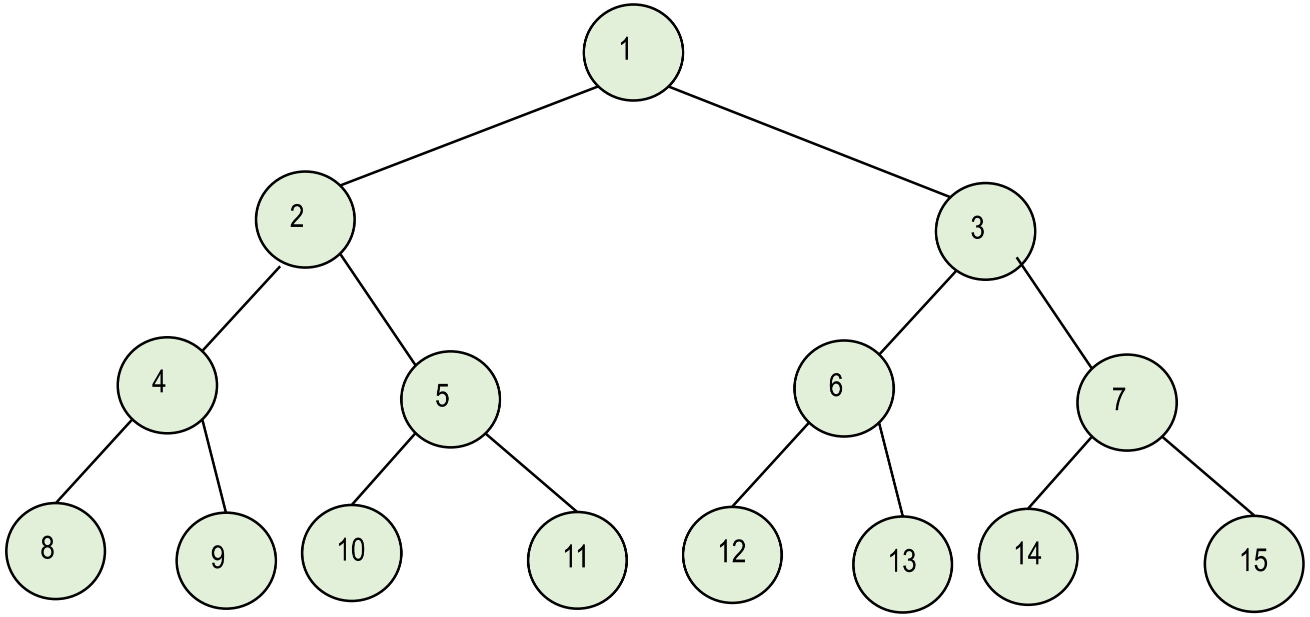
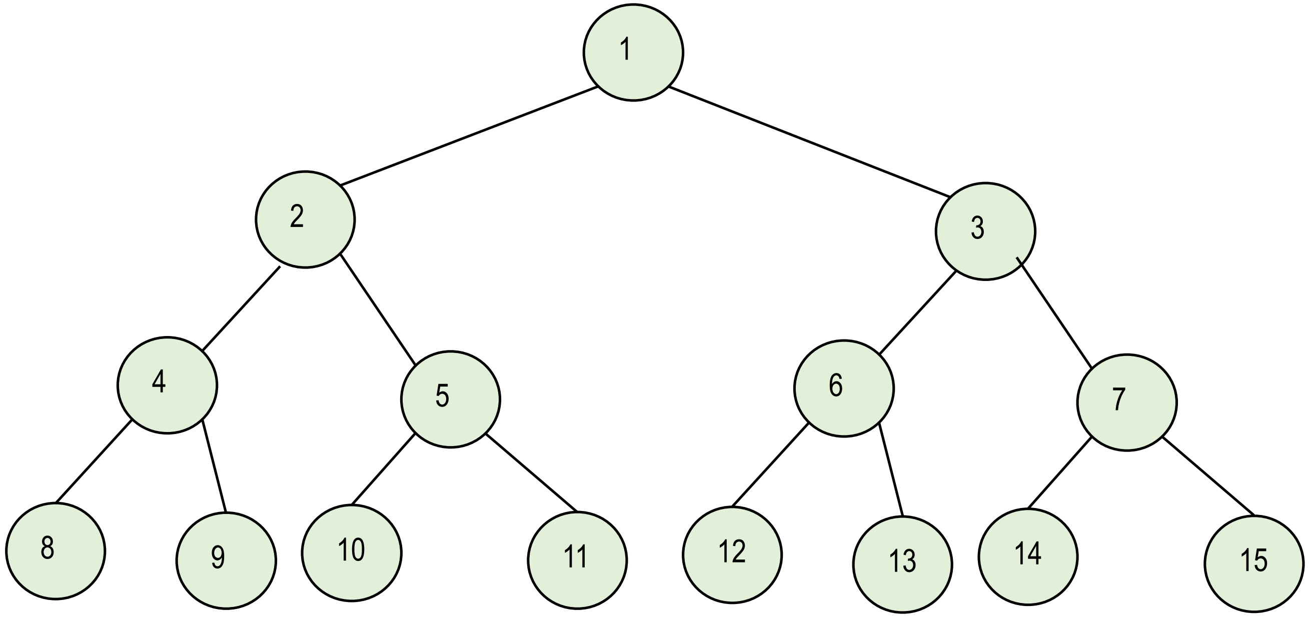
Binary Tree Traversal Methods
and
Tree Construction Methods

Binary Tree Traversal



Binary Tree Traversal

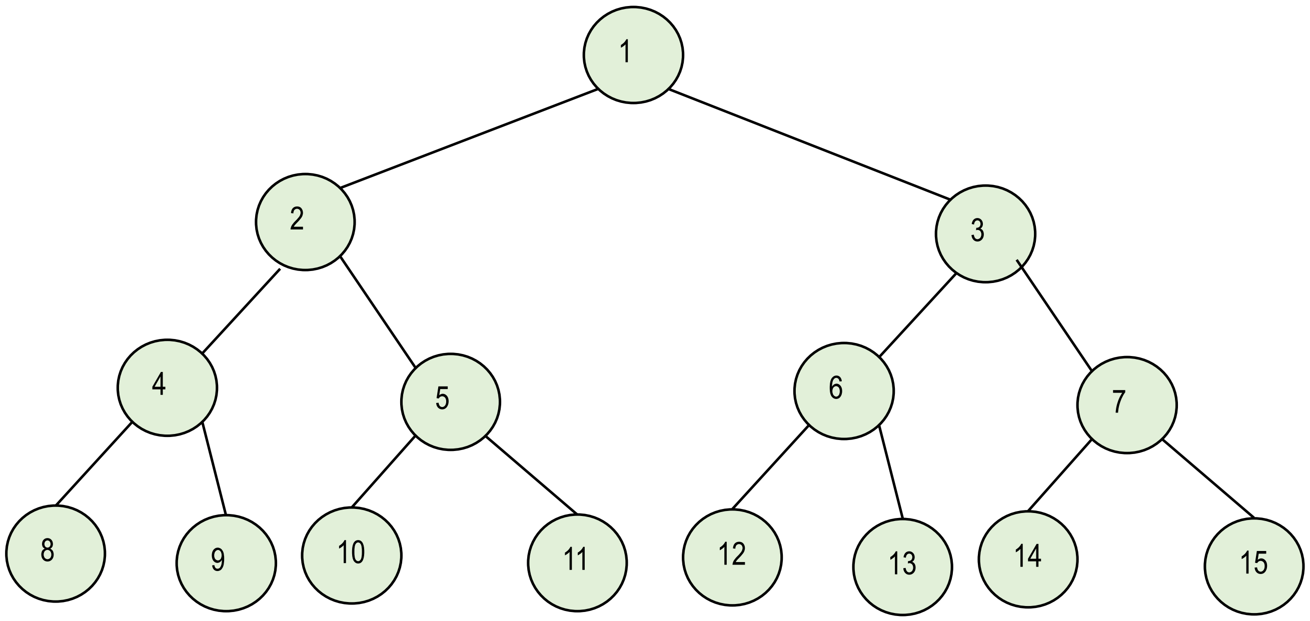
- In a traversal of a binary tree, A1 of the binary tree is visited A2



Binary Tree Traversal

- In a traversal of a binary tree, each element of the binary tree is visited exactly once.

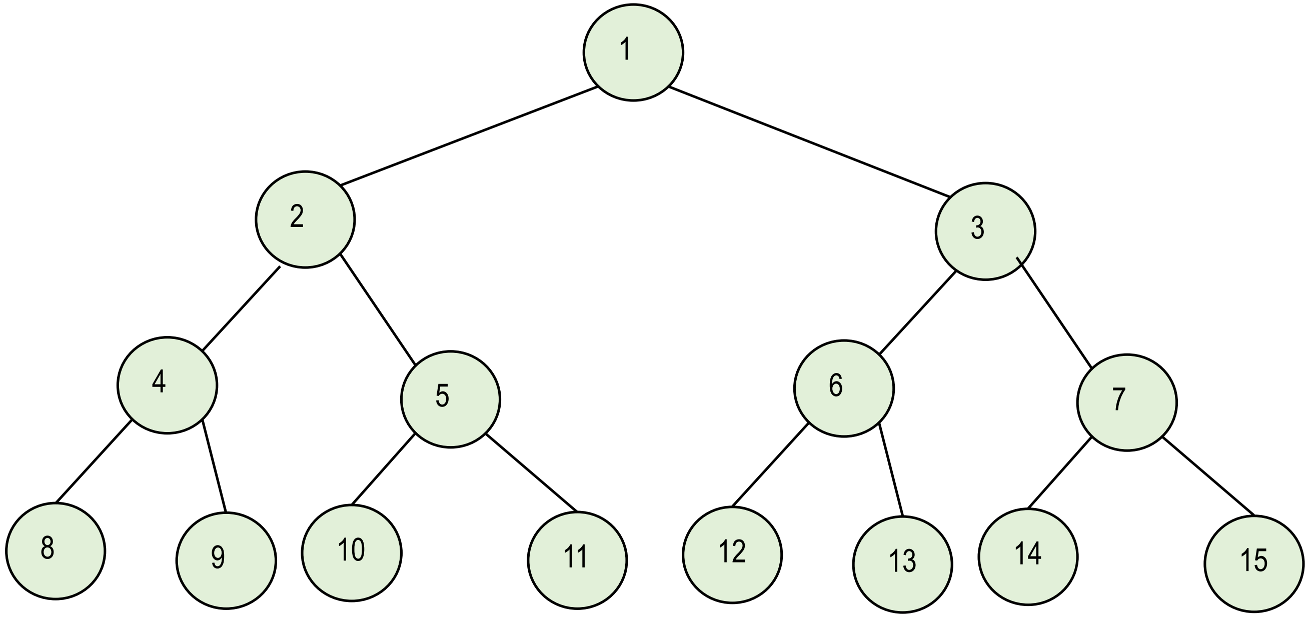
- We can perform A1 (e.g., display, operator evaluation, etc.) on the elements of the node during A2



Binary Tree Traversal

- In a traversal of a binary tree, each element of the binary tree is visited exactly once.

- We can perform operations (e.g., display, operator evaluation, etc.) on the elements of the node during the visit.

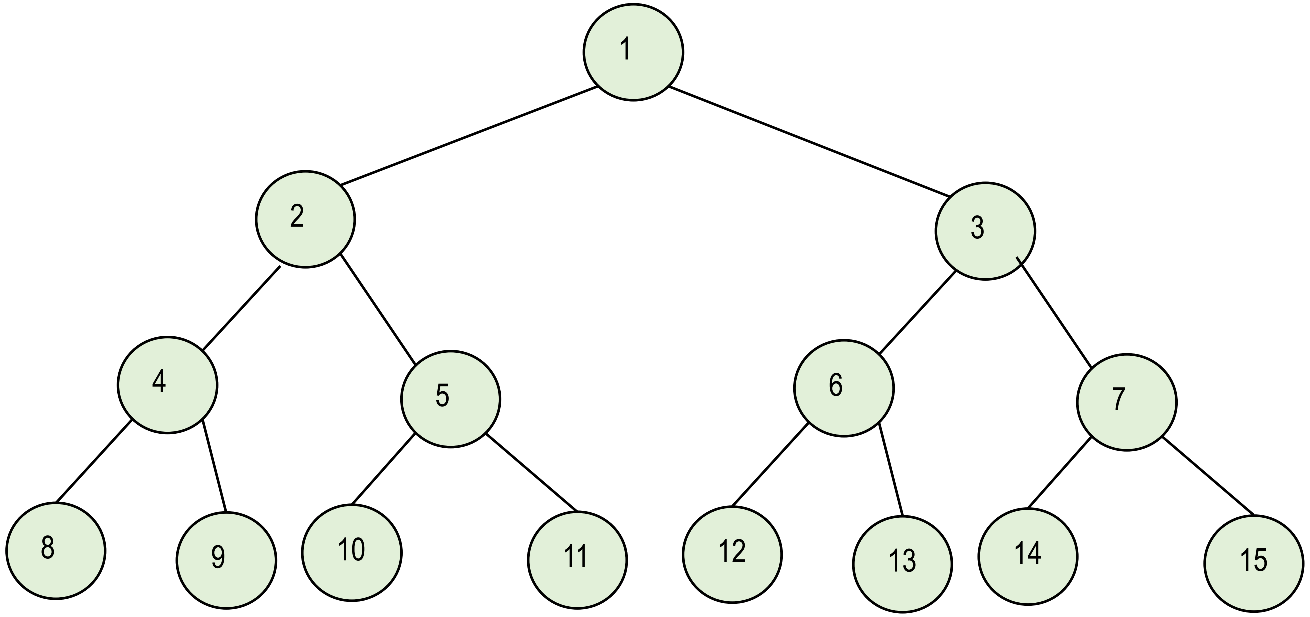


Binary Tree Traversal Methods

- Preorder
- Inorder
- Postorder
- Level order (Breadth-first)

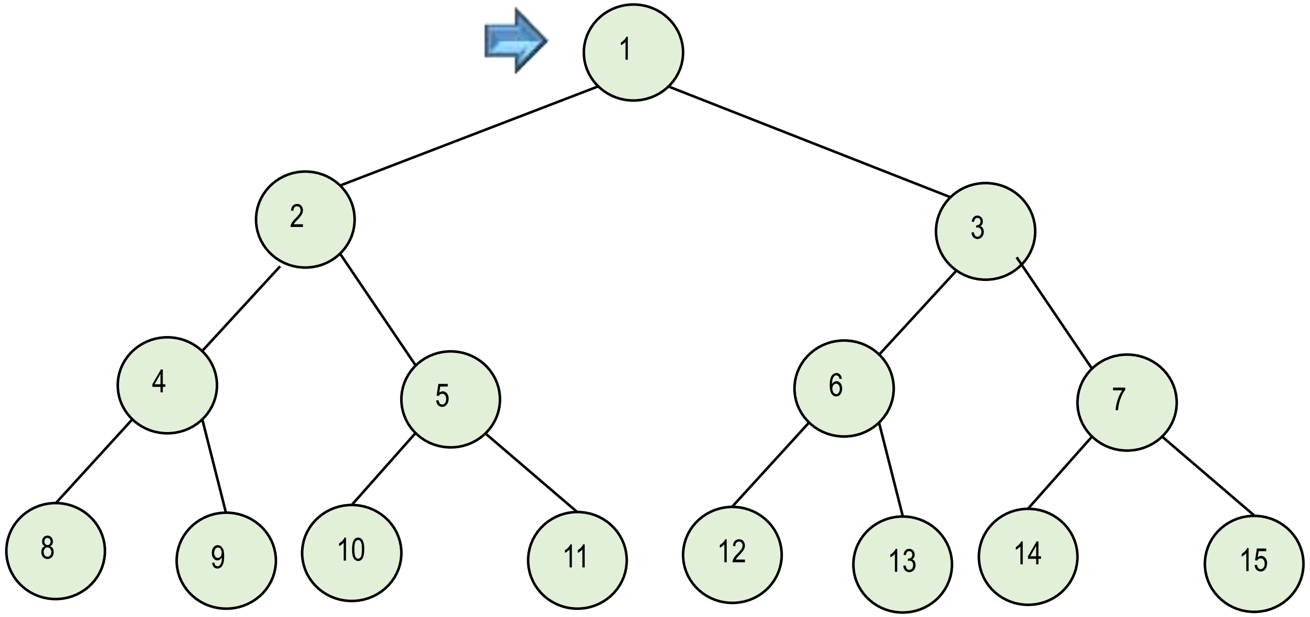
Preorder Traversal

```
template <class T>
void preOrder( Node<T> *t )
{
    if (t != NULL)
    {
        visit(t);
        preOrder(t->leftChild);
        preOrder(t->rightChild);
    }
}
void visit( Node<T> *t ) {
    t->printf( ); // display the element
}
```



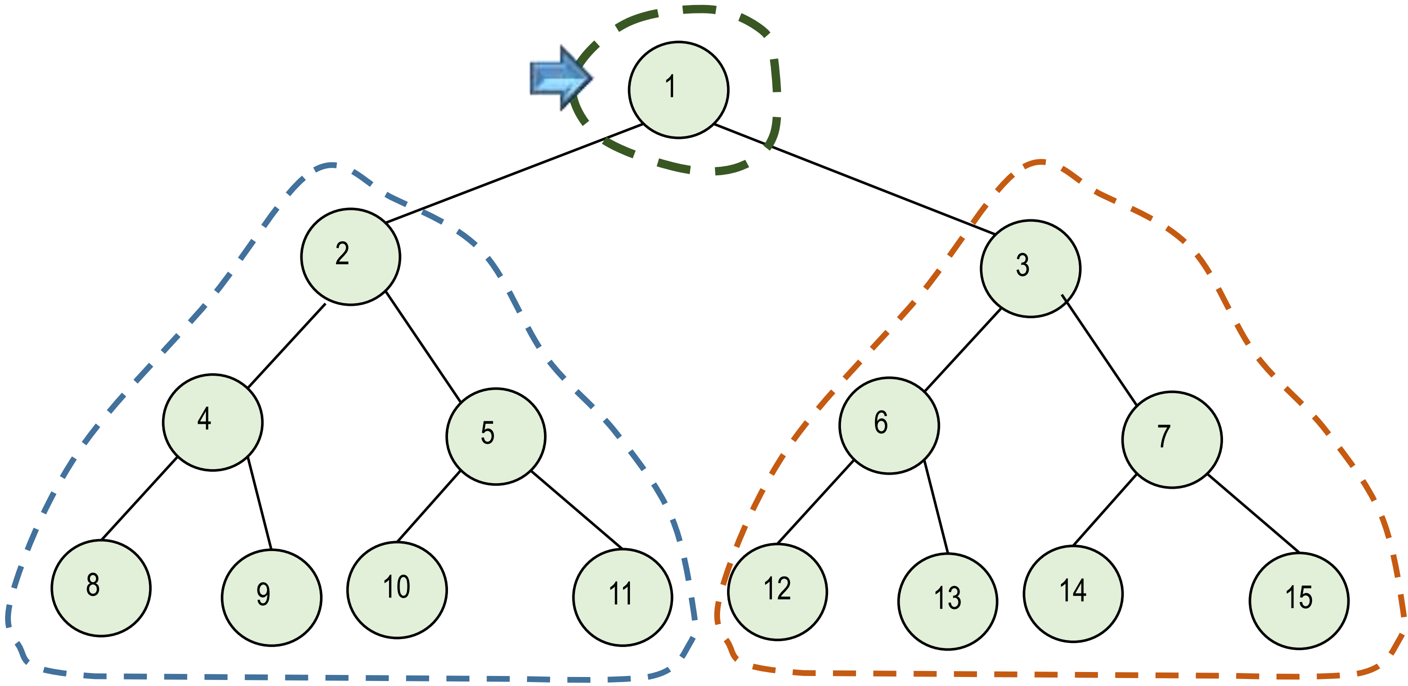
Preorder Traversal

```
template <class T>
void preOrder( Node<T> *t )
{
    if (t != NULL)
    {
        visit(t);
        preOrder(t->leftChild);
        preOrder(t->rightChild);
    }
}
void visit( Node<T> *t ) {
    t->printf( ); // display the element
}
```



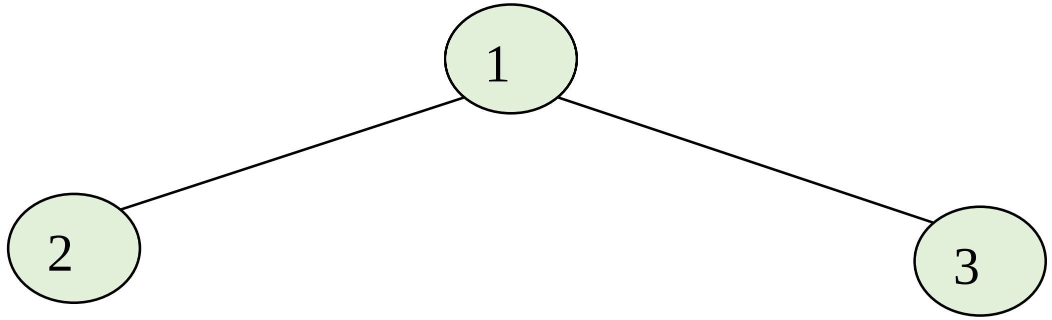
Preorder Traversal

```
template <class T>
void preOrder( Node<T> *t )
{
    if (t != NULL)
    {
        visit(t);
        preOrder(t->leftChild);
        preOrder(t->rightChild);
    }
}
void visit( Node<T> *t ) {
    t->printf( ); // display the element
}
```

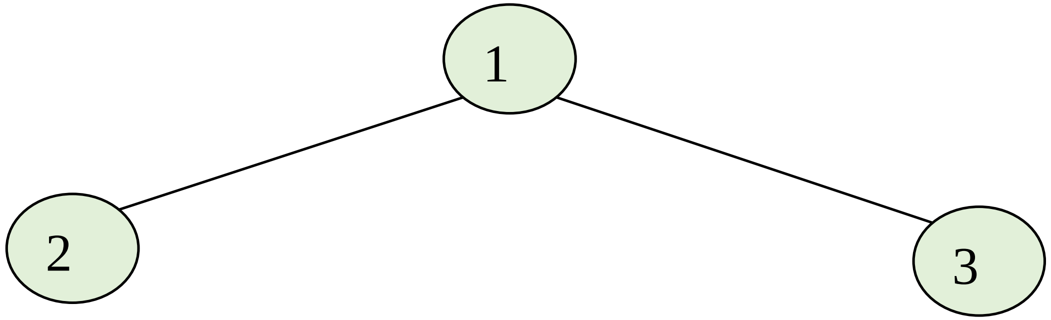


1, 2, 4, 8, 9, 5, 10, 11, 3, 6, 12, 13, 7, 14, 15

Preorder Traversal Result

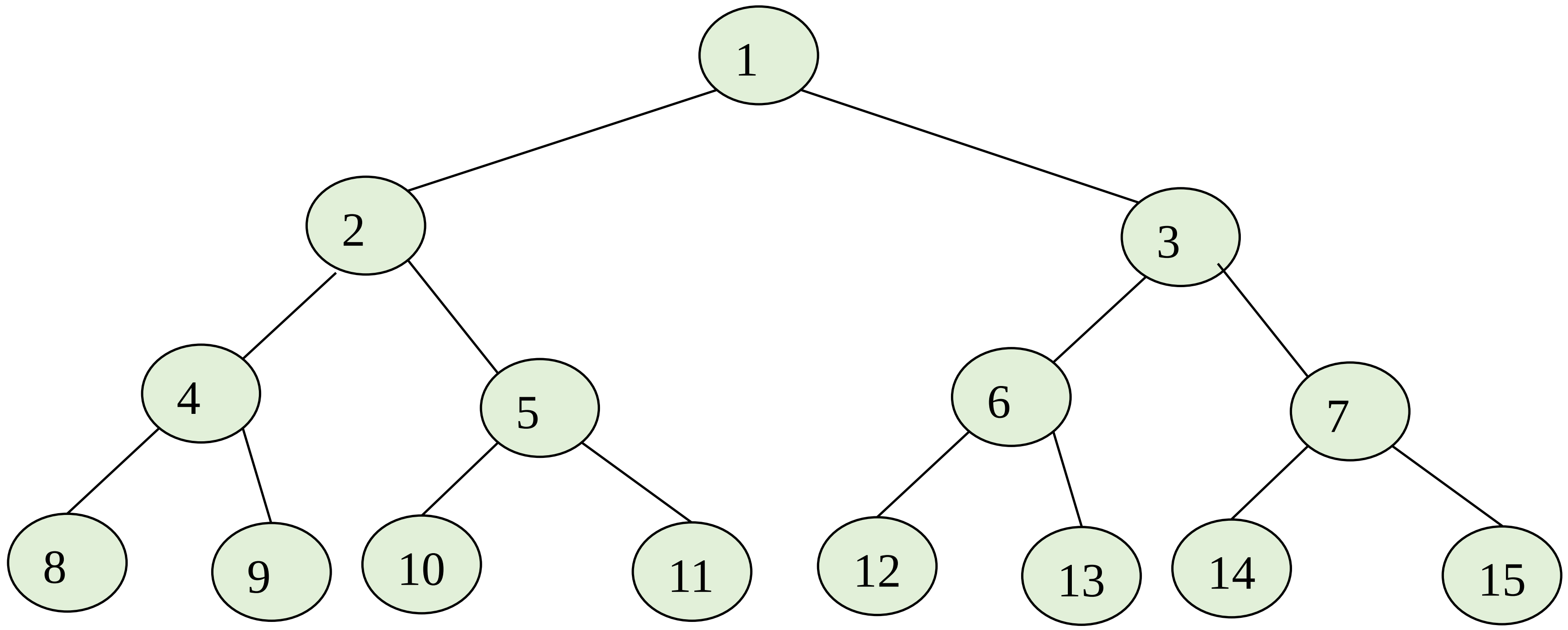


Preorder Traversal Result

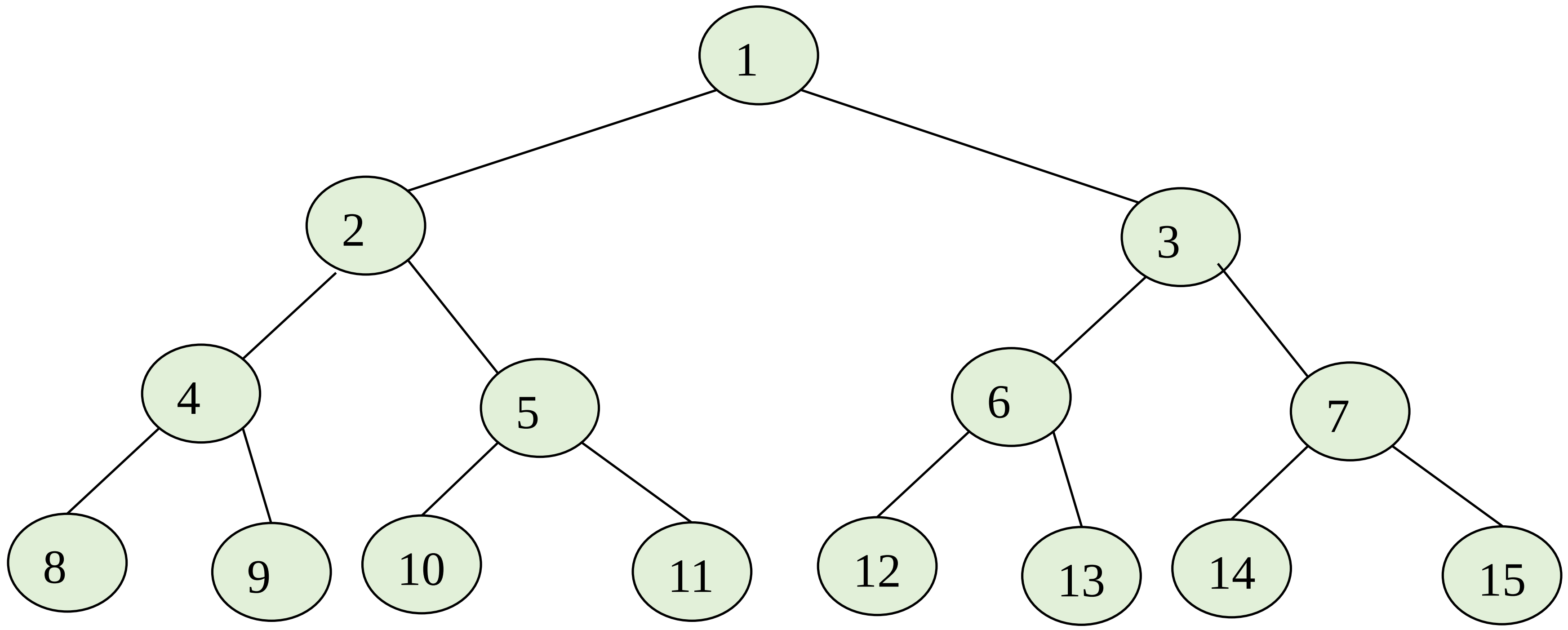


1, 2, 3

Preorder Traversal Result



Preorder Traversal Result



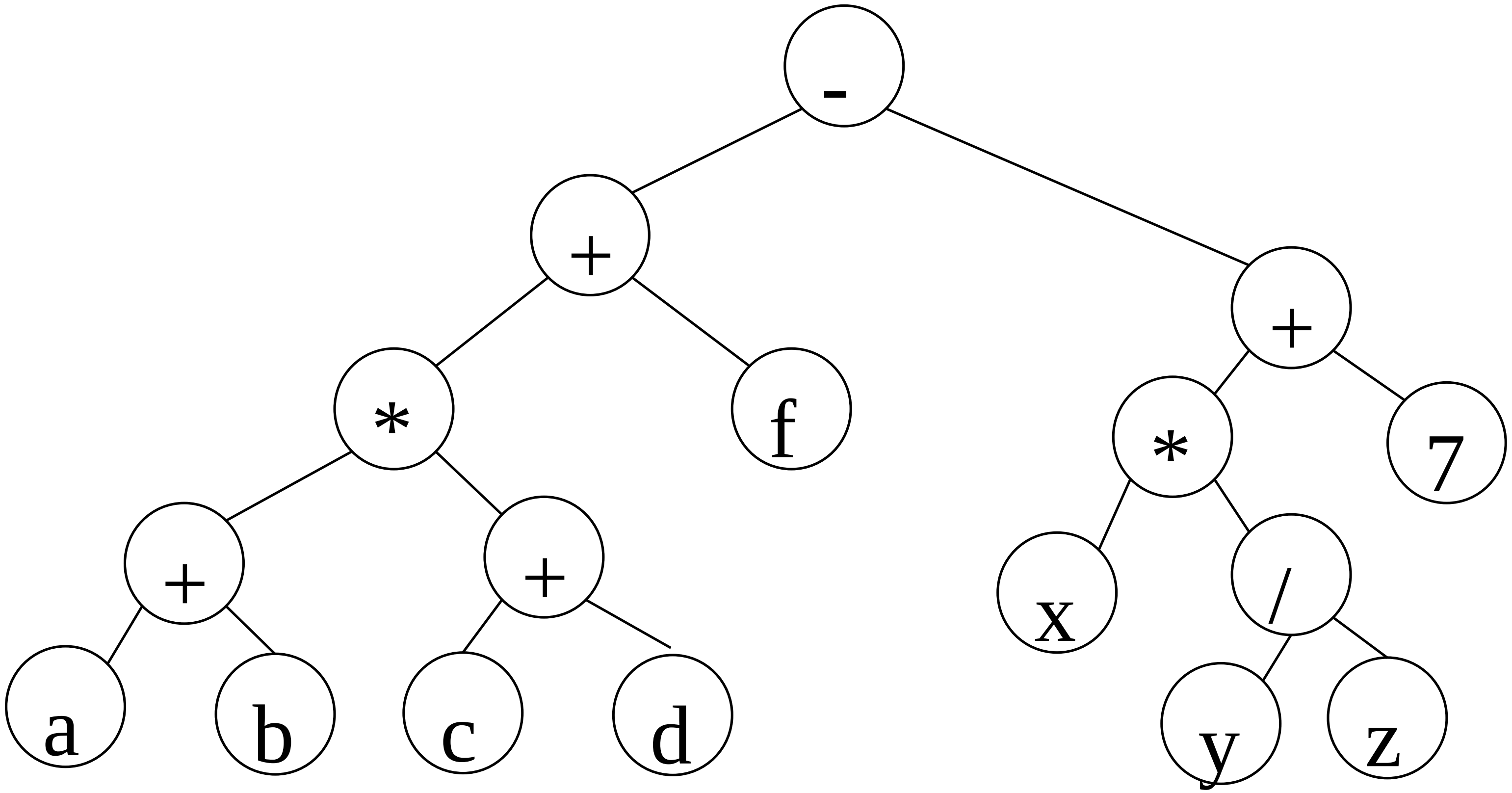
1, 2, 4, 8, 9, 5, 10, 11, 3, 6, 12, 13, 7, 14, 15

Preorder Of Expression Tree

```
Infix = (a + b) * (c + d) + f - ( x*(y/z) + 7)
```

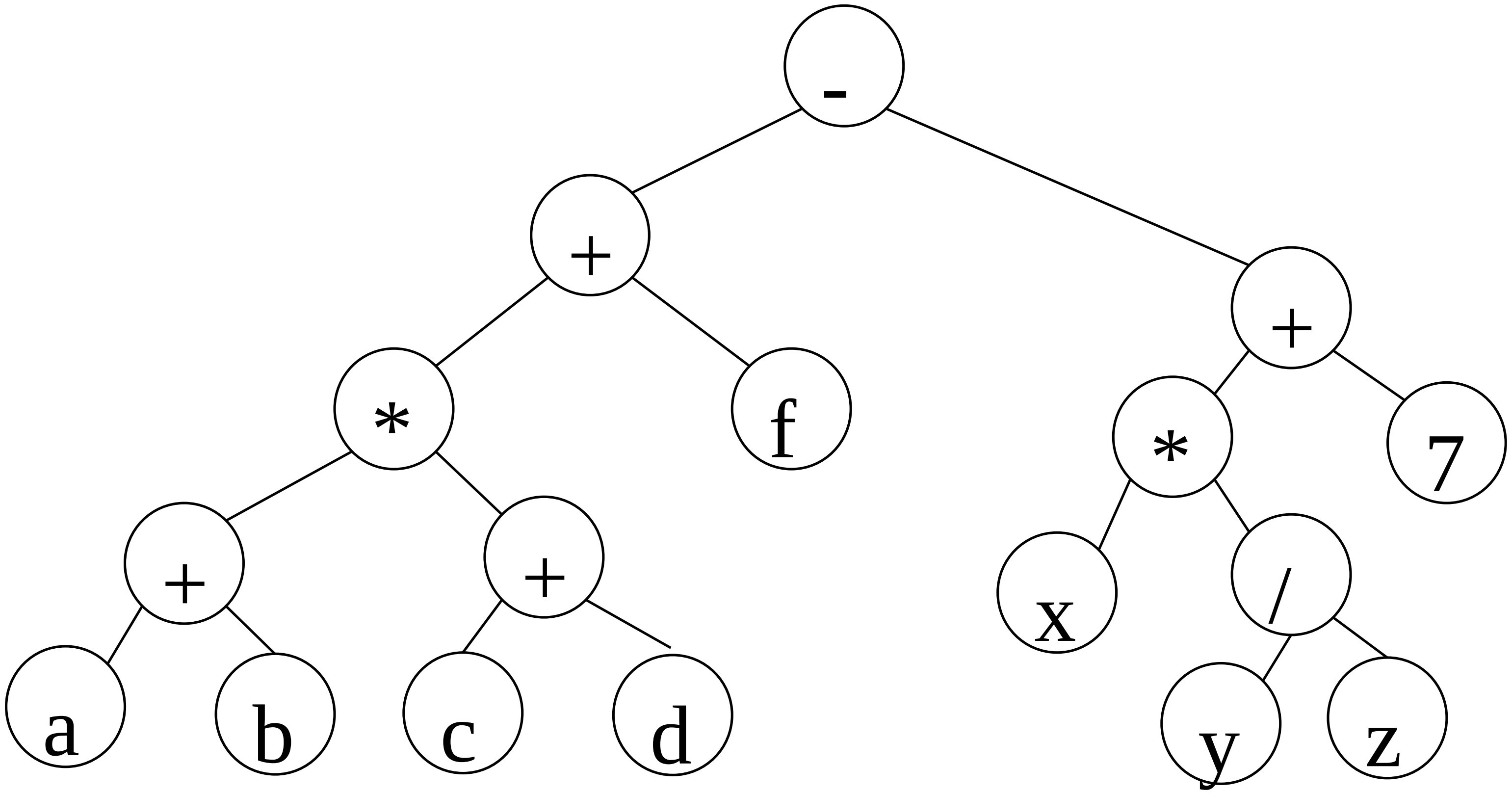
Preorder Of Expression Tree

```
Infix = (a + b) * (c + d) + f – ( x*(y/z) + 7)
```



Preorder Of Expression Tree

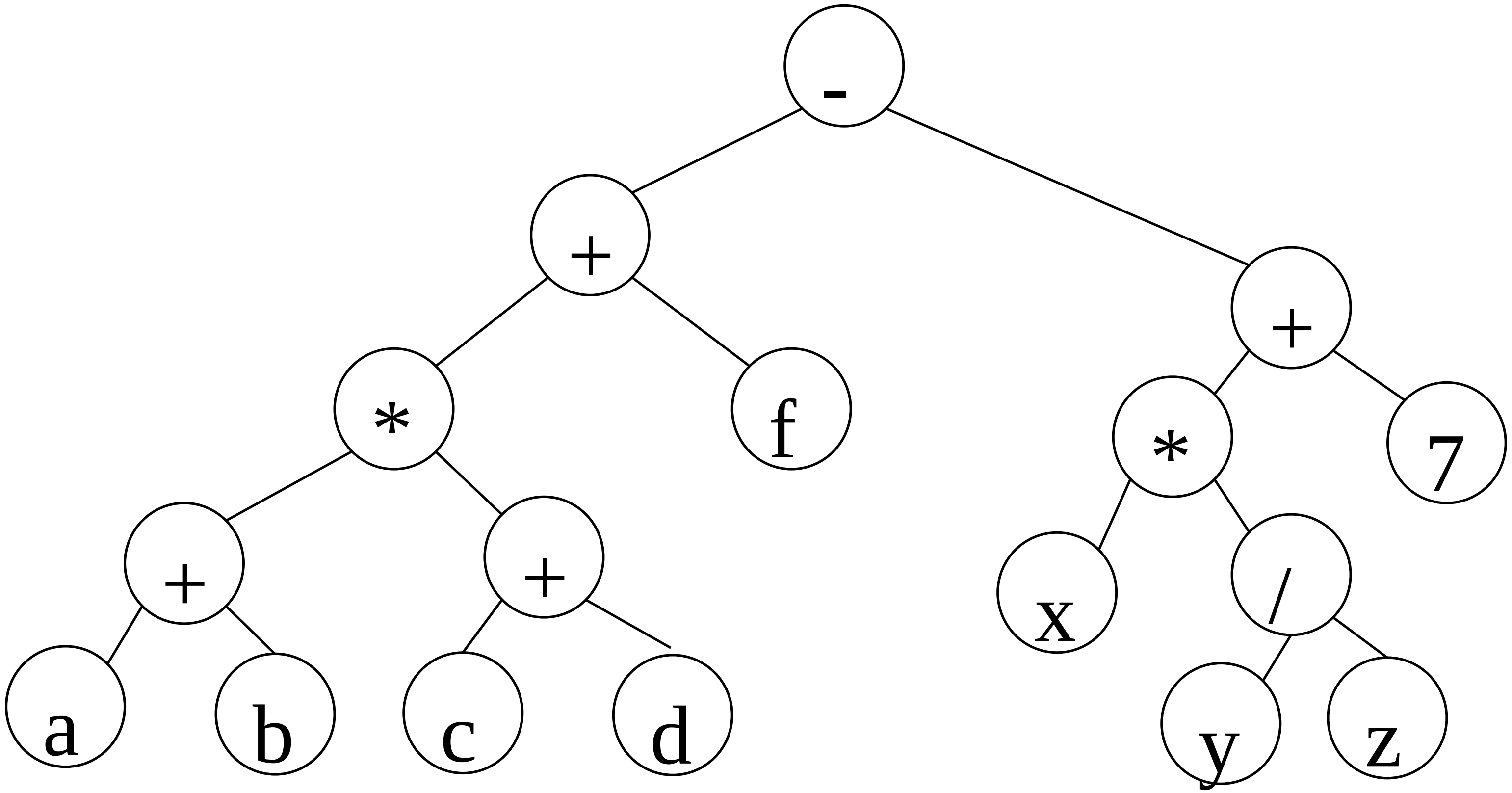
`Infix = (a + b) * (c + d) + f – ( x*(y/z) + 7)`



`the prefix form expression?`

Preorder Of Expression Tree

`Infix = (a + b) * (c + d) + f – ( x*(y/z) + 7)`

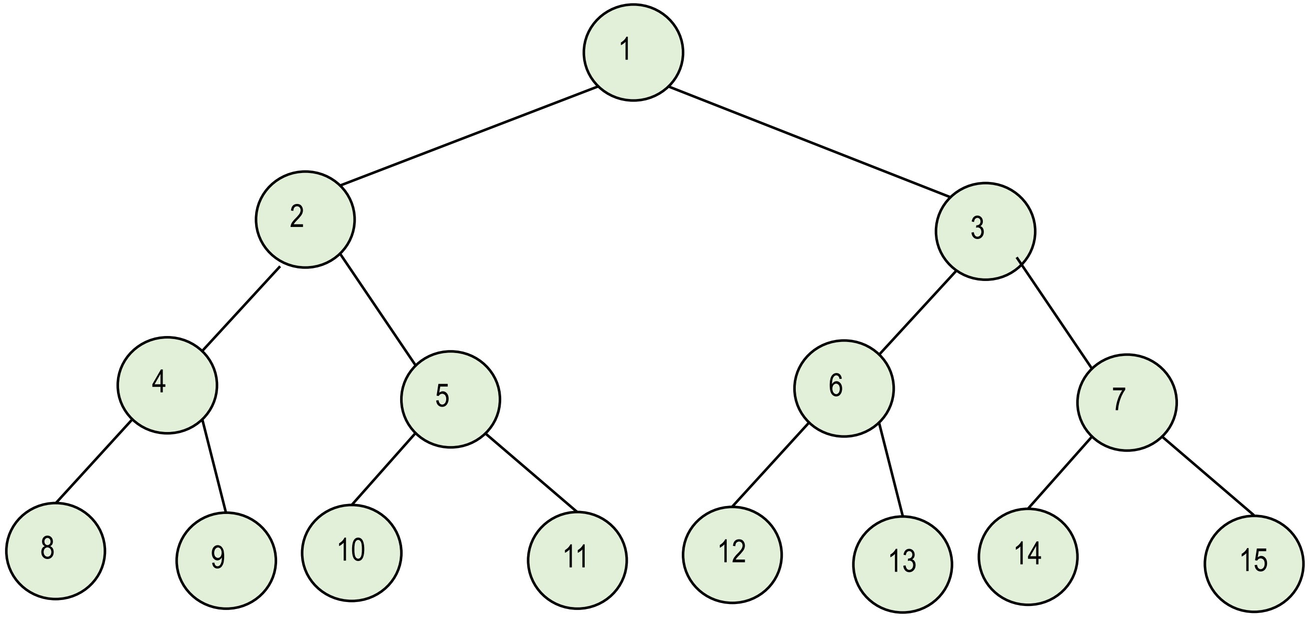


`–+*+ab+cdf+*x/yz7`

`the prefix form expression?`

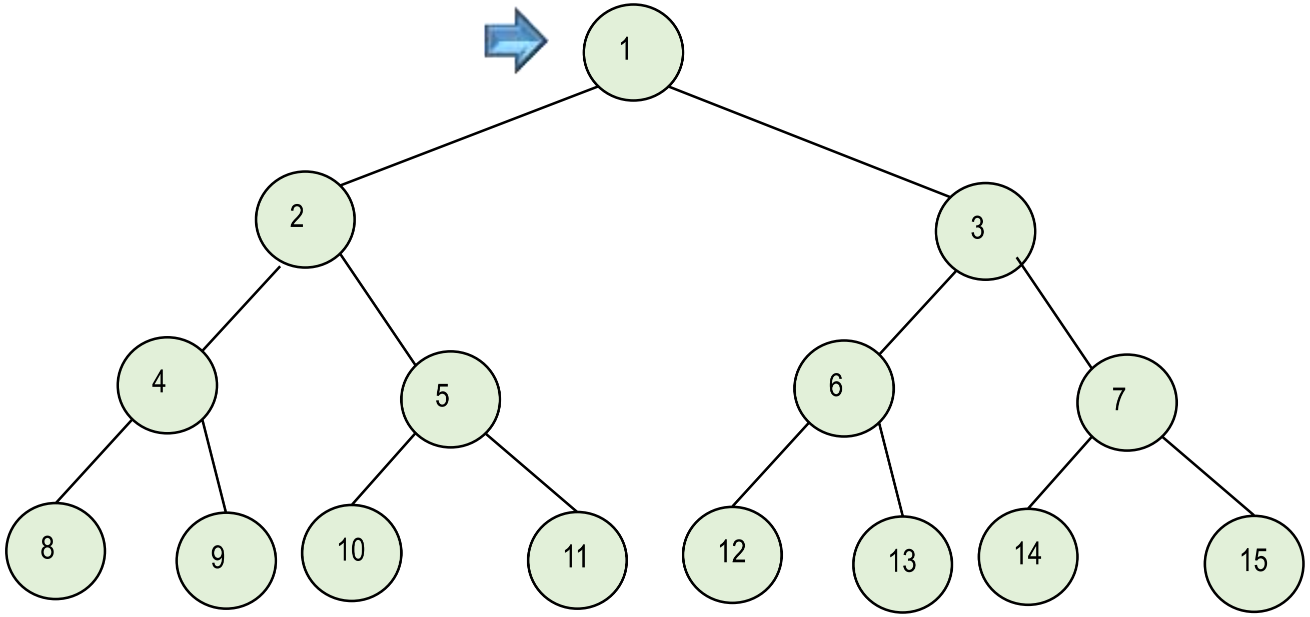
Inorder Traversal

```
template <class T>
void inOrder(Node<T> *t)
{
    if (t != NULL)
    {
        inOrder(t->leftChild);
        visit(t);
        inOrder(t->rightChild);
    }
}
```



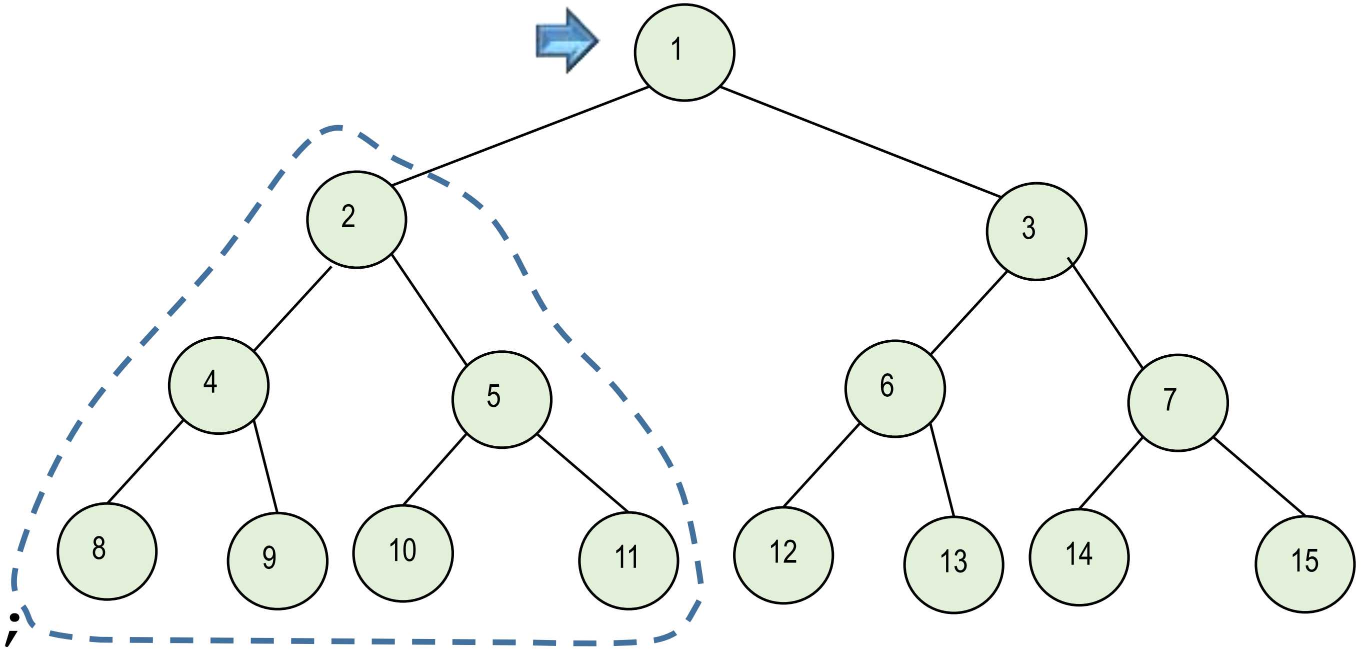
Inorder Traversal

```
template <class T>
void inOrder(Node<T> *t)
{
    if (t != NULL)
    {
        inOrder(t->leftChild);
        visit(t);
        inOrder(t->rightChild);
    }
}
```



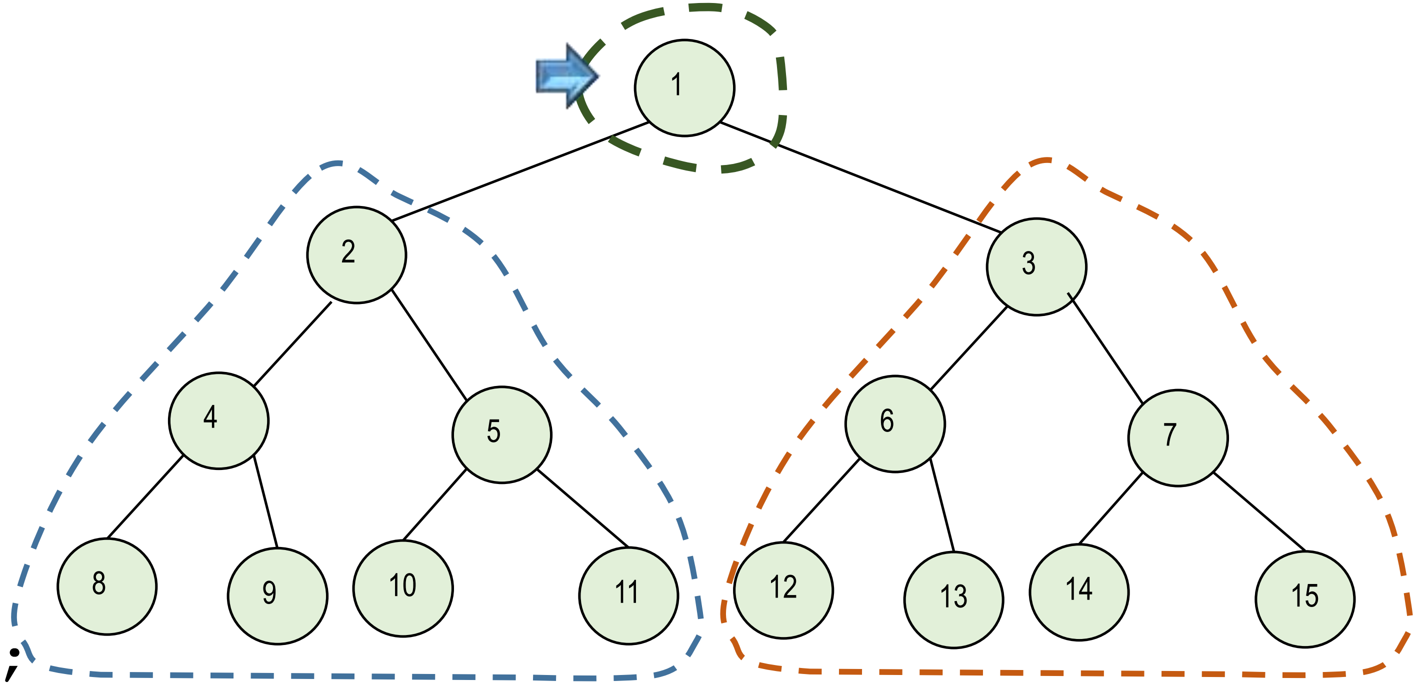
Inorder Traversal

```
template <class T>
void inOrder(Node<T> *t)
{
    if (t != NULL)
    {
        inOrder(t->leftChild);
        visit(t);
        inOrder(t->rightChild);
    }
}
```



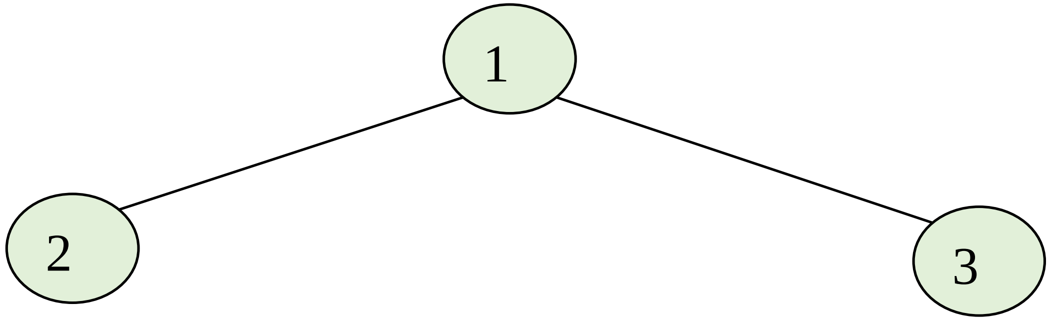
Inorder Traversal

```
template <class T>
void inOrder(Node<T> *t)
{
    if (t != NULL)
    {
        inOrder(t->leftChild);
        visit(t);
        inOrder(t->rightChild);
    }
}
```

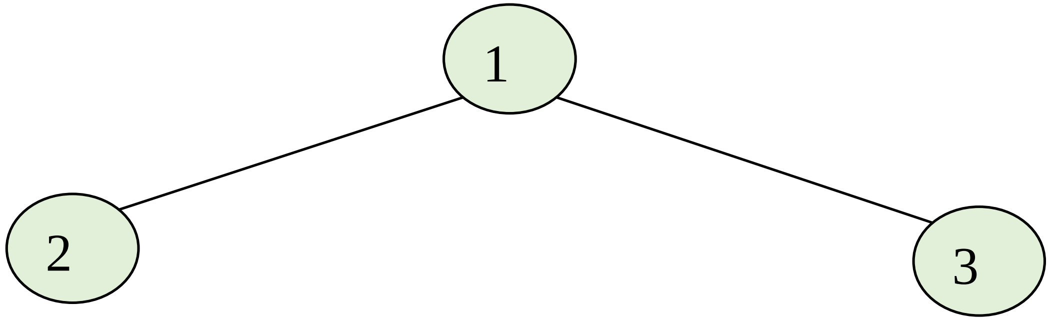


8,4,9,2,10,5,11,1,12,6,13,3,14,7,15

Inorder Traversal Result

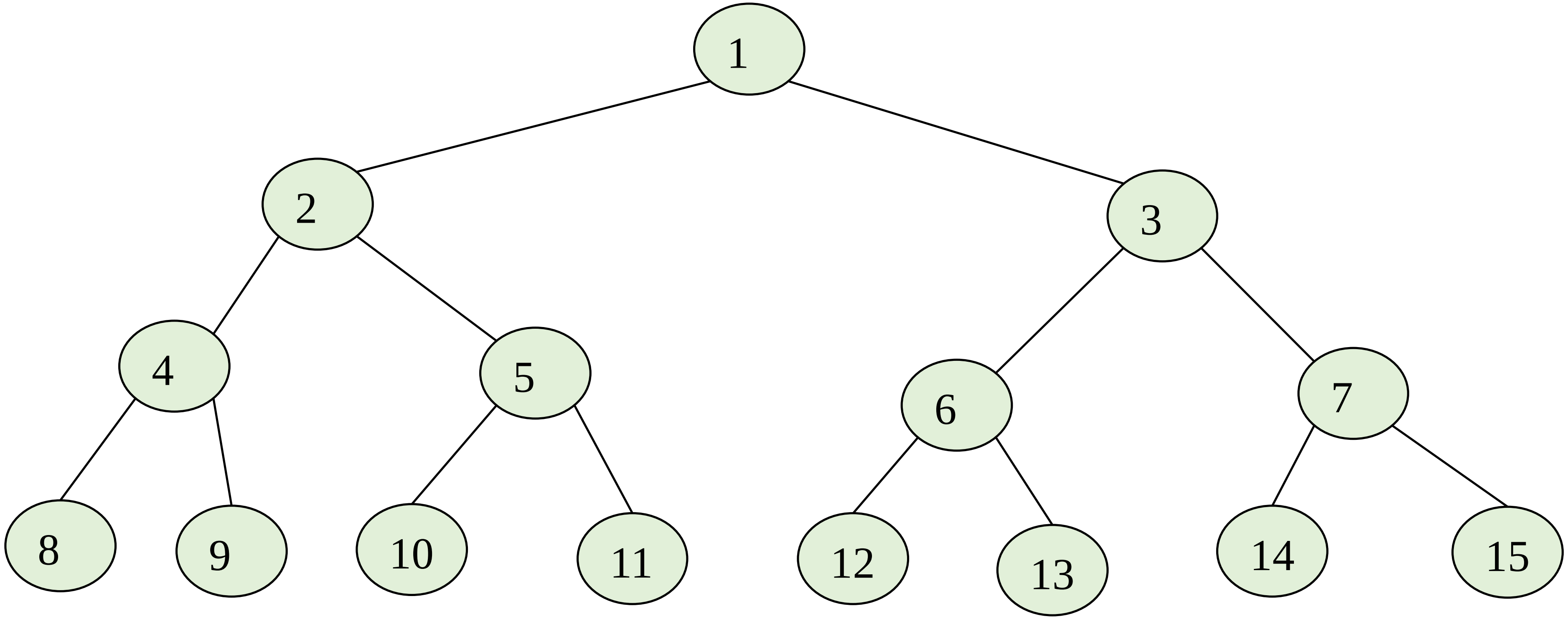


Inorder Traversal Result

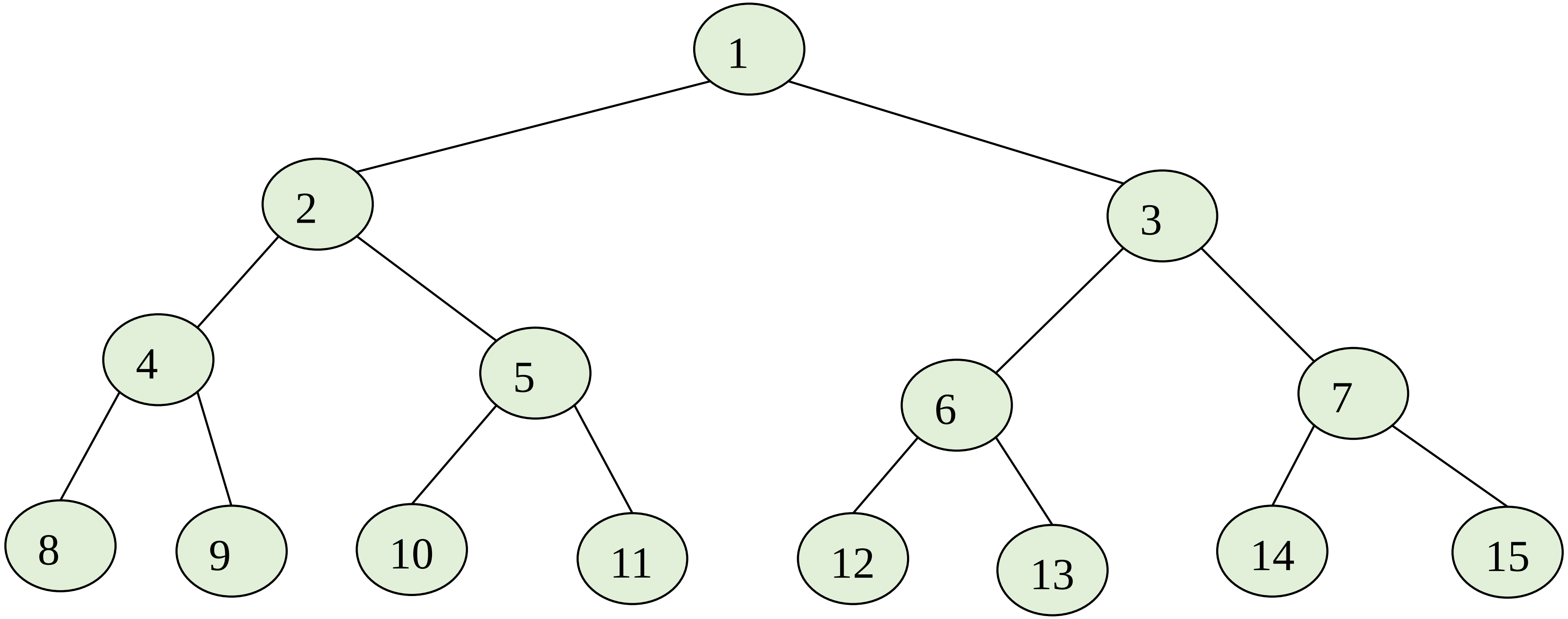


2,1,3

Inorder Traversal Result

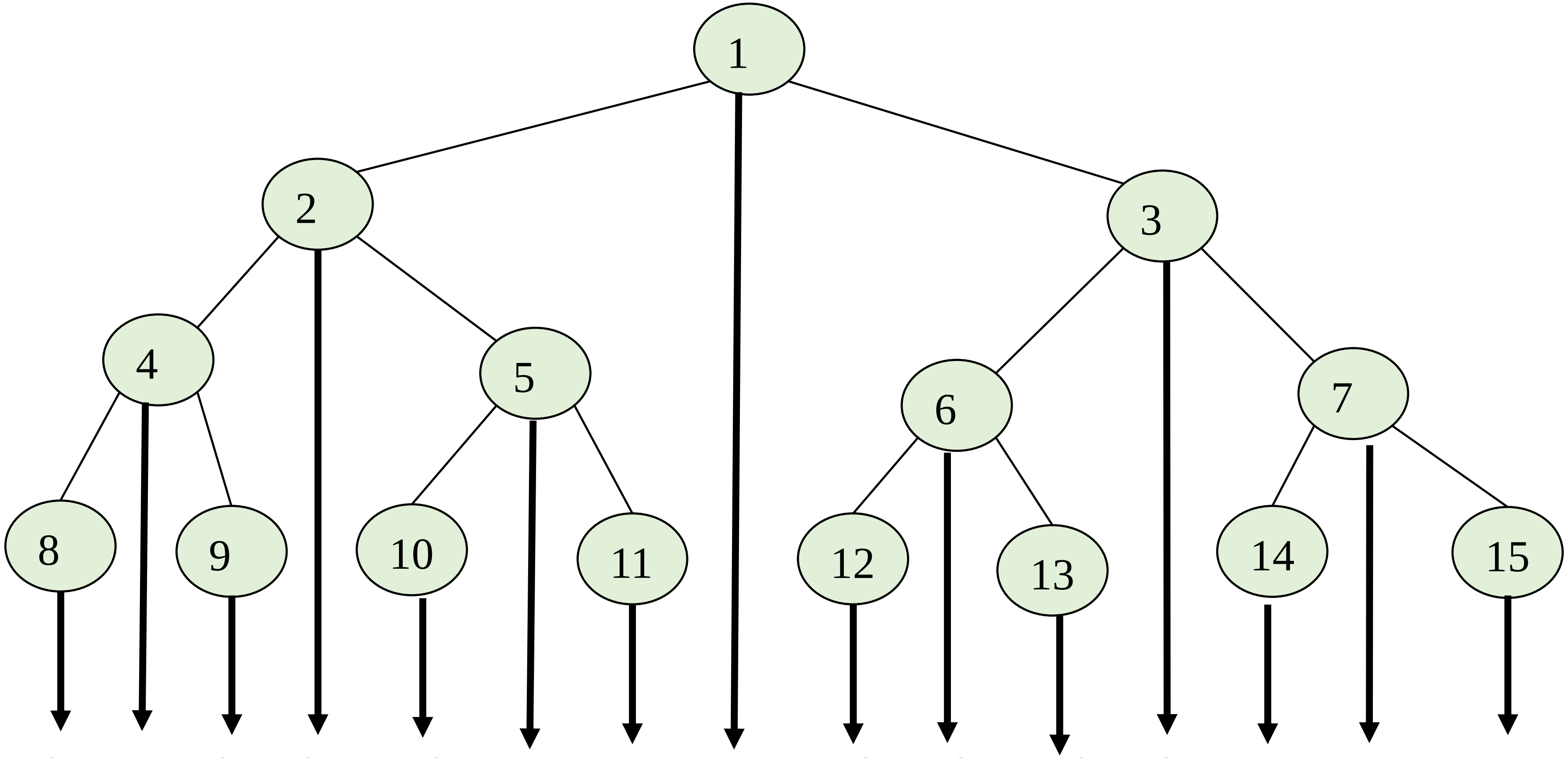


Inorder Traversal Result



8,4,9,2,10,5,11,1,12,6,13,3,14,7,15

Inorder Traversal Result

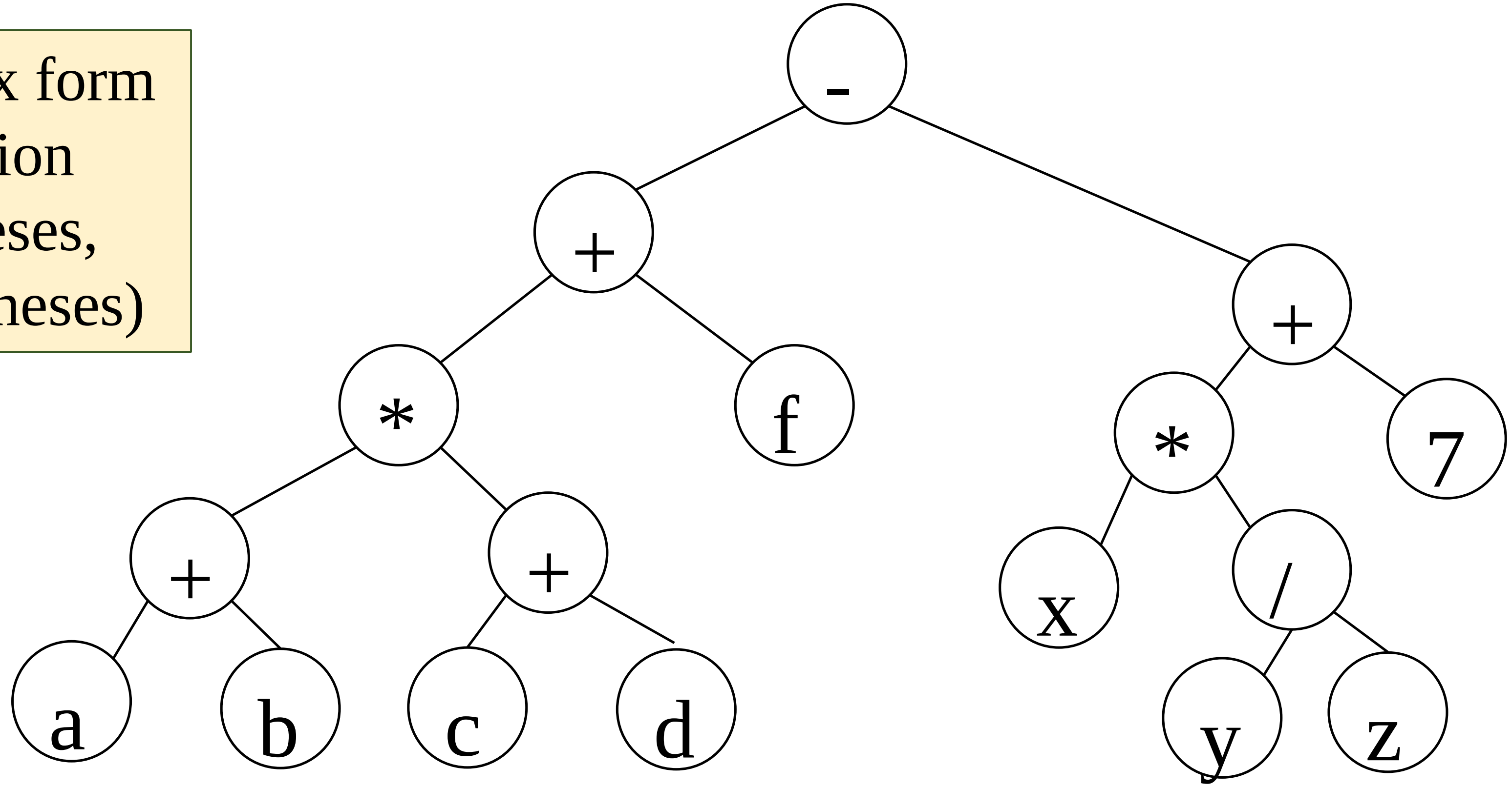


8,4,9,2,10,5,11,1,12,6,13,3,14,7,15

Inorder Of Expression Tree

`Infix = (a + b) * (c + d) + f − ( x*(y/z) + 7)`

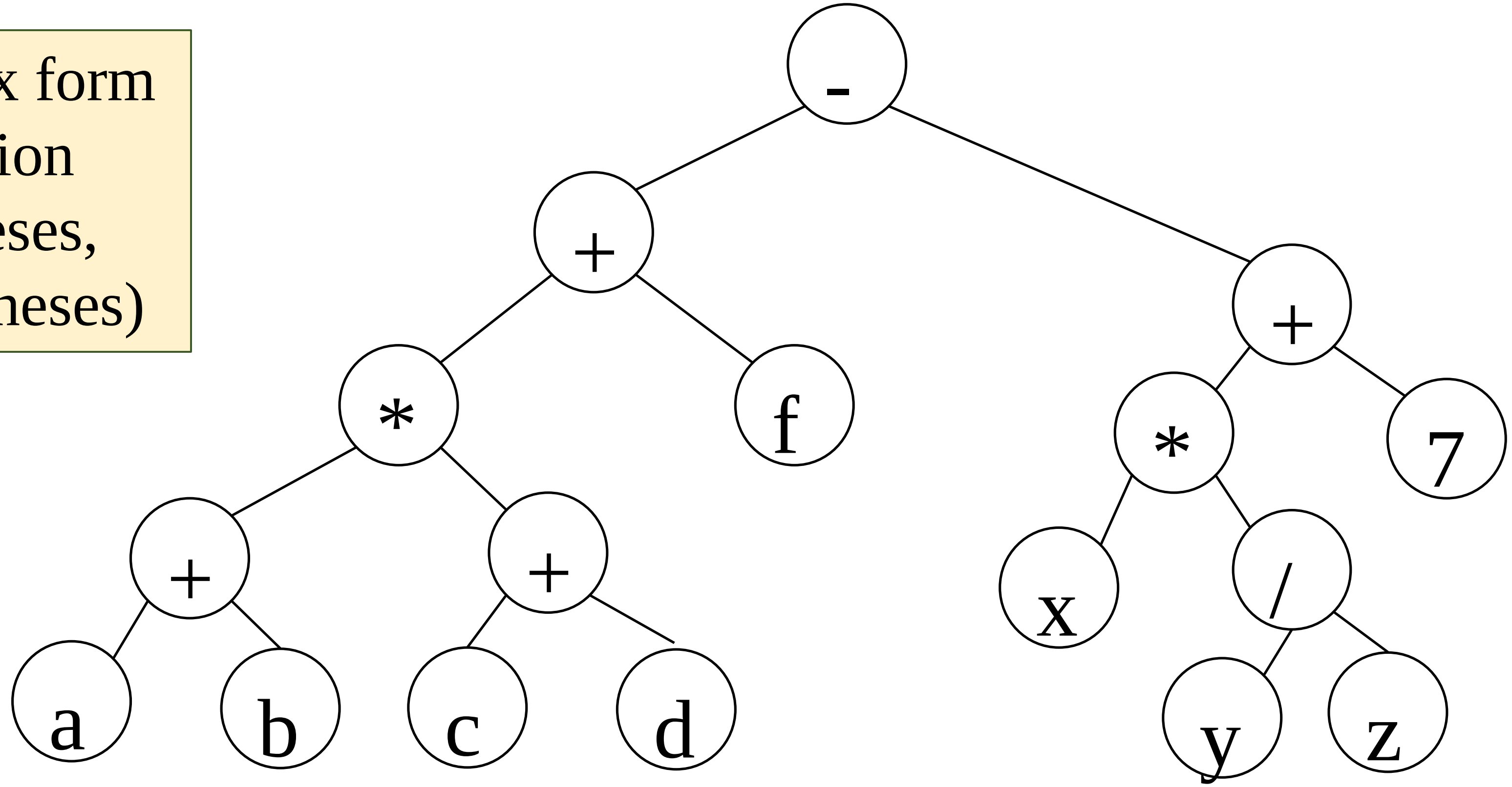
Gives the infix form of the expression (sans parentheses, i.e., no parentheses)



Inorder Of Expression Tree

`Infix = (a + b) * (c + d) + f – ( x*(y/z) + 7)`

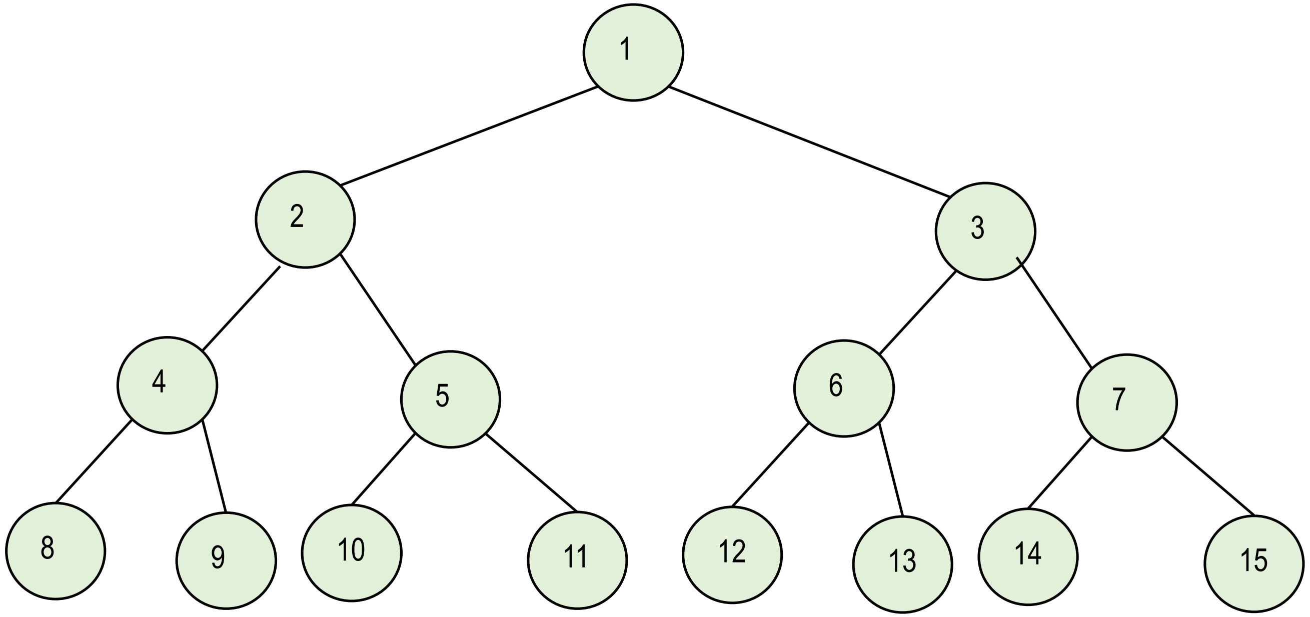
Gives the infix form of the expression (sans parentheses, i.e., no parentheses)



a+b * **c+d** + f – x ***y/z** + 7

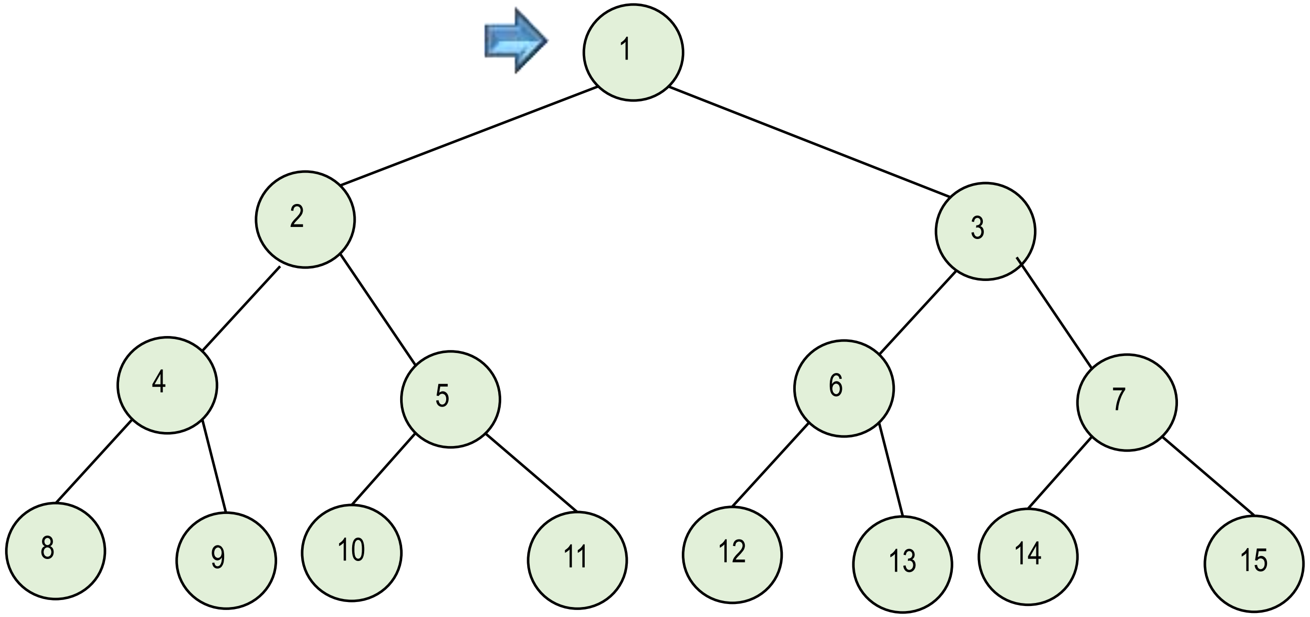
Postorder Traversal

```
template <class T>
void postOrder(Node<T> *t)
{
  if (t == NULL) return;
  postOrder(t->leftChild);
  postOrder(t->rightChild);
  visit(t);
}
```



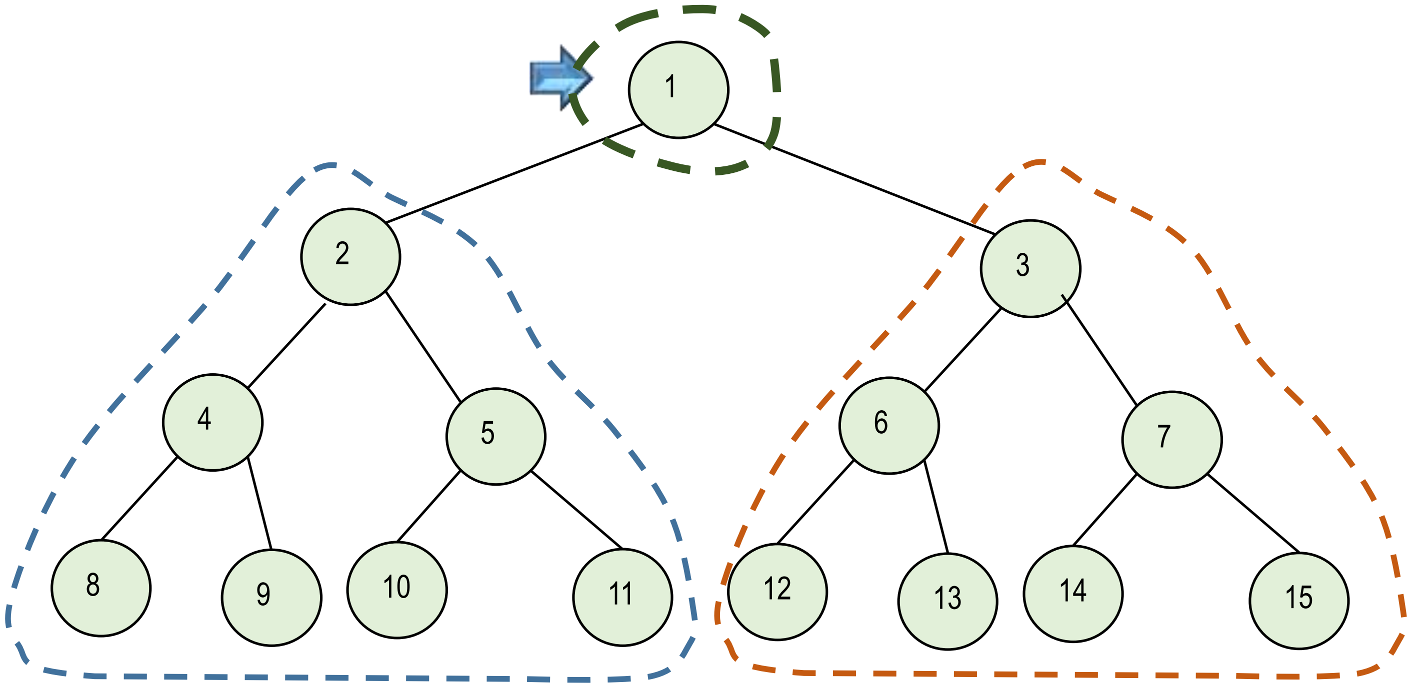
Postorder Traversal

```
template <class T>
void postOrder(Node<T> *t)
{
   if (t == NULL) return;
   postOrder(t->leftChild);
   postOrder(t->rightChild);
   visit(t);
}
```



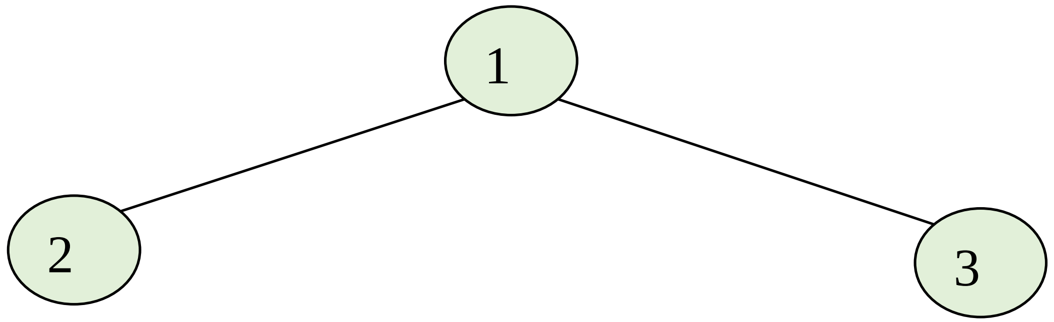
Postorder Traversal

```
template <class T>
void postOrder(Node<T> *t)
{
  if (t == NULL) return;
  postOrder(t->leftChild);
  postOrder(t->rightChild);
  visit(t);
}
```

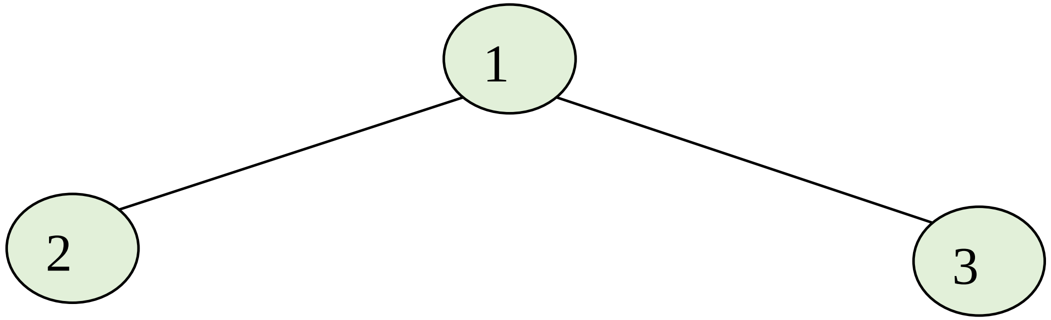


8,9,4,10,11,5,2,12,13,6,14,15,7,3,1

Post-order Traversal Result

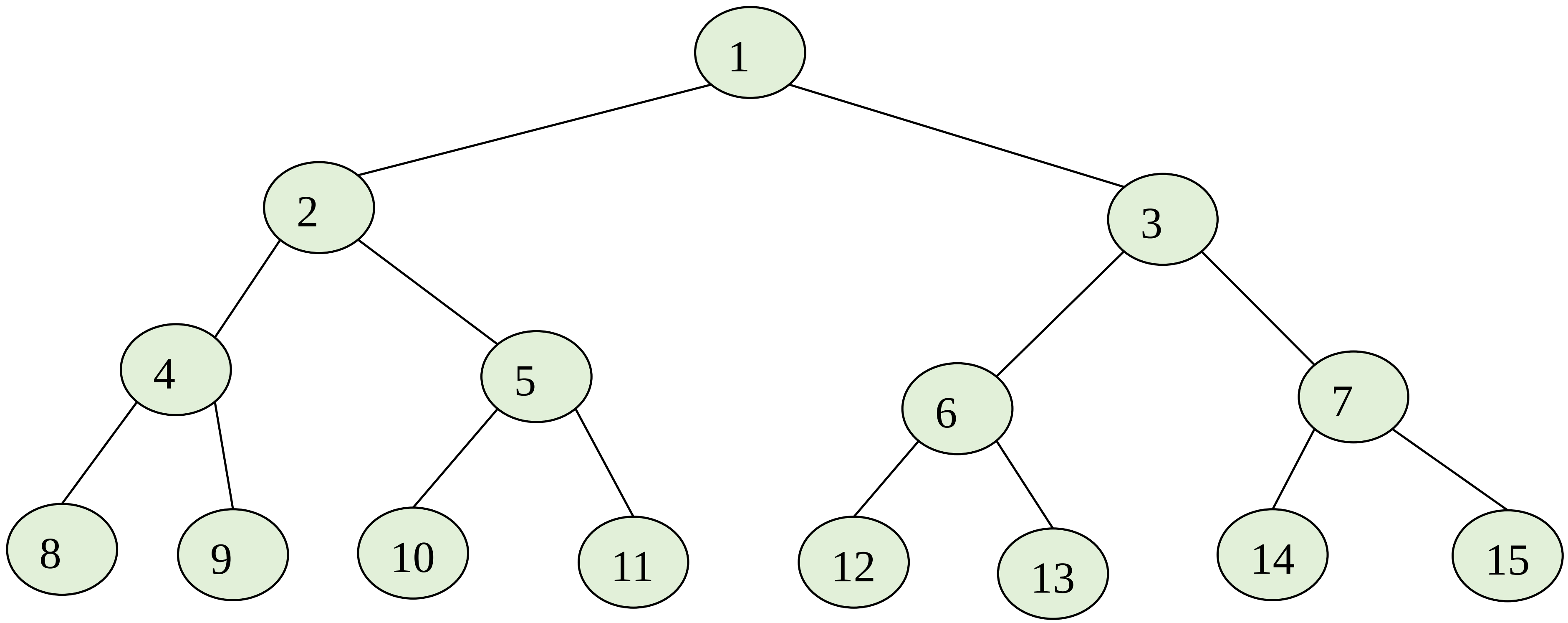


Post-order Traversal Result

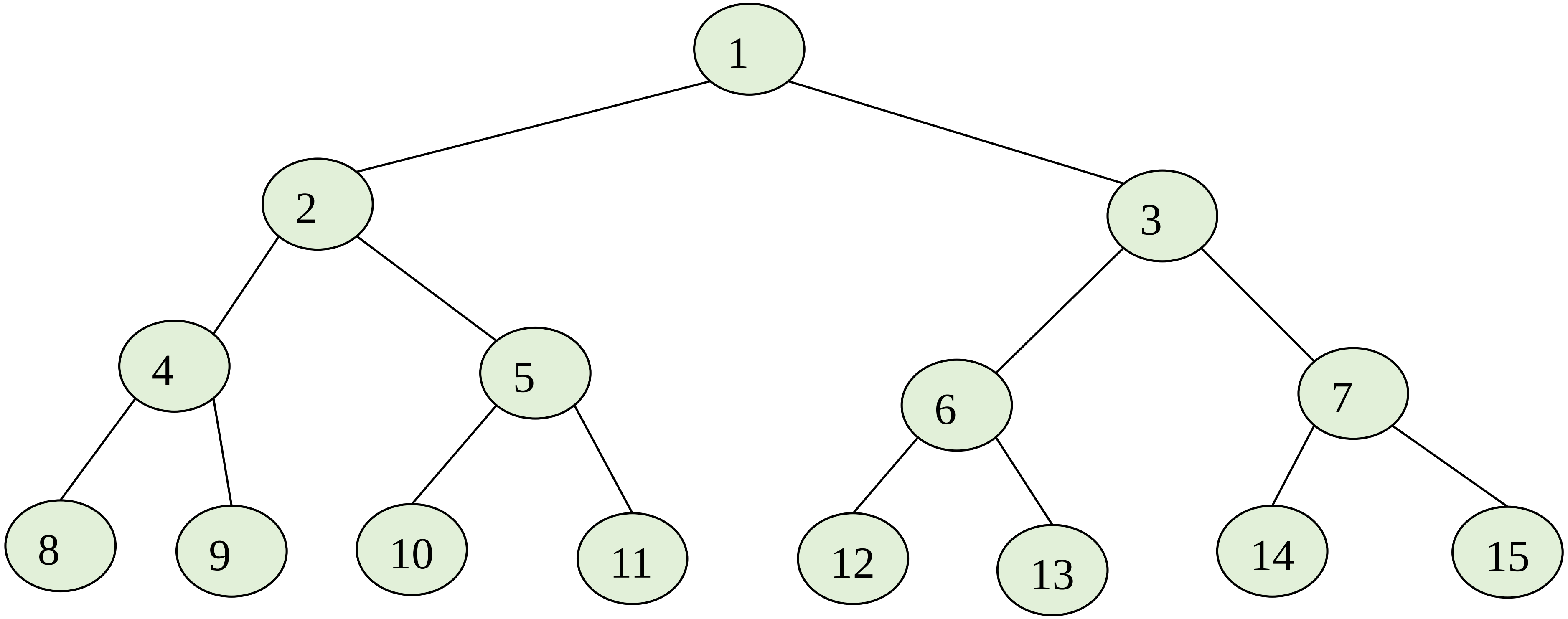


2, 3, 1

Post-order Traversal Result



Post-order Traversal Result

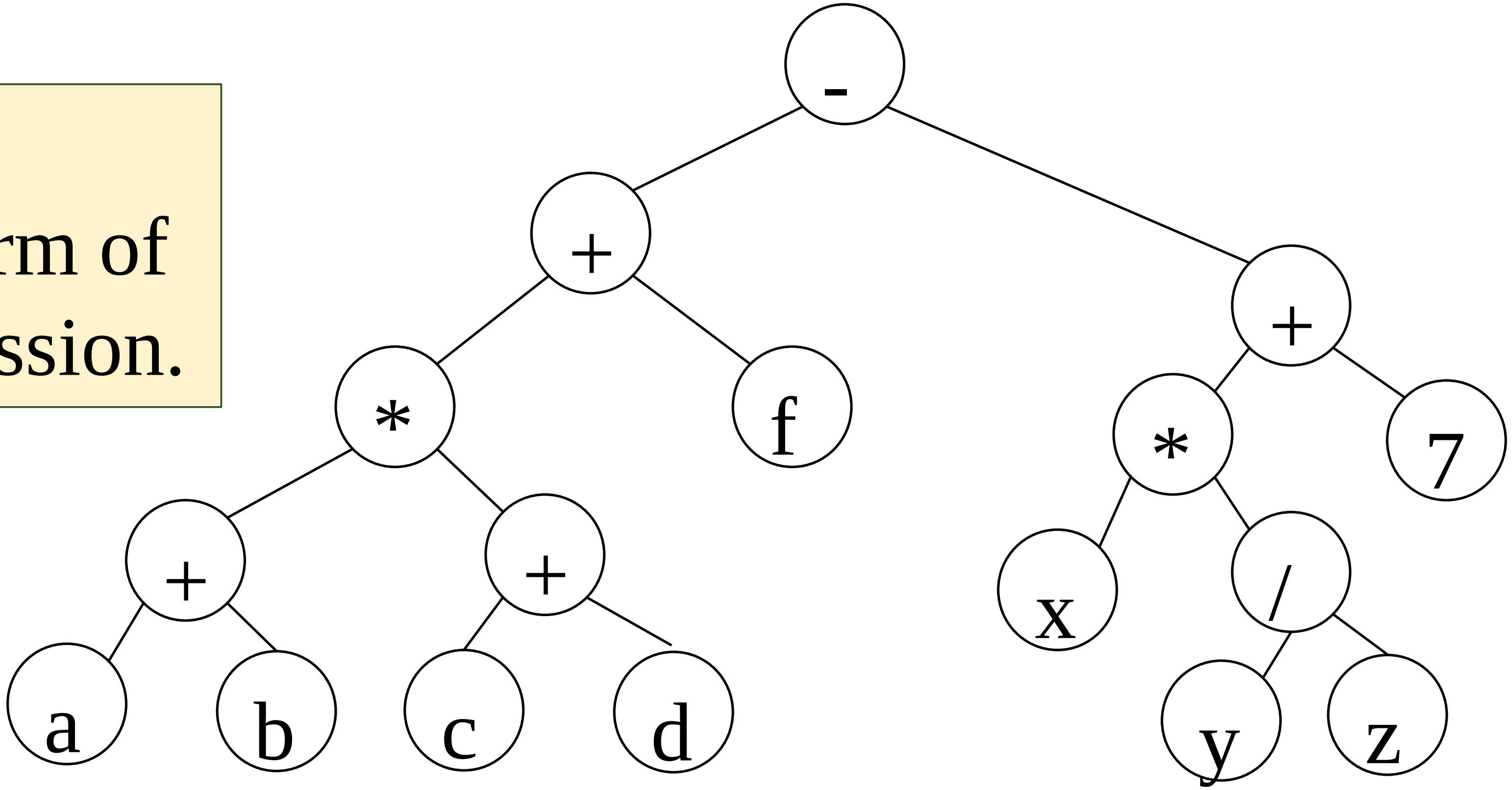


8,9,4,10,11,5,2,12,13,6,14,15,7,3,1

Postorder Of Expression Tree

Infix = (a + b) * (c + d) + f − (x*(y/z) + 7)

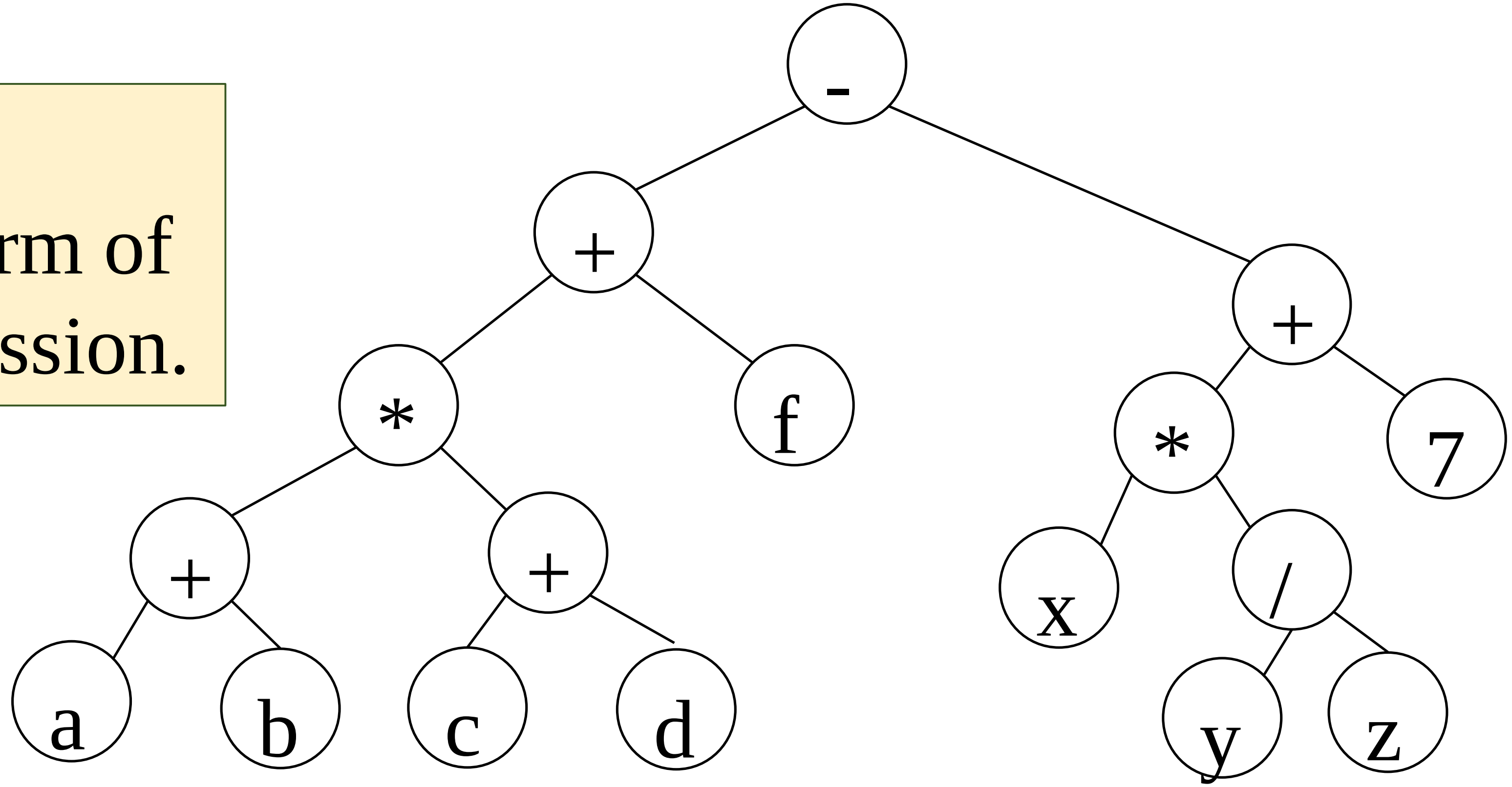
Gives the postfix form of the expression.



Postorder Of Expression Tree

Infix = (a + b) * (c + d) + f – (x*(y/z) + 7)

Gives the postfix form of the expression.



ab+cd+*f+xyz/*7+–

Traversal Applications

- Make a clone.
- Determine height.
- Determine number of nodes.

Level Order Traversal

unmark all the nodes

q = { t }   // q is a queue (FIFO)

mark t

while (q != { })

{

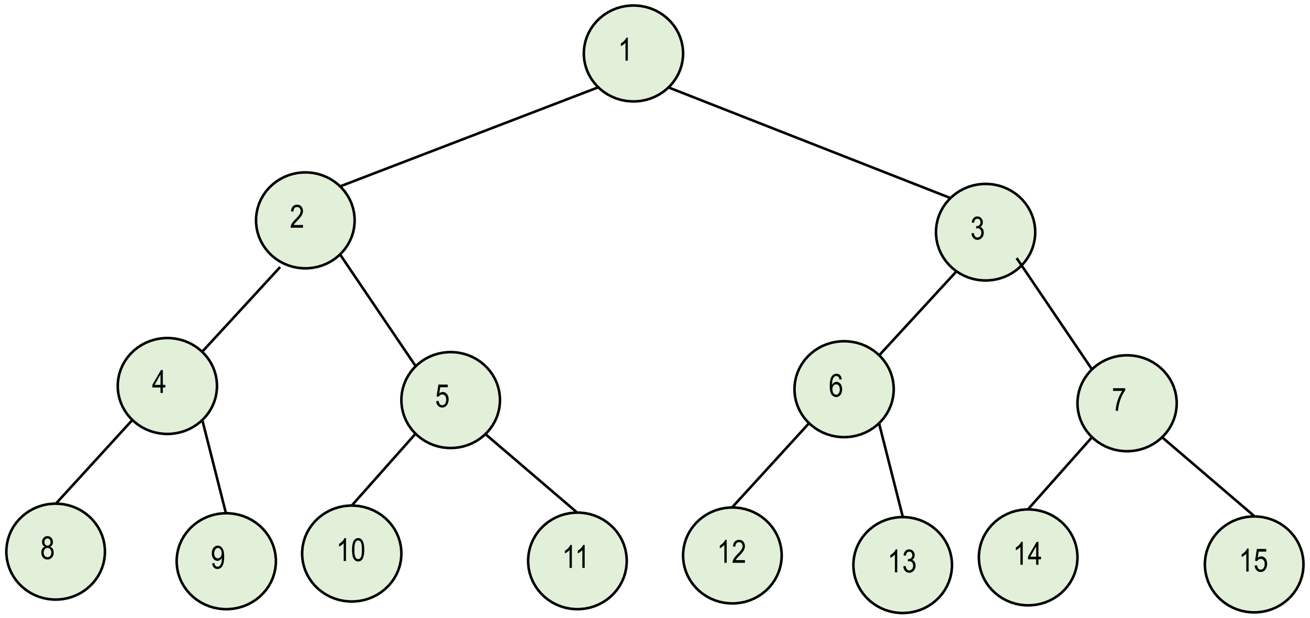
    t ← pop from q

    visit t

    put its unmarked children (from left to right) on q

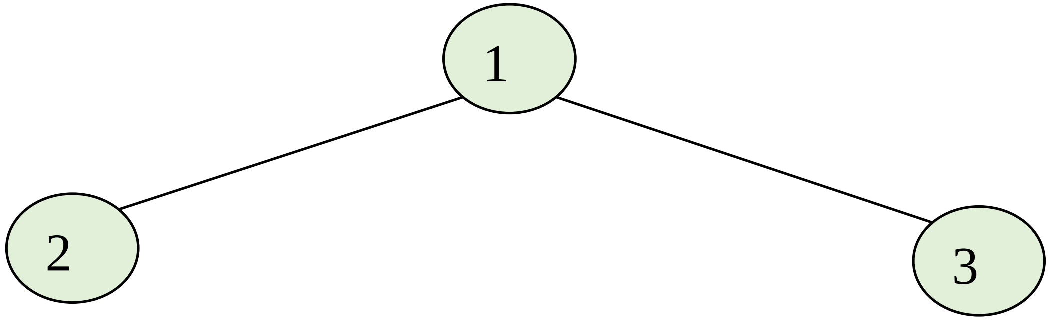
    and mark them

}

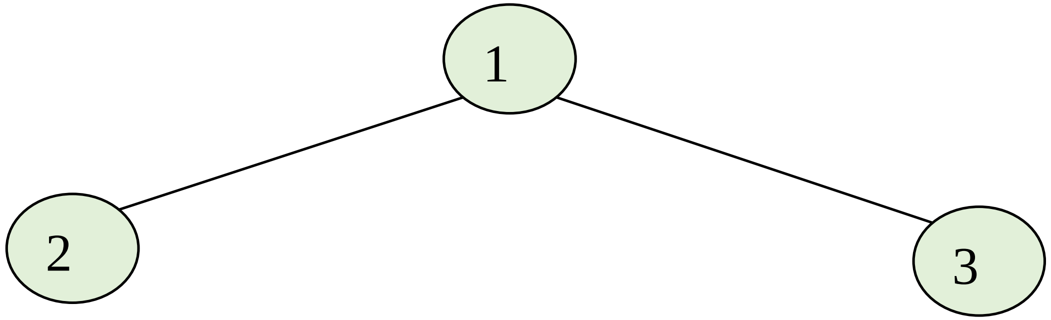


1,2,3,4,5,6,7,8,9,10,11,12,13,14,15

Level-order Traversal Result

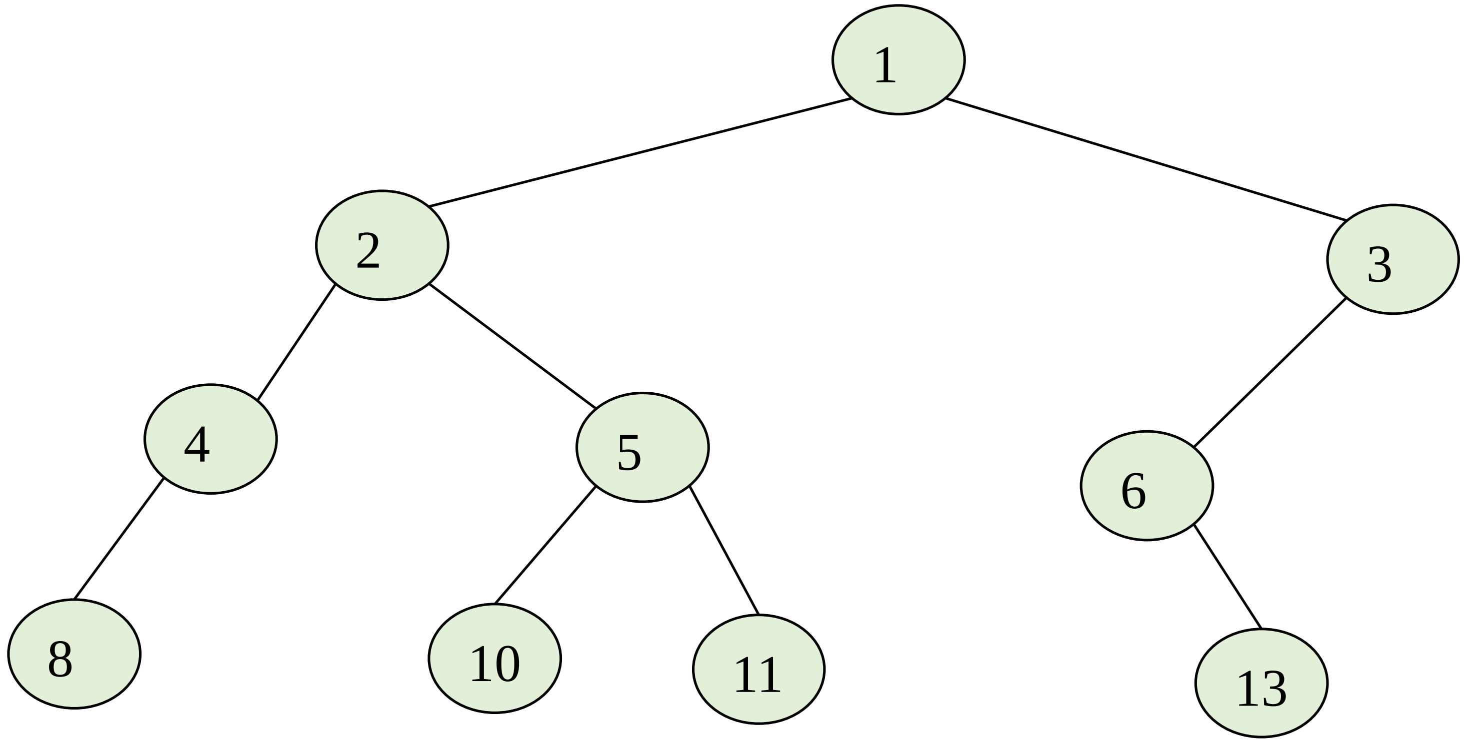


Level-order Traversal Result

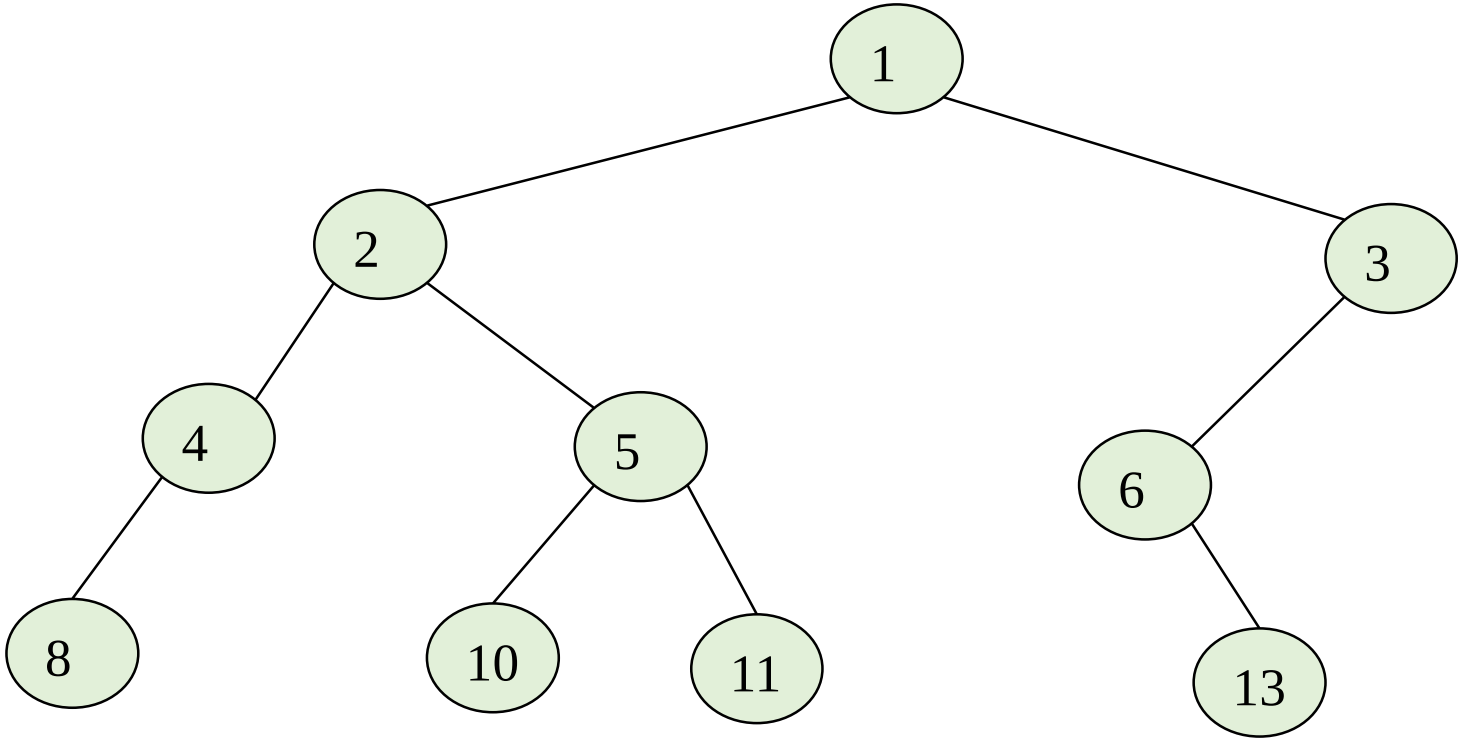


1,2,3

Level-order Traversal Result



Level-order Traversal Result



1,2,3,4,5,6,8,10,11,13

Binary Tree Construction

preorder = ab

inorder = ab

postorder = ab

level-order = ab

Binary Tree Construction

- Suppose that the elements in a binary tree are distinct.
- Given some traversal sequences, we may be able to construct one or more binary trees.
- Based on some traversal sequences, we can construct a unique binary tree.

preorder = ab

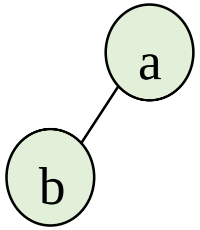
inorder = ab

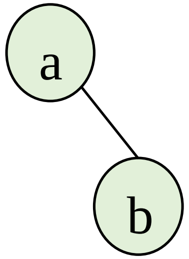
postorder = ab

level-order = ab

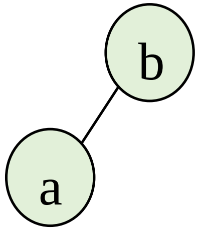
Binary Tree Construction

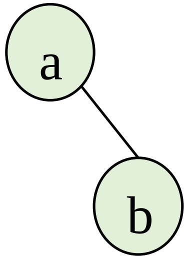
preorder = ab



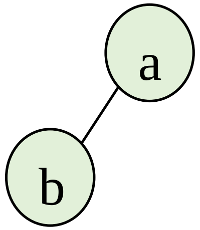


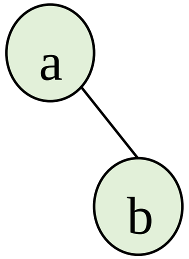
inorder = ab



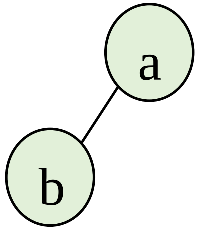


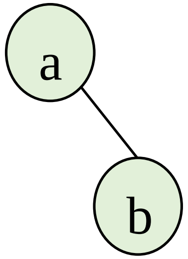
postorder = ab



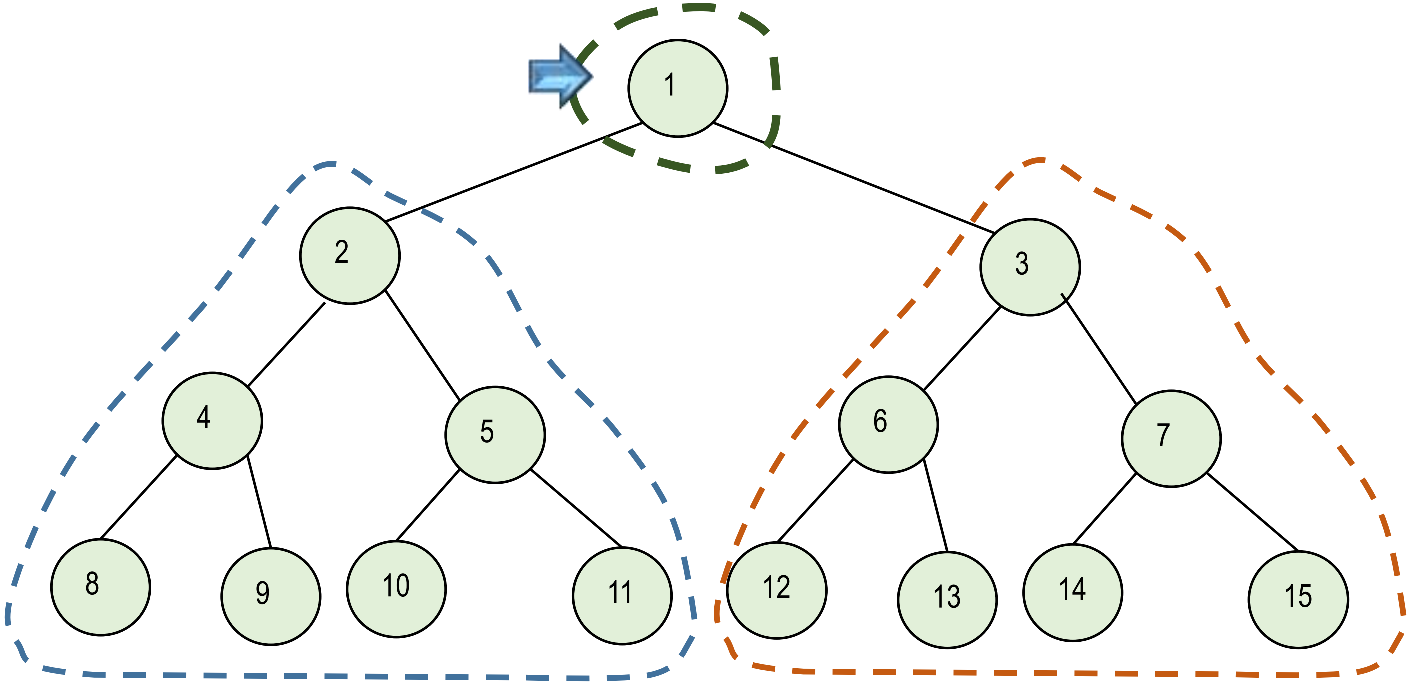


level-order = ab





Binary Tree Construction



Preorder And Postorder

preorder = ab

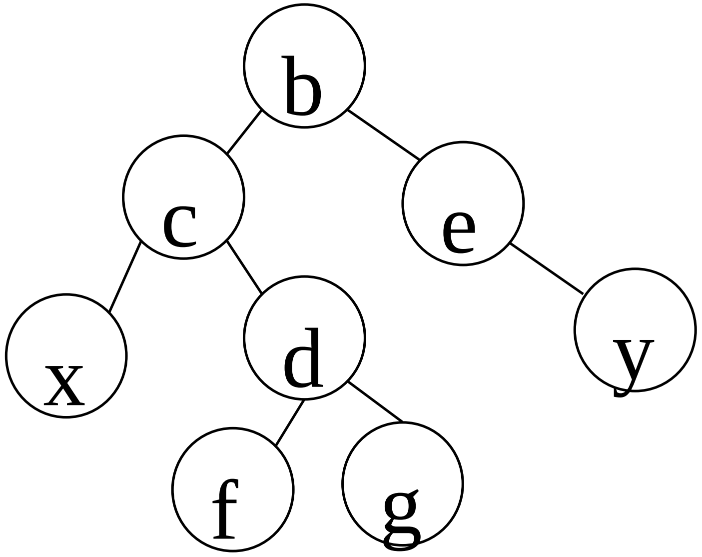
postorder = ba

- In some cases, preorder and postorder do not uniquely define a binary tree.
- Nor do preorder and level order. Give an example?
- Nor do postorder and level order. Give an example?

Inorder And Preorder

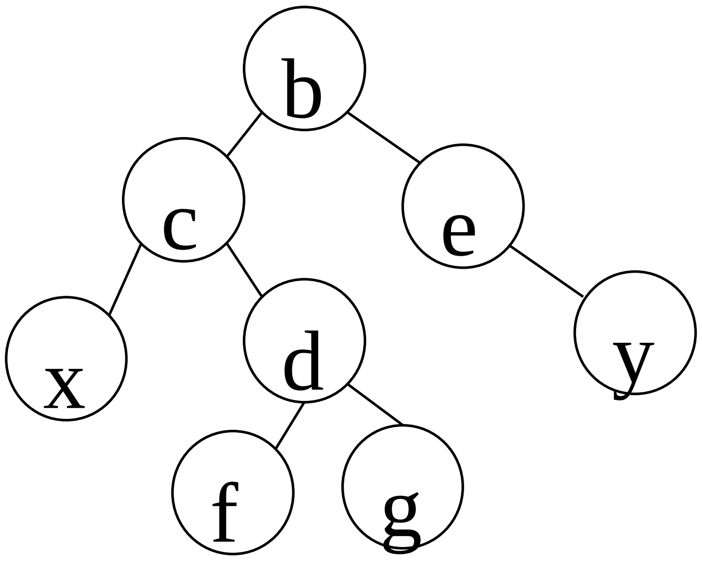
- inorder = xcfdgbey
- preorder = bcxdfgey

- Scan the preorder **left to right** using the inorder to separate left and right subtrees.
- b is the root of the tree; xcfdg are in the left subtree; ey are in the right subtree.
- The left subtree is printed before the root node.



Inorder And Preorder

- inorder = xcfdg**b**ey
- preorder = **b**cxdfgey

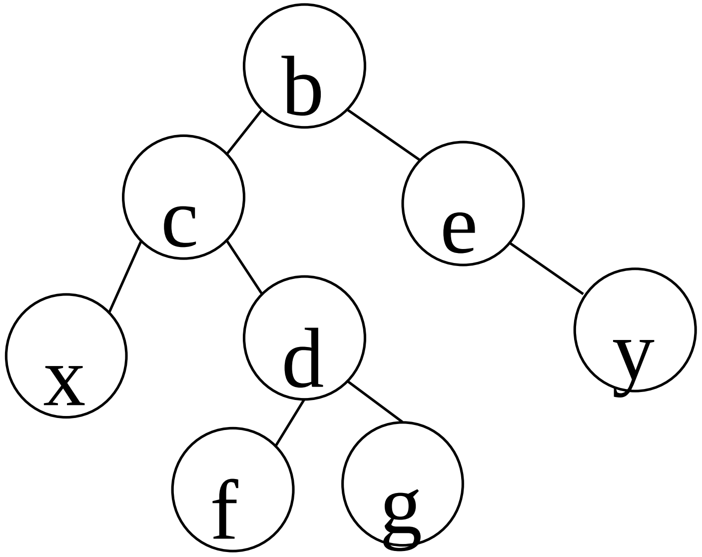


- Scan the preorder left to right using the inorder to separate left and right subtrees.
- b is the root of the tree; xcfdg are in the left subtree; ey are in the right subtree.
- The left subtree is printed before the root node.

Inorder And Preorder

- inorder = xcfdg **b** ey
- preorder = **b** cxdfgey

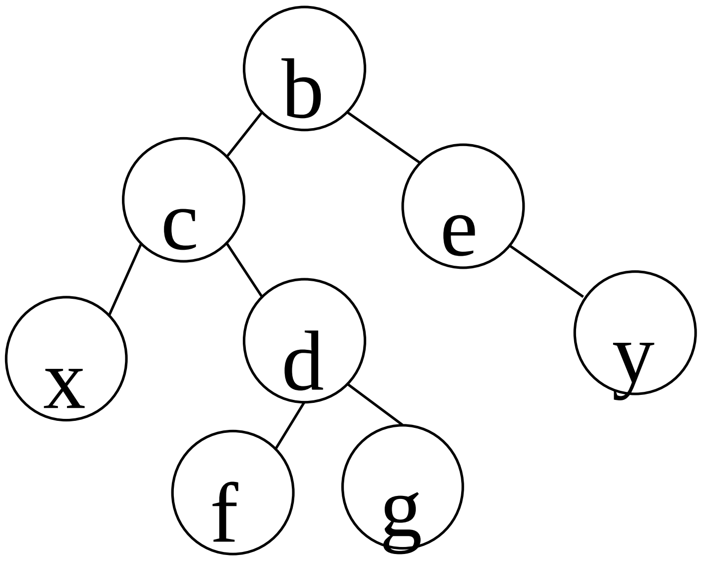
- Scan the preorder left to right using the inorder to separate left and right subtrees.
- b is the root of the tree; xcfdg are in the left subtree; ey are in the right subtree.
- The left subtree is printed before the root node.



Inorder And Preorder

- inorder = x fdg **b** ey
- preorder = **b** xdfgey

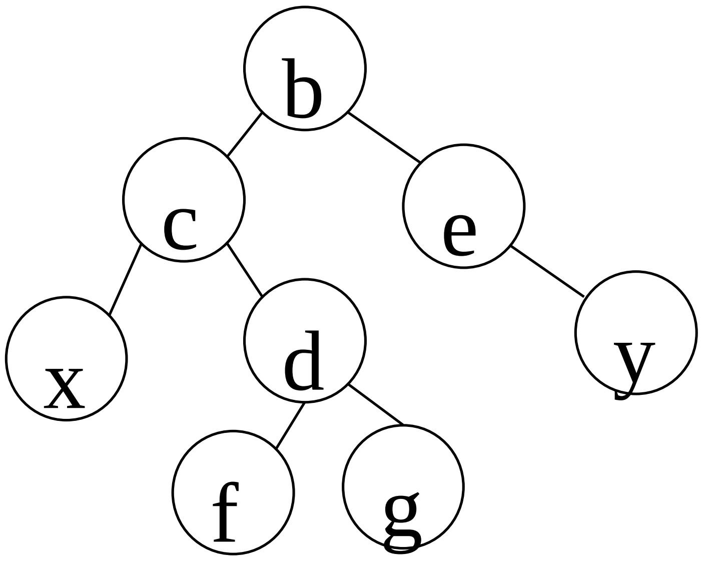
- Scan the preorder left to right using the inorder to separate left and right subtrees.
- b is the root of the tree; xcfdg are in the left subtree; ey are in the right subtree.
- The left subtree is printed before the root node.



Inorder And Preorder

- inorder = x fdg **b** ey
- preorder = **b** xdfgey

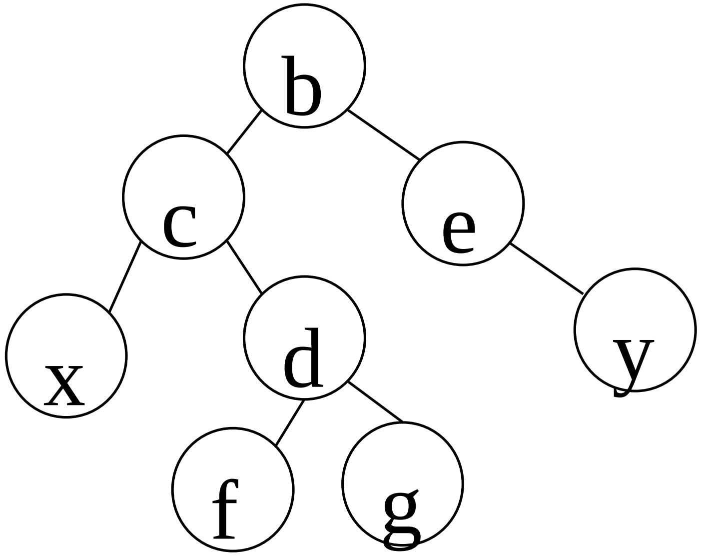
- Scan the preorder left to right using the inorder to separate left and right subtrees.
- b is the root of the tree; xcfdg are in the left subtree; ey are in the right subtree.
- The left subtree is printed before the root node.



Inorder And Preorder

- inorder = x fdg **b** ey
- preorder = **b** xdfgey

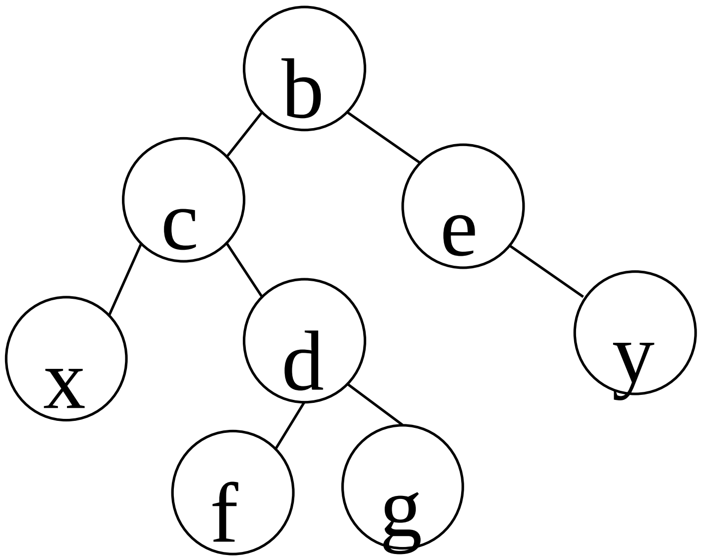
- Scan the preorder left to right using the inorder to separate left and right subtrees.
- b is the root of the tree; xcfdg are in the left subtree; ey are in the right subtree.
- The left subtree is printed before the root node.



Inorder And Postorder

- inorder = xcfdgbey
- postorder = xfgdcyeb

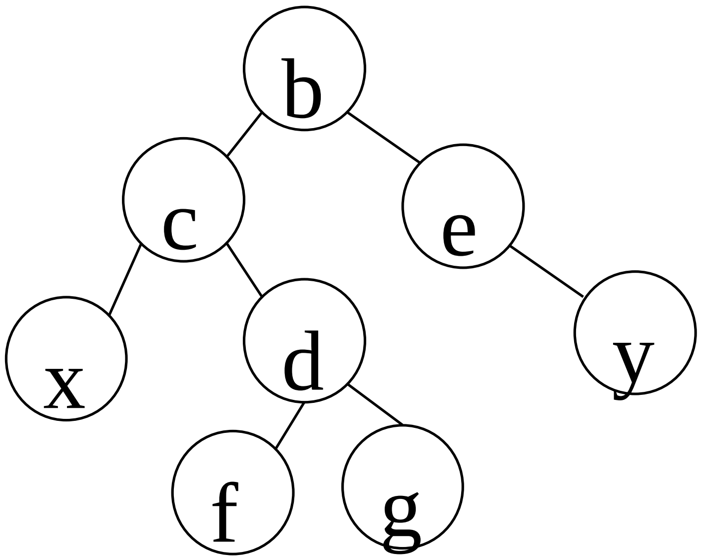
- Scan postorder from **right to left** using inorder to separate left and right subtrees.
- b is the root of the tree; xcfdg are in the left subtree; ey are in the right subtree.



Inorder And Levelorder

- inorder = xcfdgbey
- postorder = bcexdyfg

- Scan level order from left to right using inorder to separate left and right subtrees.
- b is the root of the tree; xcfdg are in the left subtree; ey are in the right subtree.



Exercises

Reconstruct the binary tree

Inorder： 8,4,2,10,5,11,1,6,13,3

Level order: 1,2,3,4,5,6,8,10,11,13

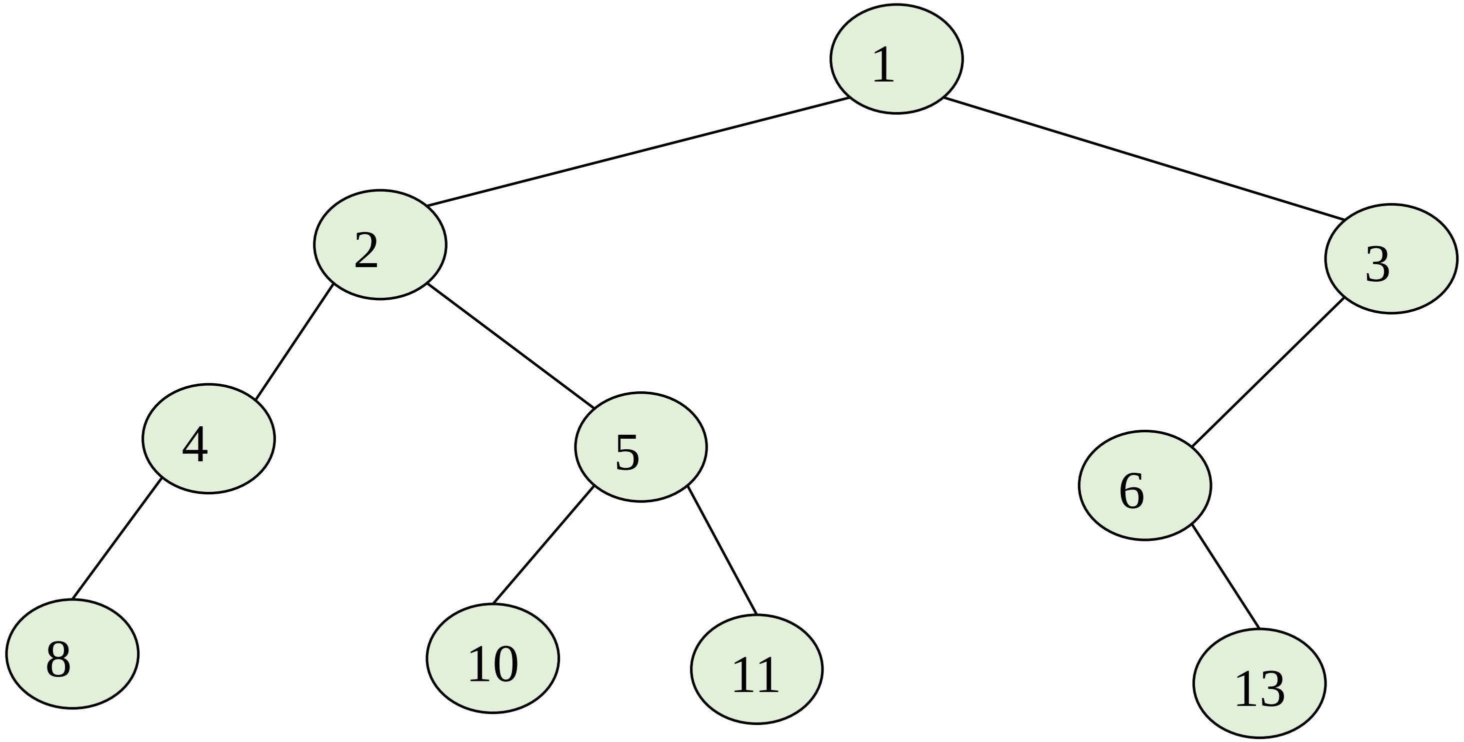
Exercises

`Reconstruct the binary tree`

Inorder: `8,4,2,10,5,11,1,6,13,3`

Postorder: `8,4,10,11,5,2,13,6,3,1`

Exercises



Exercises

- Give an example in which there can be two binary trees that can be constructed based on the sequences of the inorder and level order traversal methods.