Trees

Written by Rachel J. Morris, last updated Apr 2, 2018

About

Lists, Queues, and Stacks are all examples of **linear structures**. Now we will work with trees, which are **hierarchical. Binary search trees** in particular offer us an efficient way to store data without making significant concessions on the efficiency of the access and/or insertion times.

Topics

1. Basic terminology

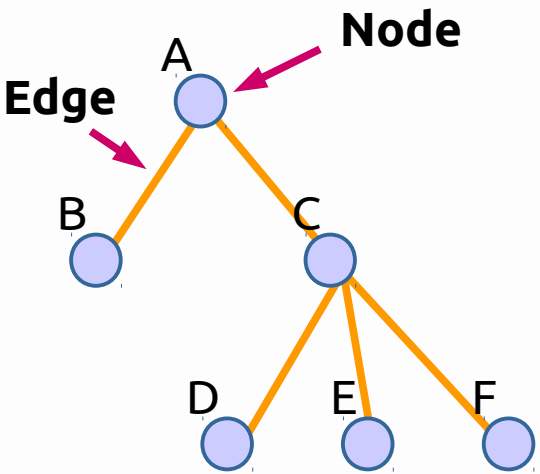
2. Types of trees

3. Tree height, balance, & completeness

4. Binary tree traversals

Basic Terminology

1. Basic Terminology



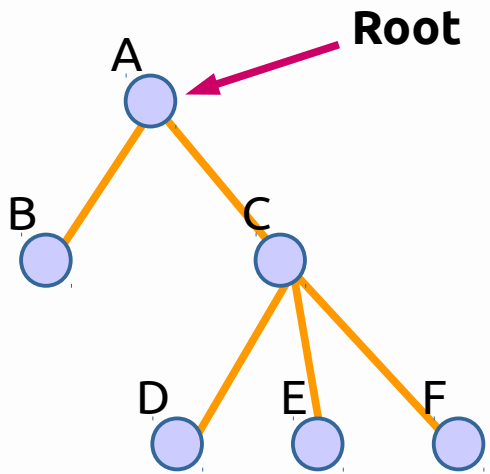
A tree is a type of graph that is completely connected and has no cycles. A tree is made up of **nodes/vertices** and **edges**.

Notes

Node/Vertex: The points in a tree

Edge: The lines that connect two nodes

1. Basic Terminology



Root:

Each tree will have exactly one **root node**, which all other nodes branch out from.

It is the top-most node.

The root has no parent, but every other node has exactly 1 parent.

Notes

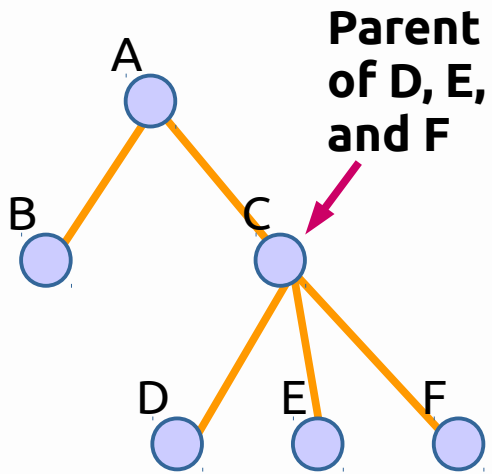
Node/Vertex: The points in a tree

Edge: The lines that connect two nodes

Root: The top-most node of a tree, which has no parent.

Parent of *n*: The node directly above *n*

1. Basic Terminology



Parents:
In a tree, each node has **exactly one parent**, except for the root, which has no parents.

The parent of node *n* is the node directly "above" *n*, on its path back to the root.

Notes

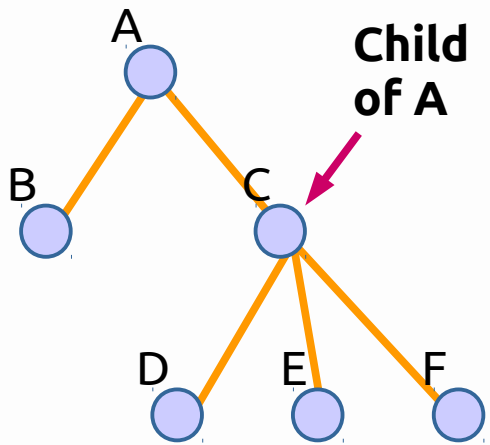
Node/Vertex: The points in a tree

Edge: The lines that connect two nodes

Root: The top-most node of a tree, which has no parent.

Parent of *n*: The node directly above *n*

1. Basic Terminology



Children:
A node can have any amount of children for a generic tree. A child of node *n* are the nodes directly under *n* , branching off from it.

Notes

Node/Vertex: The points in a tree

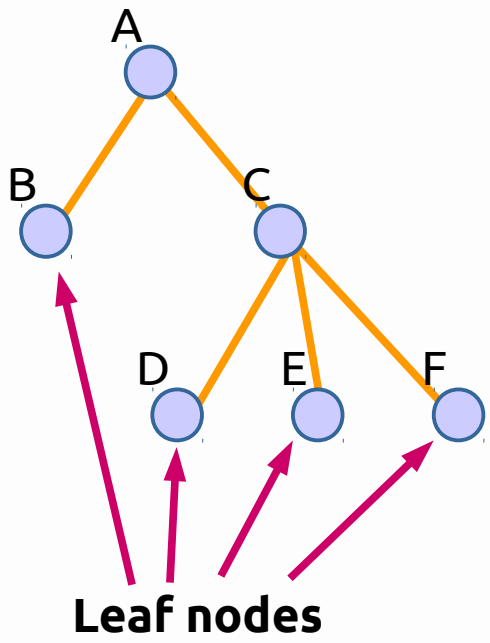
Edge: The lines that connect two nodes

Root: The top-most node of a tree, which has no parent.

Parent of *n*: The node directly above *n*

Child of *n*: A node directly below *n*

1. Basic Terminology



Leaves:
In a tree, the nodes that have no children are known as leaves.

Notes

Node/Vertex: The points in a tree

Edge: The lines that connect two nodes

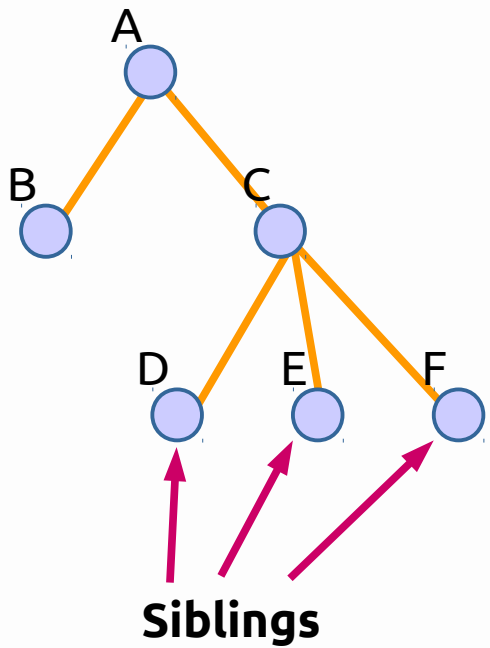
Root: The top-most node of a tree, which has no parent.

Parent of *n*: The node directly above *n*

Child of *n*: A node directly below *n*

Leaf: A node that has no children.

1. Basic Terminology



Siblings:
Children of a single node are siblings of each other.

Notes

Node/Vertex: The points in a tree

Edge: The lines that connect two nodes

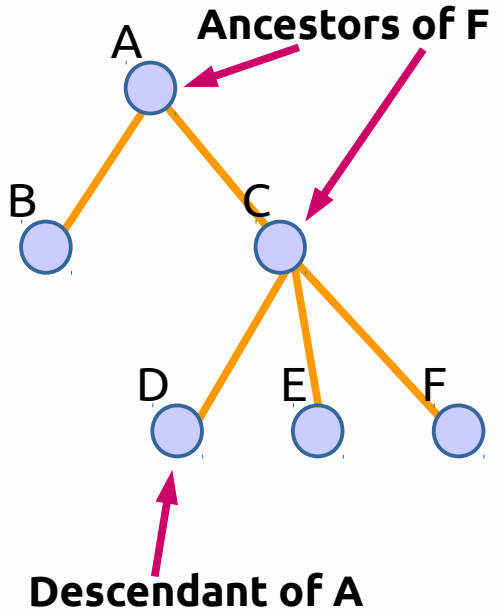
Root: The top-most node of a tree, which has no parent.

Parent of *n*: The node directly above *n*

Child of *n*: A node directly below *n*

Leaf: A node that has no children.

1. Basic Terminology



Ancestor of *n*:
Any node between *n* and the root (including the root).

Descendants of *n*:
Any node derived from node *n*, going all the way down to the leaves.

Notes

Node/Vertex: The points in a tree

Edge: The lines that connect two nodes

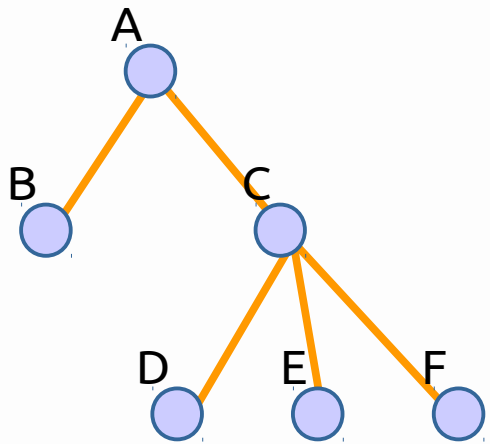
Root: The top-most node of a tree, which has no parent.

Parent of *n*: The node directly above *n*

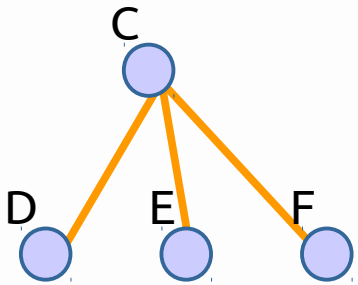
Child of *n*: A node directly below *n*

Leaf: A node that has no children.

1. Basic Terminology



Subtree of C



Subtree of *n*:
Given some tree that contains the node *n*, the subtree of *n* is going to be a new tree with *n* as the root.

Notes

Node/Vertex: The points in a tree

Edge: The lines that connect two nodes

Root: The top-most node of a tree, which has no parent.

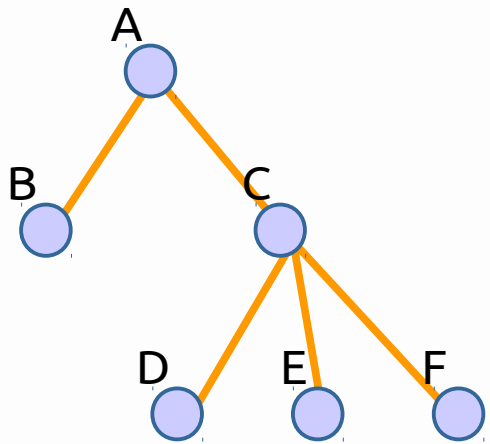
Parent of *n*: The node directly above *n*

Child of *n*: A node directly below *n*

Leaf: A node that has no children.

Types of Trees

2. Types of Trees



General Tree:

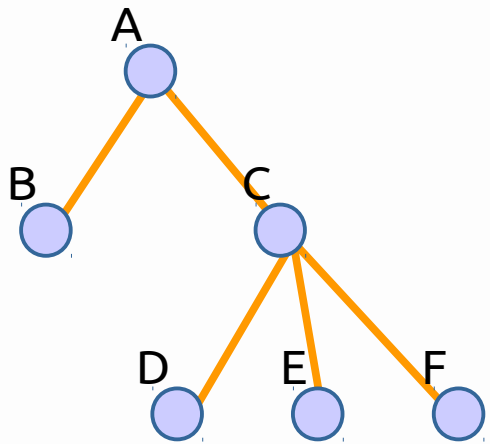
A general tree contains a root and subtrees branching off from that root.

There are no restrictions on the way it must be filled, or how many children a node can have.

Notes

General tree: A root with subtrees; no restrictions.

2. Types of Trees



n-ary tree

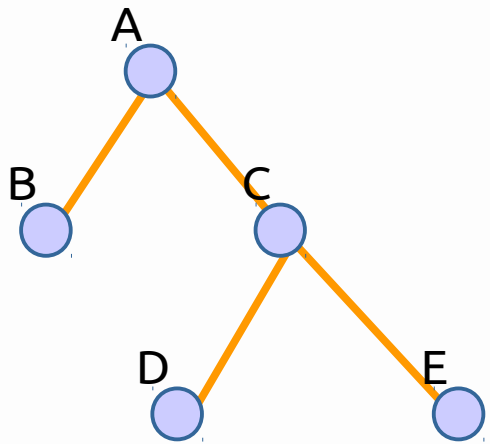
A n-ary tree is a type of tree where each node can have no more than *n* children.

Notes

General tree: A root with subtrees; no restrictions.

n-ary tree: Each node can have no more than *n* children.

2. Types of Trees



Binary tree:
A tree where each node can have no more than 2 children. This means a node can have 0, 1, or 2 children.

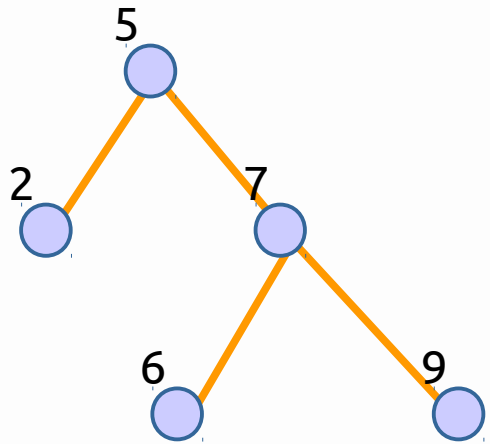
Notes

General tree: A root with subtrees; no restrictions.

n-ary tree: Each node can have no more than *n* children.

Binary tree: A tree where each node can have no more than 2 children.

2. Types of Trees



Binary search tree:
A binary tree that has sorted nodes – for any given node, any nodes to the *left* have a lower value, and any nodes to the *right* have a higher value.

Notes

General tree: A root with subtrees; no restrictions.

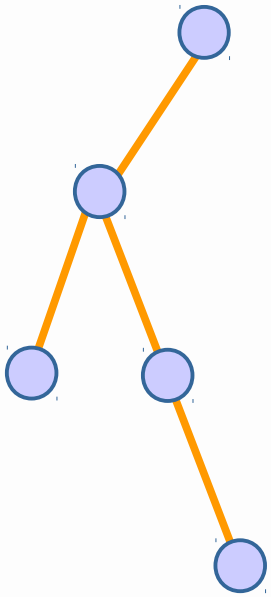
n-ary tree: Each node can have no more than *n* children.

Binary tree: A tree where each node can have no more than 2 children1

Binary search tree: A binary tree that has sorted nodes – for any given node, any nodes to the left have a lower value, and any nodes to the right have a higher value.

Tree height, balance, & completeness

3. Tree height, balance, & completeness



Height of 4

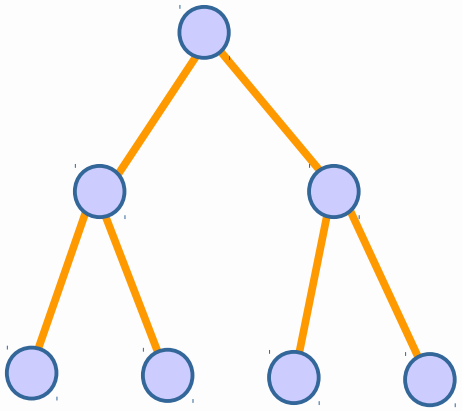
Height of a tree:

The height of a tree is the number of nodes on the longest path from the root to the leaf.

Definition from Data Abstraction & Problem Solving with C++: Walls and Mirrors, 7th ed, Carrano & Henry, page 450

Notes

3. Tree height, balance, & completeness



A full binary tree of height 3

Full binary tree:

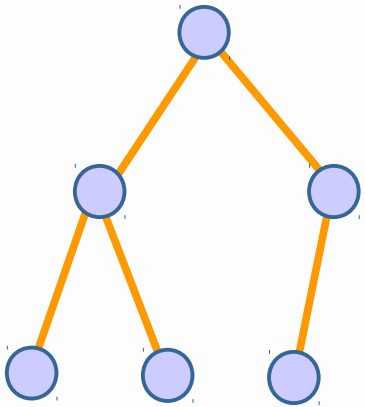
For a binary tree of height *h*, a full binary tree is one where **all nodes that are at a level less than *h* have two children each.**

Definition from Data Abstraction & Problem Solving with C++: Walls and Mirrors, 7th ed, Carrano & Henry, page 451

Notes

Full binary tree:
For a binary tree of height *h*, a full binary tree is one where **all nodes that are at a level less than *h* have two children each.**

3. Tree height, balance, & completeness



A complete binary tree of height 3

Complete binary tree:

For a binary tree of height *h*, a complete binary tree is one that is full down to level *h – 1* , with level *h* filled in from left-to-right.

Formally,
1. All nodes at level *h – 2* and above have two children each.

2. When a node at level *h – 1* has children, all nodes to its left at the same level have two children each

3. When a node at level *h – 1* has one child, it is a left chid.

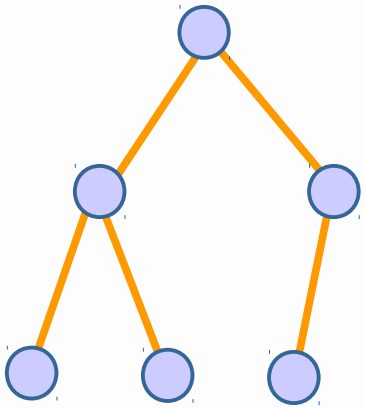
Definition from Data Abstraction & Problem Solving with C++: Walls and Mirrors, 7th ed, Carrano & Henry, page 451

Notes

Full binary tree:
For a binary tree of height *h*, a full binary tree is one where **all nodes that are at a level less than *h* have two children each.**

Complete binary tree:
For a binary tree of height *h*, a complete binary tree is one that is full down to level *h – 1* , with level *h* filled in from left-to-right.

3. Tree height, balance, & completeness



It's height balanced

Balanced binary tree:

A binary tree is **height balanced** if the height of any node's right subtree differs from the hight of the node's left subtree by no more than 1.

Definition from Data Abstraction & Problem Solving with C++: Walls and Mirrors, 7th ed, Carrano & Henry, page 452

Notes

Full binary tree:
For a binary tree of height *h*, a full binary tree is one where **all nodes that are at a level less than *h* have two children each.**

Complete binary tree:
For a binary tree of height *h*, a complete binary tree is one that is full down to level *h – 1* , with level *h* filled in from left-to-right.

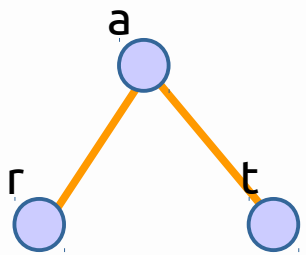
Binary tree traversals

4. Binary tree traversals

We can recurse through a tree in three different ways: Pre-order, In-order, and Post-order.

Notes

4. Binary tree traversals



Preorder:
a-r-t

Pre-order:

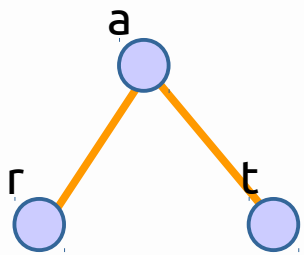
At some node n…

1. Display node n ’s value
2. Traverse the left subtree
3. Traverse the right subtree

Notes

Preorder:
- **Display self**
- **Recurse to left**
- **Recurse to right**

4. Binary tree traversals



Preorder:
a-r-t

Inorder:
r-a-t

In-order:

At some node n…

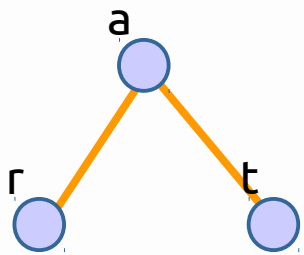
1. Traverse the left subtree
2. Display node n ’s value
3. Traverse the right subtree

Notes

Preorder:
- **Display self**
- **Recurse to left**
- **Recurse to right**

Inorder:
- **Recurse to left**
- **Display self**
- **Recurse to right**

4. Binary tree traversals



Preorder:
a-r-t

Inorder:
r-a-t

Postorder:
r-t-a

Post-order:

At some node n…

1. Traverse the left subtree
2. Traverse the right subtree
3. Display node n ’s value

Notes

Preorder:
- **Display self**
- **Recurse to left**
- **Recurse to right**

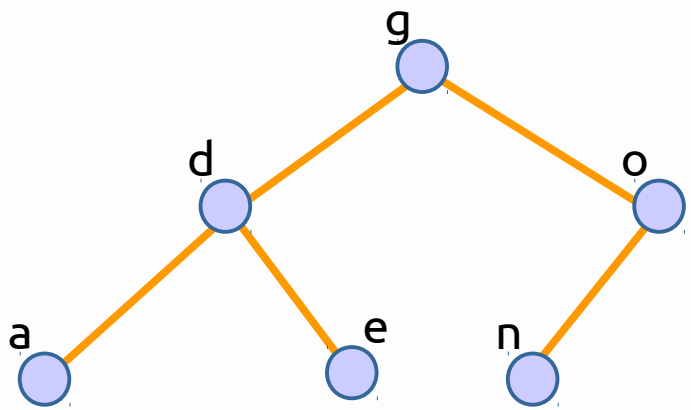
Inorder:
- **Recurse to left**
- **Display self**
- **Recurse to right**

Postorder:
- **Recurse to left**
- **Recurse to right**
- **Display self**

4. Binary tree traversals

A more complex example:

```
Preorder( n ):
    Print( self )
    Preorder( n→left )
    Preorder( n→right )
```



Notes

Preorder:
- **Display self**
- **Recurse to left**
- **Recurse to right**

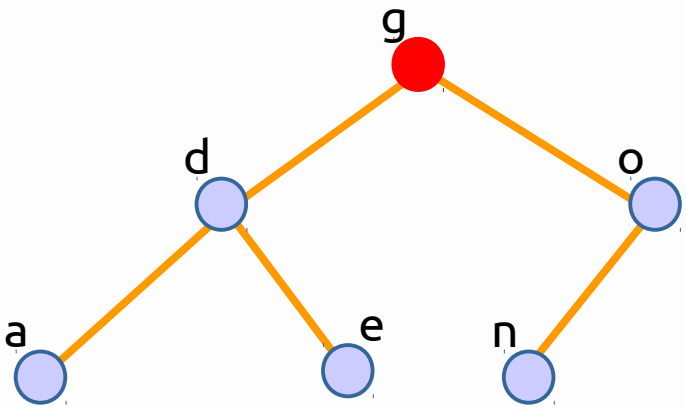
Inorder:
- **Recurse to left**
- **Display self**
- **Recurse to right**

Postorder:
- **Recurse to left**
- **Recurse to right**
- **Display self**

4. Binary tree traversals

A more complex example:

```
Preorder( n ):
    Print( self )
    Preorder( n→left )
    Preorder( n→right )
```



We always start at the root, so we'd begin by calling

`Preorder( g )`

Notes

Preorder:
- **Display self**
- **Recurse to left**
- **Recurse to right**

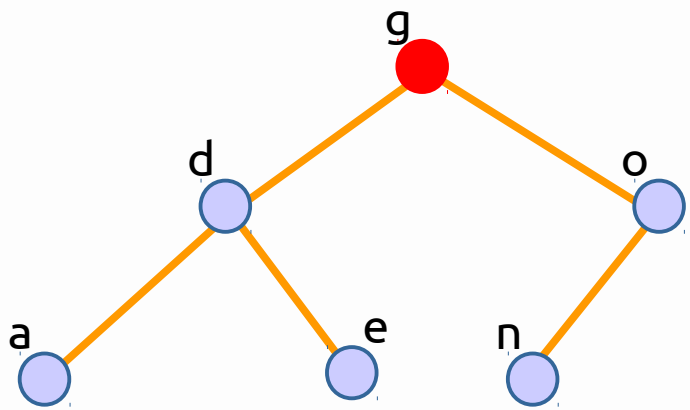
Inorder:
- **Recurse to left**
- **Display self**
- **Recurse to right**

Postorder:
- **Recurse to left**
- **Recurse to right**
- **Display self**

4. Binary tree traversals

A more complex example:

```
Preorder( n ):
    Print( self )
    Preorder( n→left )
    Preorder( n→right )
```



Output: g

First, we print the node itself, then traverse to the left.

`Preorder( g→left )`

Notes

Preorder:
- **Display self**
- **Recurse to left**
- **Recurse to right**

Inorder:
- **Recurse to left**
- **Display self**
- **Recurse to right**

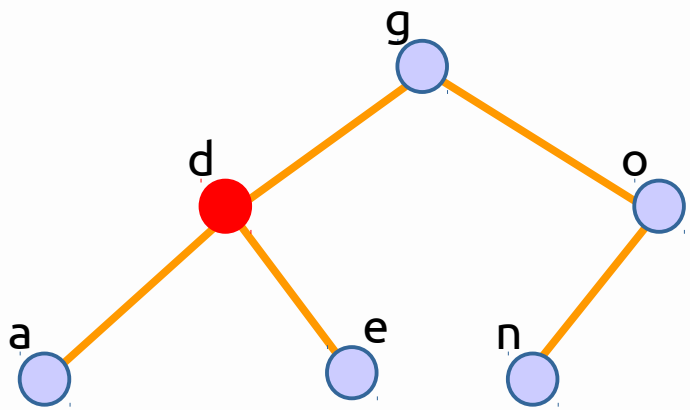
Postorder:
- **Recurse to left**
- **Recurse to right**
- **Display self**

4. Binary tree traversals

A more complex example:

```
Preorder( n ):
    Print( self )
    Preorder( n→left )
    Preorder( n→right )
```

Display self, traverse left.



Output: g d

Notes

Preorder:
- **Display self**
- **Recurse to left**
- **Recurse to right**

Inorder:
- **Recurse to left**
- **Display self**
- **Recurse to right**

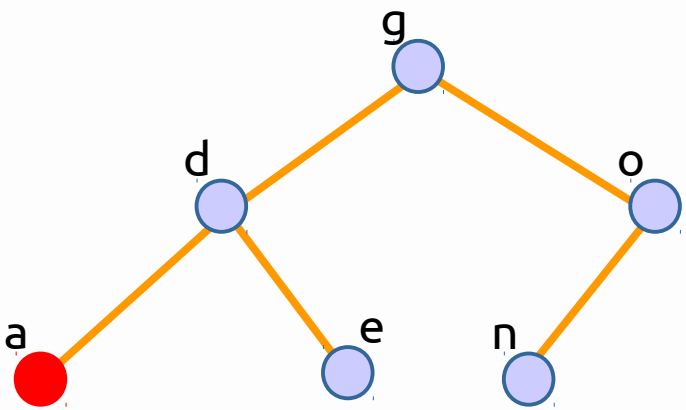
Postorder:
- **Recurse to left**
- **Recurse to right**
- **Display self**

4. Binary tree traversals

A more complex example:

```
Preorder( n ):
    Print( self )
    Preorder( n→left )
    Preorder( n→right )
```

Display self, no other children, so return back to d.



Output: g d a

Notes

Preorder:
- **Display self**
- **Recurse to left**
- **Recurse to right**

Inorder:
- **Recurse to left**
- **Display self**
- **Recurse to right**

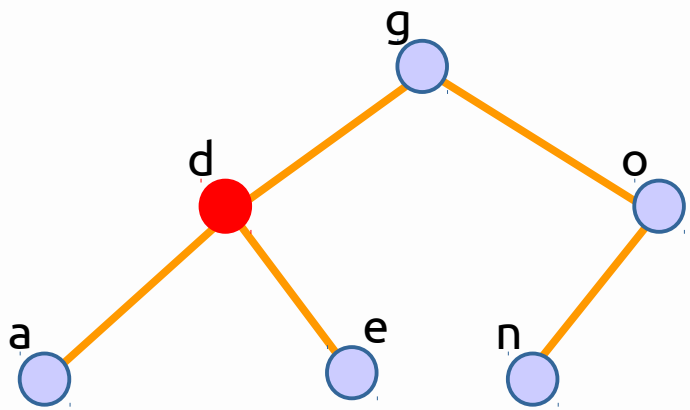
Postorder:
- **Recurse to left**
- **Recurse to right**
- **Display self**

4. Binary tree traversals

A more complex example:

```
Preorder( n ):
    Print( self )
    Preorder( n→left )
    Preorder( n→right )
```

At d, we've displayed self, and traversed left, so now we traverse right.



Output: g d a

Notes

Preorder:
- **Display self**
- **Recurse to left**
- **Recurse to right**

Inorder:
- **Recurse to left**
- **Display self**
- **Recurse to right**

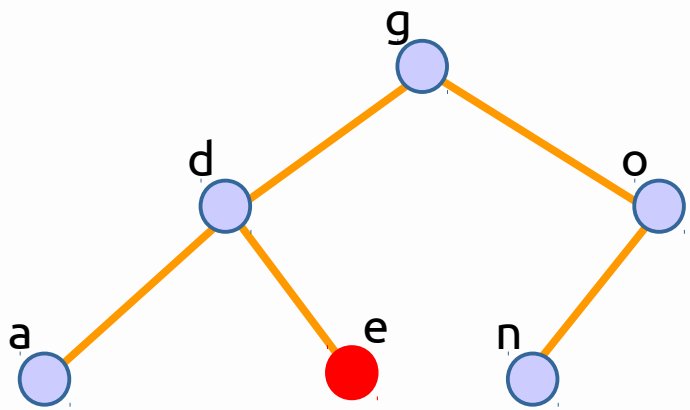
Postorder:
- **Recurse to left**
- **Recurse to right**
- **Display self**

4. Binary tree traversals

A more complex example:

```
Preorder( n ):
    Print( self )
    Preorder( n→left )
    Preorder( n→right )
```

Display e, then return back to d.



Output: g d a e

Notes

Preorder:
- **Display self**
- **Recurse to left**
- **Recurse to right**

Inorder:
- **Recurse to left**
- **Display self**
- **Recurse to right**

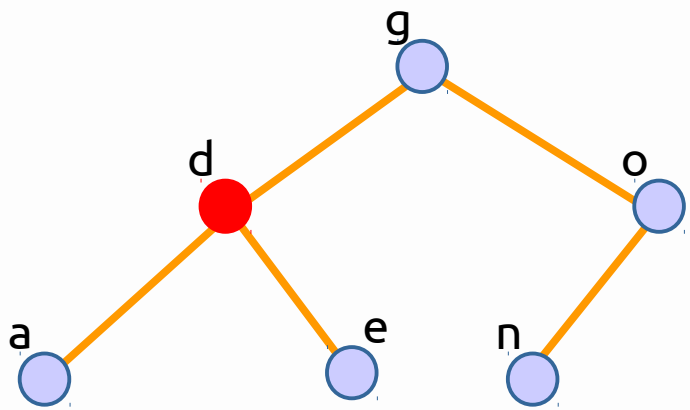
Postorder:
- **Recurse to left**
- **Recurse to right**
- **Display self**

4. Binary tree traversals

A more complex example:

```
Preorder( n ):
    Print( self )
    Preorder( n→left )
    Preorder( n→right )
```

Finished all steps for d, so return back to g.



Output: g d a e

Notes

Preorder:
- **Display self**
- **Recurse to left**
- **Recurse to right**

Inorder:
- **Recurse to left**
- **Display self**
- **Recurse to right**

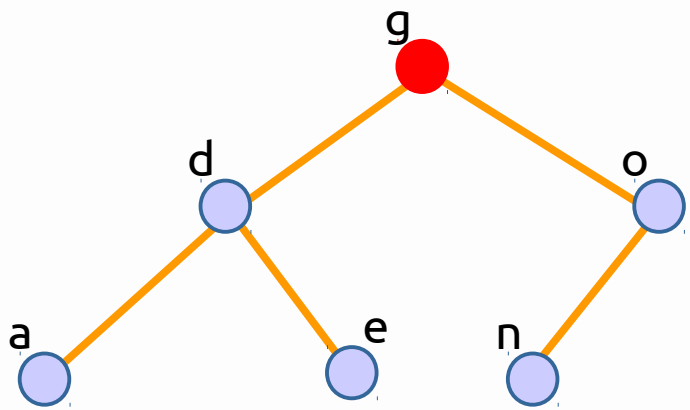
Postorder:
- **Recurse to left**
- **Recurse to right**
- **Display self**

4. Binary tree traversals

A more complex example:

```
Preorder( n ):
    Print( self )
    Preorder( n→left )
    Preorder( n→right )
```

Time to traverse right.



Output: g d a e

Notes

Preorder:
- **Display self**
- **Recurse to left**
- **Recurse to right**

Inorder:
- **Recurse to left**
- **Display self**
- **Recurse to right**

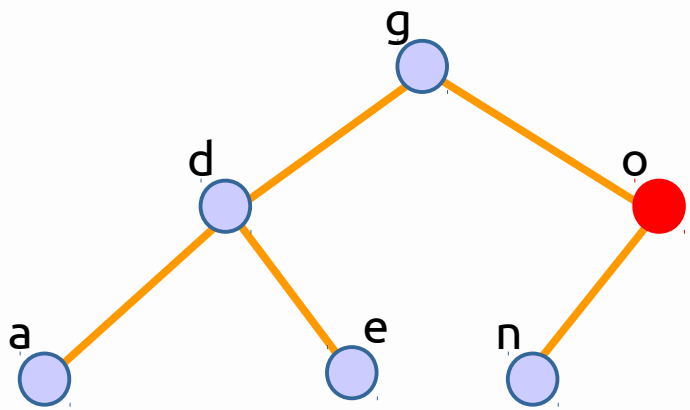
Postorder:
- **Recurse to left**
- **Recurse to right**
- **Display self**

4. Binary tree traversals

A more complex example:

```
Preorder( n ):
    Print( self )
    Preorder( n→left )
    Preorder( n→right )
```

Display o, then go left…



Output: g d a e o

Notes

Preorder:
- **Display self**
- **Recurse to left**
- **Recurse to right**

Inorder:
- **Recurse to left**
- **Display self**
- **Recurse to right**

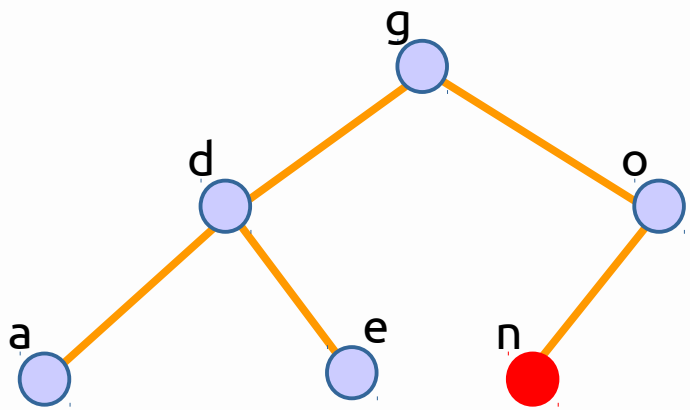
Postorder:
- **Recurse to left**
- **Recurse to right**
- **Display self**

4. Binary tree traversals

A more complex example:

```
Preorder( n ):
    Print( self )
    Preorder( n→left )
    Preorder( n→right )
```

Display n, and then begin returning.



Output: g d a e o n

Notes

Preorder:
- **Display self**
- **Recurse to left**
- **Recurse to right**

Inorder:
- **Recurse to left**
- **Display self**
- **Recurse to right**

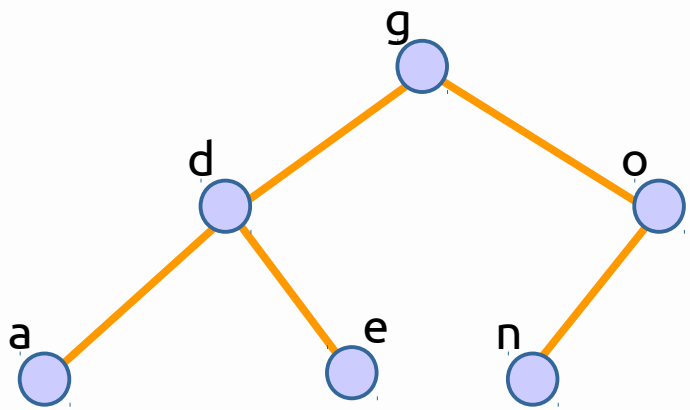
Postorder:
- **Recurse to left**
- **Recurse to right**
- **Display self**

4. Binary tree traversals

A more complex example:

```
Preorder( n ):
    Print( self )
    Preorder( n→left )
    Preorder( n→right )
```

All done traversing.



Output: g d a e o n

Notes

Preorder:
- **Display self**
- **Recurse to left**
- **Recurse to right**

Inorder:
- **Recurse to left**
- **Display self**
- **Recurse to right**

Postorder:
- **Recurse to left**
- **Recurse to right**
- **Display self**

Conclusion

We will cover the specifics of Binary Search Trees in another lecture, since that will be more of what we focus on, programming-wise.

Make sure you understand the core terminology for trees.