

Trees

Tiziana Ligorio

Today's Plan



Trees

Binary Tree ADT

Binary Search Tree ADT

Announcements

Questions?

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

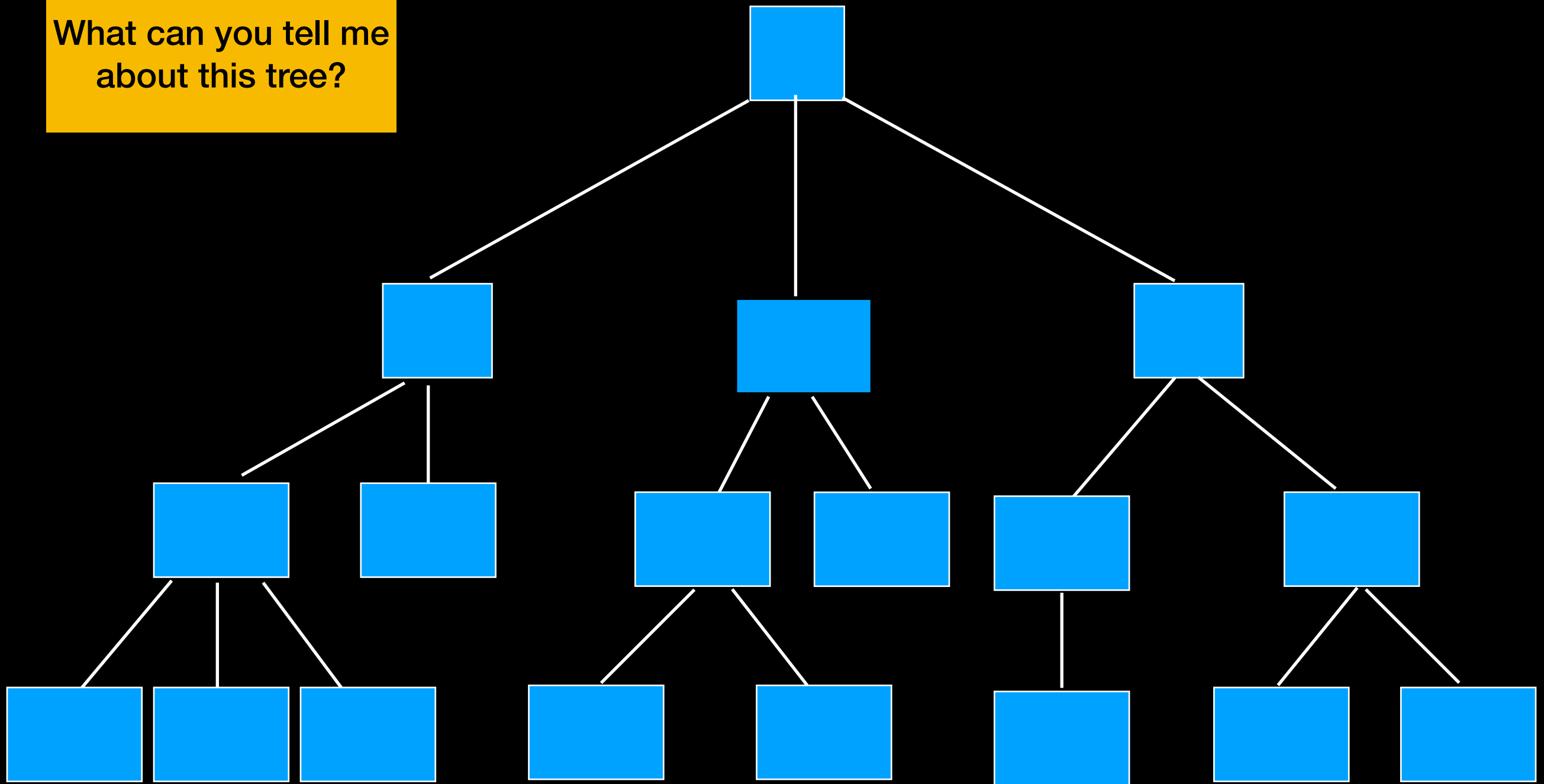
Can represent relationships

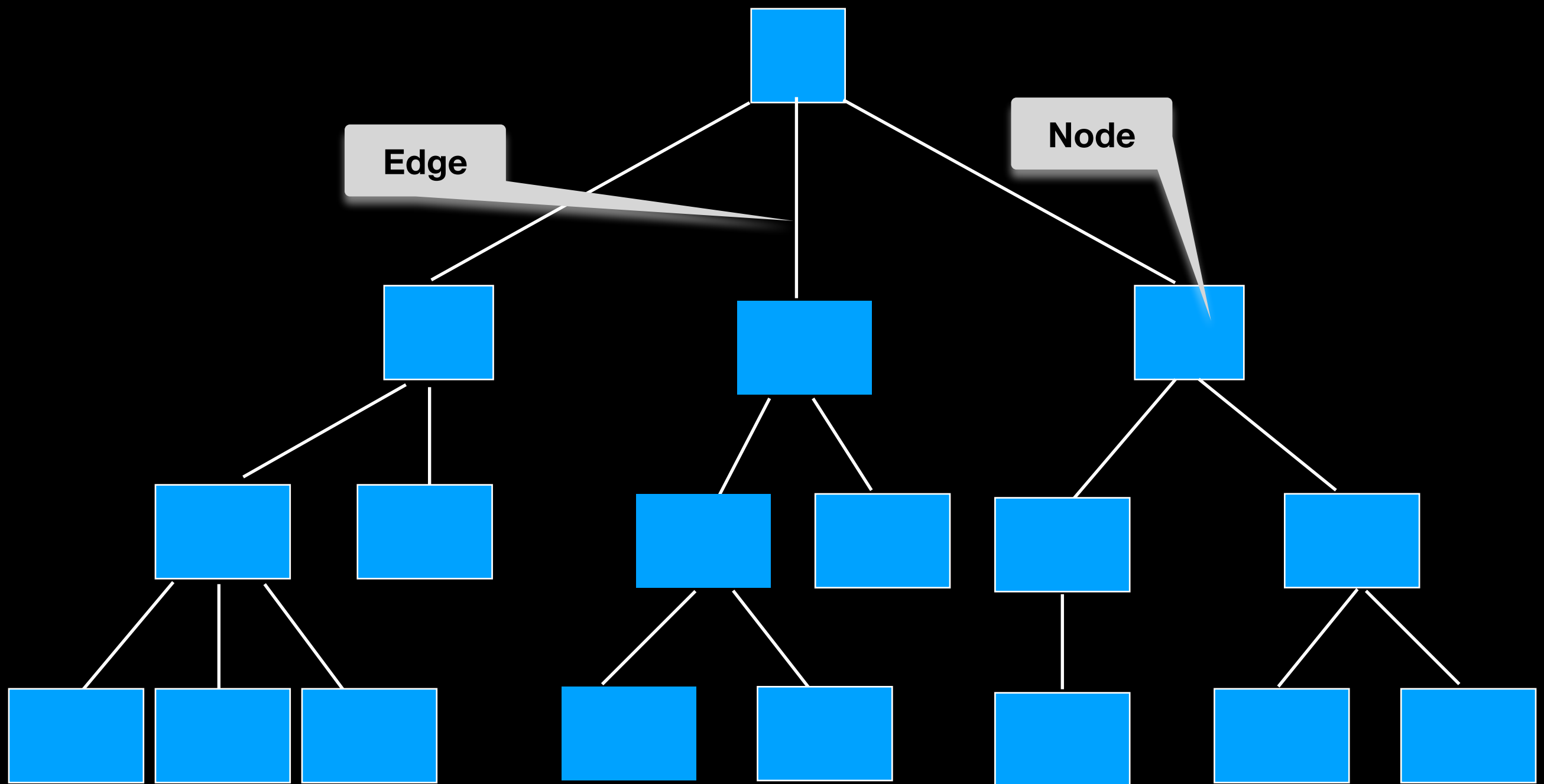
Hierarchical (directional) organization

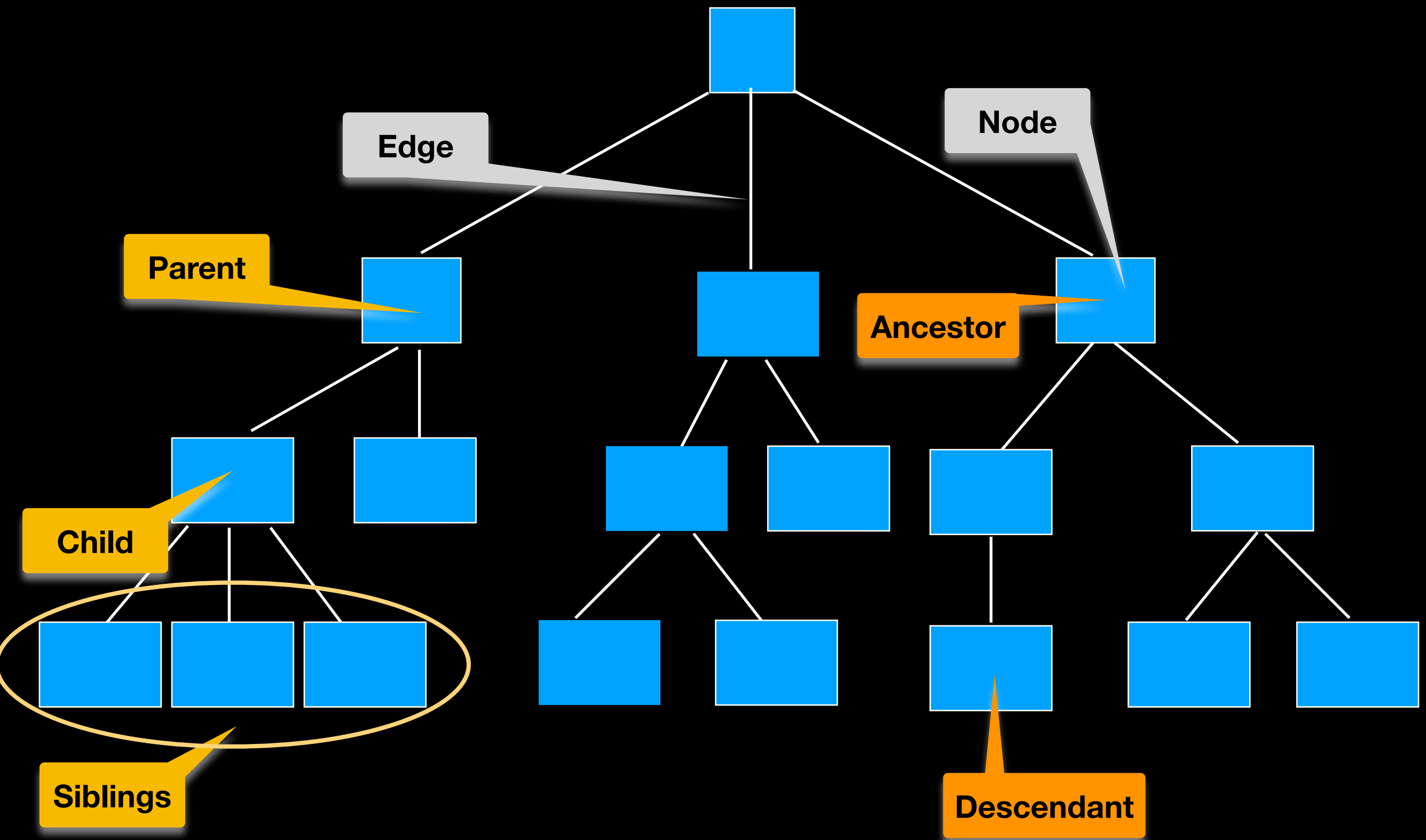
(E.g. family tree)

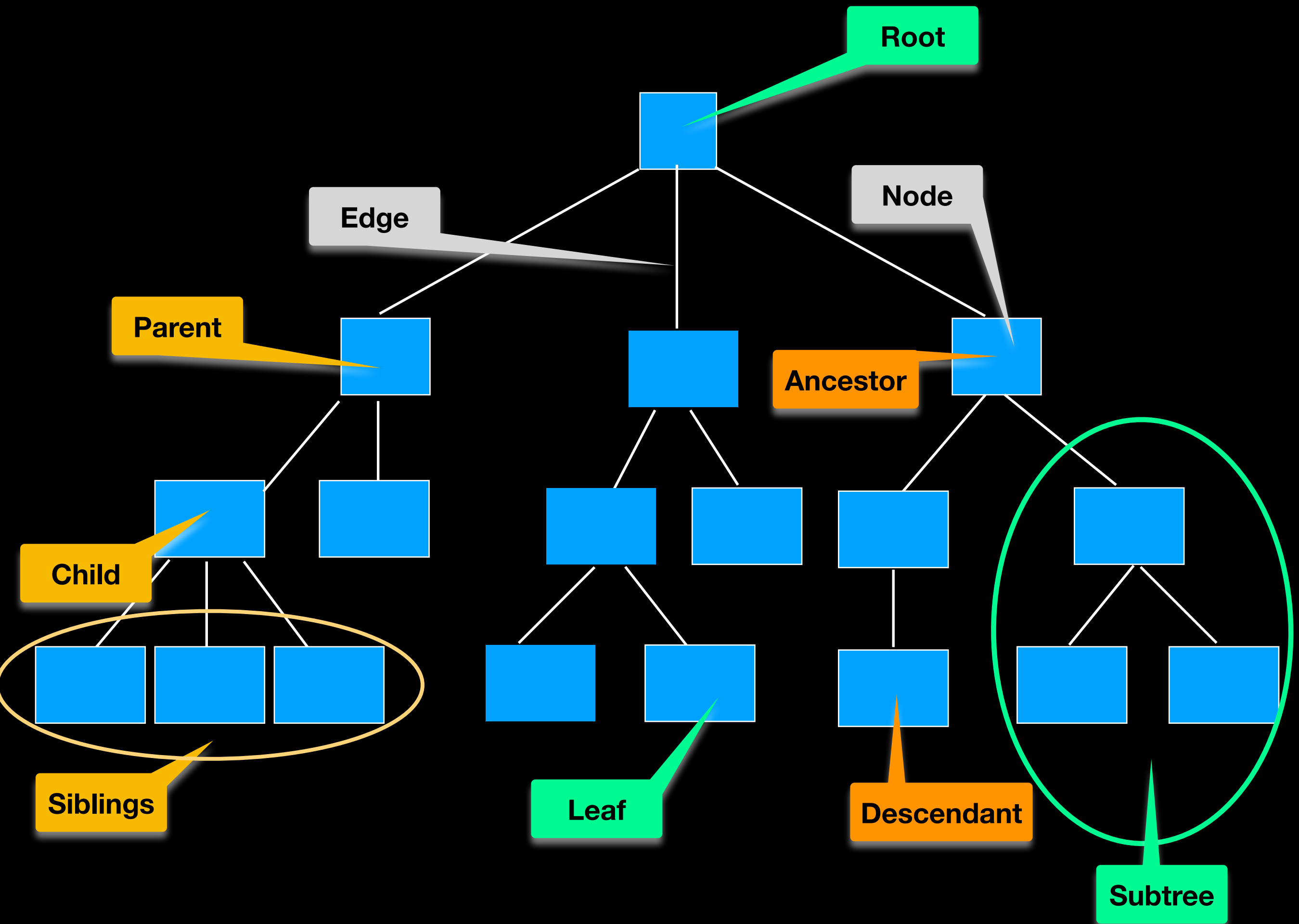


What can you tell me
about this tree?

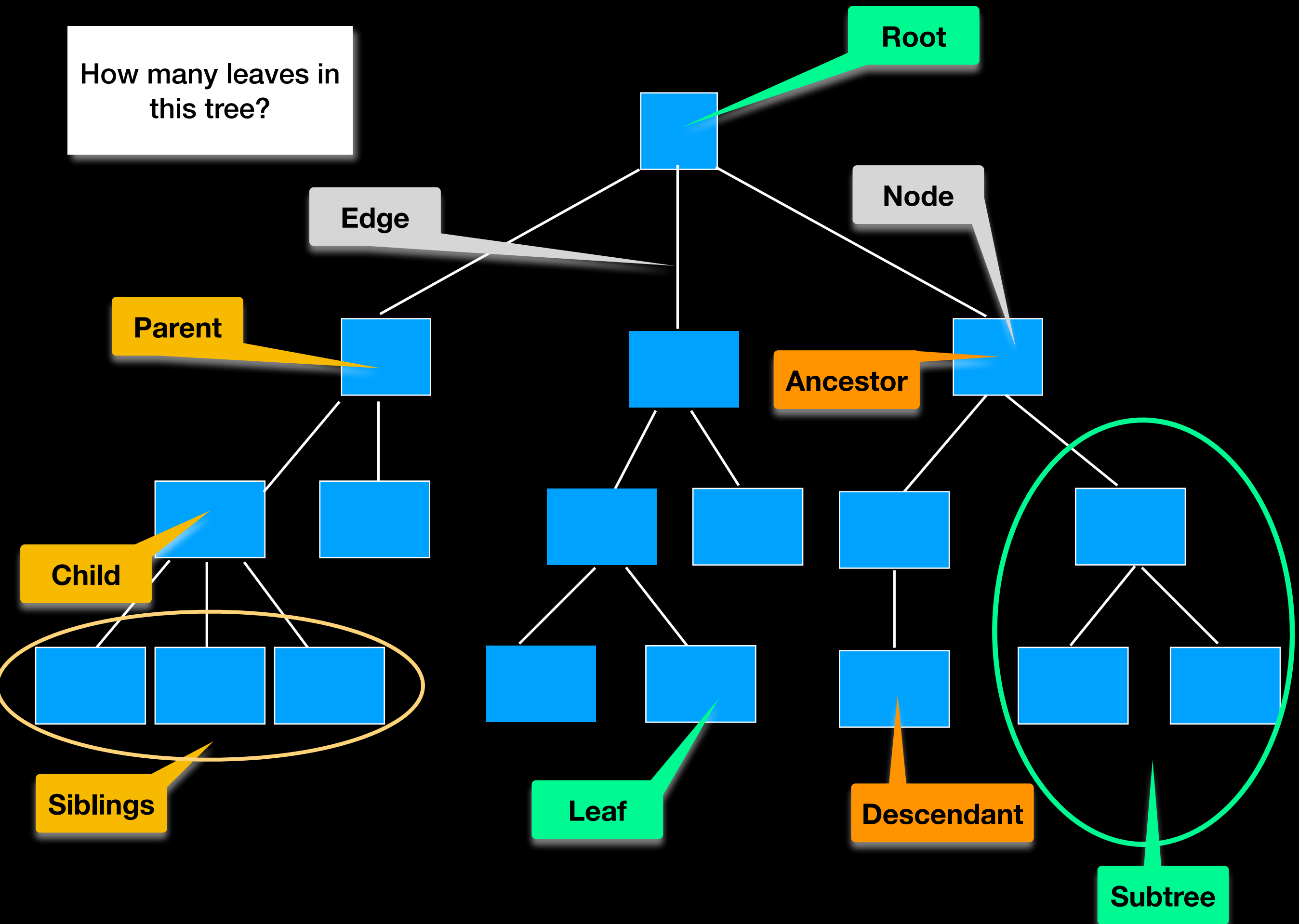


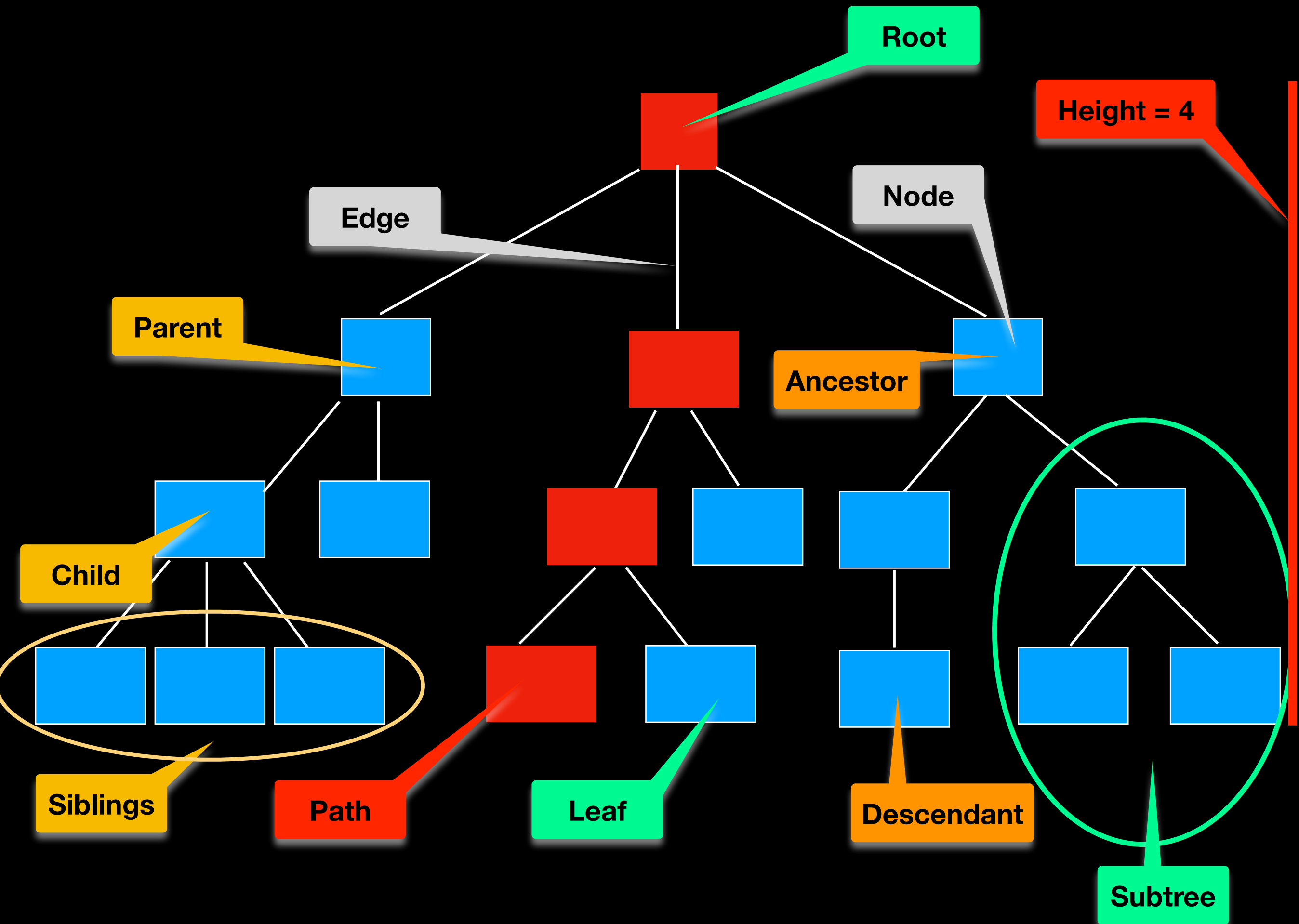






How many leaves in this tree?



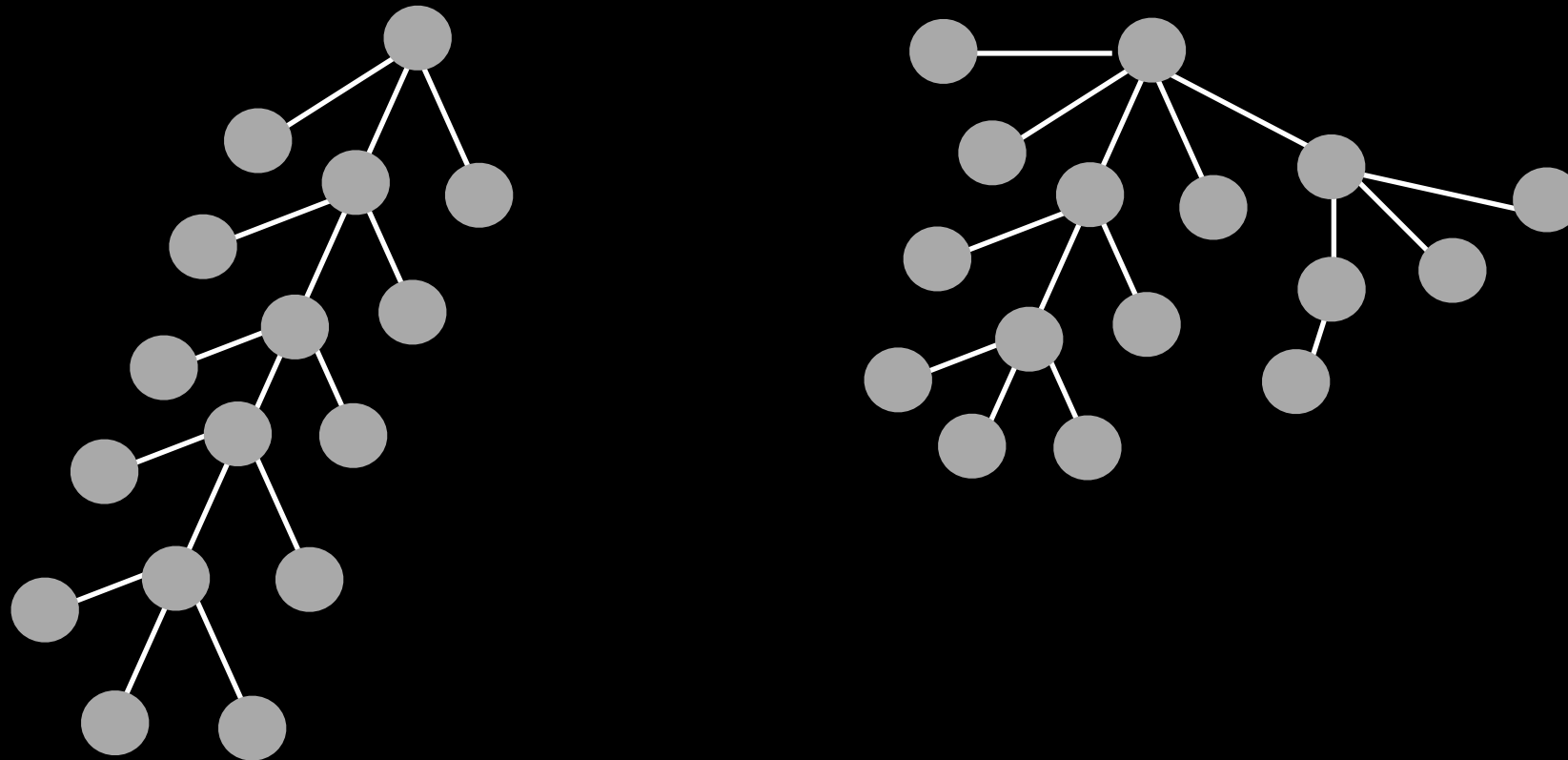


Path: a sequence of nodes c_1, c_2, \dots, c_k where c_{i+1} is a child of c_i .

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

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

Different shapes/structures



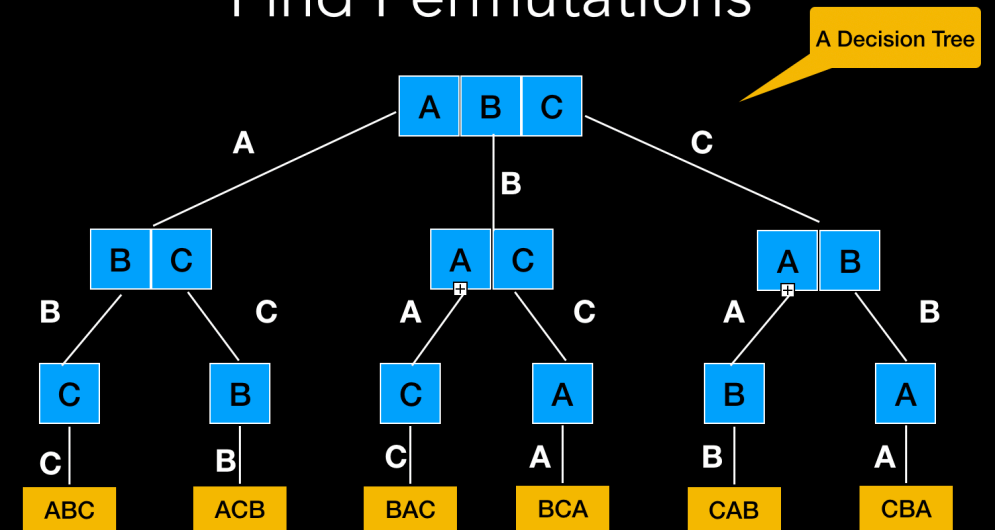
Both $n = 16$
Both 11 leaves
Different height

We have already seen Trees!

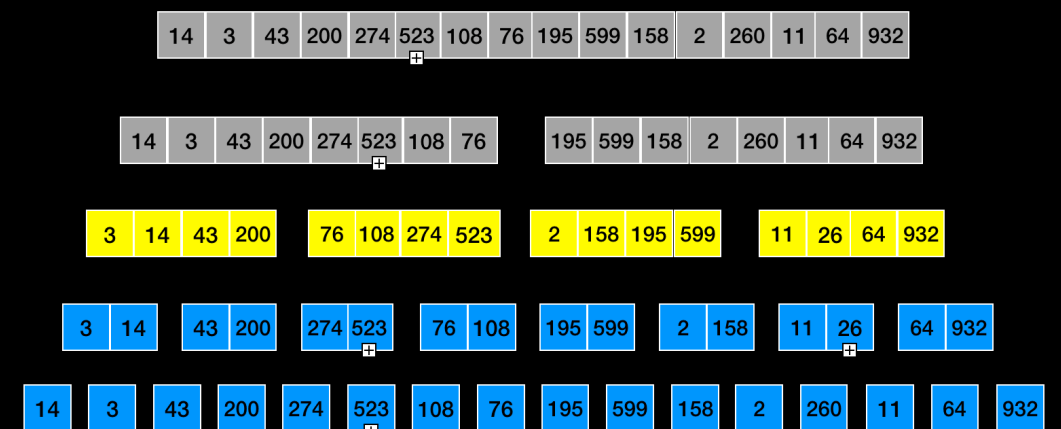
Mostly as a “thinking tool”

- Decision Trees
- Divide and Conquer

Find Permutations



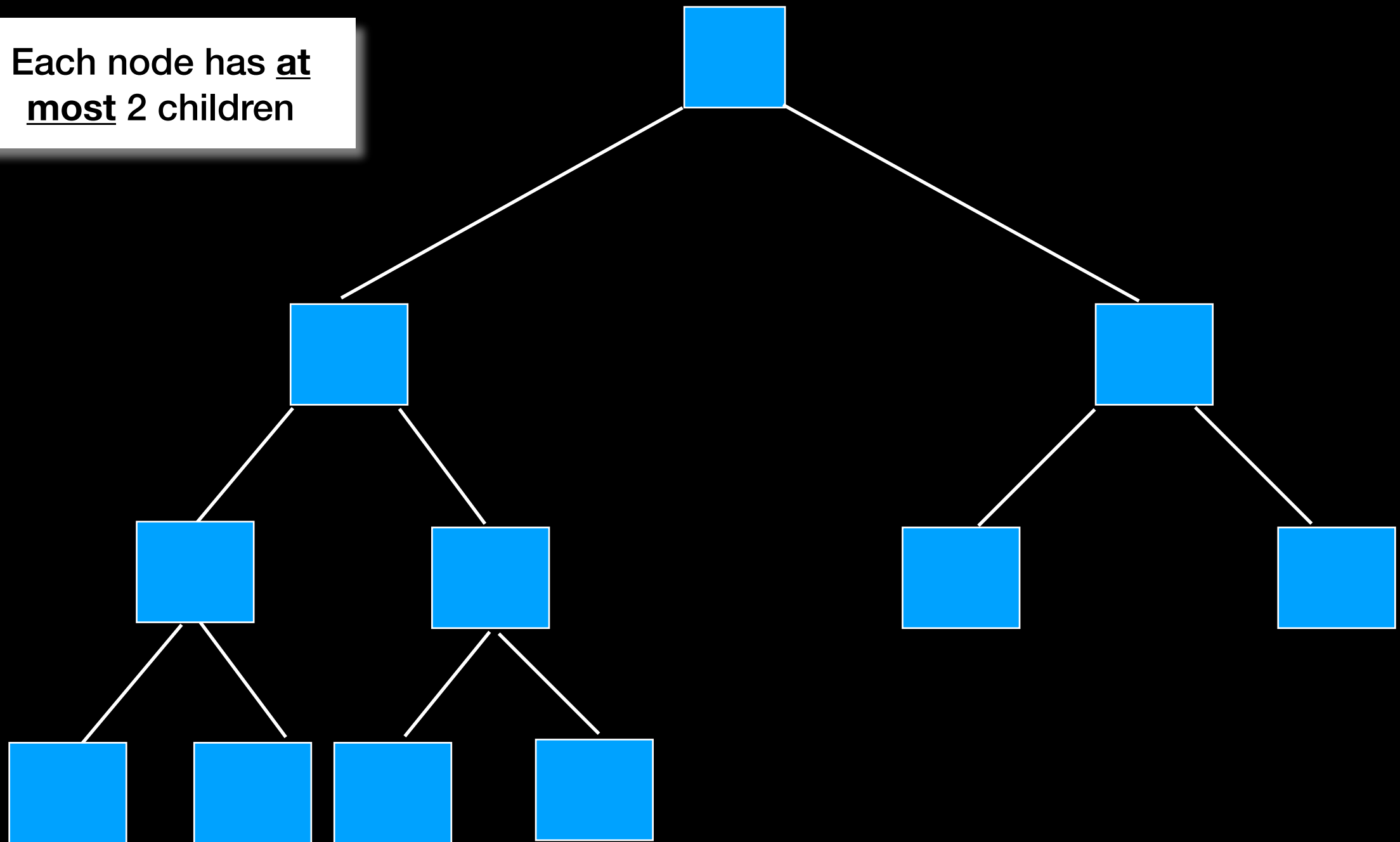
Merge Sort



Binary Tree ADT

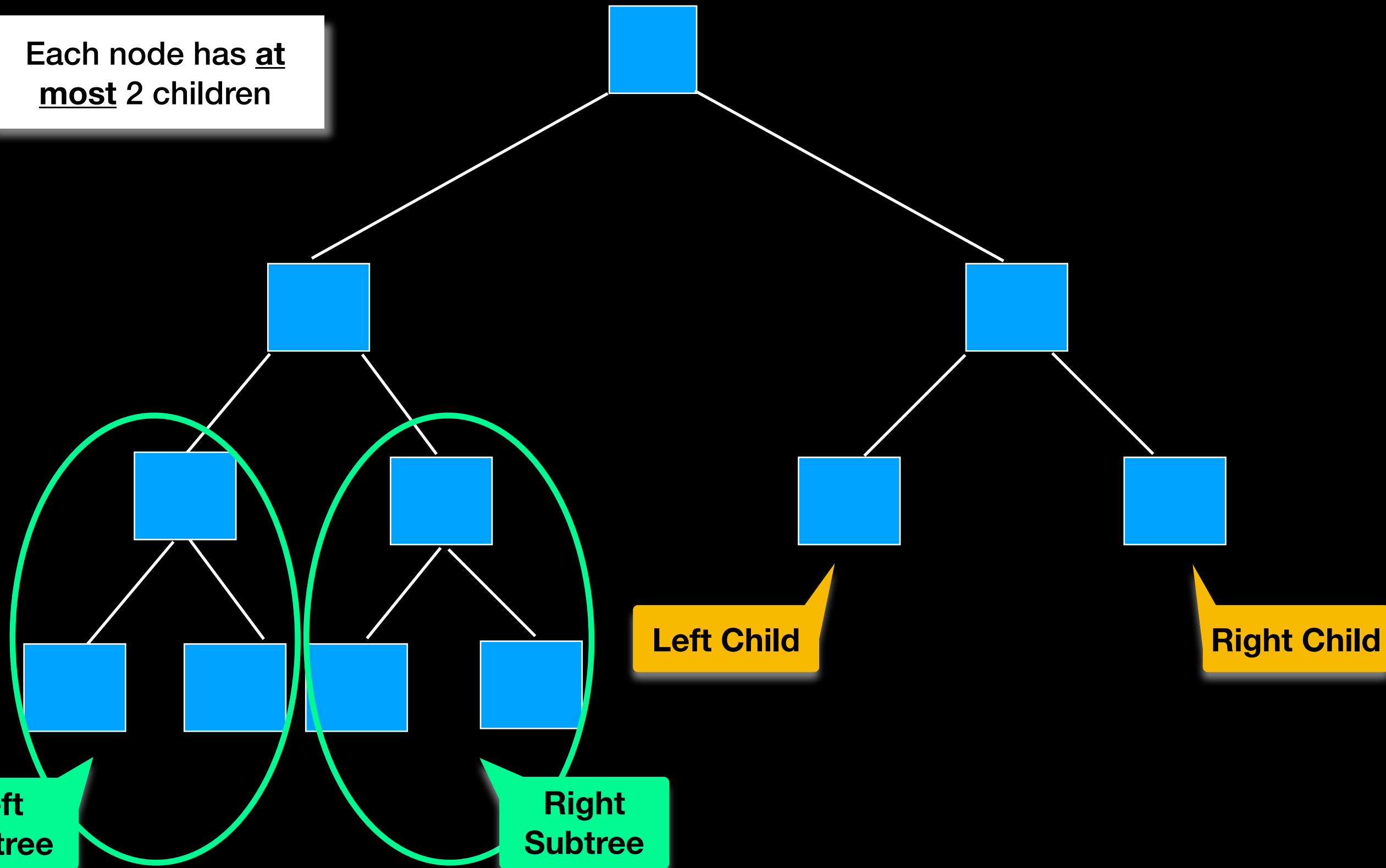
BinaryTree

Each node has at
most 2 children

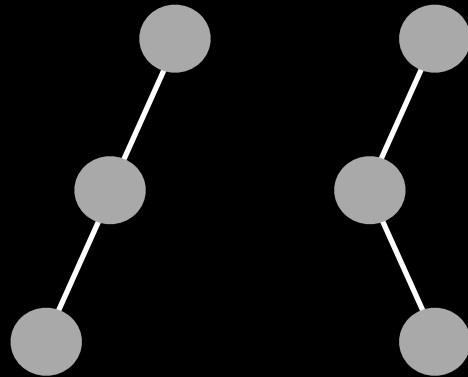


BinaryTree

Each node has at most 2 children



Different shapes/structures

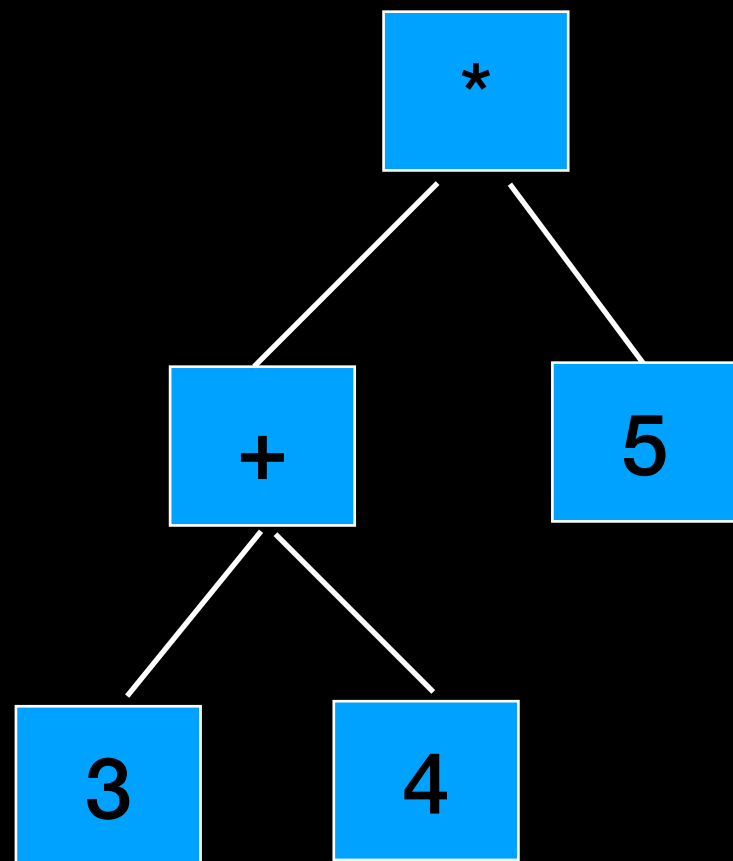


**Both $h = 3$ and one leaf
But different**

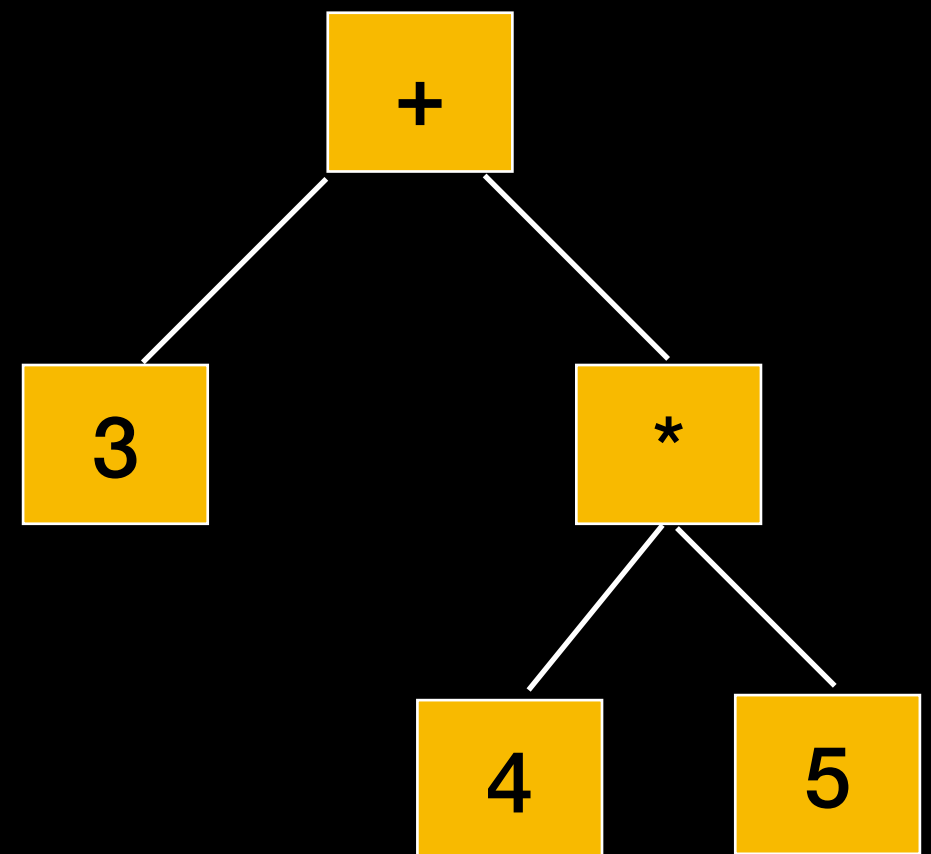
Binary Tree Applications

Algebraic Expressions

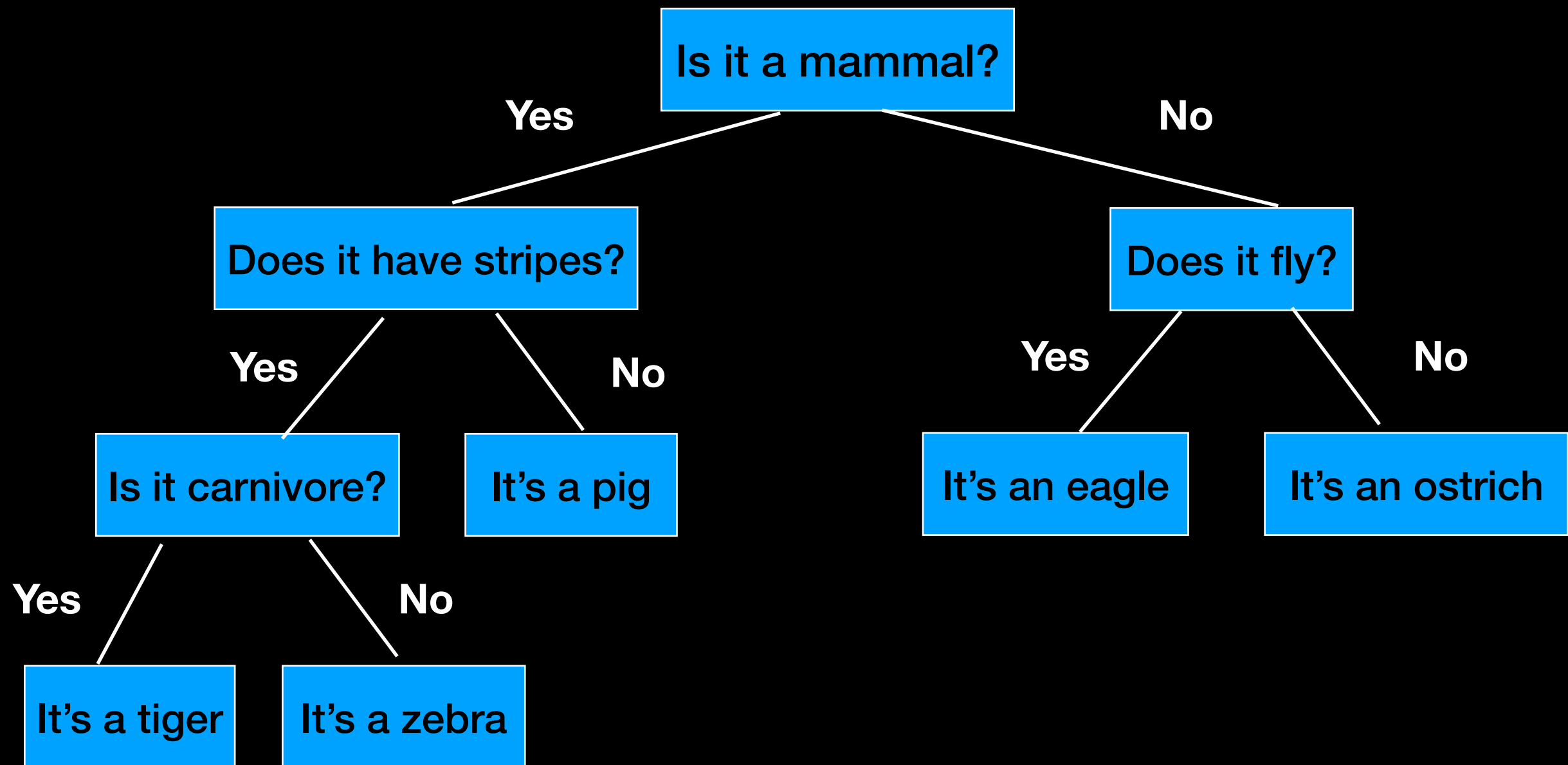
$(3 + 4) * 5$



$3 + 4 * 5$



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**

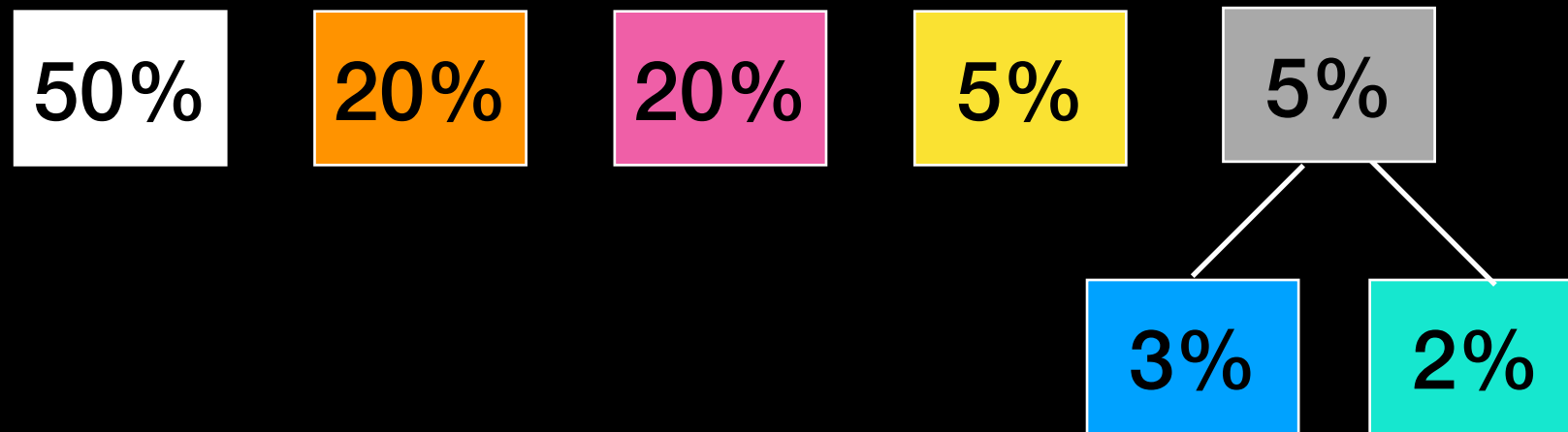
Not encryption but **compression** => use shortest code for most frequent symbols

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

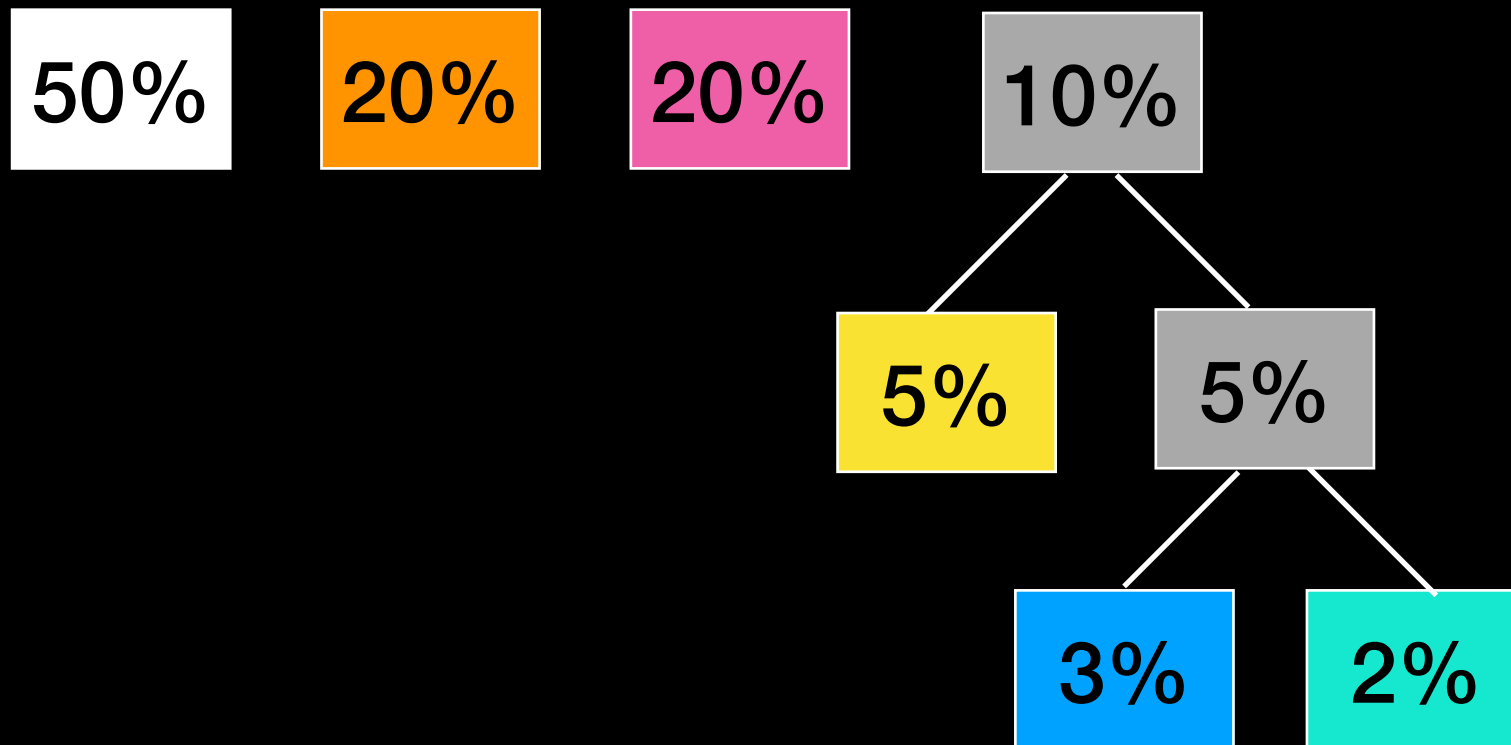
Huffman Tree



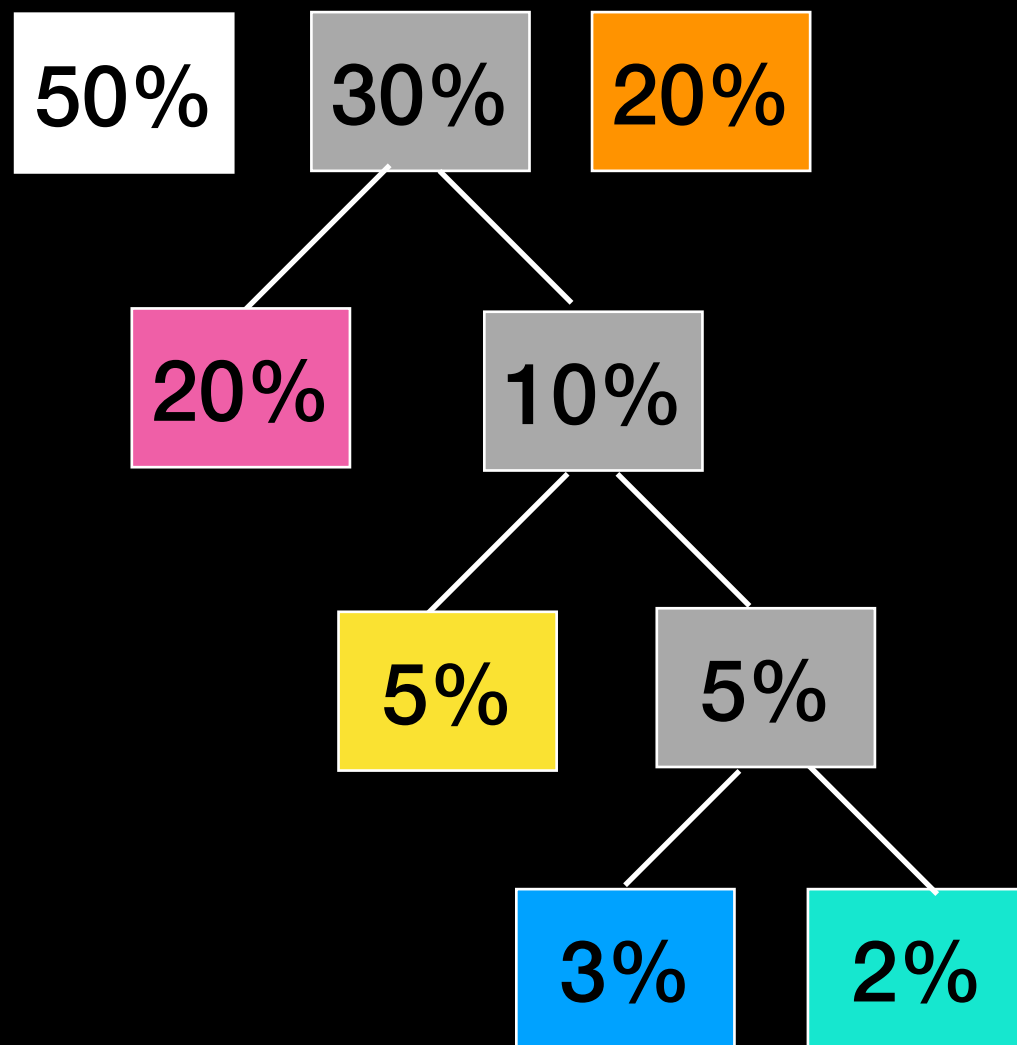
Huffman Tree



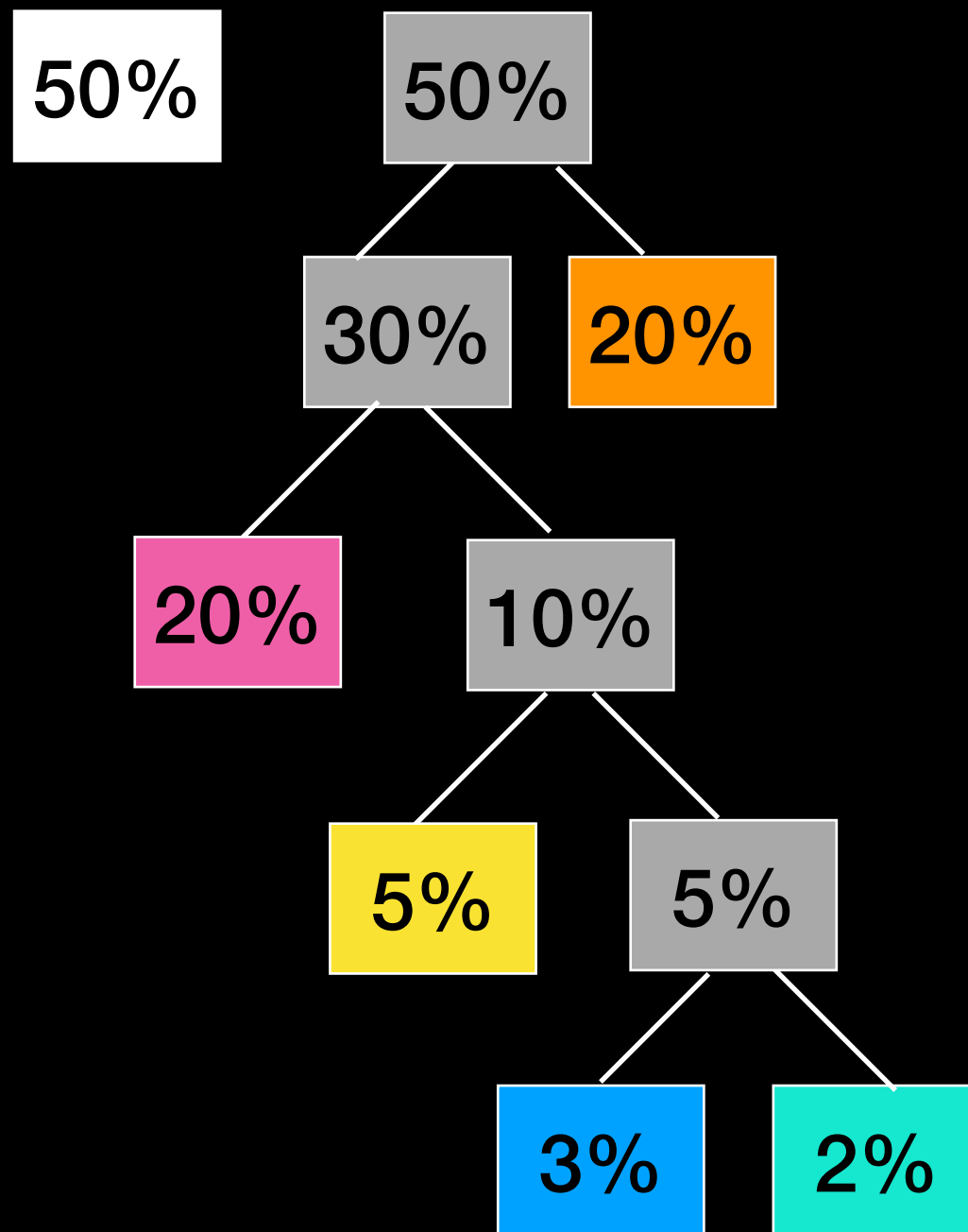
Huffman Tree



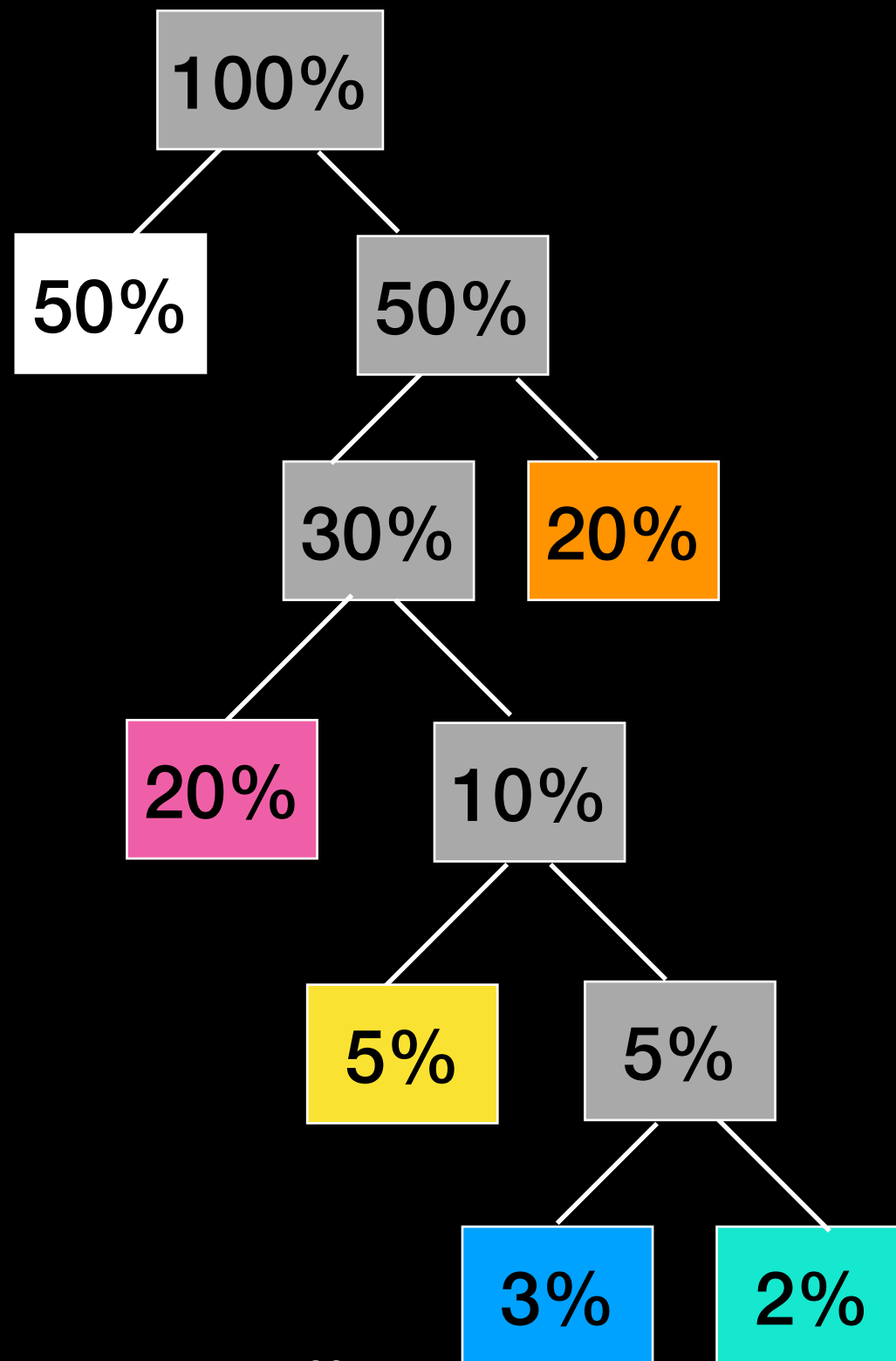
Huffman Tree



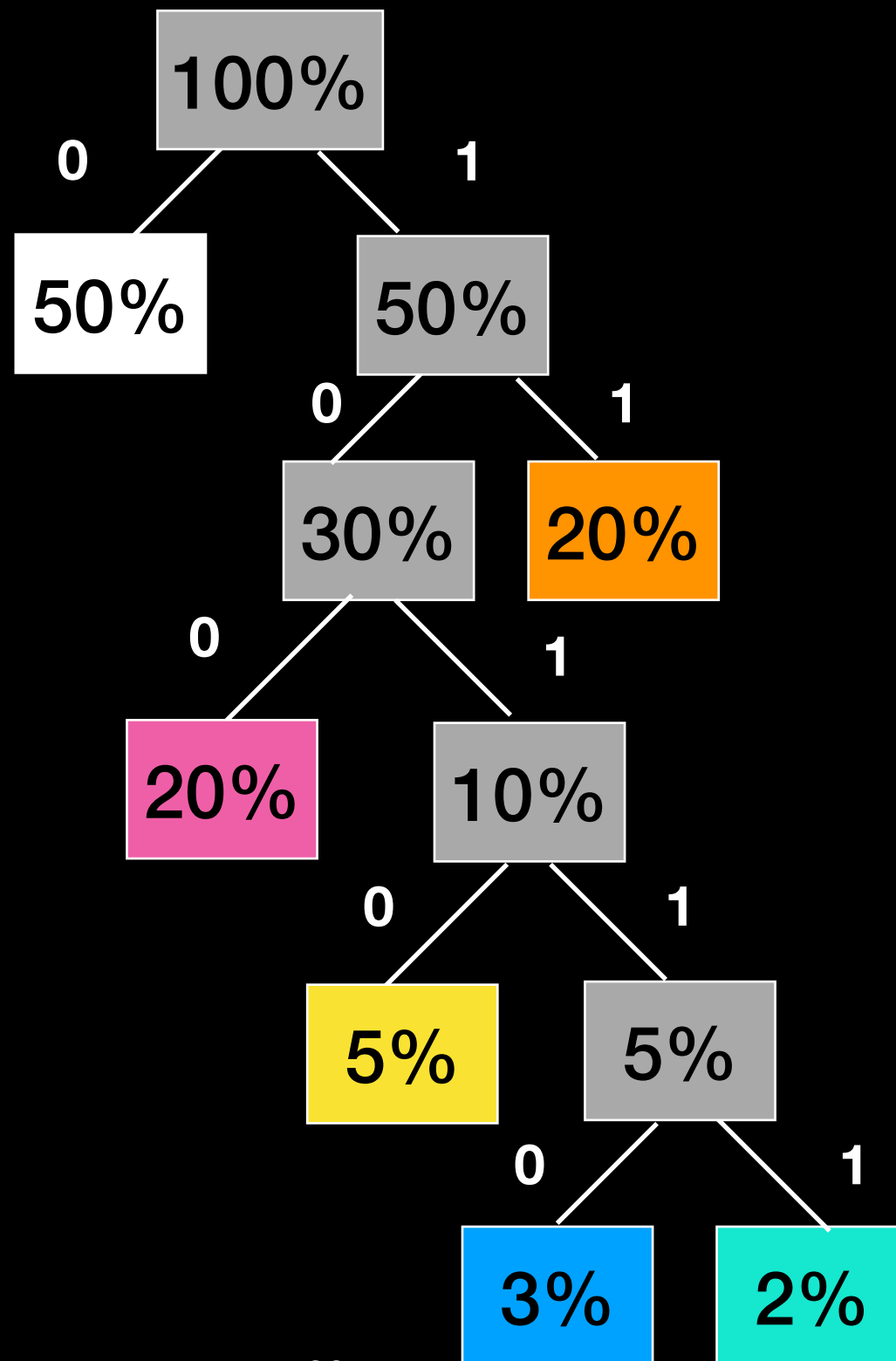
Huffman Tree



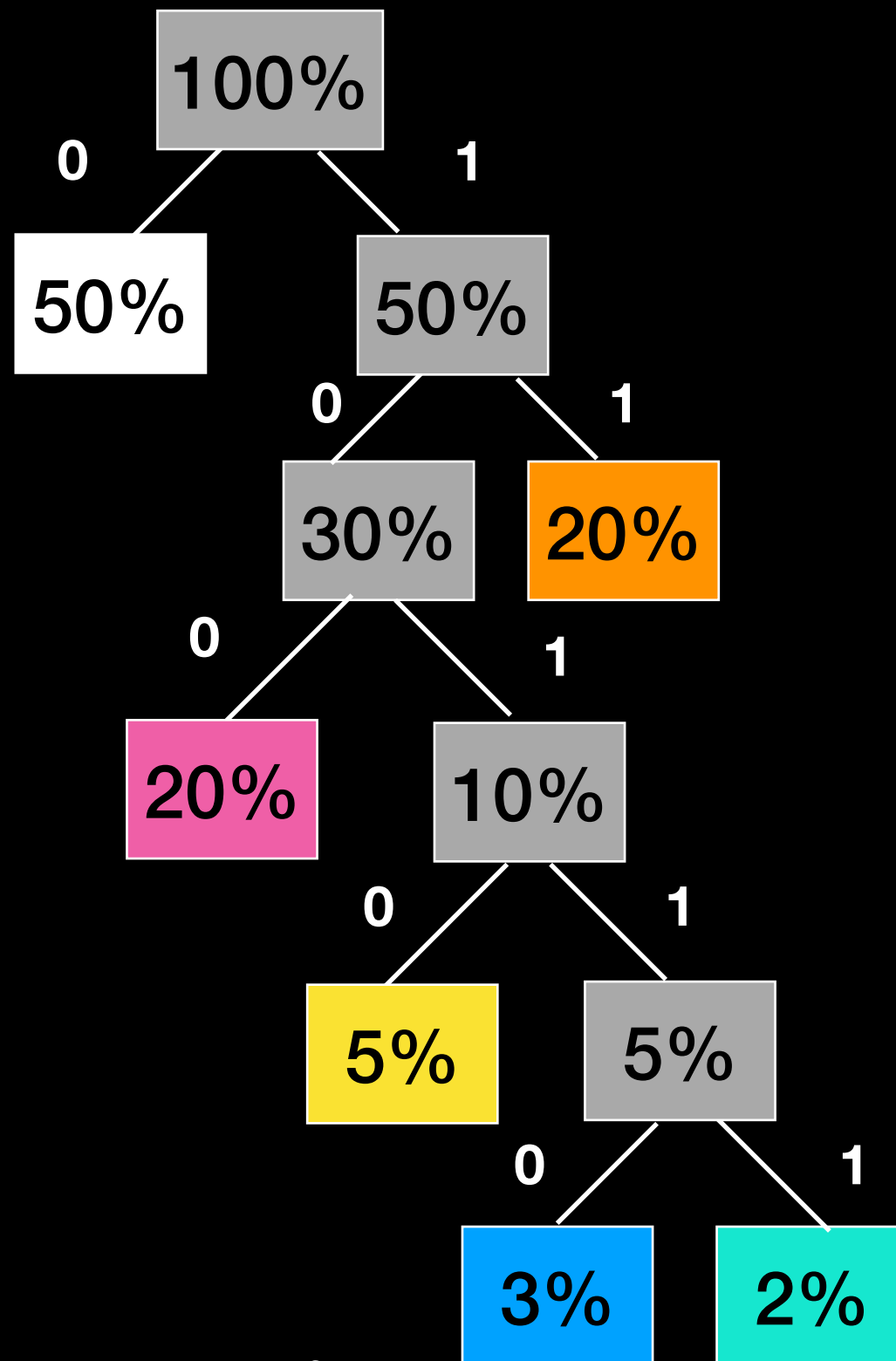
Huffman Tree



Huffman Tree



Huffman Tree



0
11
100
1010
10110
10111

Lecture Activity

Think about structure!

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

Label each tree with its height and number of leaves.

Binary Trees - 4 Vertices

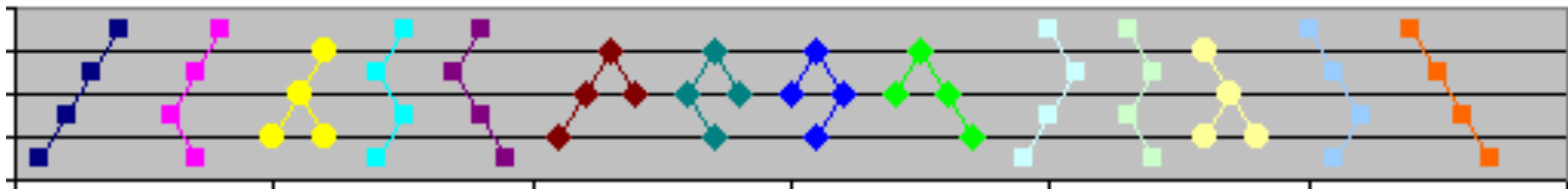
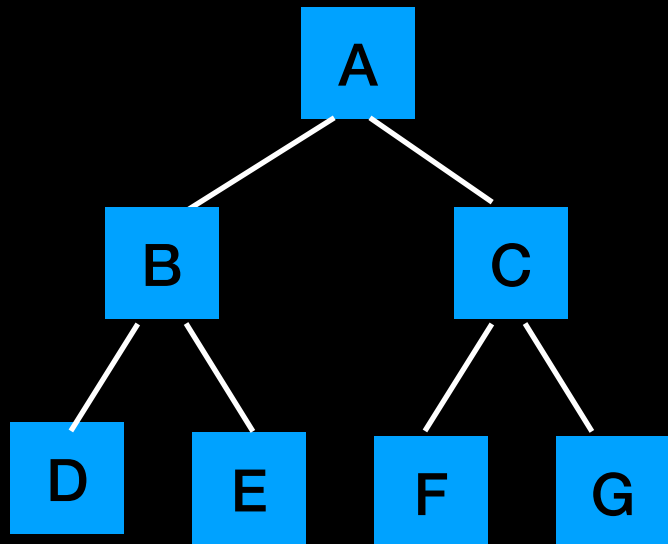


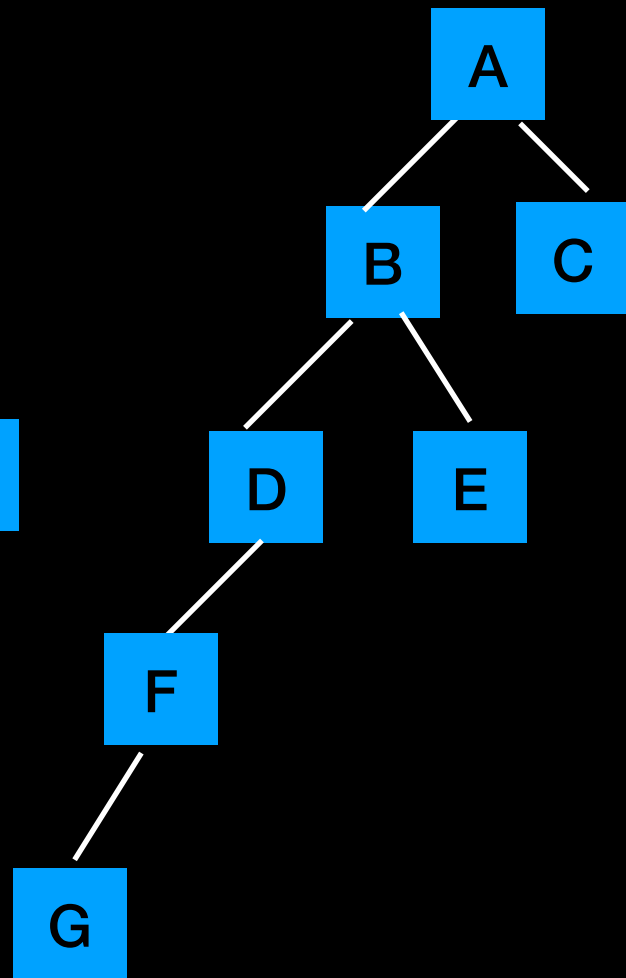
IMAGE FROM : <http://www.durangobill.com/BinTrees.html>

Tree Structure

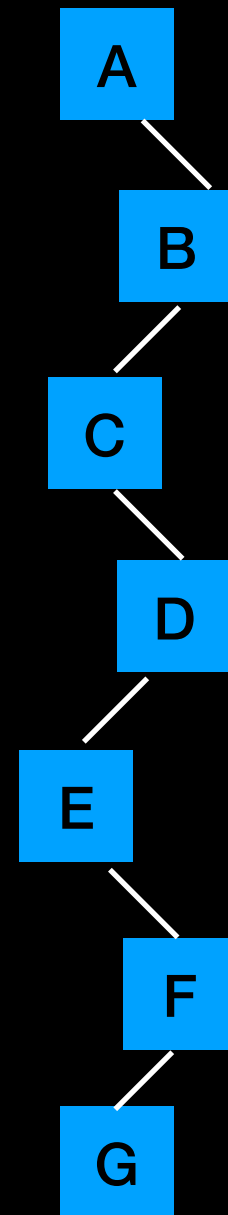
$h = 3$



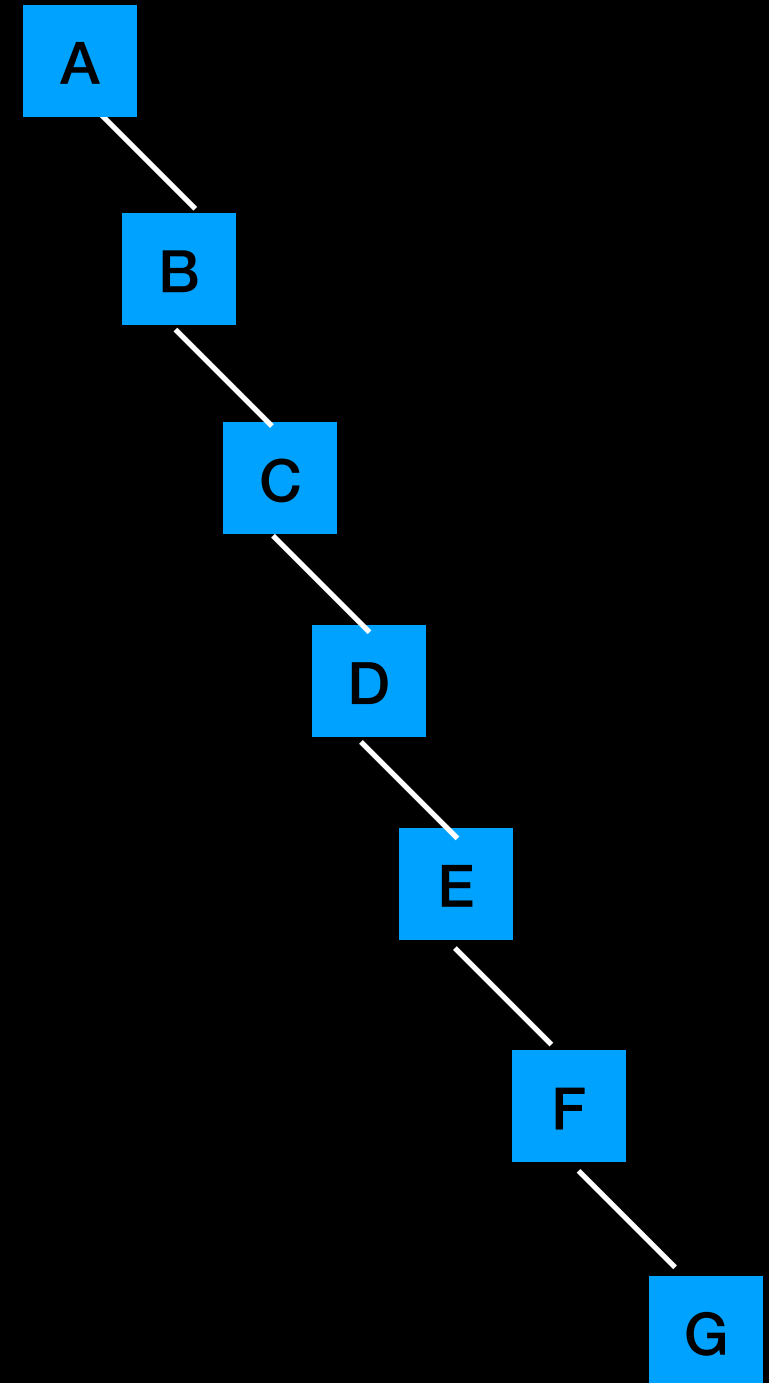
$h = 5$



$h = 7$



$h = 7$

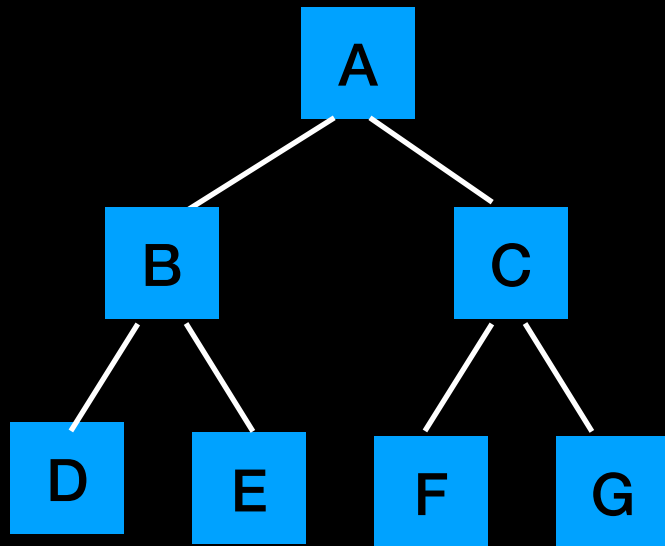


Structure definitions may vary across different sources.

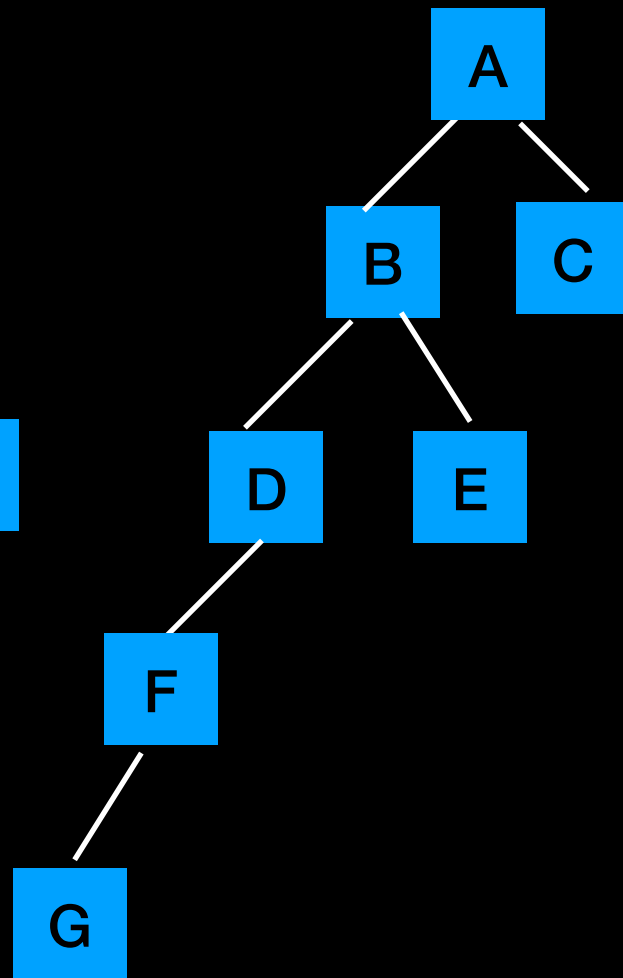
The following comes from your textbook and will be used in this course and on exams

Tree Structure

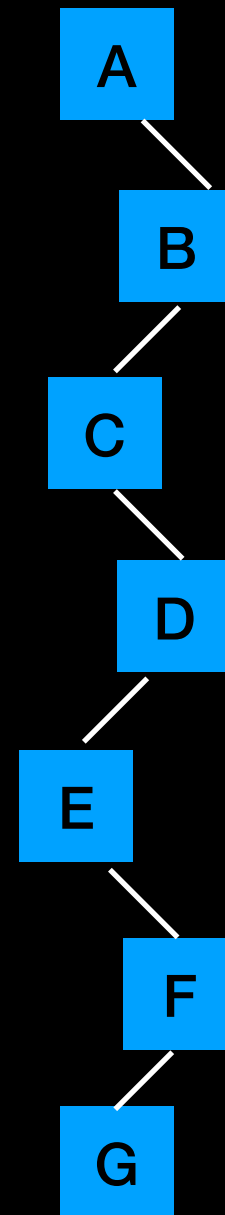
$h = 3$



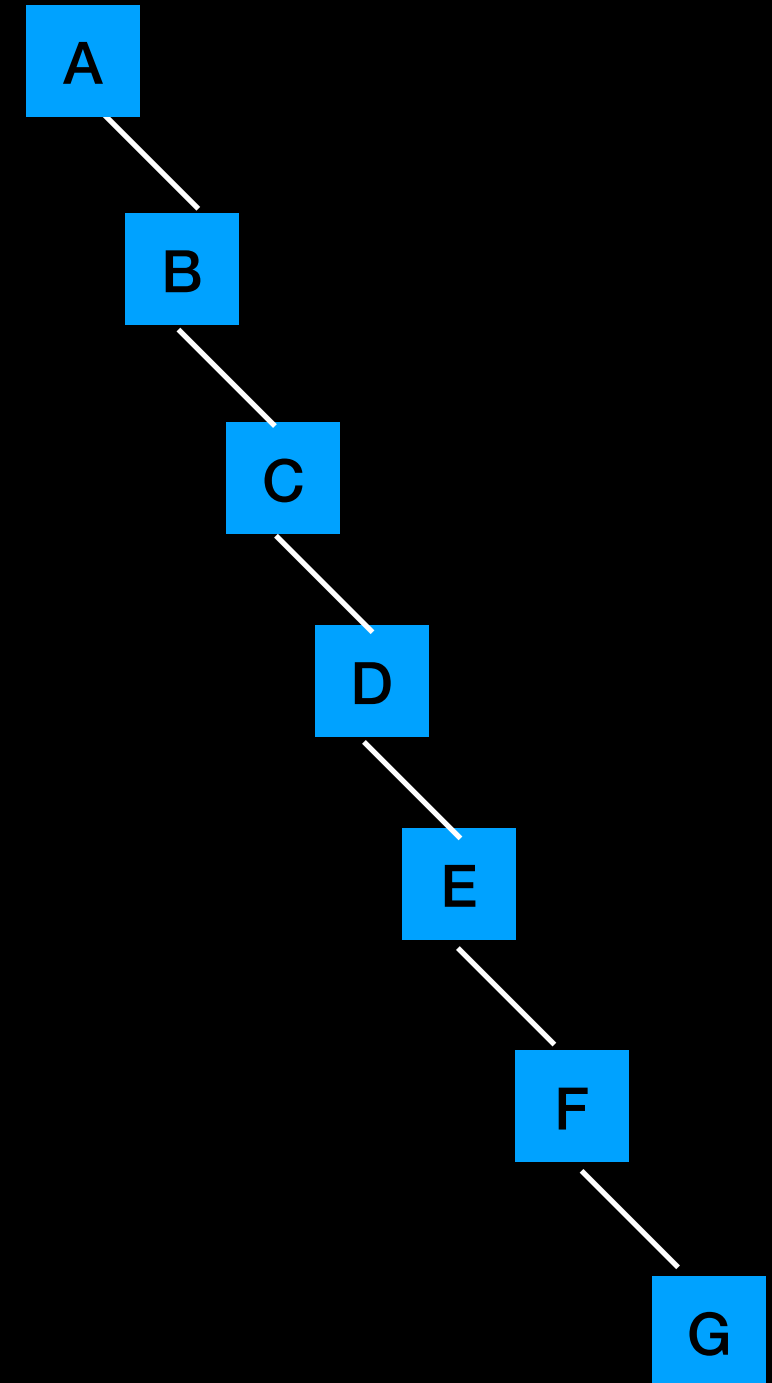
$h = 5$



$h = 7$



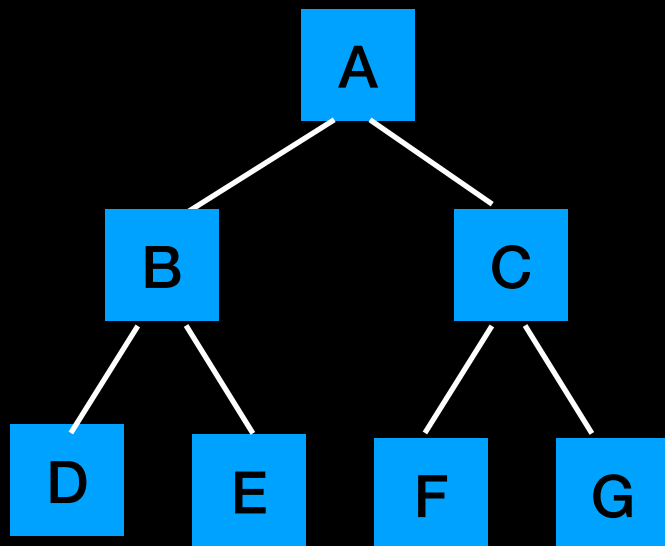
$h = 7$



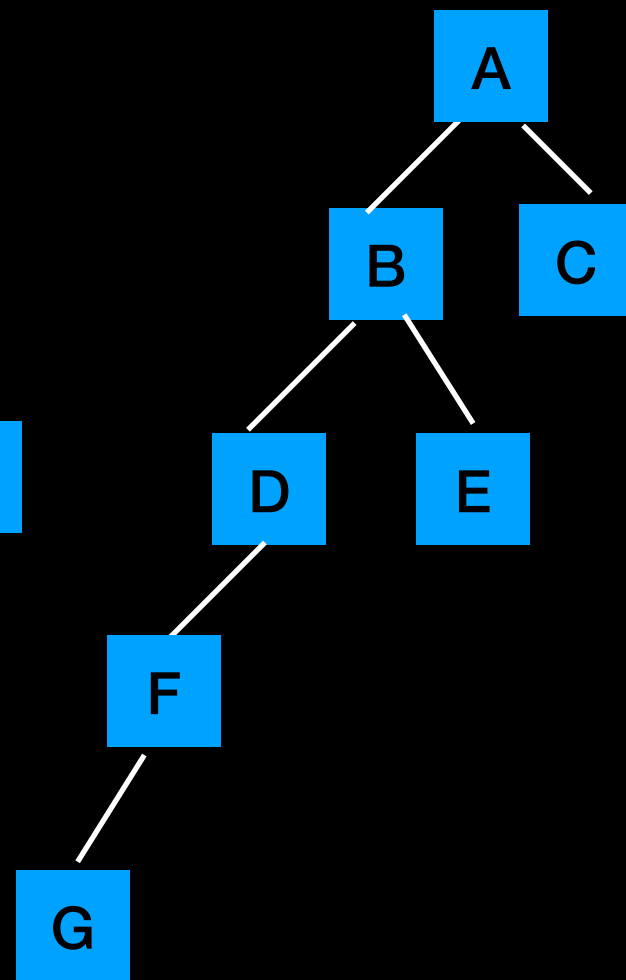
What is the maximum (minimum) height of a tree with 7 nodes?

Tree Structure

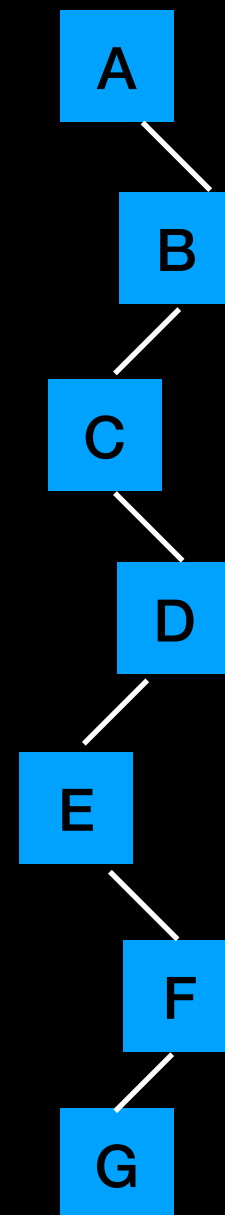
$h = 3$



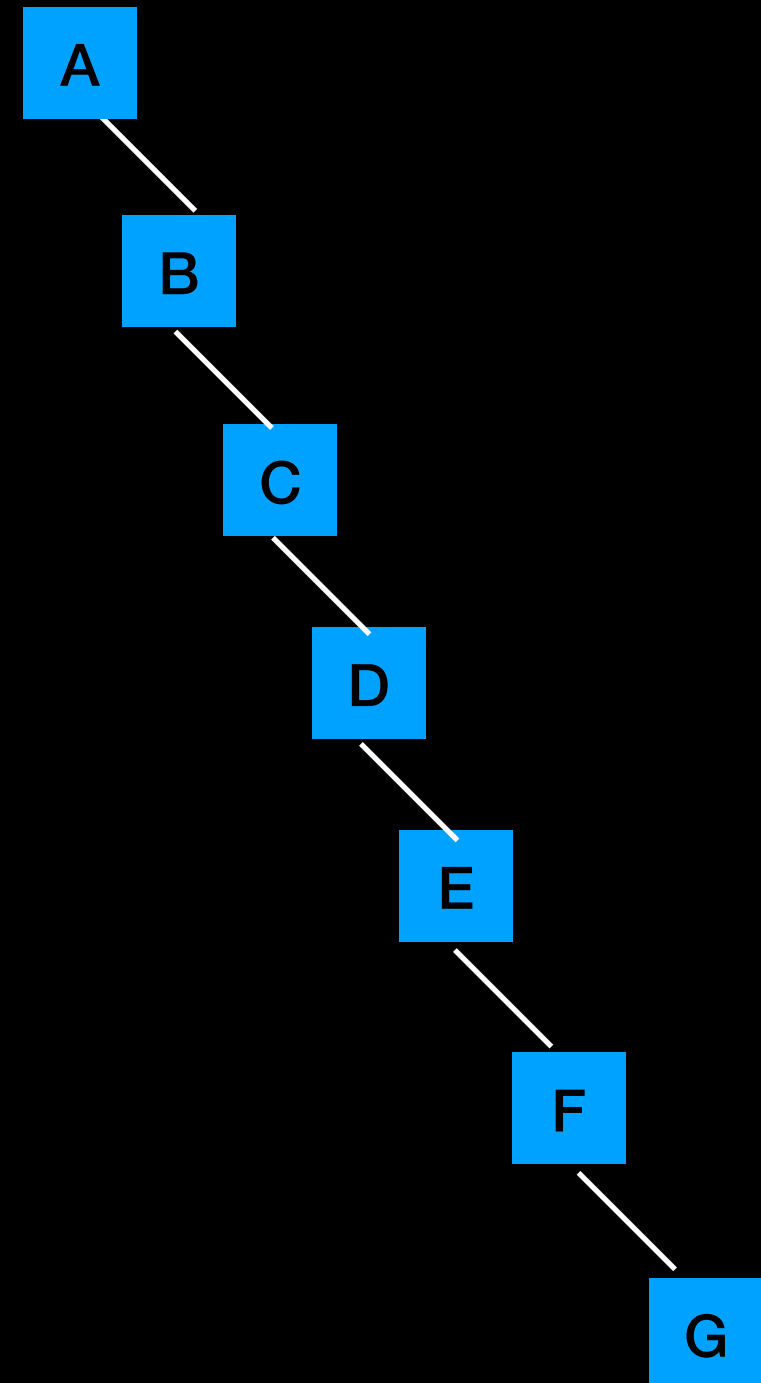
$h = 5$



$h = 7$



$h = 7$



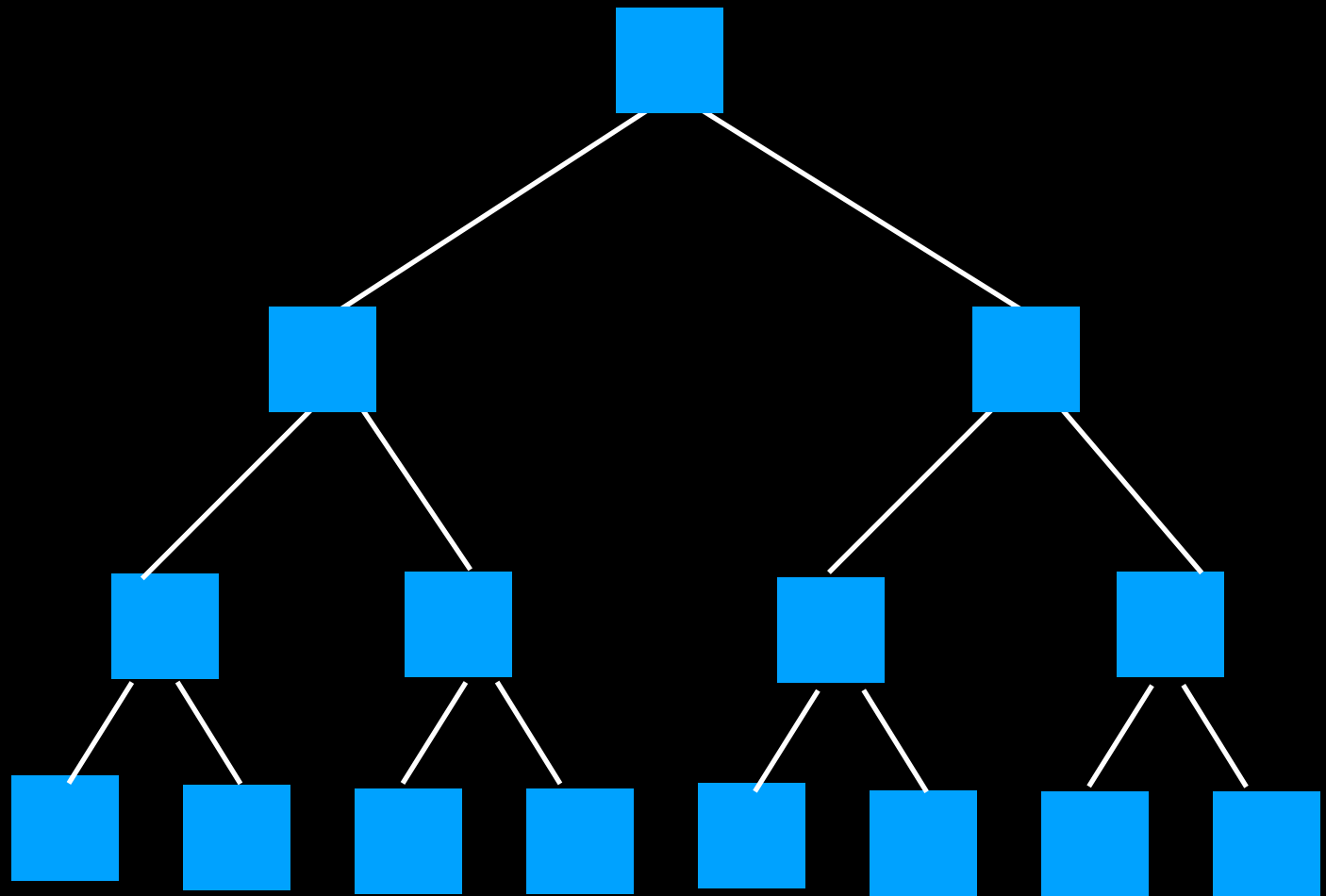
**WE WILL LOOK AT THE
GENERAL ANSWER NEXT**

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**



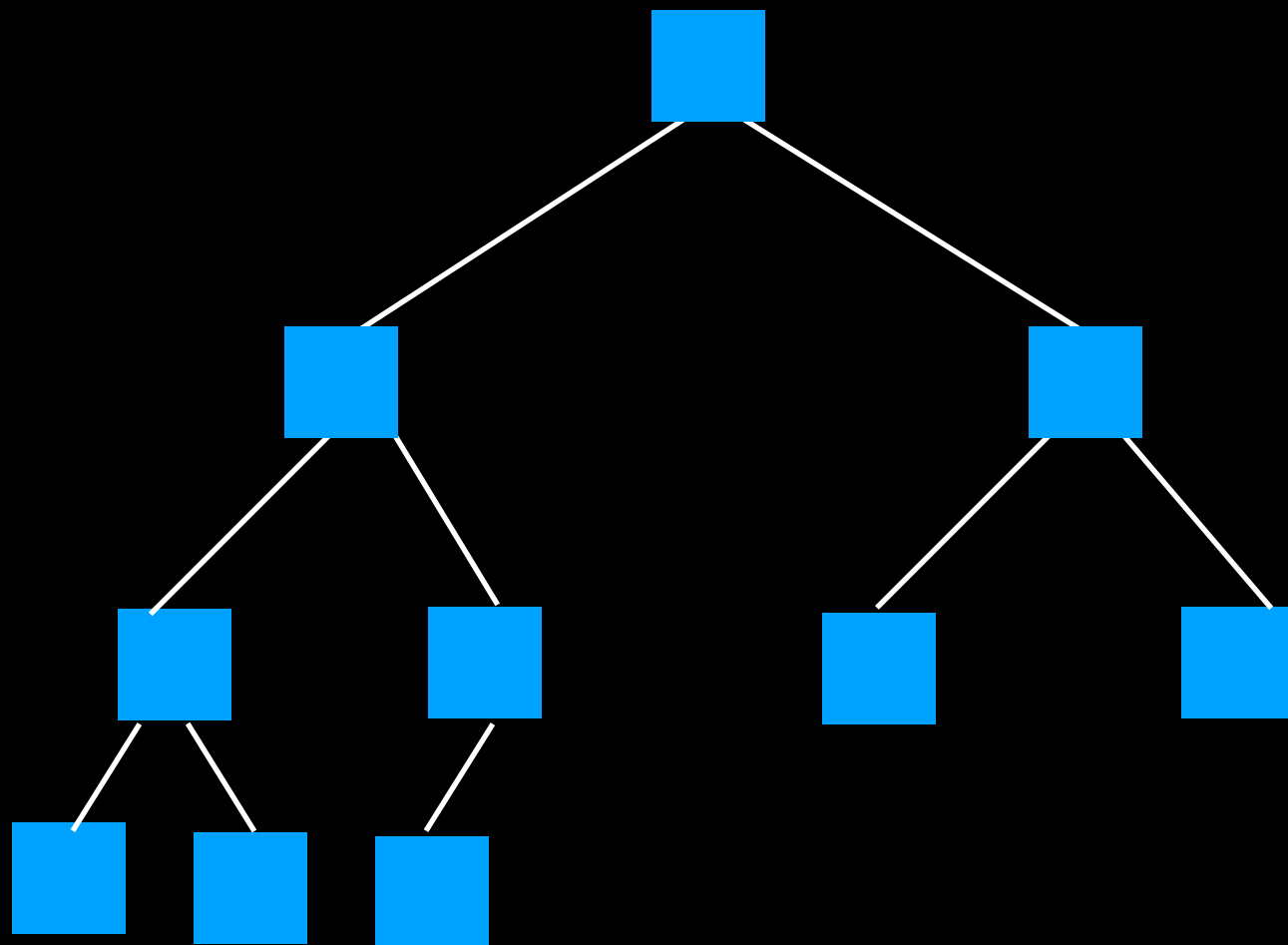
Complete Binary Tree

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

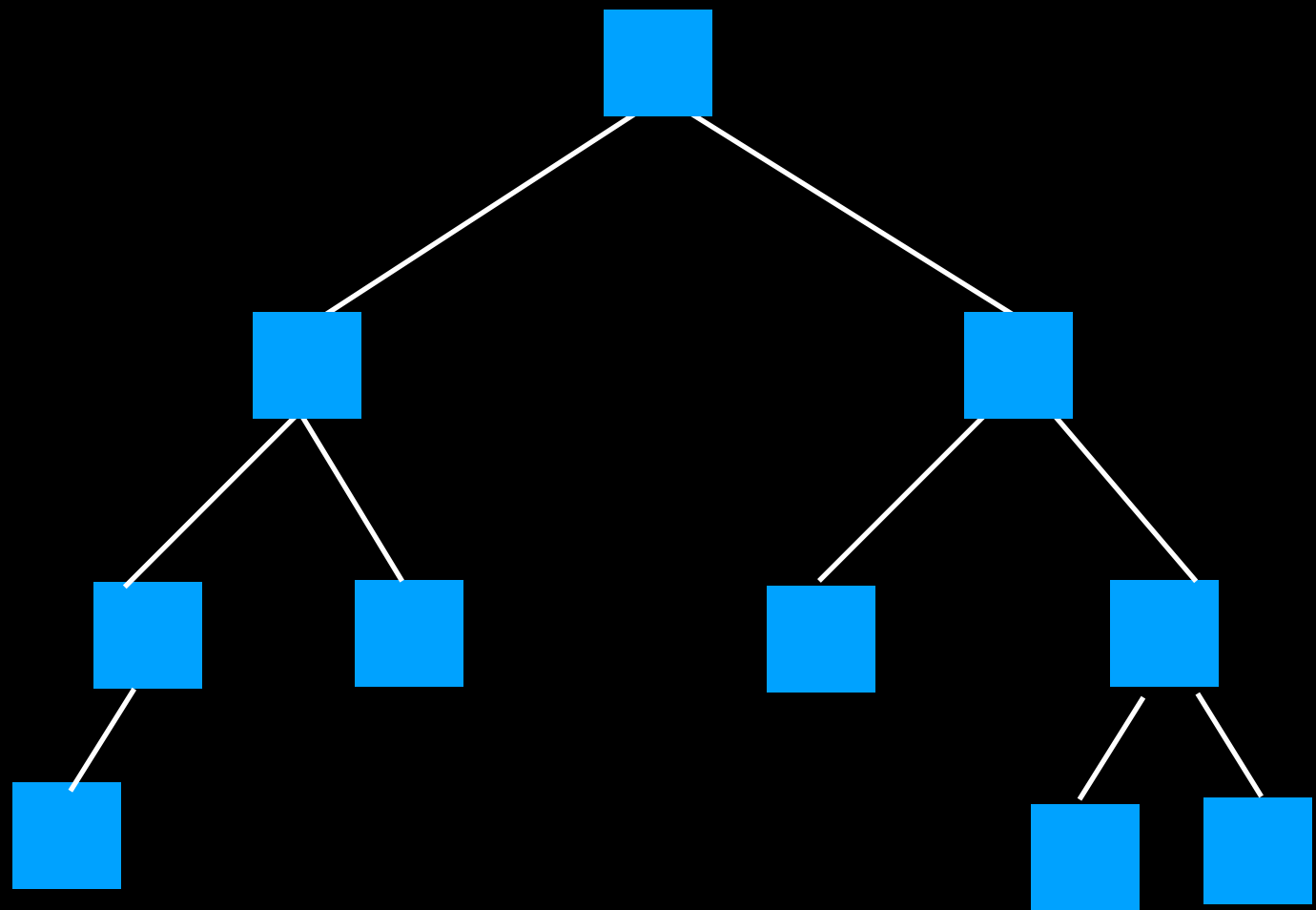
When a node at level $h-1$ has one child, it is a left child



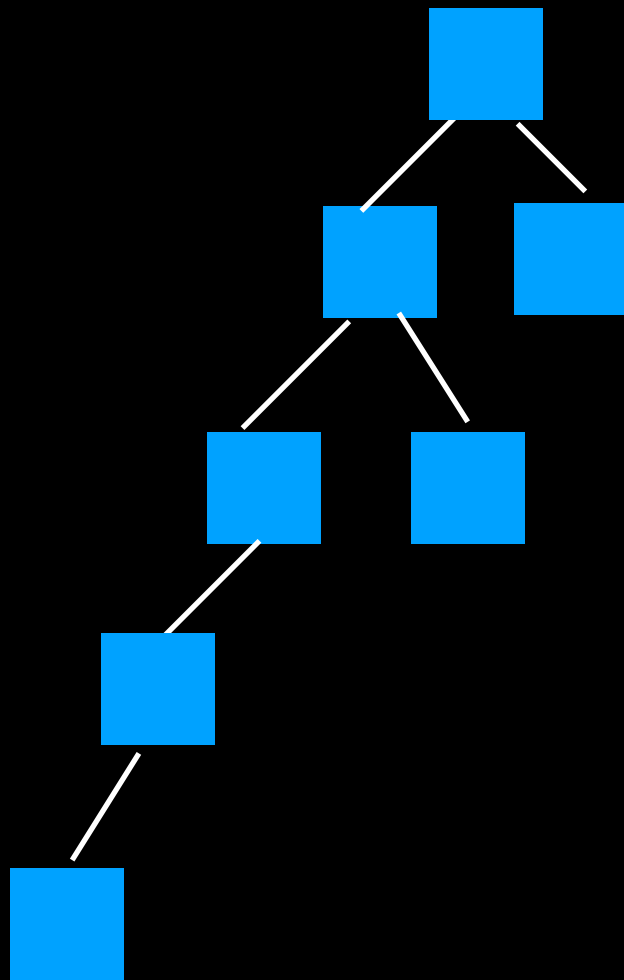
(Height) Balanced Binary Tree

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

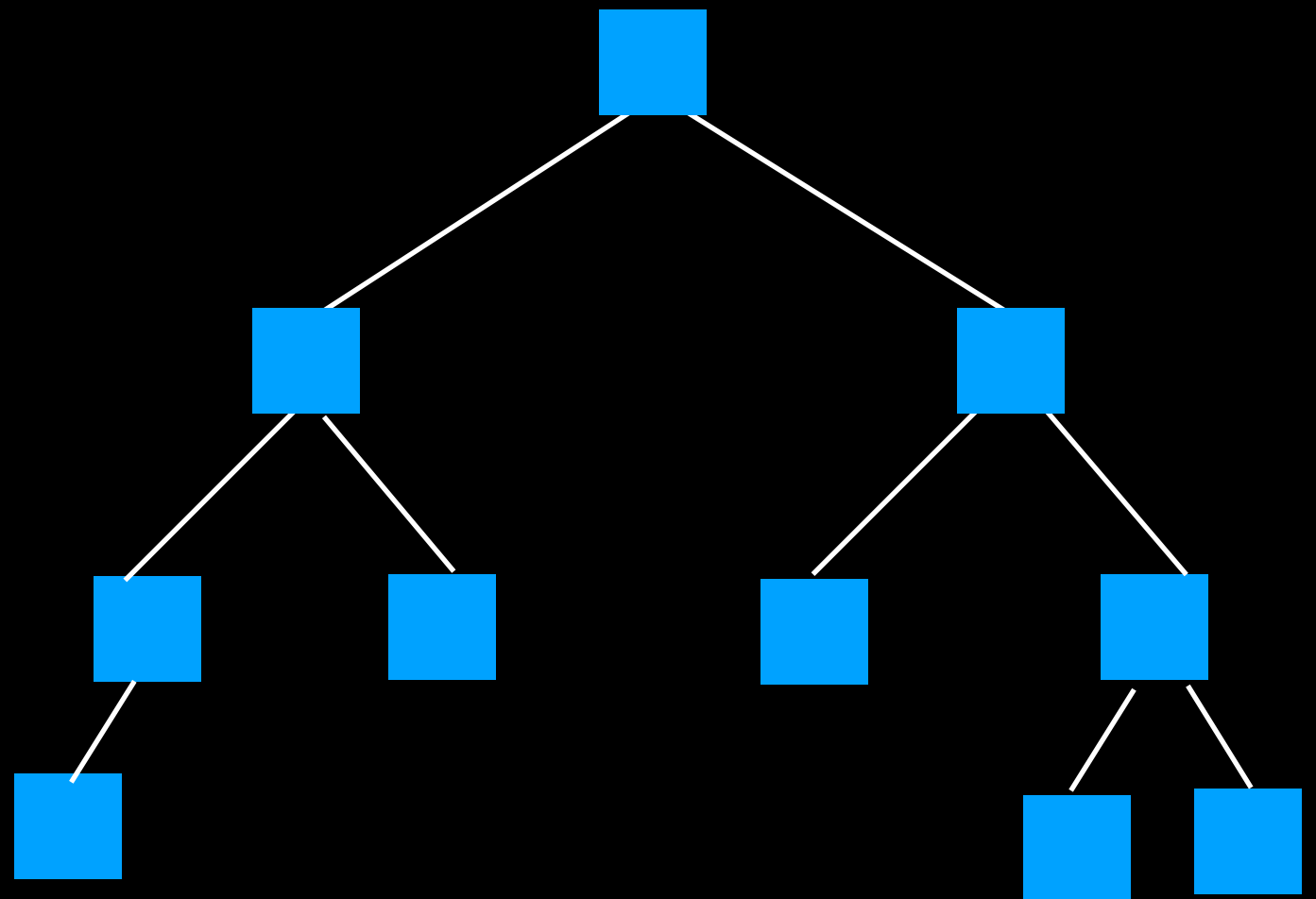
All paths from root to leaf differ in length by at most 1



Unbalanced



Balanced



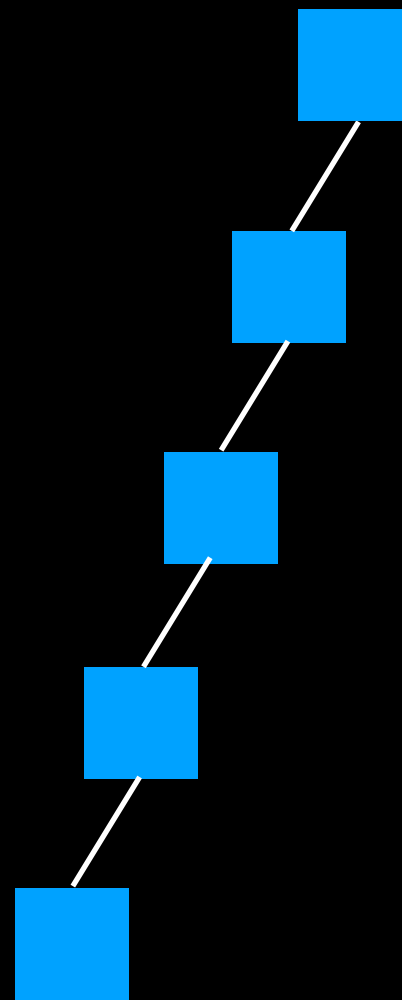
Maximum Height

n nodes

every node 1 child

$h = n$

Essentially a chain



Minimum Height

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

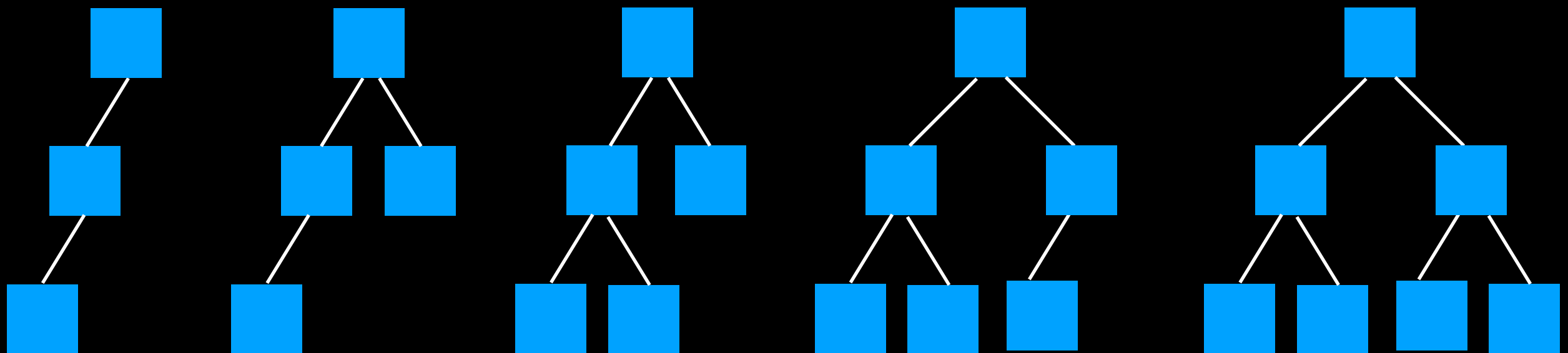
For example for $h = 3$, $1 + 2 + 4 = 7 = 2^3 - 1$

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

For example:

1,000 nodes $h \approx 10$ ($1,000 \approx 2^{10}$)

1,000,000 nodes $h \approx 20$ ($10^6 \approx 2^{20}$)



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**

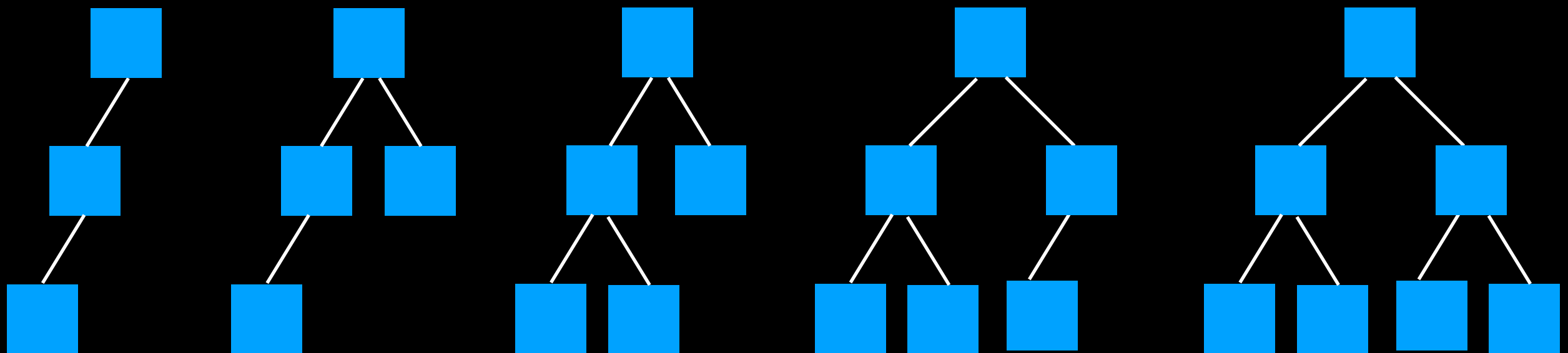
For example:

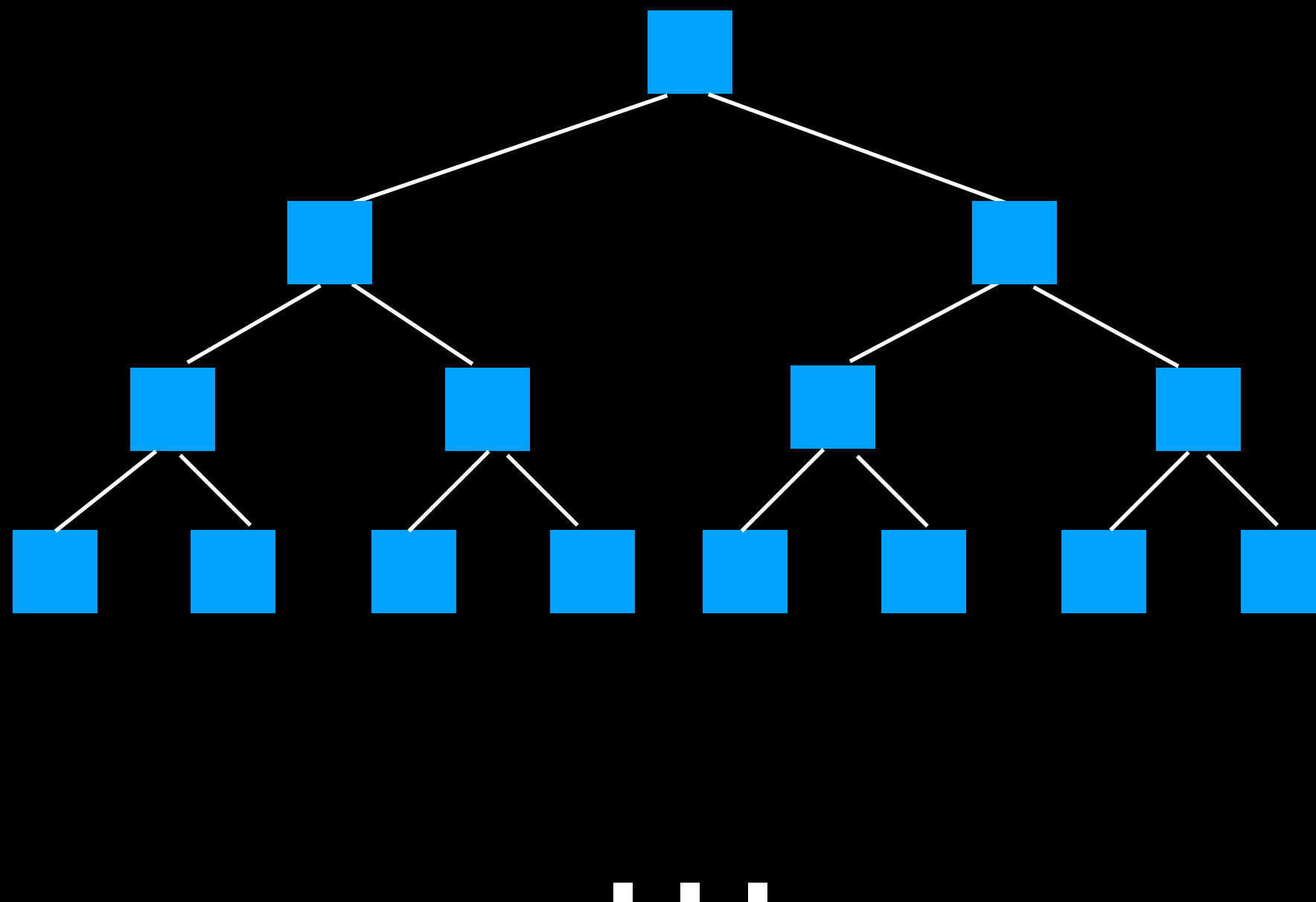
1,000 nodes $h \approx 10$ ($1,000 \approx 2^{10}$)

1,000,000 nodes $h \approx 20$ ($10^6 \approx 2^{20}$)

Recall analysis of
Divide and Conquer
algorithms

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





In a full tree:

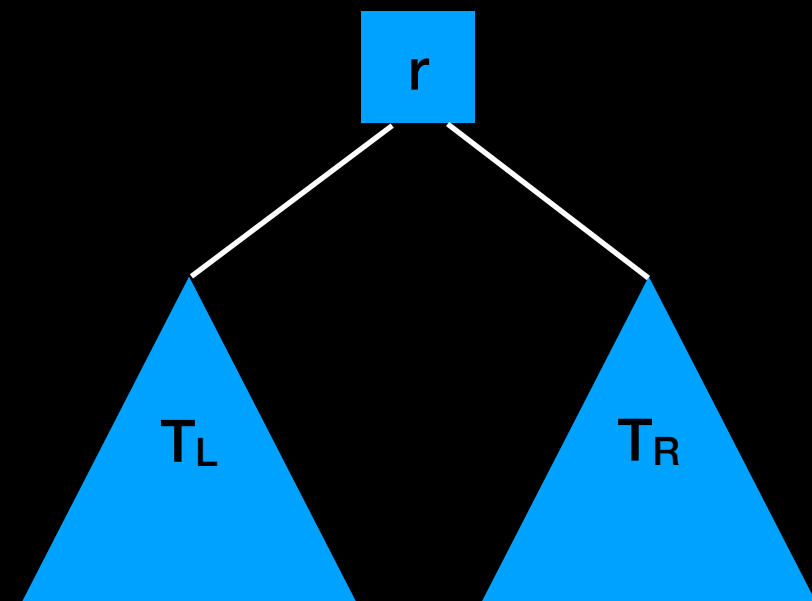
h	n @ level	Total n
1	$1 = 2^0$	$1 = 2^1 - 1$
2	$2 = 2^1$	$3 = 2^2 - 1$
3	$4 = 2^2$	$7 = 2^3 - 1$
4	$8 = 2^3$	$15 = 2^4 - 1$
h	2^{h-1}	$2^h - 1$

Binary Tree Traversals

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

Essentially visit the root as well as it's subtrees

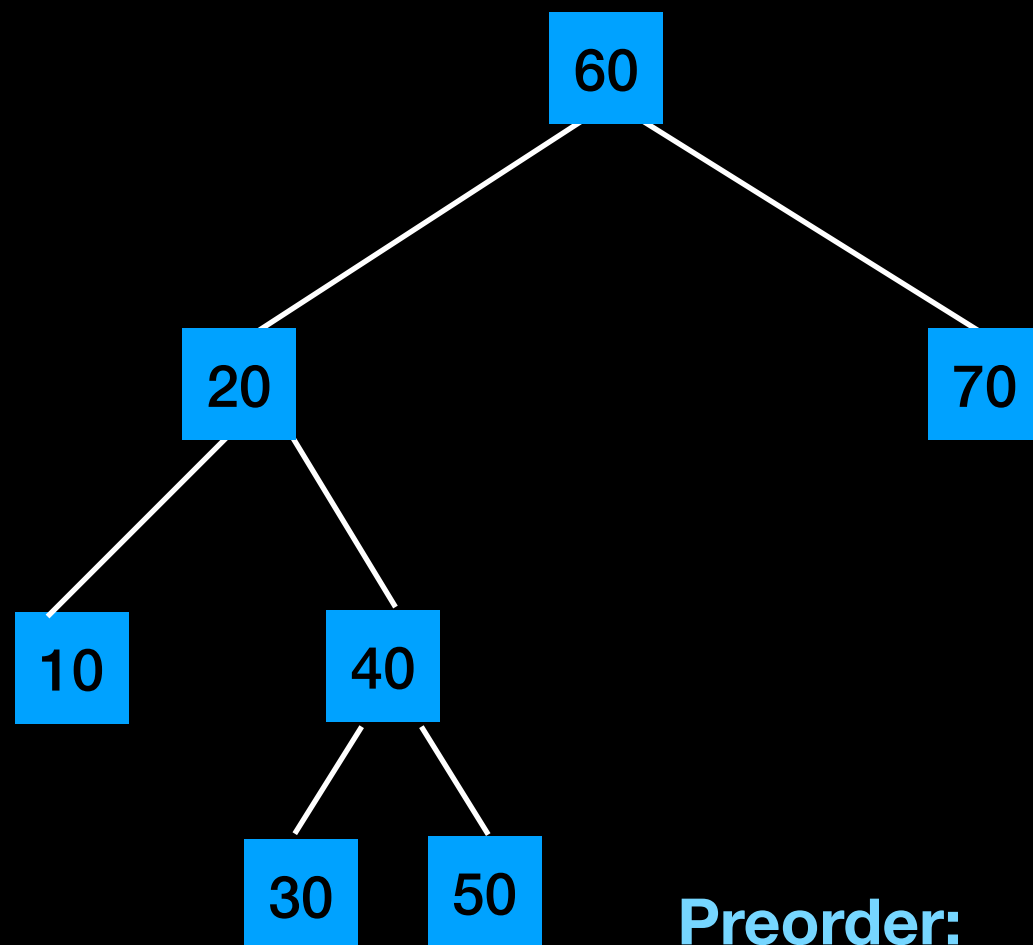
Order matters!!!



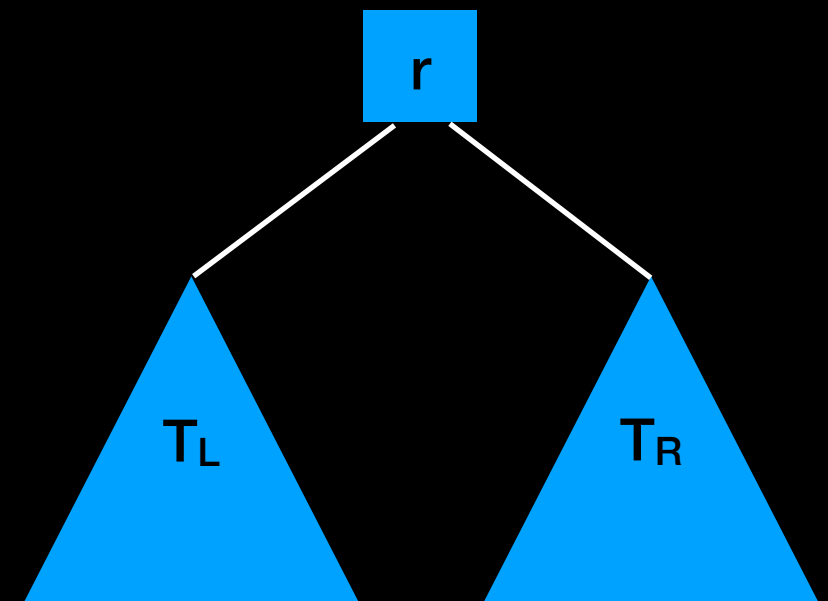
Visit (retrieve, print, modify ...) every node in the tree

Preorder Traversal:

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



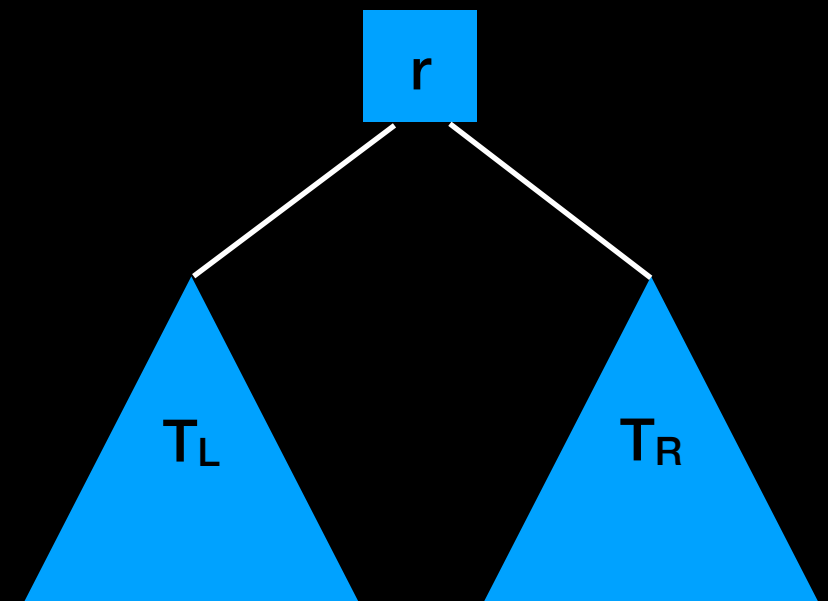
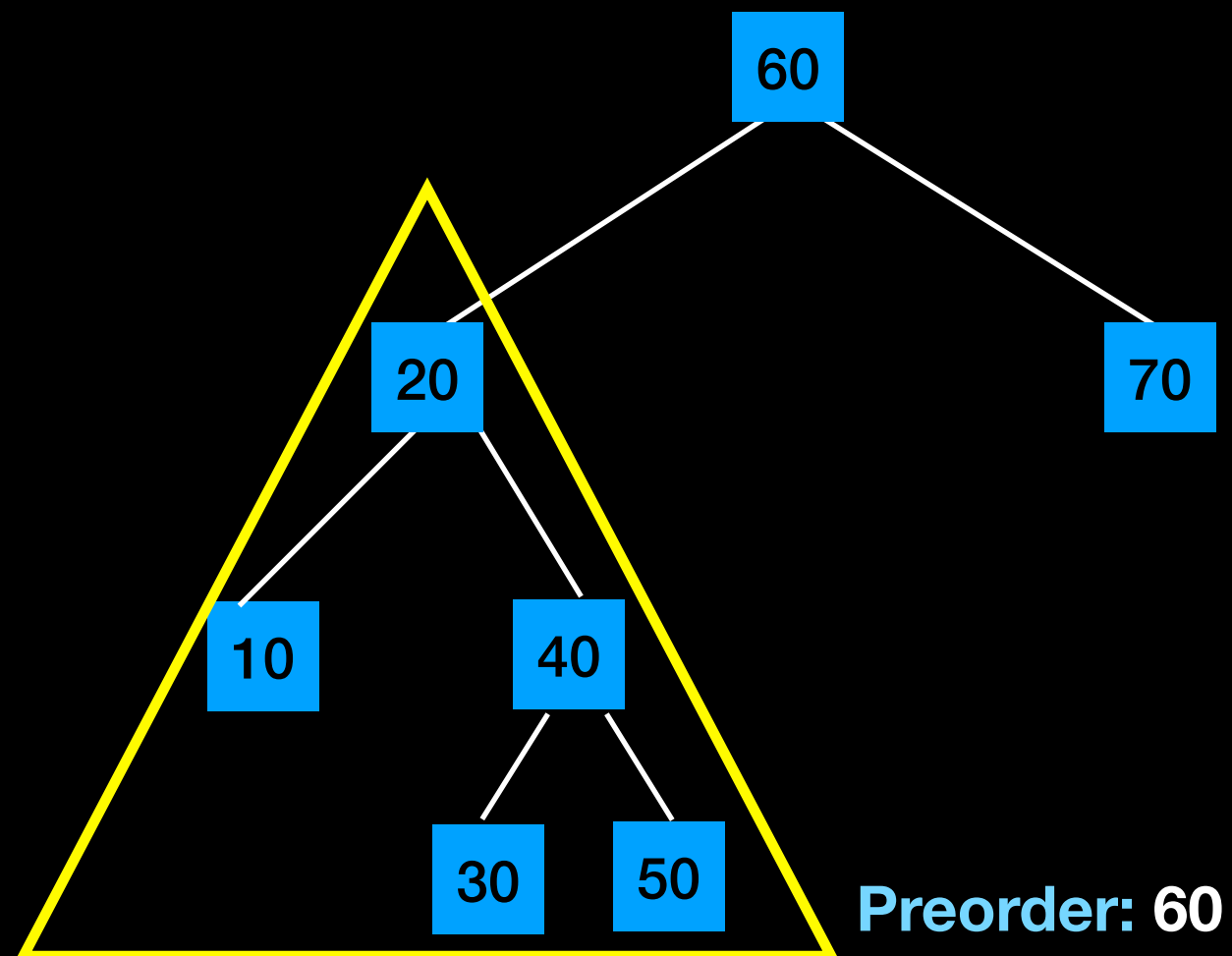
Preorder:



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

Preorder Traversal:

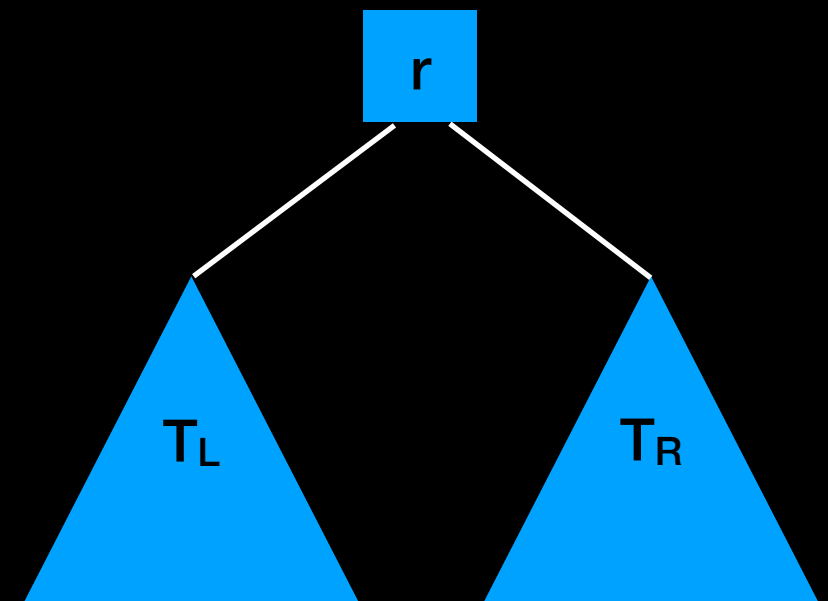
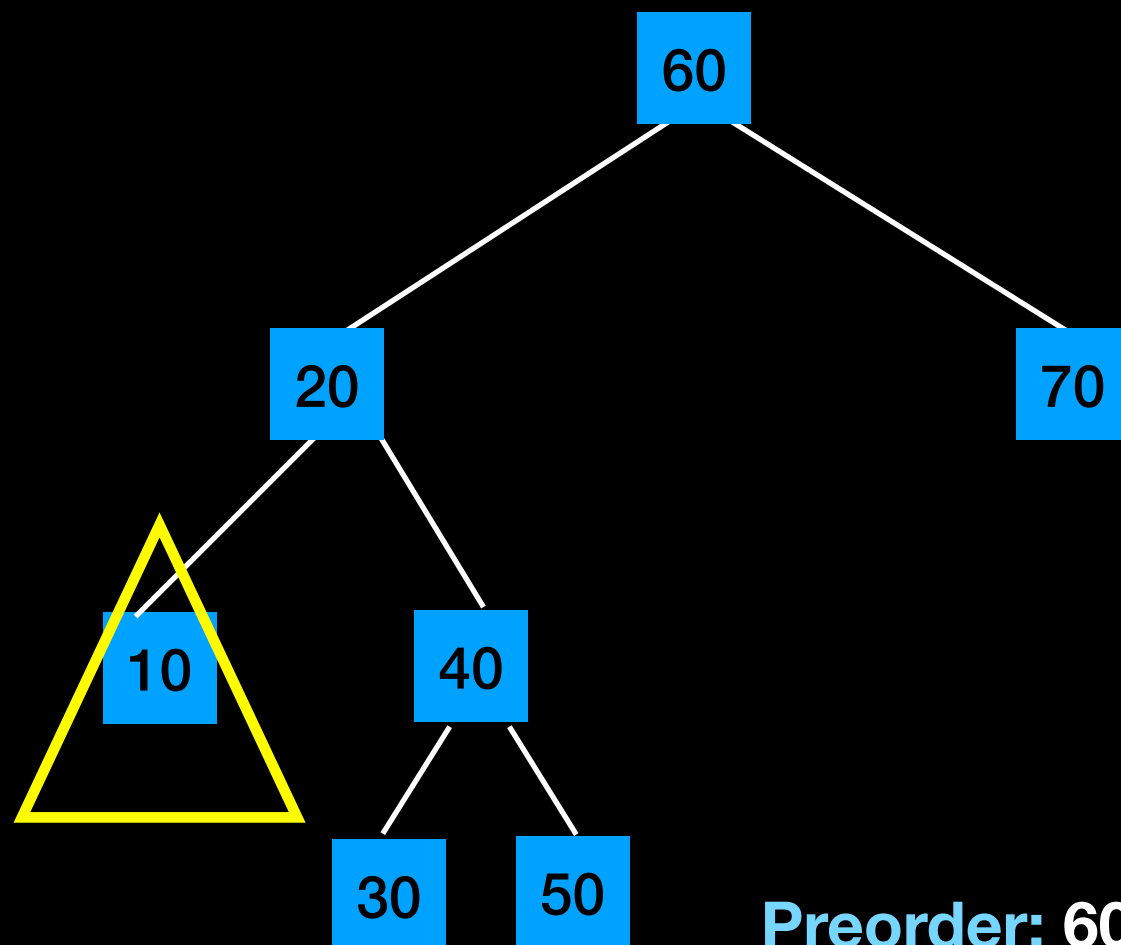
```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse  $T_L$ 
    traverse  $T_R$ 
}
```



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

Preorder Traversal:

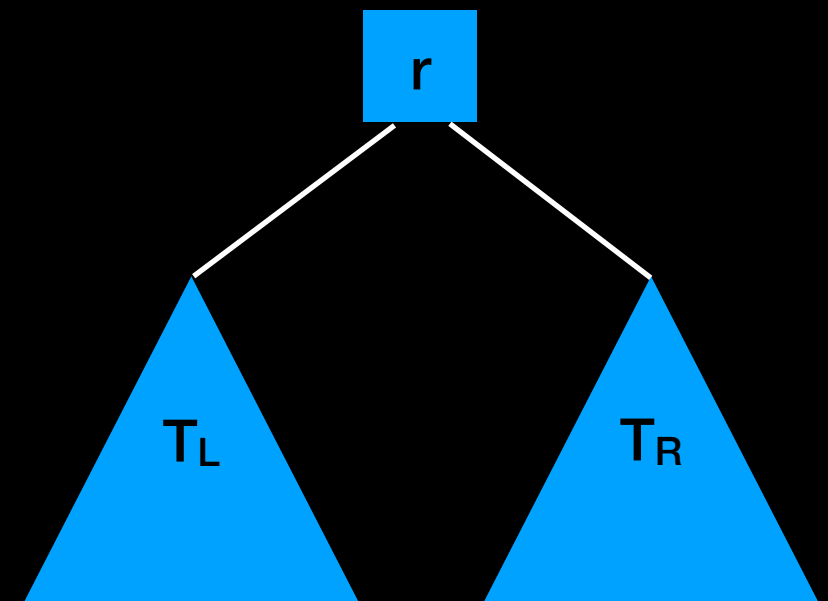
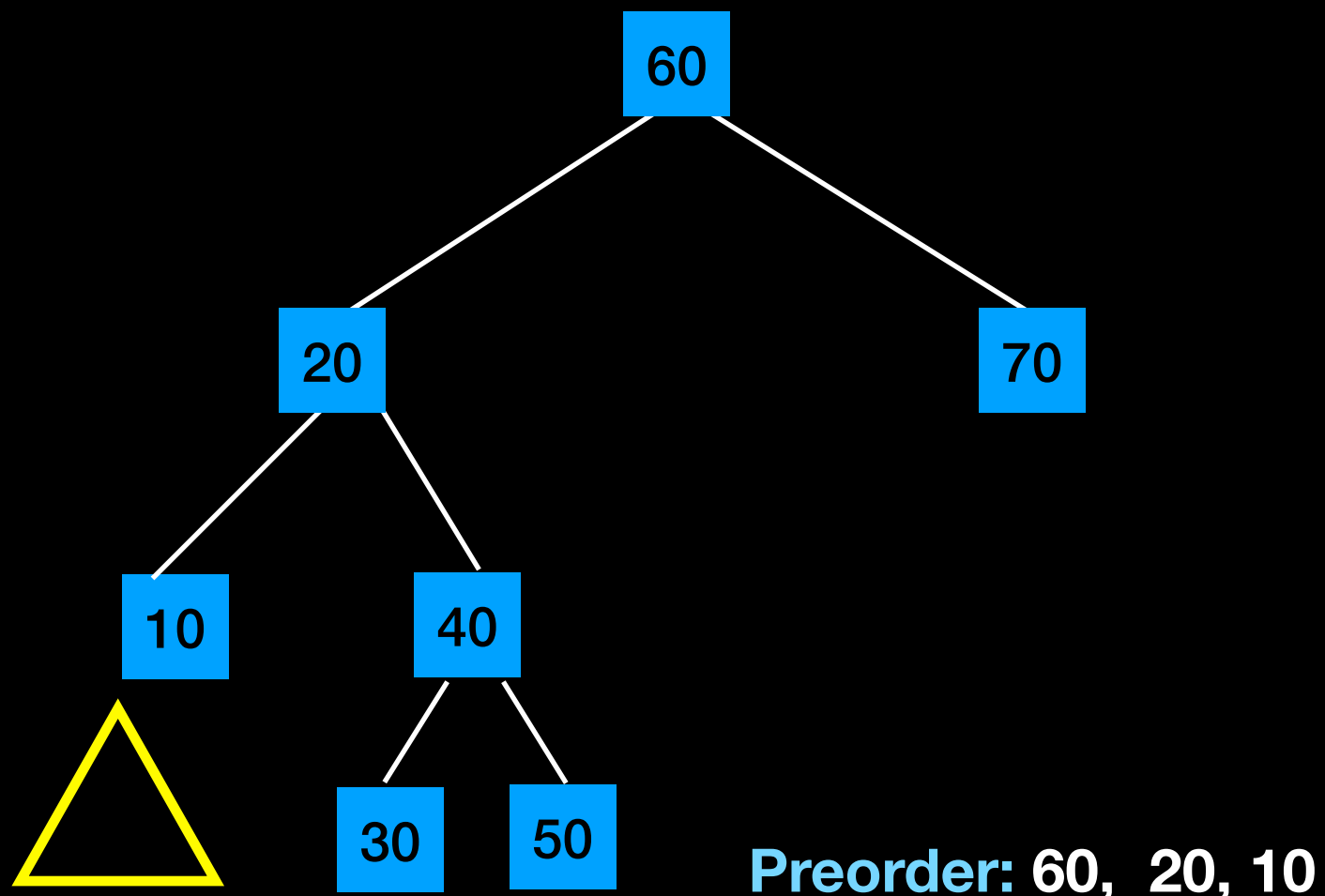
```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse  $T_L$ 
    traverse  $T_R$ 
}
```



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

Preorder Traversal:

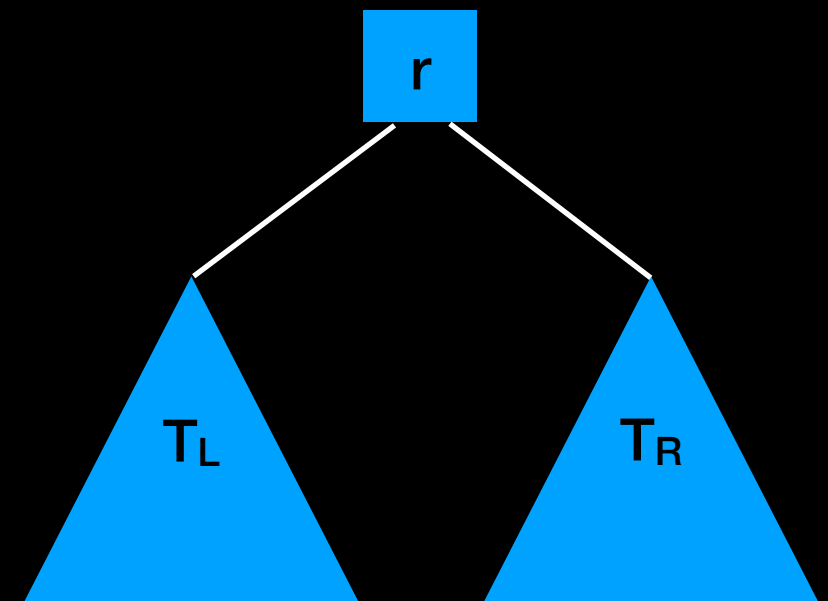
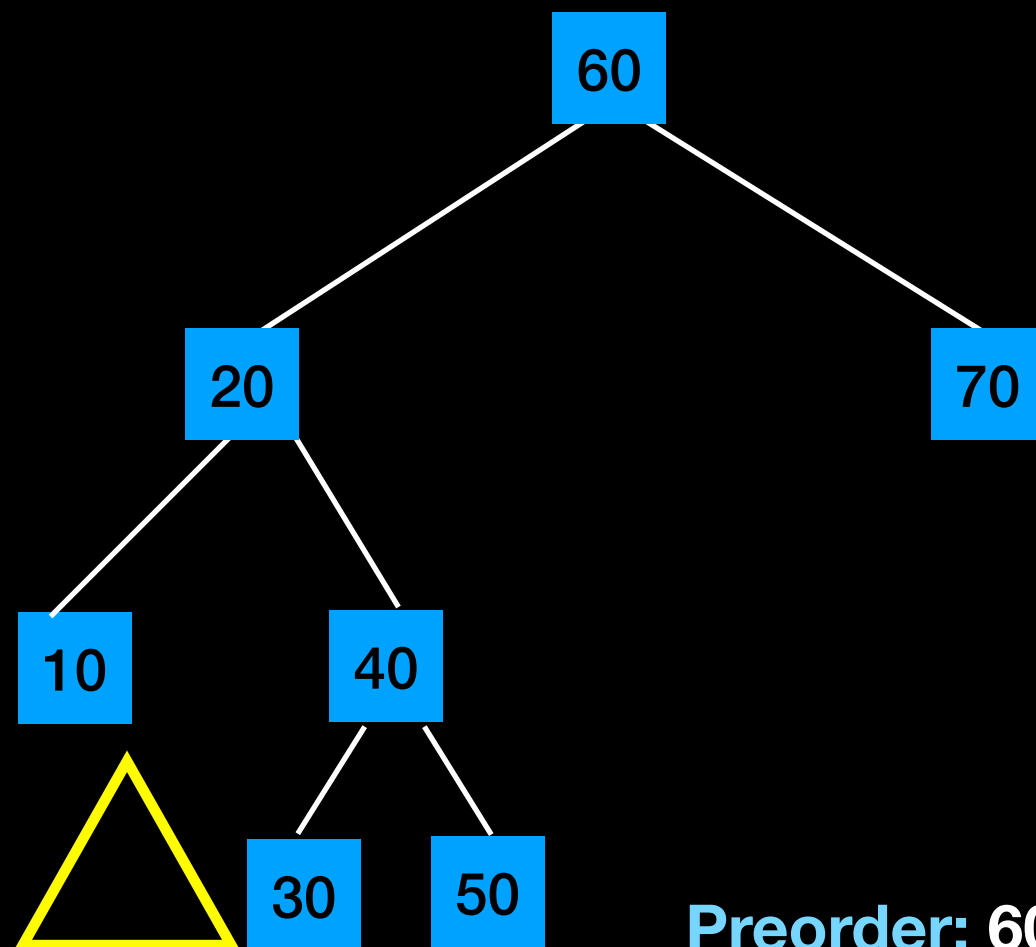
```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse  $T_L$ 
    traverse  $T_R$ 
}
```



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

Preorder Traversal:

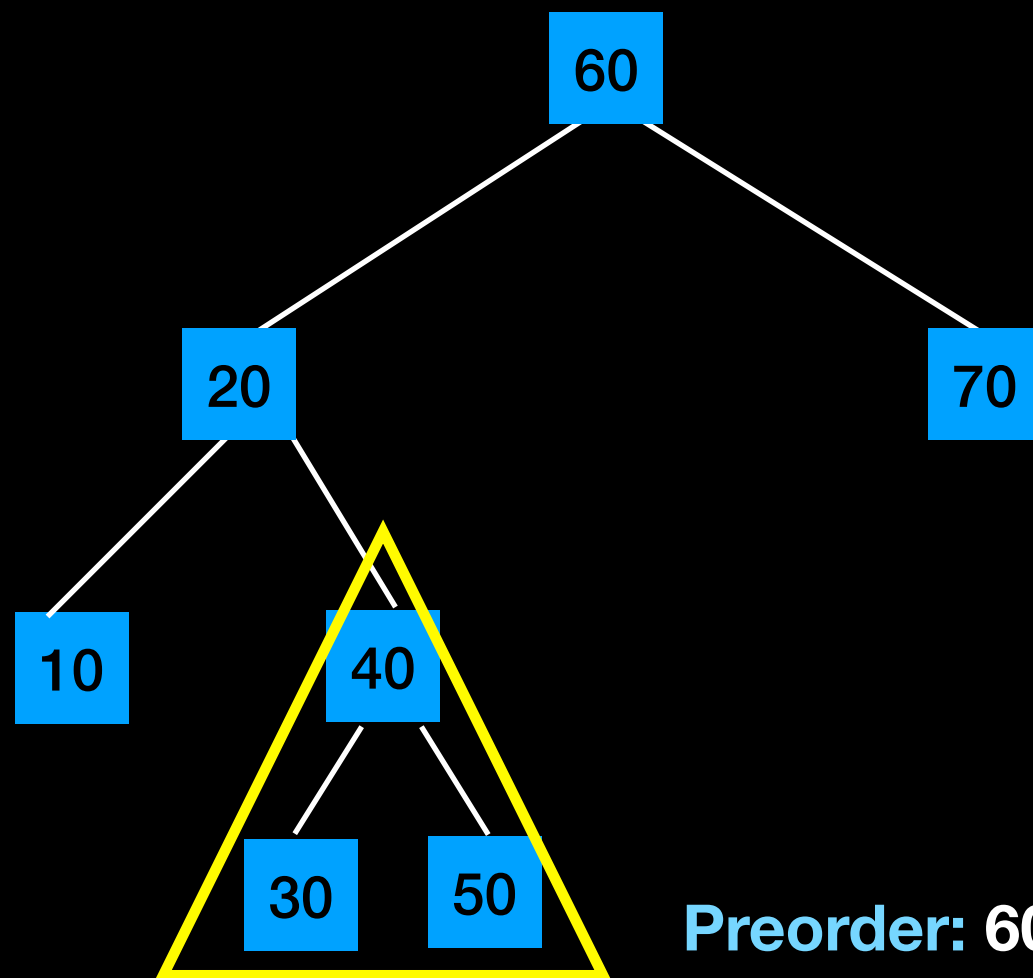
```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse  $T_L$ 
    traverse  $T_R$ 
}
```



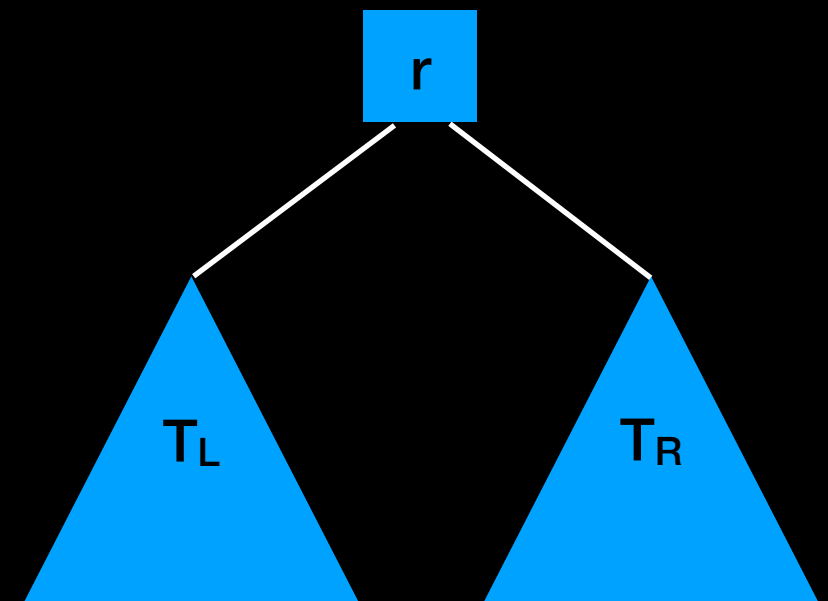
Visit (retrieve, print, modify ...) every node in the tree

Preorder Traversal:

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



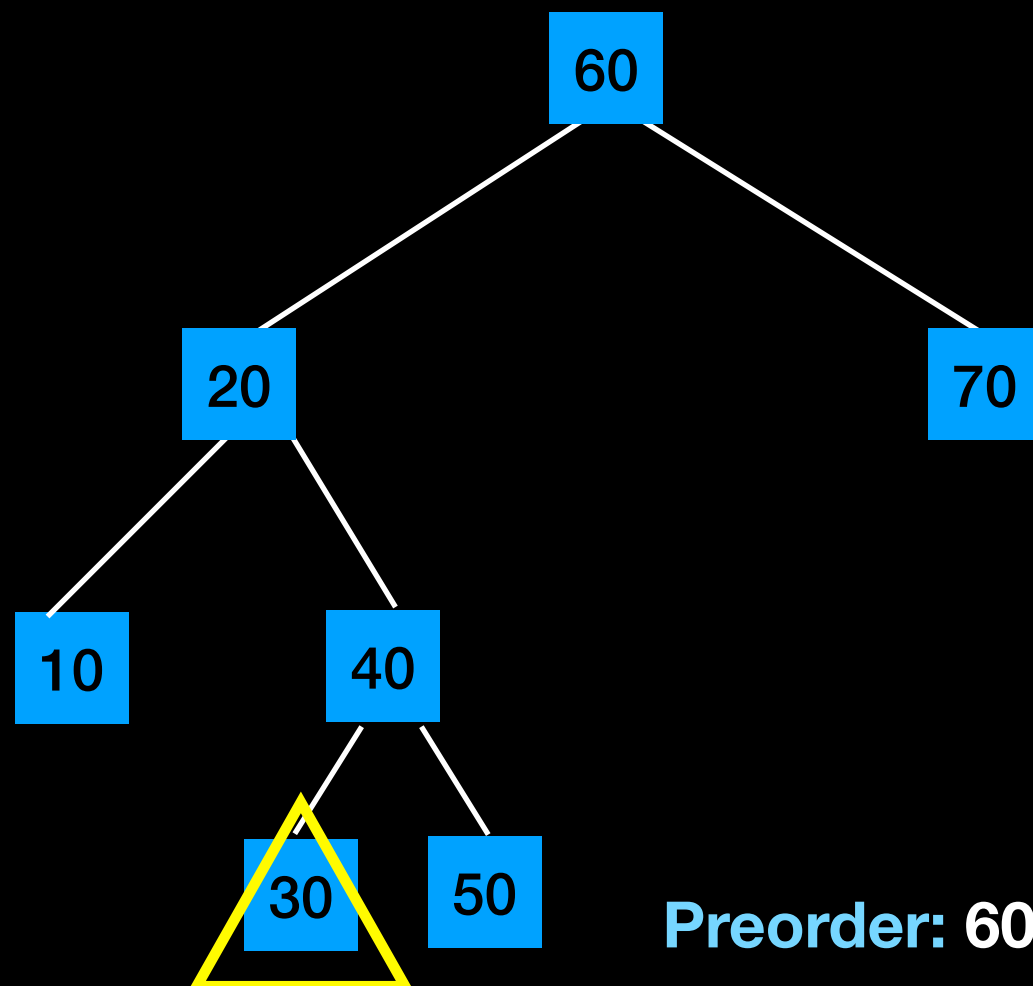
Preorder: 60, 20, 10



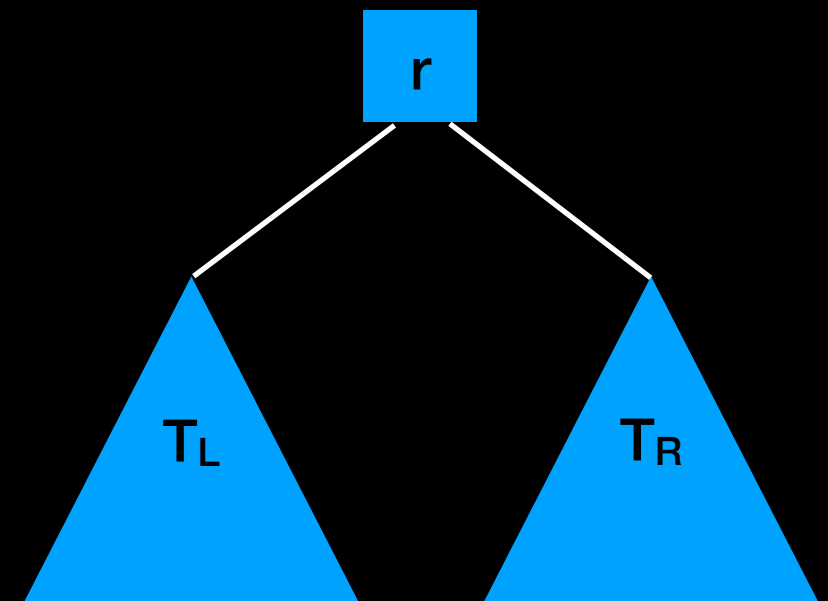
Visit (retrieve, print, modify ...) every node in the tree

Preorder Traversal:

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



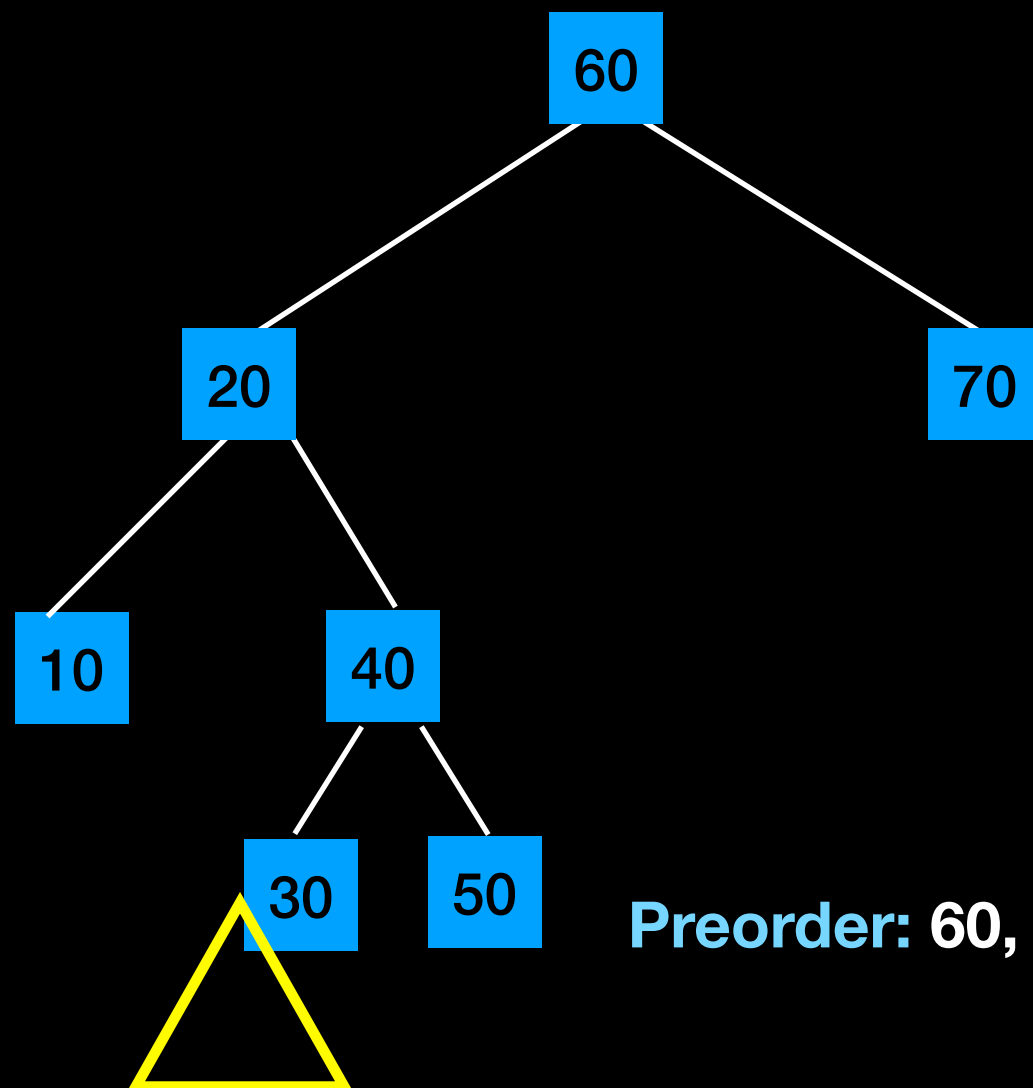
Preorder: 60, 20, 10, 40



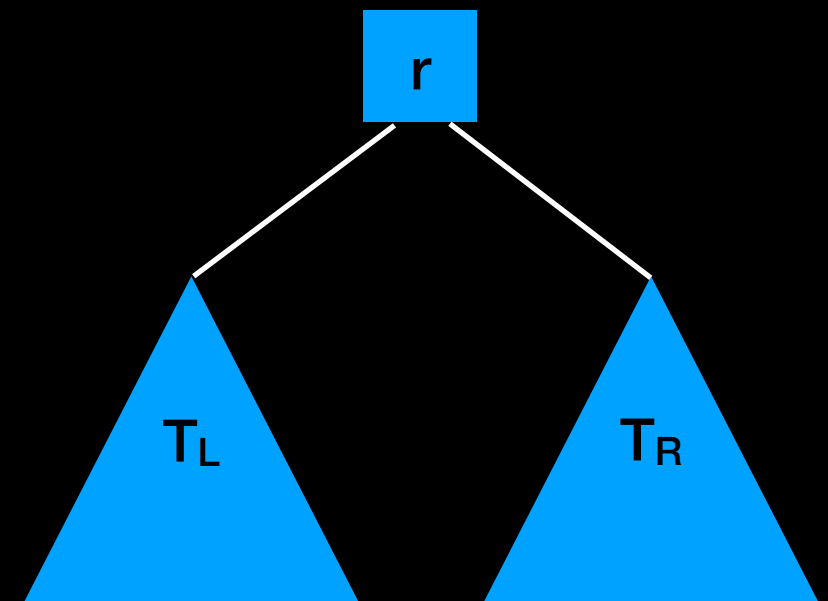
Visit (retrieve, print, modify ...) every node in the tree

Preorder Traversal:

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



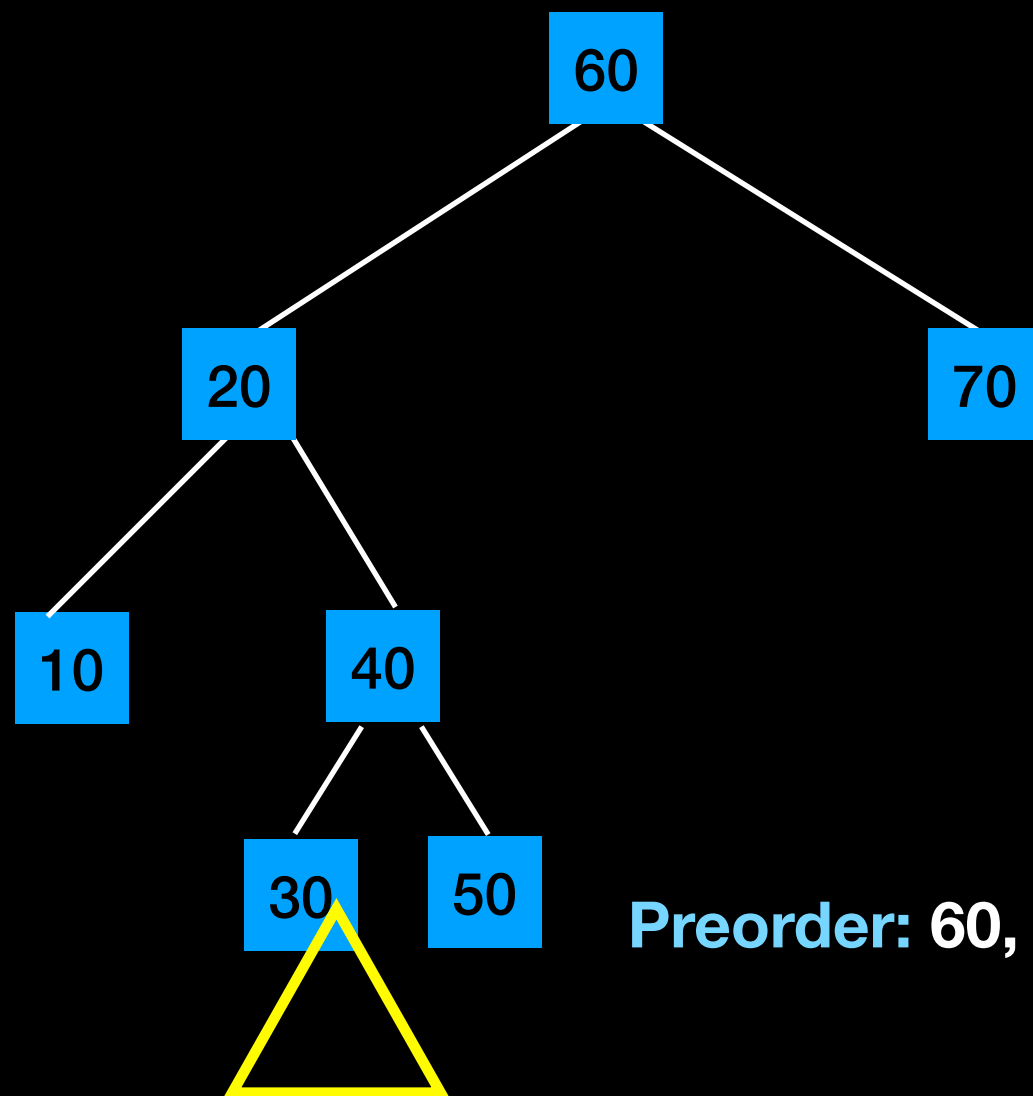
Preorder: 60, 20, 10, 40, 30



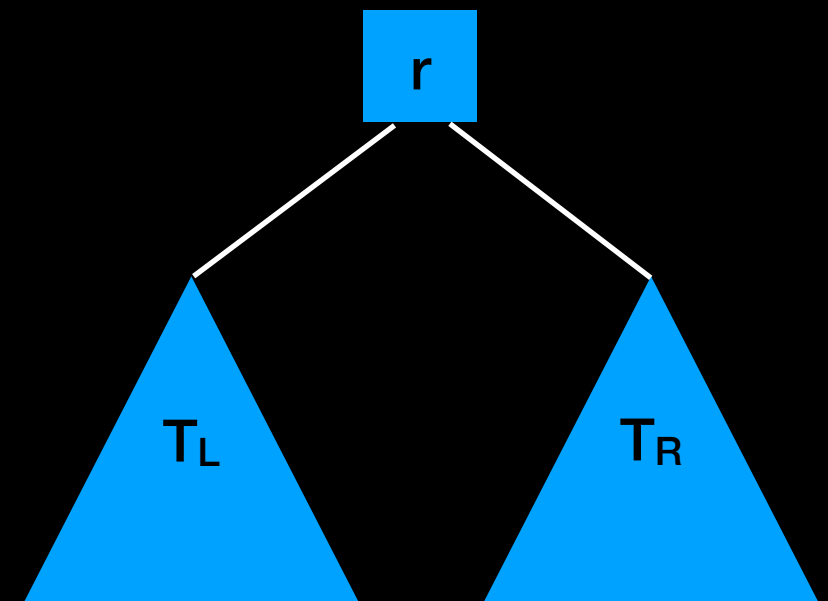
Visit (retrieve, print, modify ...) every node in the tree

Preorder Traversal:

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



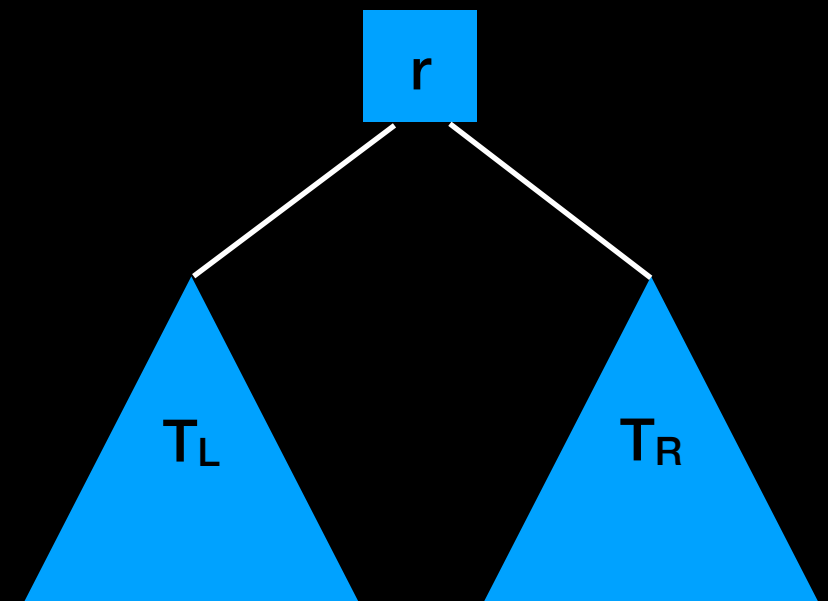
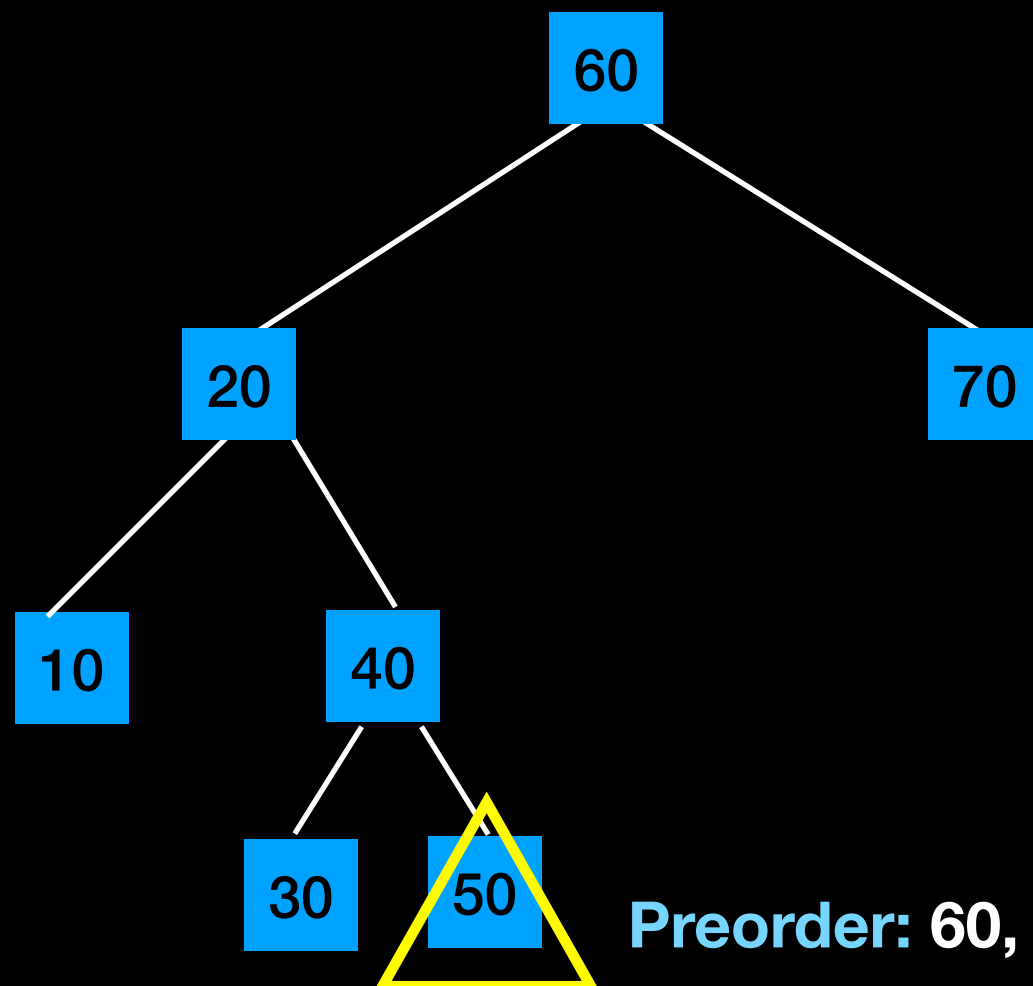
Preorder: 60, 20, 10, 40, 30



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

Preorder Traversal:

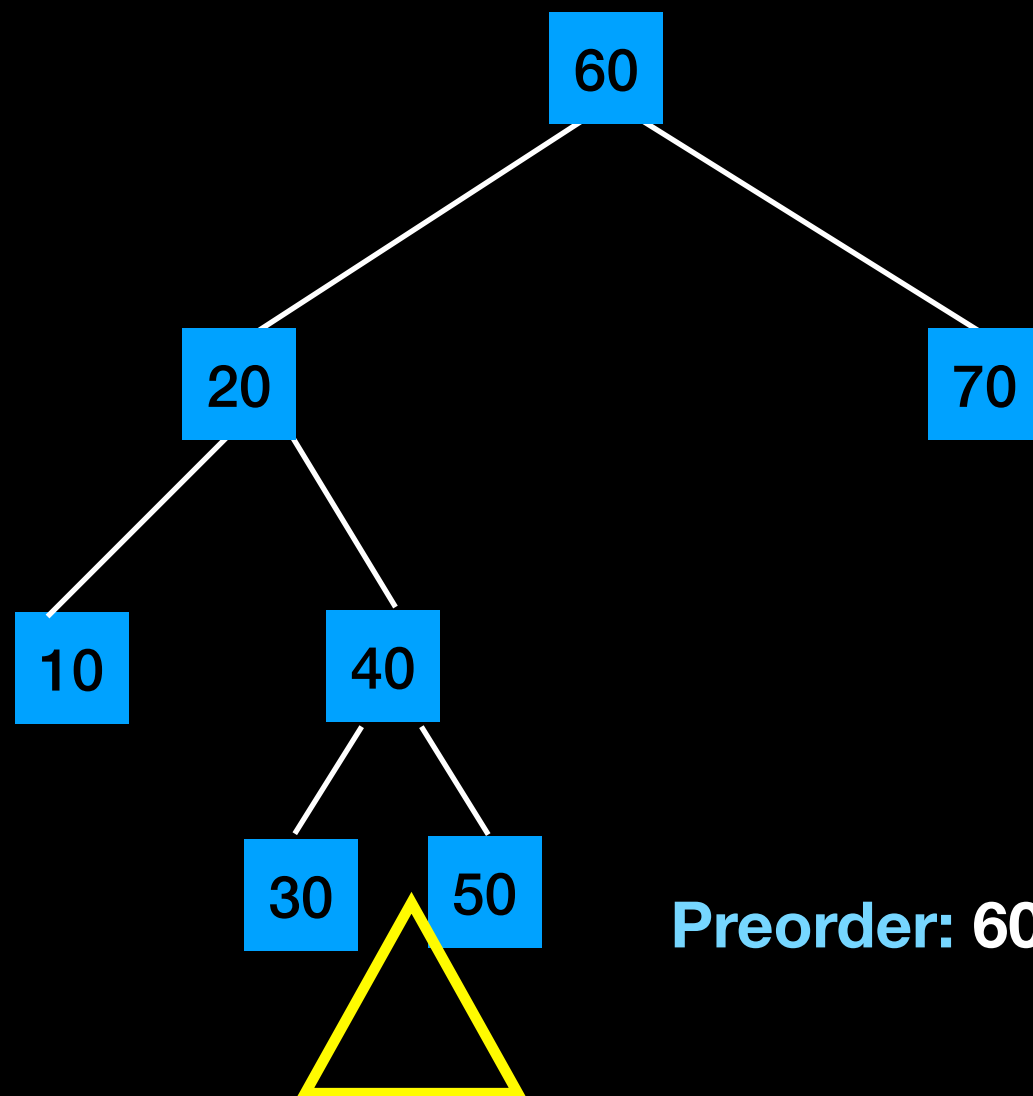
```
if (T is not empty) //implicit base case
{
    visit the root r
    traverse  $T_L$ 
    traverse  $T_R$ 
}
```



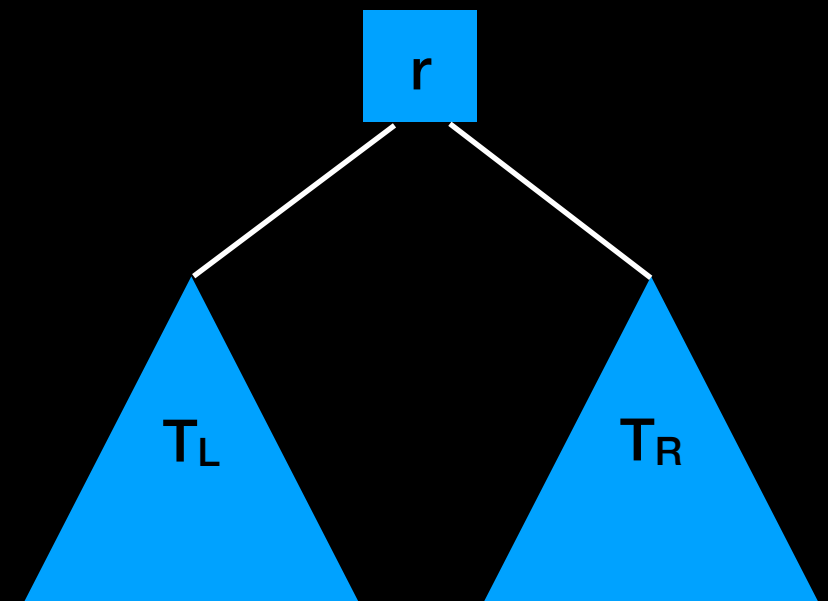
Visit (retrieve, print, modify ...) every node in the tree

Preorder Traversal:

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



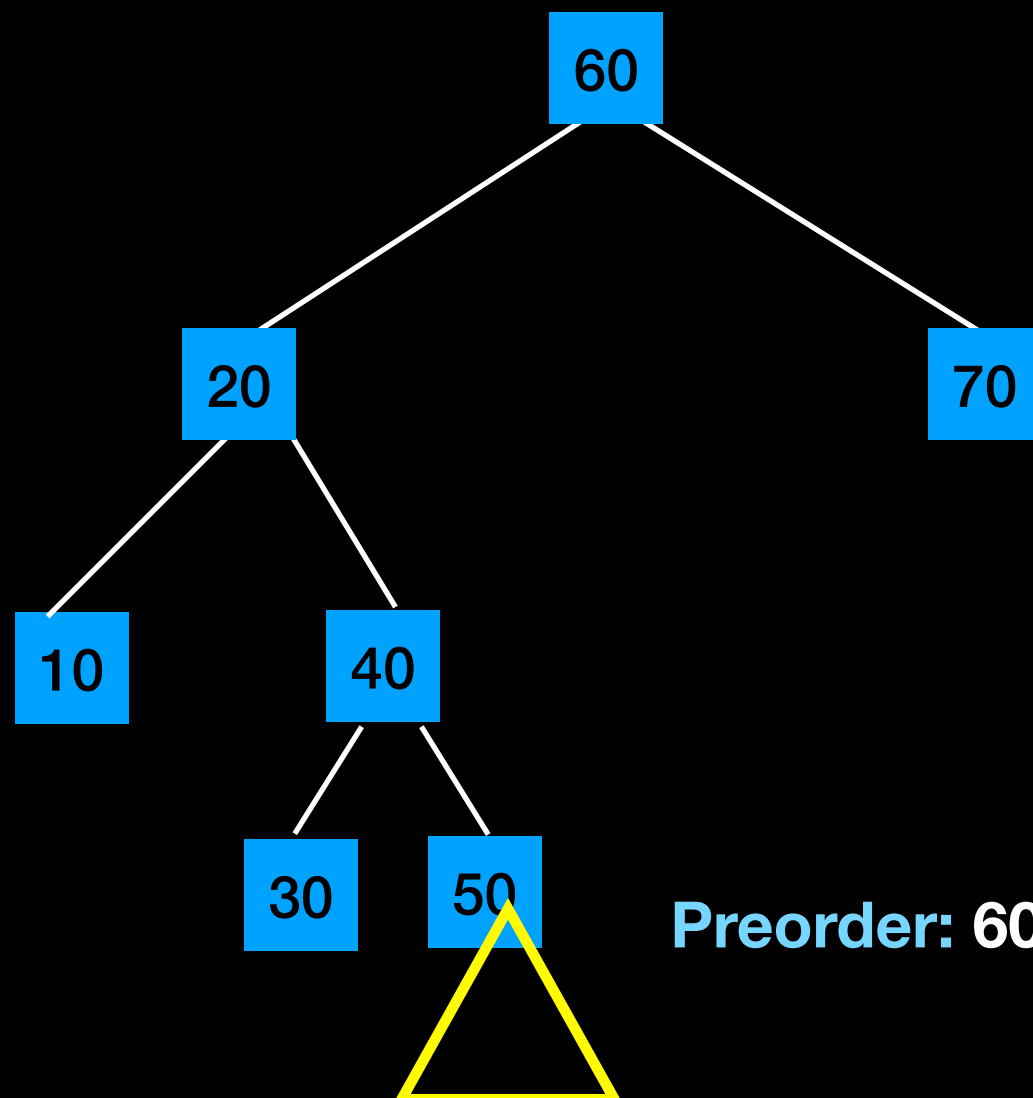
Preorder: 60, 20, 10, 40, 30, 50



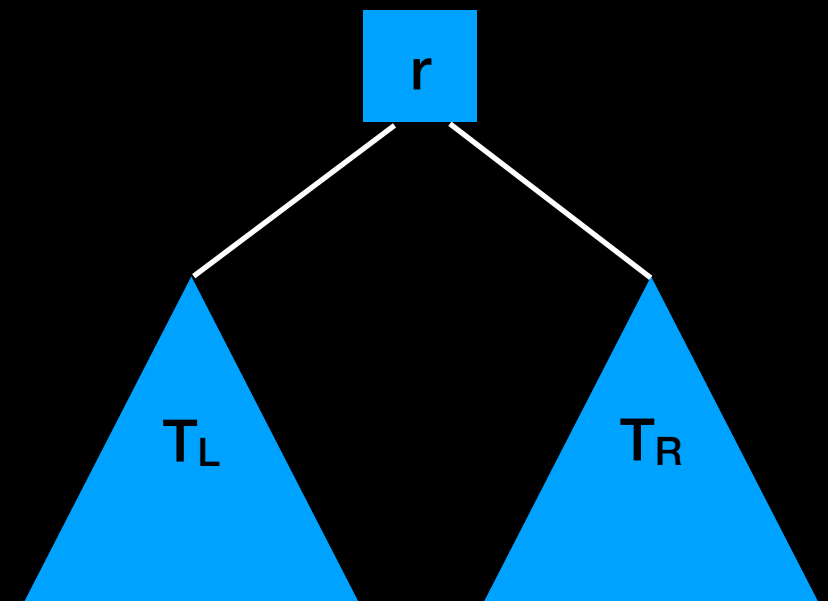
Visit (retrieve, print, modify ...) every node in the tree

Preorder Traversal:

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



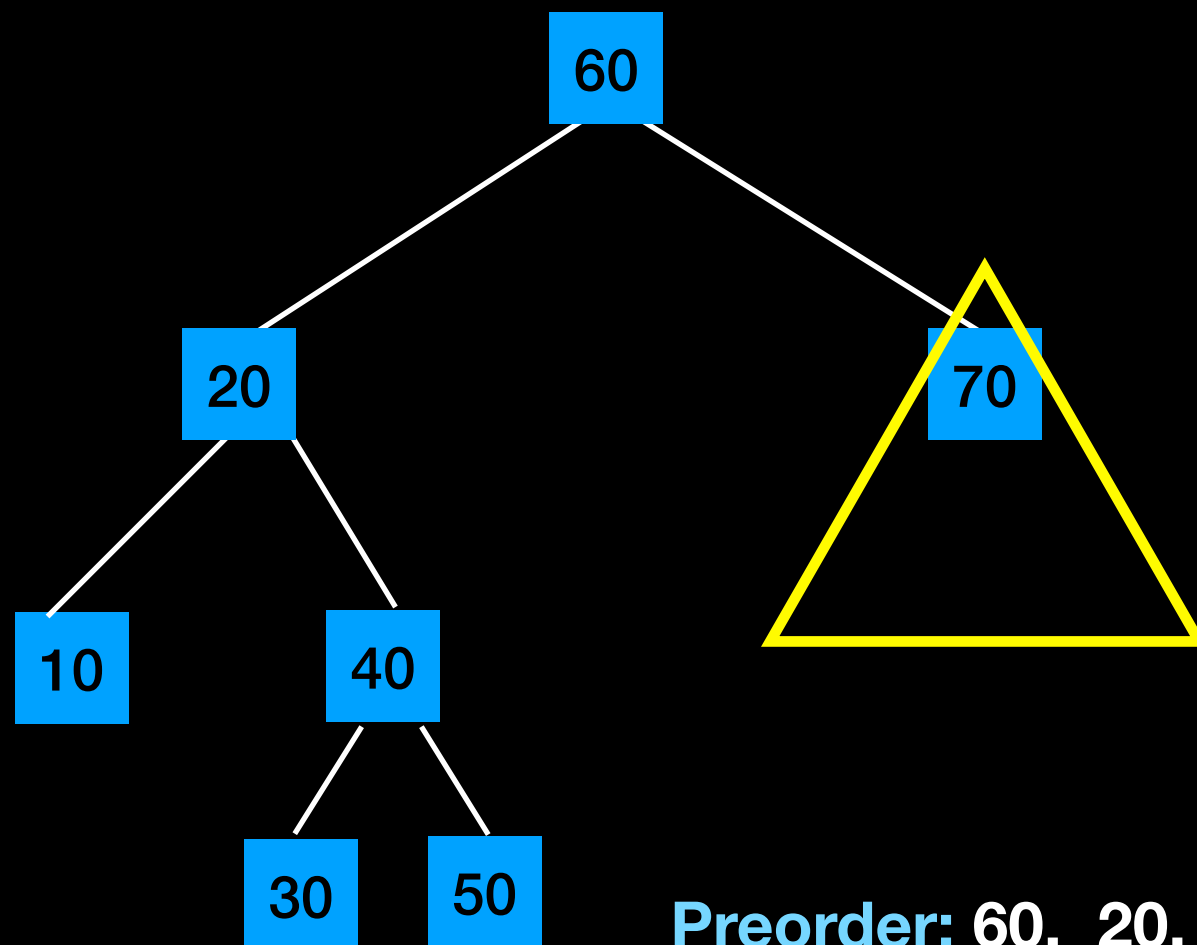
Preorder: 60, 20, 10, 40, 30, 50



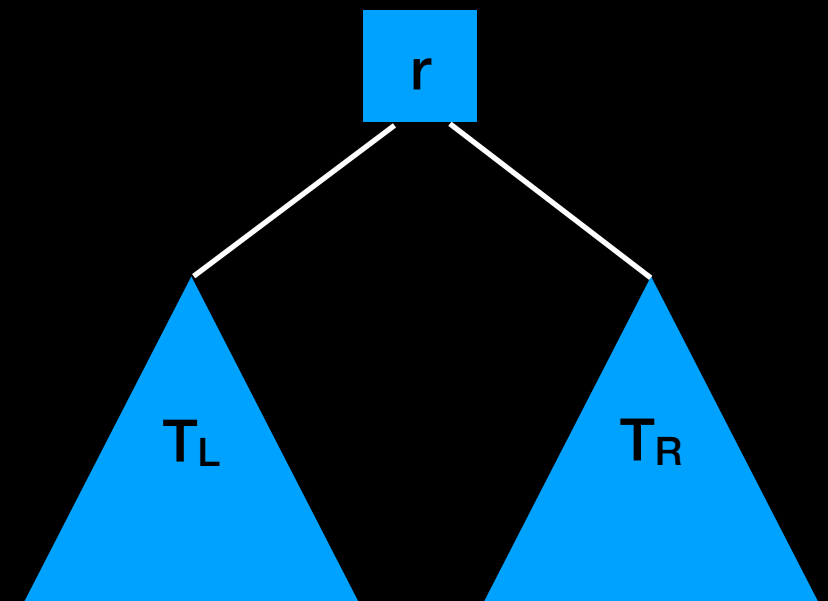
Visit (retrieve, print, modify ...) every node in the tree

Preorder Traversal:

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



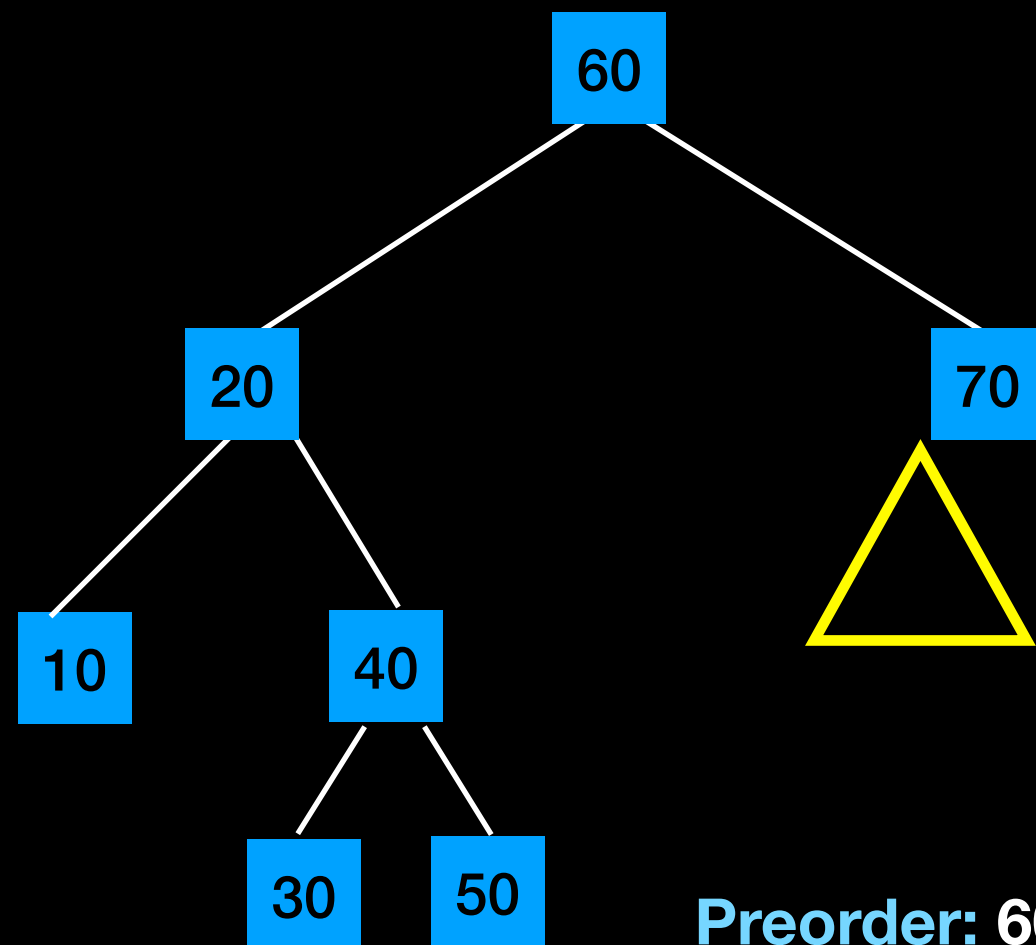
Preorder: 60, 20, 10, 40, 30, 50



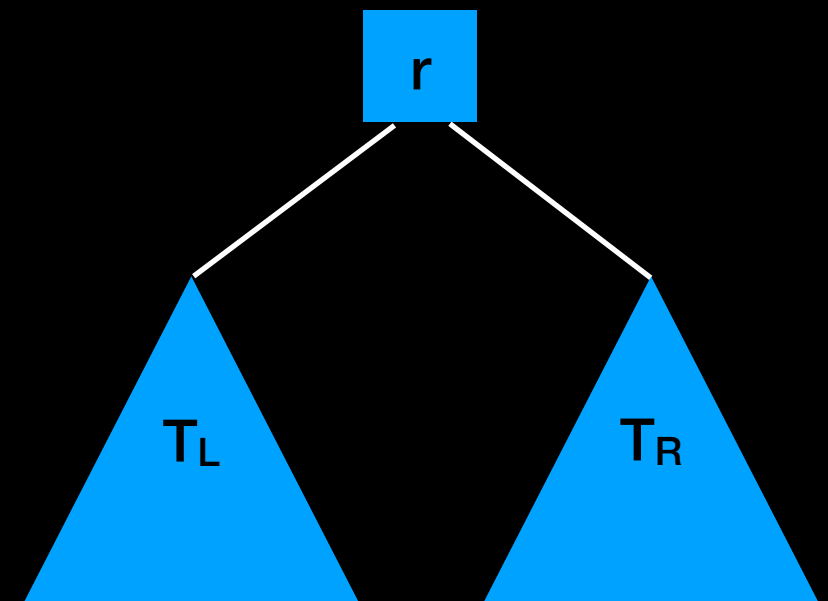
Visit (retrieve, print, modify ...) every node in the tree

Preorder Traversal:

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



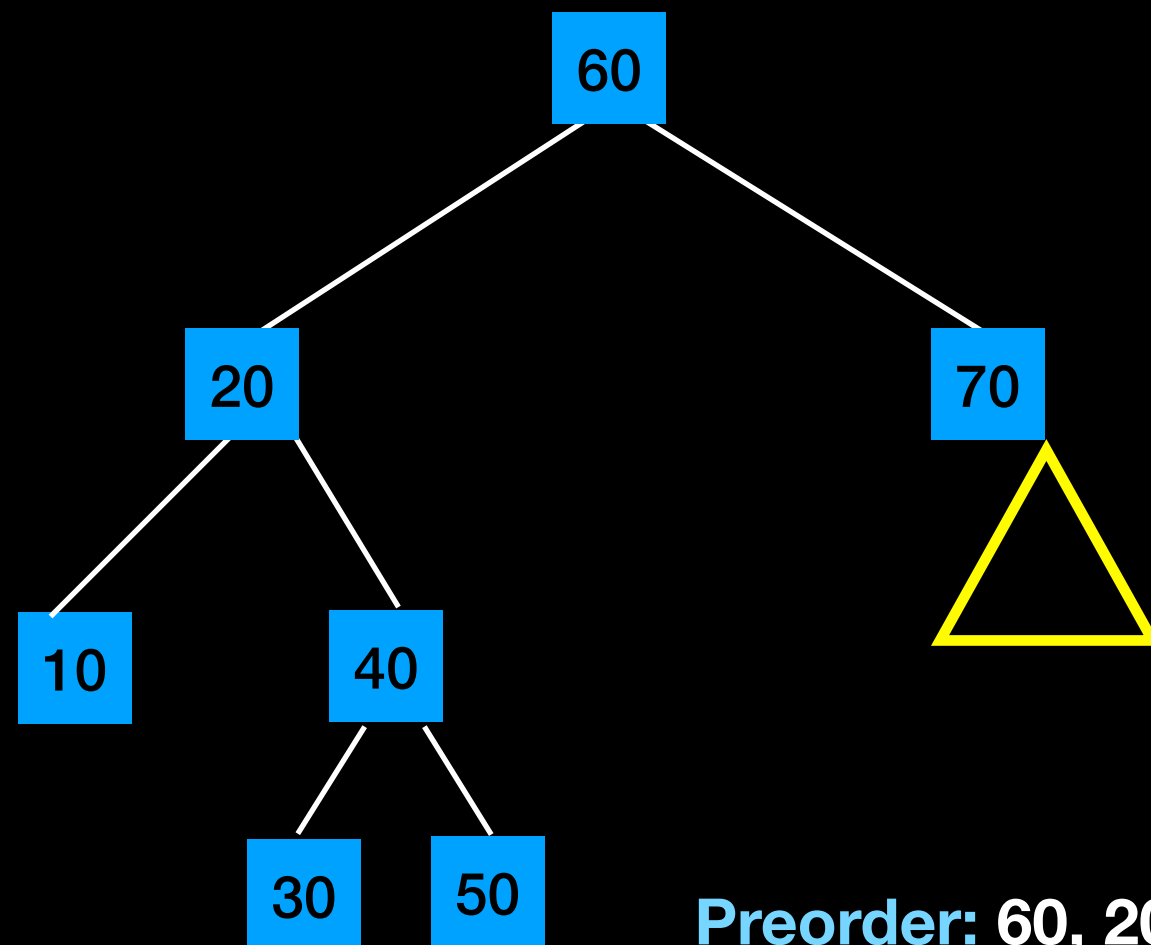
Preorder: 60, 20, 10, 40, 30, 50, 70



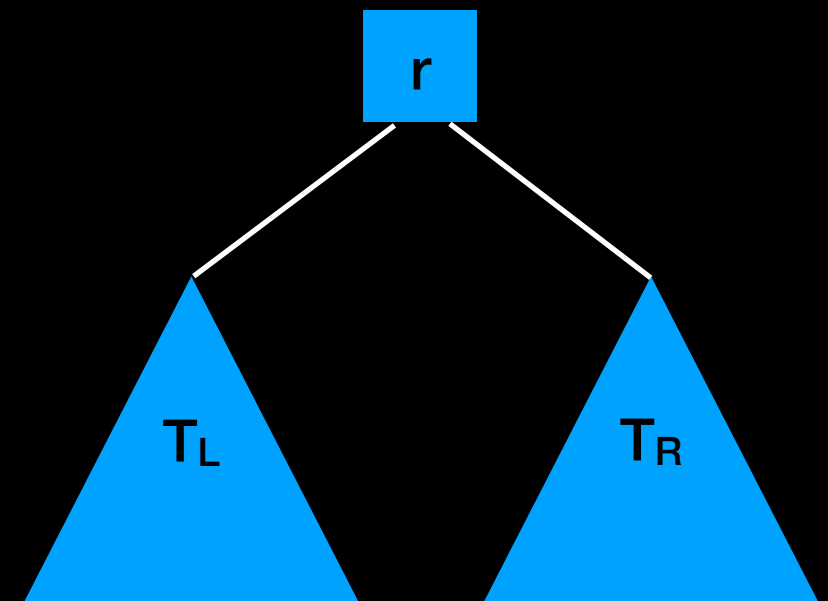
Visit (retrieve, print, modify ...) every node in the tree

Preorder Traversal:

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



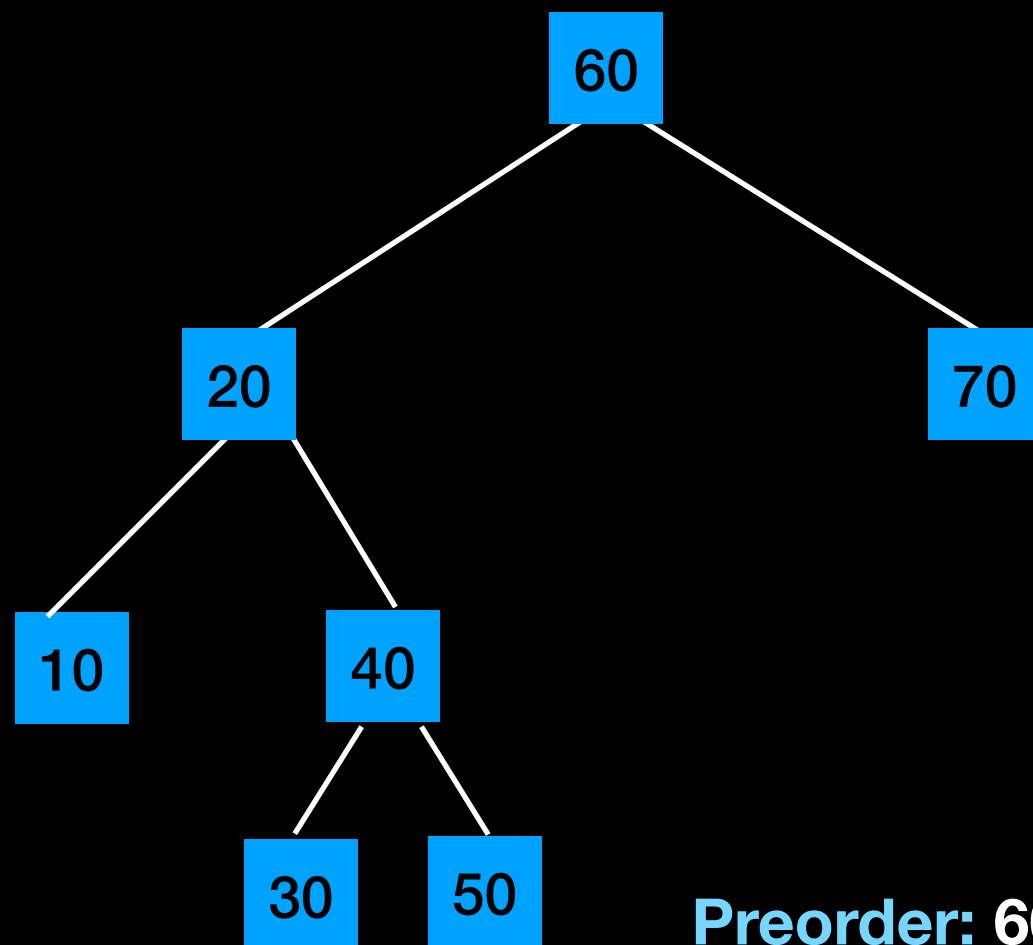
Preorder: 60, 20, 10, 40, 30, 50, 70



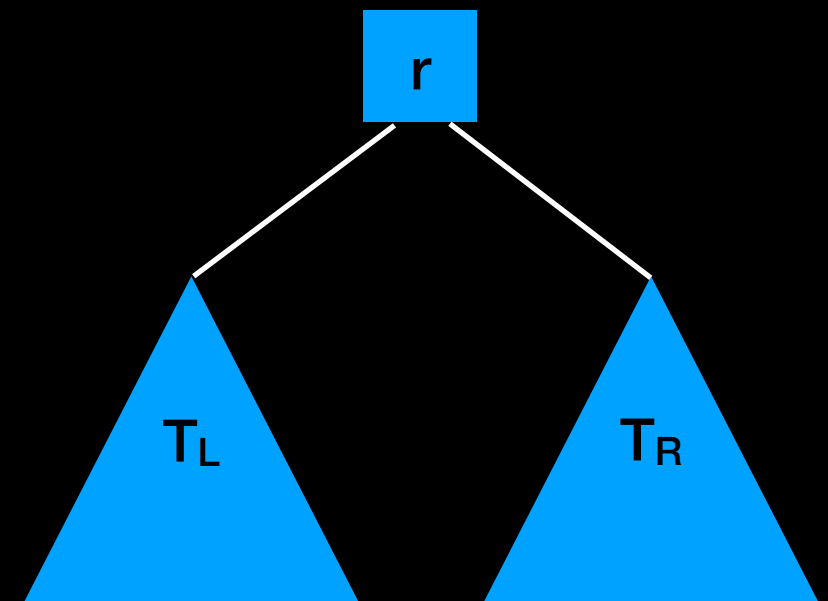
Visit (retrieve, print, modify ...) every node in the tree

Preorder Traversal:

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



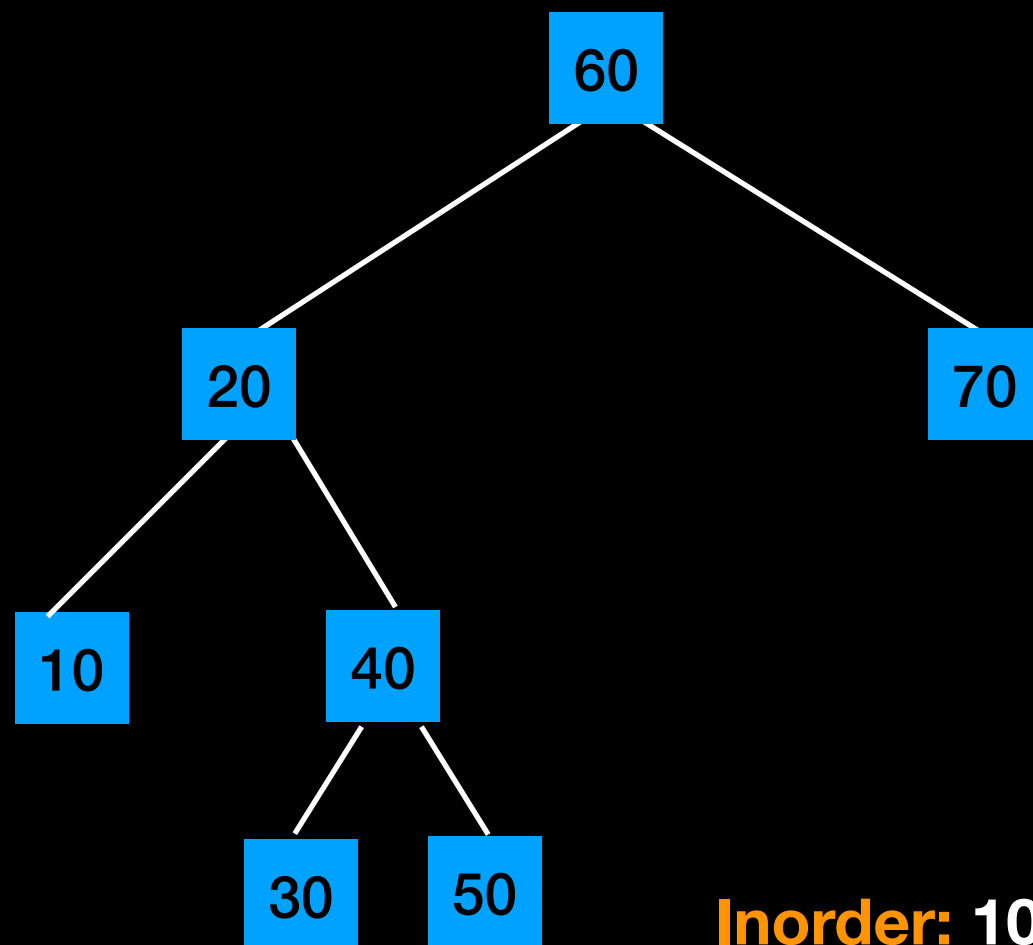
Preorder: 60, 20, 10, 40, 30, 50, 70



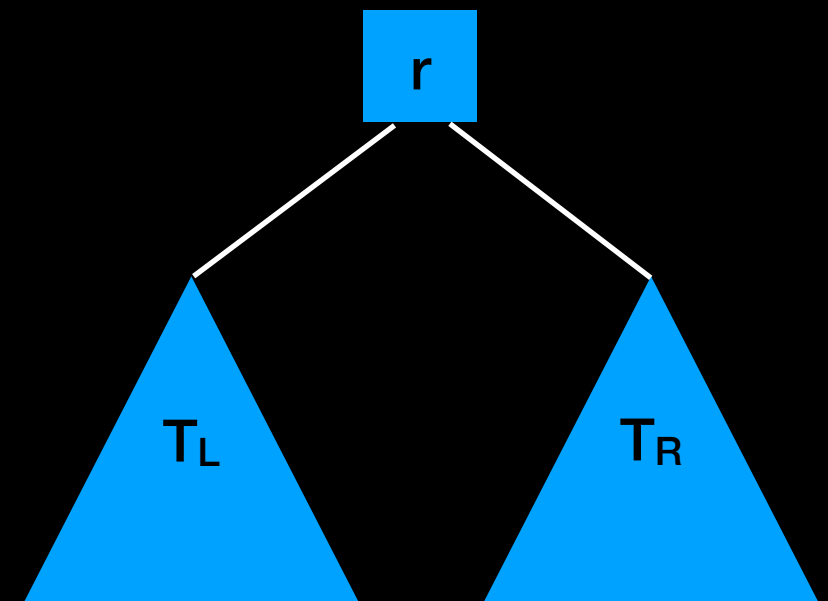
Visit (retrieve, print, modify ...) every node in the tree

Inorder Traversal:

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



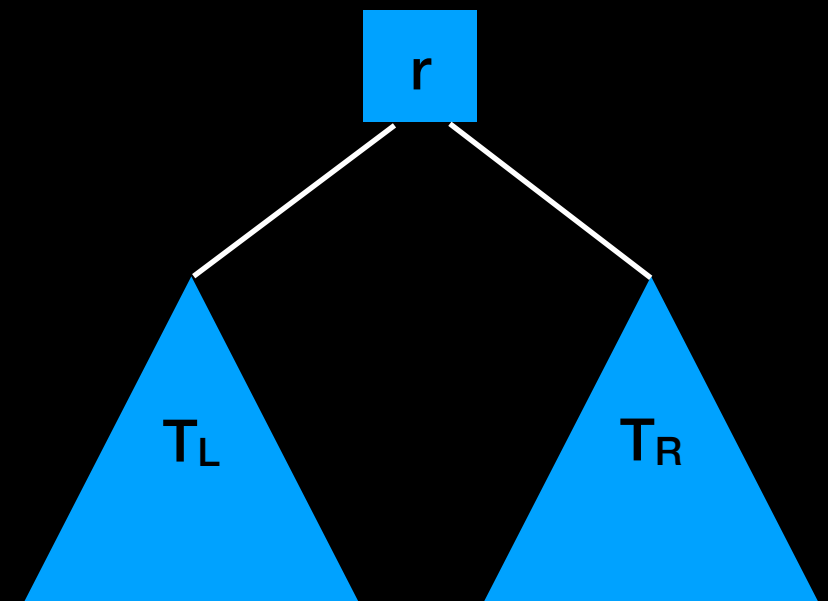
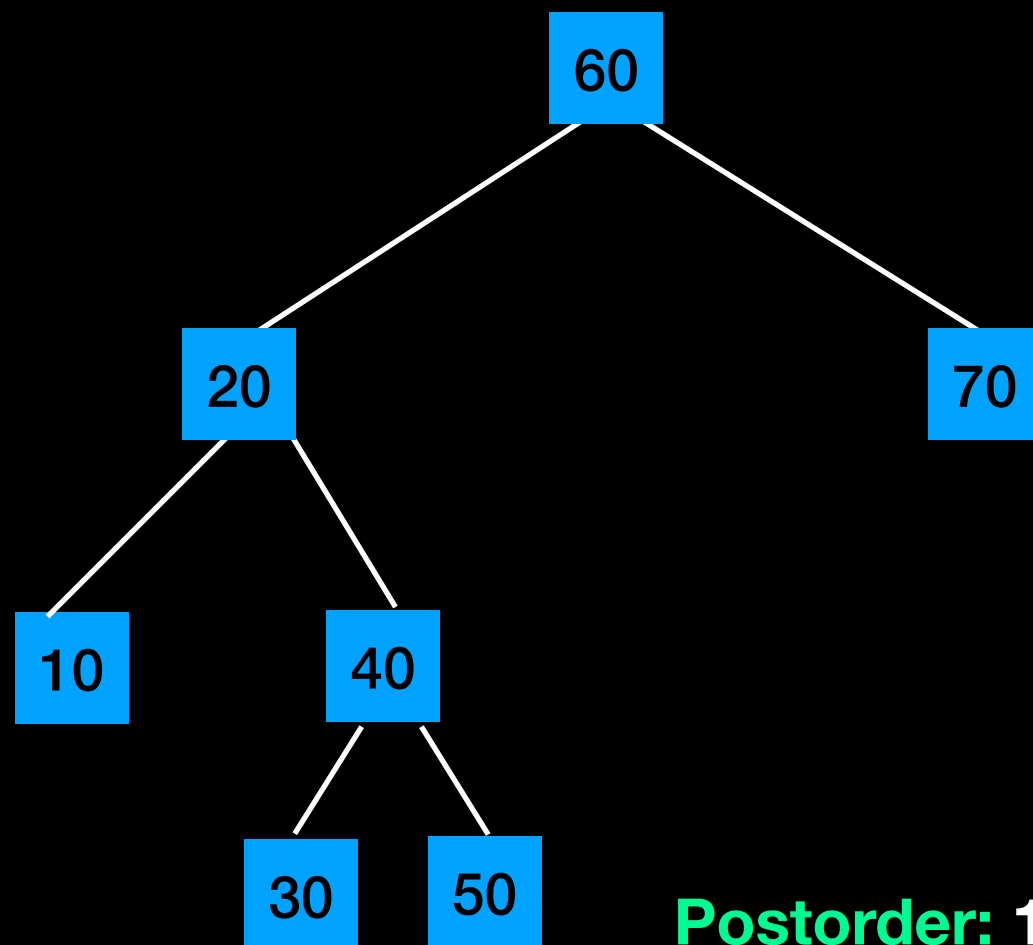
Inorder: 10, 20, 30, 40, 50, 60, 70



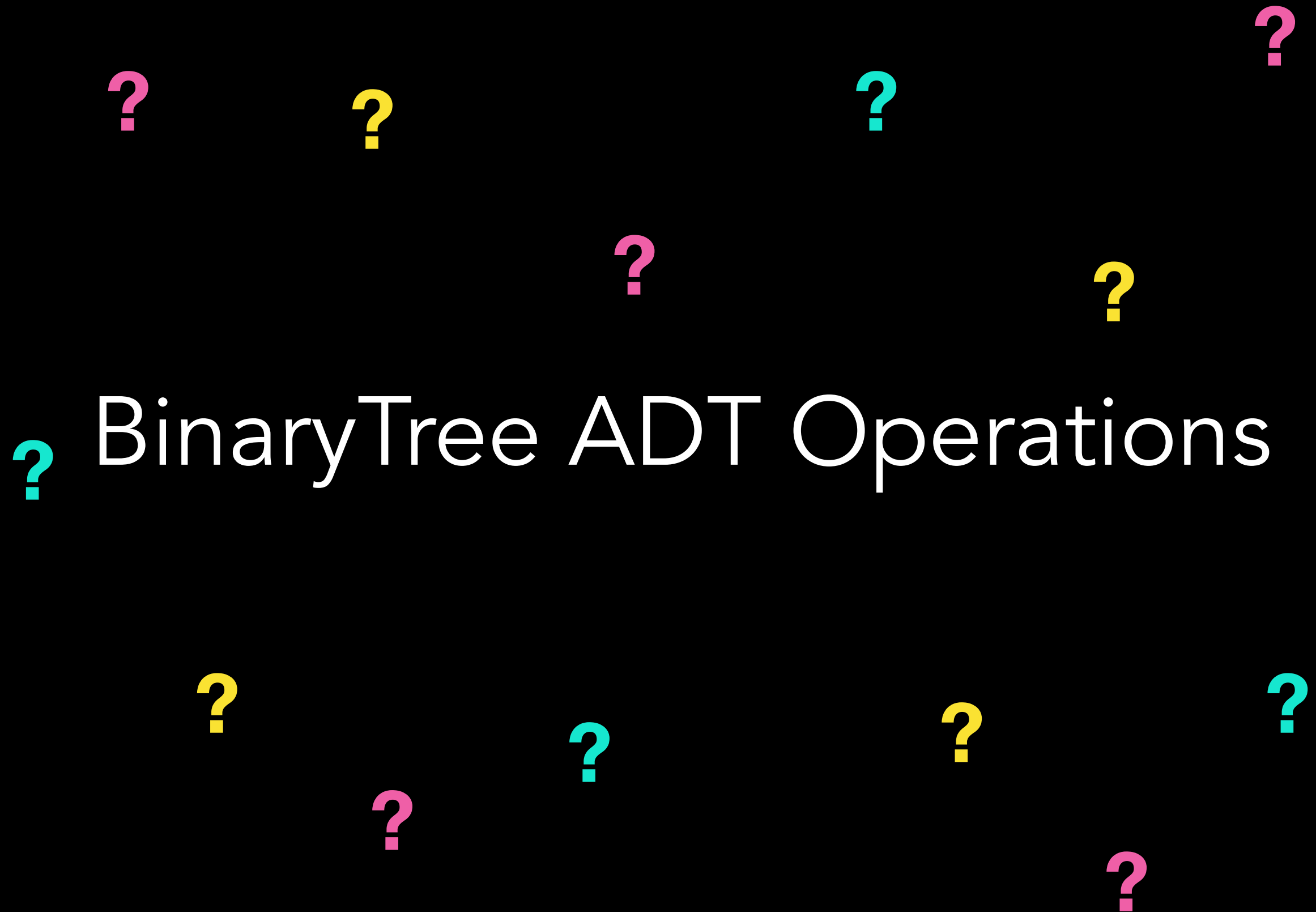
Visit (retrieve, print, modify ...) every node in the tree

Postorder Traversal:

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



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



```

#ifndef BinaryTree_H_
#define BinaryTree_H_

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

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

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

private: // implementation details here

}; // end BST

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

```

```

#ifndef BinaryTree_H_
#define BinaryTree_H_

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

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

    BinaryTree& operator= (const BinaryTree<ItemType>& 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

Considerations

Recall

Remember our Bag ADT?

- Array implementation
- Linked Chain implementation
- Assume no duplicates

Find an element: $O(n)$

Remove: Find element and if there remove it $O(n)$

Add: Check if element is there and if not add it $O(n)$

Recall

Remember our **Bag ADT**?

- Array implementation
- Linked Chain implementation
- Assume no duplicates

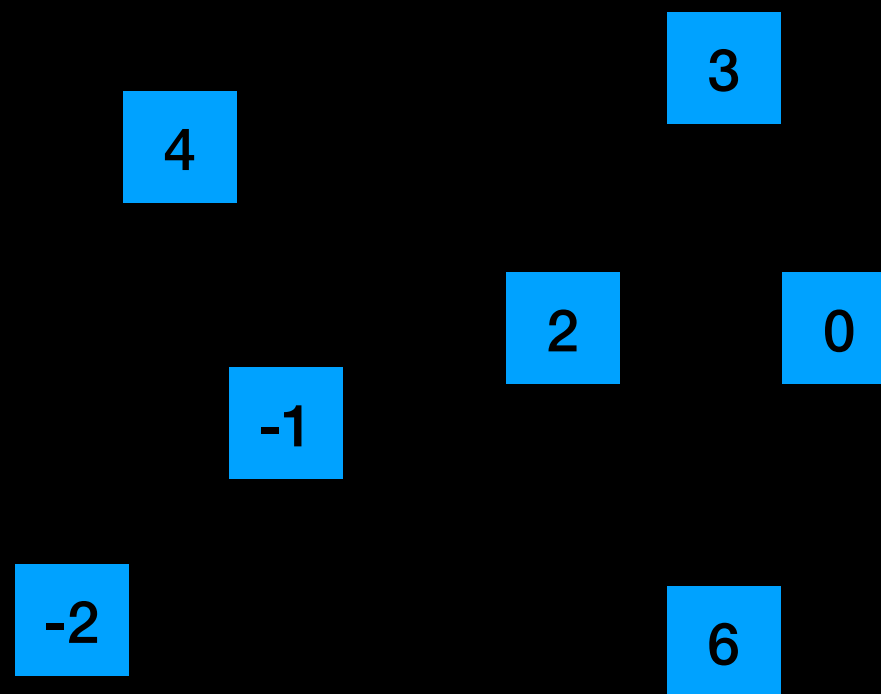


Find an element: $O(n)$

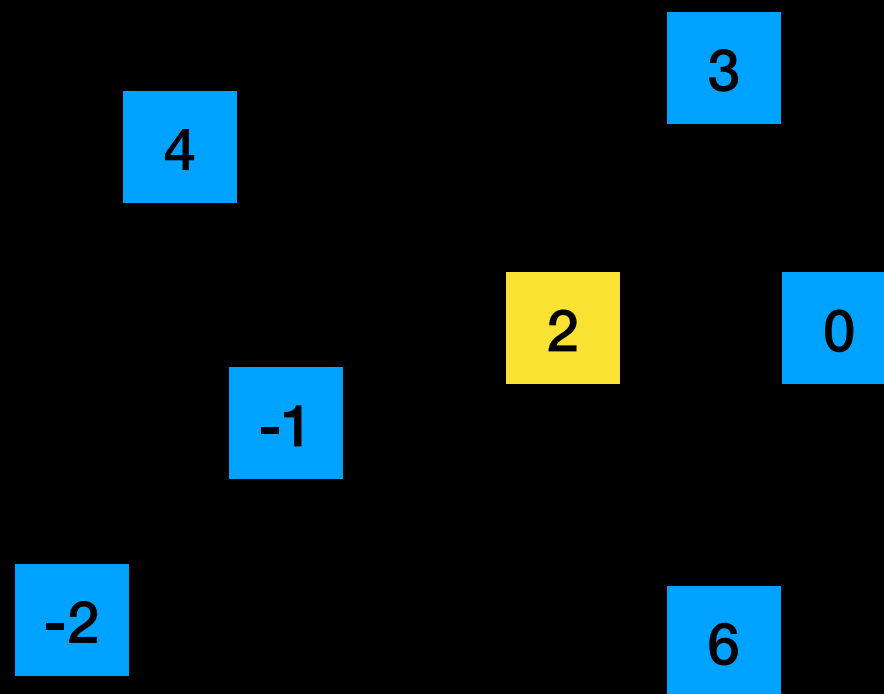
Remove: Find element and if there remove it $O(n)$

Add: Check if element is there and if not add it $O(n)$

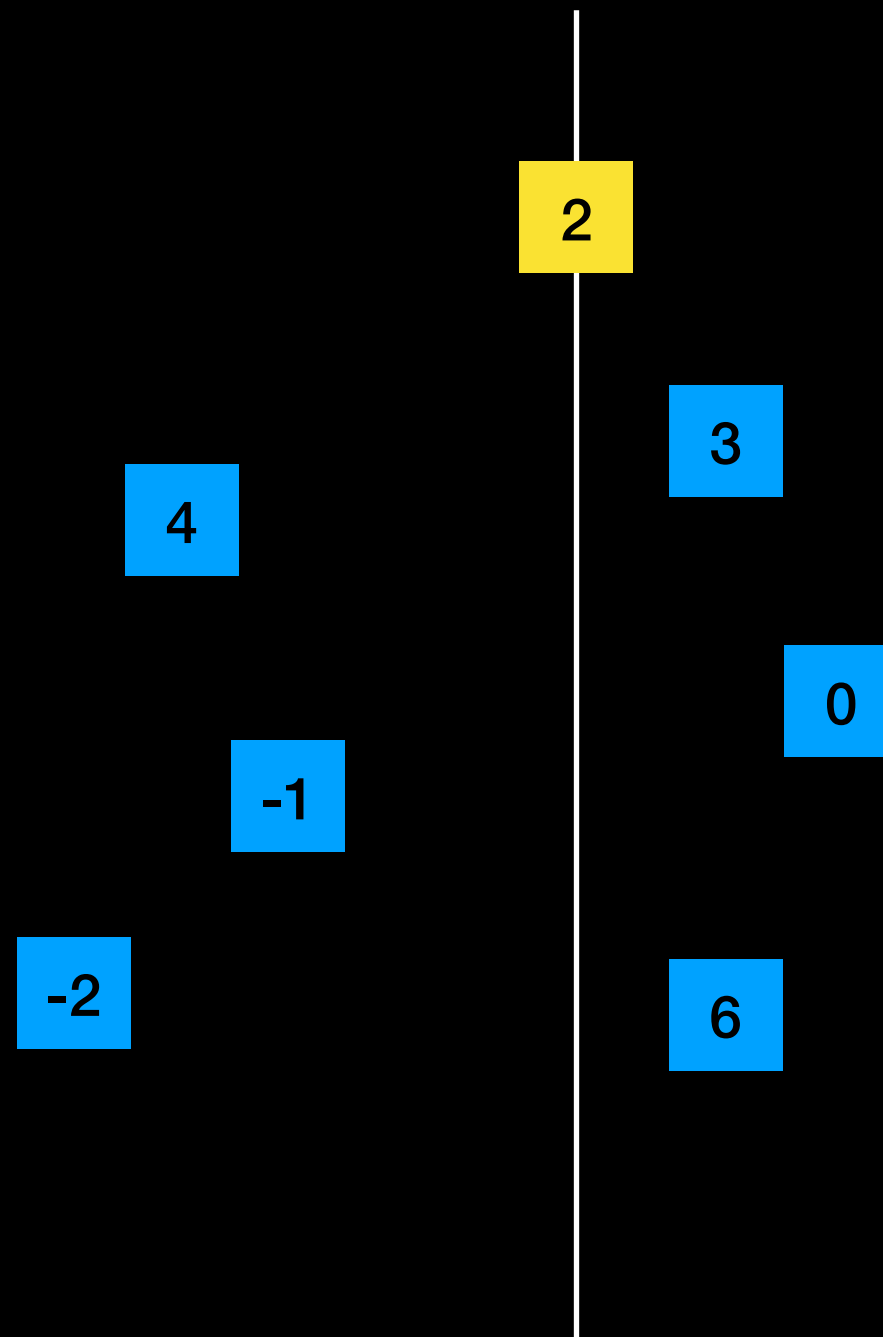
A Different Approach



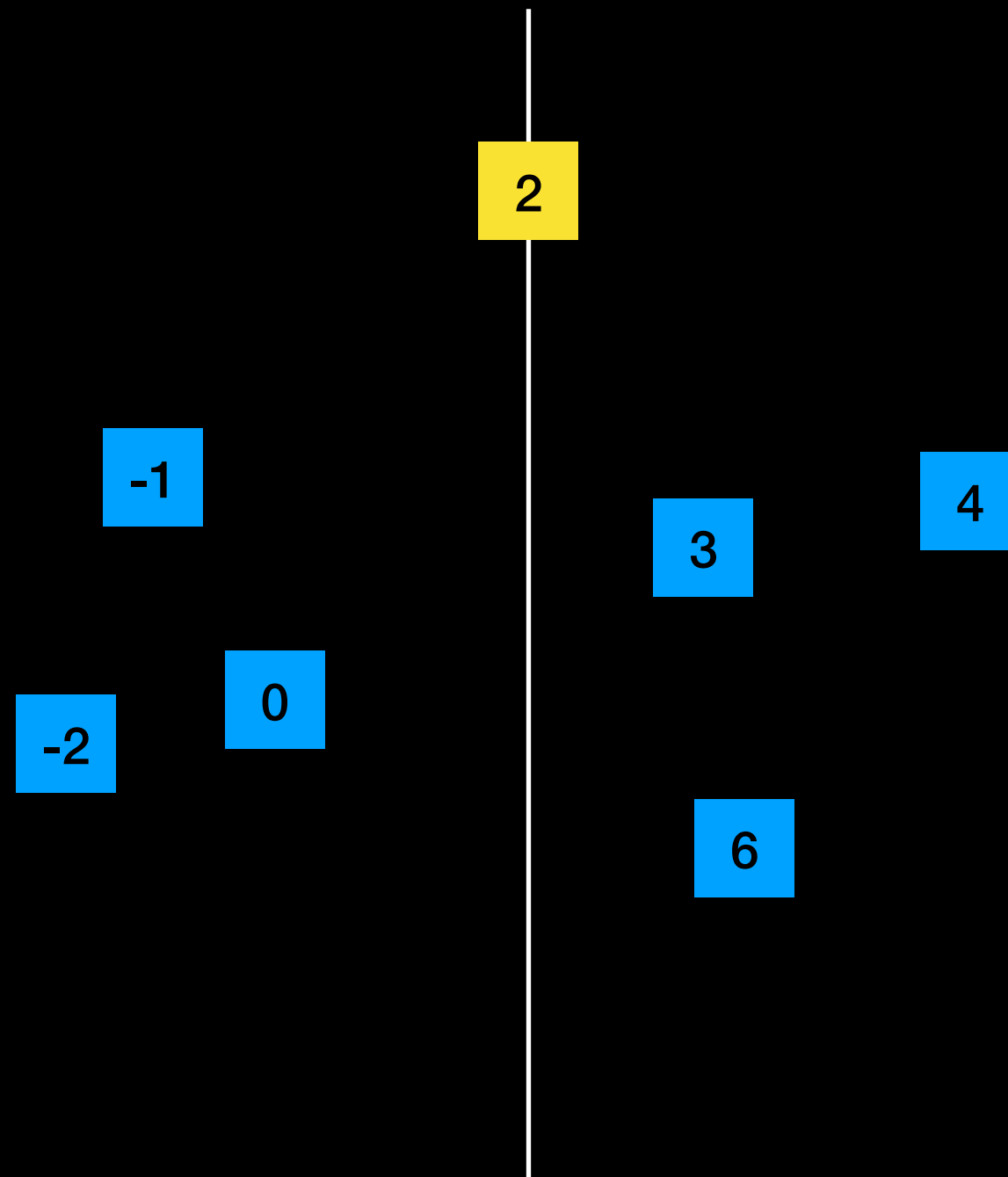
A Different Approach



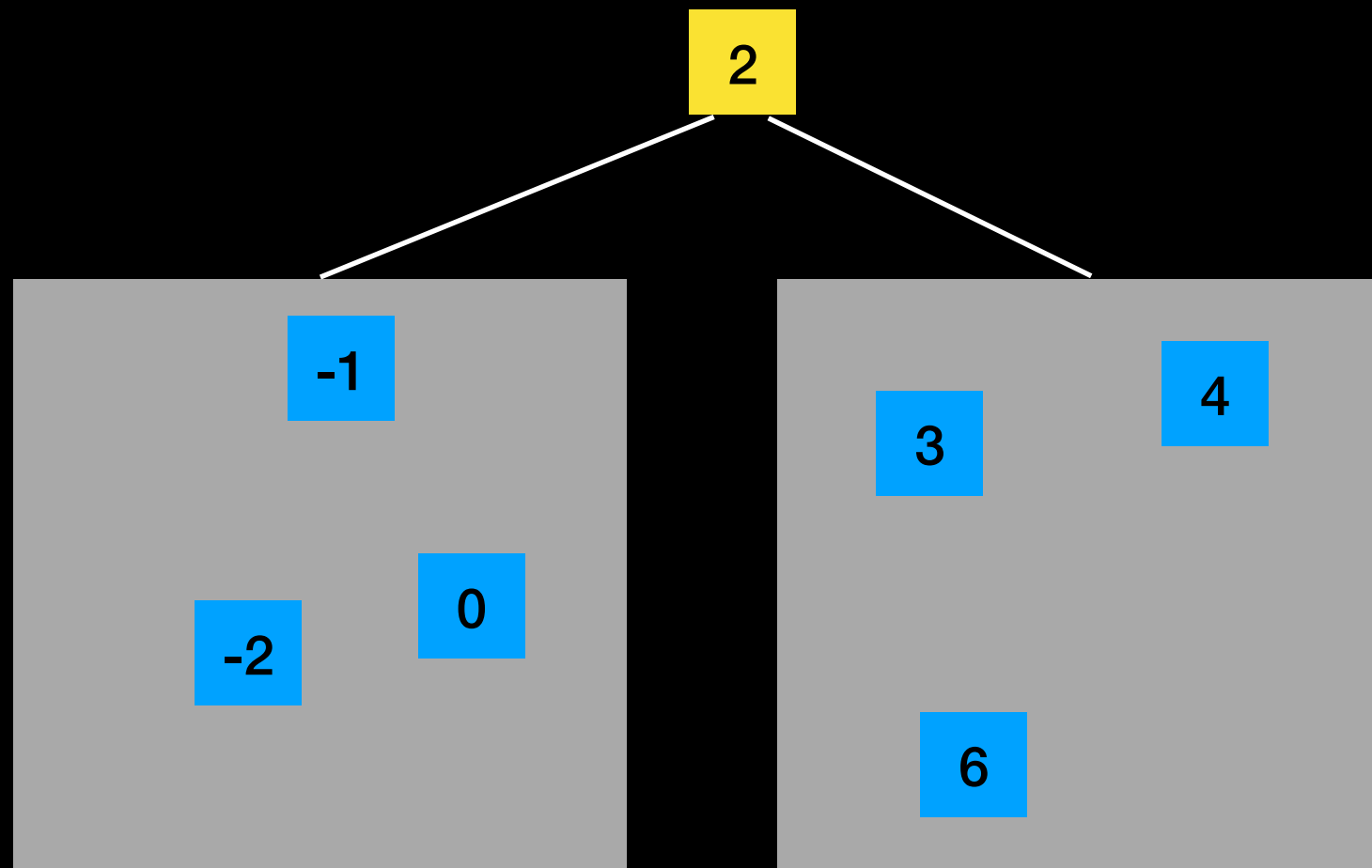
A Different Approach



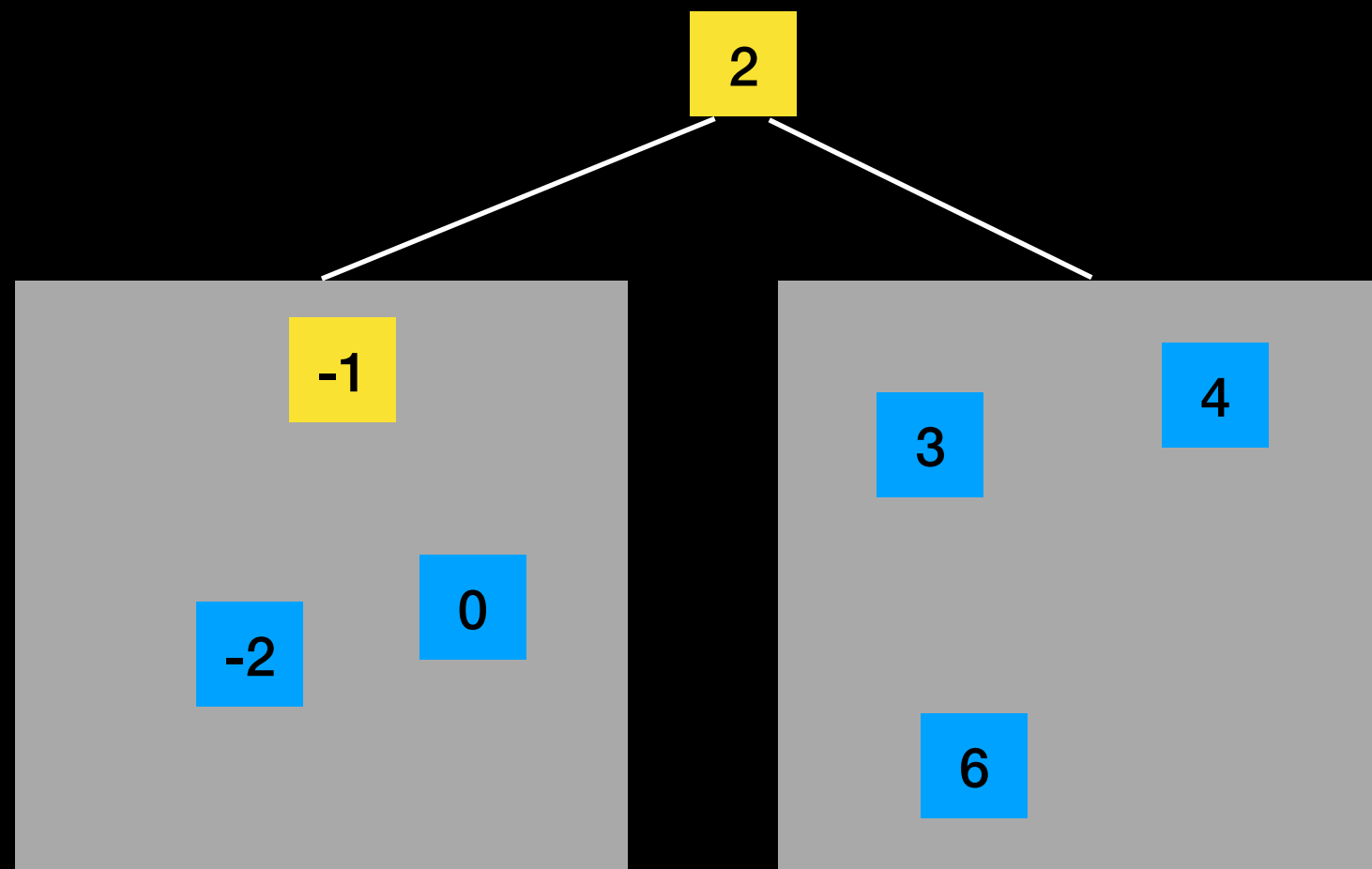
A Different Approach



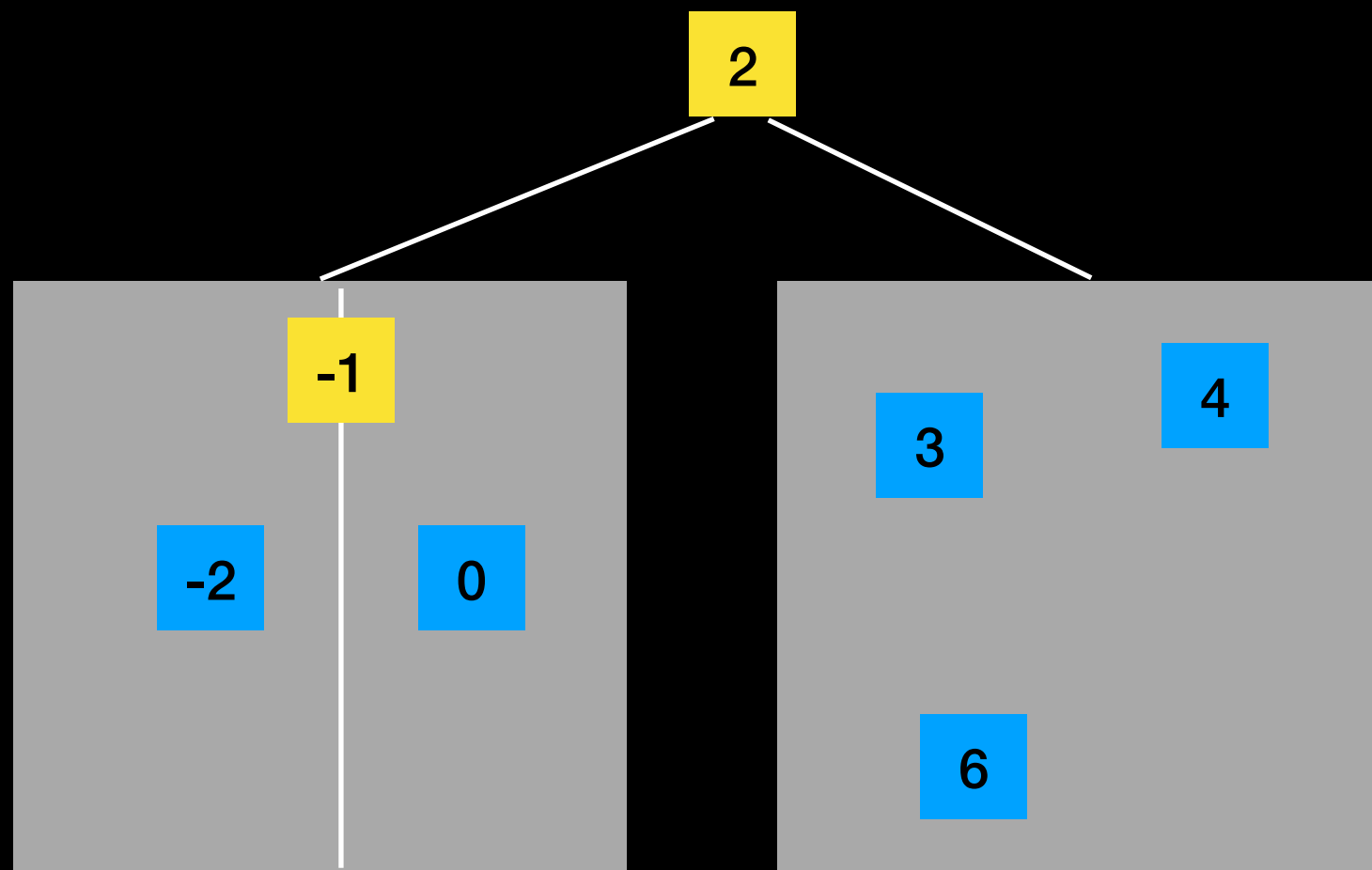
A Different Approach



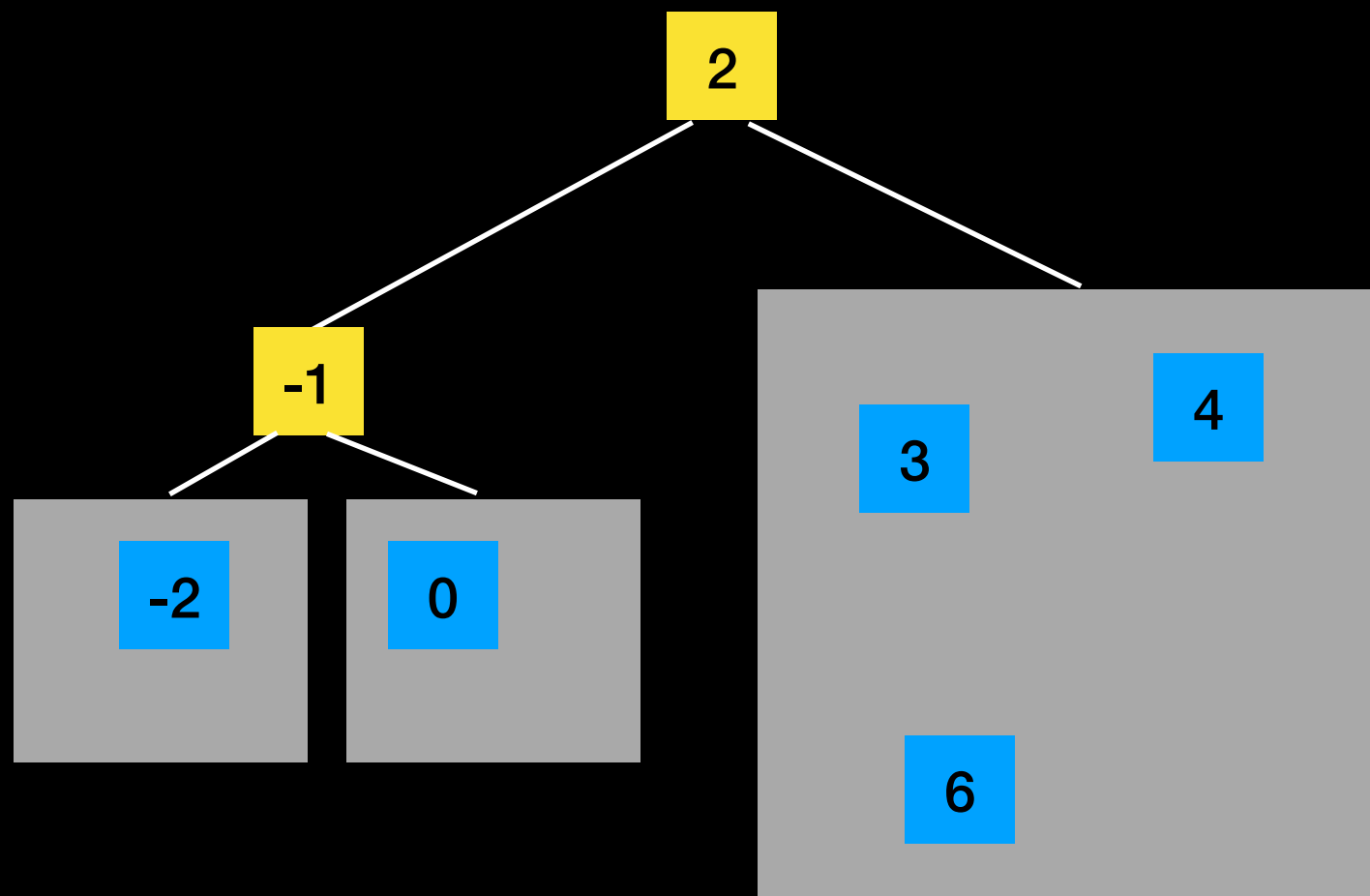
A Different Approach



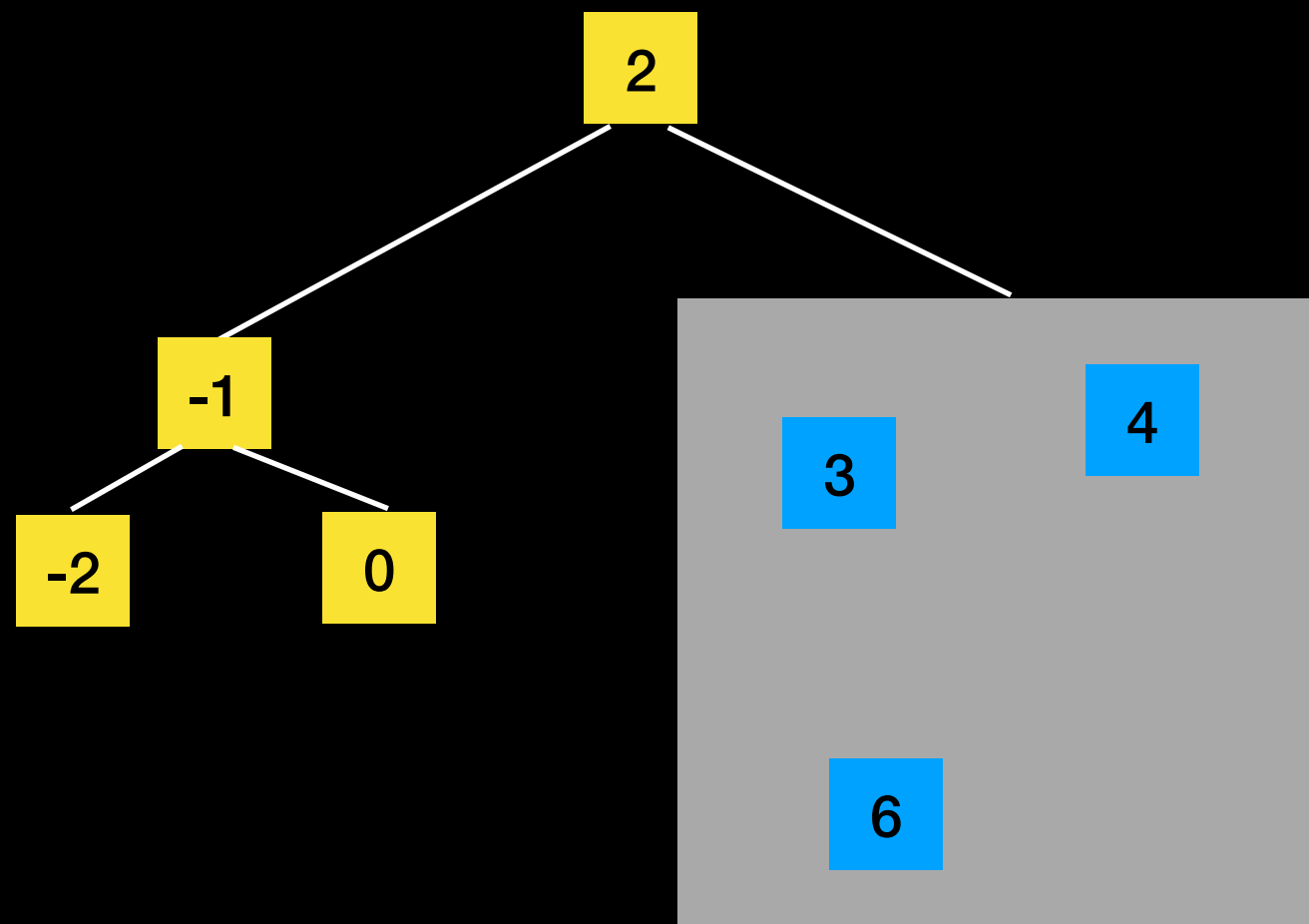
A Different Approach



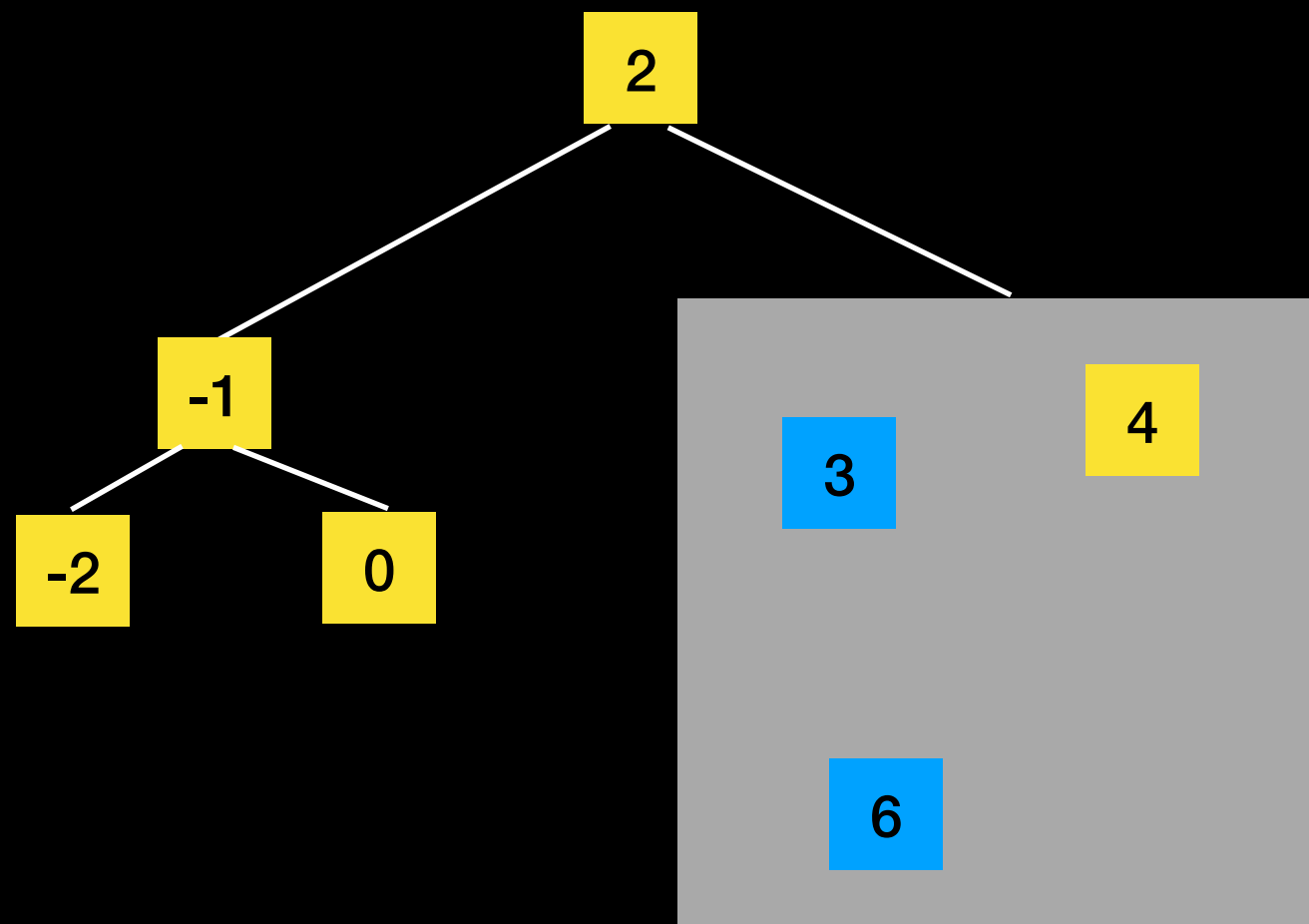
A Different Approach



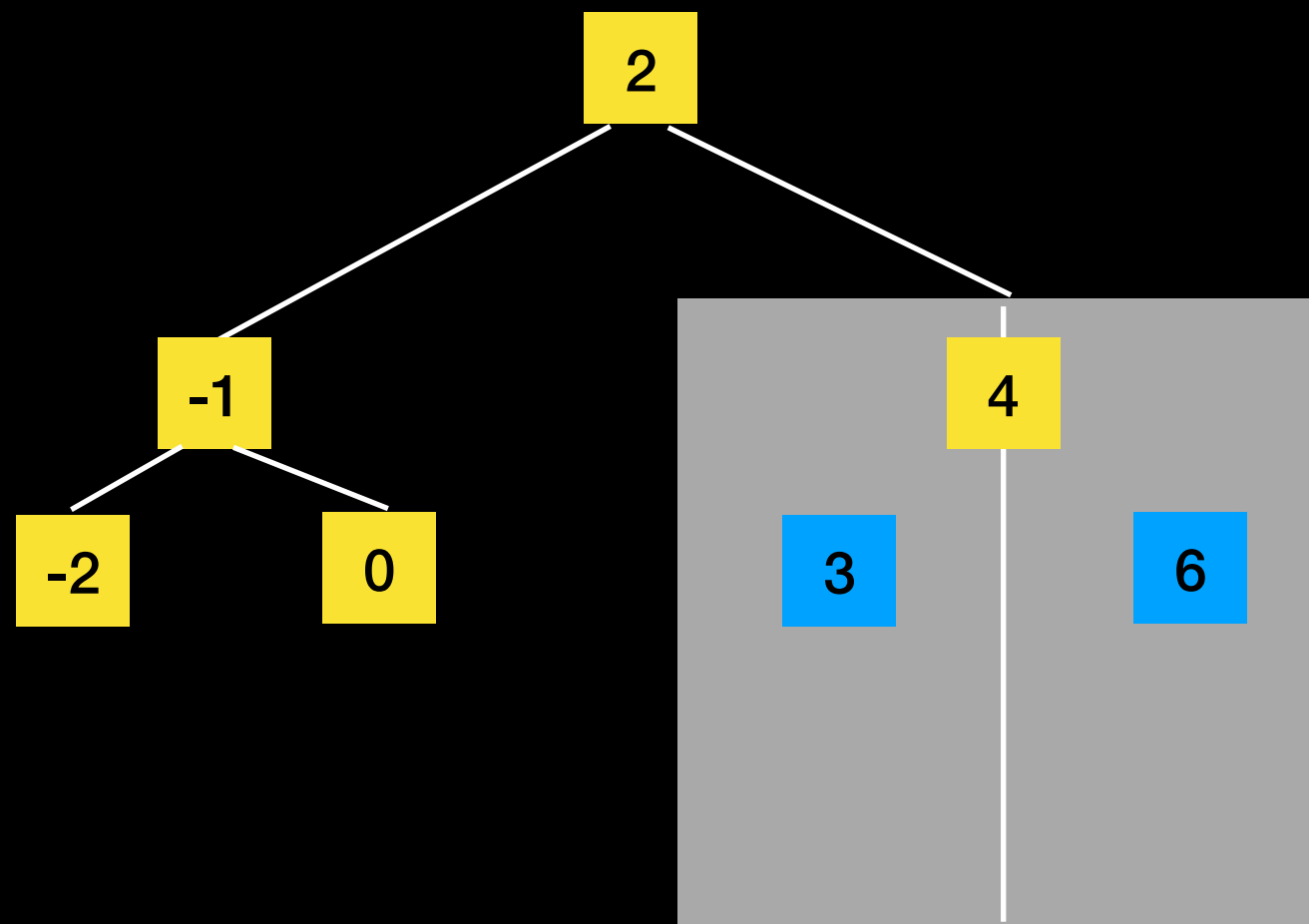
A Different Approach



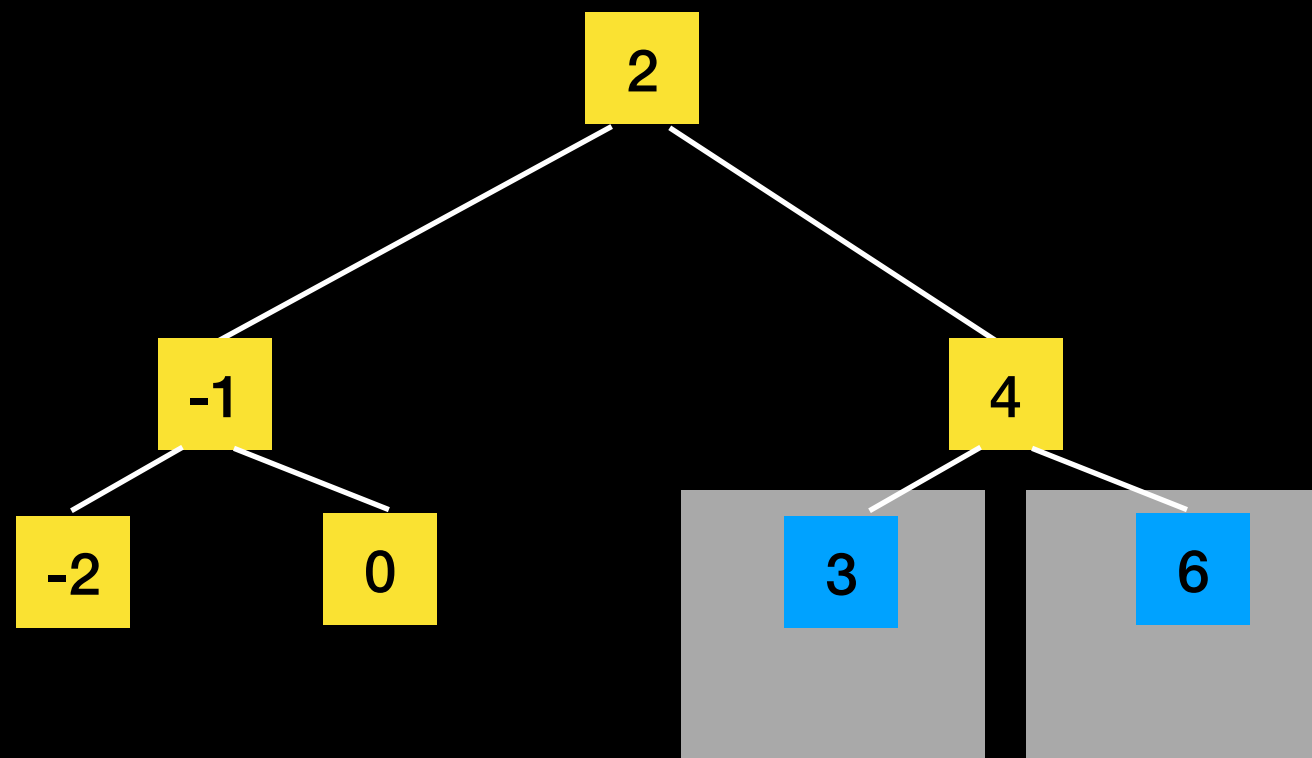
A Different Approach



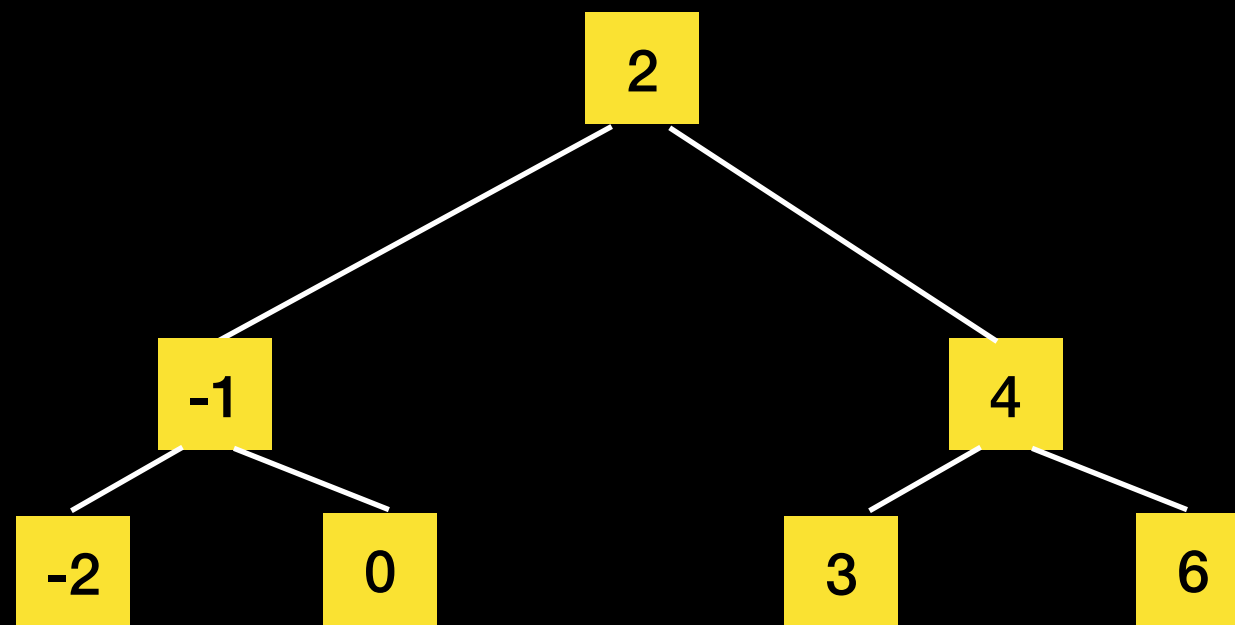
A Different Approach



A Different Approach

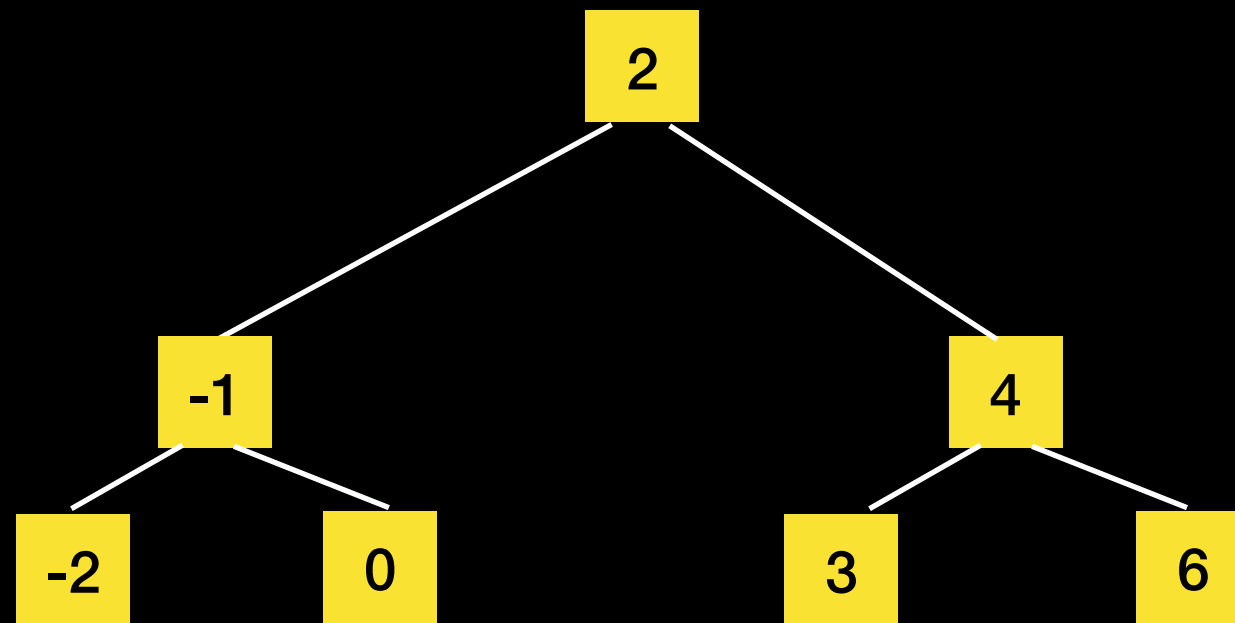


A Different Approach



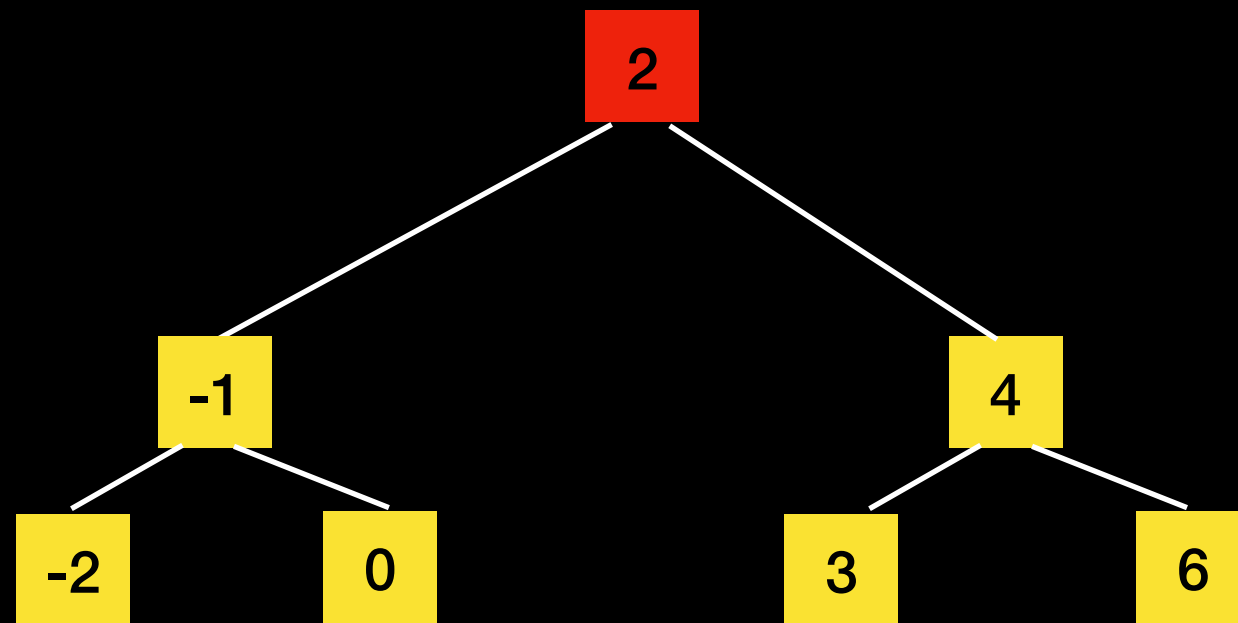
A Different Approach

Find 5



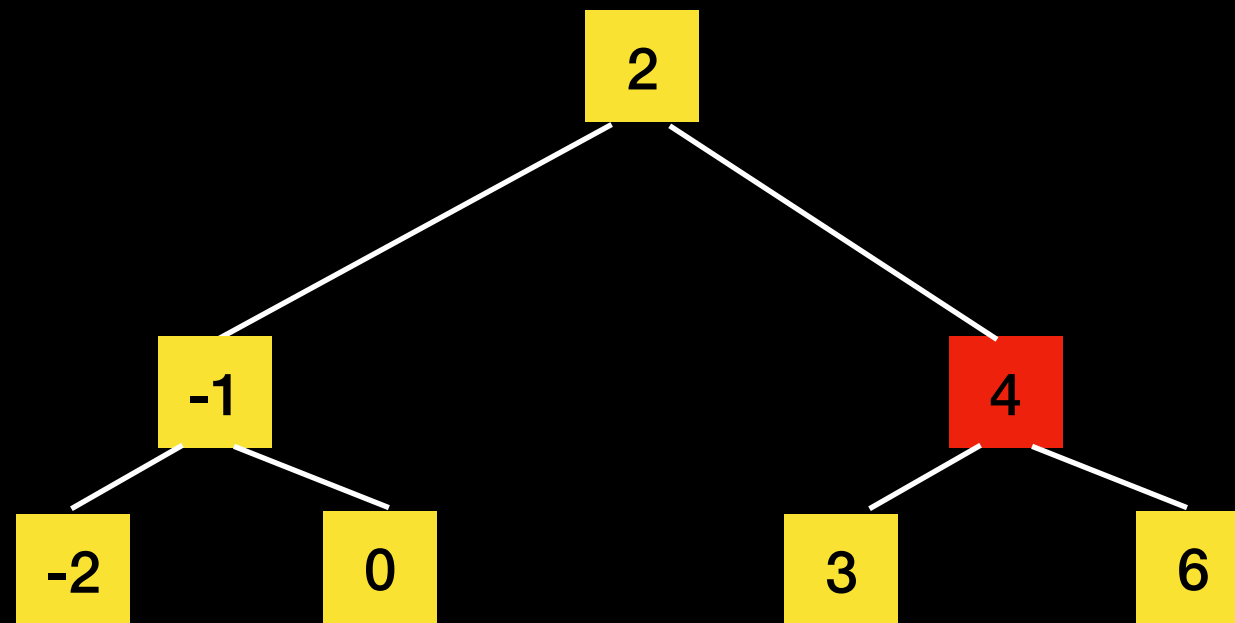
A Different Approach

Find 5



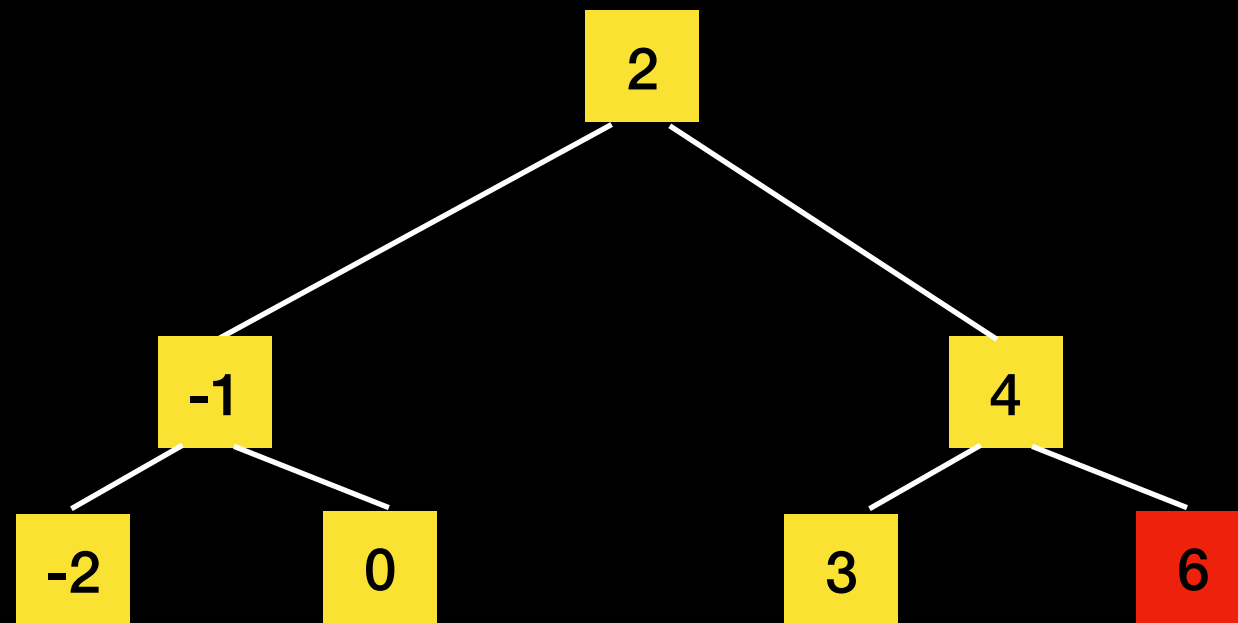
A Different Approach

Find 5



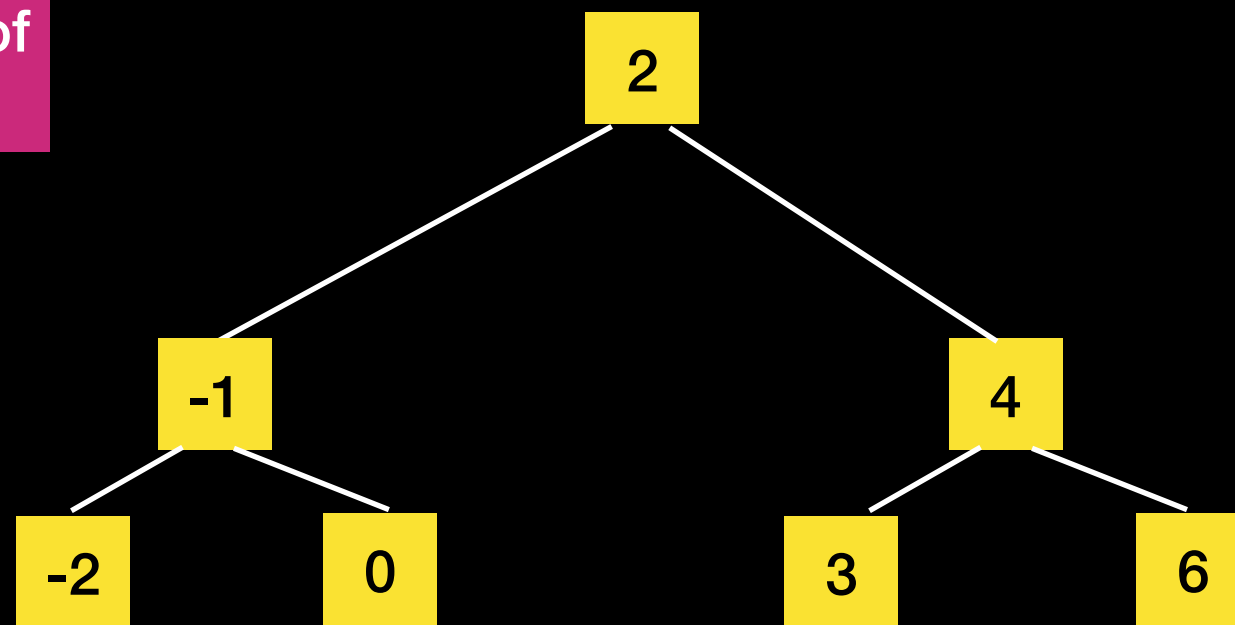
A Different Approach

Find 5



A Different Approach

What's special about the shape of this tree?



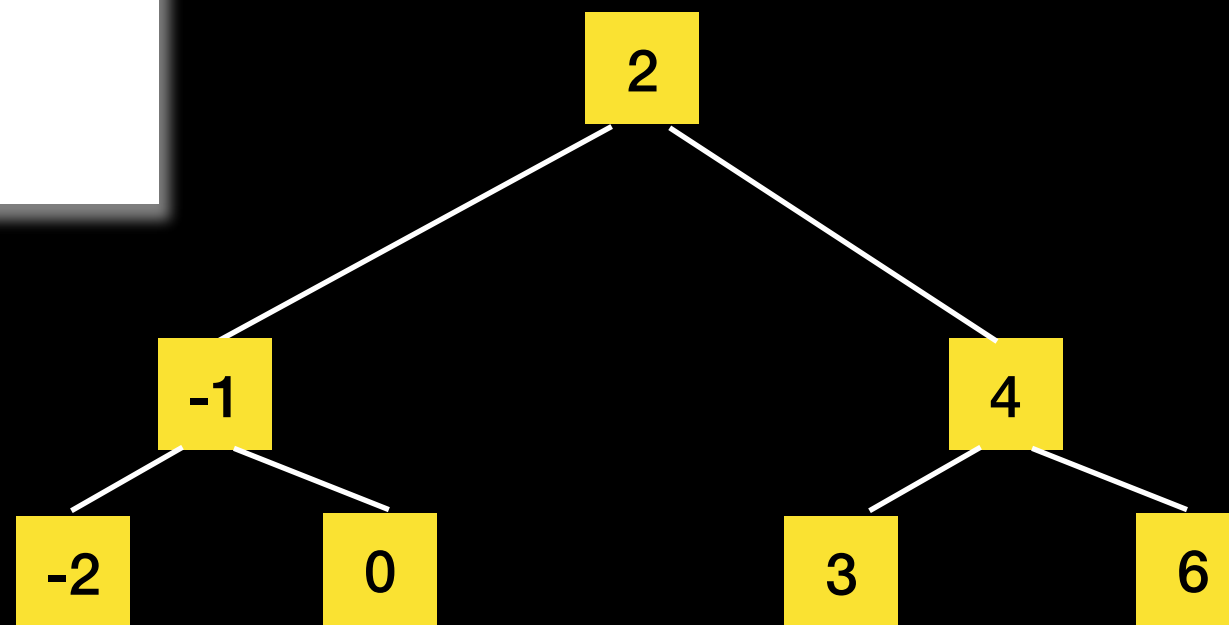
Binary Search Tree

Structural Property:

For each node n

$n > \text{all values in } T_L$

$n < \text{all values in } T_R$



BST Formally

Let S be a set of values upon which a **total ordering relation** $<$, is defined. For example, S can be the set of integers.

A **binary search tree** (**BST**) T for the ordered set $(S, <)$ is a binary tree with the following properties:

- Each node of T has a value. If p and q are **nodes**, then we write $p < q$ to mean that the value of p is less than the value of q .
- For each node $n \in T$, if p is a node in the left subtree of n , then $p < n$.
- For each node $n \in T$, if p is a node in the right subtree of n , then $n < p$.
- For each element $s \in S$ there exists a node $n \in T$ such that $s = n$.

Binary Search Tree

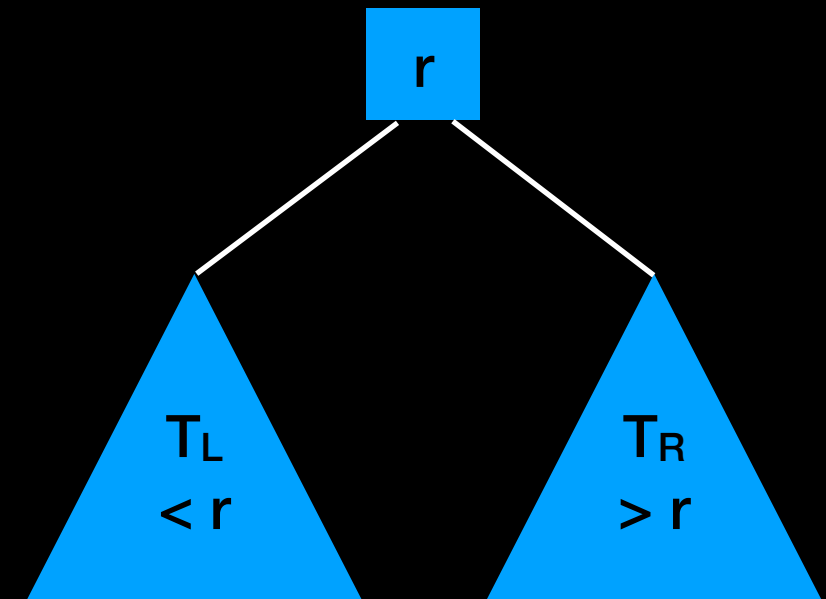
Structural Property:

For each node n

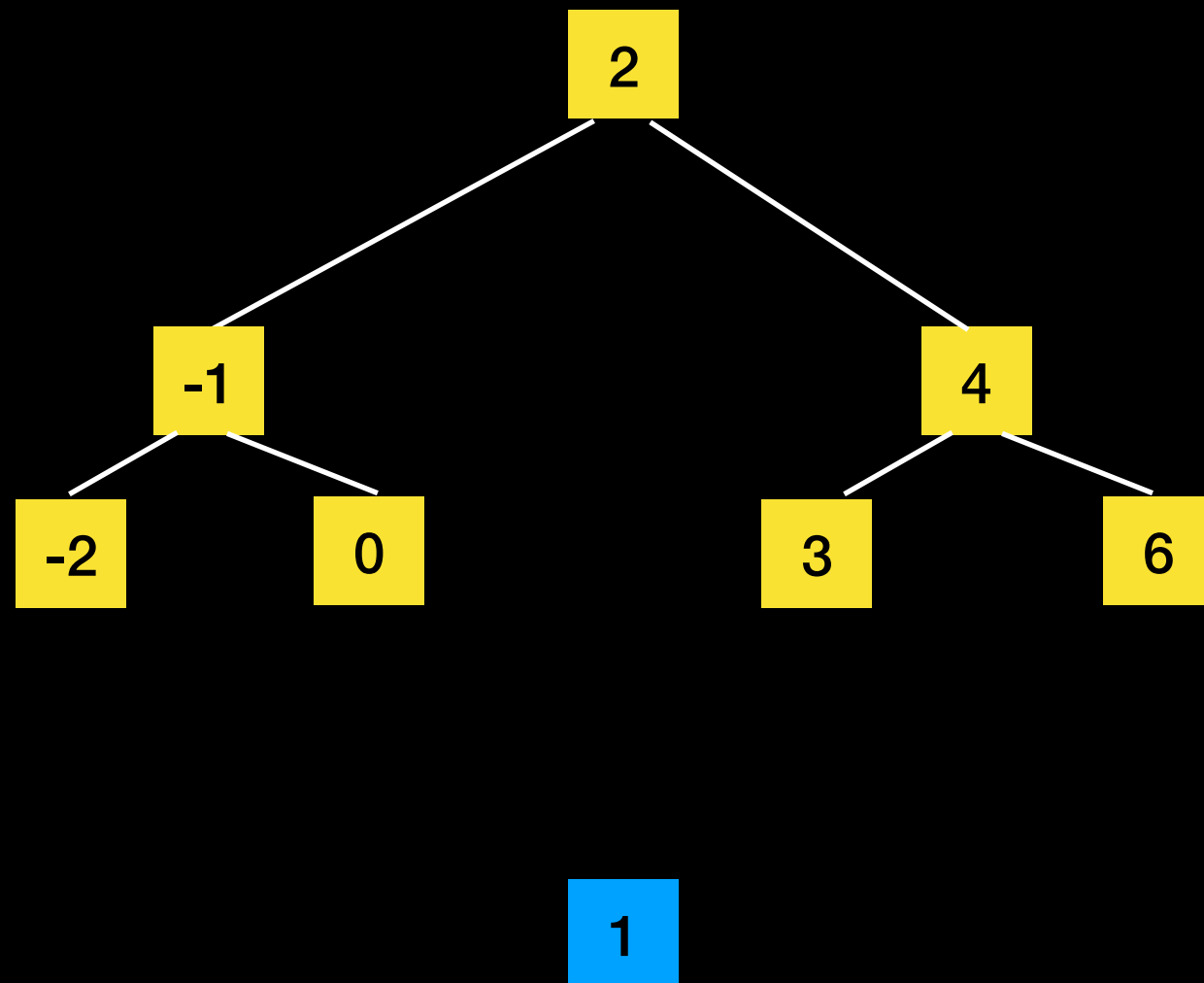
$n >$ all values in T_L

$n <$ all values in T_R

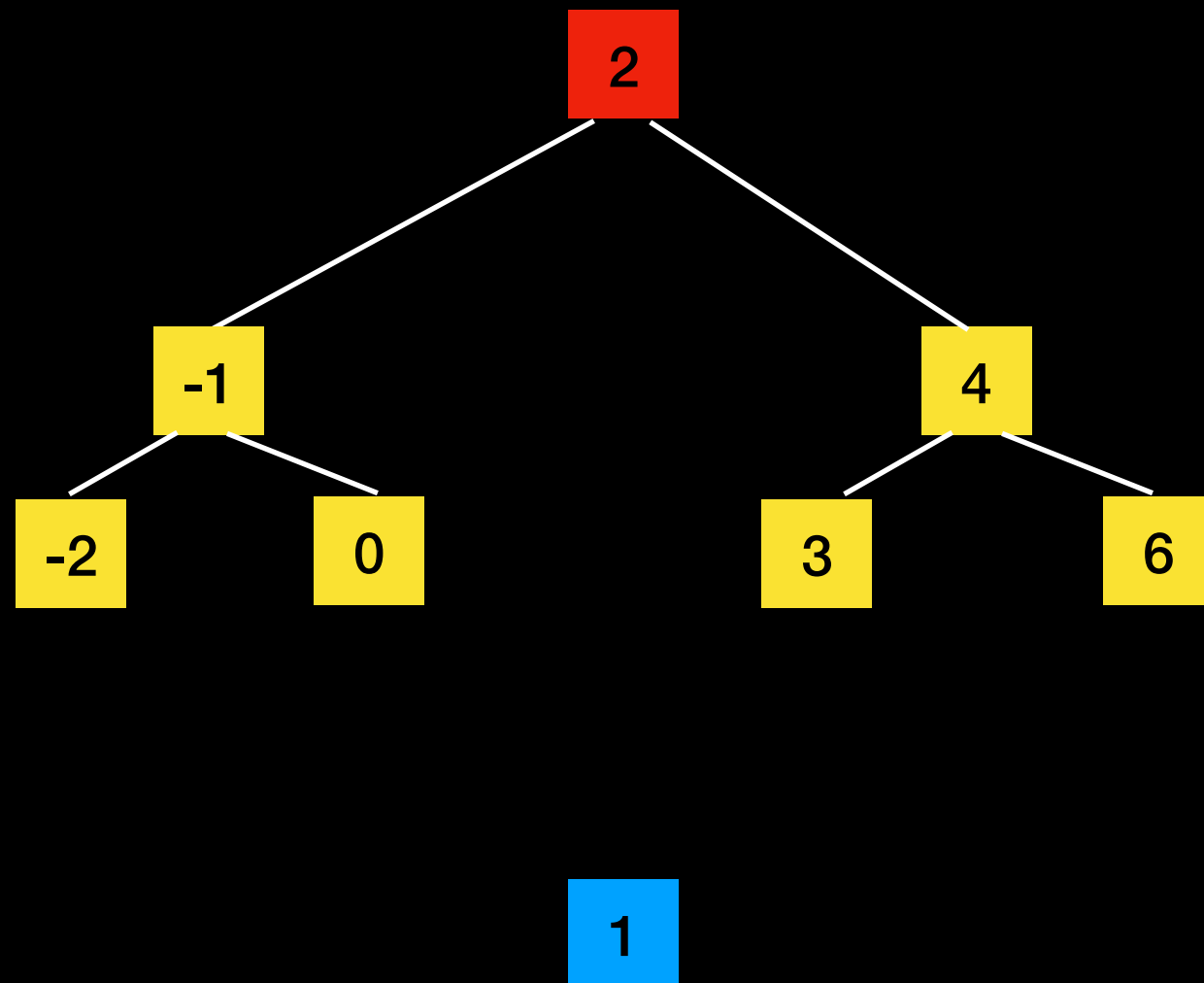
```
search(bs_tree, item)
{
    if (bs_tree is empty) //base case
        item not found
    else if (item == root)
        return root
    else if (item < root)
        search( $T_L$ , item)
    else // item > root
        search( $T_R$ , item)
}
```



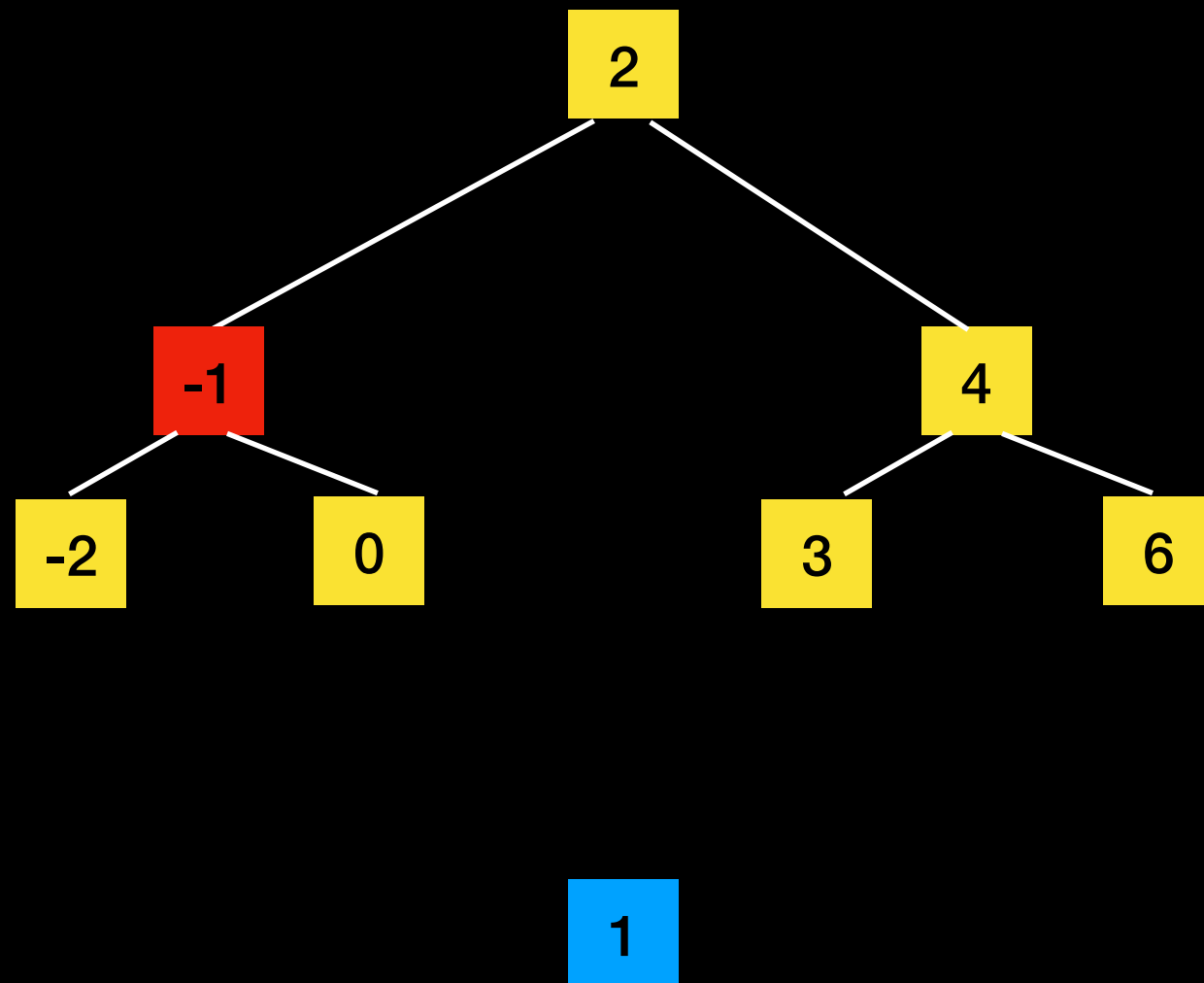
Inserting into a BST



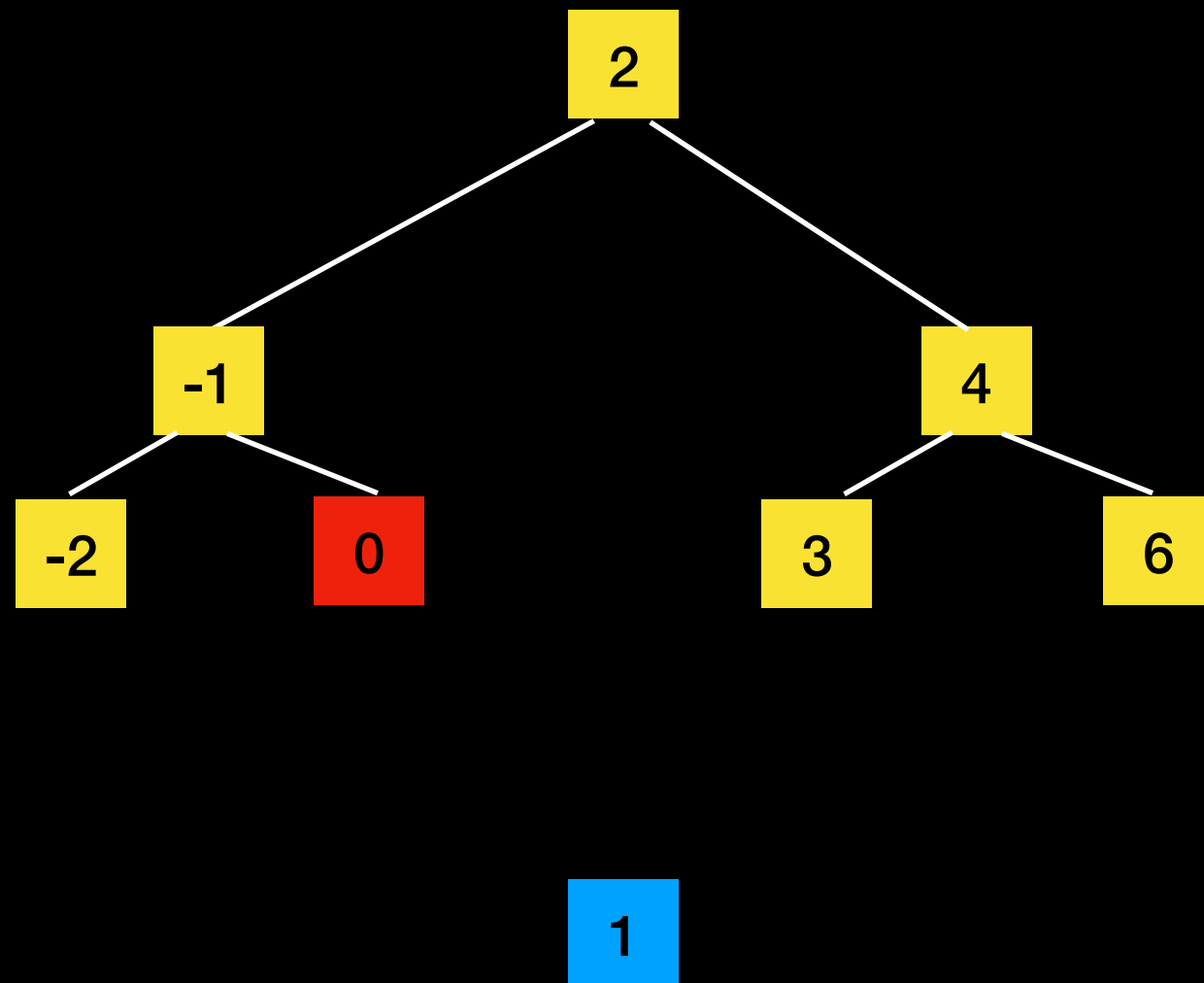
Inserting into a BST



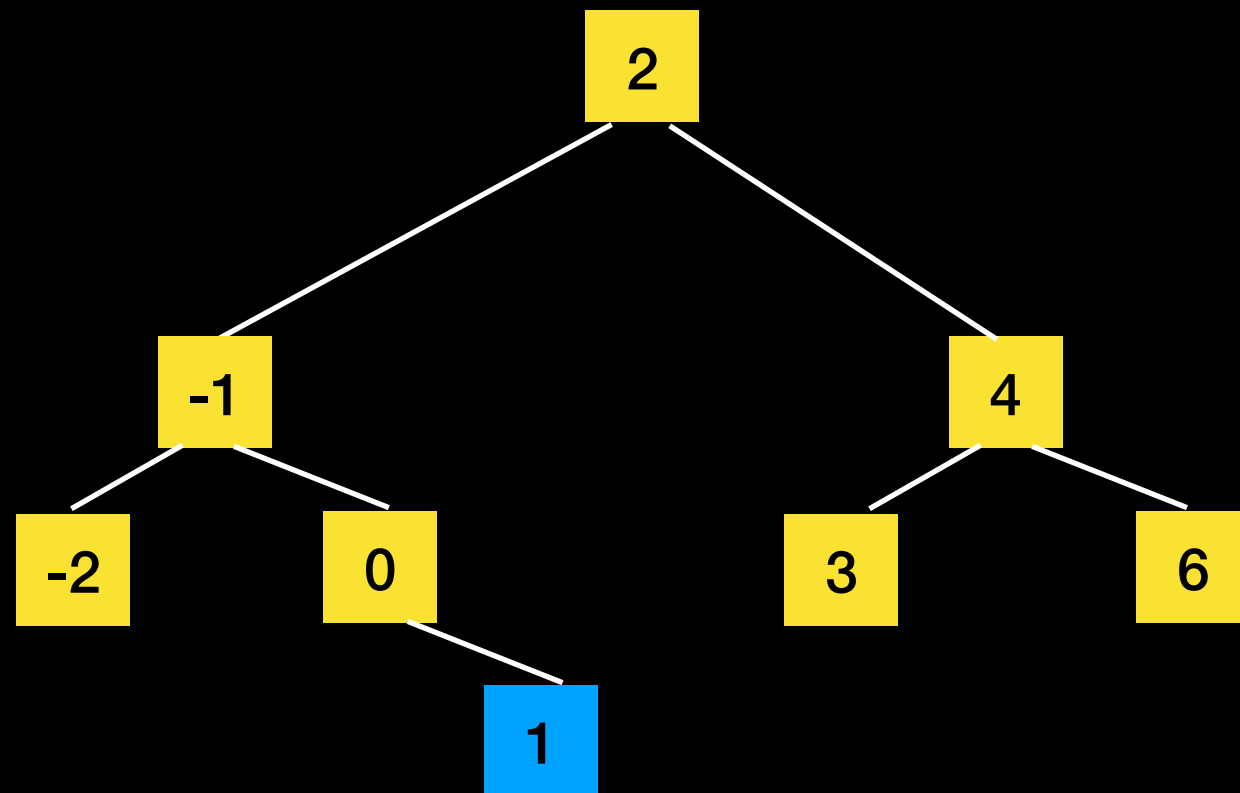
Inserting into a BST



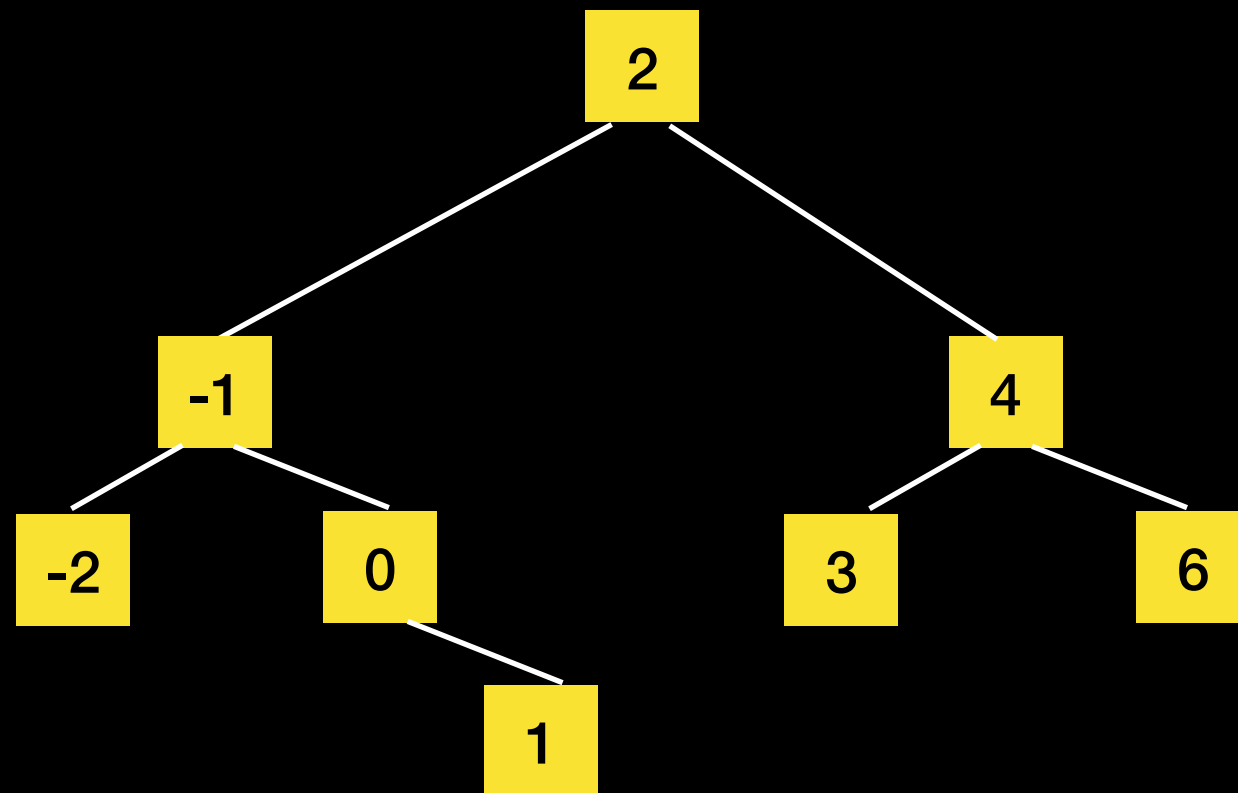
Inserting into a BST



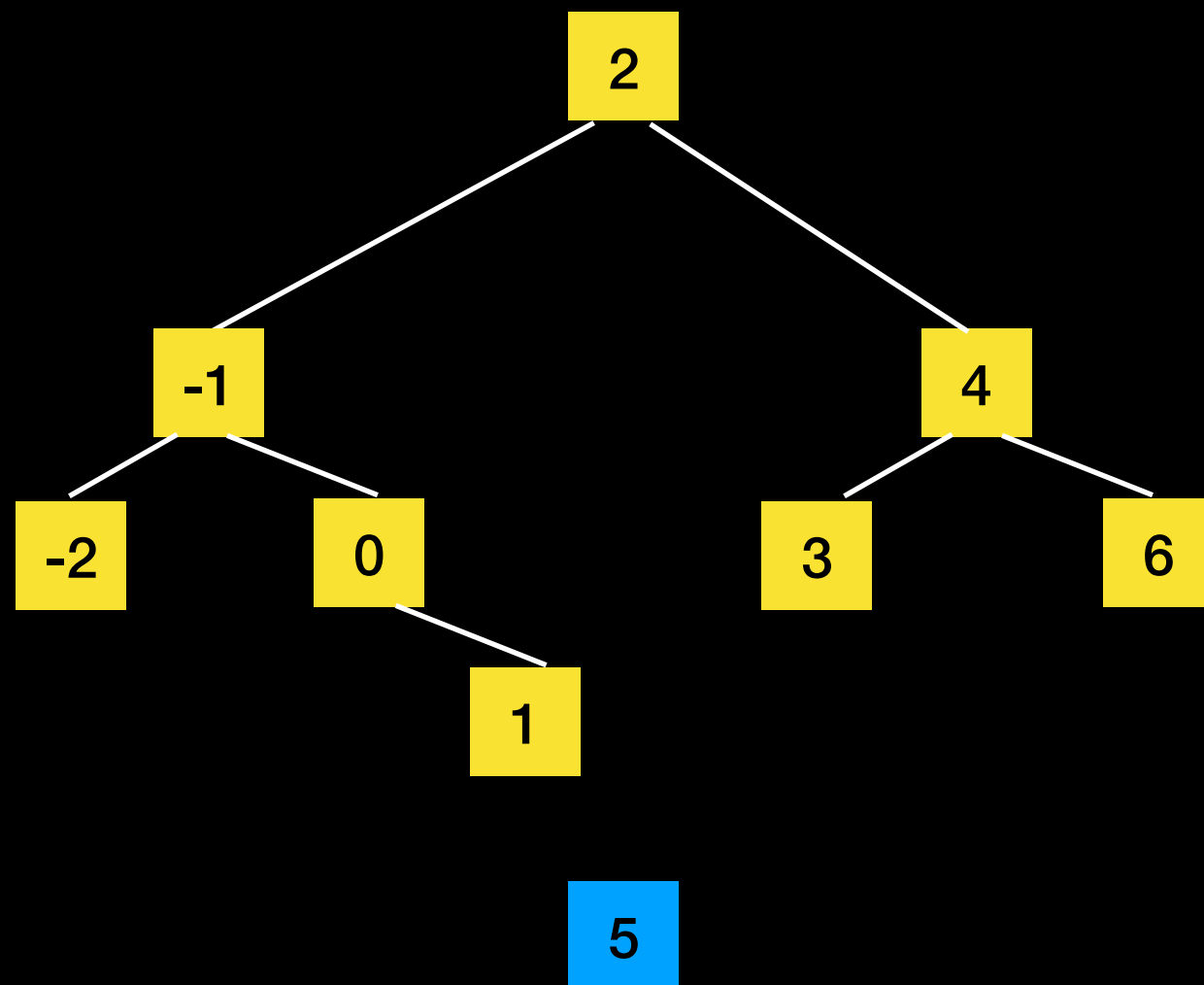
Inserting into a BST



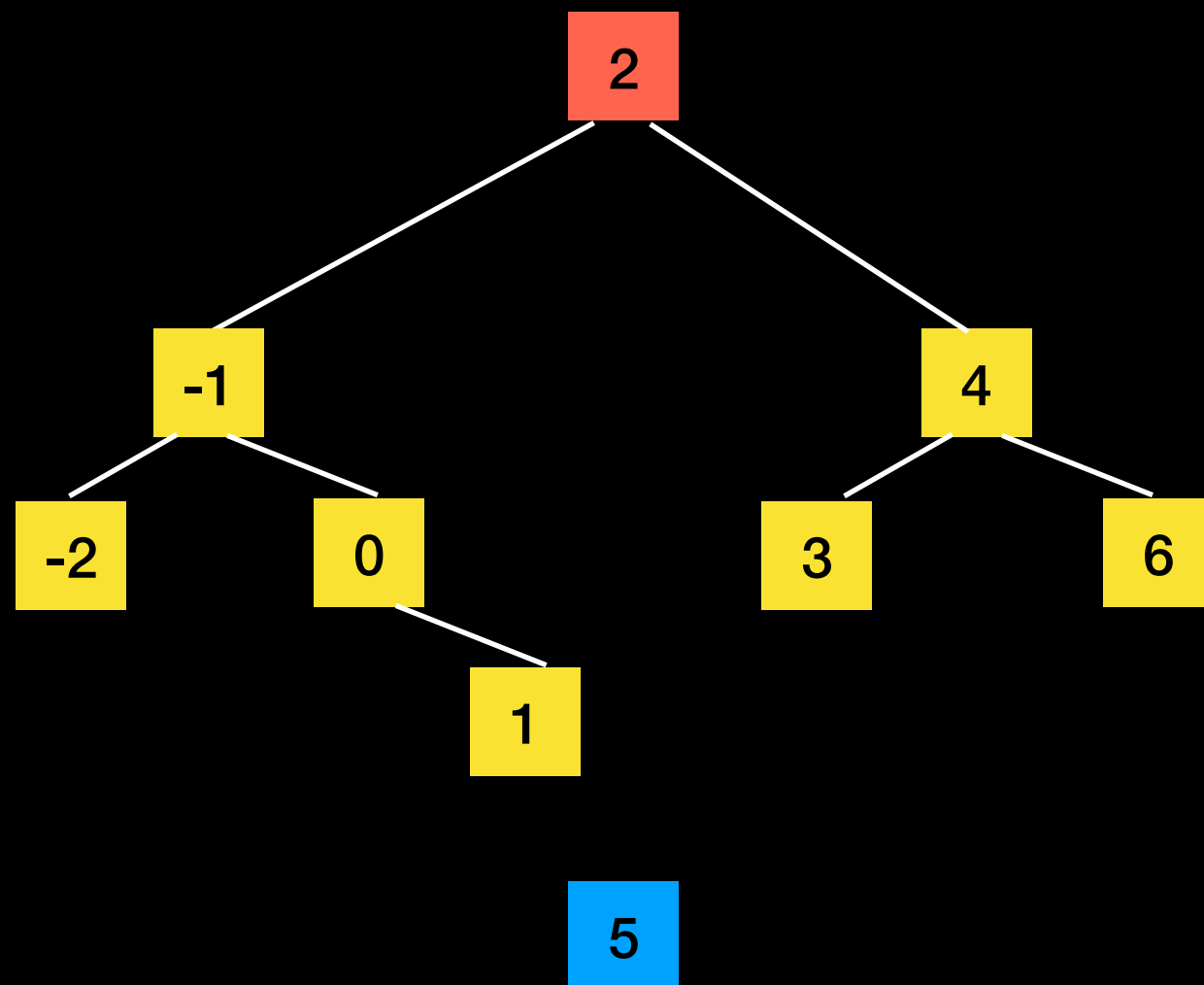
Inserting into a BST



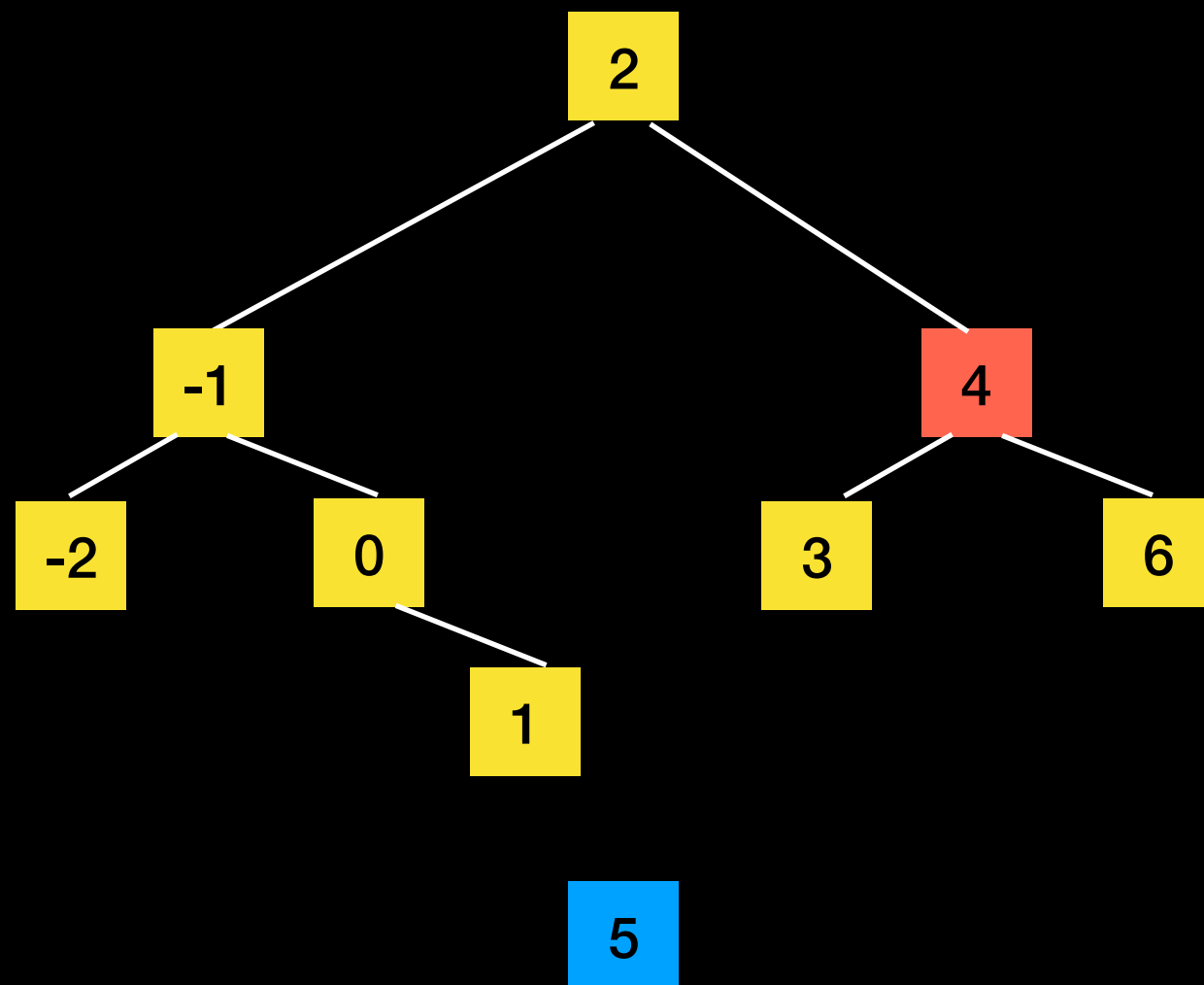
Inserting into a BST



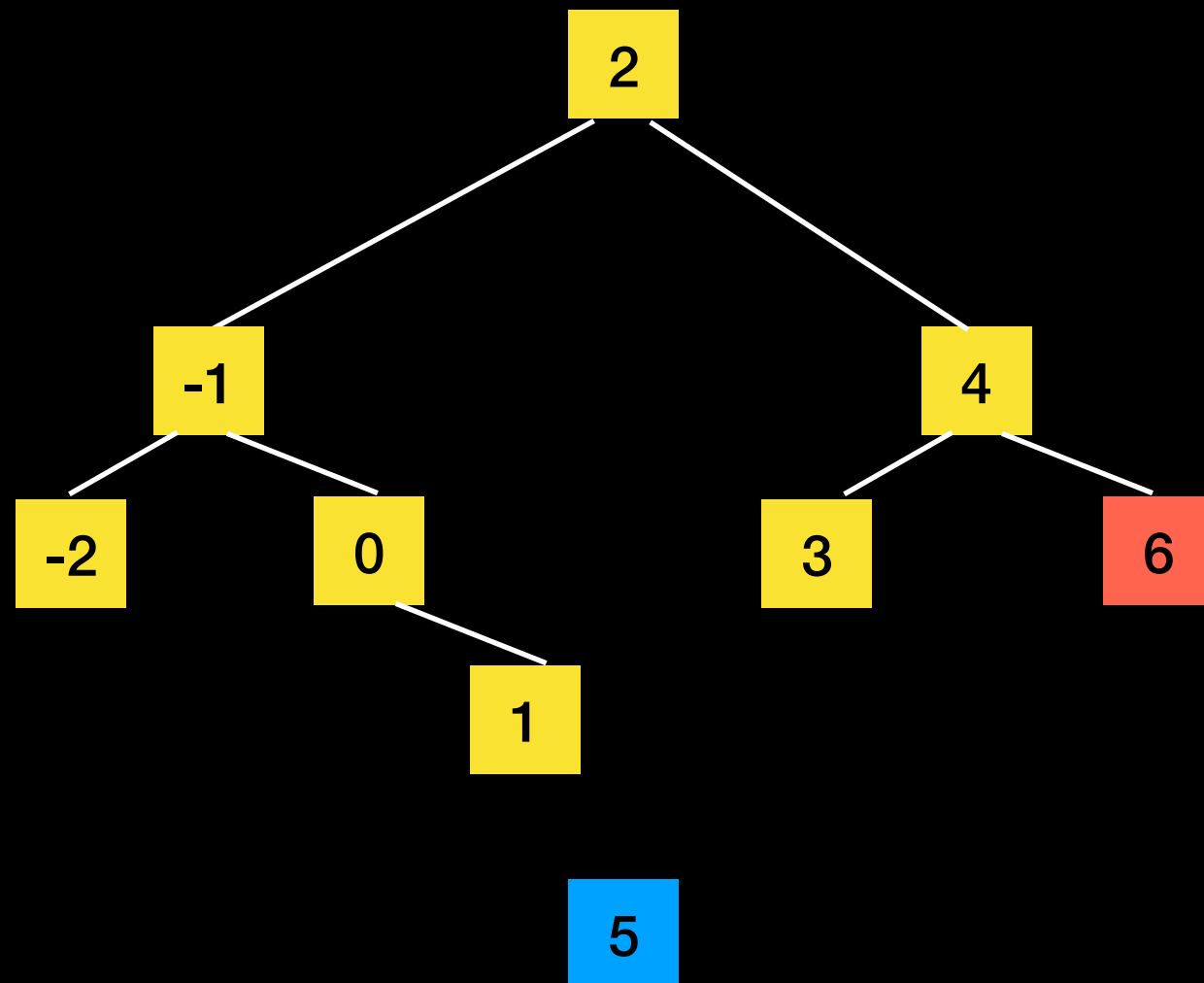
Inserting into a BST



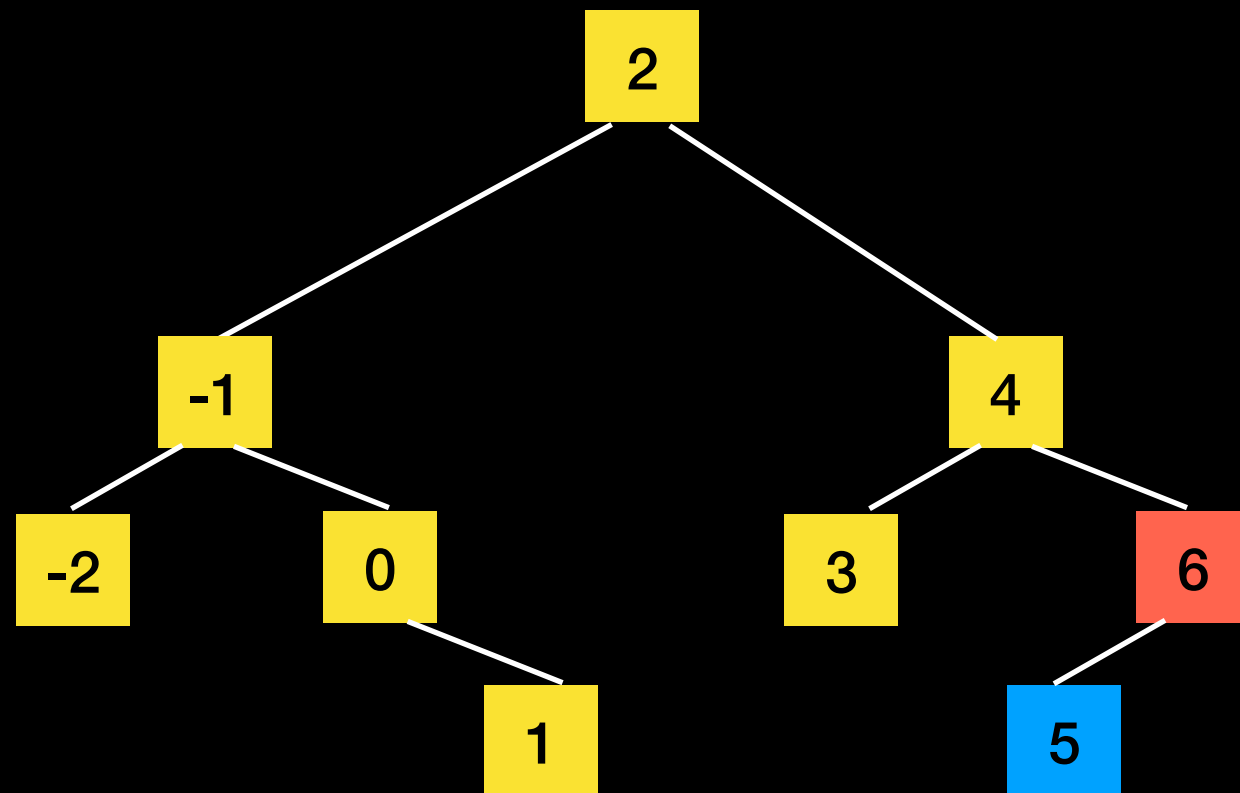
Inserting into a BST



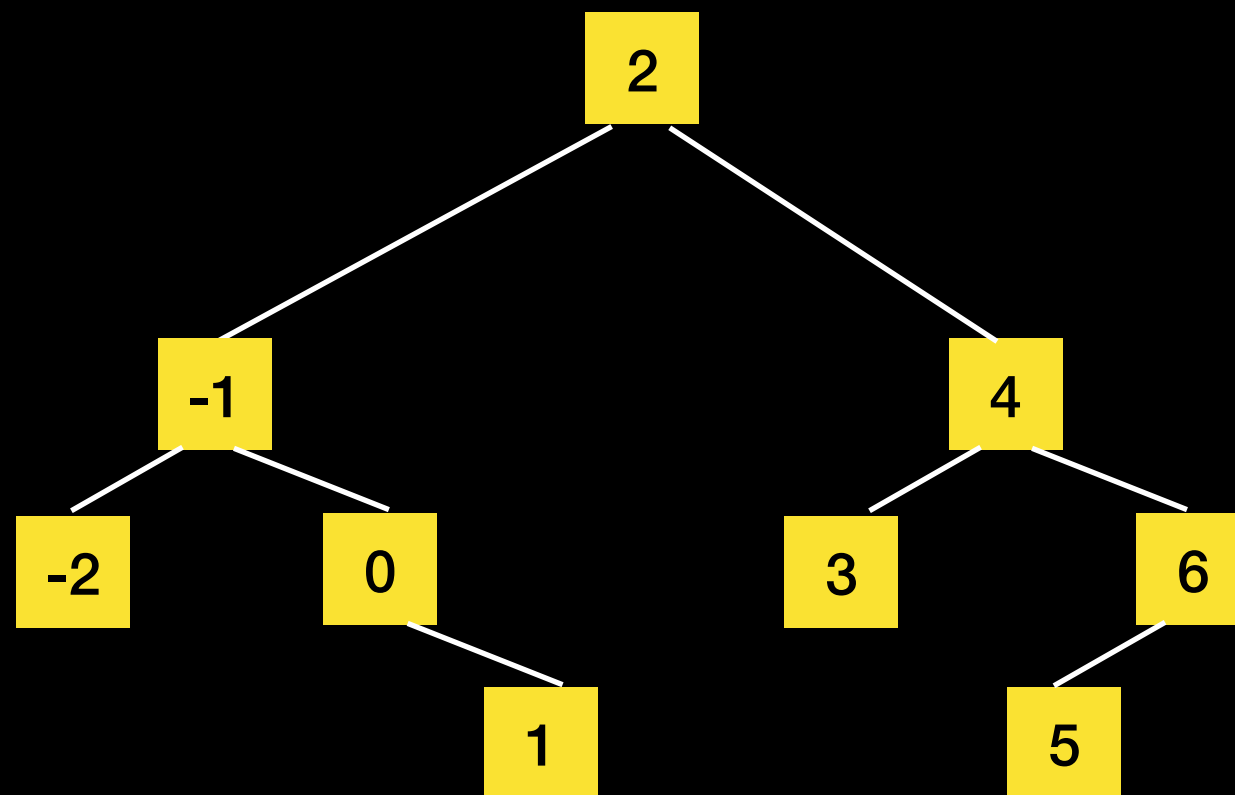
Inserting into a BST



Inserting into a BST

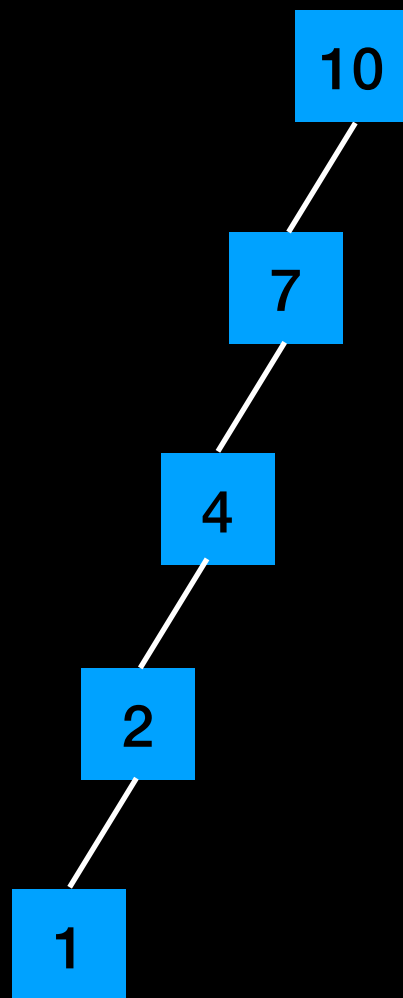


Inserting into a BST

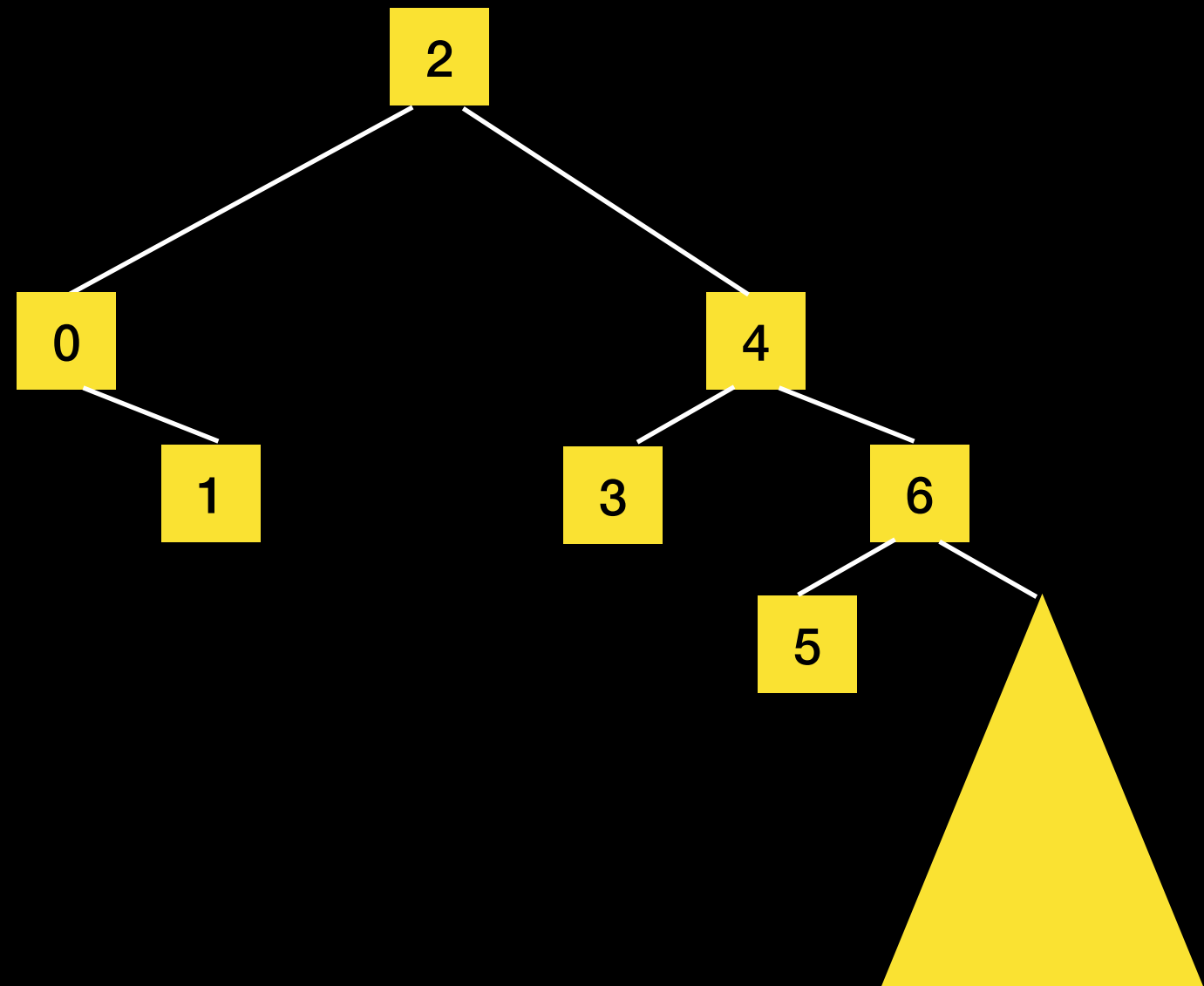
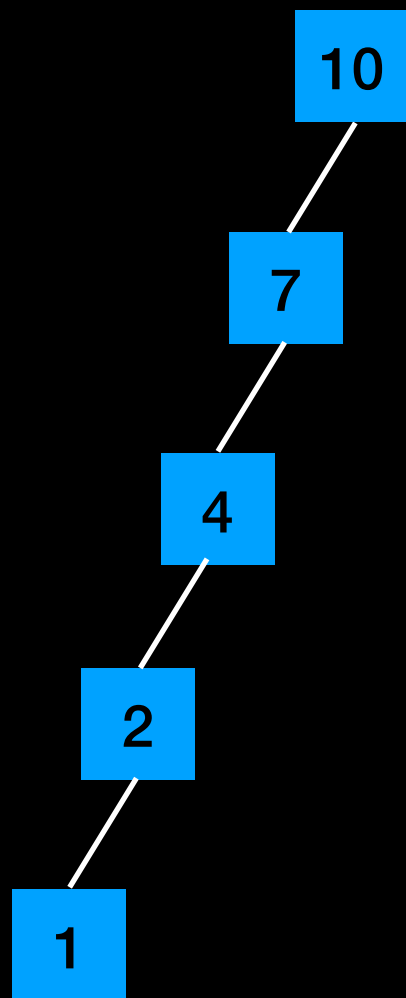


You **Grow** a tree with BST property, you don't get to restructure it
(Self-balancing trees (e.g. Red-Black trees) will do that, perhaps in CSCI 335)

Growing a BST



Growing a BST

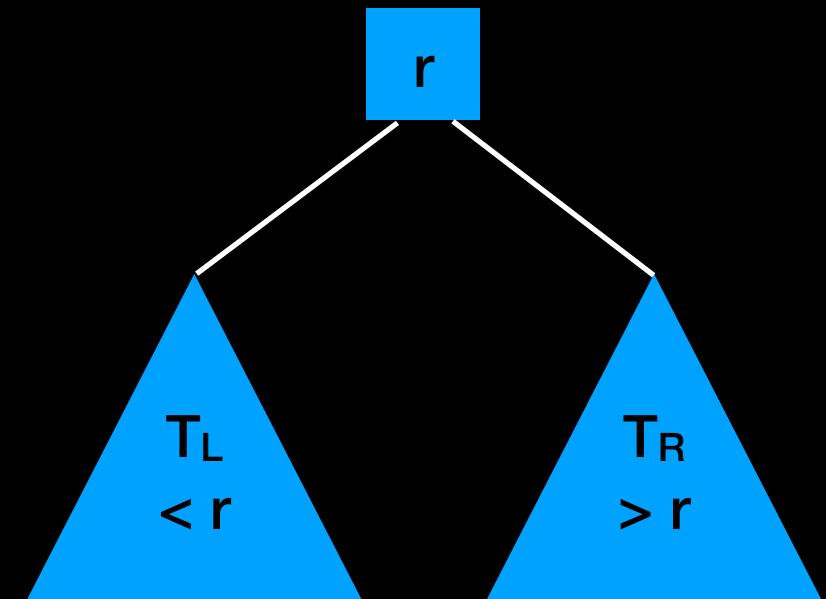


Lecture Activity

Write **pseudocode** to insert an item into a BST

Inserting into a BST

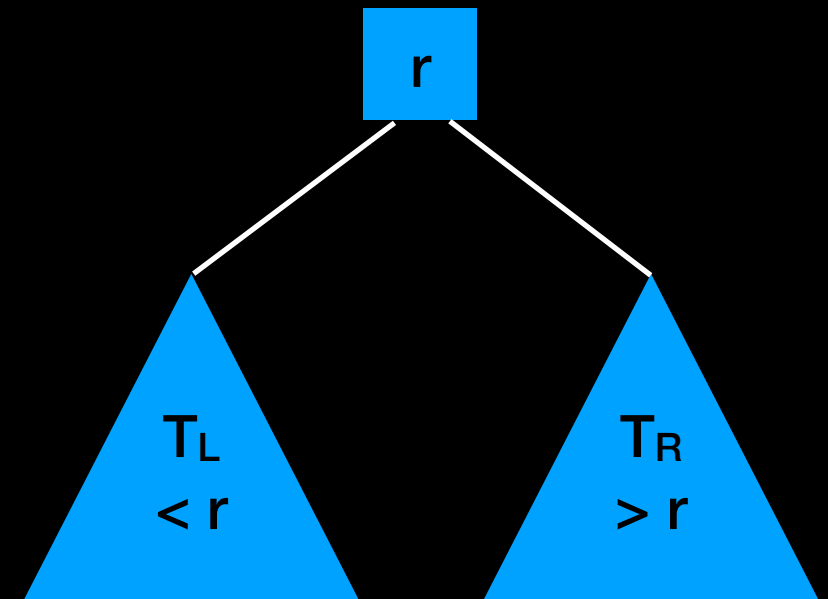
```
add(bs_tree, item)
{
    if (bs_tree is empty) //base case
        make item the root
    else if (item < root)
        add(TL, item)
    else // item > root
        add(TR, item)
}
```



Traversing a BST

Same as traversing
any binary tree

Which type of
traversal is special
for a BST?

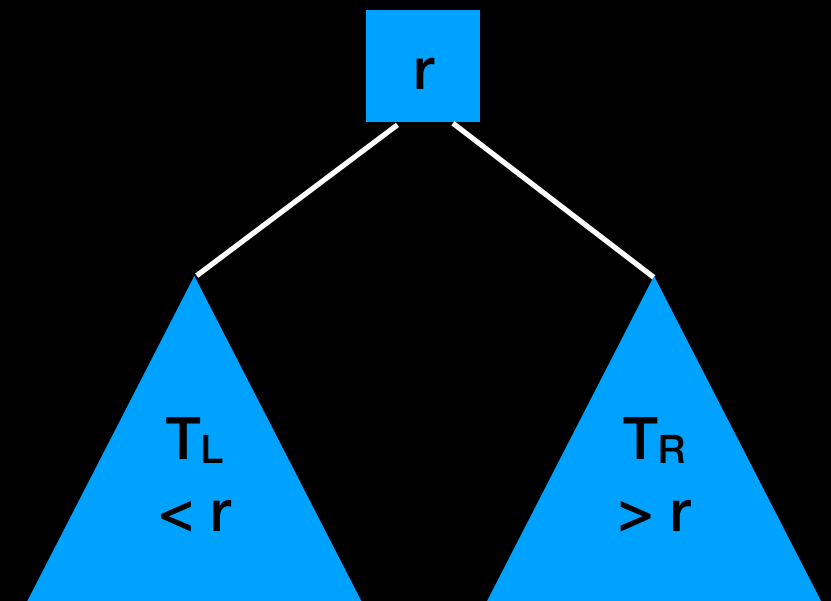


Traversing a BST

Same as traversing
any binary tree

```
inorder(bs_tree)
{
    //implicit base case
    if (bs_tree is not empty)
    {
        inorder(TL)
        visit the root
        inorder(TR)
    }
}
```

Visits nodes in sorted
ascending order



Efficiency of BST

Searching is key to most operations

Think about the structure and height of the tree

Efficiency of BST

Searching is key to most operations

Think about the structure and height of the tree

$O(h)$

What is the maximum height?

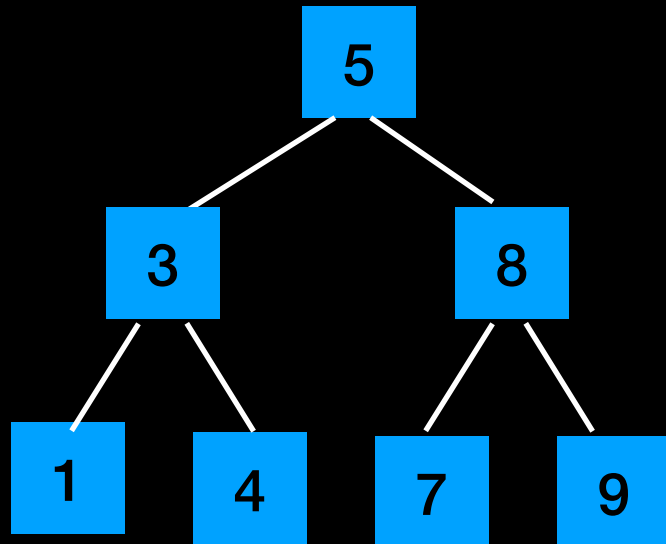
What is the minimum height?

Tree Structure

$n = 7$

$h = 3$

Full BST

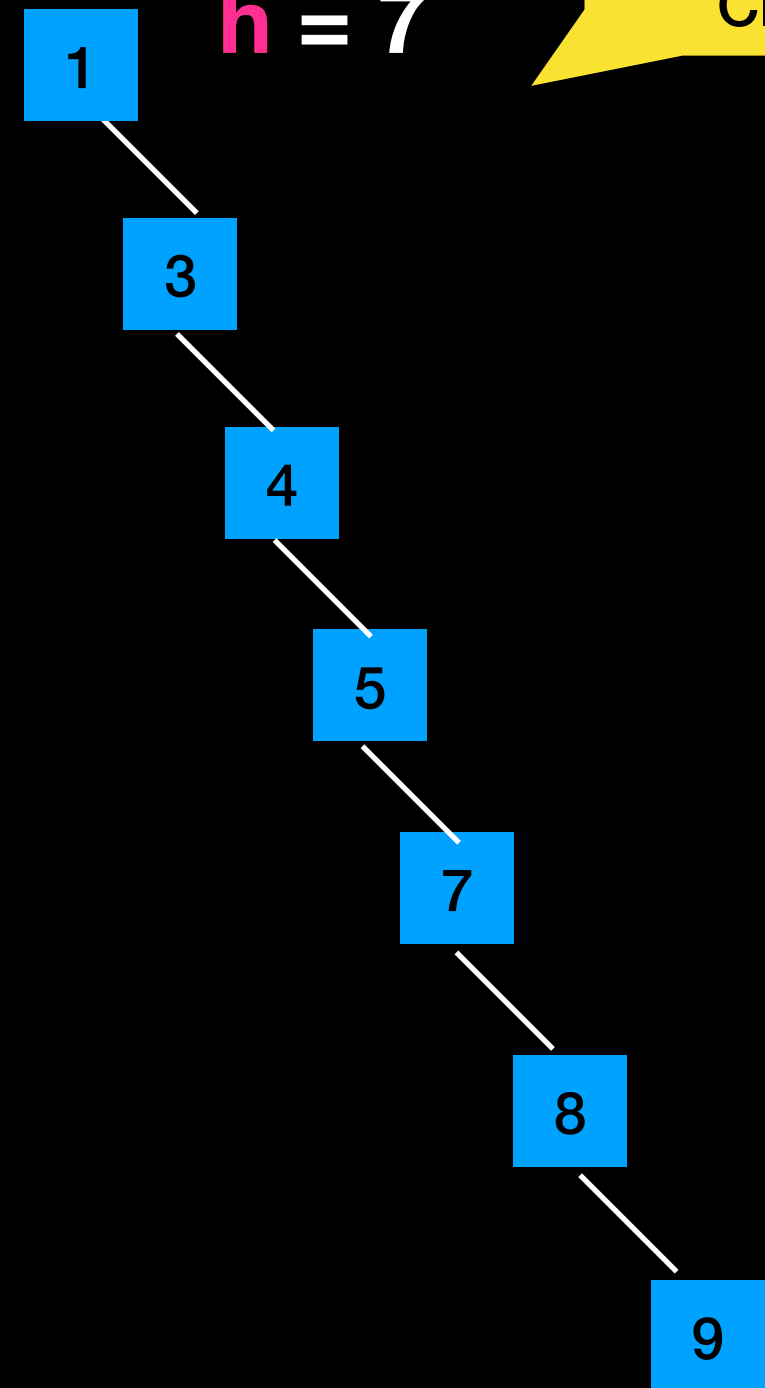


n nodes

$$\log_2 (n+1) \leq h \leq n$$

$h = 7$

Chain



Operation	In Full Tree	Worst-case
Search	$\log_2(n+1)$	$O(h)$
Add	$\log_2(n+1)$	$O(h)$
Remove	$\log_2(n+1)$	$O(h)$
Traverse	n	$O(n)$

BST Operations

```

#ifndef BST_H_
#define BST_H_
template<typename ItemType>
class BST
{
public:
    BST(); // constructor
    BST(const BST<ItemType>& tree); // copy constructor
    ~BST(); // destructor
    bool isEmpty() const;
    size_t getHeight() const;
    size_t getNumberOfNodes() const;
    void add(const ItemType& new_item);
    void remove(const ItemType& new_item);
    ItemType find(const ItemType& item) const;
    void clear();
    void preorderTraverse(Visitor<ItemType>& visit) const;
    void inorderTraverse(Visitor<ItemType>& visit) const;
    void postorderTraverse(Visitor<ItemType>& visit) const;
    BST& operator= (const BST<ItemType>& rhs);
private:
    //implementation details here
}; //end BST
#include "BST.cpp"
#endif // BST_H_

```

Looks a lot like a
BinaryTree

Might you inherit
from it?

What would you
override?

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

```

#ifndef BST_H_
#define BST_H_
template<typename ItemType>
class BST
{
public:
    BST(); // constructor
    BST(const BST<ItemType>& tree); // copy constructor
    ~BST(); // destructor
    bool isEmpty() const;
    size_t getHeight() const;
    size_t getNumberOfNodes() const;
    void add(const ItemType& new_item);
    void remove(const ItemType& new_item);
    ItemType find(const ItemType& item) const;
    void clear();
    void preorderTraverse(Visitor<ItemType>& visit) const;
    void inorderTraverse(Visitor<ItemType>& visit) const;
    void postorderTraverse(Visitor<ItemType>& visit) const;
    BST& operator= (const BST<ItemType>& rhs);
private:
    //implementation details here
}; //end BST
#include "BST.cpp"
#endif // BST_H_

```

Looks a lot like a
BinaryTree

Might you inherit
from it?

What would you
override?

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