Trees

Tiziana Ligorio
Hunter College of The City University of New York

Today's Plan

Trees

Binary Tree ADT

Announcements

ADT Operations we have seen so far

Bag, List, Stack, Queue

Add data to collection
Remove data from collection
Retrieve data from collection

Stack and Queue always **position based**

Bag, retrieval always **value based** (there are no positions)

List has **both**.

For all of them, data organization is linear

Tree

Non-linear structure

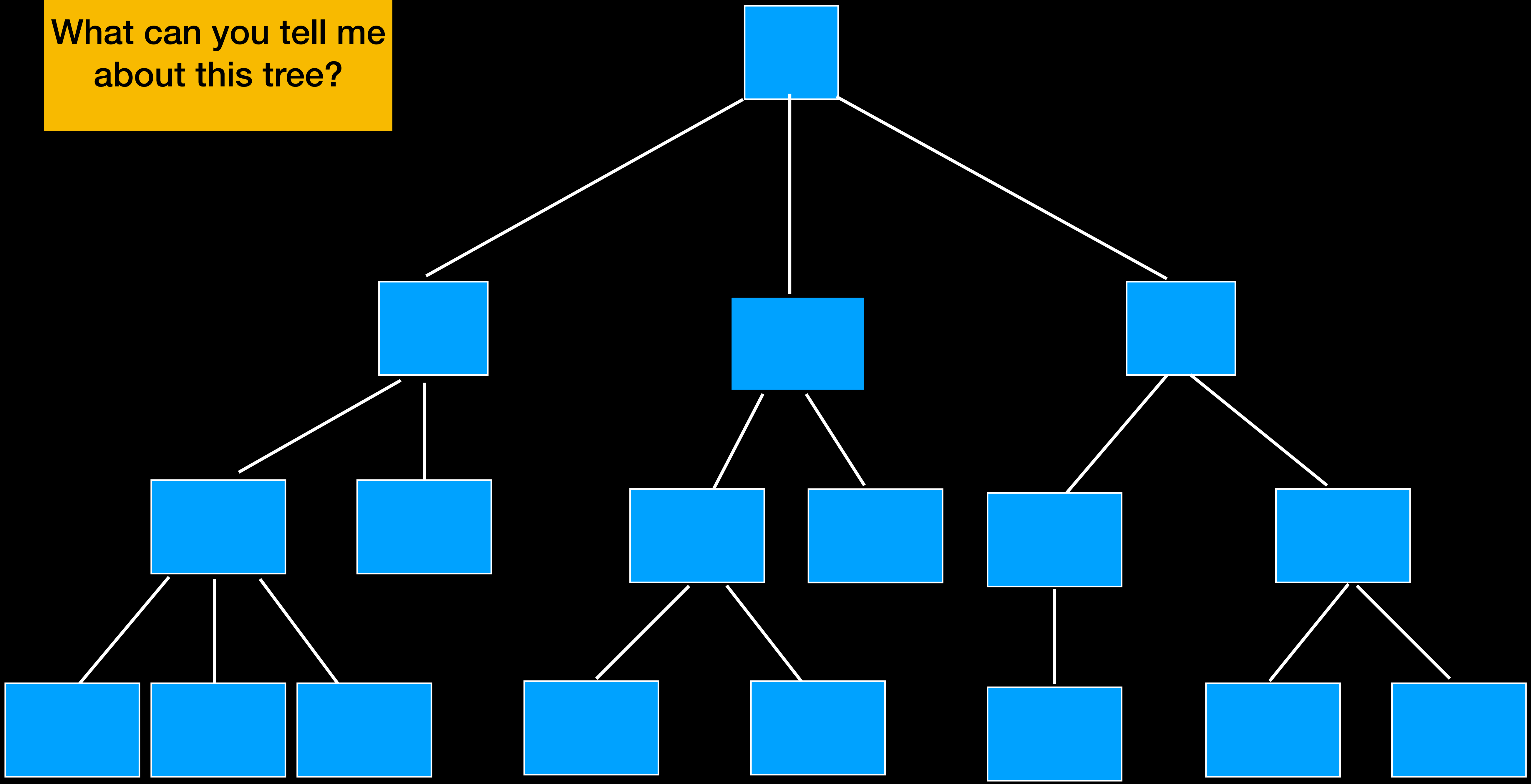
A special type of graph

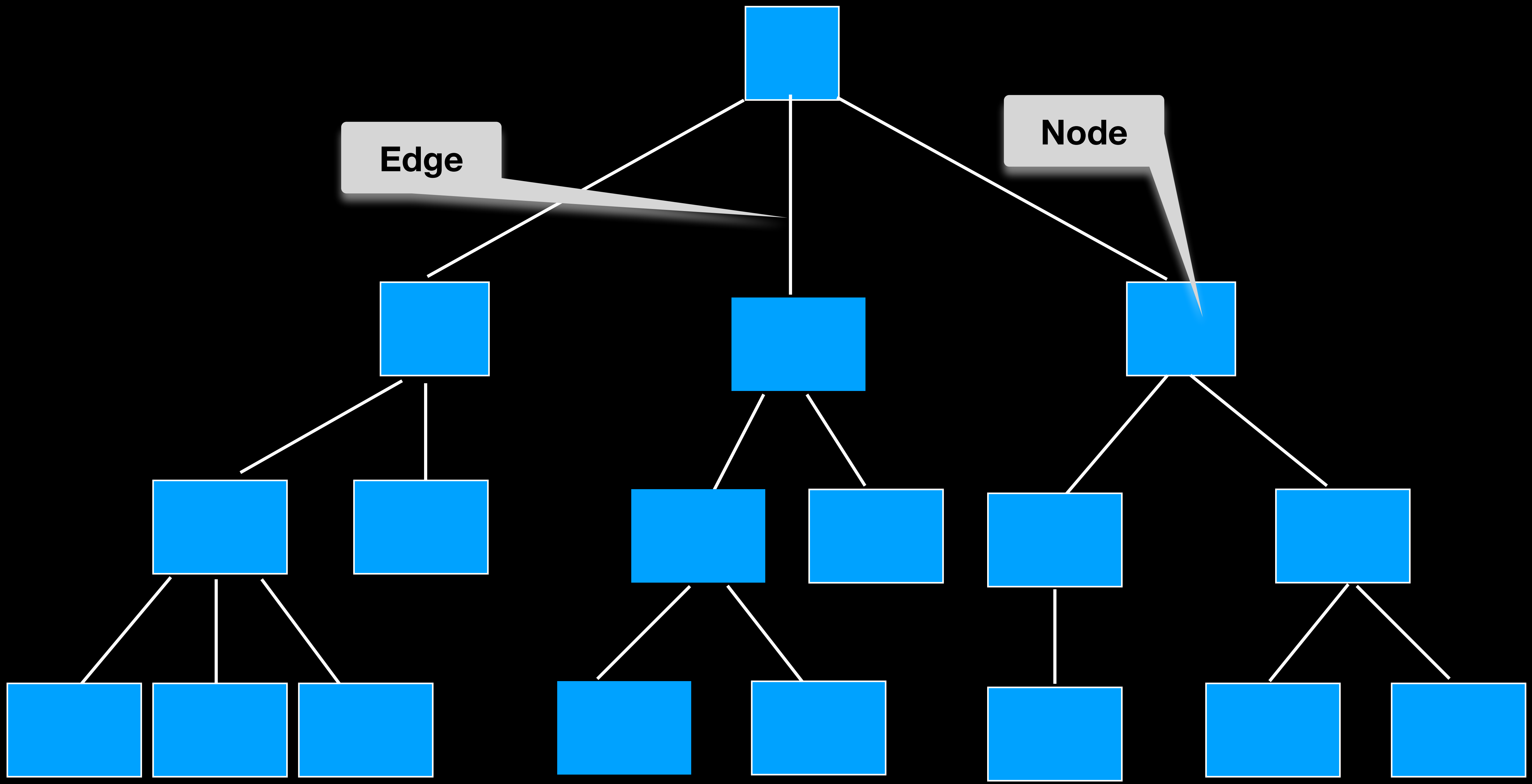
Can represent relationships

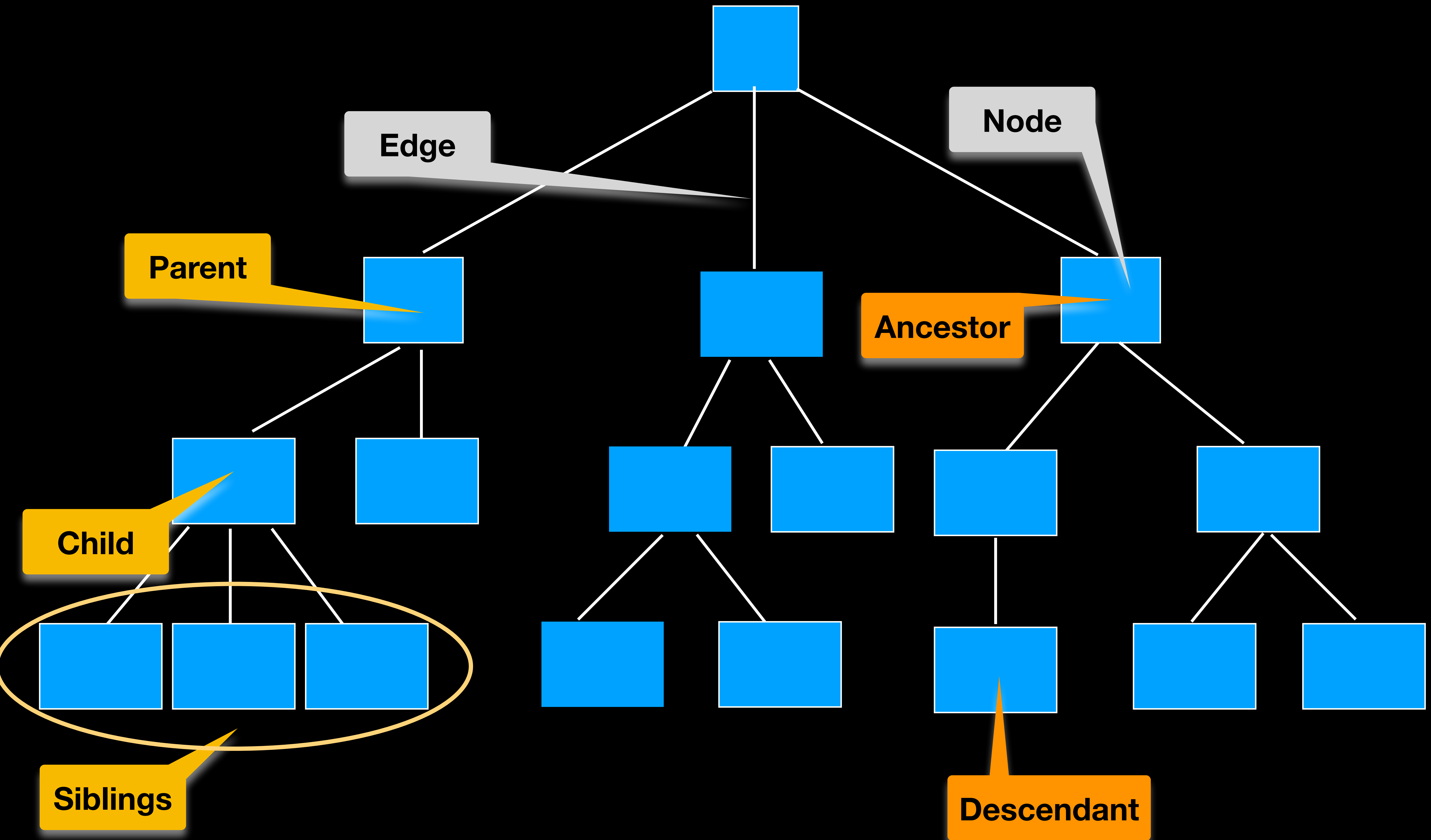
Hierarchical (directional) organization

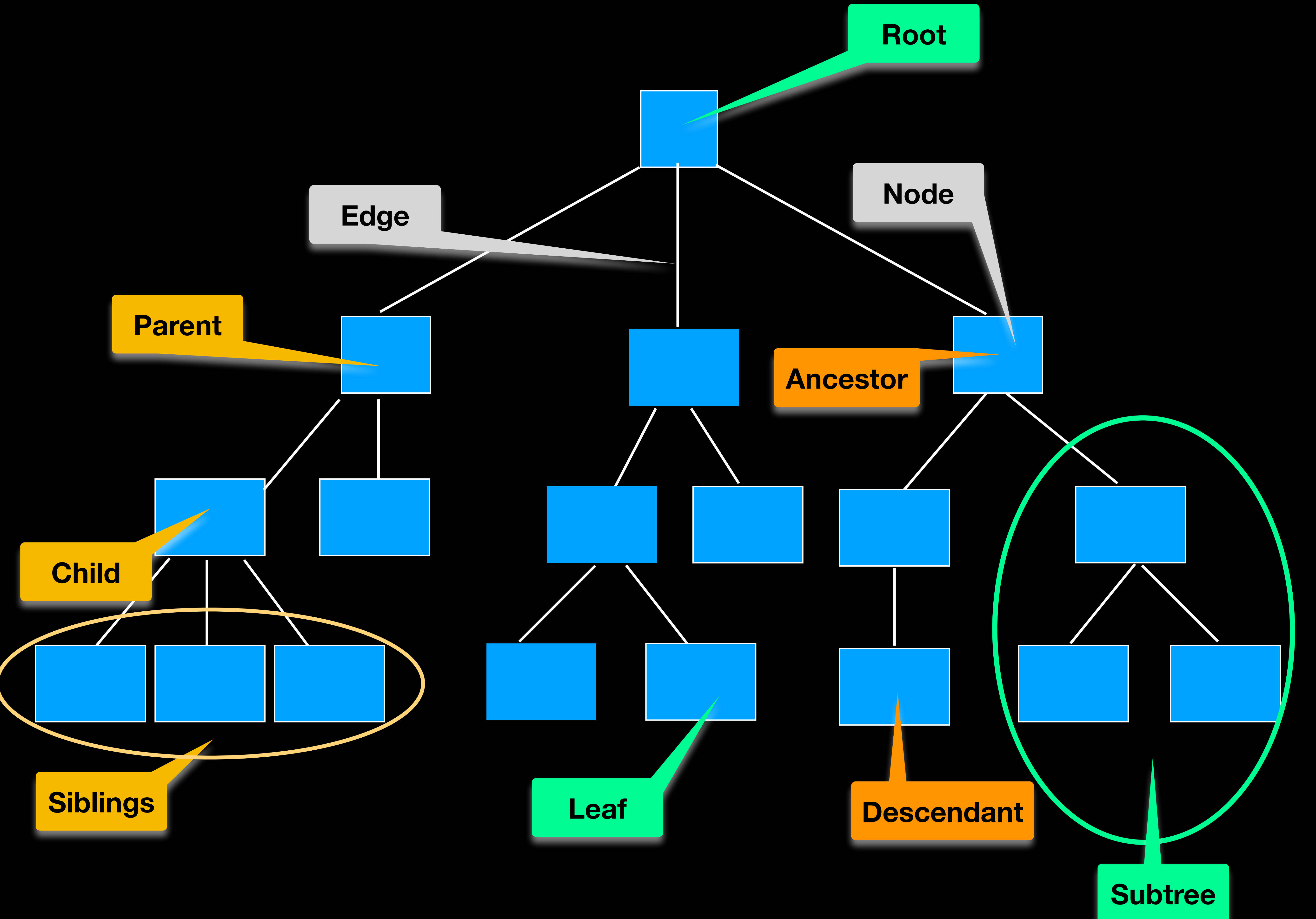
(E.g. family tree)

What can you tell me about this tree?

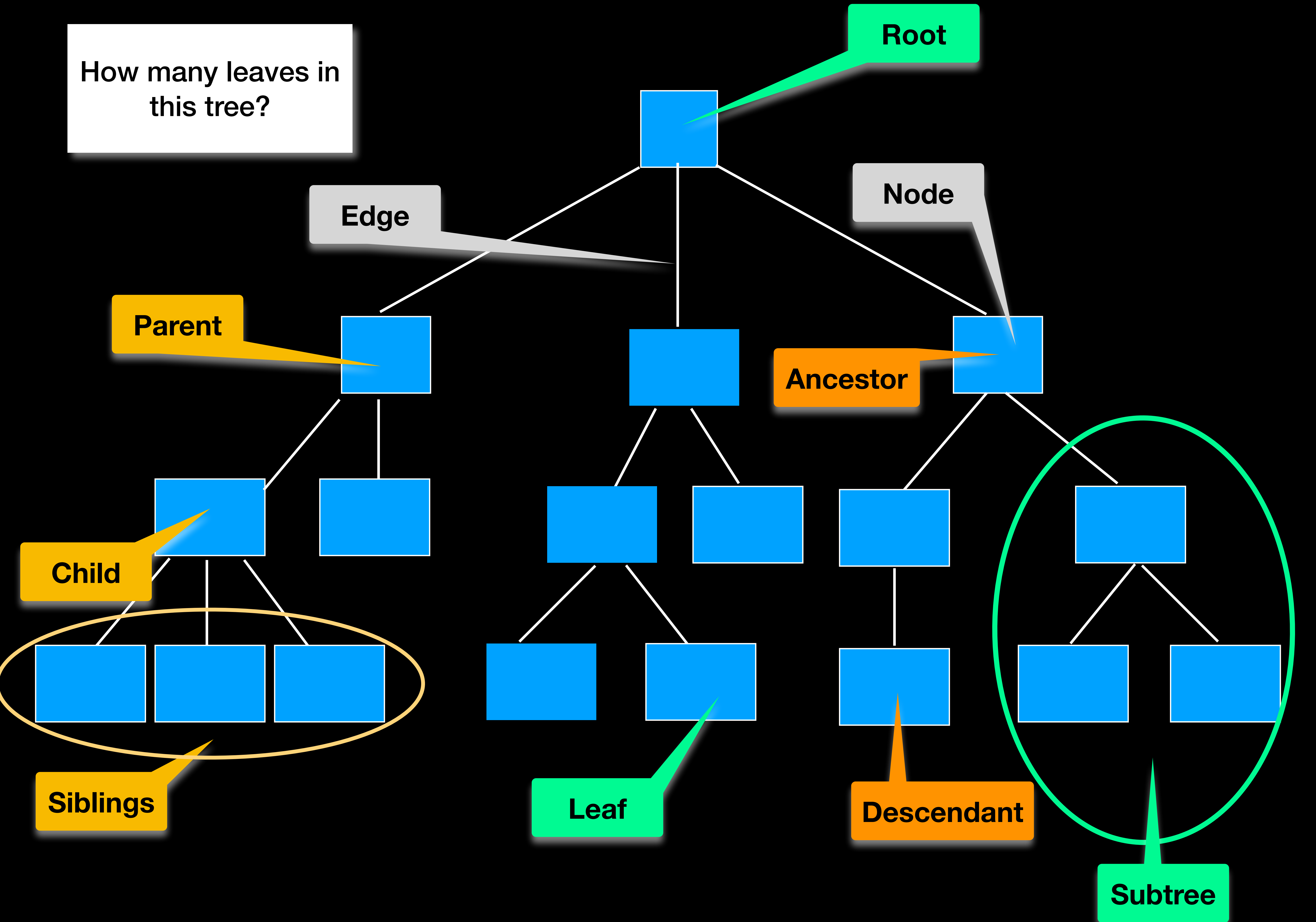



Edge
Node


Edge
Node
Parent
Ancestor
Child
Siblings
Descendant


Root
Edge
Node
Parent
Ancestor
Child
Siblings
Leaf
Descendant
Subtree

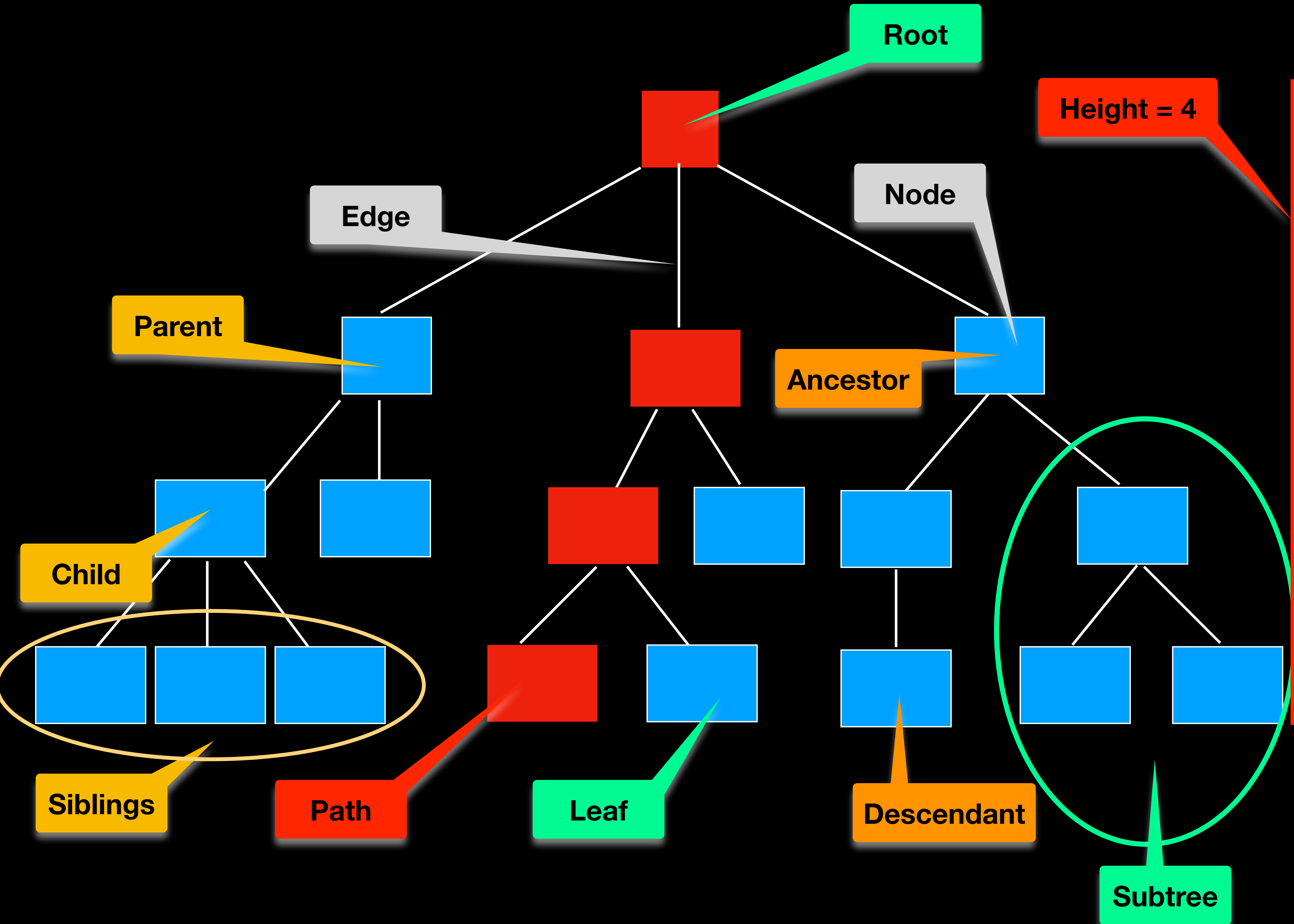
How many leaves in this tree?



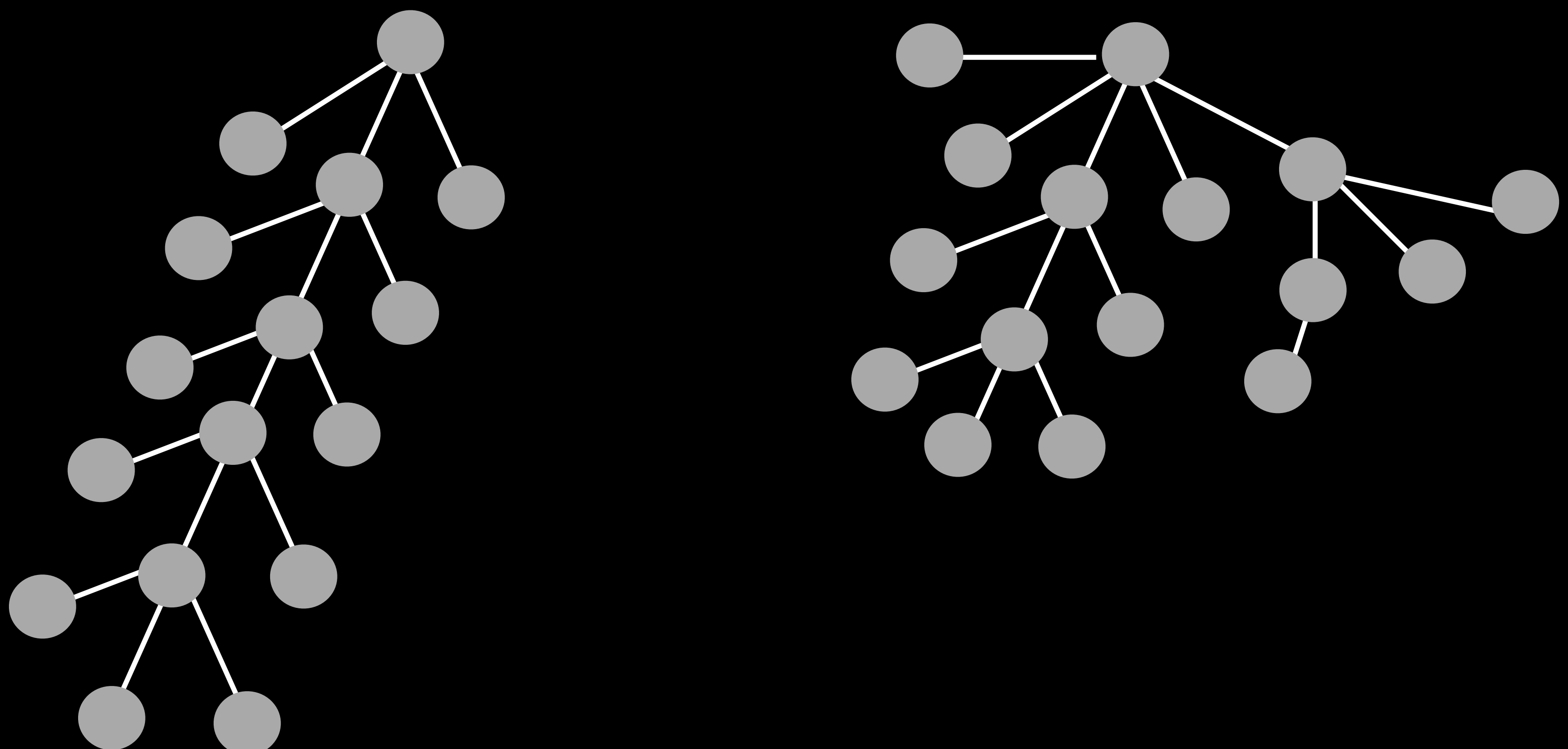
Subtree: the subtree rooted at node *n* is the tree formed by taking *n* as the root node and including all its descendants.

Path: a sequence of nodes c_1, c_2, ..., c_k where c_{i+1} is a child of c_i.

Height: the **number of nodes** in the *longest* path from the root to a leaf.


Root
Height = 4
Edge
Node
Parent
Ancestor
Child
Siblings
Path
Leaf
Descendant
Subtree

Different shapes/structures



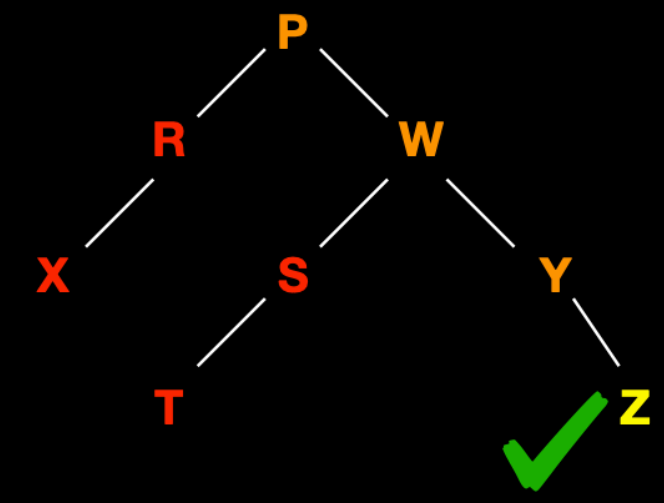
Both n = 16
Both 11 leaves
Different height

We have already seen Trees!

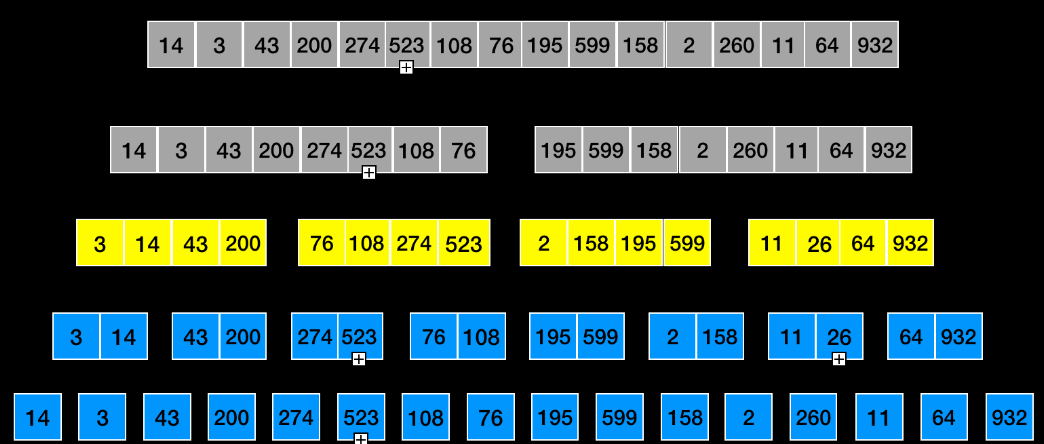
Mostly as a "thinking tool"

- Decision Trees

- Divide and Conquer



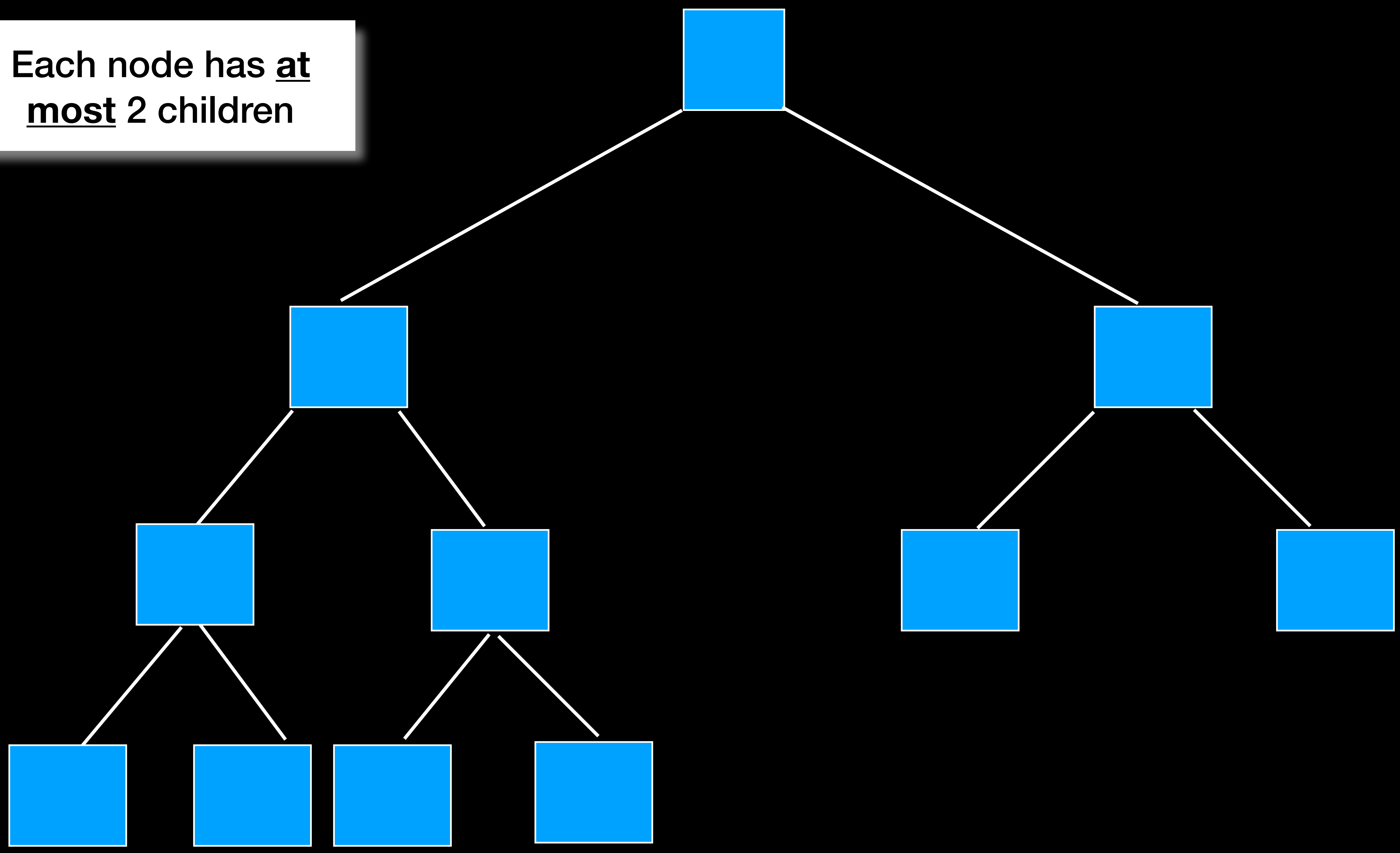
Merge Sort



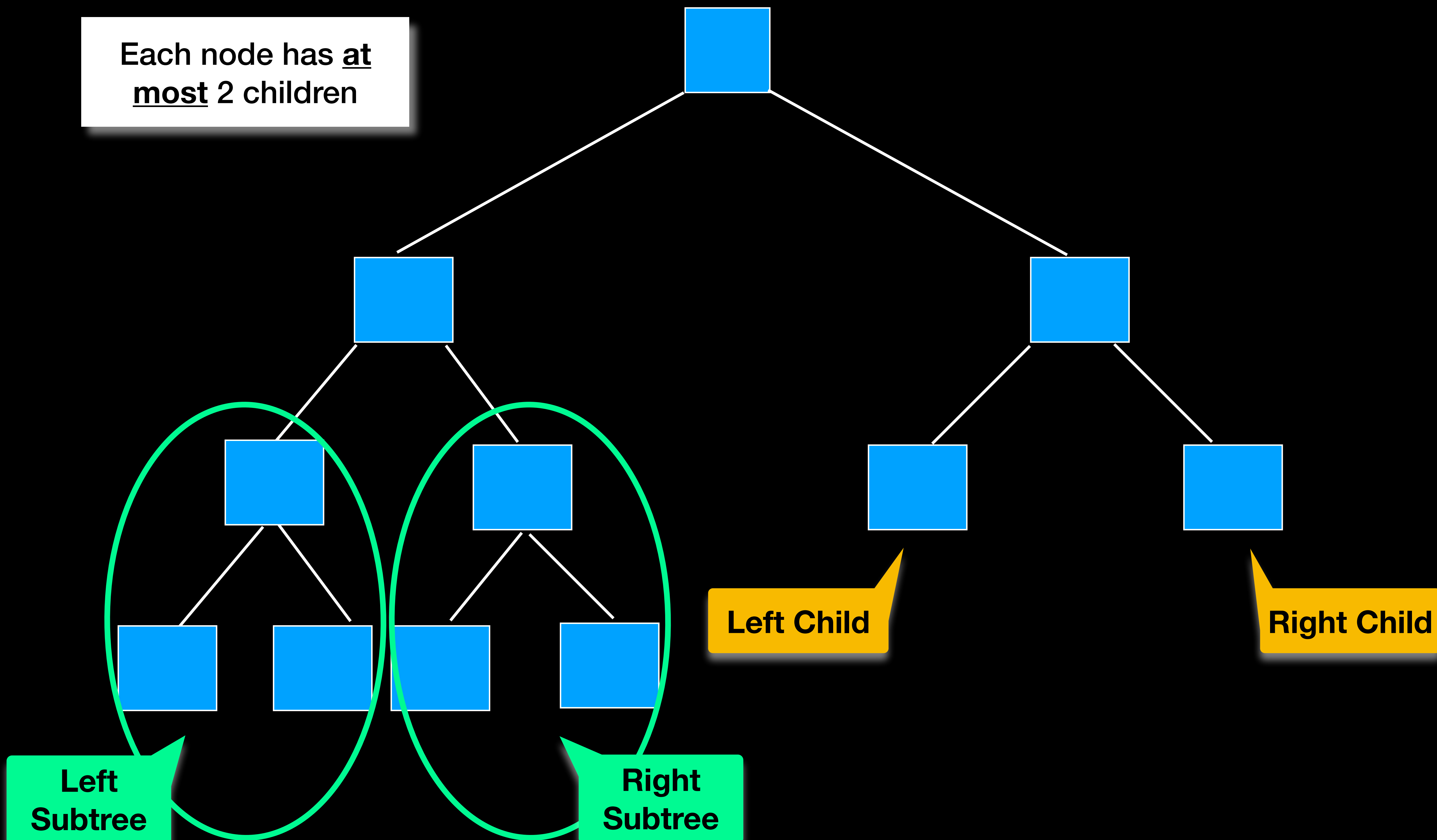
Binary Tree ADT

Binary Tree

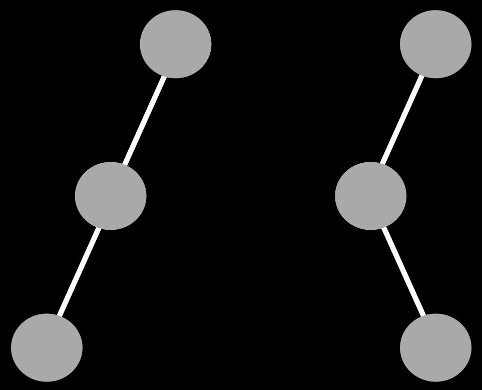
Each node has **__at most__** 2 children



Binary Tree


Each node has **at most** 2 children
Left Child
Right Child
Left Subtree
Right Subtree

Different shapes/structures

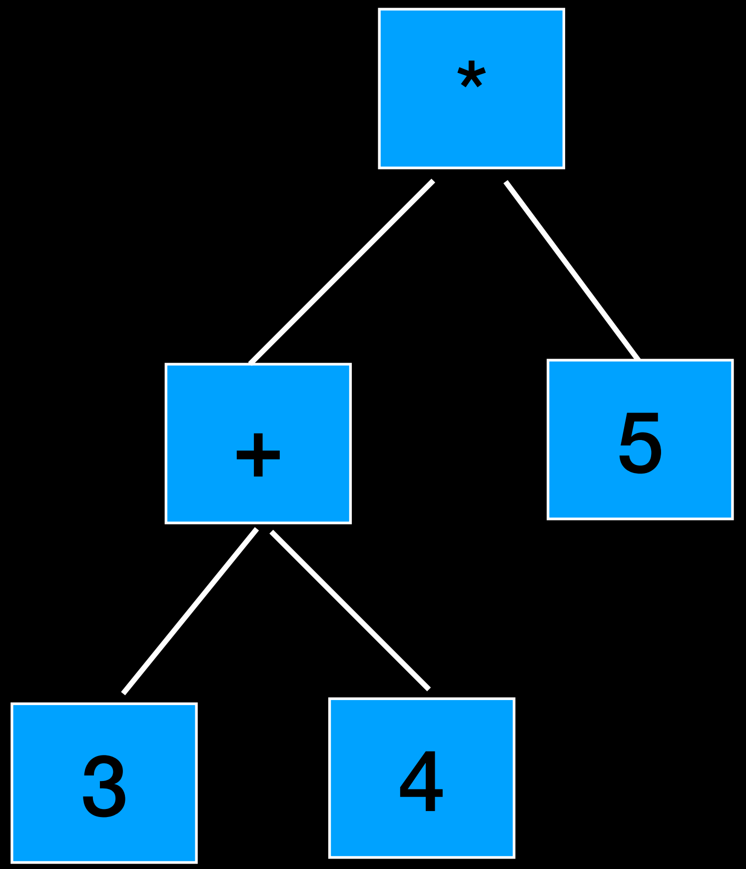


Both h = 3 and one leaf
But different

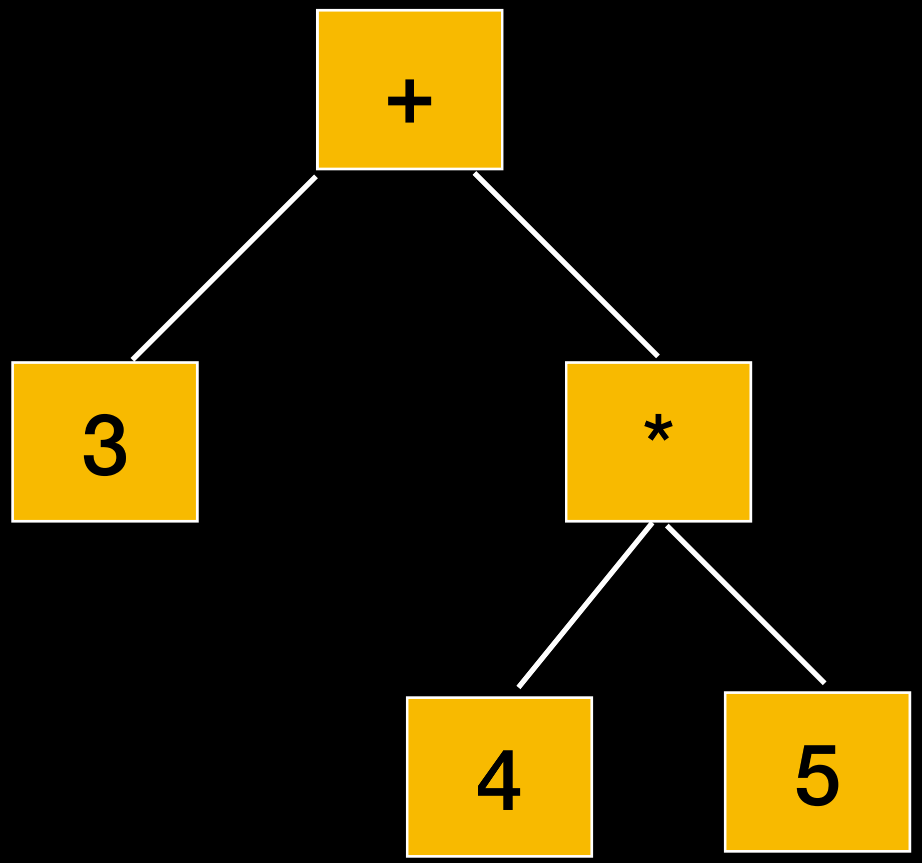
Binary Tree Applications

Algebraic Expressions

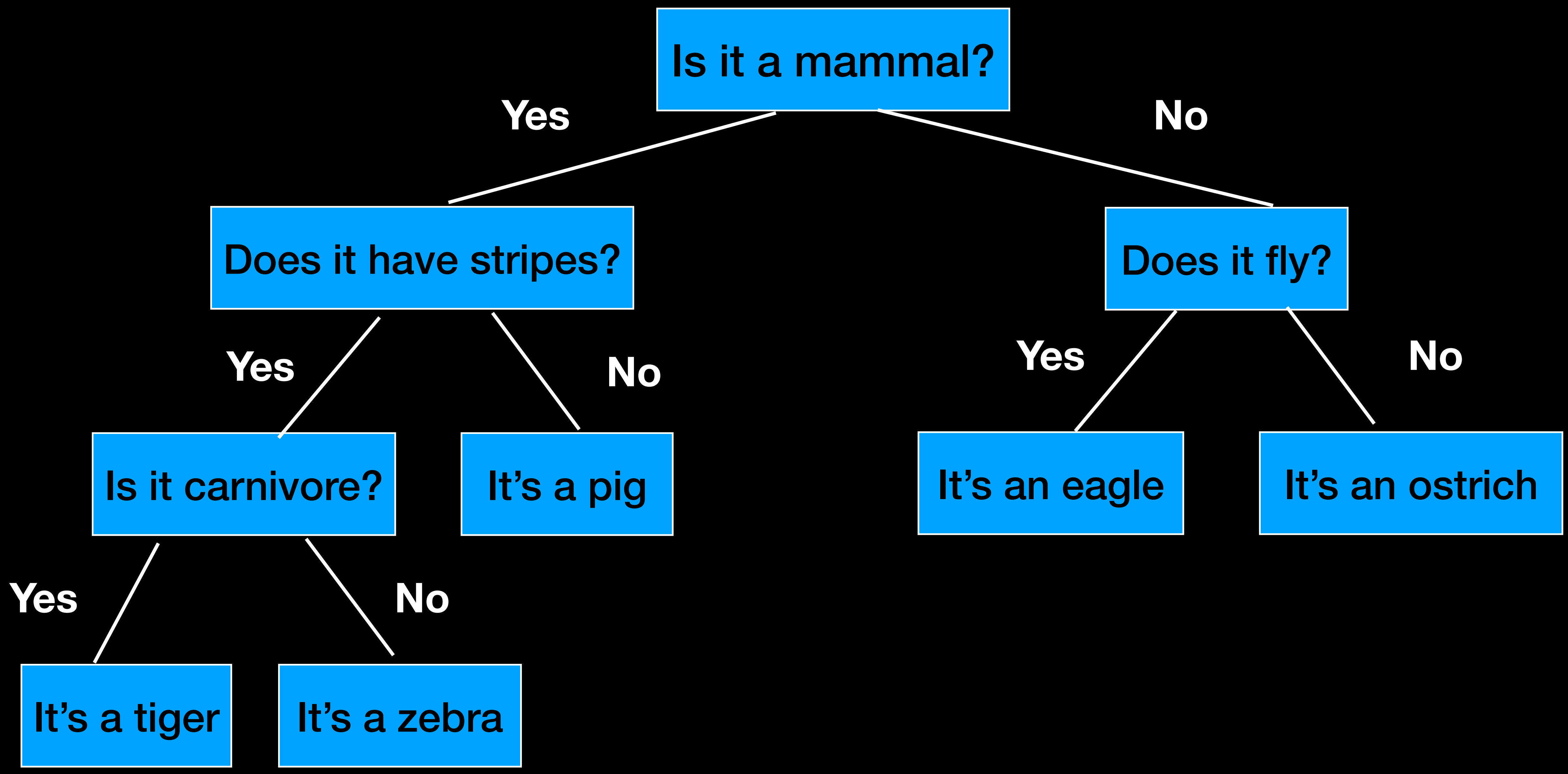
(3 + 4) * 5



3 + 4 * 5



Decision Tree



Huffman Tree

Huffman Encoding Compression Algorithm (**Huffman Encoding**):
"In 1951, David A. Huffman for his MIT Information Theory class term paper hit upon the idea of using a **frequency-sorted binary tree** and quickly proved this method the most efficient."

IDEA: Encode symbols into a sequence of bits s.t. most frequent symbols have shortest encoding

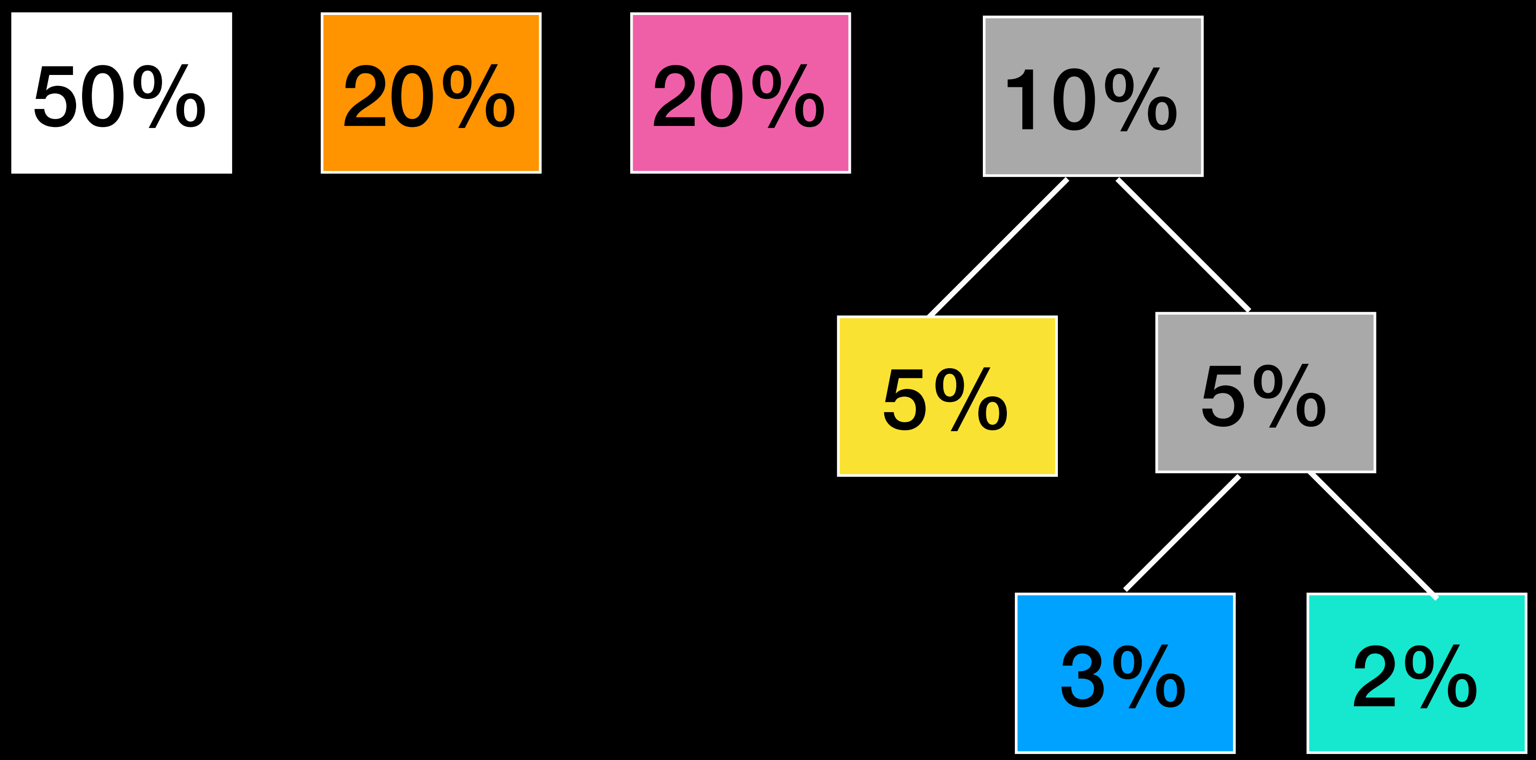
Not encryption but compression => use shortest code for most frequent symbols

No codeword is prefix to another codeword (i.e. if a symbol is encoded as 00 no other codeword can start with 00)

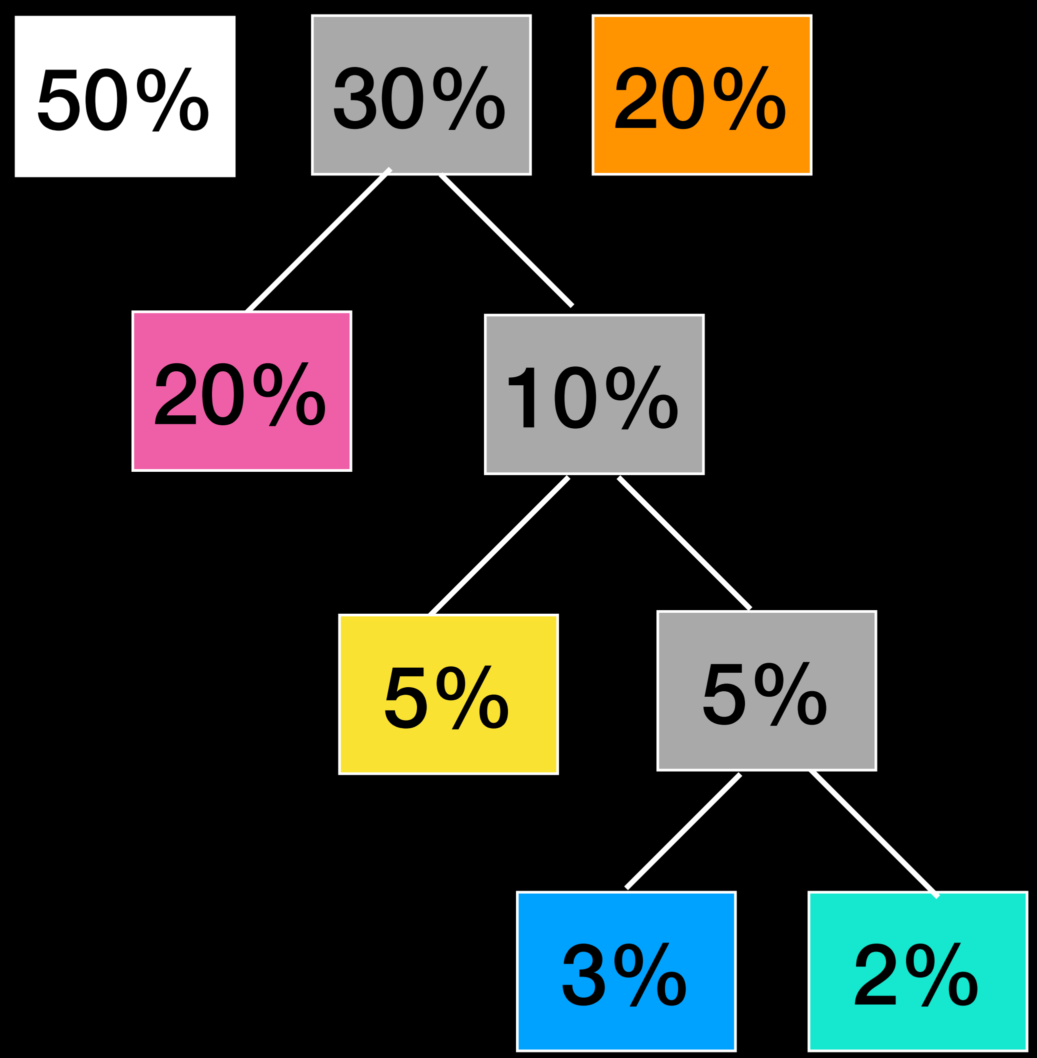
Huffman Tree

Huffman Tree

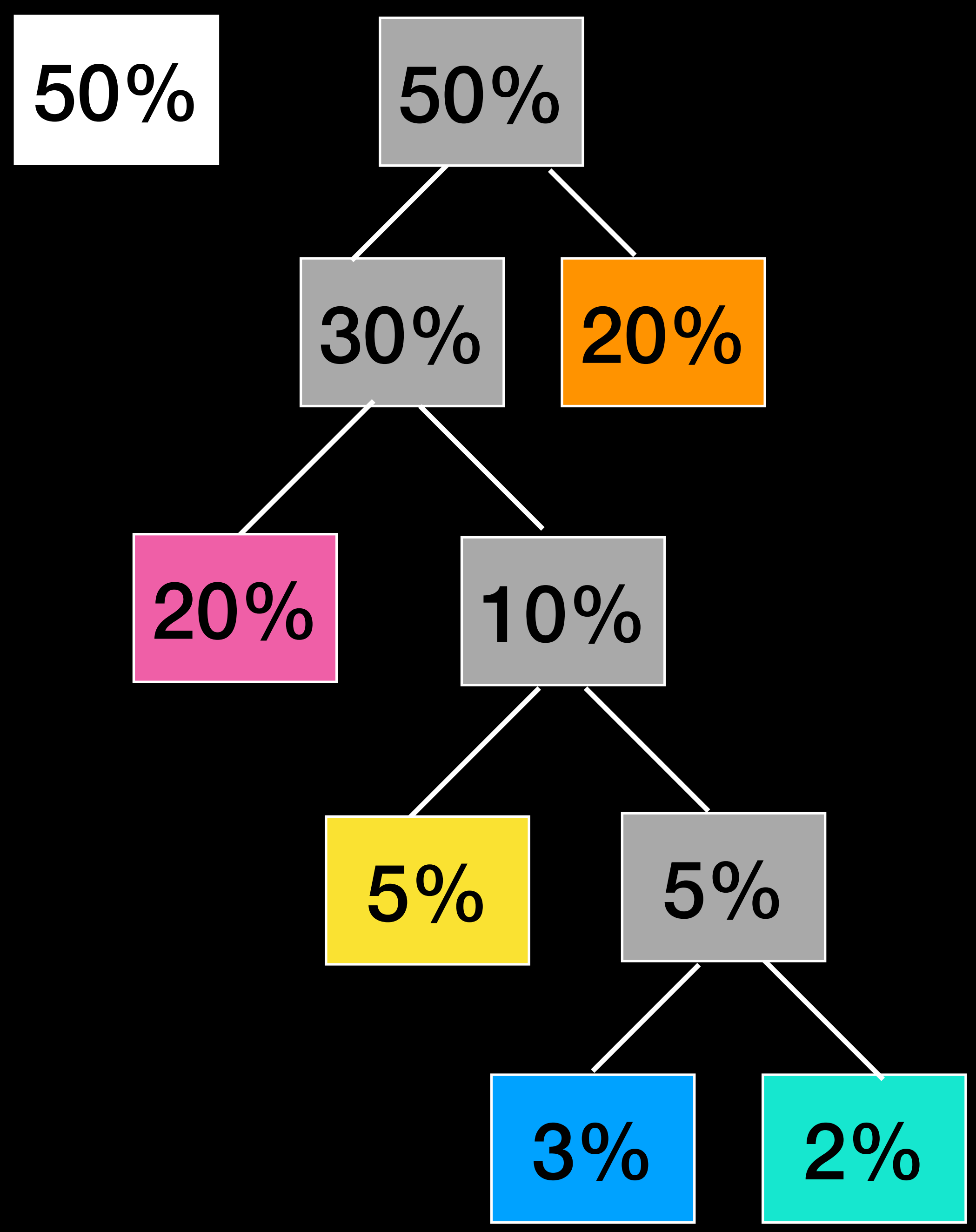
Huffman Tree



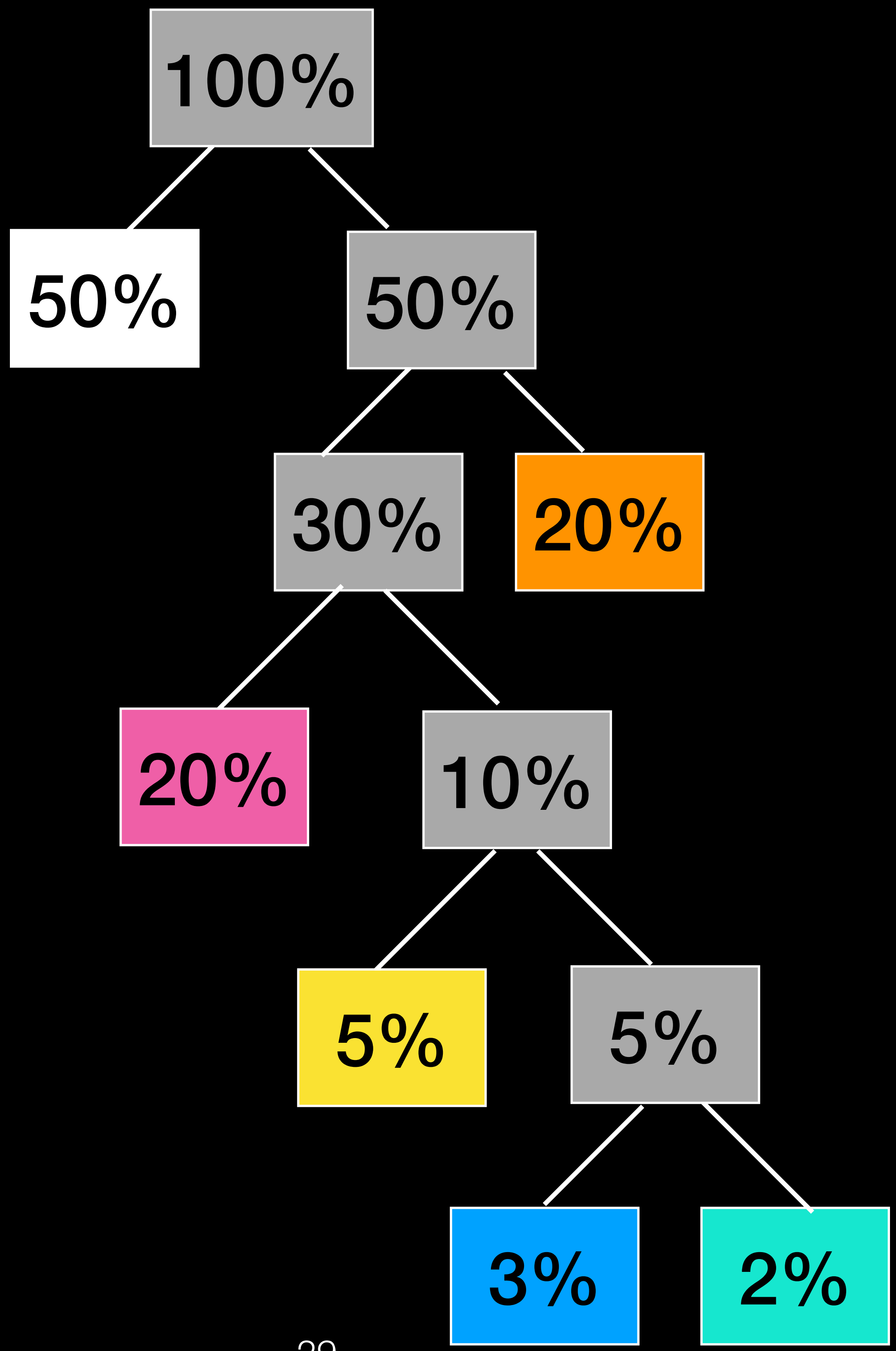
Huffman Tree



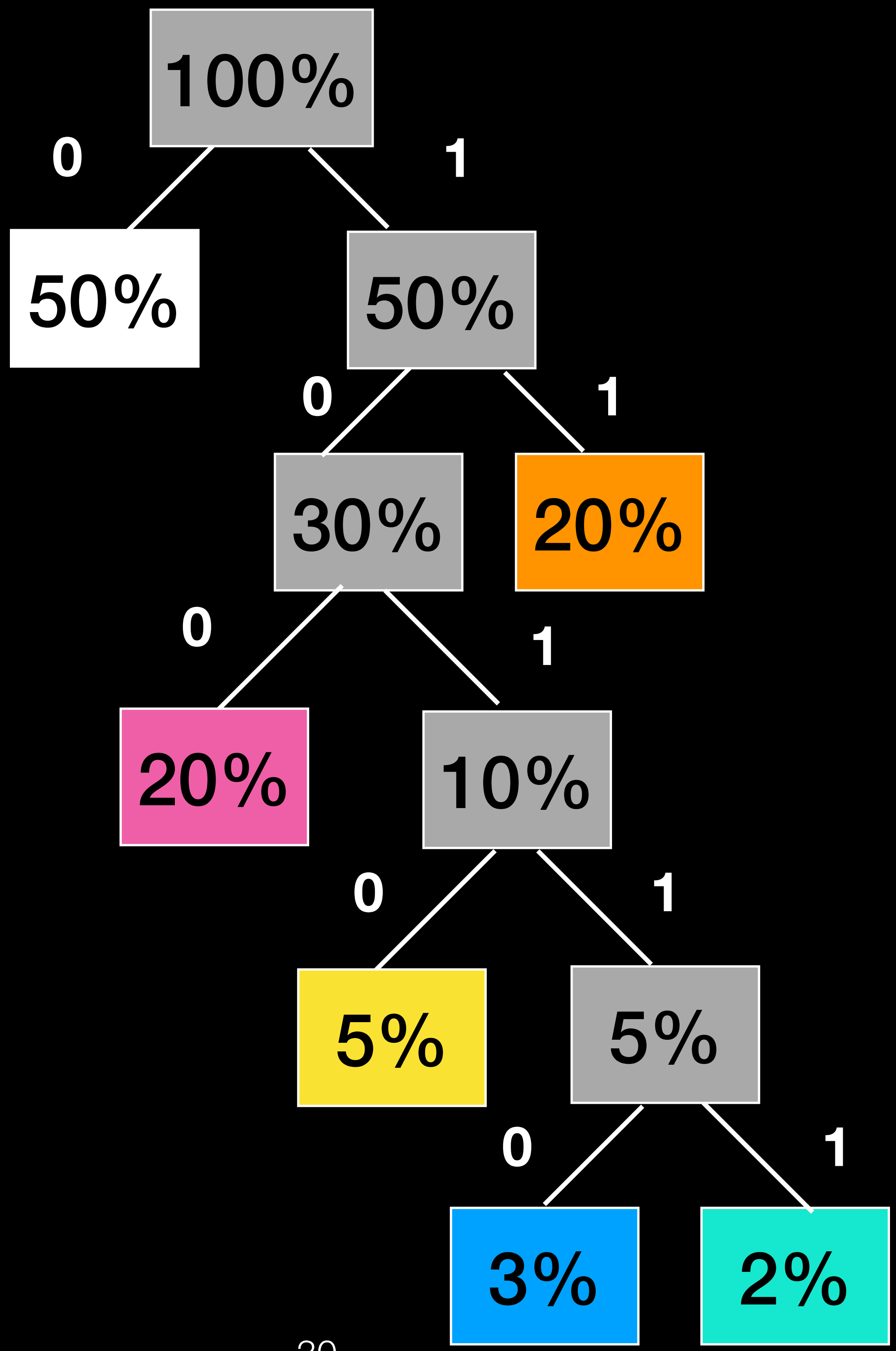
Huffman Tree



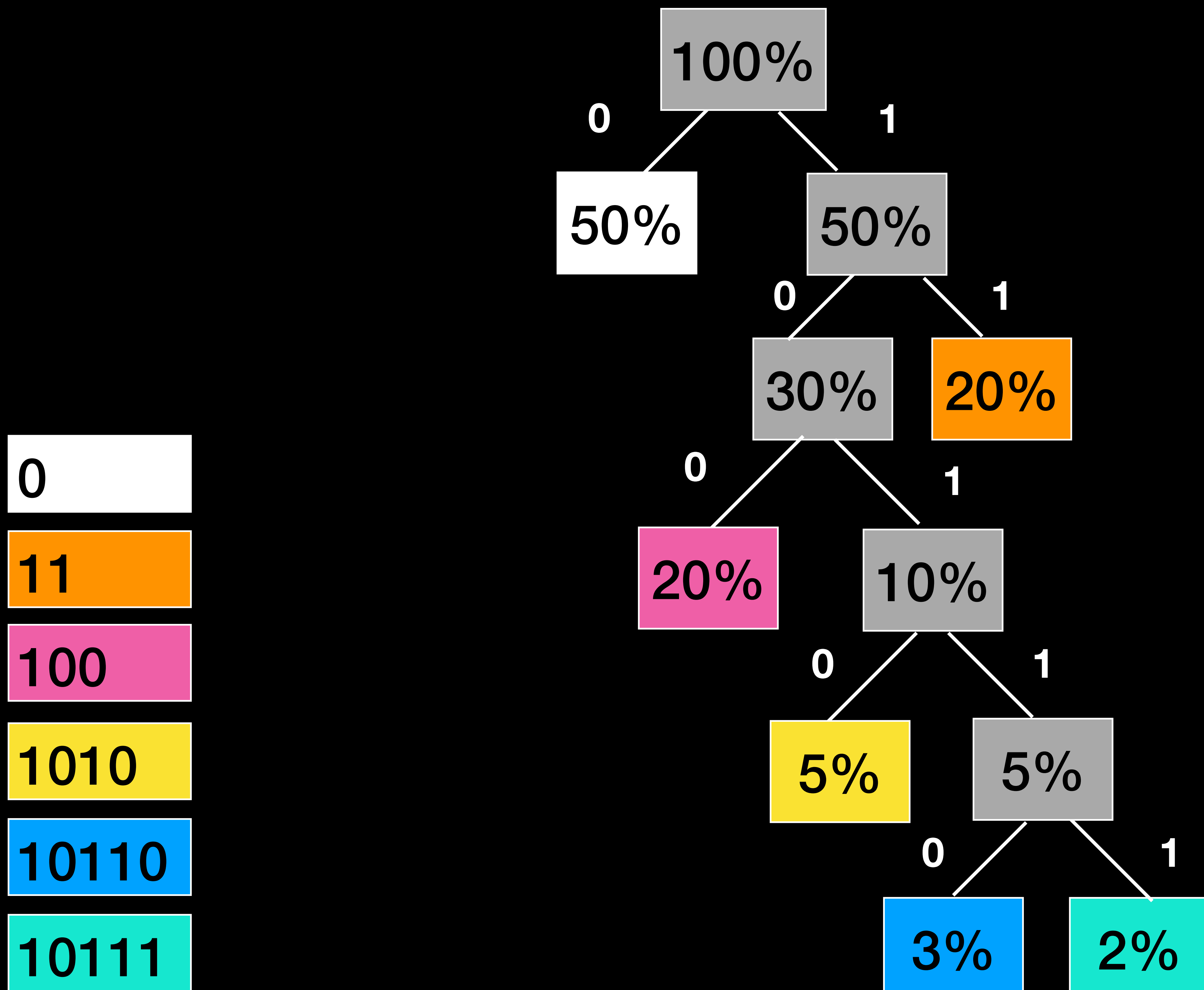
Huffman Tree



Huffman Tree



Huffman Tree



Lecture Activity

Think about structure!

Draw **ALL POSSIBLE** binary trees with 4 nodes

Label each tree with its height and number of leaves.

Lecture Activity

Think about structure!

Draw **ALL POSSIBLE** binary trees with 4 nodes

Label each tree with its height and number of leaves.

How many did you draw?

What's the maximum/minimum height?

What's the maximum/minimum number of leaves?

Lecture Activity

Think about structure!

Draw **ALL POSSIBLE** binary trees with 4 nodes

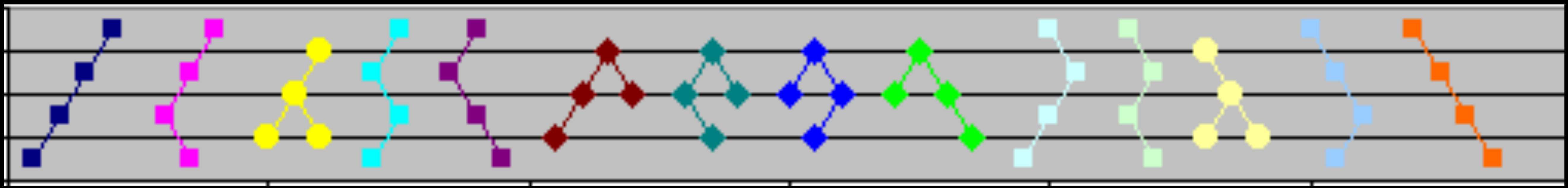
Label each tree with its height and number of leaves.

How many did you draw? **14**

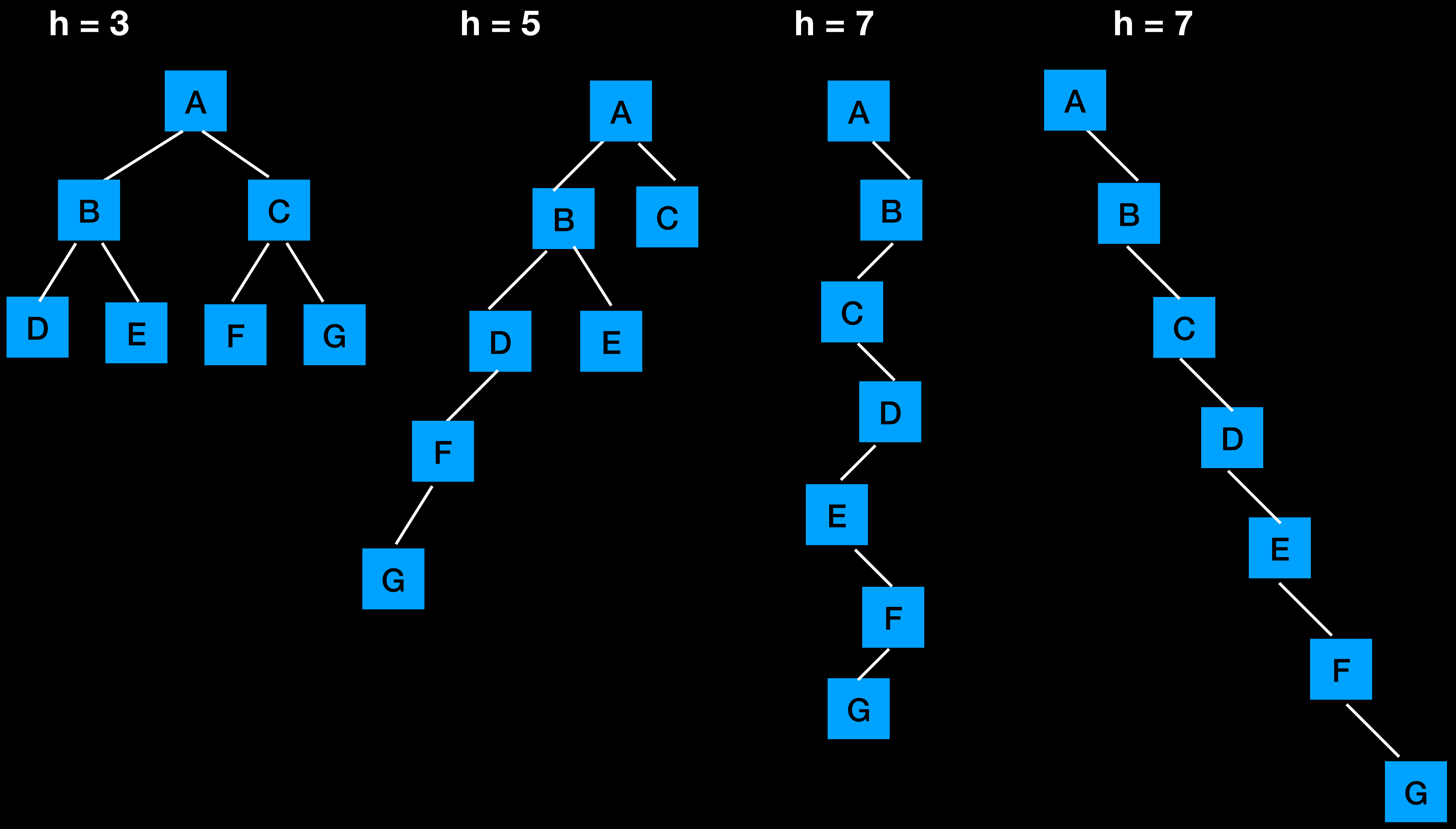
What's the maximum/minimum height? **max = 4, min = 3**

What's the maximum/minimum number of leaves?

max = 2, min = 1



Tree Structure

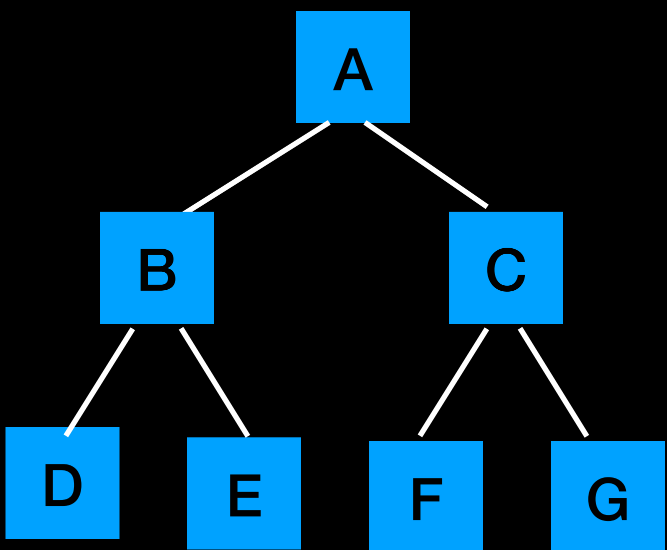


Structure definitions may vary across different sources.

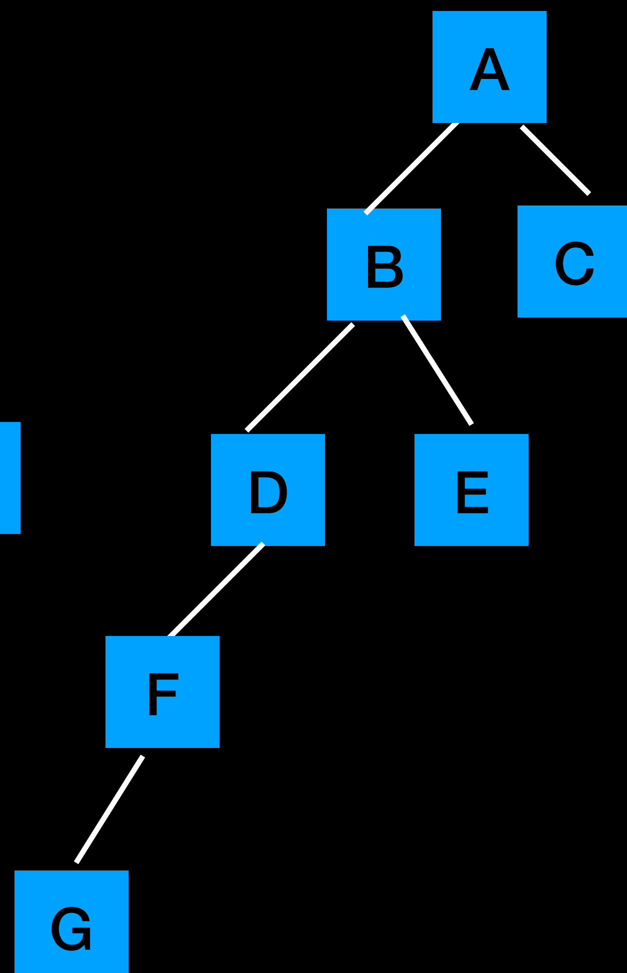
The following comes from your textbook and will be used in this course and on exams

Tree Structure

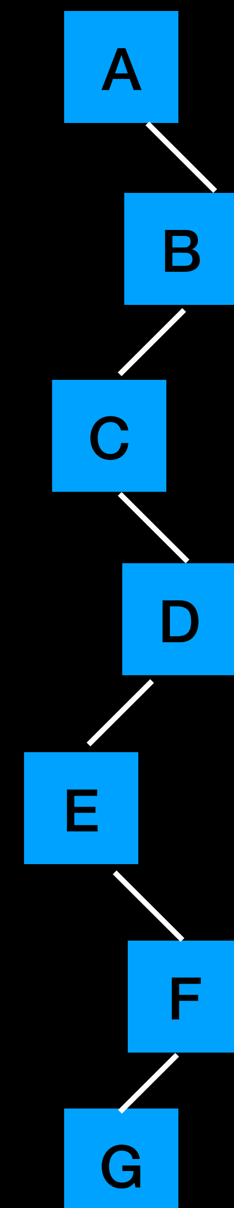
h = 3



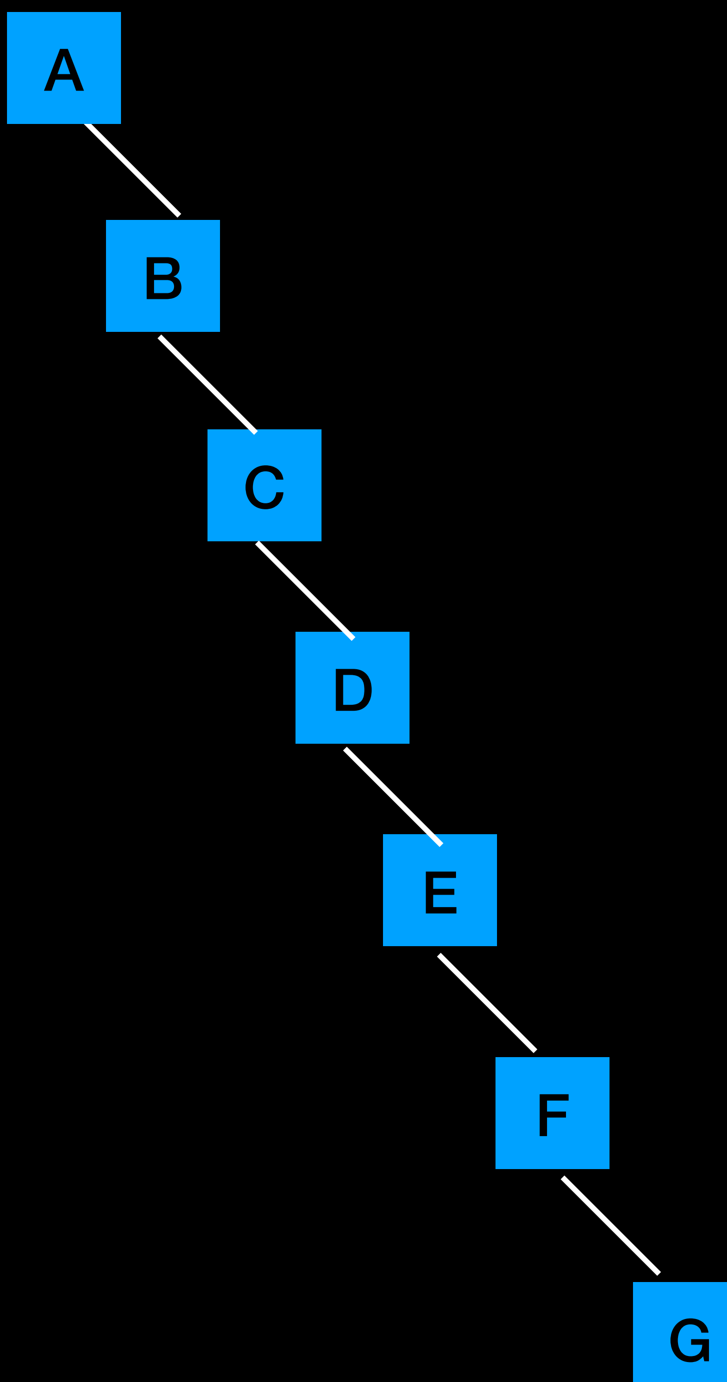
h = 5



h = 7



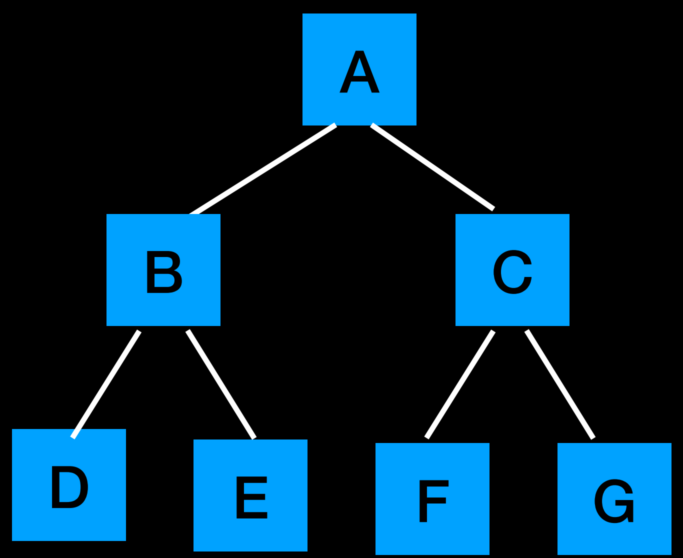
h = 7



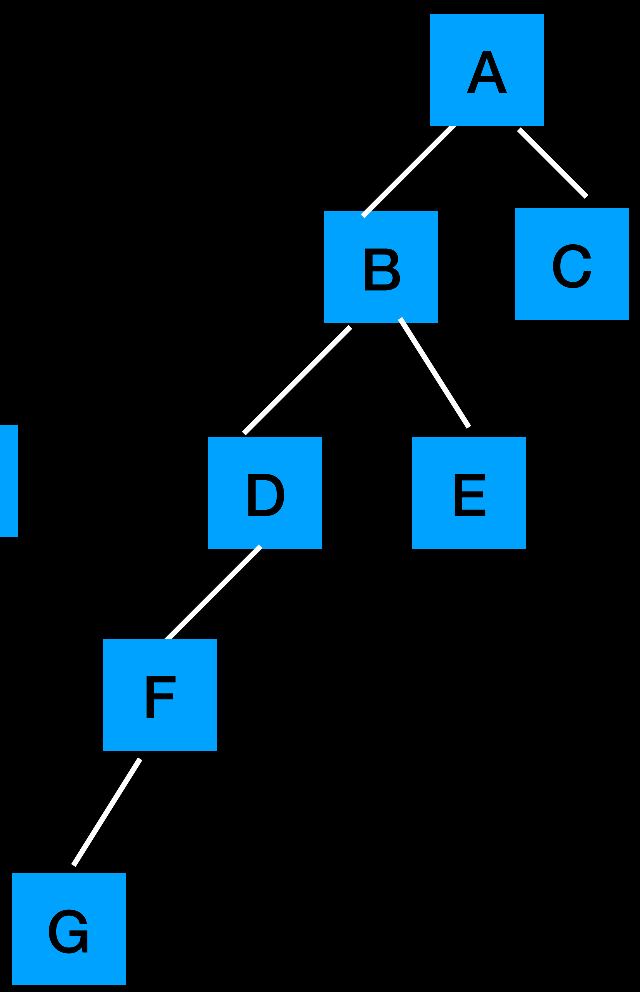
What is the maximum (minimum) height of a tree with 7 nodes?

Tree Structure

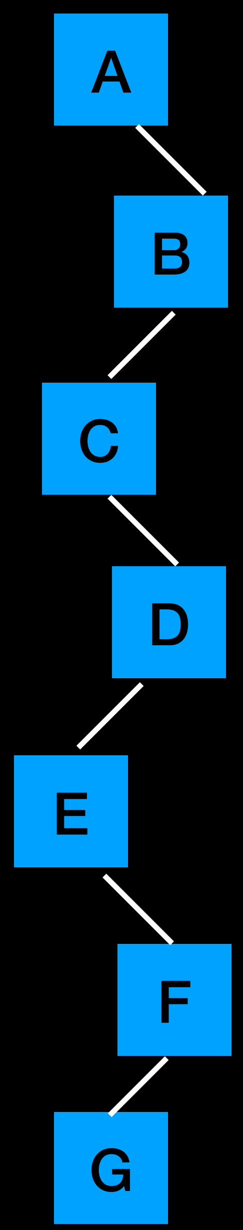
h = 3



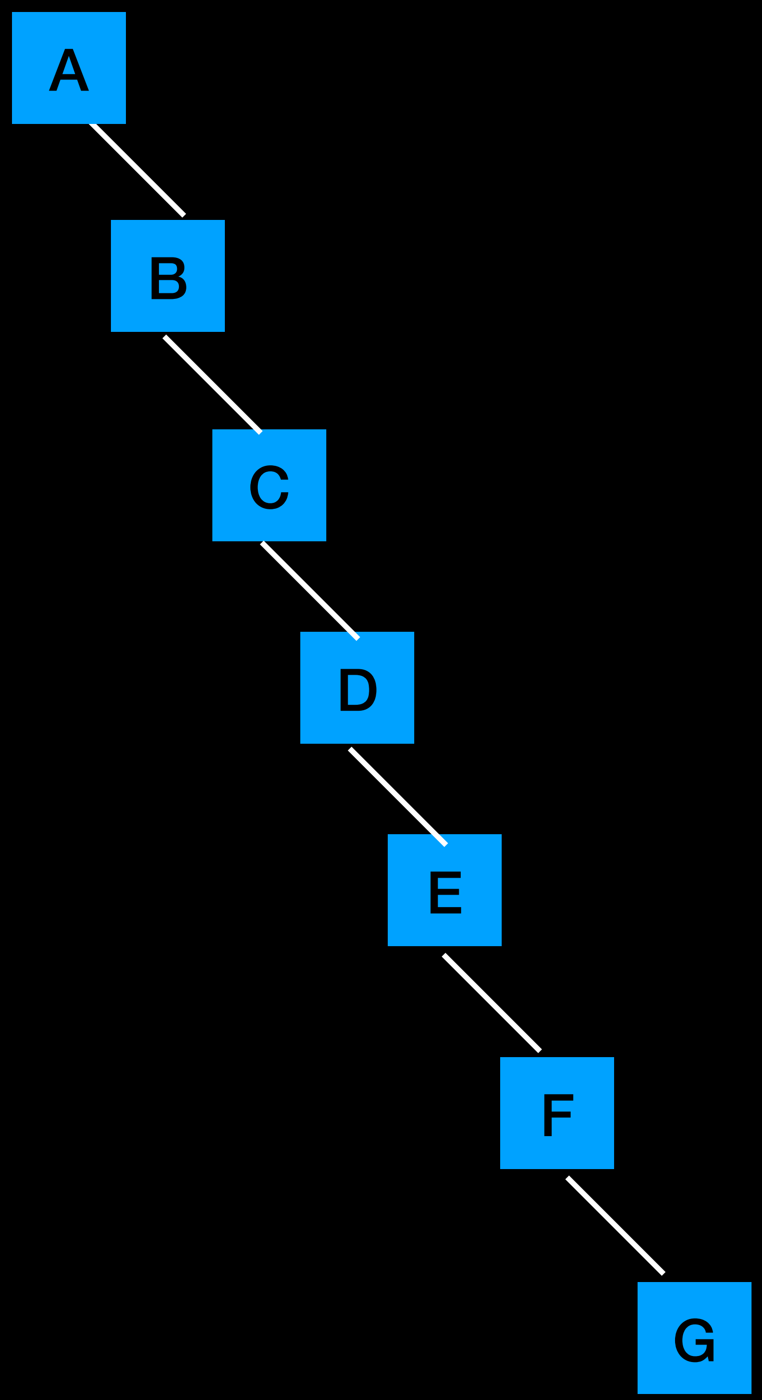
h = 5



h = 7



h = 7



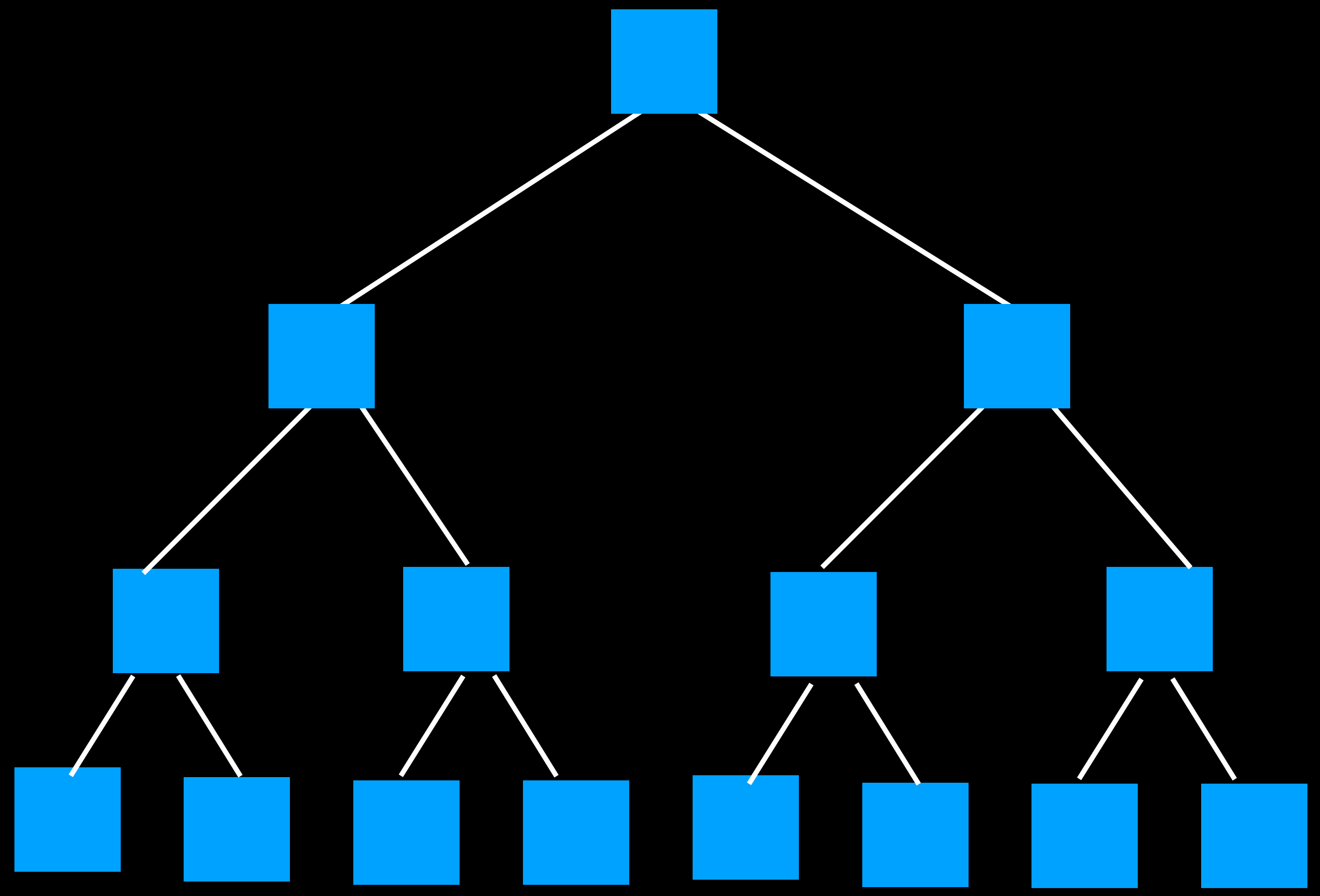
WE WILL LOOK AT THE GENERAL ANSWER NEXT

Full Binary Tree

Every node that is not a leaf has exactly 2 children

Every node has left and right subtrees of same height

All leaves are at same level *h*



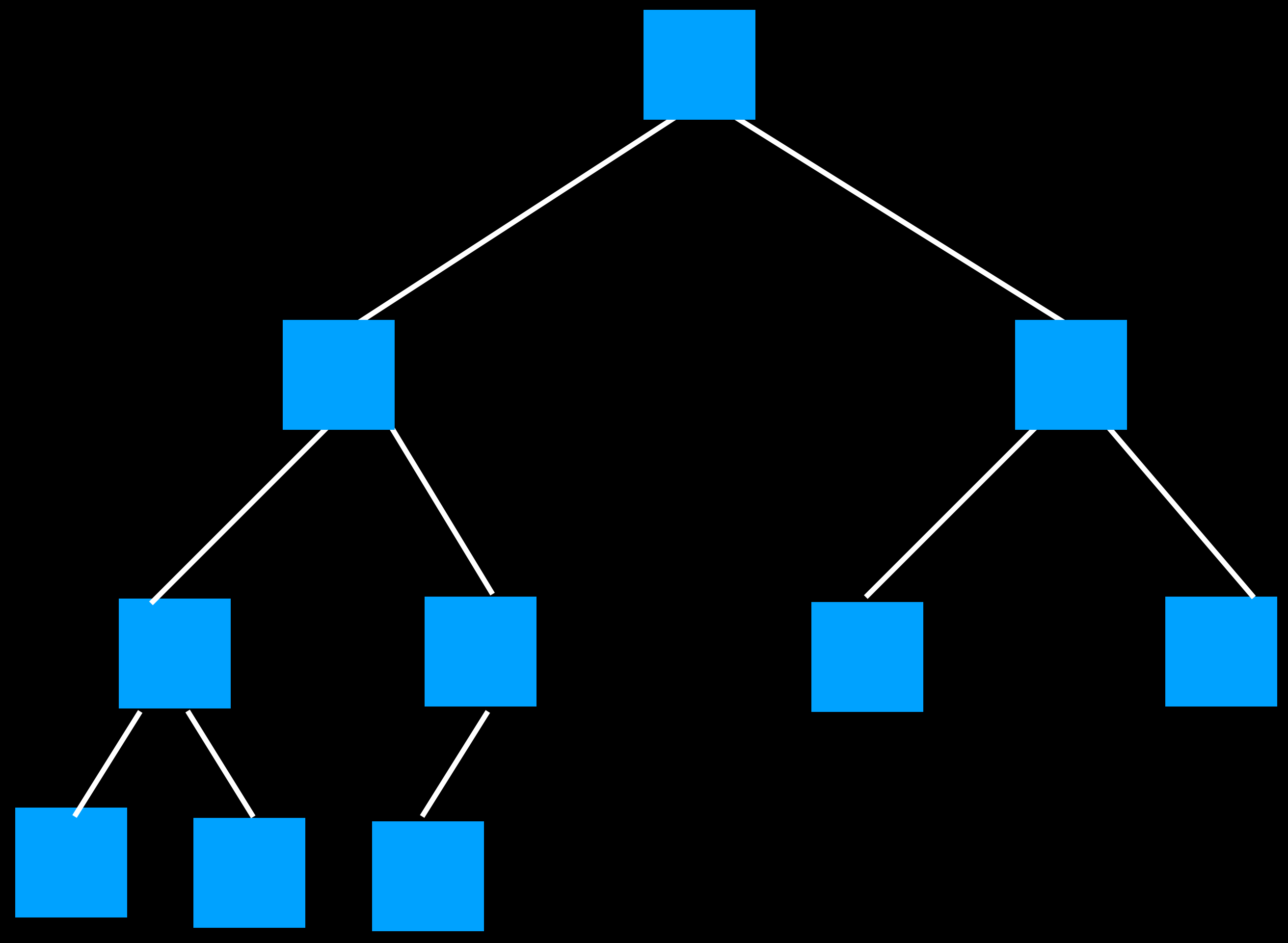
Complete Binary Tree

A tree that is full up to level *h-1*, with level *h* filled in from left to right

All nodes at levels *h-2* and above have exactly 2 children

When a node at level *h-1* has children, all nodes to its left have exactly 2 children

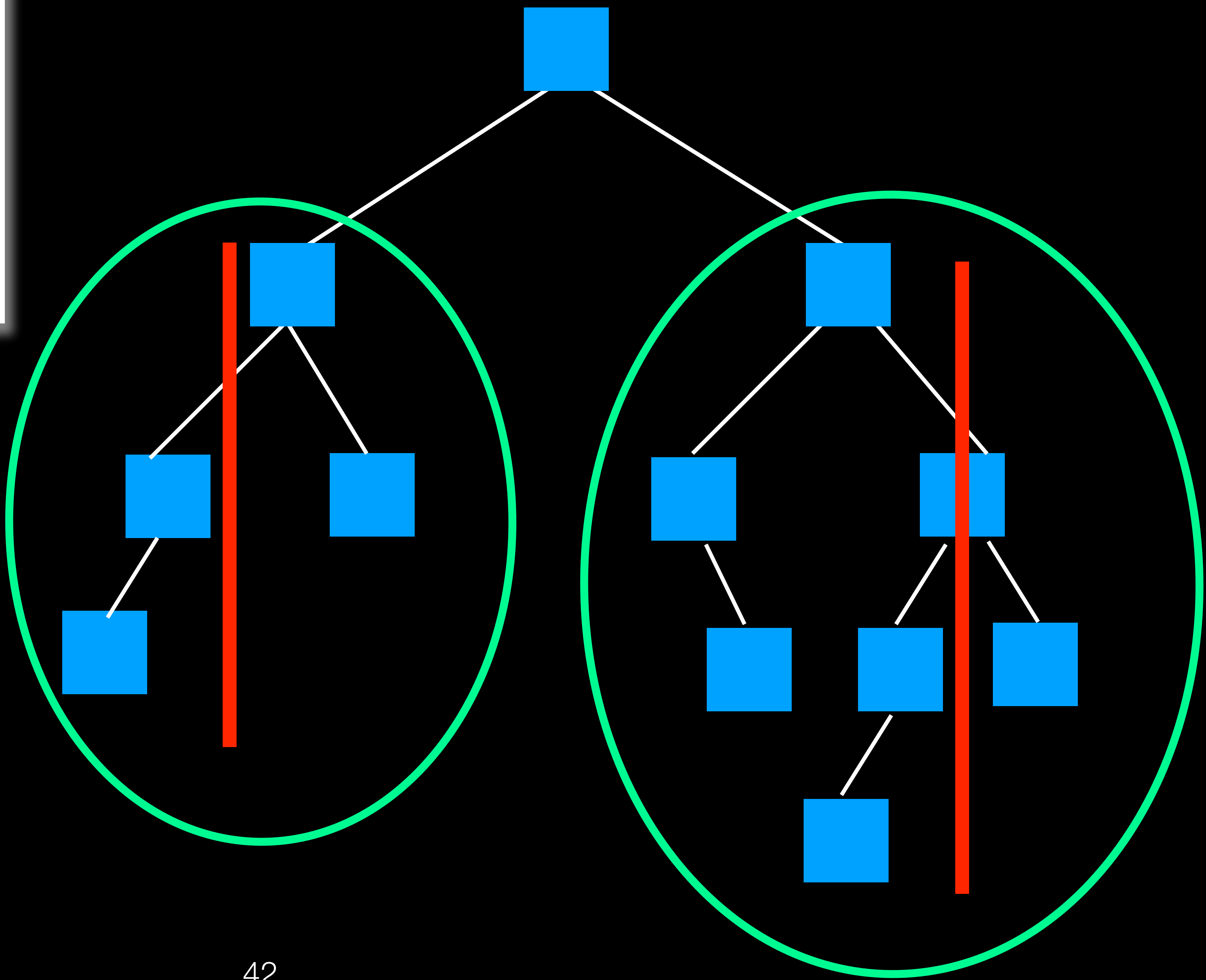
When a node at level *h-1* has one child, it is a left child



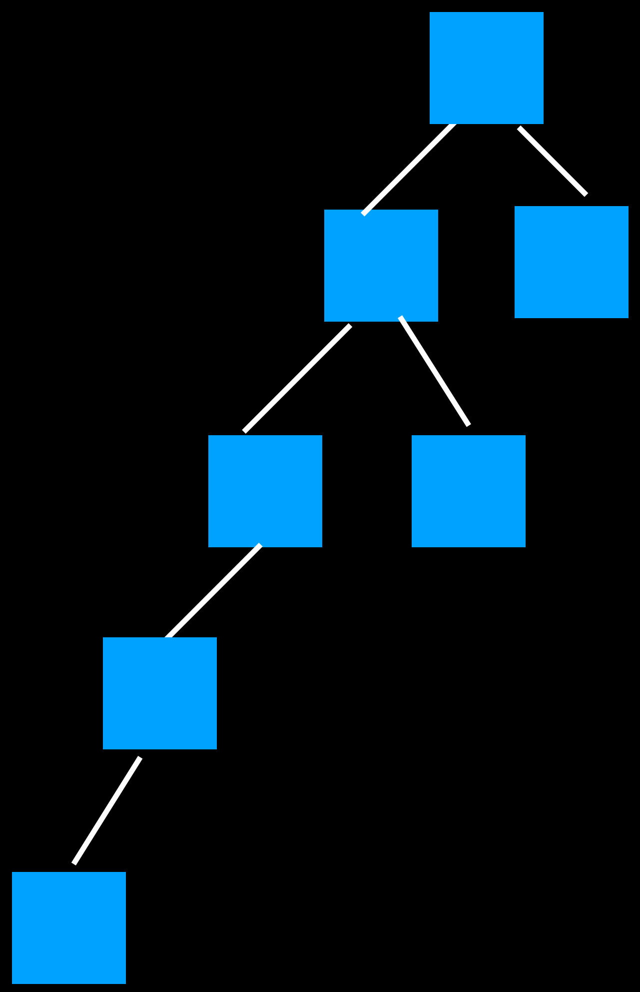
(Height) Balanced Binary Tree

For any node, its left and right subtrees differ in **height** by no more than 1

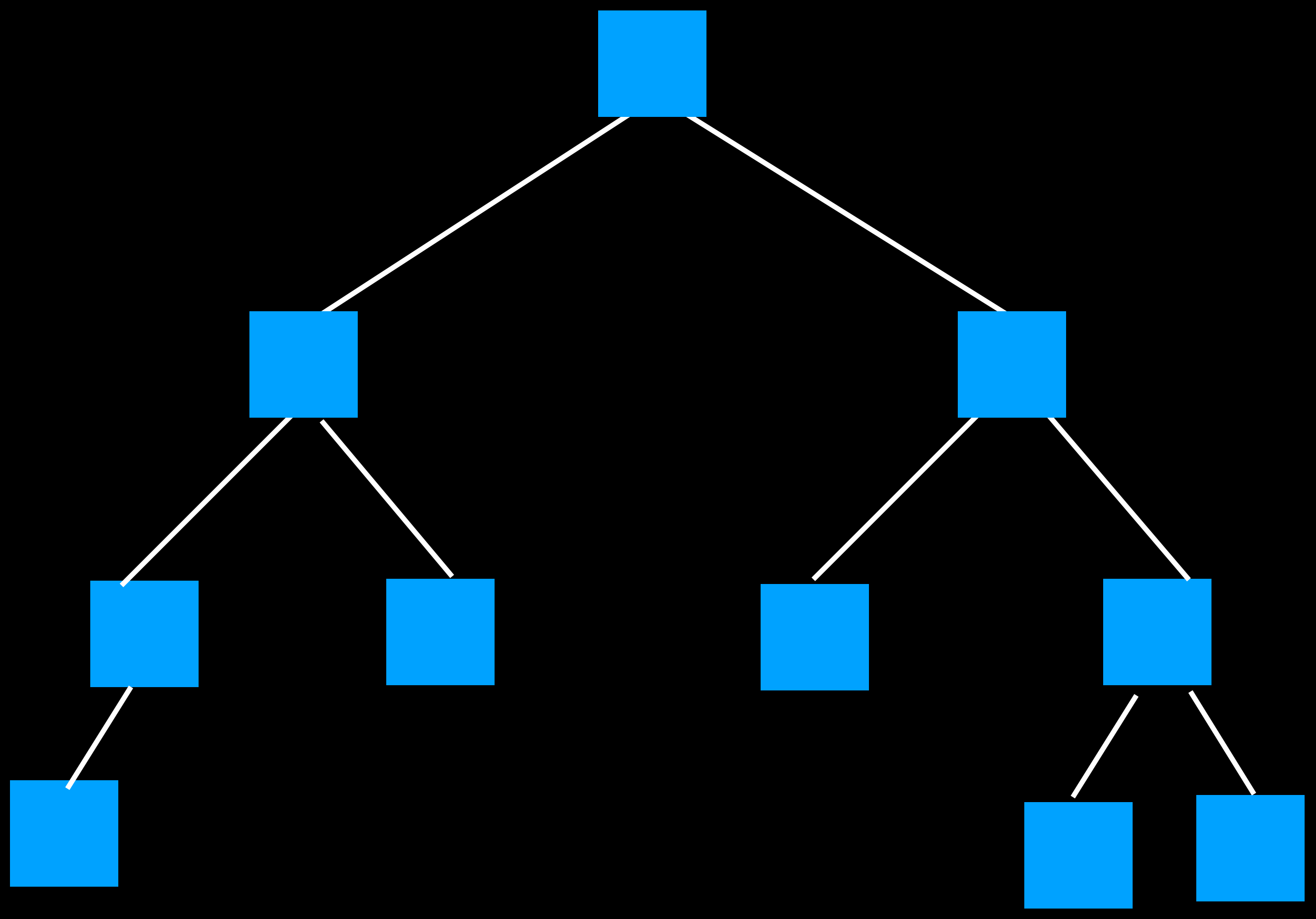
All paths from root of subtrees to leaf differ in length by at most 1



Unbalanced



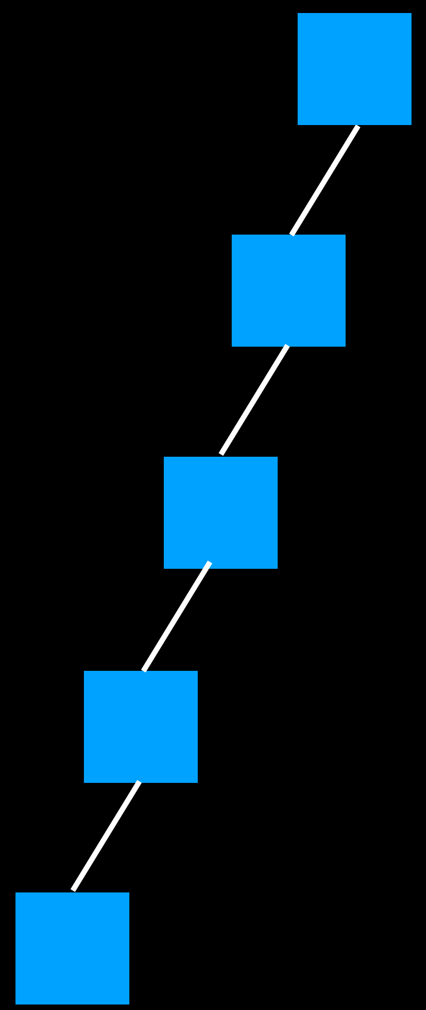
Balanced



Maximum Height

n nodes
every node 1 child
h = n

Essentially a chain



Minimum Height

Binary tree of height h can have up to $n = 2^h - 1$

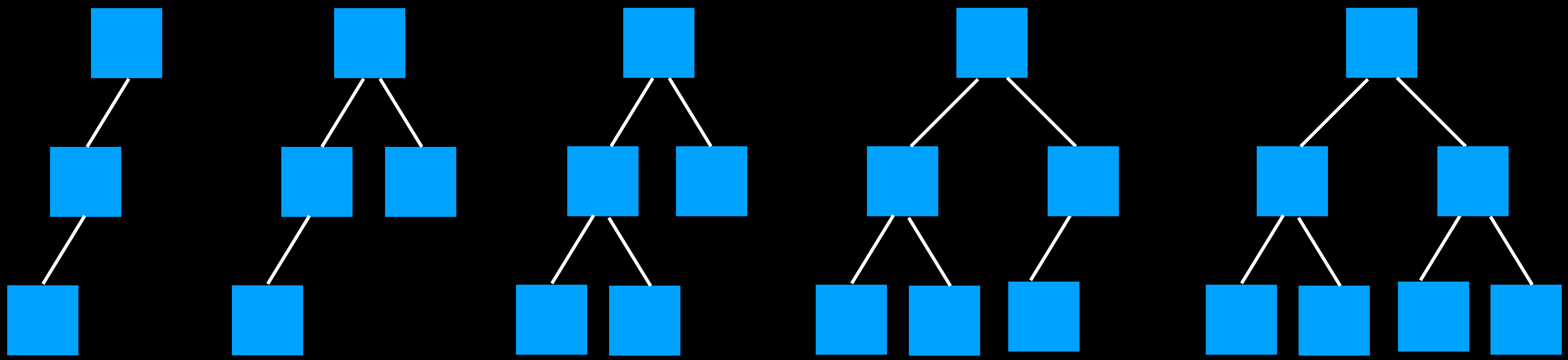
For example for $h = 3$, $1 + 2 + 4 = 7 = 2^3 - 1$

$h = \log_2(n+1)$ for a full binary tree

For example:

1,000 nodes h ≈ 10 (1,000 ≈ 2^{10})

1,000,000 nodes h ≈ 20 (10^6 ≈ 2^{20})



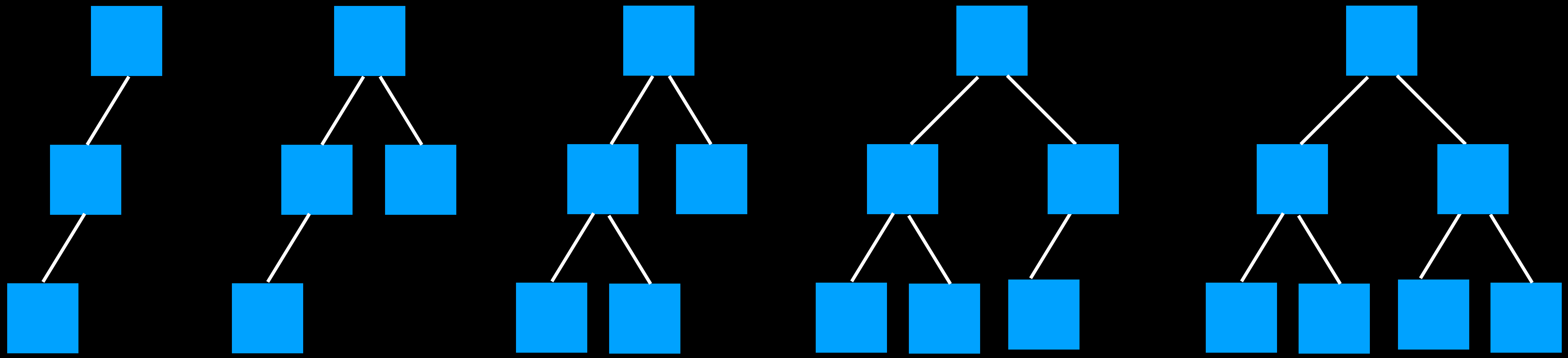
Minimum Height

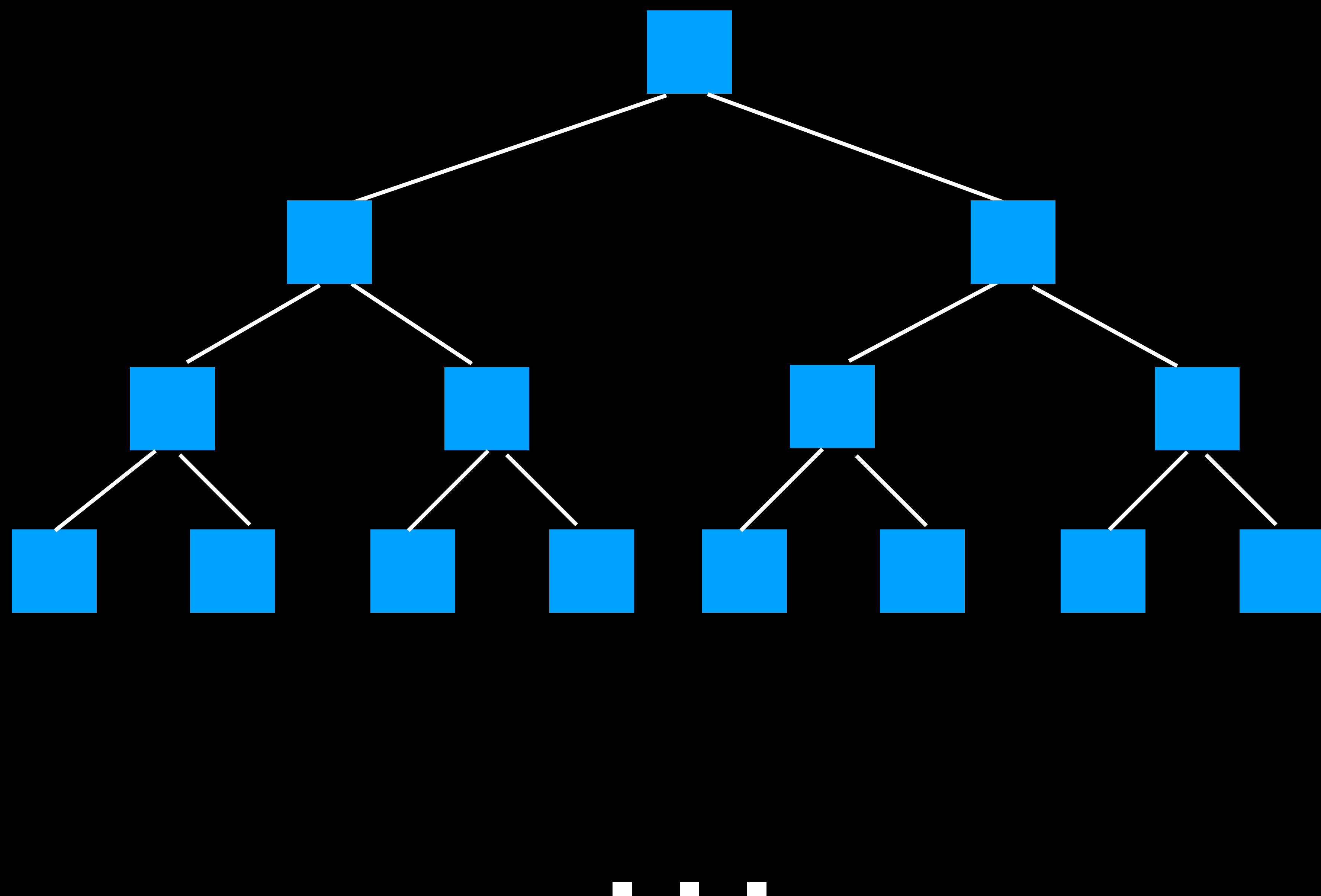
Binary tree of height h can have up to $n = 2^h - 1$
For example for $h = 3$, $1 + 2 + 4 = 7 = 2^3 - 1$
$h = \log_2(n+1)$ for a full binary tree

For example:
1,000 nodes h ≈ 10 (1,000 ≈ 2^{10})
1,000,000 nodes h ≈ 20 (10^6 ≈ 2^{20})

Recall analysis of Divide and Conquer algorithms

Important when we will be looking for things in trees given some order!!!





In a full tree:

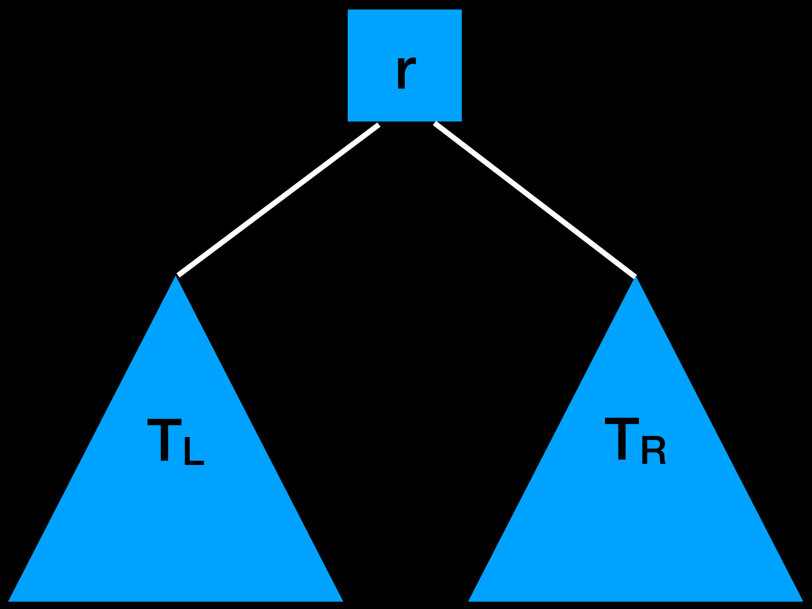
h	n @ level	Total n
1	$1 = 2^0$	$1 = 2^1 - 1$
2	$2 = 2^1$	$3 = 2^2 - 1$
3	$4 = 2^2$	$7 = 2^3 - 1$
4	$8 = 2^3$	$15 = 2^4 - 1$
h	2^{h-1}	$2^h - 1$

Binary Tree Traversals

Visit (retrieve, print, modify …) **every node** in the tree

Essentially visit the root as well as it's subtrees

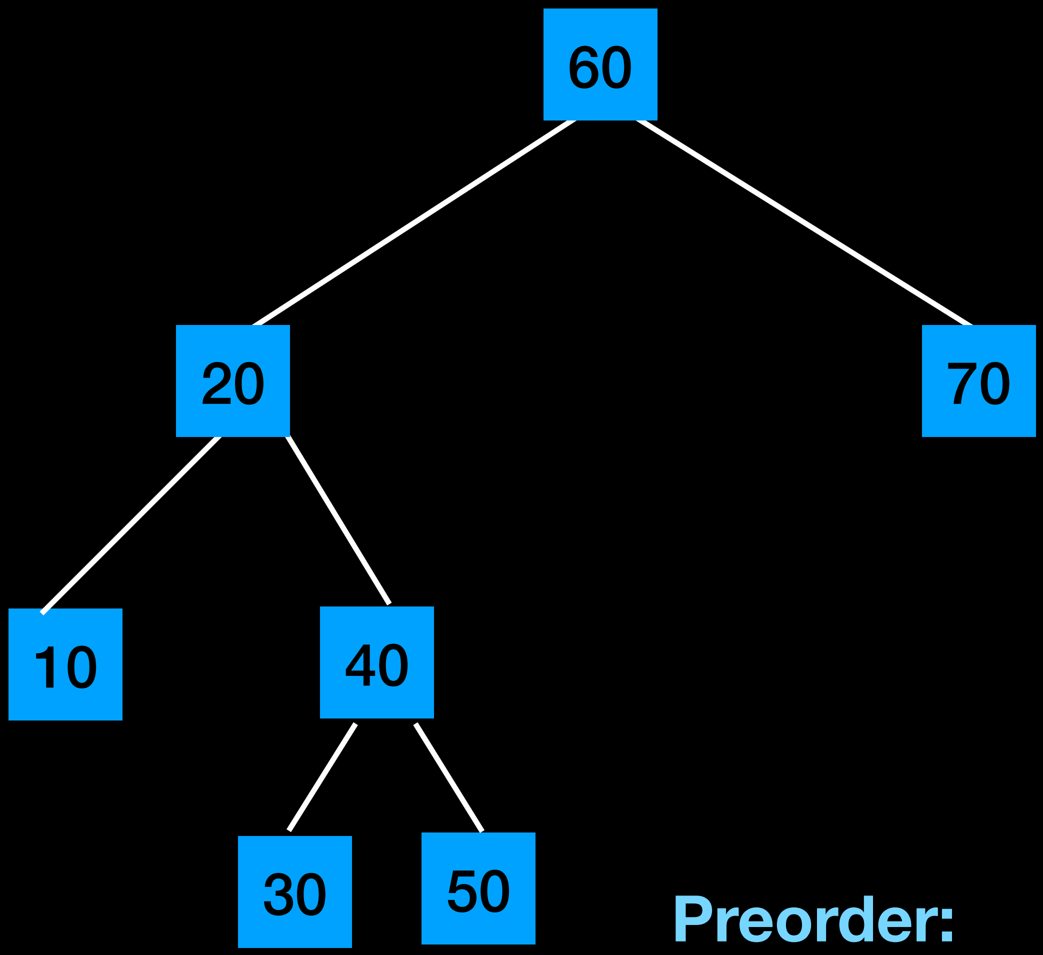
Order matters!!!

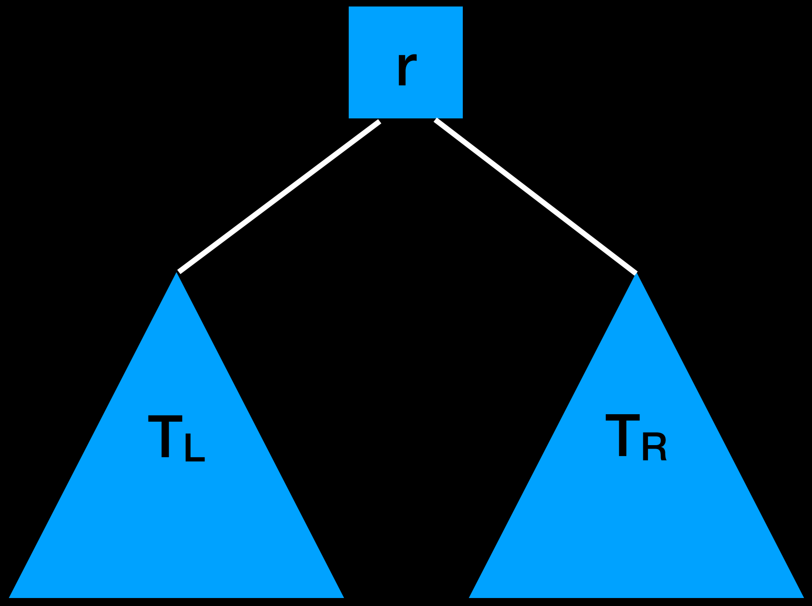


Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse TL
    traverse TR
}
```



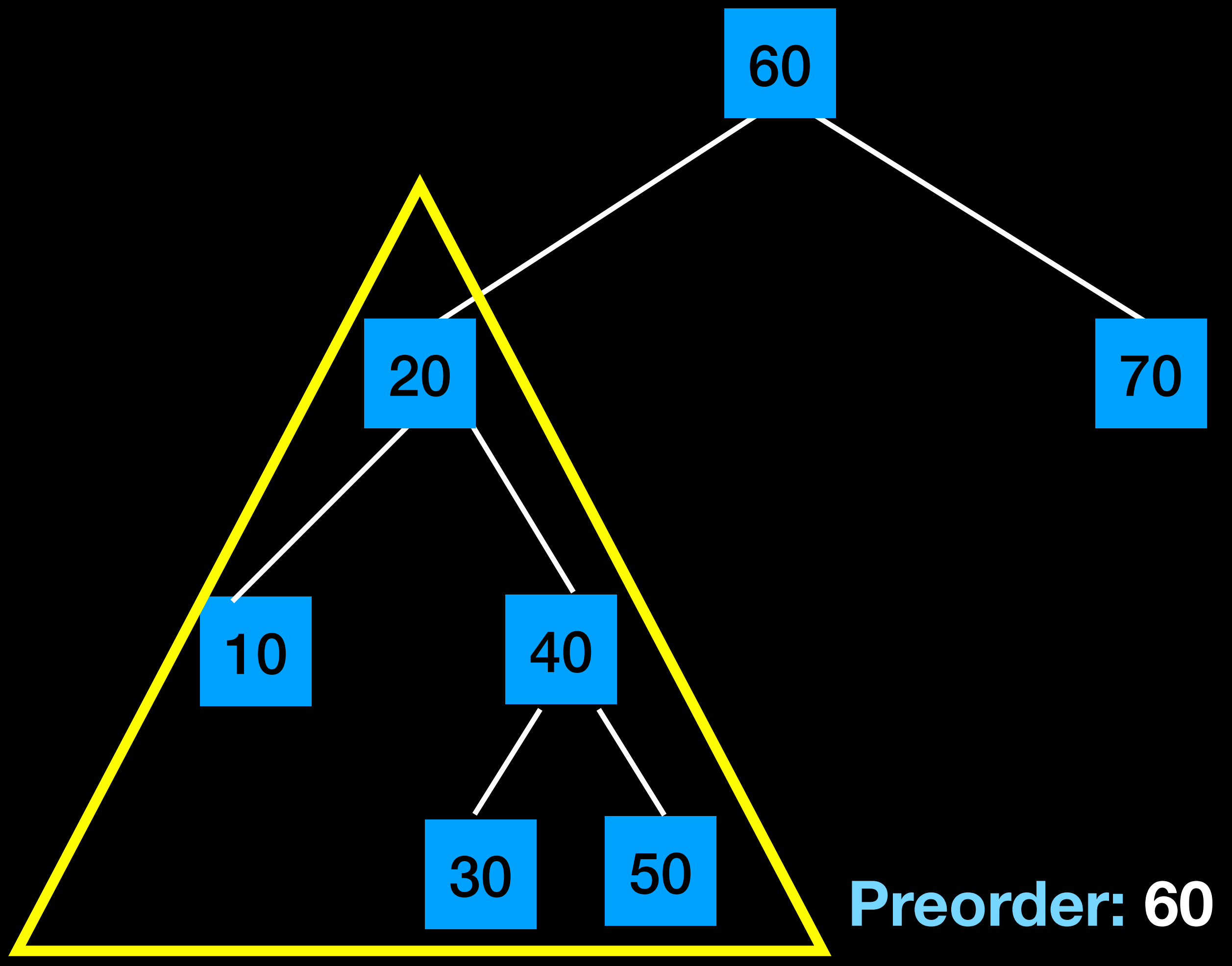


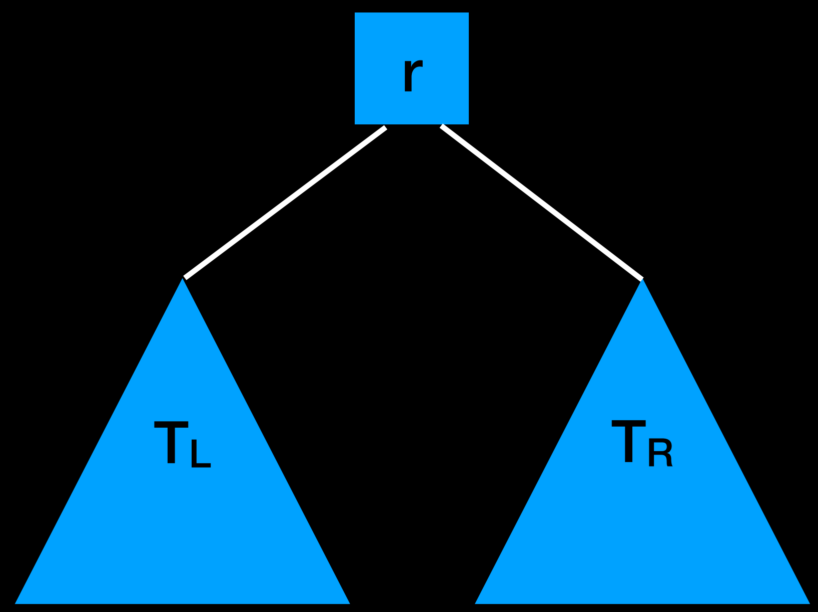
Preorder:

Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse TL
    traverse TR
}
```

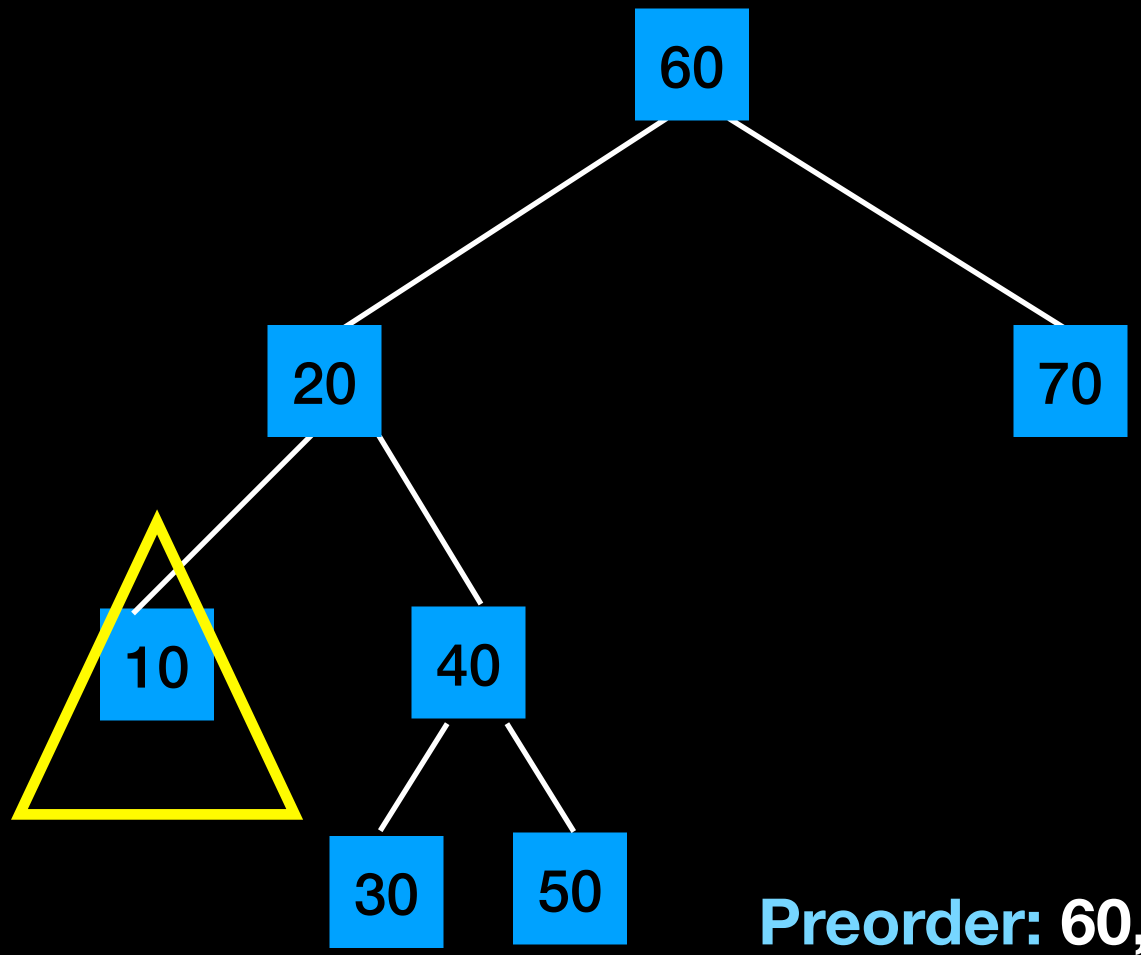


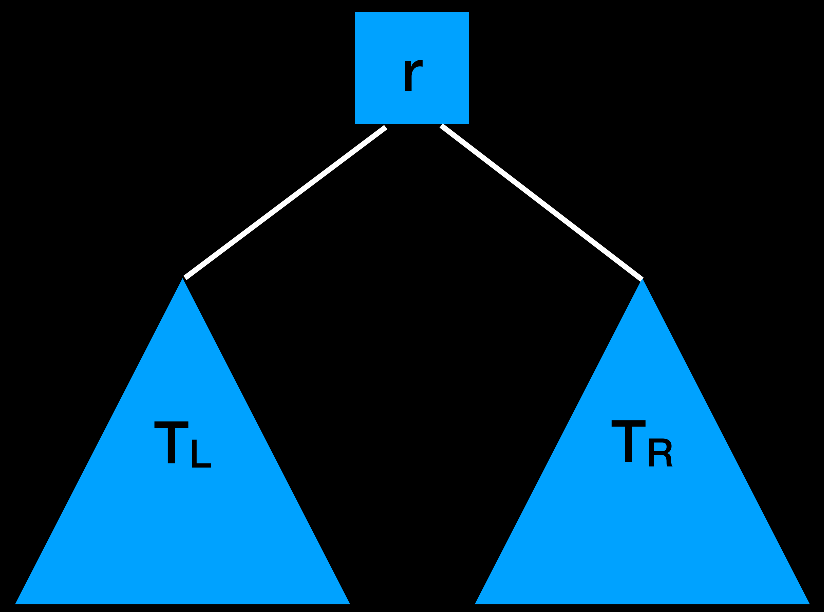


Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse TL
    traverse TR
}
```



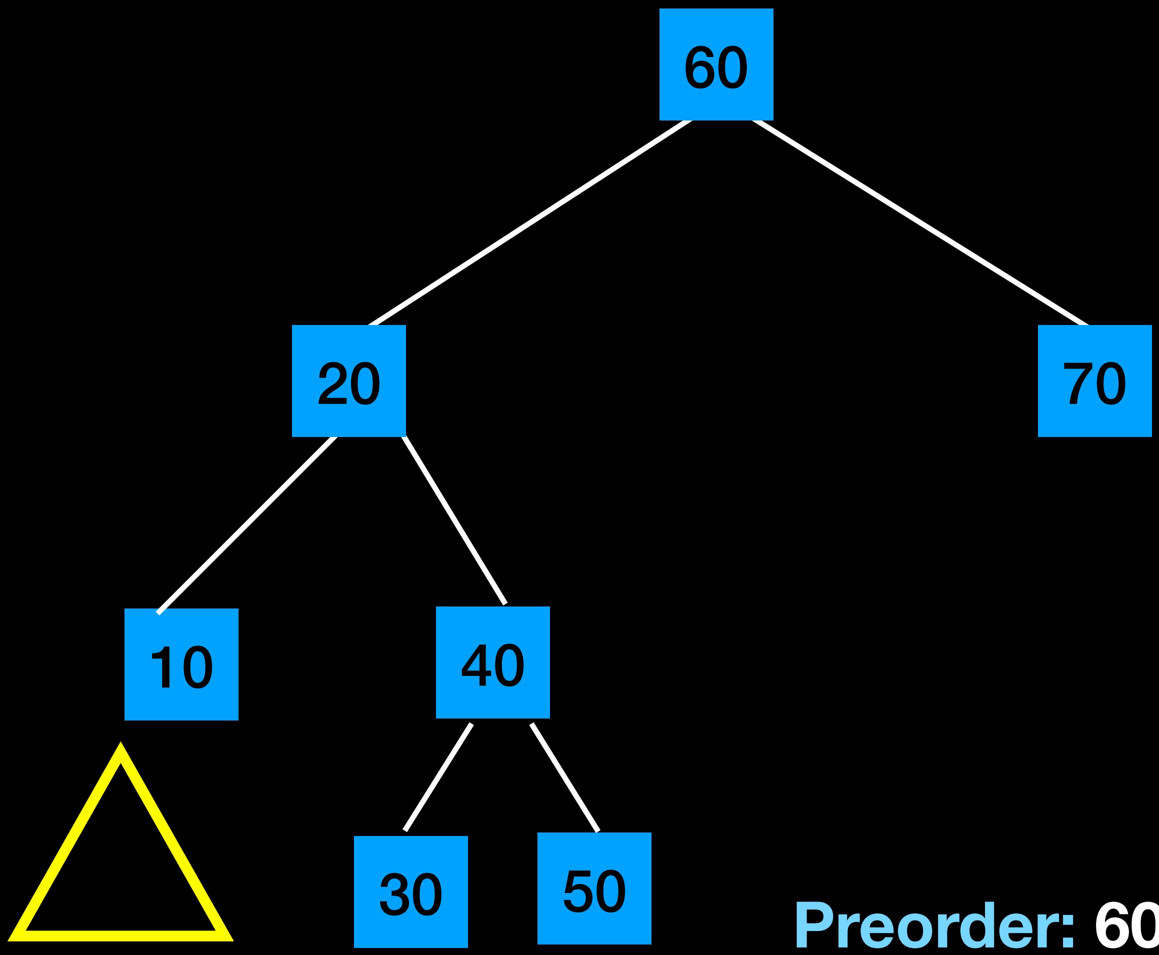


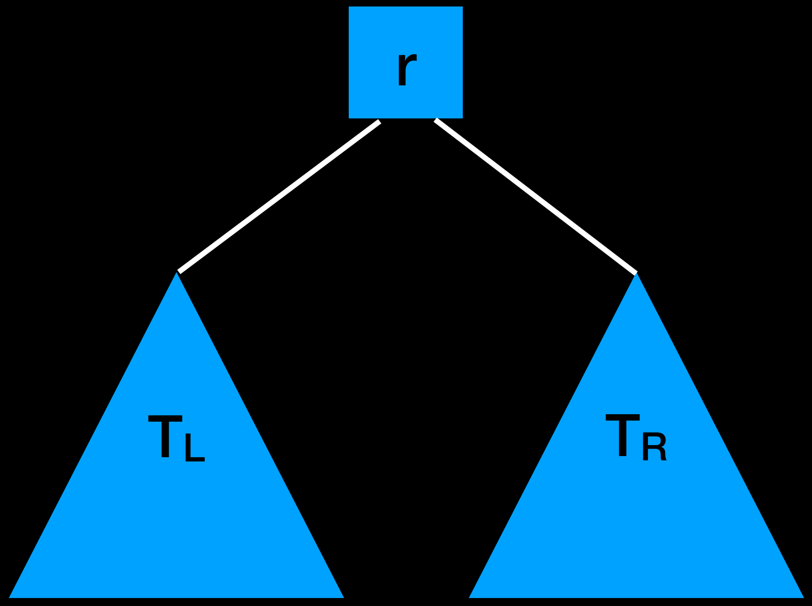
Preorder: 60, 20

Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse TL
    traverse TR
}
```



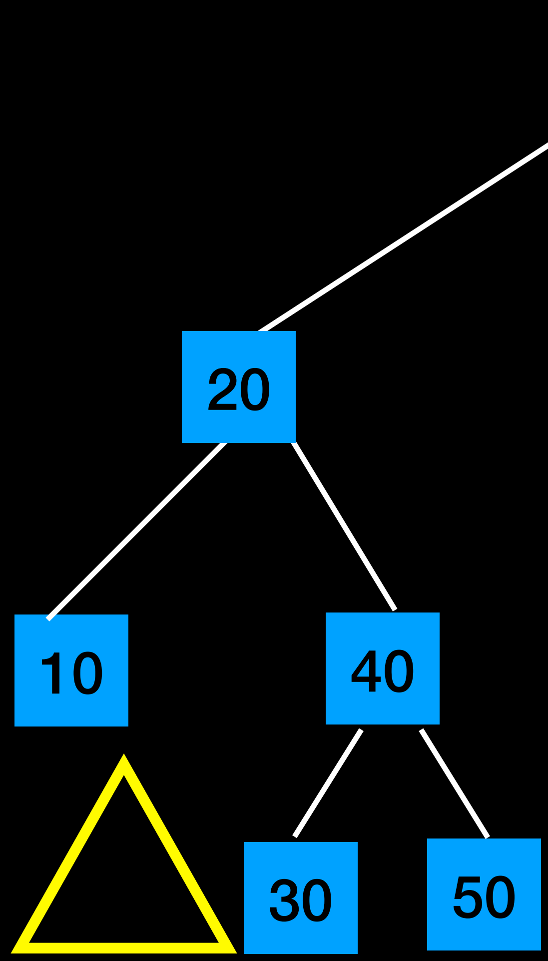


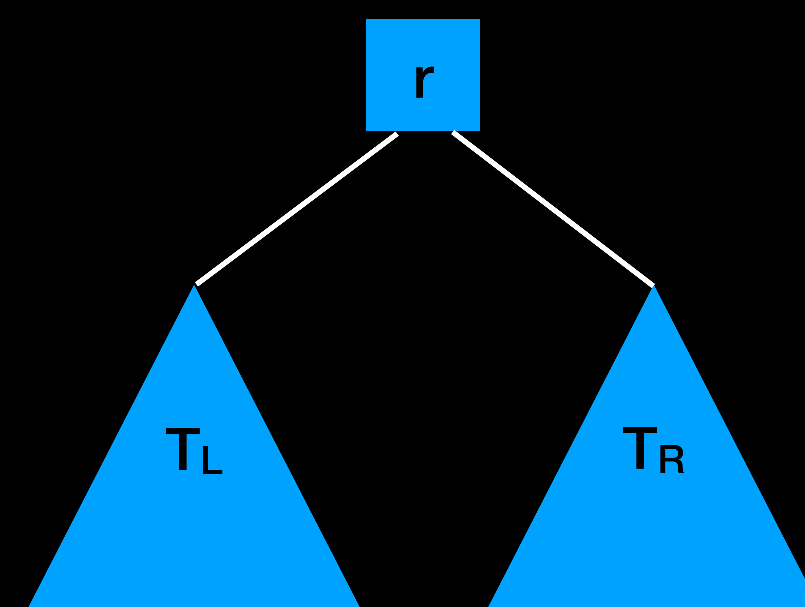
Preorder: 60, 20, 10

Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse T_L
    traverse T_R
}
```



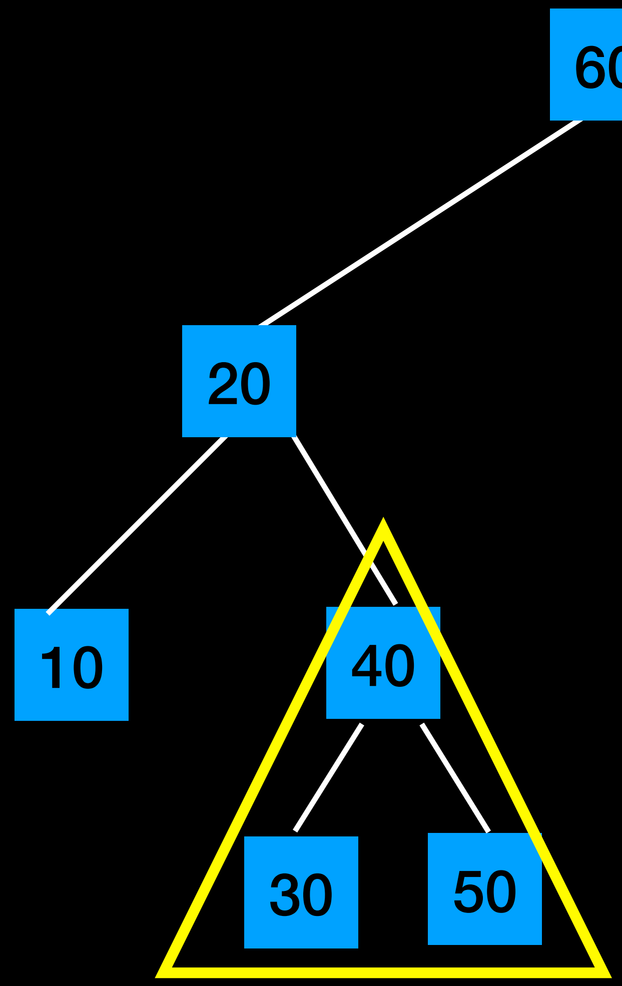


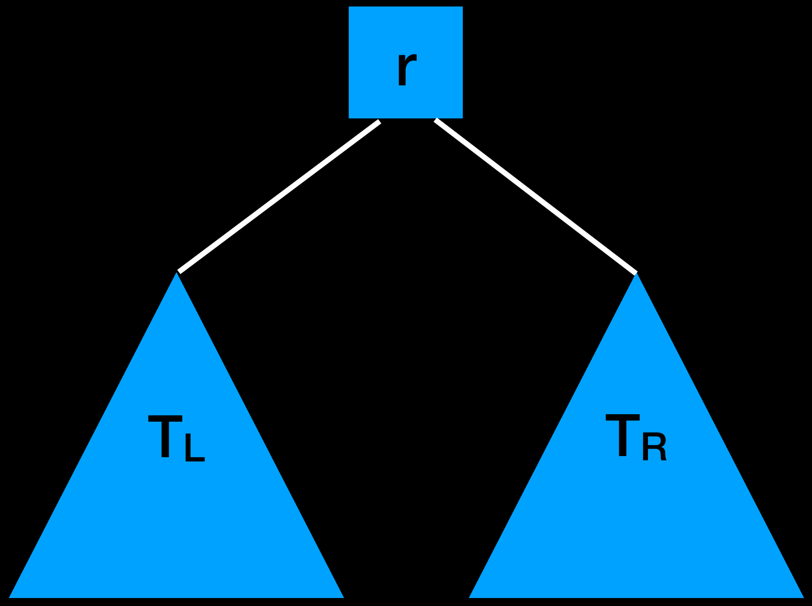
Preorder: 60, 20, 10

Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse TL
    traverse TR
}
```



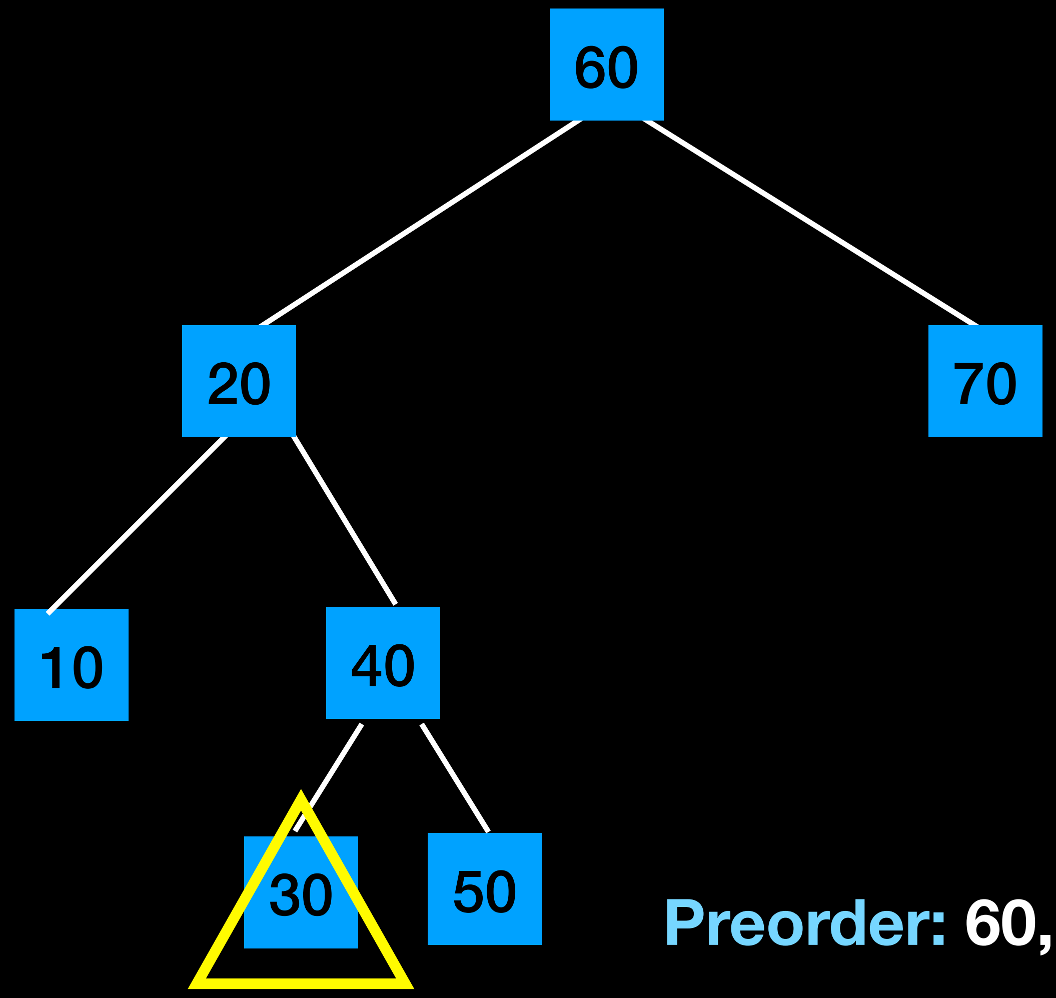


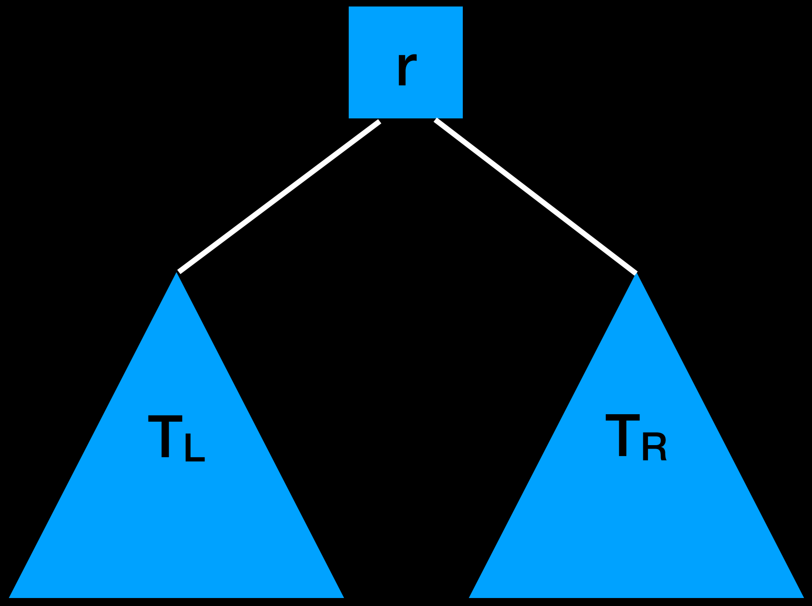
Preorder: 60, 20, 10

Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse TL
    traverse TR
}
```



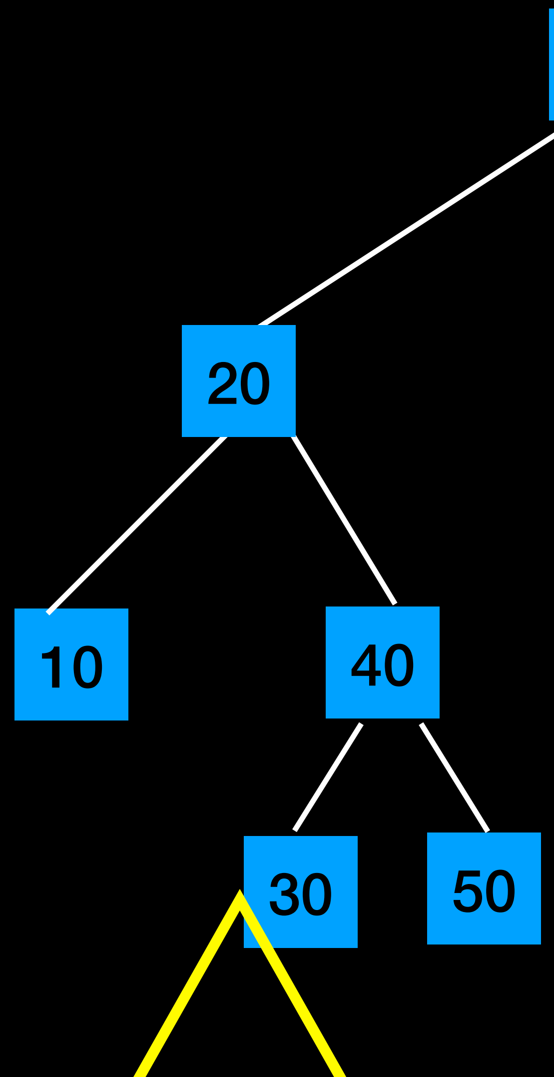


Preorder: 60, 20, 10, 40

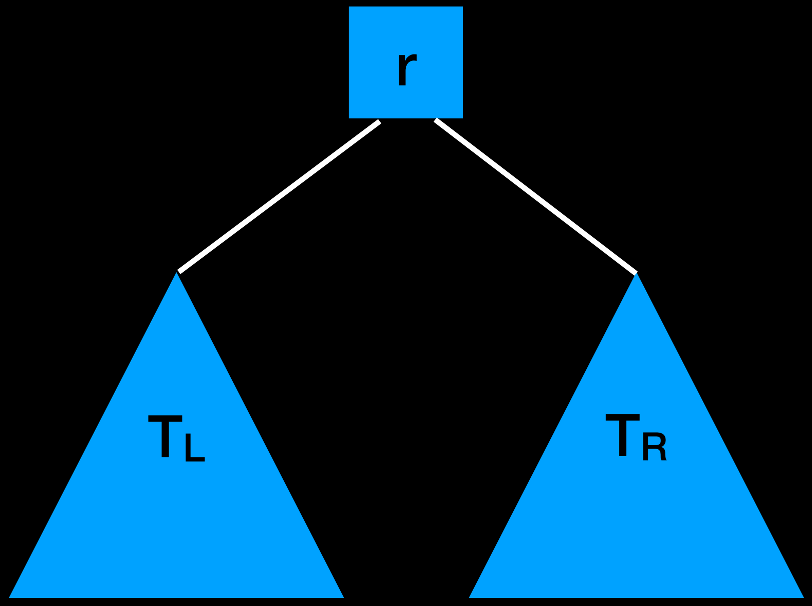
Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse TL
    traverse TR
}
```



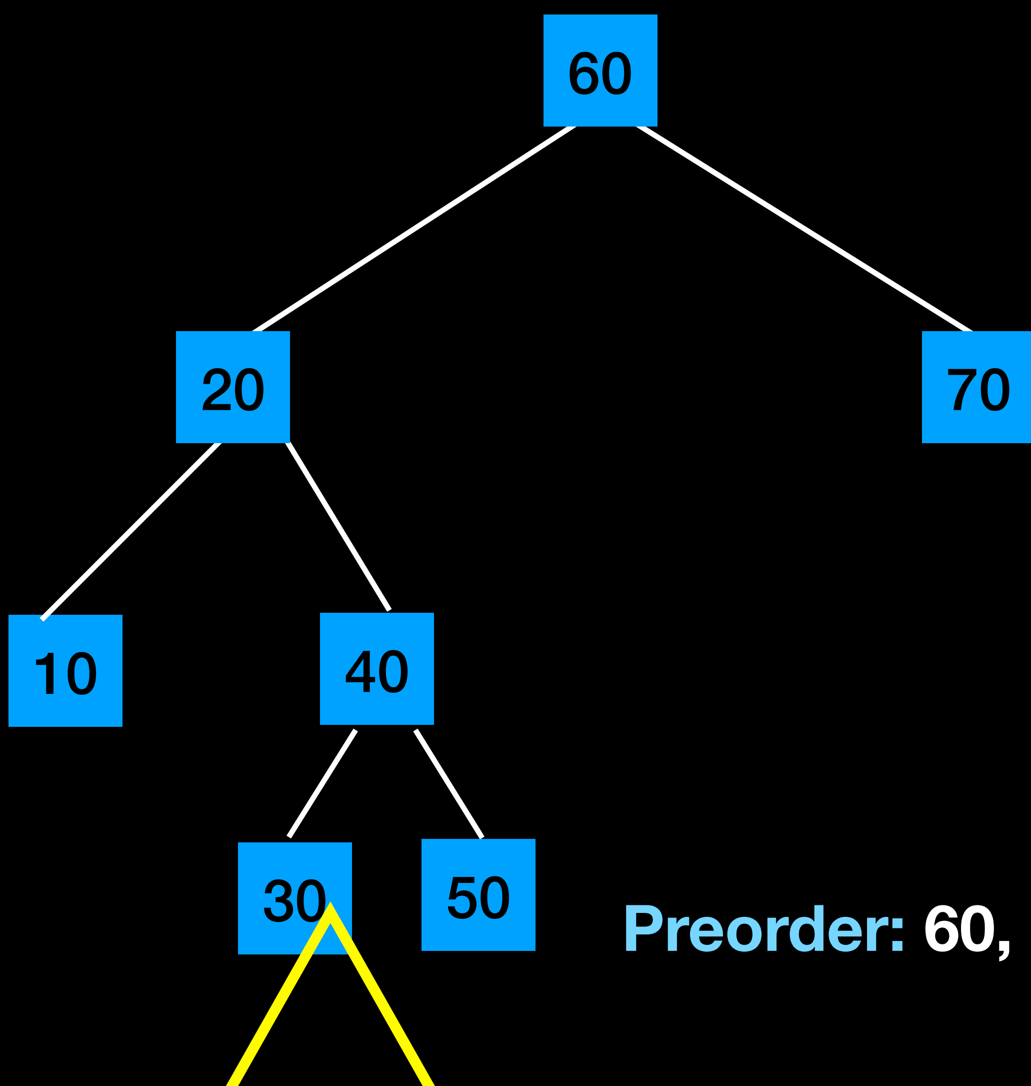
Preorder: 60, 20, 10, 40, 30

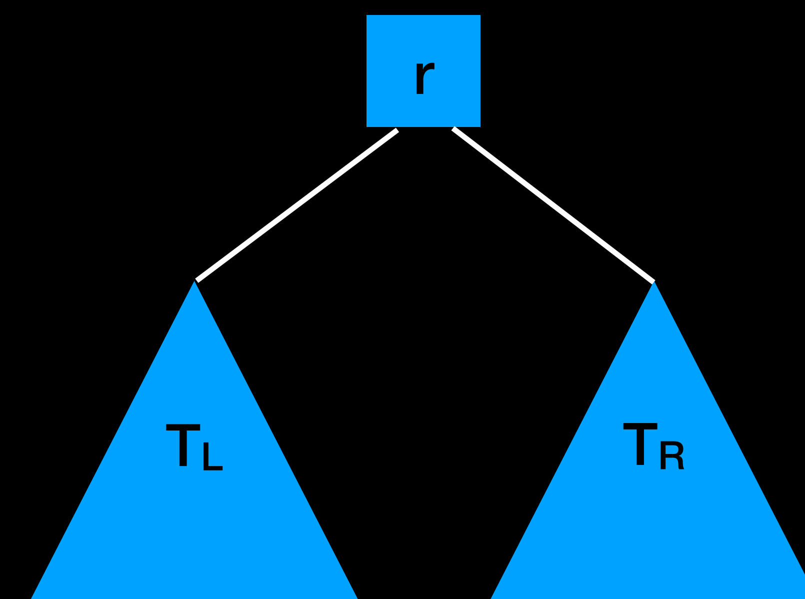


Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse T_L
    traverse T_R
}
```



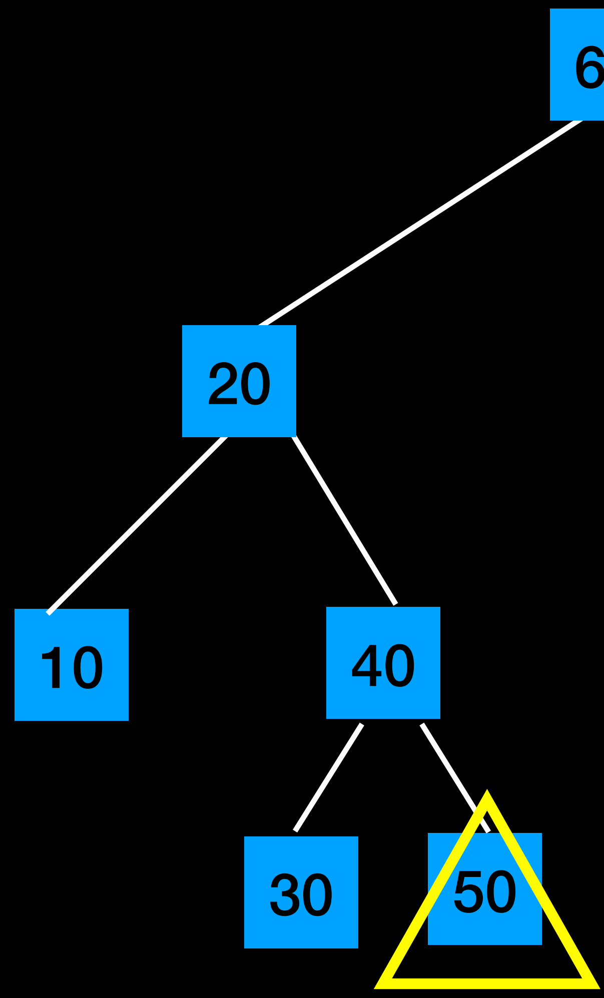


Preorder: **60, 20, 10, 40, 30**

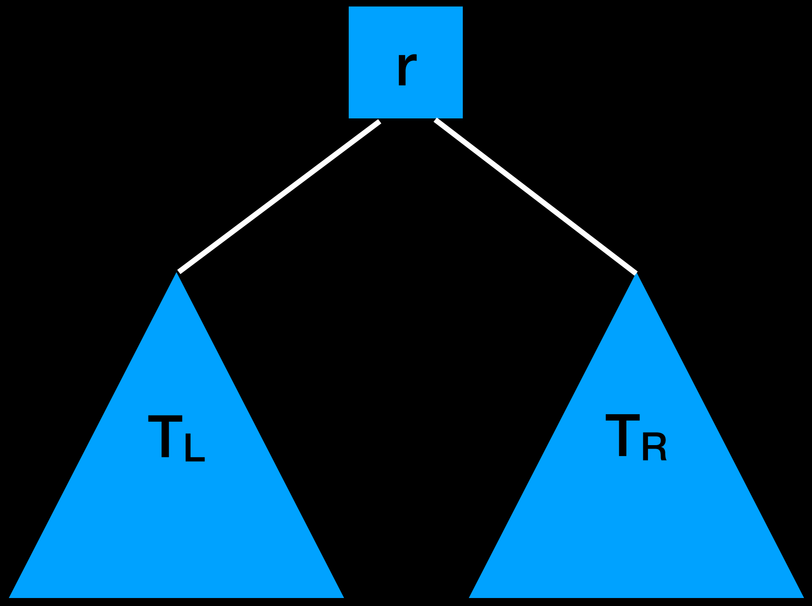
Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse TL
    traverse TR
}
```



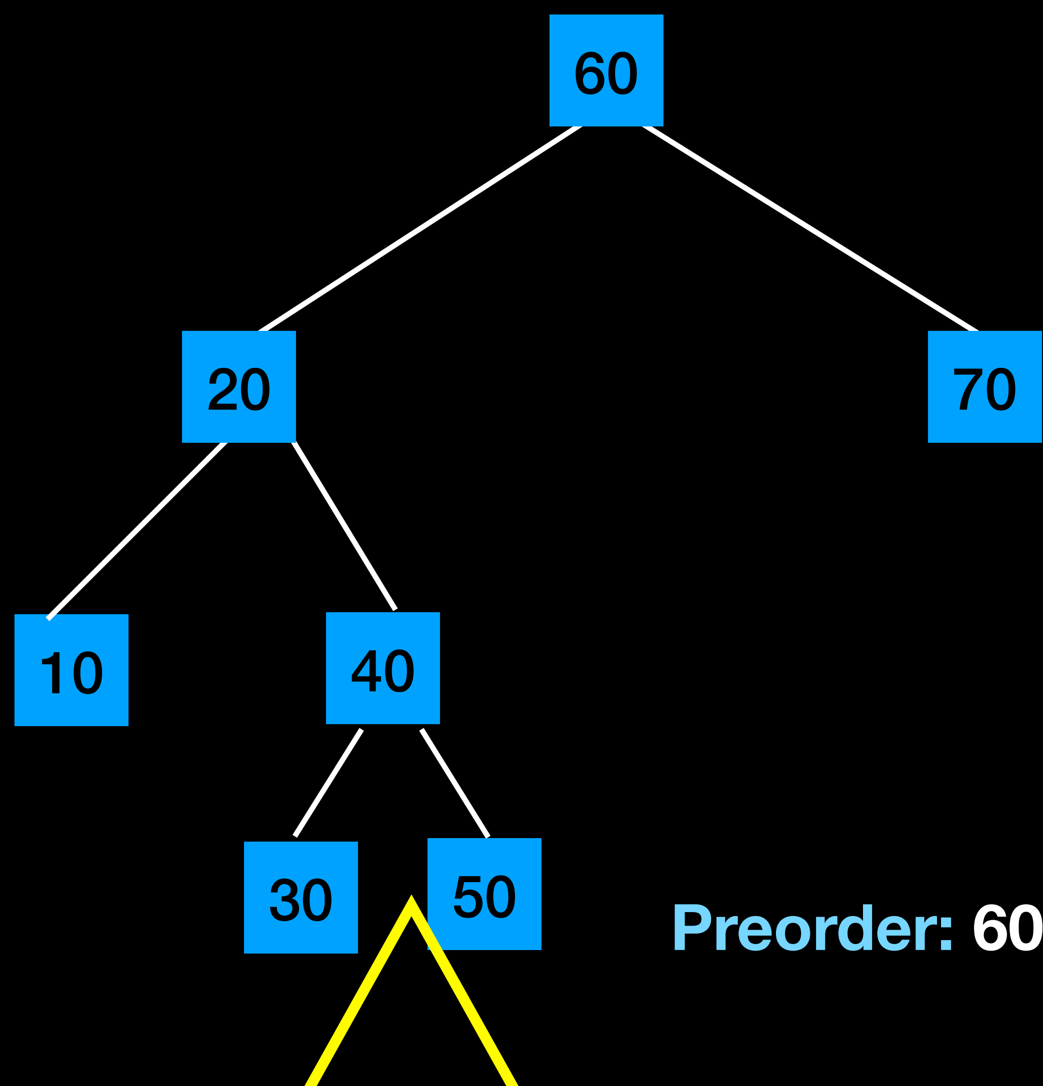
Preorder: **60, 20, 10, 40, 30**

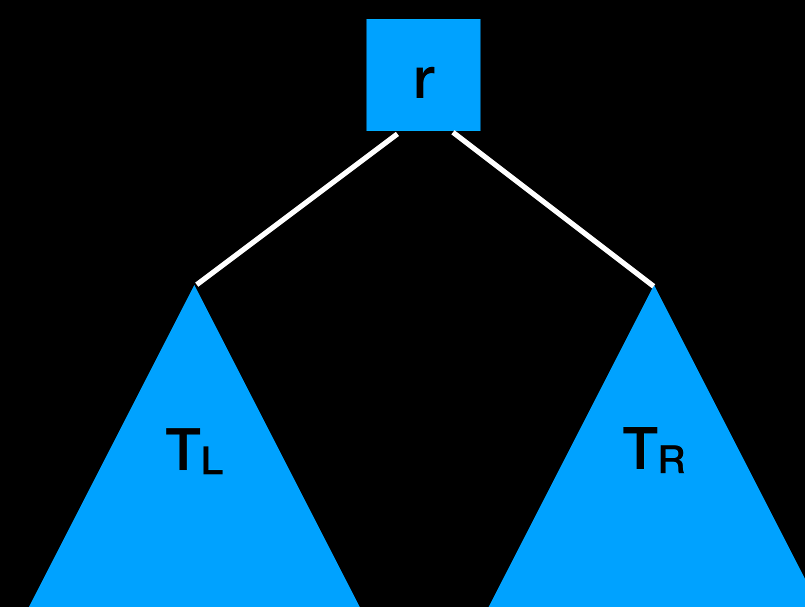


Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse T_L
    traverse T_R
}
```



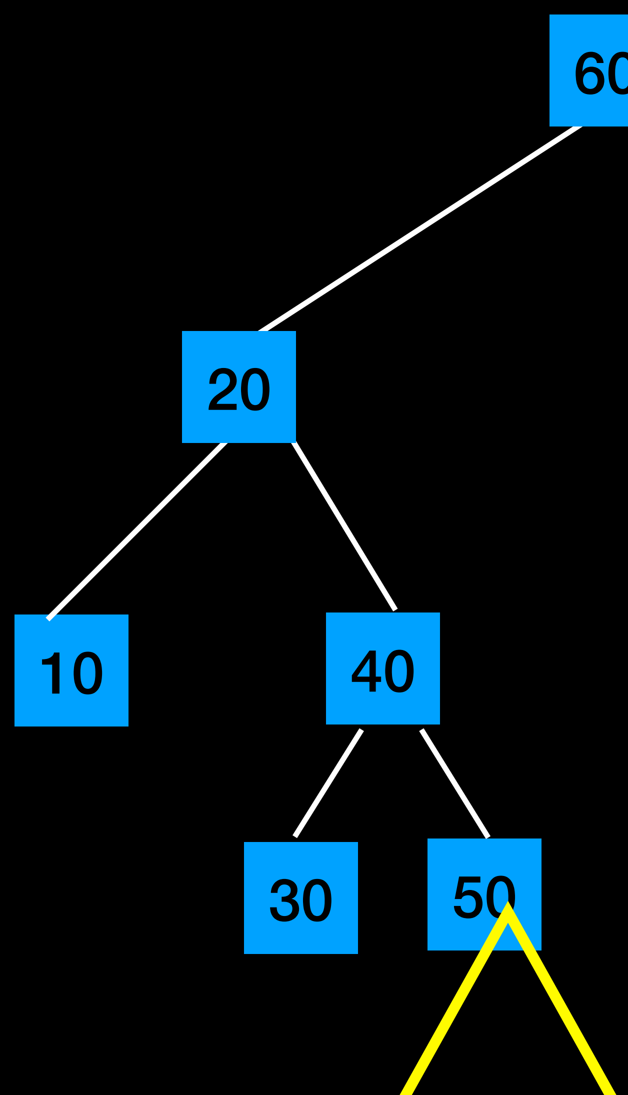


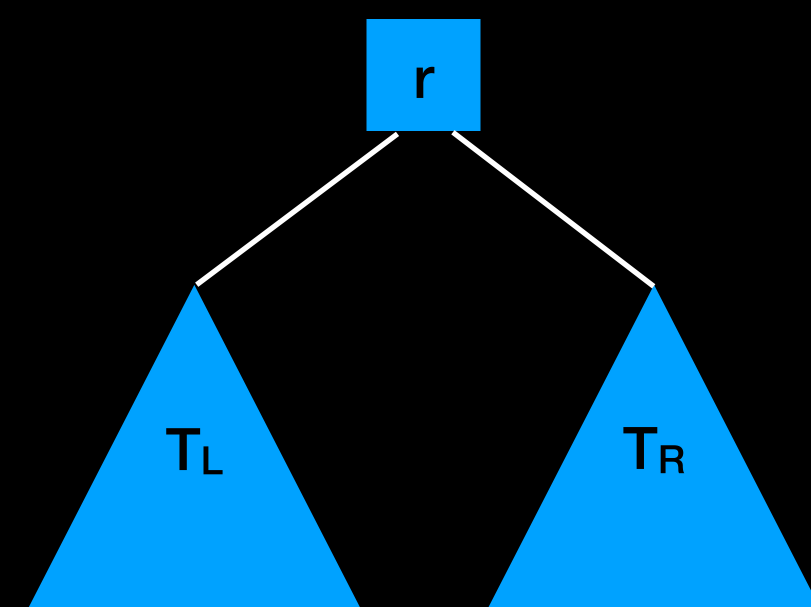
Preorder: **60, 20, 10, 40, 30, 50**

Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse T_L
    traverse T_R
}
```



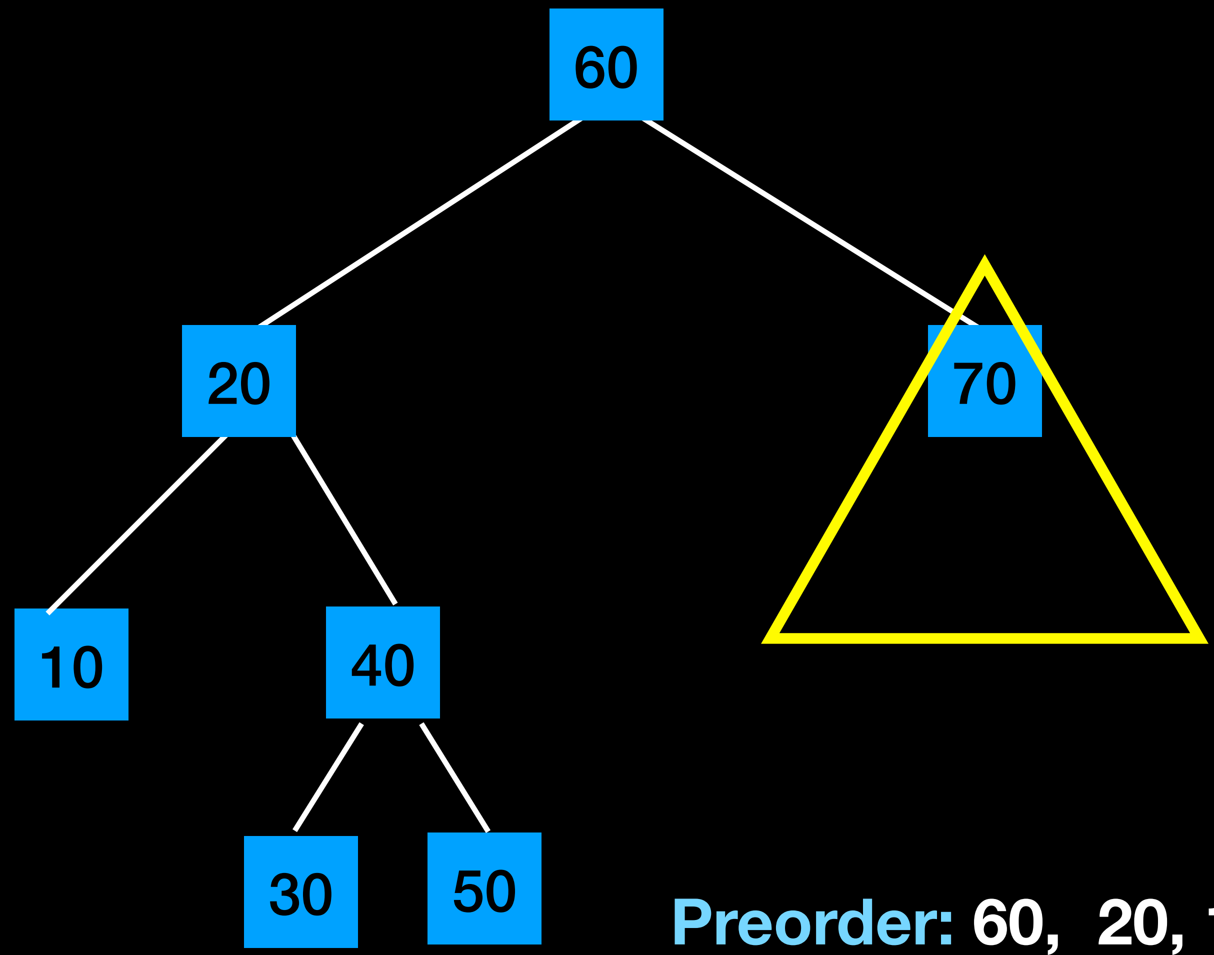


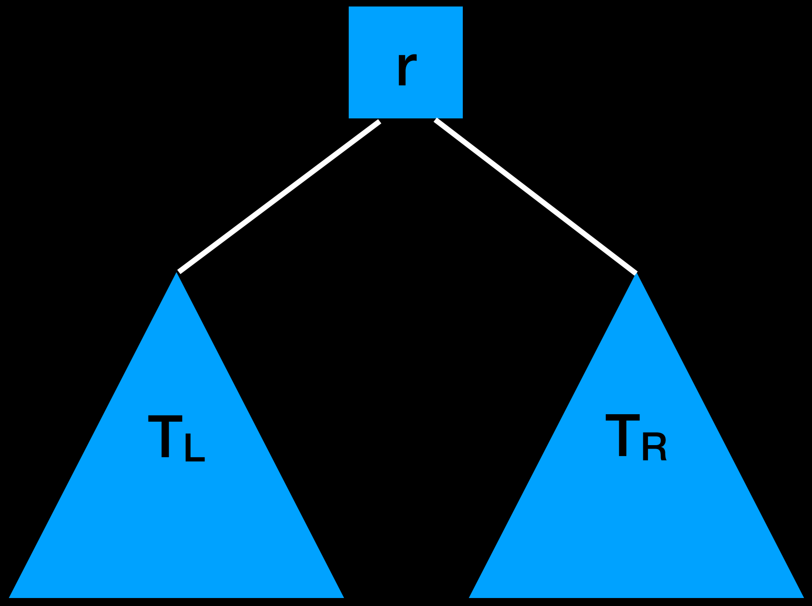
Preorder: 60, 20, 10, 40, 30, 50

Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse TL
    traverse TR
}
```



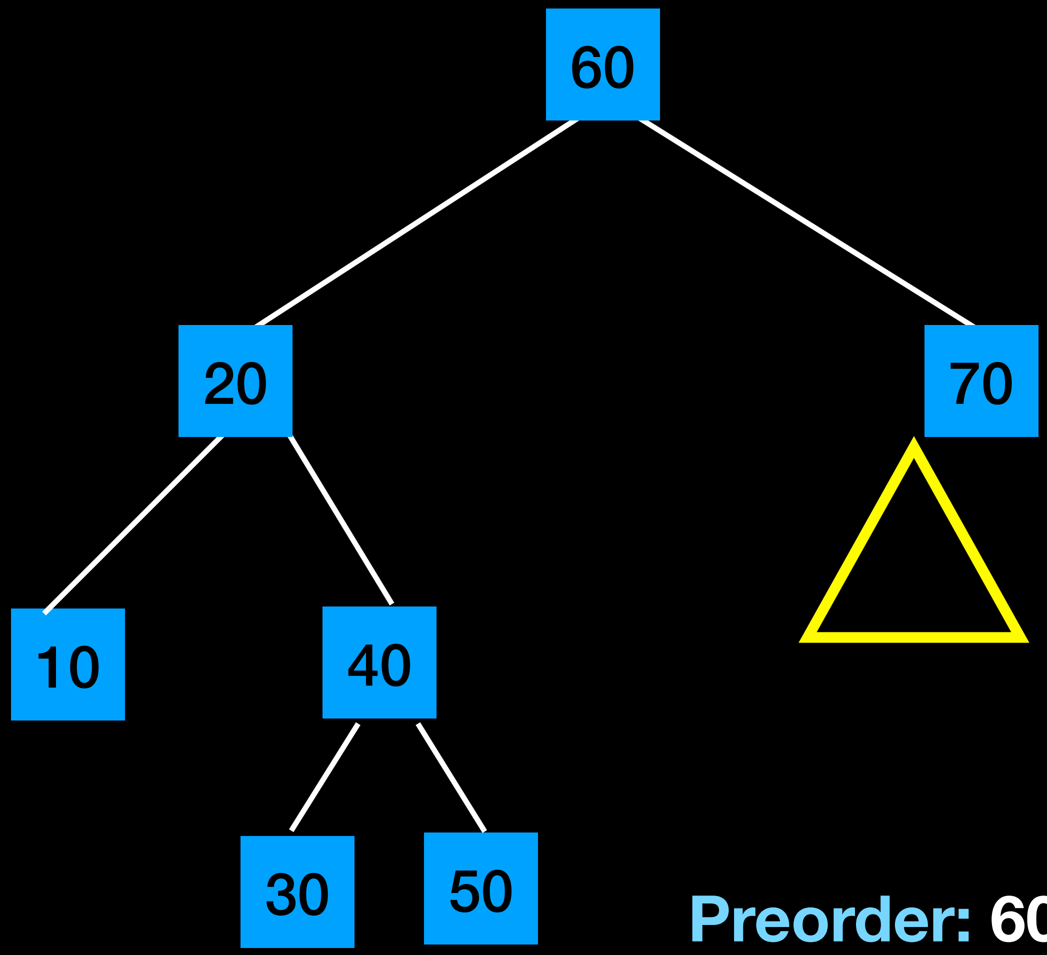


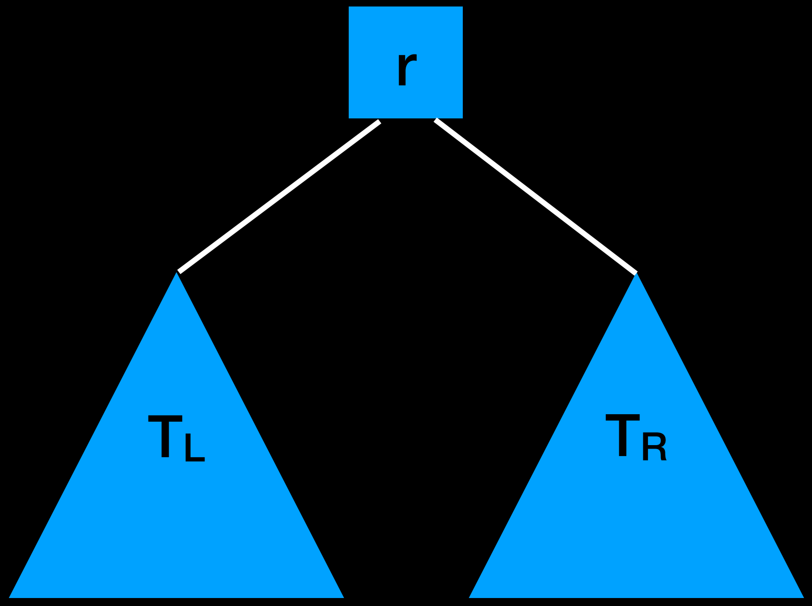
Preorder: 60, 20, 10, 40, 30, 50

Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse TL
    traverse TR
}
```



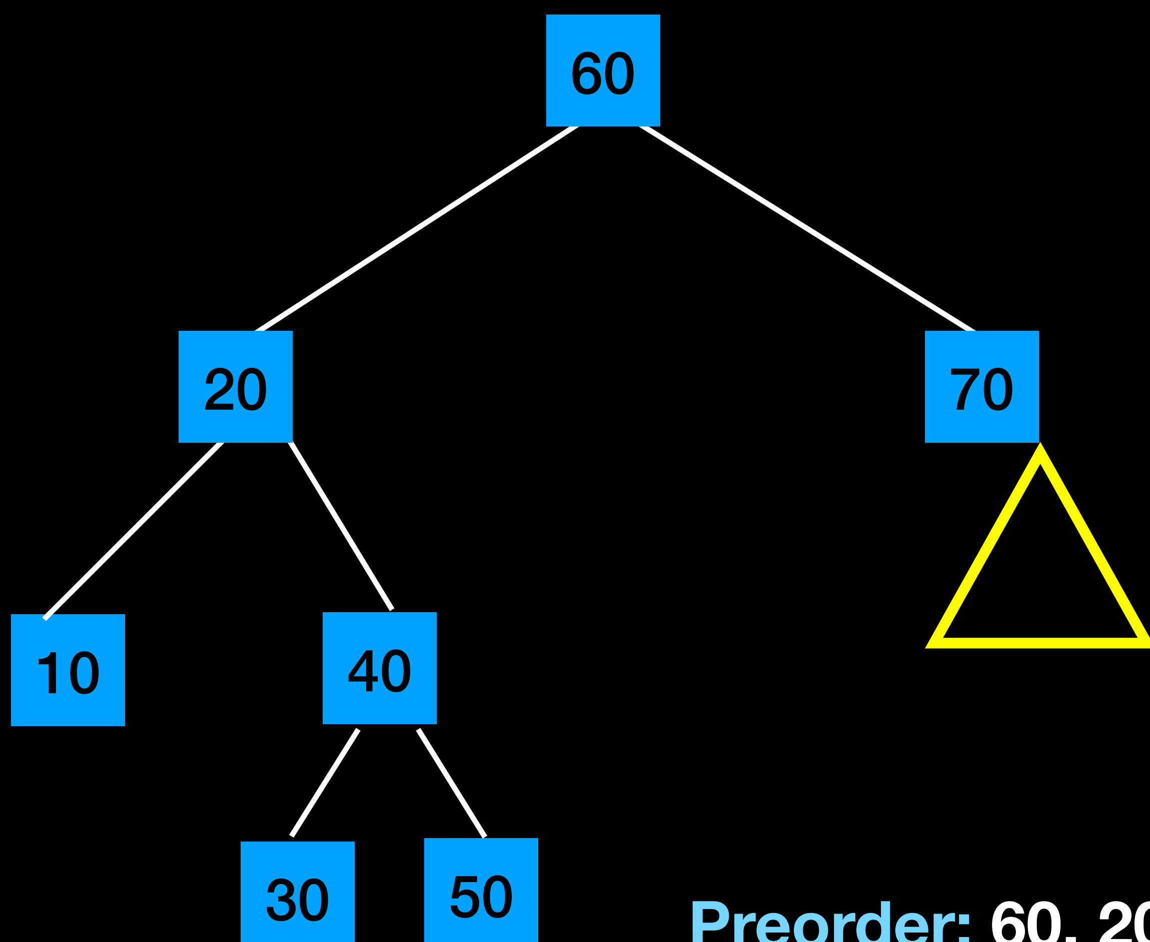


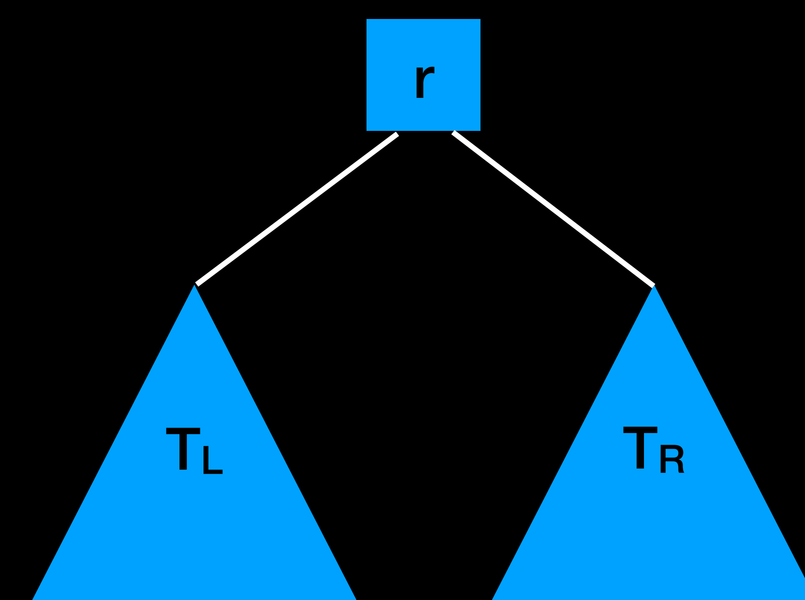
Preorder: 60, 20, 10, 40, 30, 50, 70

Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse TL
    traverse TR
}
```



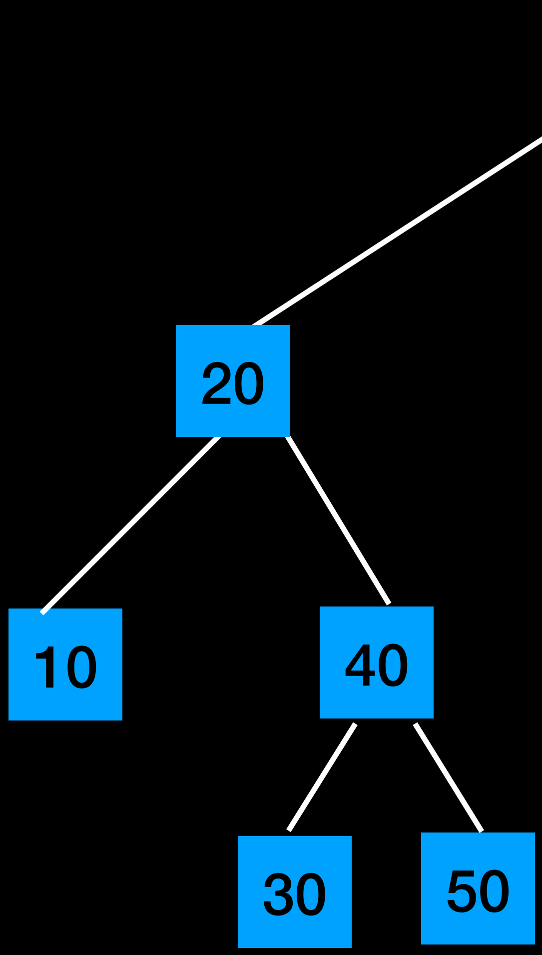


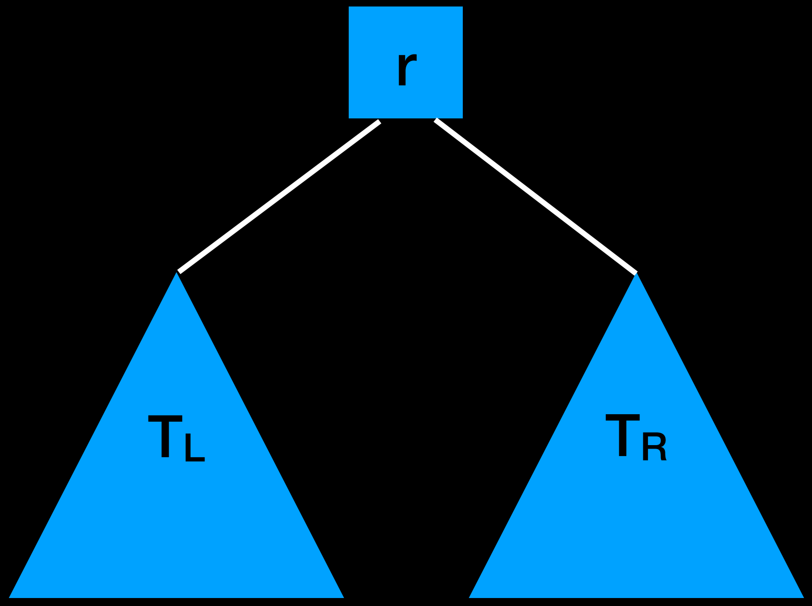
Preorder: 60, 20, 10, 40, 30, 50, 70

Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse TL
    traverse TR
}
```



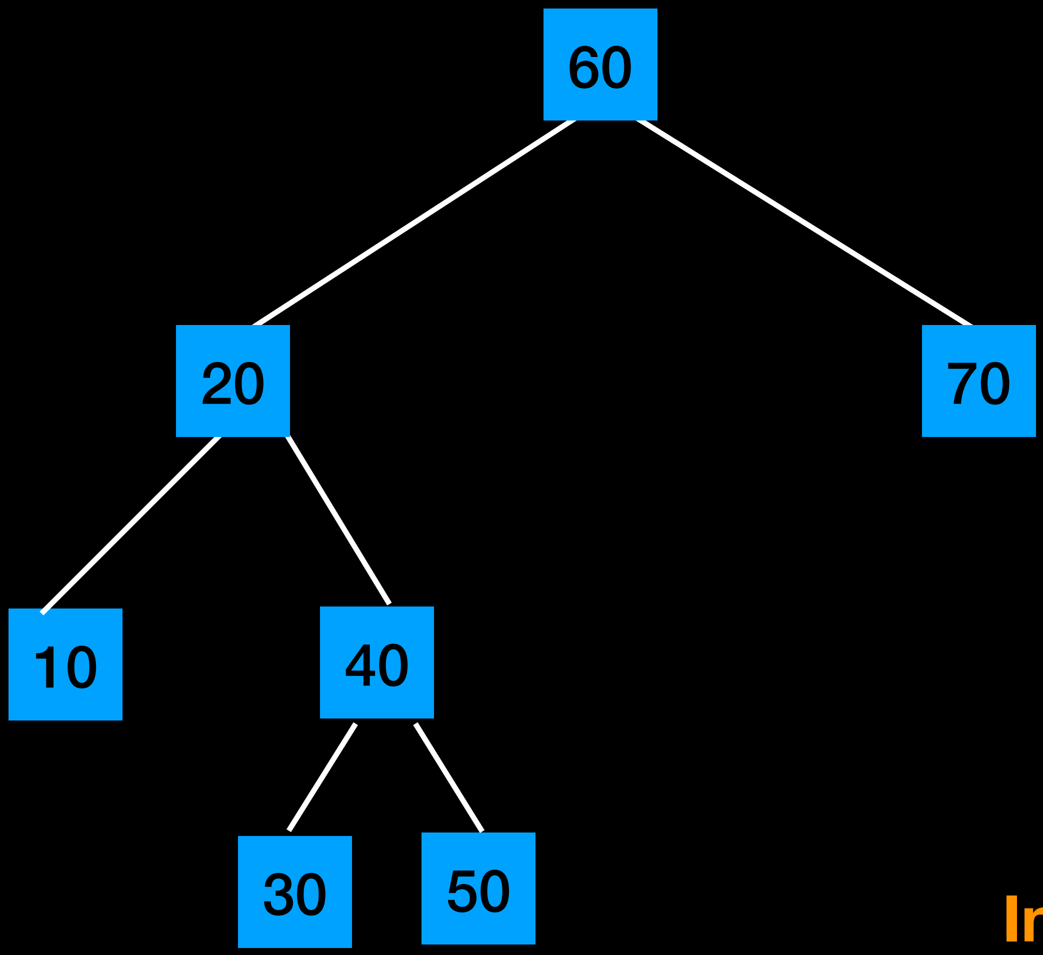


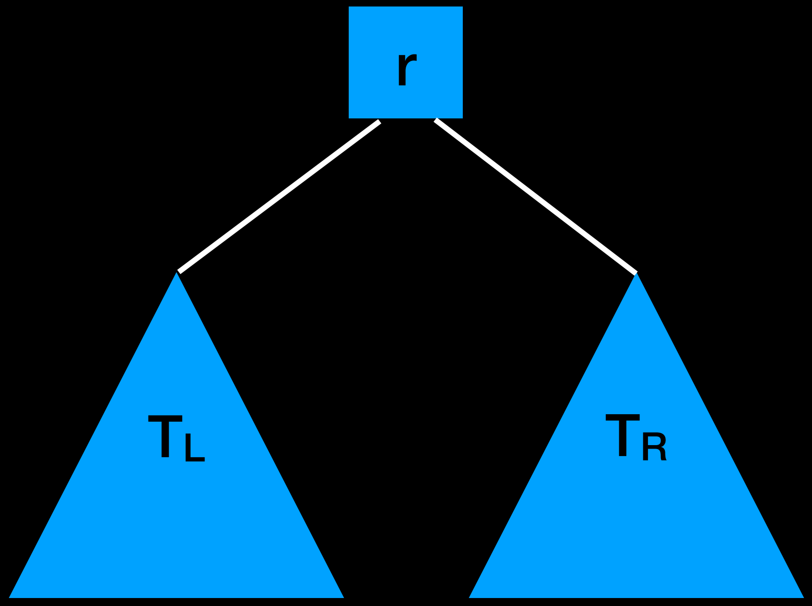
Preorder: 60, 20, 10, 40, 30, 50, 70

Visit (retrieve, print, modify …) every node in the tree

Inorder Traversal:

```
if (T is not empty) //implicit base case
{
    traverse T_L
    visit the root r
    traverse T_R
}
```



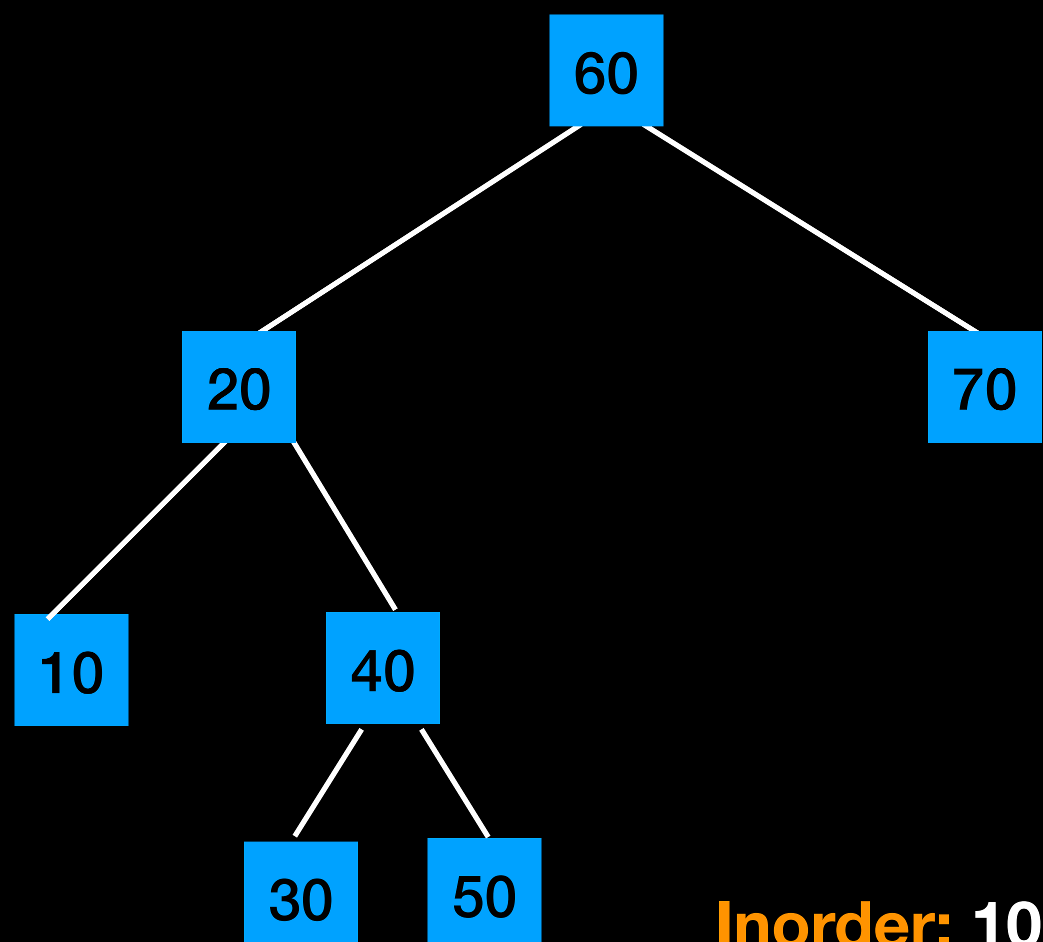


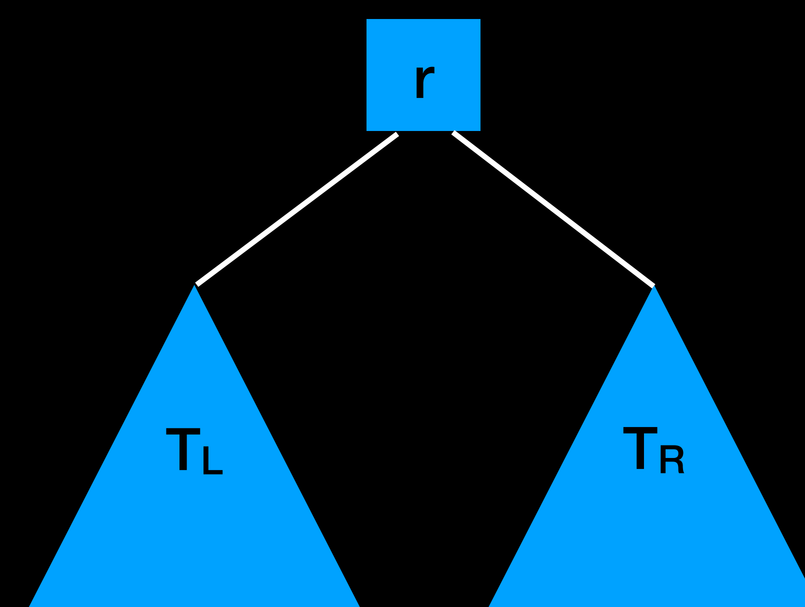
Inorder: ???

Visit (retrieve, print, modify …) every node in the tree

Inorder Traversal:

```
if (T is not empty) //implicit base case
{
    traverse T_L
    visit the root r
    traverse T_R
}
```



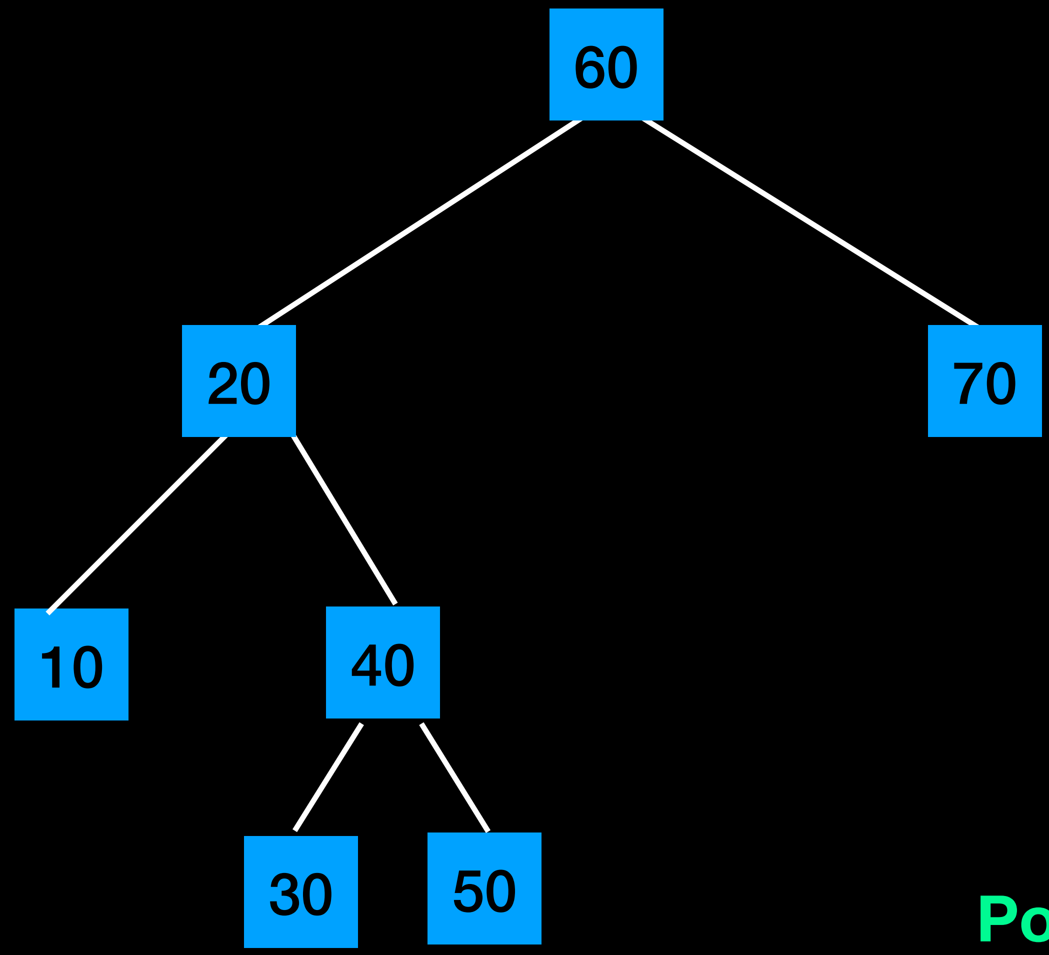


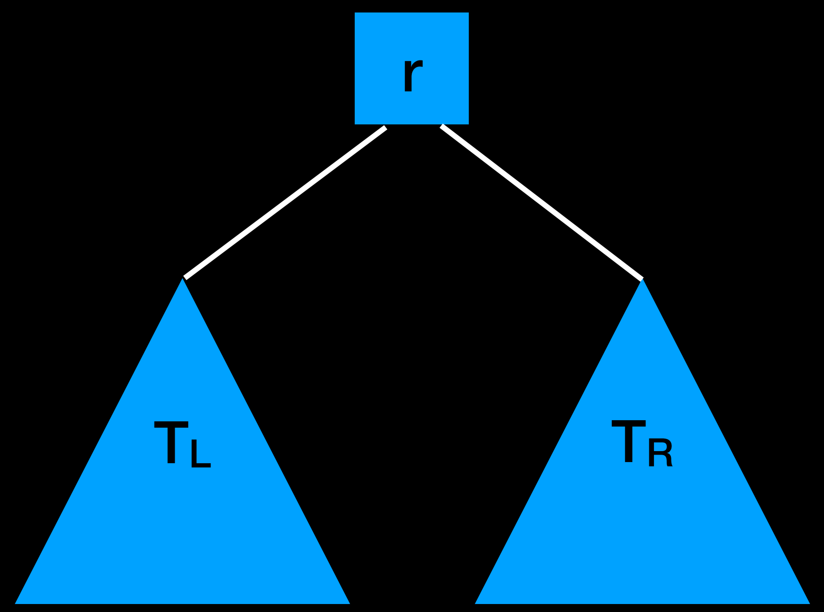
Inorder: **10, 20, 30, 40, 50, 60, 70**

Visit (retrieve, print, modify …) every node in the tree

Postorder Traversal:

```
if (T is not empty) //implicit base case
{
    traverse TL
    traverse TR
    visit the root r
}
```



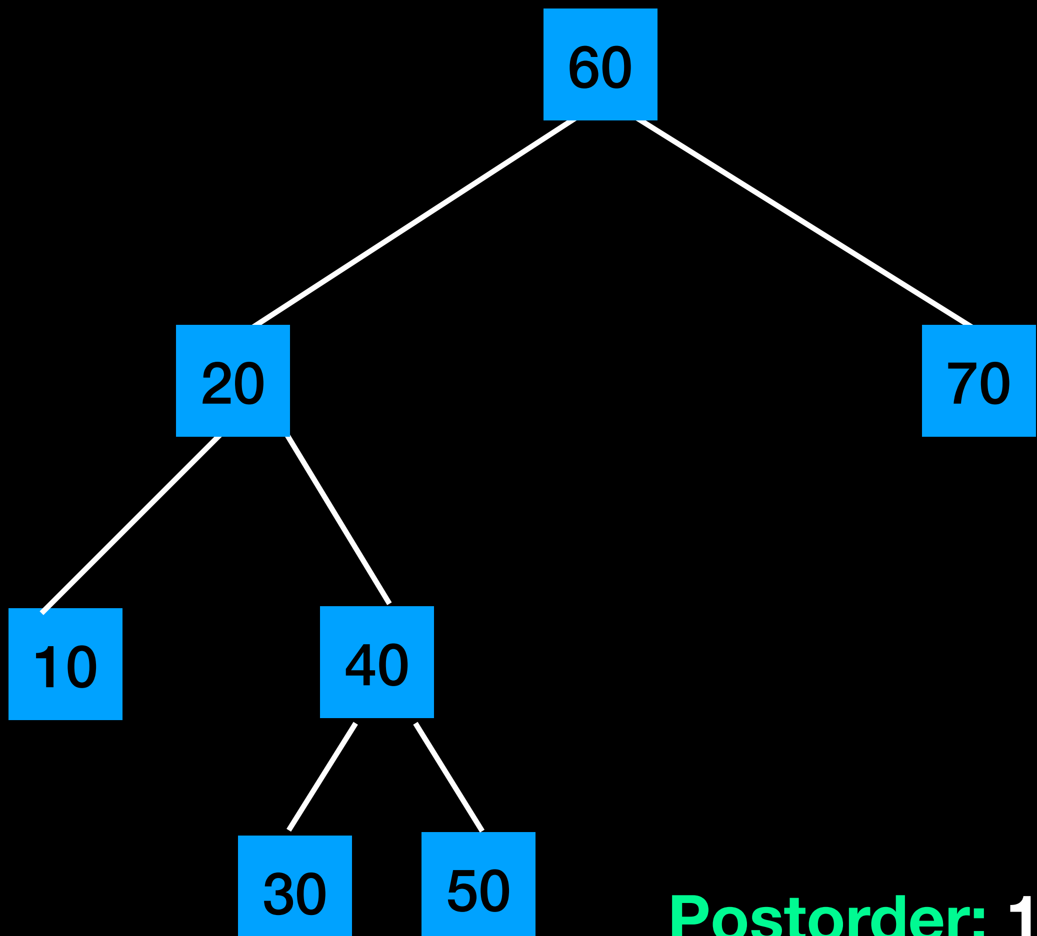


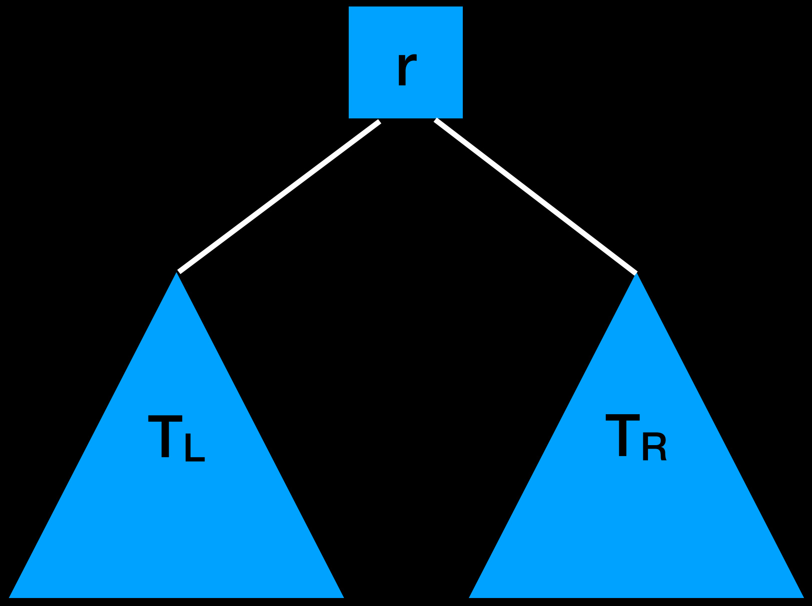
Postorder: ???

Visit (retrieve, print, modify …) every node in the tree

Postorder Traversal:

```
if (T is not empty) //implicit base case
{
    traverse T_L
    traverse T_R
    visit the root r
}
```





Postorder: 10, 30, 50, 40, 20, 70, 60

BinaryTree ADT Operations

```
#ifndef BinaryTree_H_
#define BinaryTree_H_

template<class T>
class BinaryTree
{

public:
    BinaryTree(); // constructor
    BinaryTree(const BinaryTree<T>& tree); // copy constructor
    ~BinaryTree(); // destructor
    bool isEmpty() const;
    size_t getHeight() const;
    size_t getNumberOfNodes() const;
    void add(const T& new_item);
    void remove(const T& new_item);
    T find(const T& item) const;
    void clear();

    void preorderTraverse(Visitor<T>& visit) const;
    void inorderTraverse(Visitor<T>& visit) const;
    void postorderTraverse(Visitor<T>& visit) const;

    BinaryTree& operator= (const BinaryTree<T>& rhs);

private: // implementation details here

}; // end BST

#include "BinaryTree.cpp"
#endif // BinaryTree_H_
```

```
#ifndef BinaryTree_H_
#define BinaryTree_H_

template<class T>
class BinaryTree
{

public:
     BinaryTree(); // constructor
     BinaryTree(const BinaryTree<T>& tree); // copy constructor
     ~BinaryTree(); // destructor
     bool isEmpty() const;
     size_t getHeight() const;
     size_t getNumberOfNodes() const;
     void add(const T& new_item);
     void remove(const T& new_item);
     T find(const T& item) const;
     void clear();

     void preorderTraverse(Visitor<T>& visit) const;
     void inorderTraverse(Visitor<T>& visit) const;
     void postorderTraverse(Visitor<T>& visit) const;

     BinaryTree& operator= (const BinaryTree<T>& rhs);

private: // implementation details here

}; // end BST

#include "BinaryTree.cpp"
#endif // BinaryTree_H_
```

How might you add
Will determine the tree structure

This is an abstract class from which we can derive desired behavior keeping the traversal general