBST(n->right));  
174 }
```

# BST: printLeaves()

Write a function to print all of the leaf values of a binary tree.



**1 3 5 7**



# BST: printLeaves()

```
176 ▼ void printLeaves(node *n){  
177     if(n == nullptr)  
178         return;  
179 ▼ if(n->left == nullptr && n->right == nullptr){  
180     cout << n->value << " ";  
181     return;  
182 }  
183 printLeaves(n->left);  
184 printLeaves(n->right);  
185 }
```