LECTURE 18

BST AND TREE TRAVERSALS

MCS 275 Spring 2023
Emily Dumas

LECTURE 18: BST AND TREE TRAVERSALS

Reminders and announcements:

- Project 2 due today.
- Homework 7 available.
- Project 1 will be graded by Monday.
  - It's only the manual review you're waiting on; the autograder results account for most of the project grade.

UPDATED BST CLASS

I put an implementation of binary search tree (as class `BST`) in `trees.py`.

Also modified so `BST()` is considered a valid, empty tree. (That is, `None` as a key is treated specially.)

`BST.search` also supports a `verbose` mode.

TREEUTIL

I added a module to the `datastructures` directory of the sample code repository which can generate random trees. You'll use it in lab this week.

- Documentation of `treeutil` module

INTEGERSET

As a sample application of BST, we can make a class that stores a set of integers, supporting membership testing and adding new elements.

Compare alternatives:

- Unsorted list - fast to insert, but slow membership test
- Sorted list - fast membership test, slow insert

IMPLEMENTATION HIDING

To use `BST`, you need to know about and work with `Node` objects.

In contrast, `IntegerSet` has an interface based directly on the values to be stored. It hides the fact that its implementation uses a BST.

WALKING A TREE

Back to discussing binary trees (not necessarily BST).

For some purposes we need to visit every node in a tree and perform some action on them.

To do this is to **traverse** or **walk** the tree.

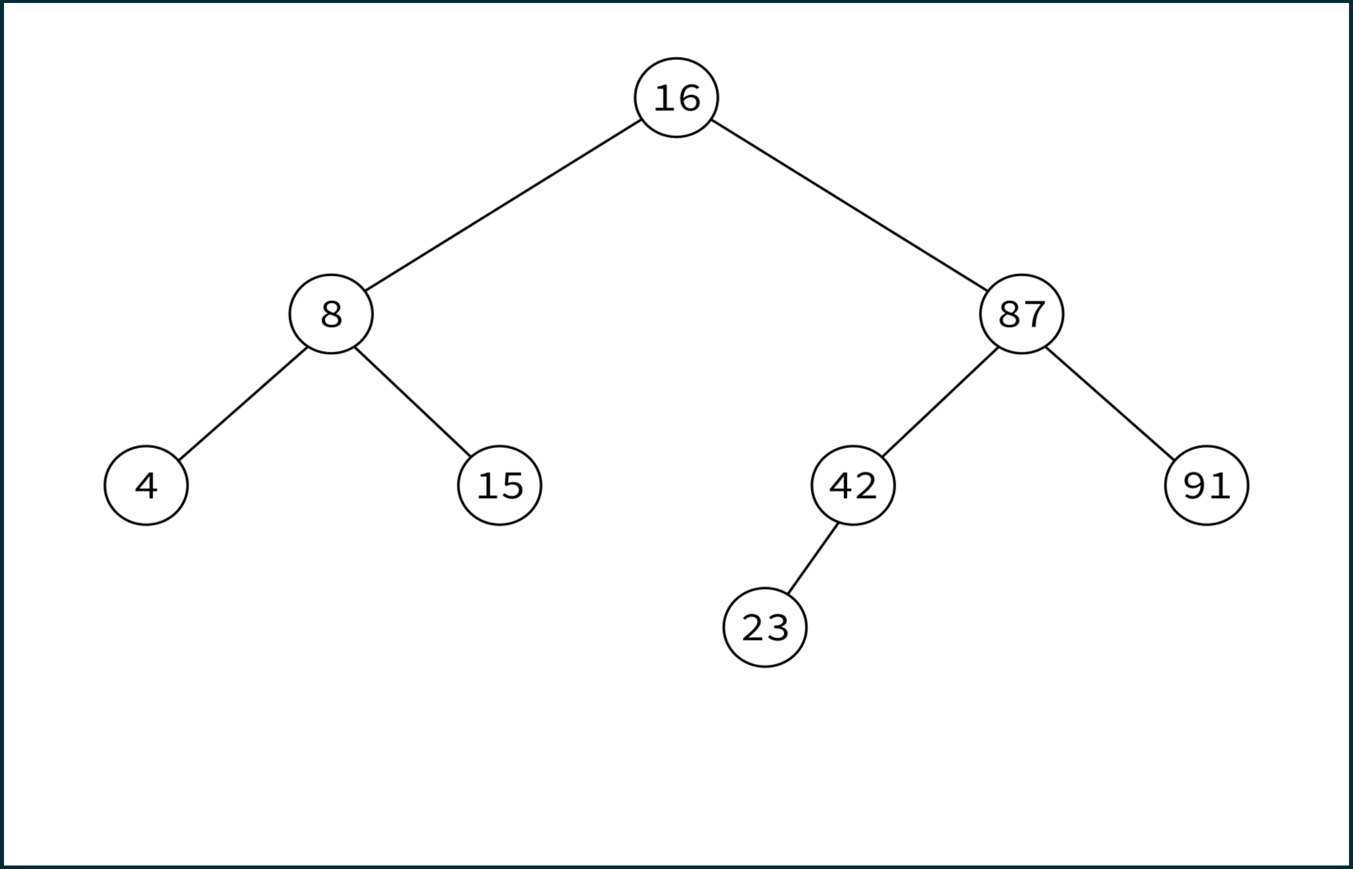
NAMED TRAVERSALS

The three most-often used recursive traversals:

- **preorder** - Node, left subtree, then right subtree.
- **postorder** - Left subtree, right subtree, then node.
- **inorder** - Left subtree, node, then right subtree.

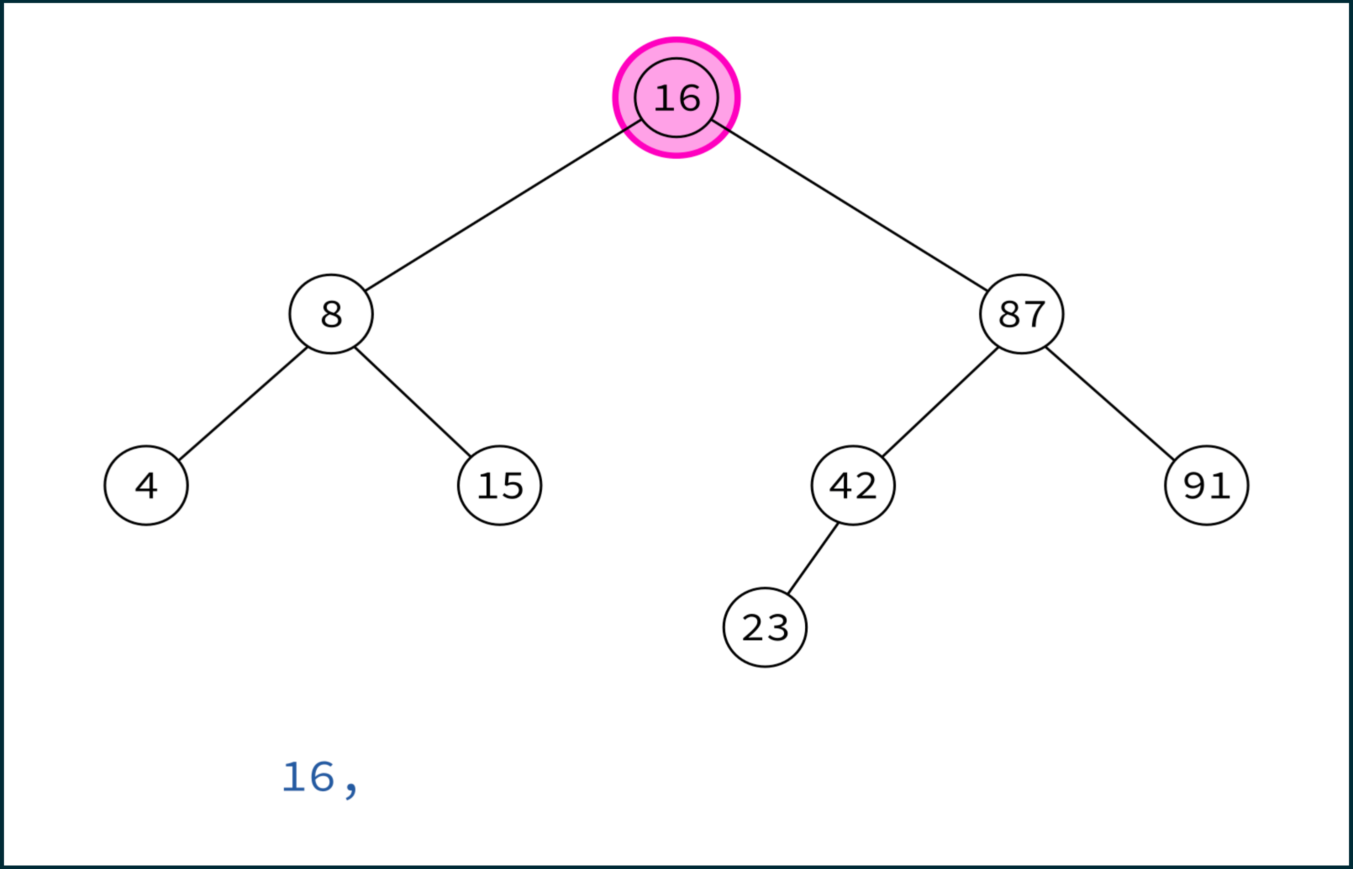
Note: They *all* visit left child before right child.

PREORDER TRAVERSAL



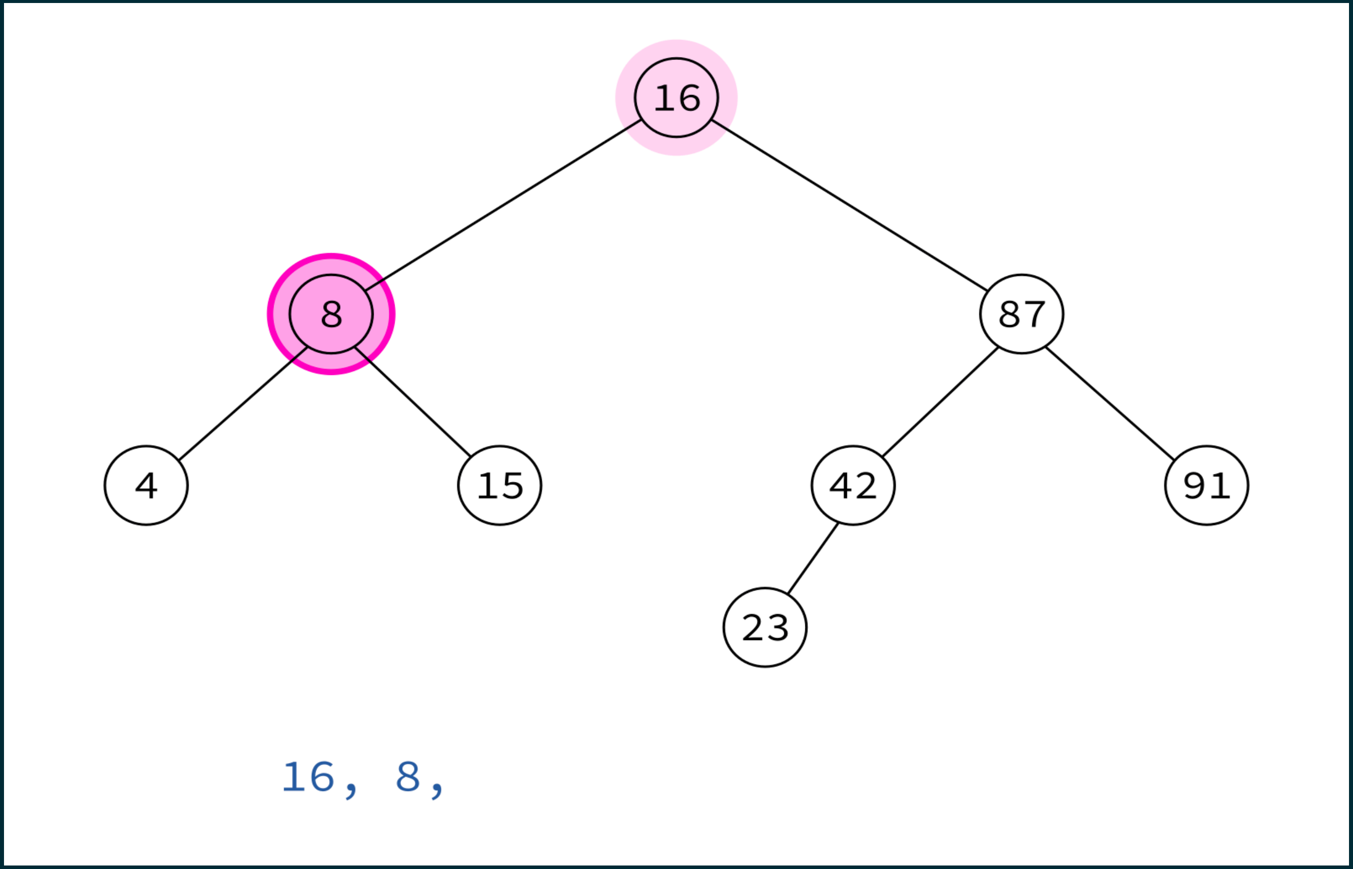
node, left, right

PREORDER TRAVERSAL



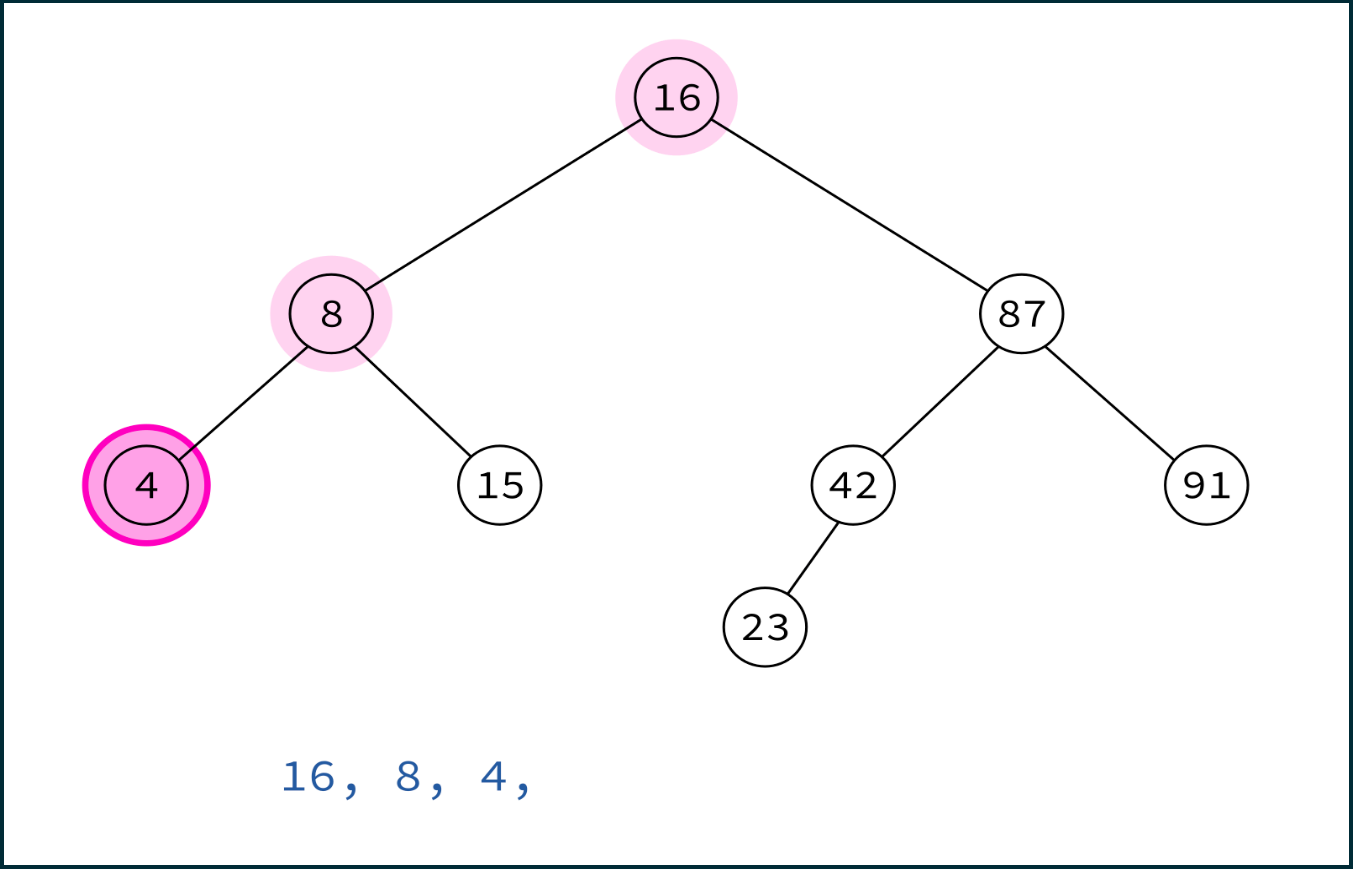
node, left, right

PREORDER TRAVERSAL



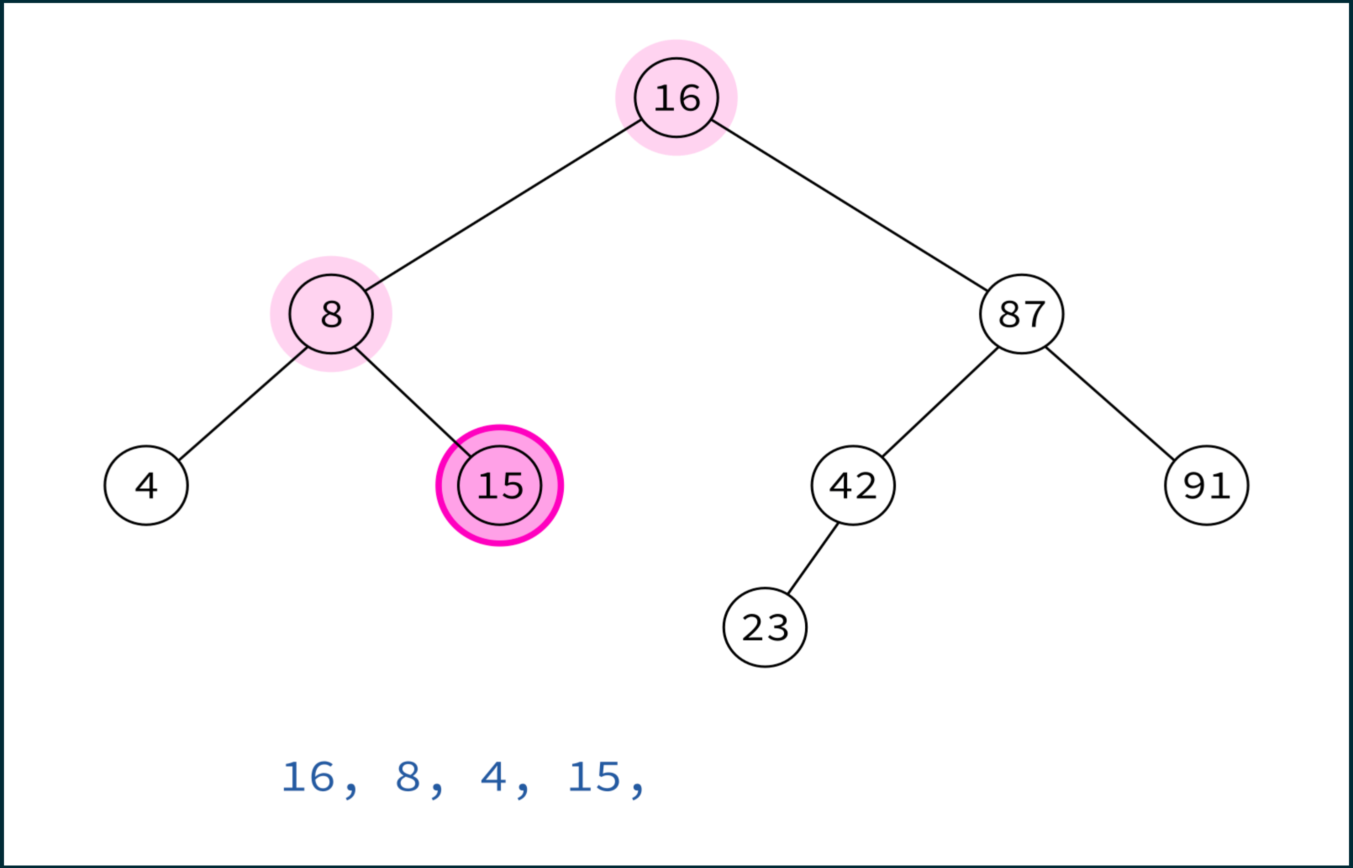
node, left, right

PREORDER TRAVERSAL



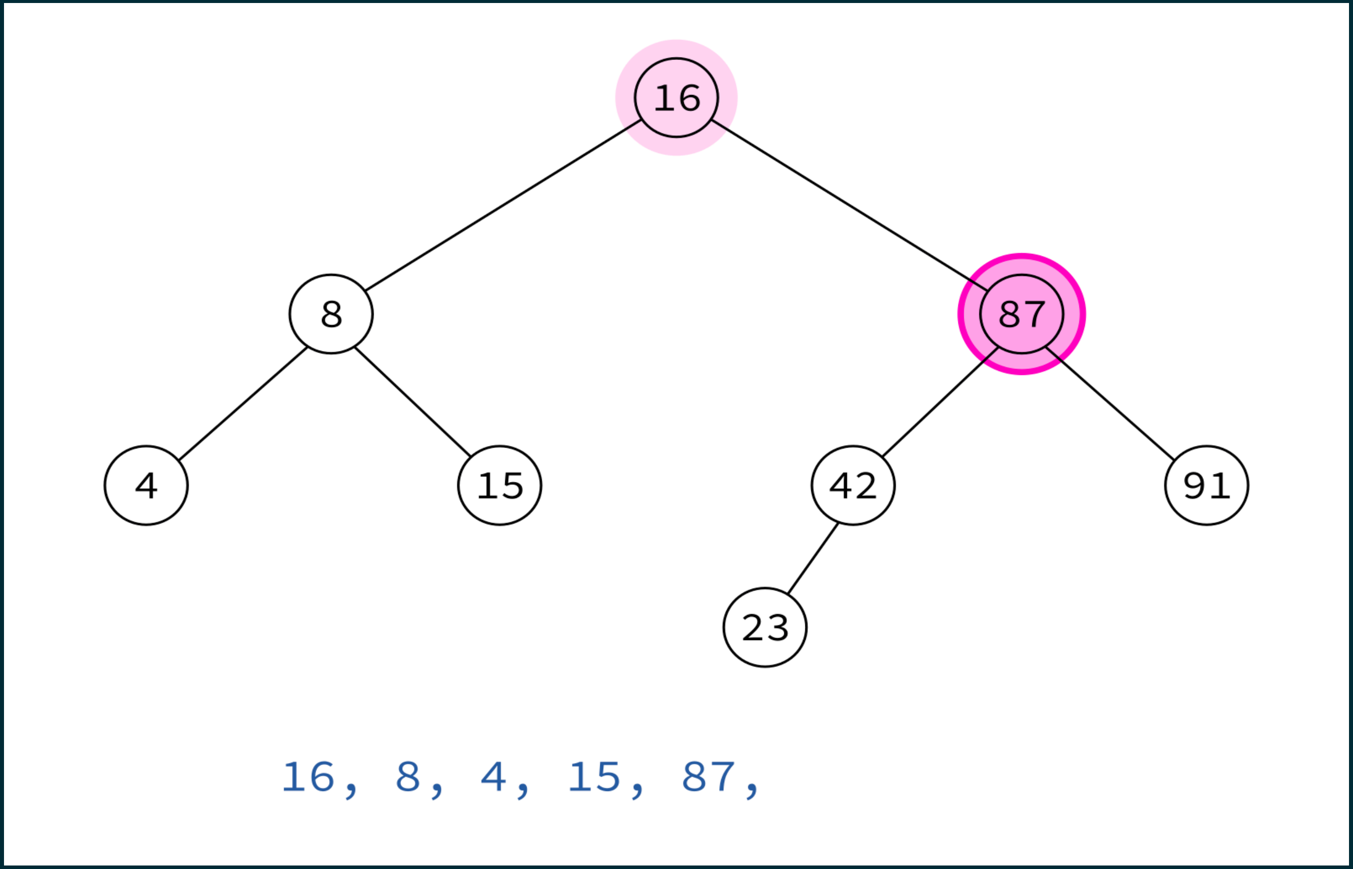
node, left, right

PREORDER TRAVERSAL



node, left, right

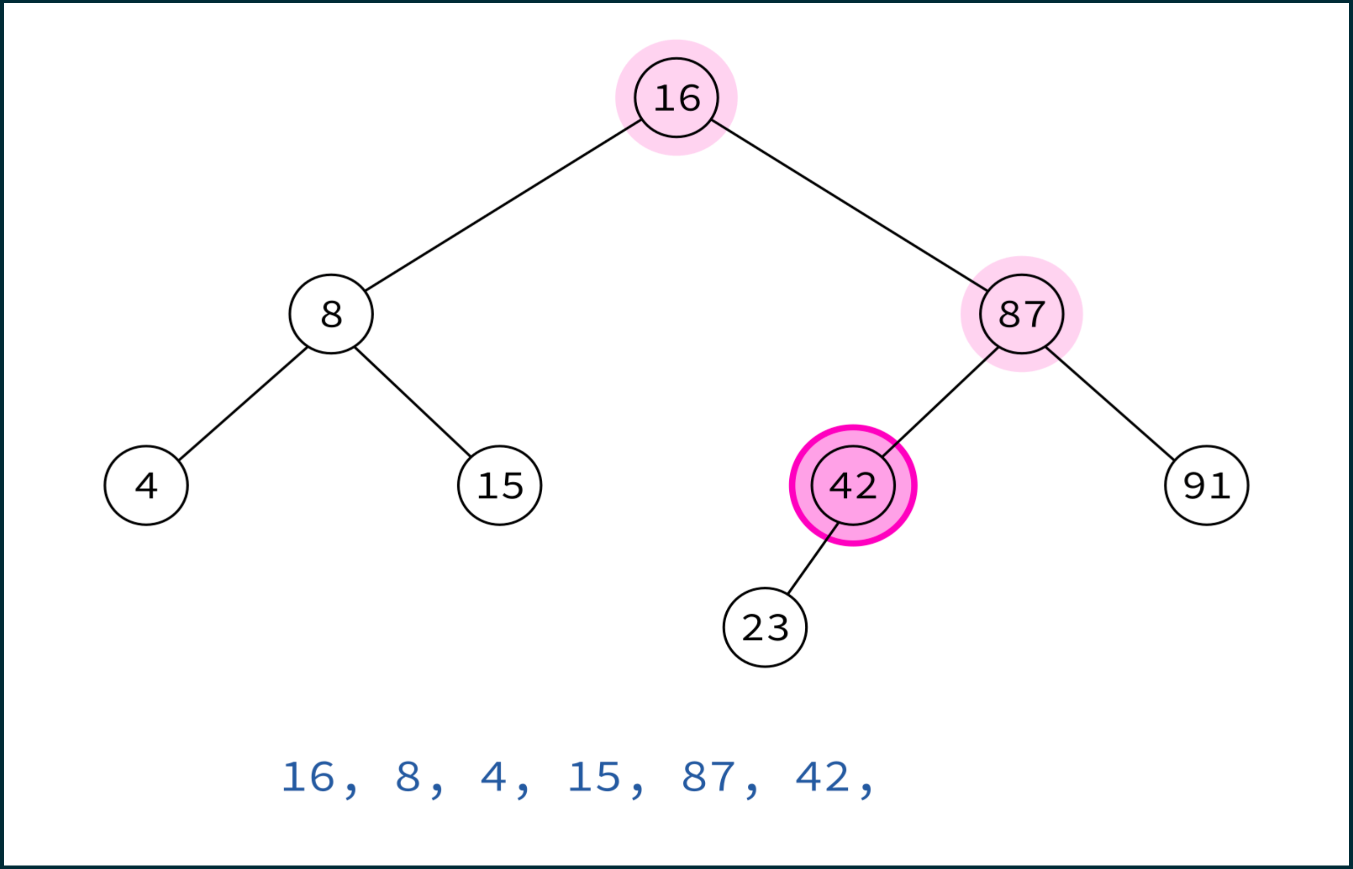
PREORDER TRAVERSAL



16, 8, 4, 15, 87,

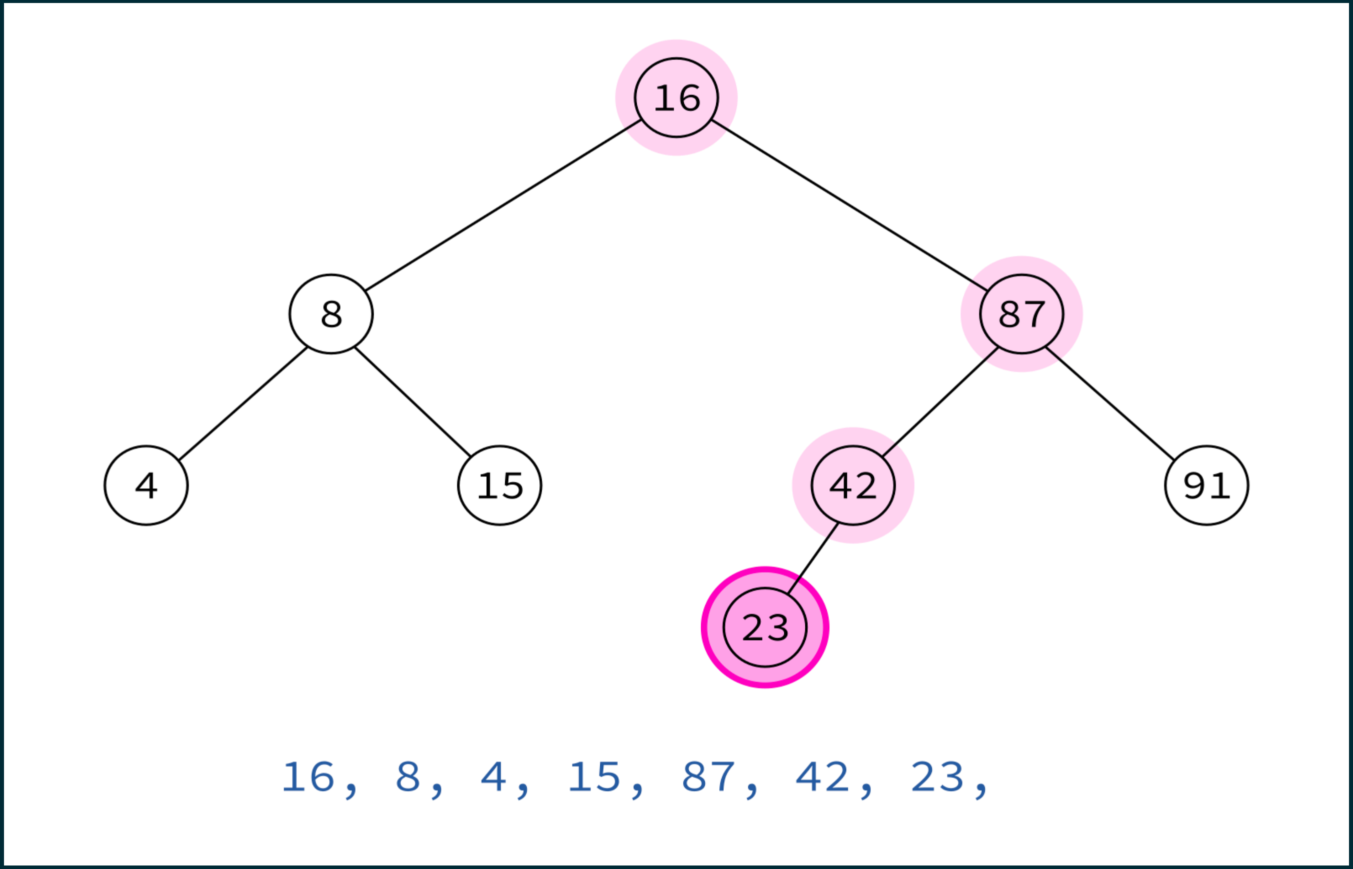
node, left, right

PREORDER TRAVERSAL



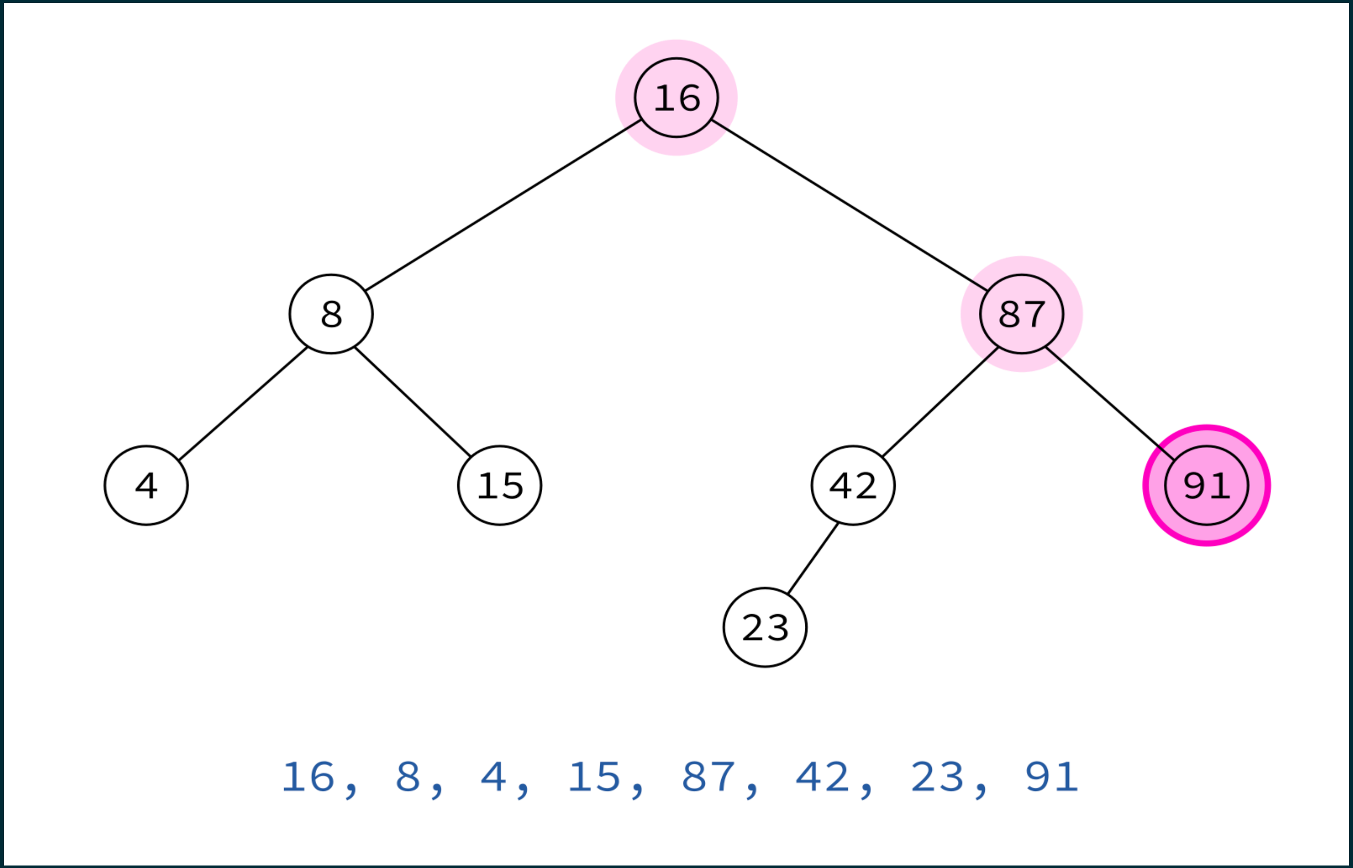
node, left, right

PREORDER TRAVERSAL



node, left, right

PREORDER TRAVERSAL



16, 8, 4, 15, 87, 42, 23, 91

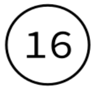
node, left, right

PREORDER TRAVERSAL

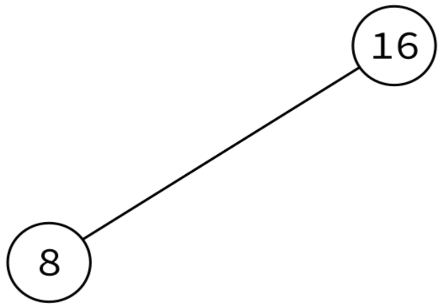
Typical use: Make a copy of the tree.

Insert the keys into an empty BST in this order to recreate the original tree.

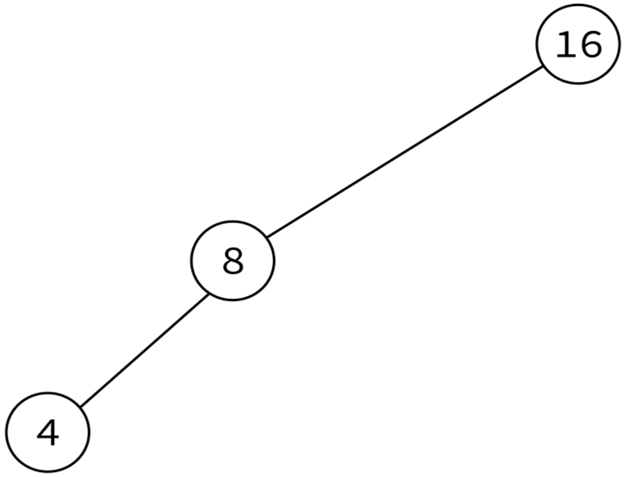
```
16, 8, 4, 15, 87, 42, 23, 91
```



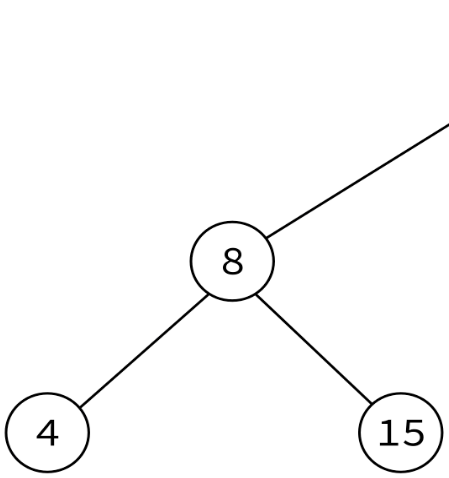
16, 8, 4, 15, 87, 42, 23, 91



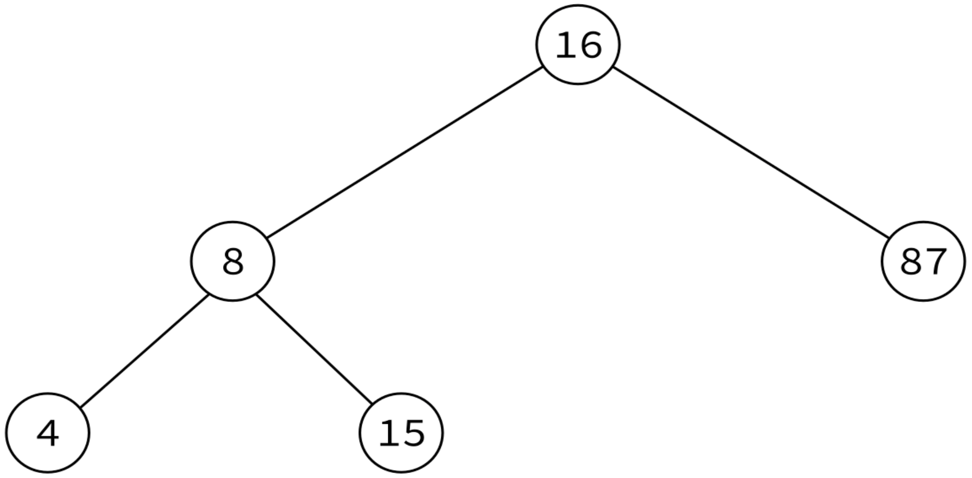
16, 8, 4, 15, 87, 42, 23, 91



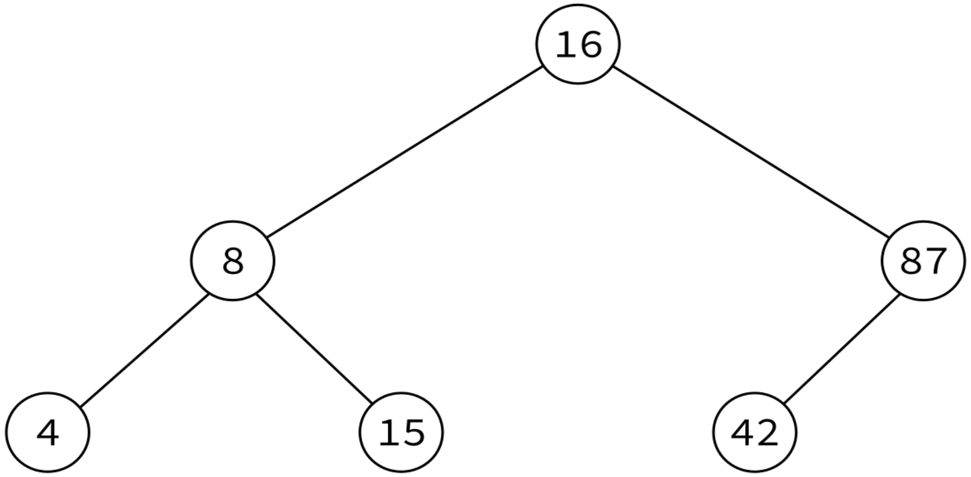
16, 8, 4, 15, 87, 42, 23, 91



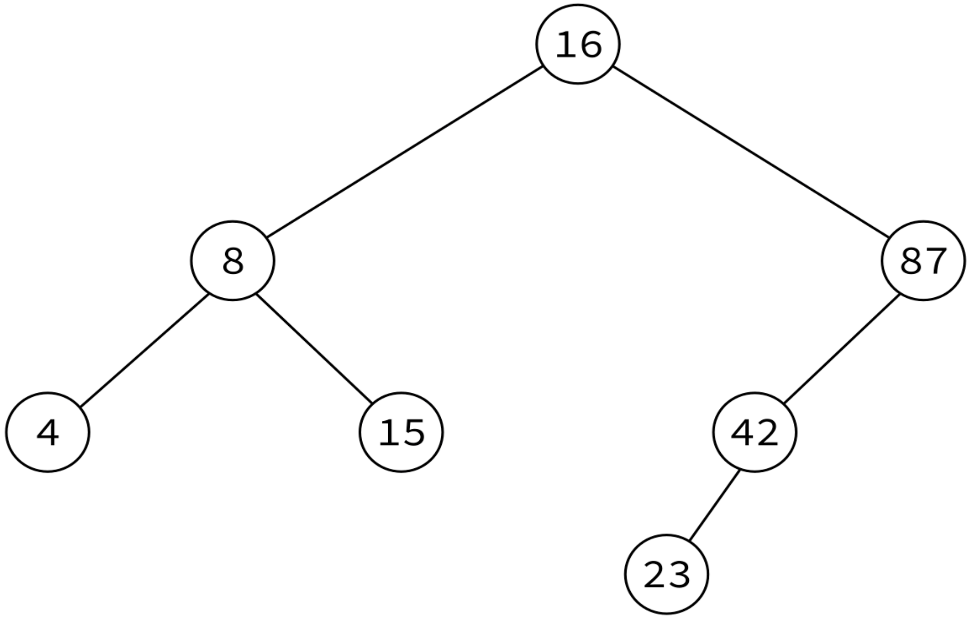
16, 8, 4, 15, 87, 42, 23, 91



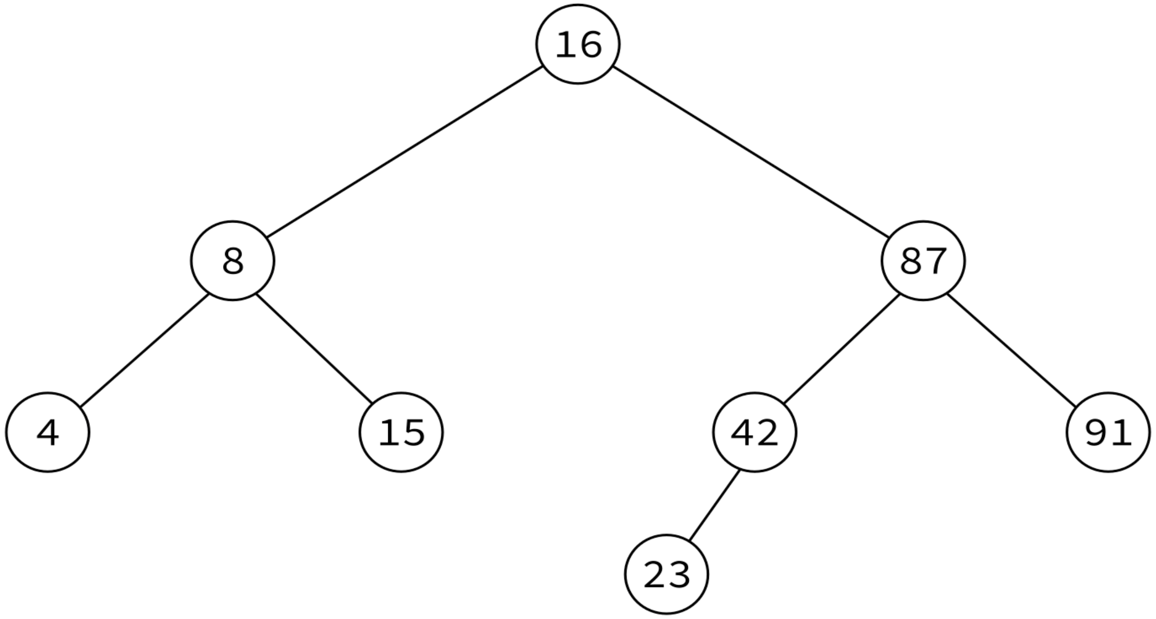
16, 8, 4, 15, 87, 42, 23, 91



16, 8, 4, 15, 87, 42, 23, 91

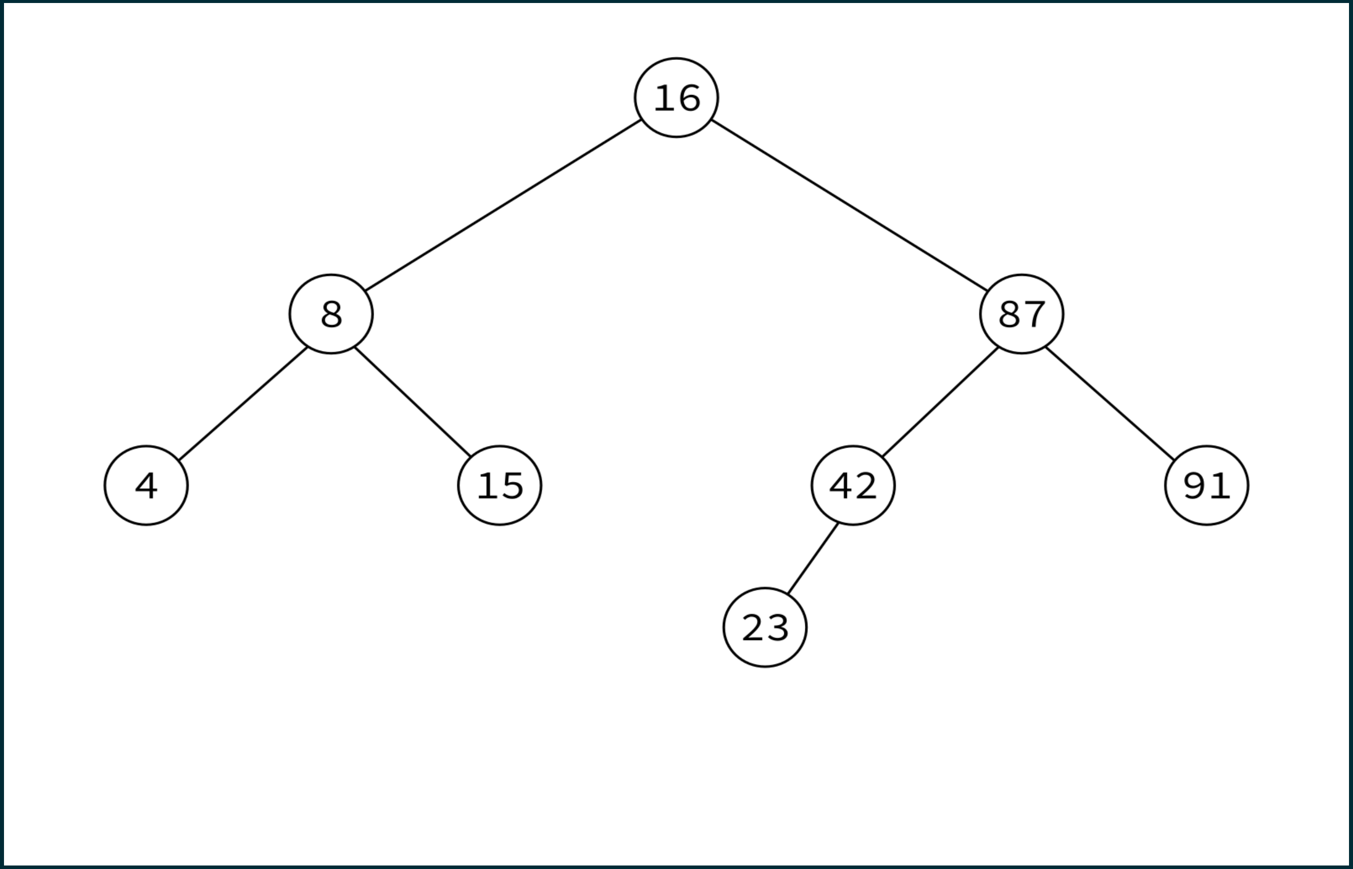


16, 8, 4, 15, 87, 42, 23, 91



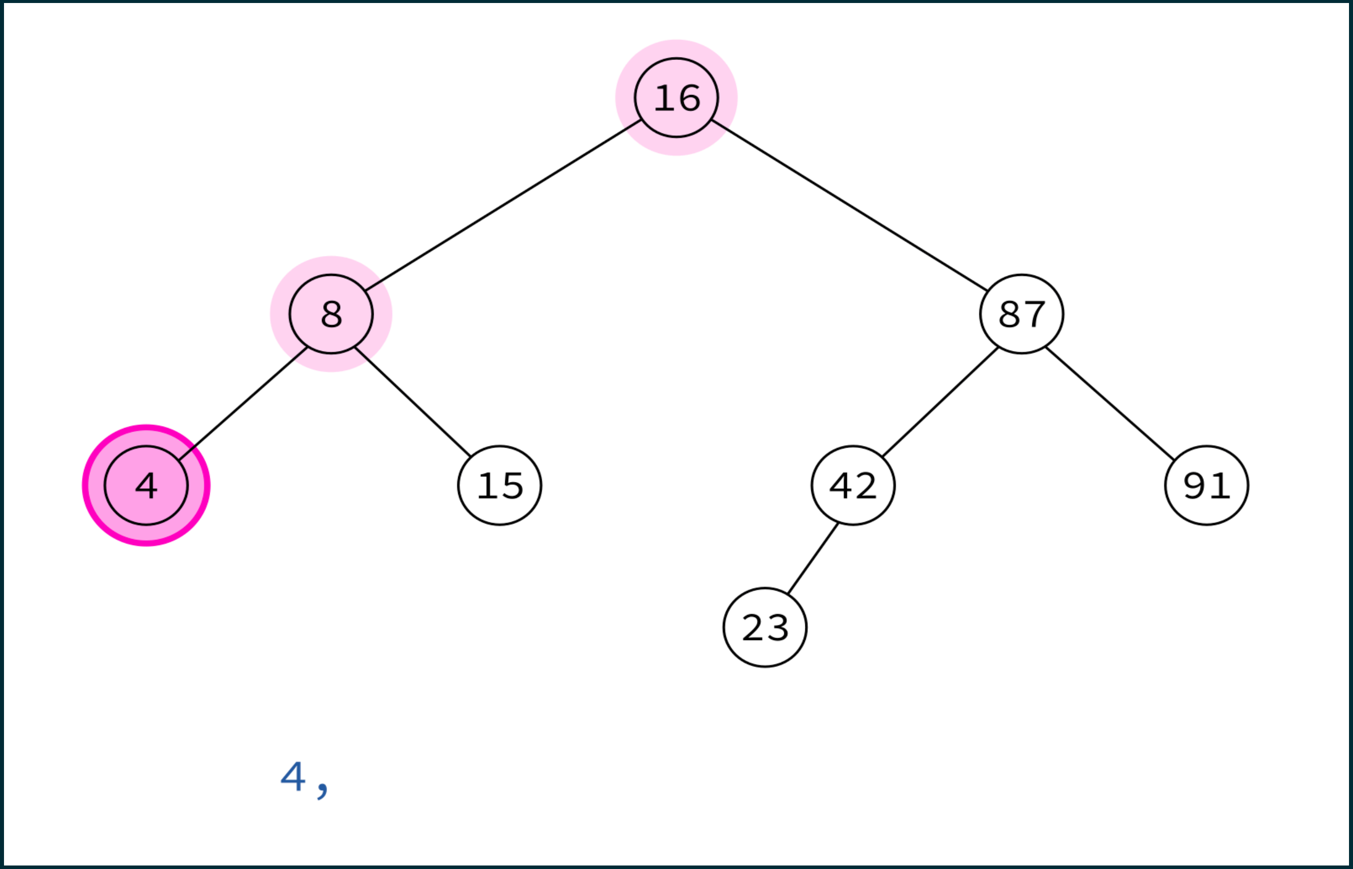
16, 8, 4, 15, 87, 42, 23, 91

POSTORDER TRAVERSAL



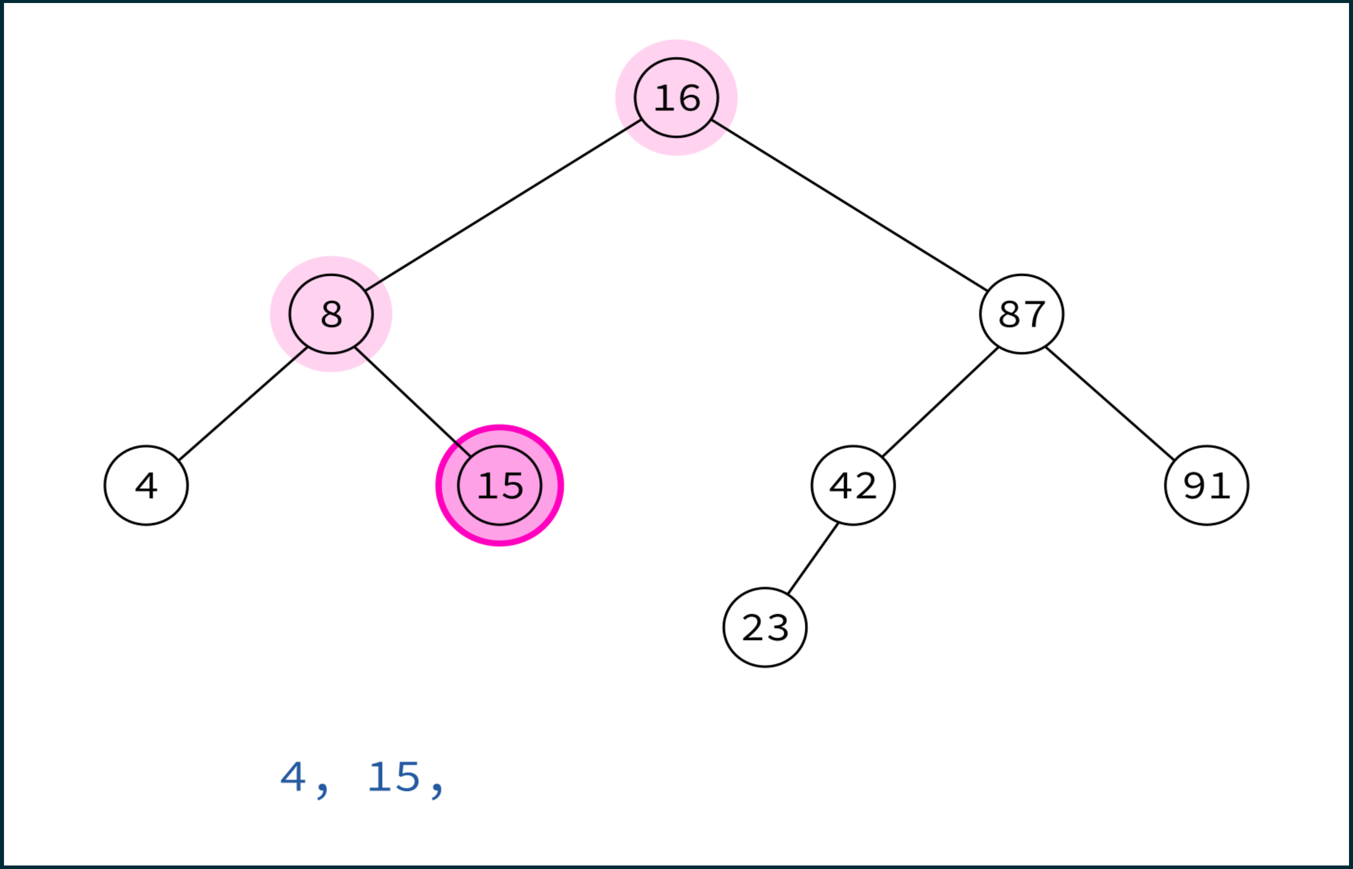
left, right, **node**

POSTORDER TRAVERSAL



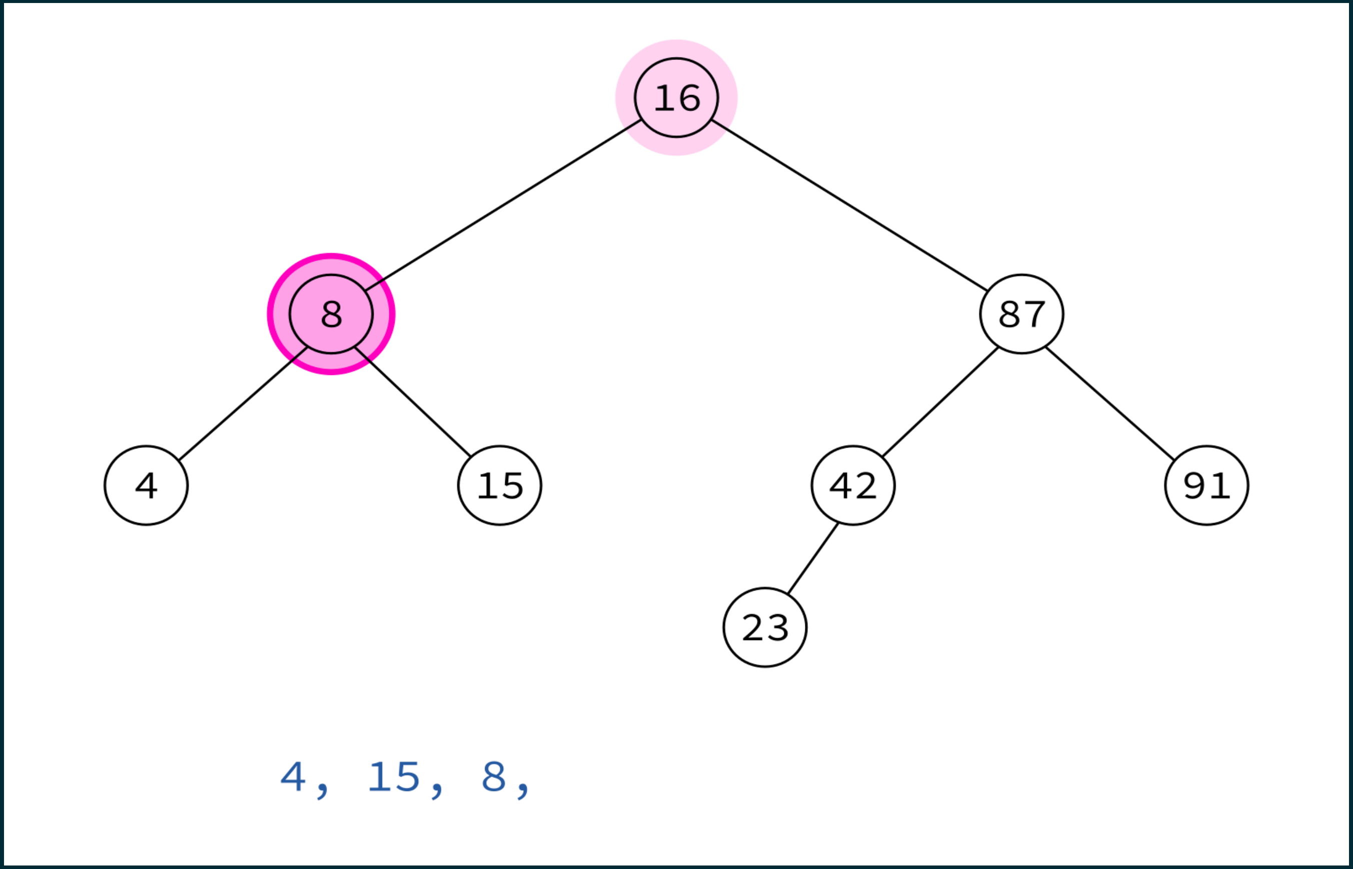
left, right, **node**

POSTORDER TRAVERSAL



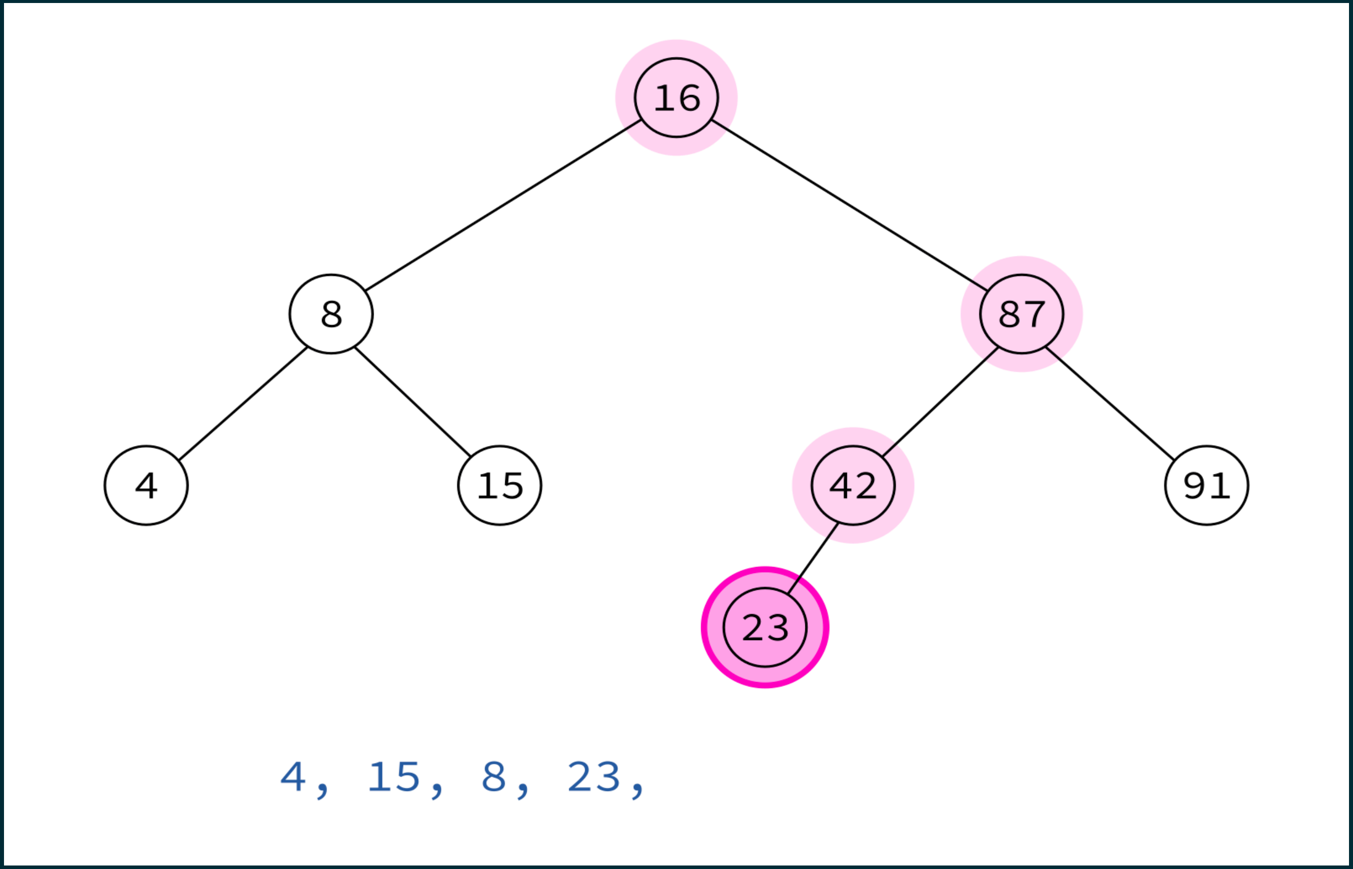
left, right, **node**

POSTORDER TRAVERSAL



left, right, **node**

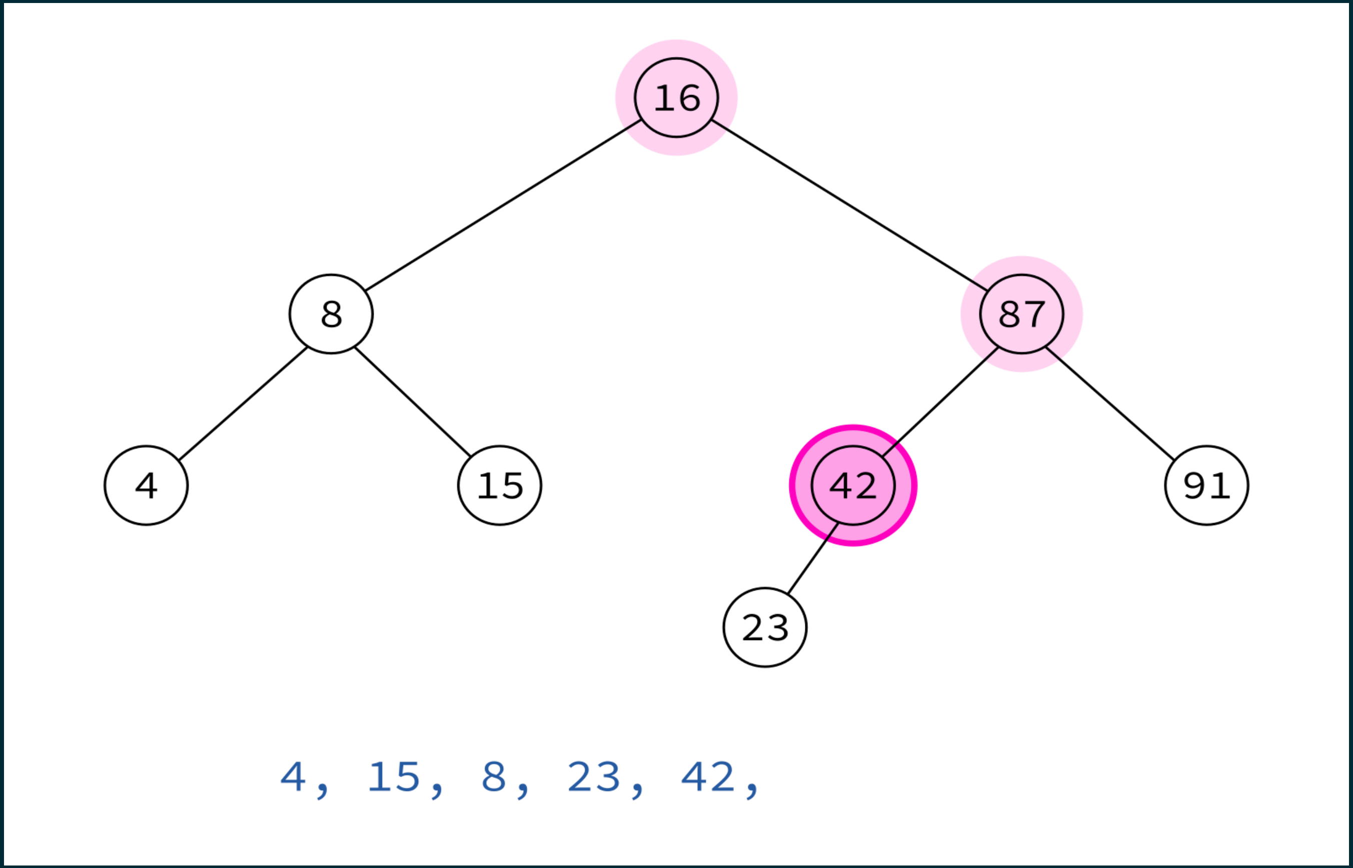
POSTORDER TRAVERSAL



4, 15, 8, 23,

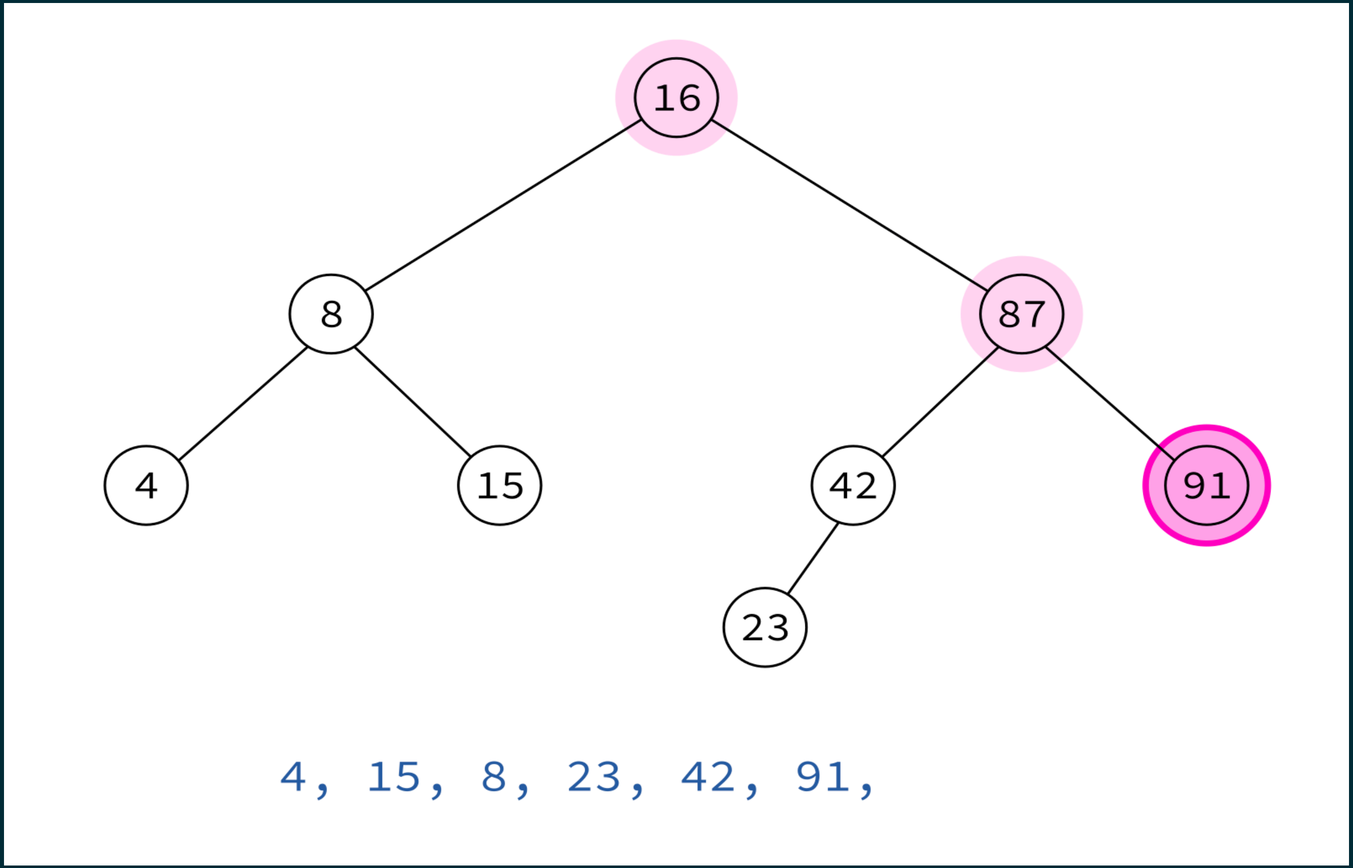
left, right, **node**

POSTORDER TRAVERSAL



left, right, **node**

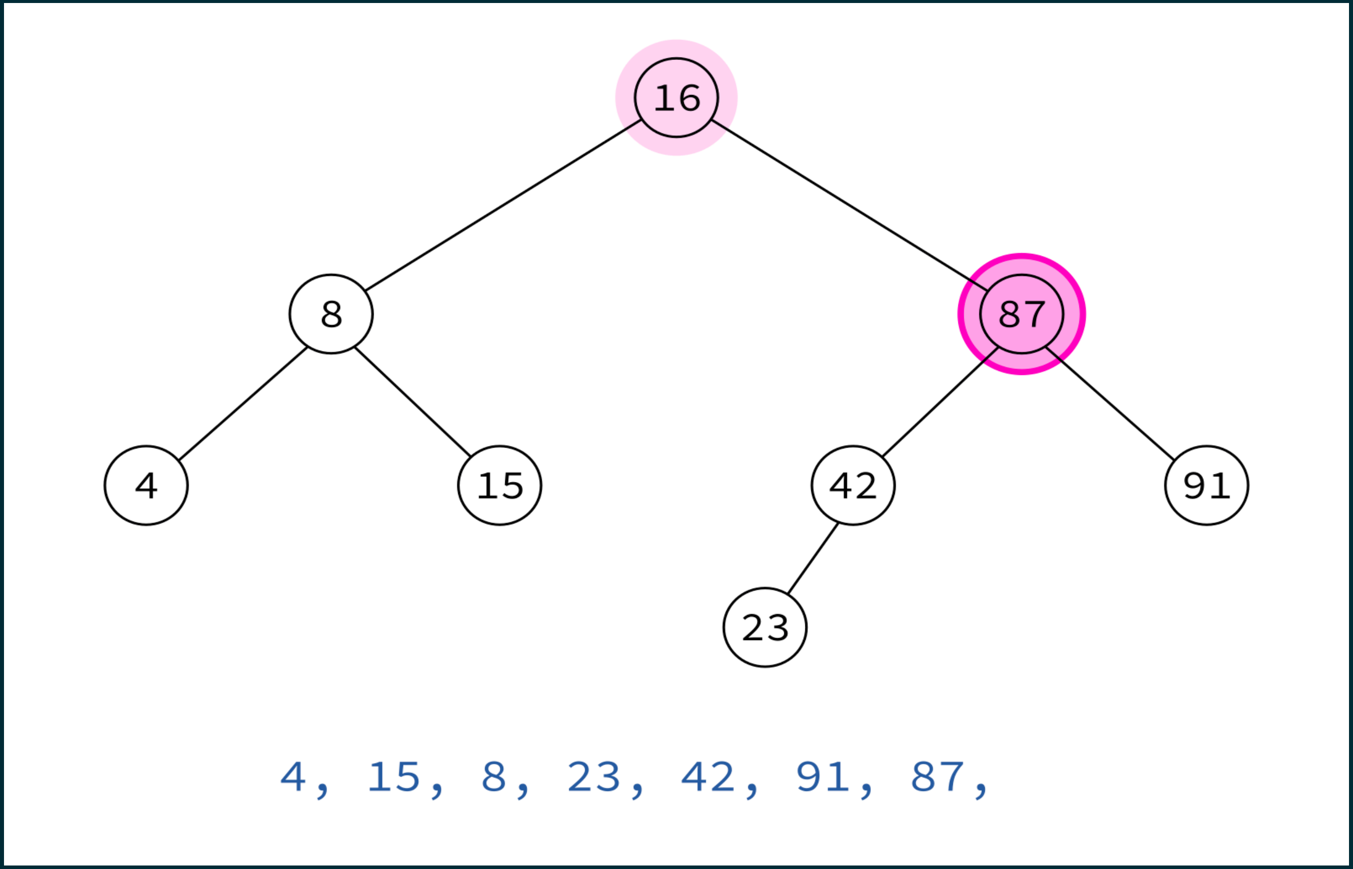
POSTORDER TRAVERSAL



4, 15, 8, 23, 42, 91,

left, right, **node**

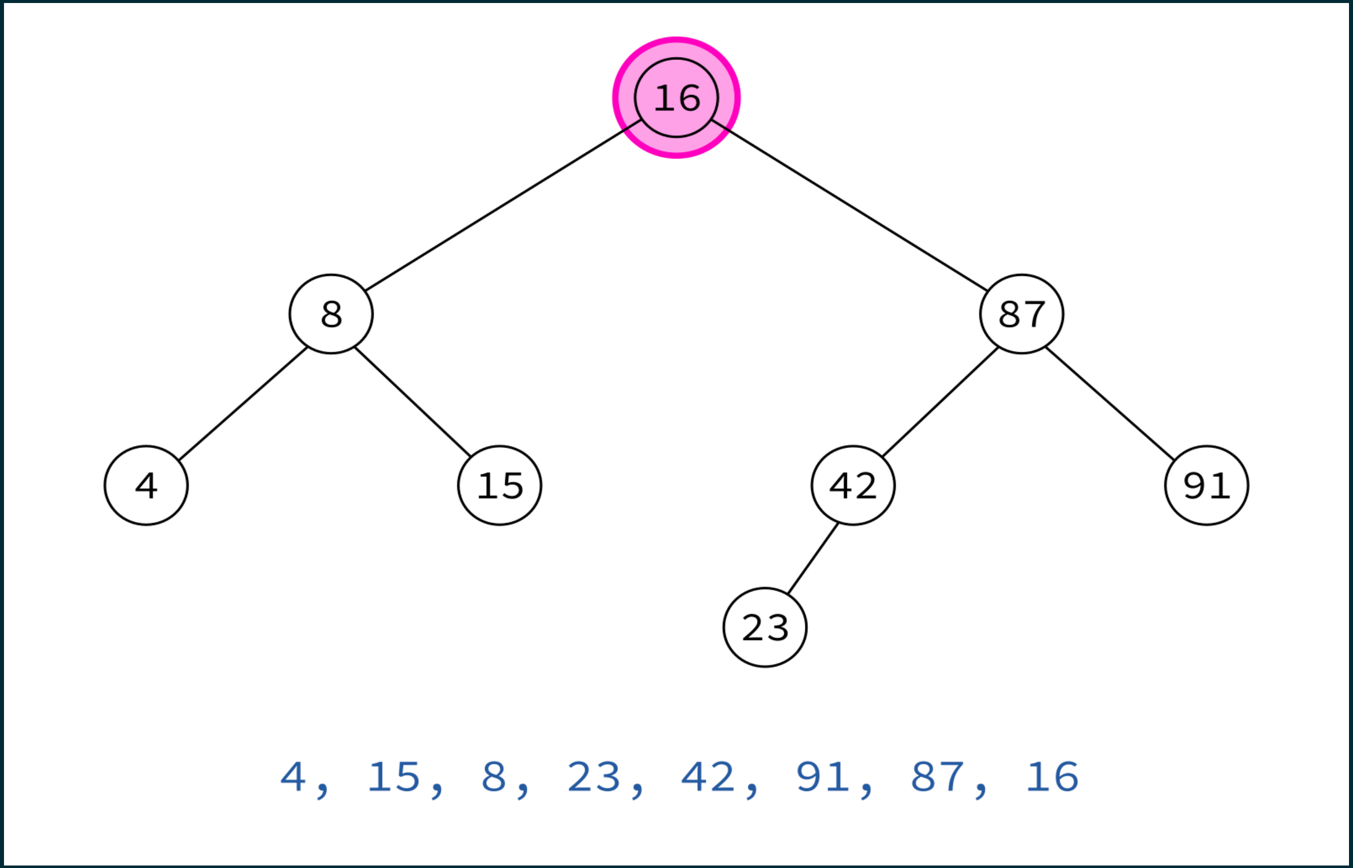
POSTORDER TRAVERSAL



4, 15, 8, 23, 42, 91, 87,

left, right, **node**

POSTORDER TRAVERSAL

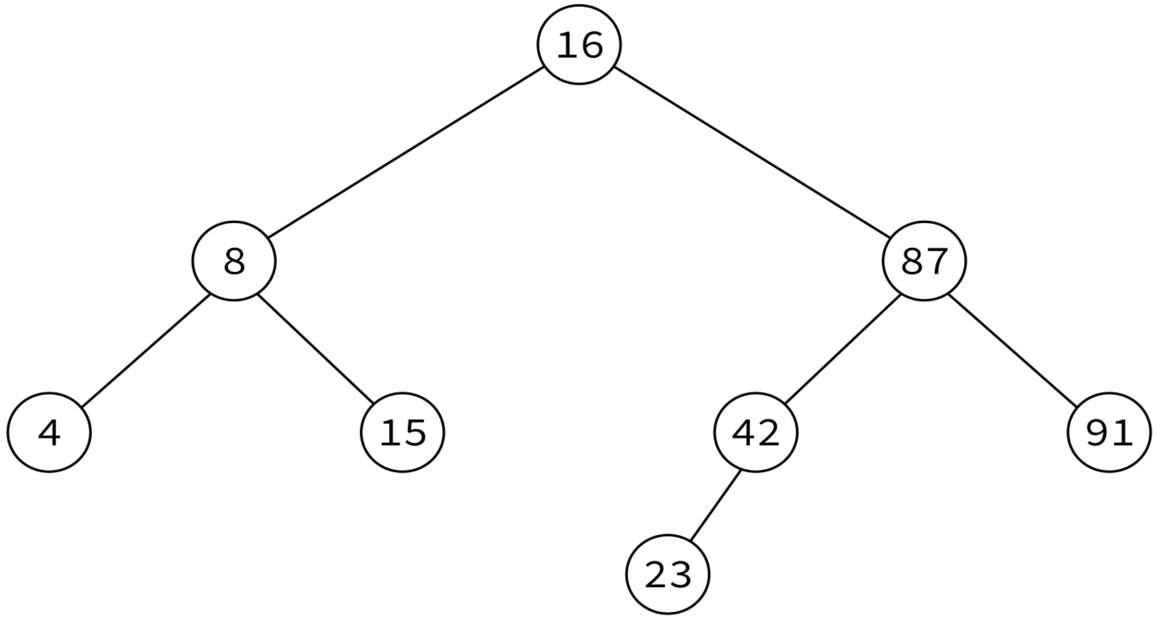


left, right, **node**

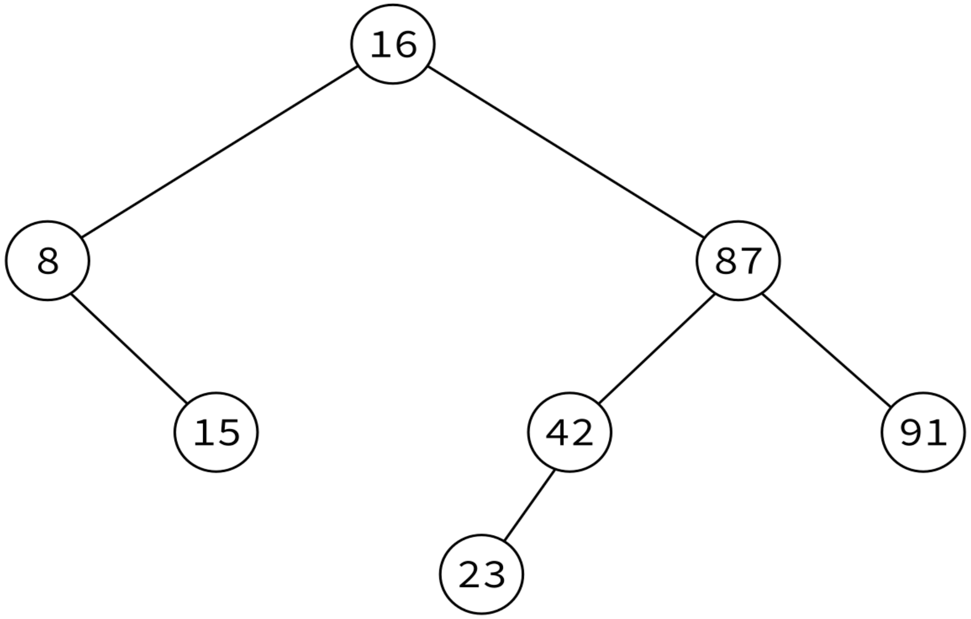
POSTORDER TRAVERSAL

Typical use: Delete the tree.

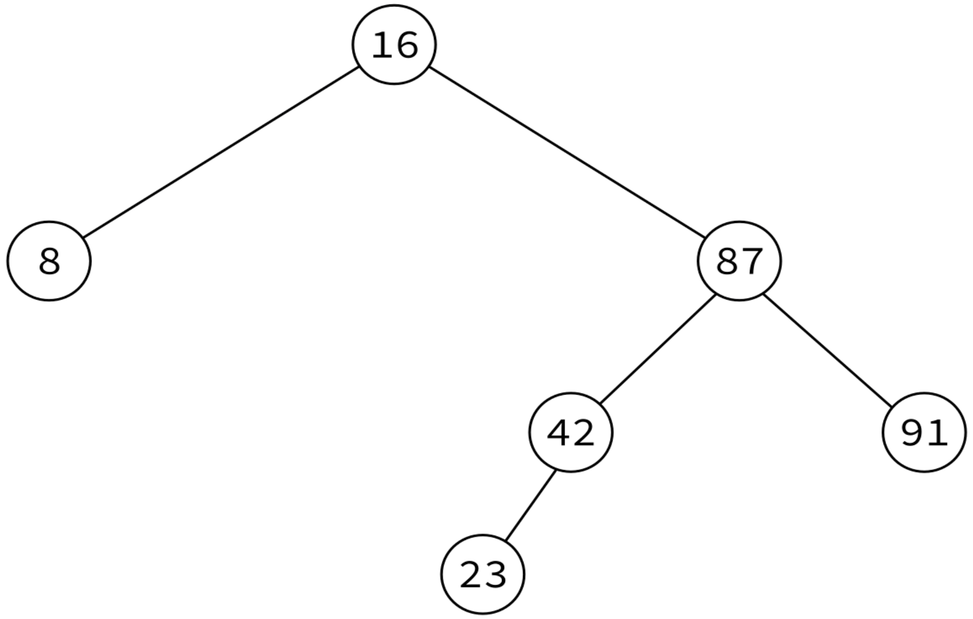
If you delete keys in postorder, then you will only ever be removing nodes without children.



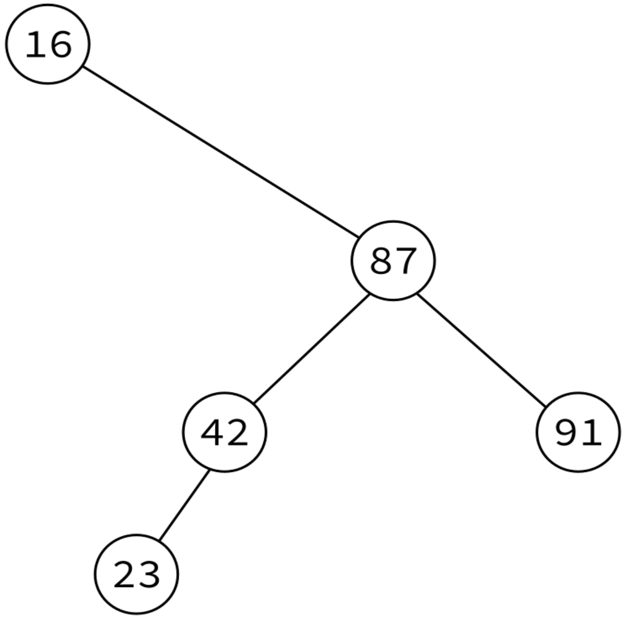
4, 15, 8, 23, 42, 91, 87, 16



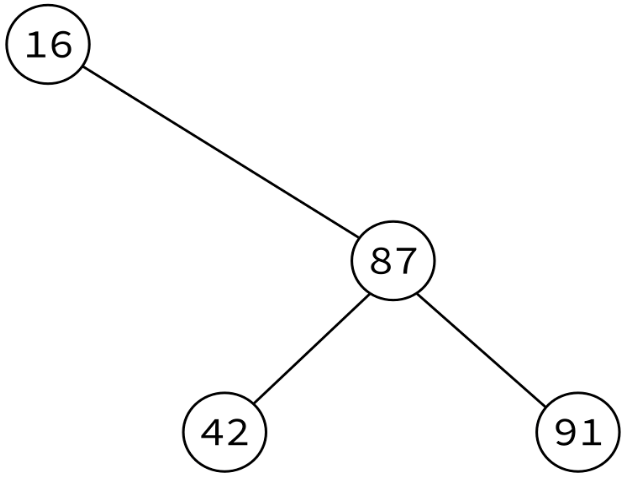
4, 15, 8, 23, 42, 91, 87, 16



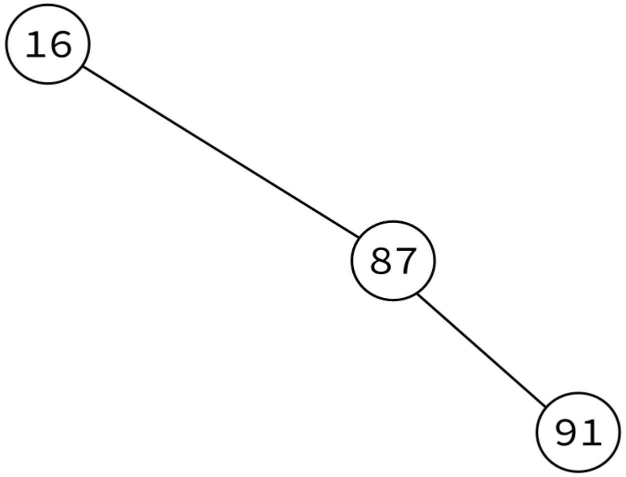
4, 15, 8, 23, 42, 91, 87, 16



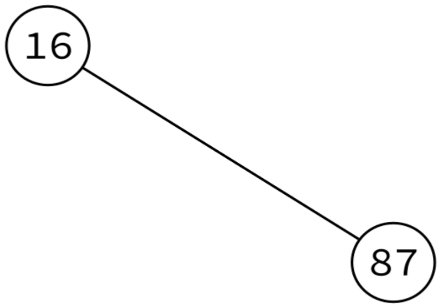
4, 15, 8, 23, 42, 91, 87, 16



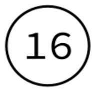
4, 15, 8, 23, 42, 91, 87, 16



4, 15, 8, 23, 42, 91, 87, 16



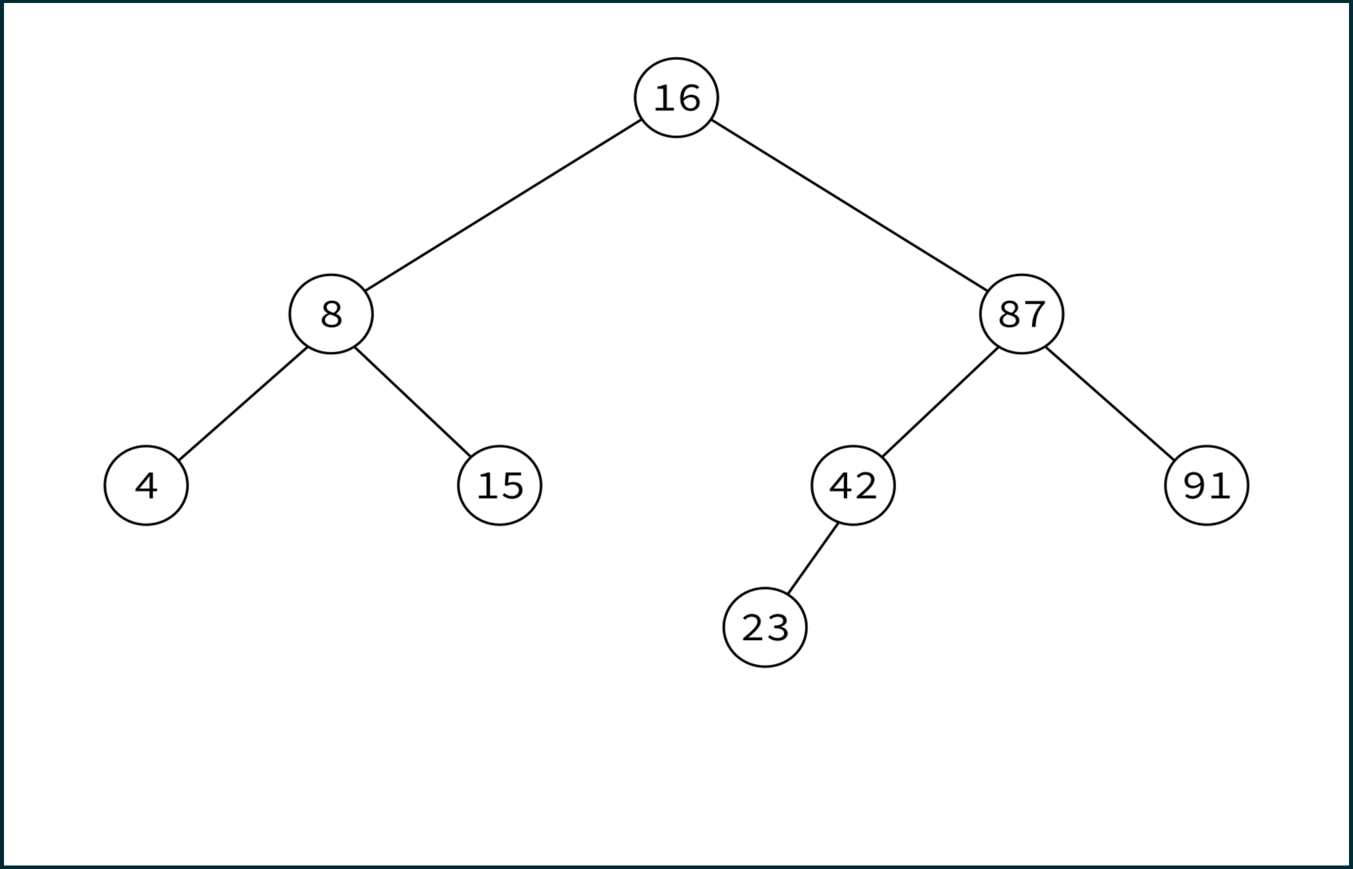
4, 15, 8, 23, 42, 91, 87, 16



4, 15, 8, 23, 42, 91, 87, 16

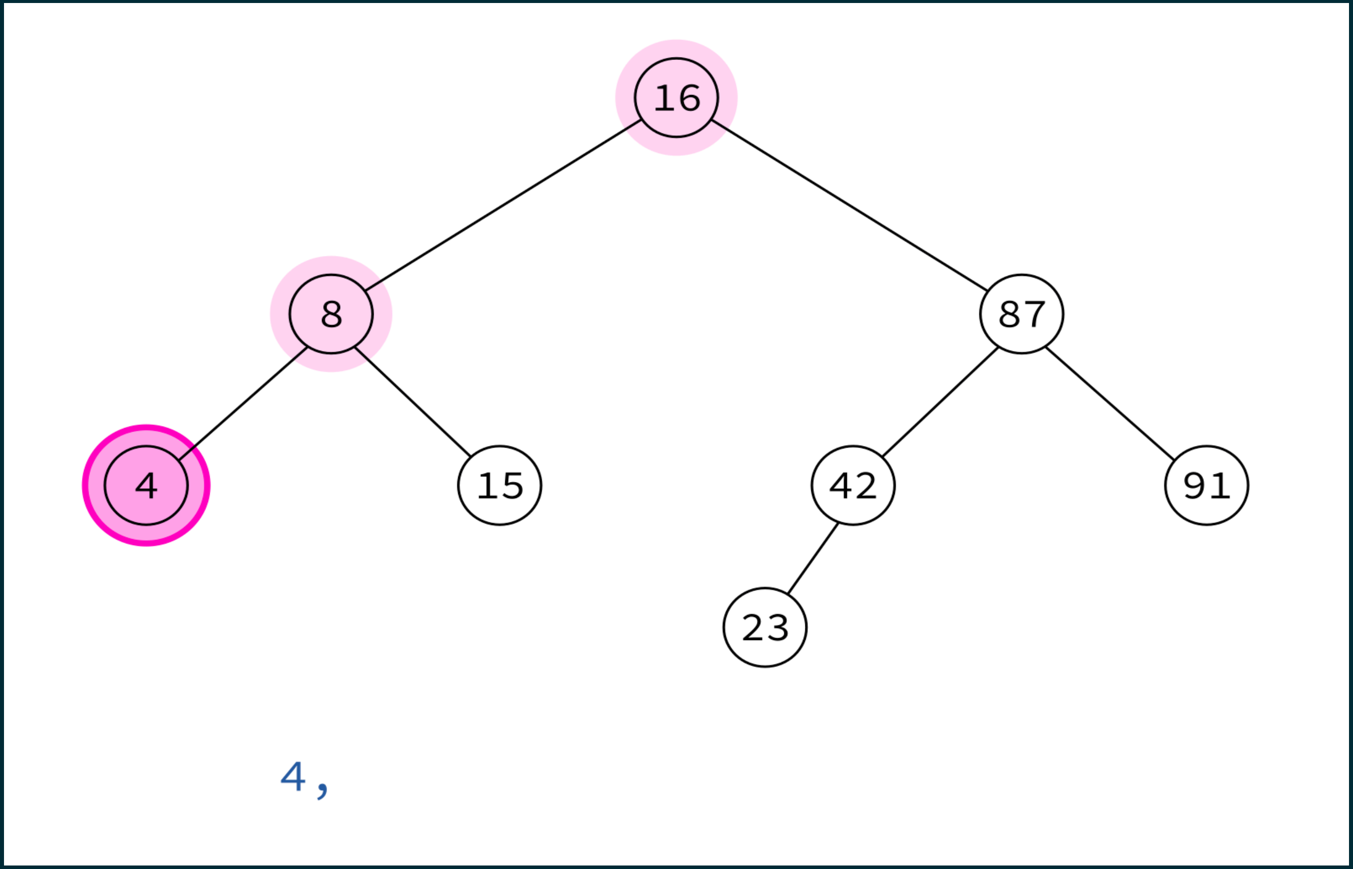
4, 15, 8, 23, 42, 91, 87, 16

INORDER TRAVERSAL



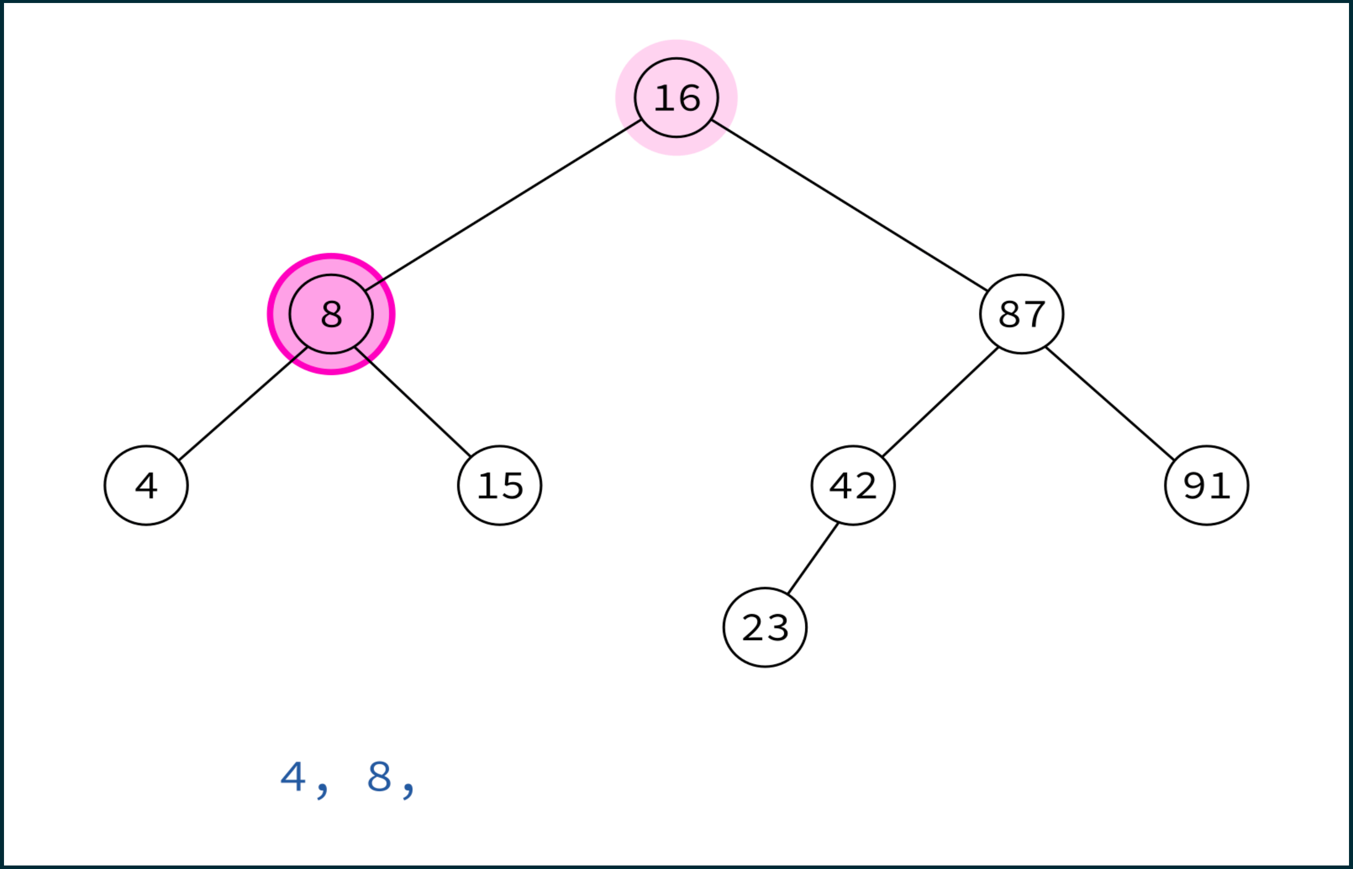
left, **node**, right

INORDER TRAVERSAL



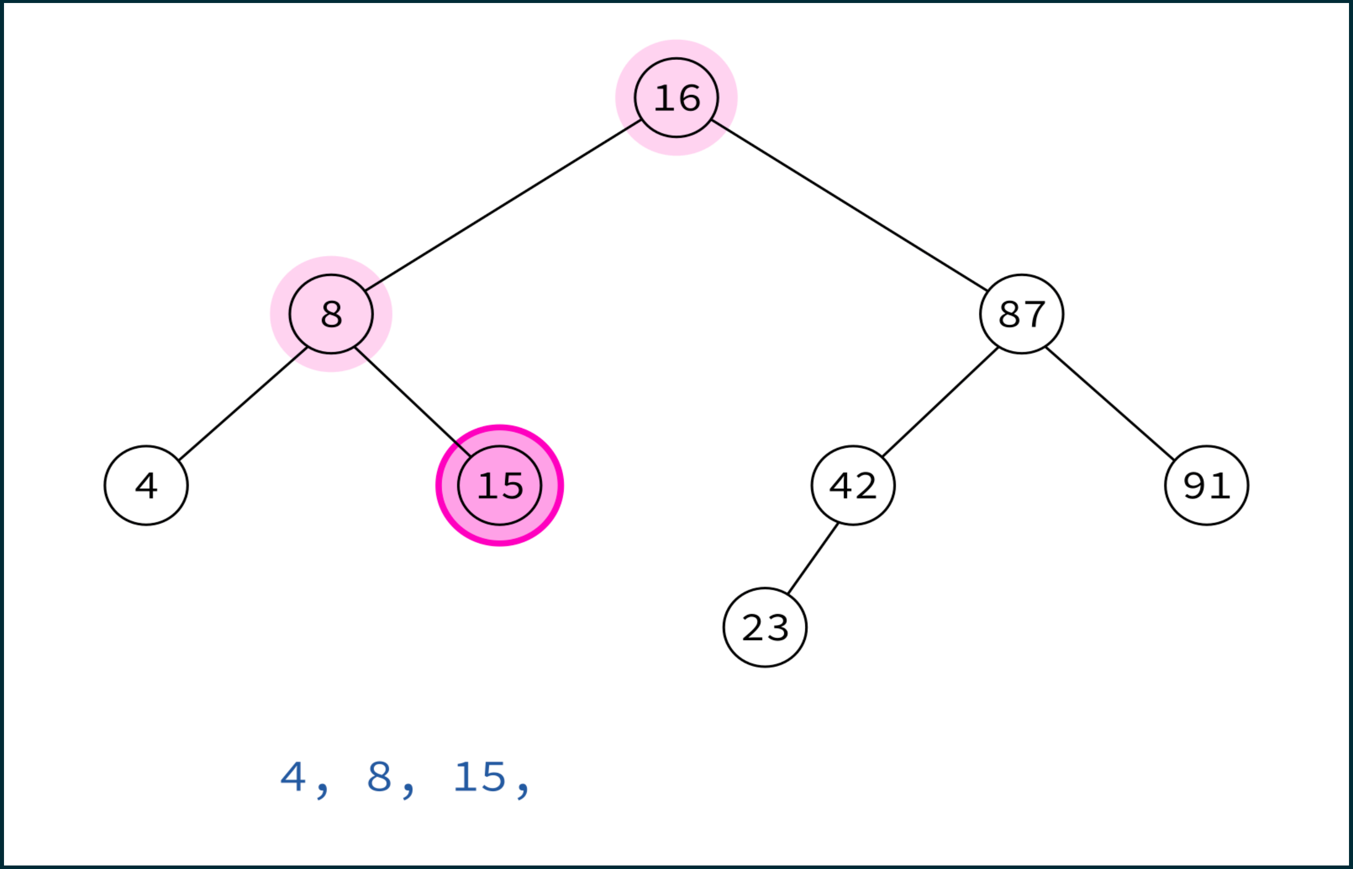
left, **node**, right

INORDER TRAVERSAL



left, **node**, right

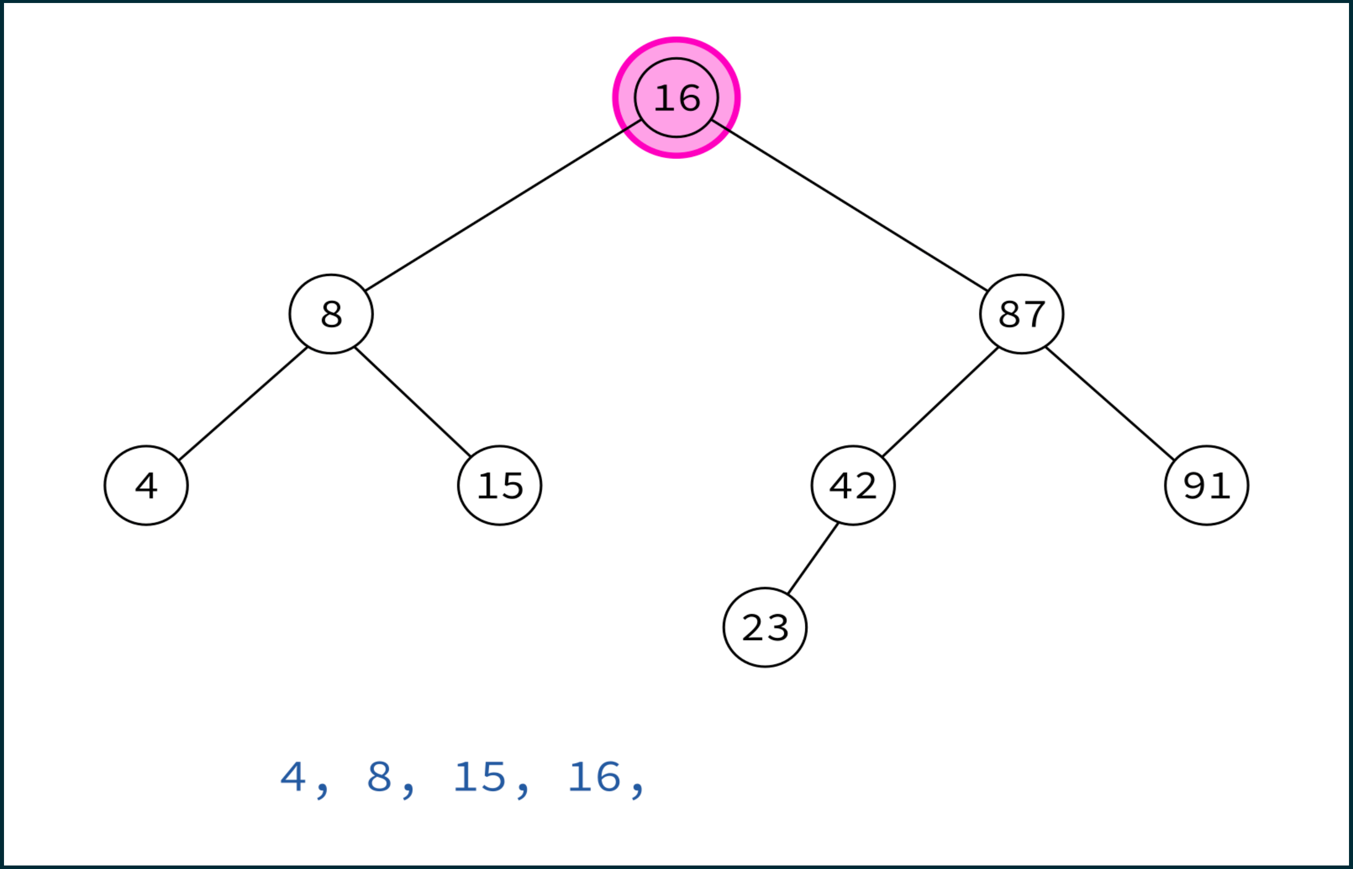
INORDER TRAVERSAL



4, 8, 15,

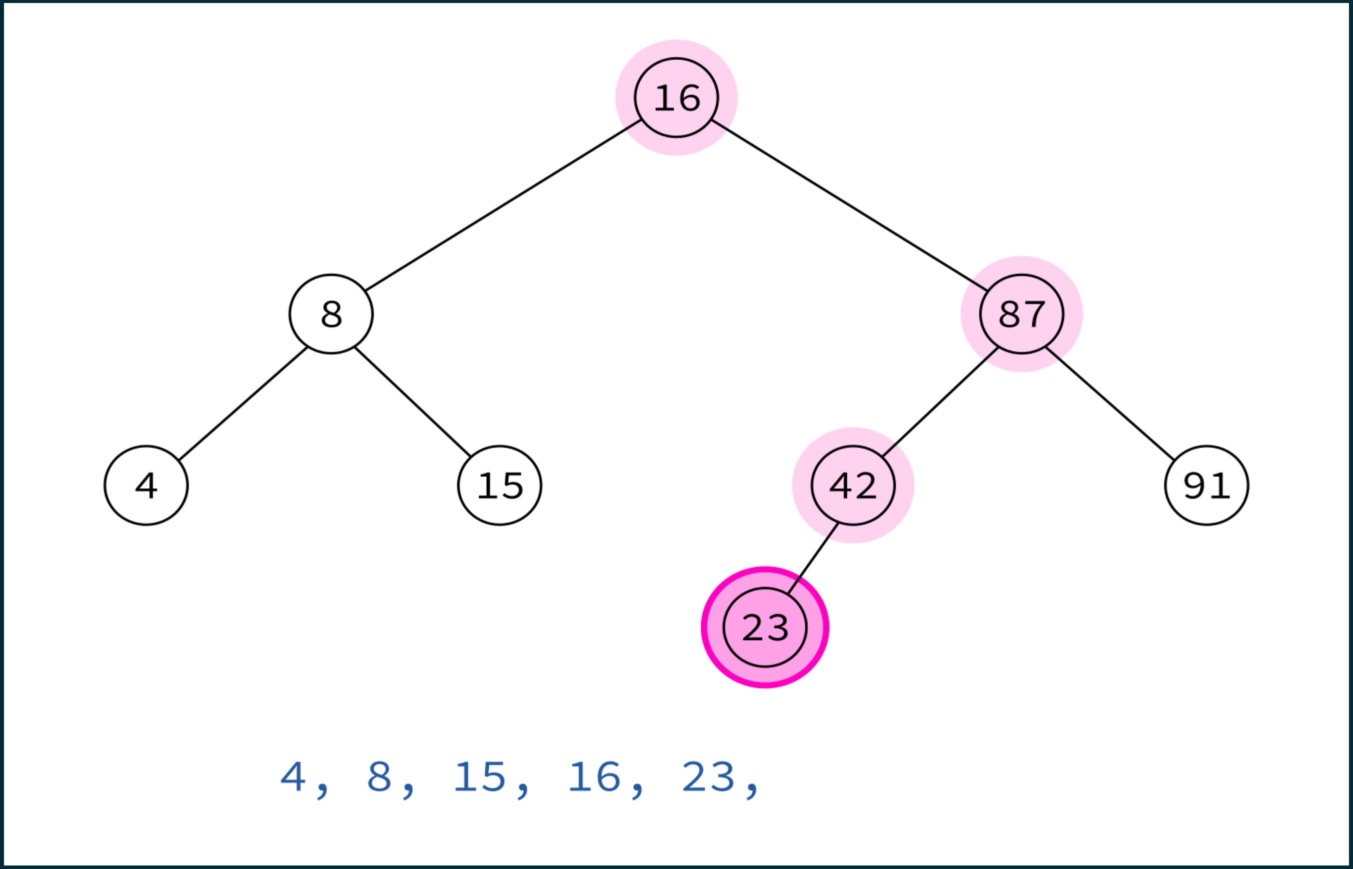
left, **node**, right

INORDER TRAVERSAL



left, **node**, right

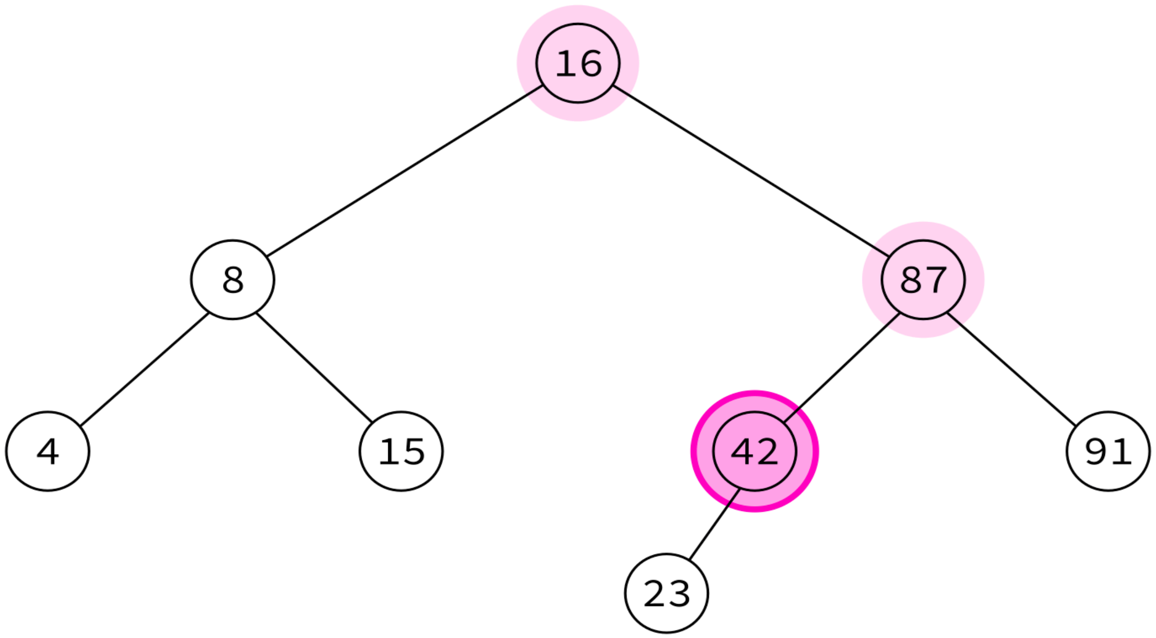
INORDER TRAVERSAL



```
4, 8, 15, 16, 23,
```

left, **node**, right

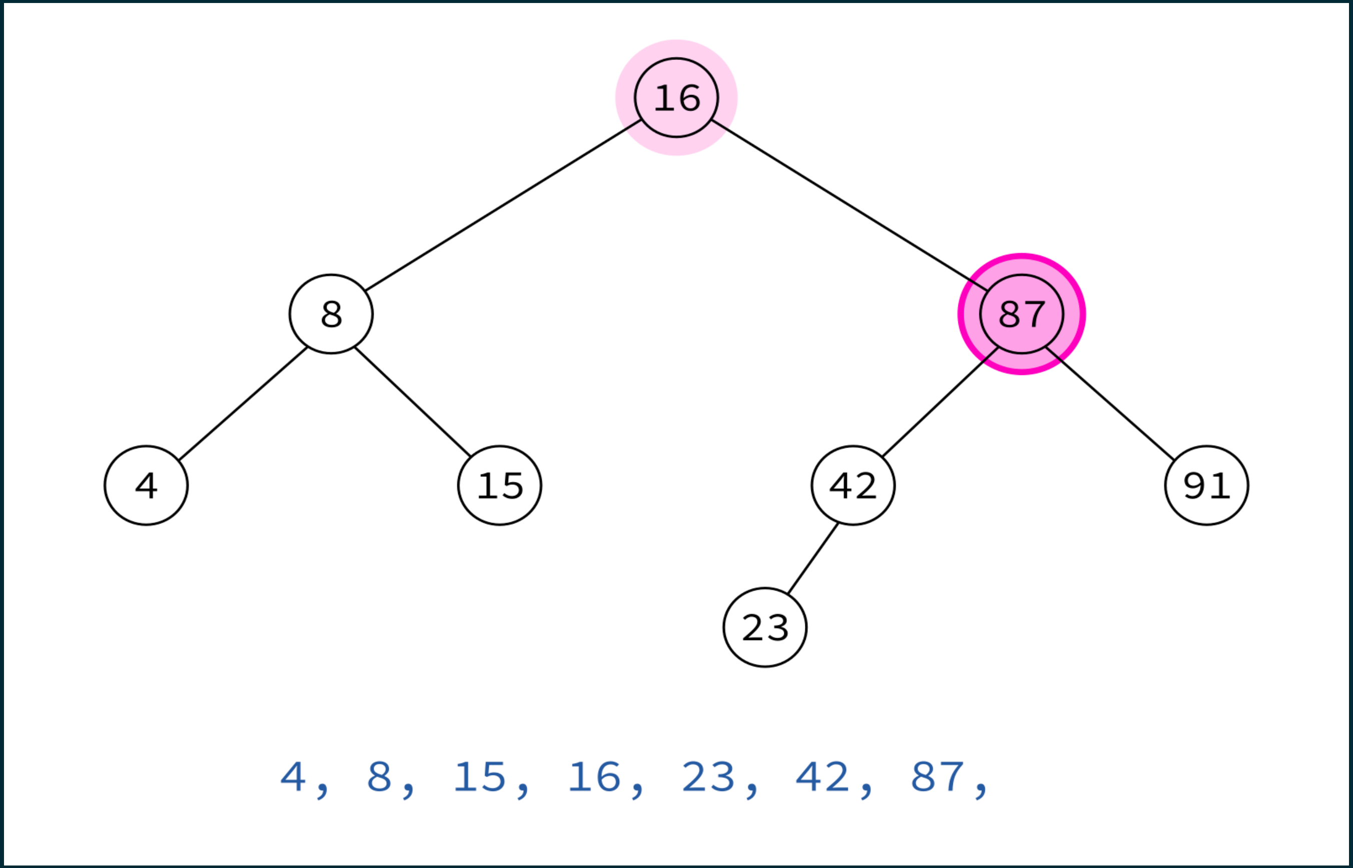
INORDER TRAVERSAL



```
4, 8, 15, 16, 23, 42,
```

left, **node**, right

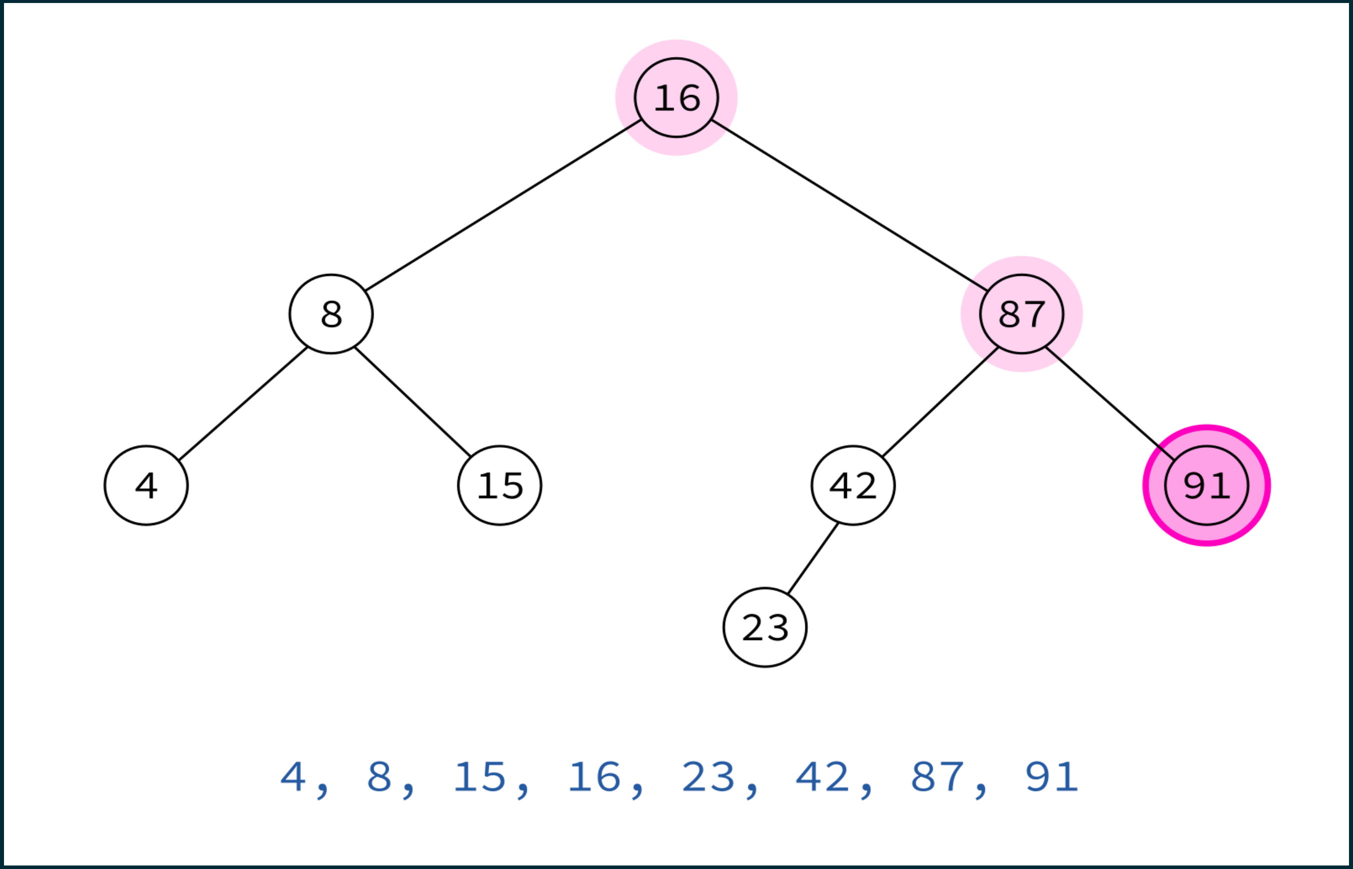
INORDER TRAVERSAL



4, 8, 15, 16, 23, 42, 87,

left, **node**, right

INORDER TRAVERSAL



4, 8, 15, 16, 23, 42, 87, 91

left, **node**, right

INORDER TRAVERSAL

Typical use: Turn a BST into a sorted list of keys.

UNIQUELY DESCRIBING A TREE

Many different binary trees can have the same inorder traversal.

Many different binary trees can have the same preorder traversal.

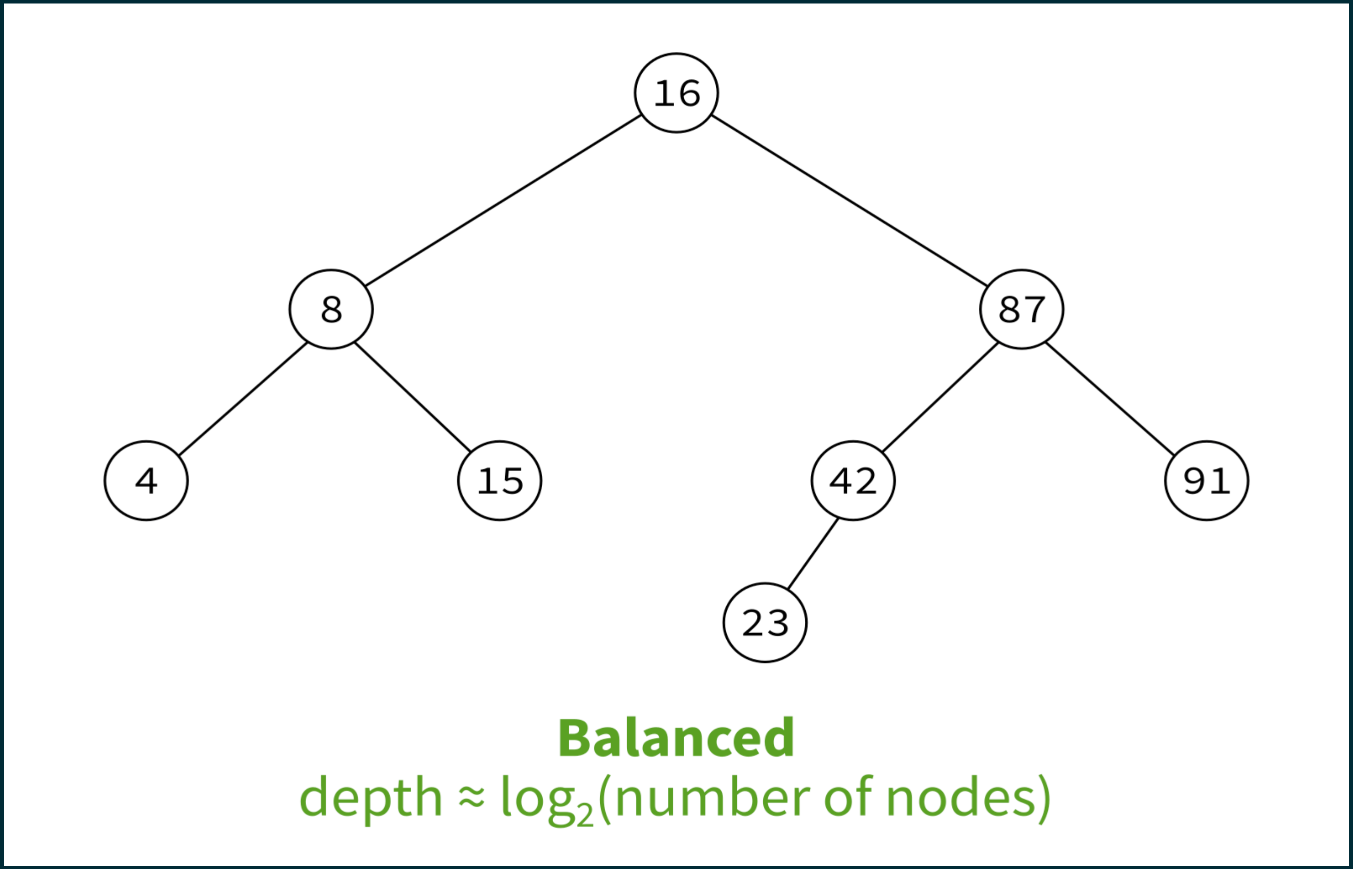
And yet:

Theorem: A binary tree `T` is uniquely determined by its inorder *and* preorder traversals.

LAST WORDS ON BINARY TREES

- BSTs make a lot of data accessible in a few "hops" from the root.
- They are a good choice for mutable data structures involving search operations.
- Deletion of a node is an important feature we didn't implement. (Take MCS 360!)

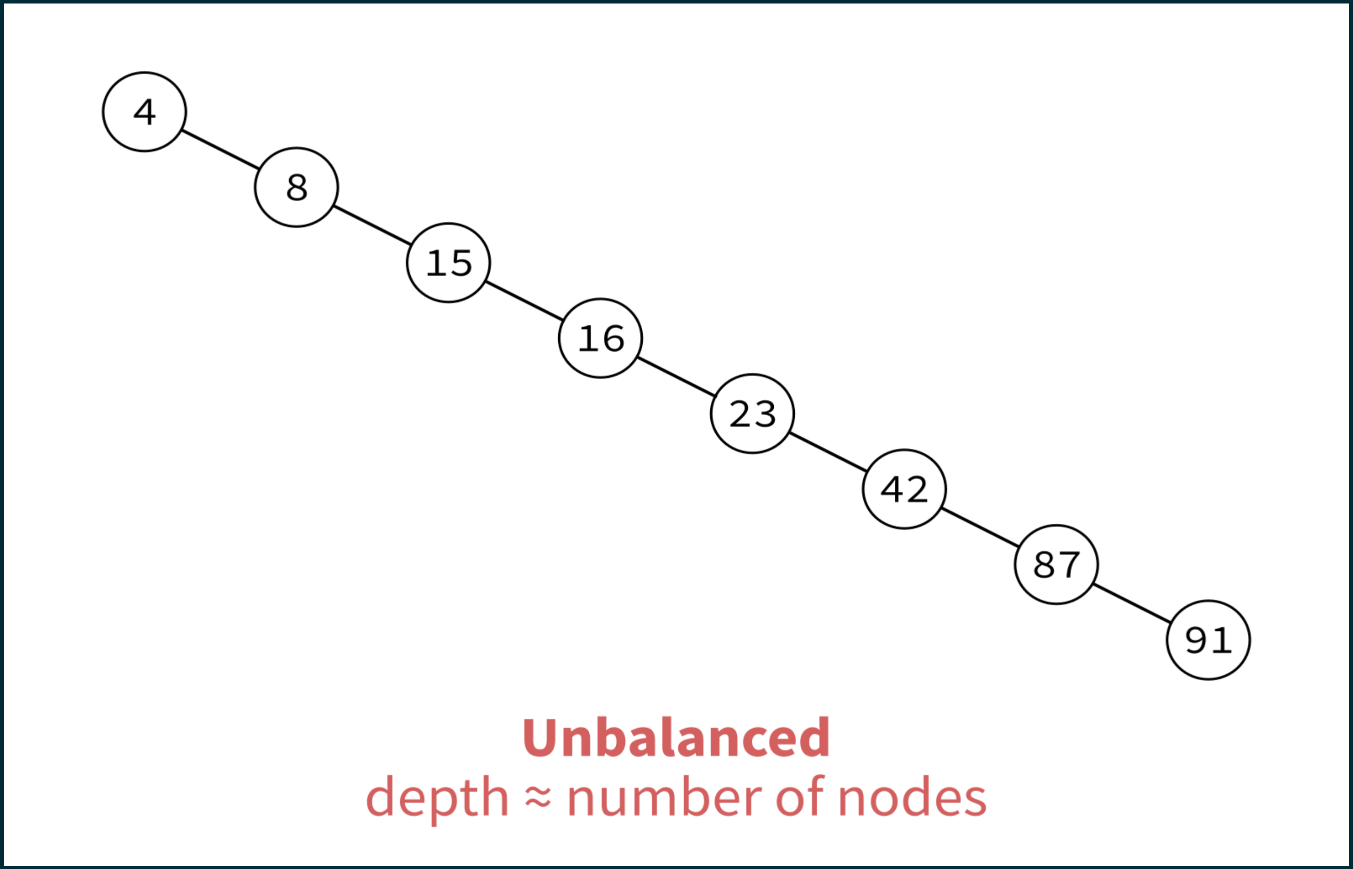
- Unbalanced trees are less efficient.



Balanced
depth ≈ $\log_2$(number of nodes)

MCS 360 usually covers rebalancing operations.

- Unbalanced trees are less efficient.



MCS 360 usually covers rebalancing operations.

REFERENCES

- In optional course texts:
  - *Problem Solving with Algorithms and Data Structures using Python* by Miller and Ranum, discusses binary trees in Chapter 7.
- Elsewhere:
  - Cormen, Leiserson, Rivest, and Stein discusses graph theory and trees in Appendices B.4 and B.5, and binary search trees in Chapter 12.

REVISION HISTORY

- 2022-02-28 Last year's lecture on this topic finalized
- 2023-02-23 Updated for 2023