Trees

Tiziana Ligorio
tligorio@hunter.cuny.edu

Today’s Plan

Trees

Binary Tree ADT

Binary Search Tree ADT

Announcements and Syllabus Check

Sorry and thank you for your patience on the projects!

We will have 5 projects total, lowest will be dropped.

Questions?

ADT Operations we have seen so far

List, Stack, Queue

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

Always position based

For list, retrieval can be value based

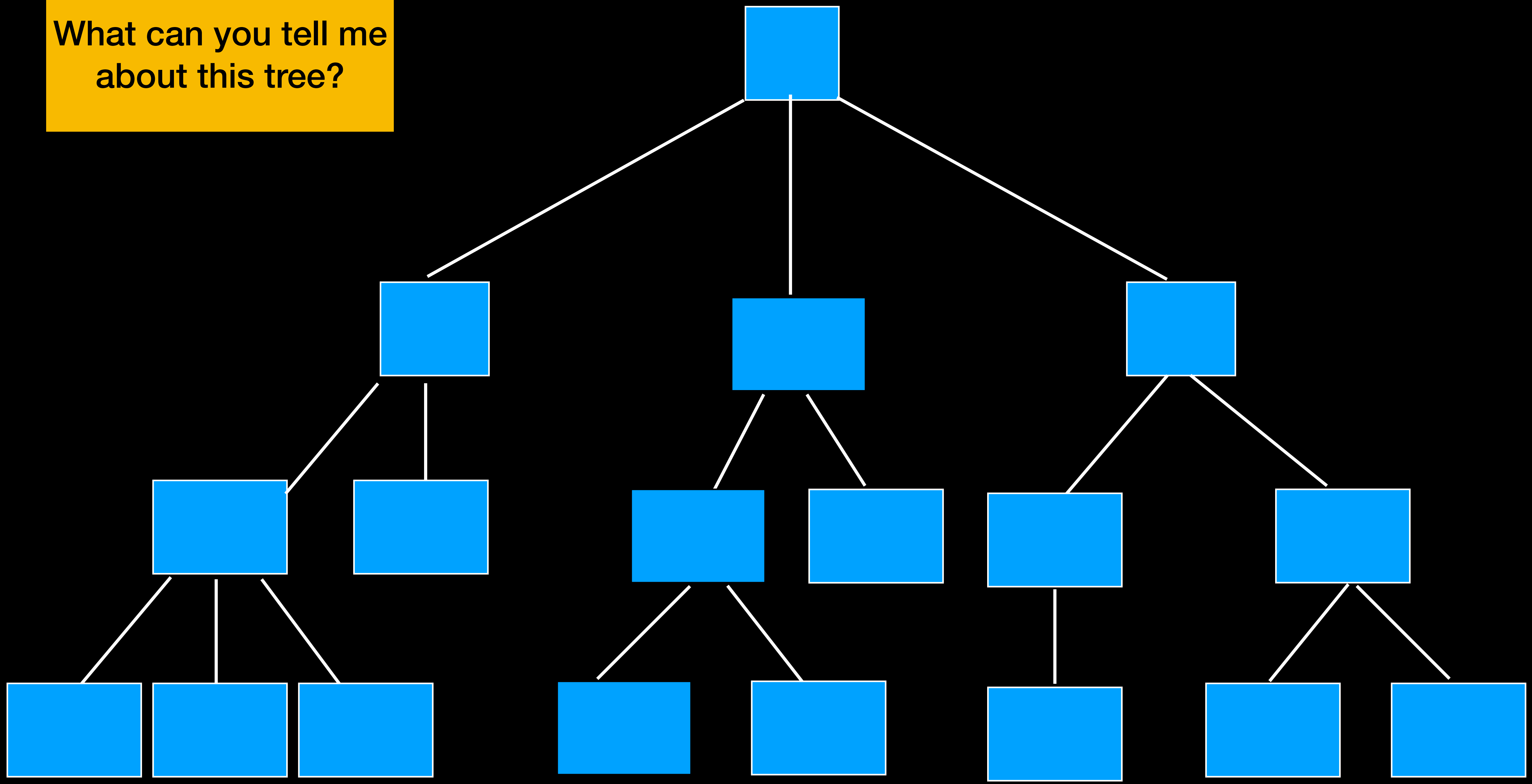
Data organization is linear

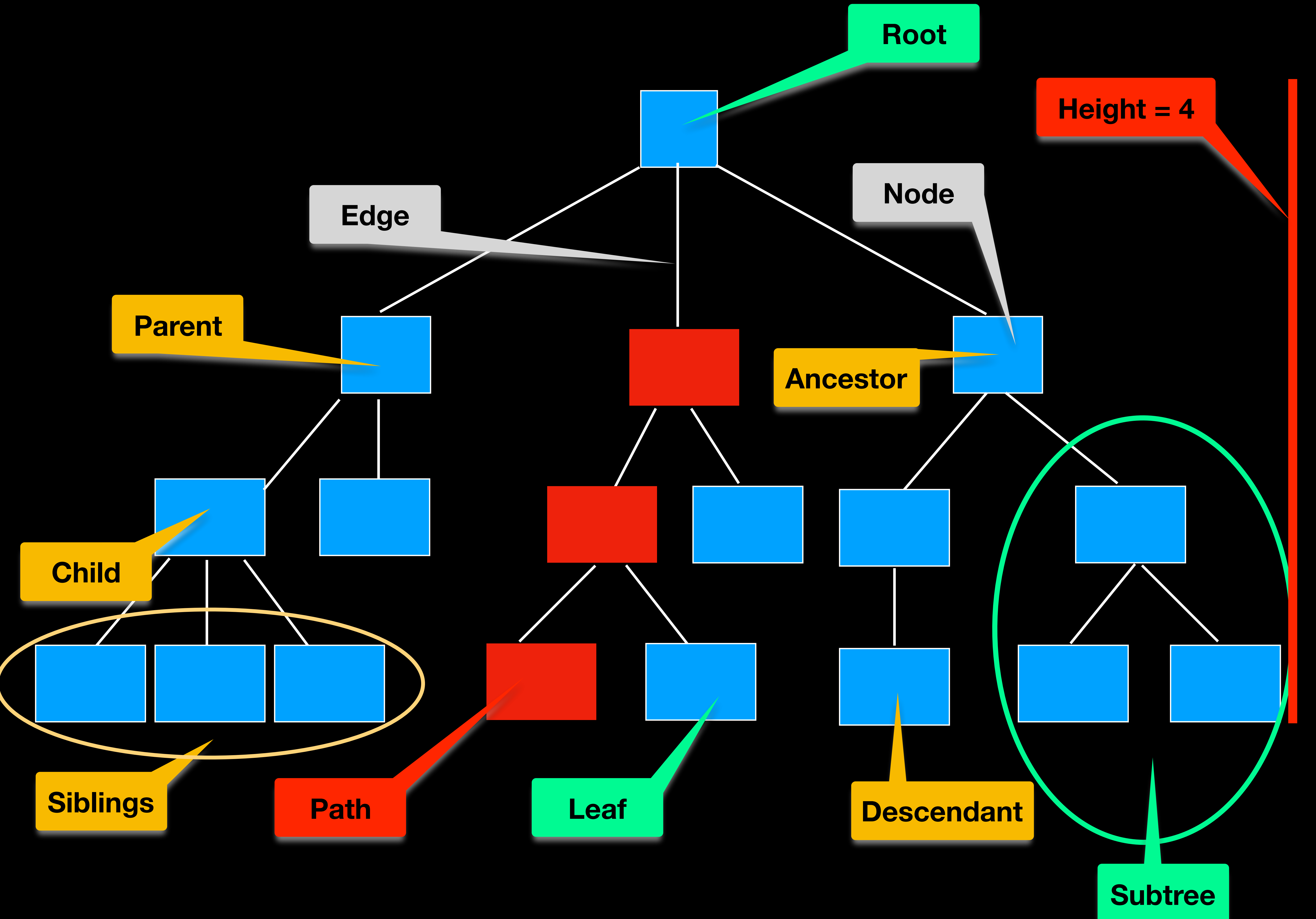
Tree

Represent relationships

Hierarchical (directional) organization

What can you tell me about this tree?




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

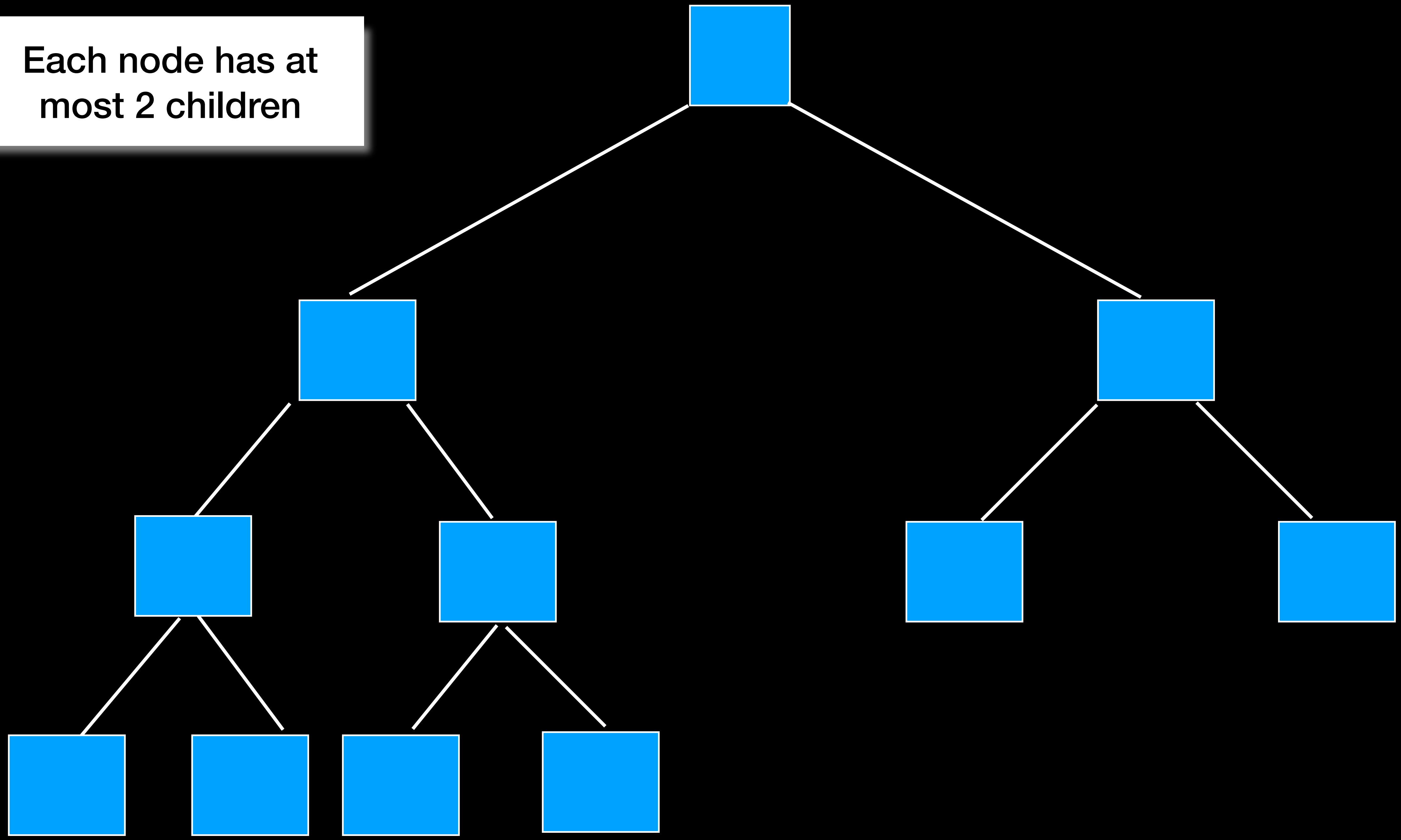
Path: a sequence of nodes c_1, c_2, ..., c_k where c_{i+1} is a child of c_i.

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

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

BinaryTree

Each node has at most 2 children

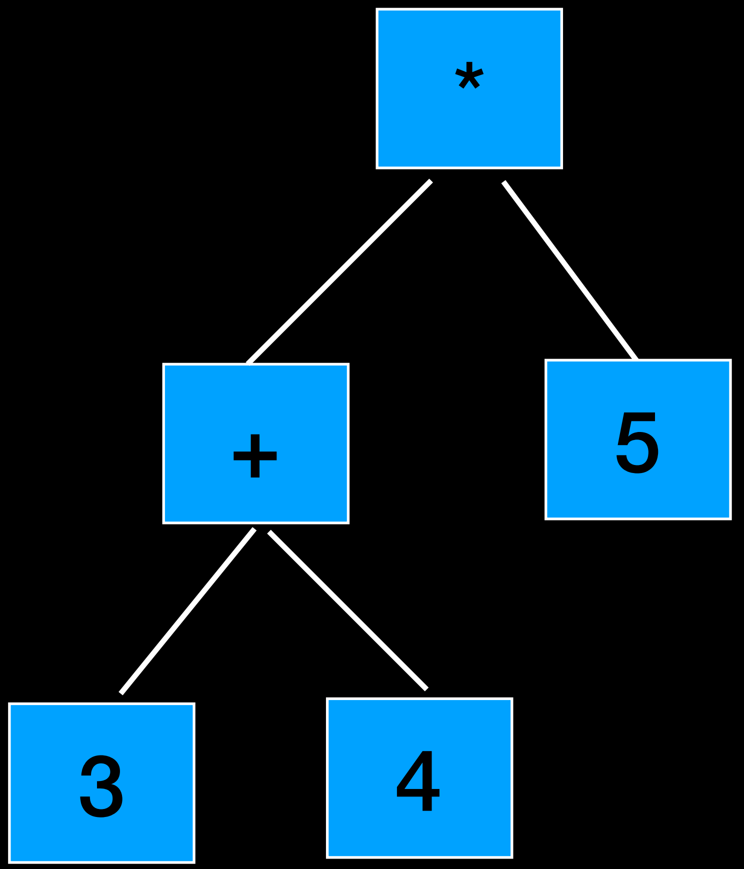


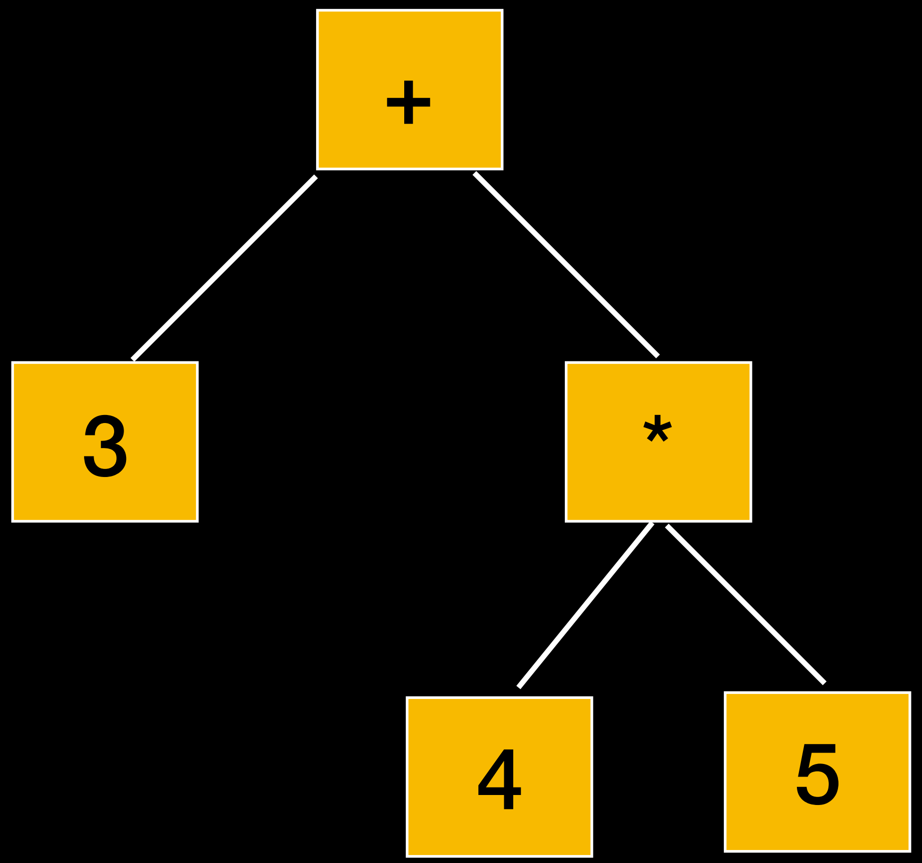
Binary Tree Applications

Algebraic Expressions

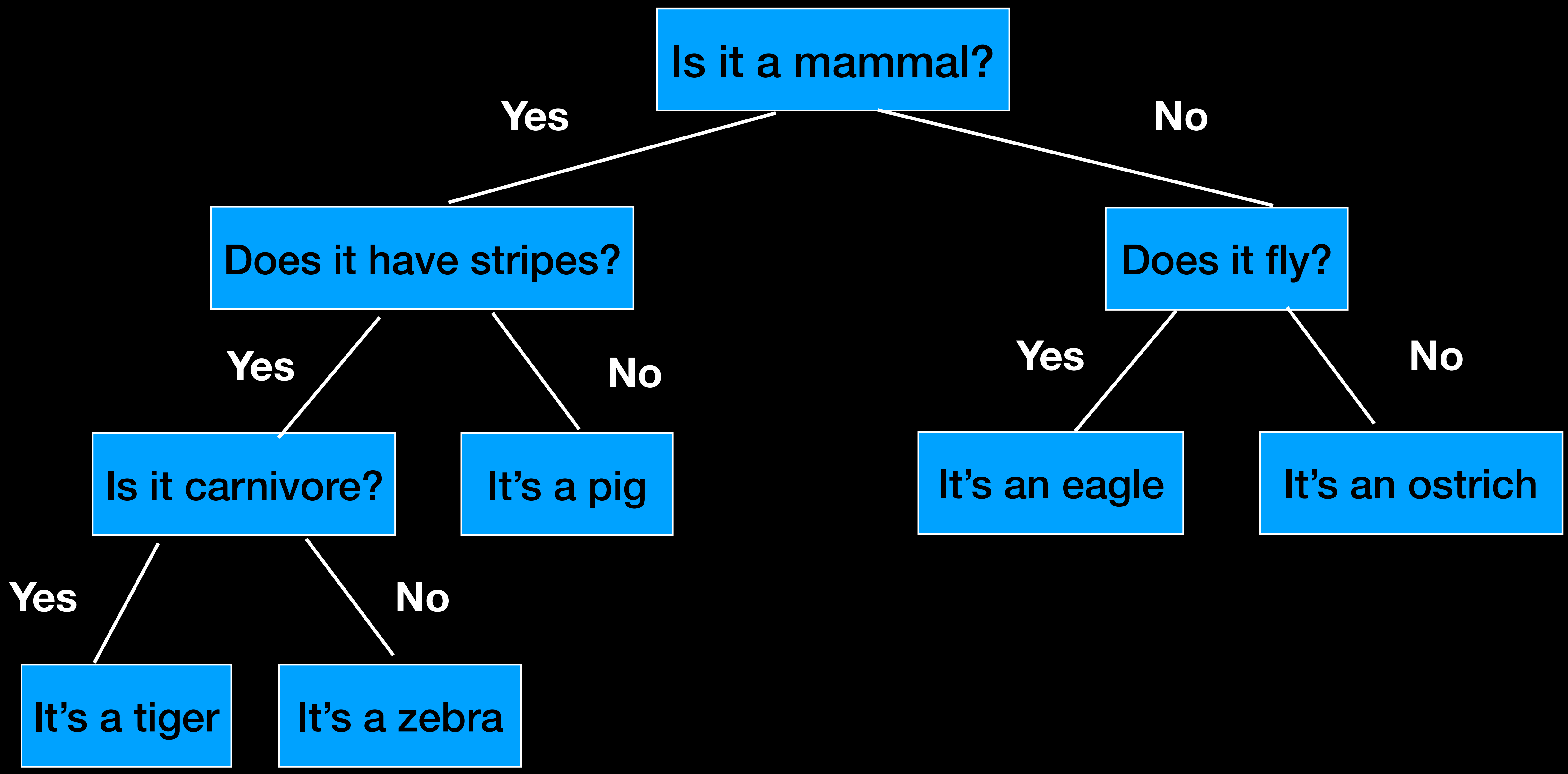
(3 + 4) * 5

3 + 4 * 5





Decision Tree



Huffman Tree

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

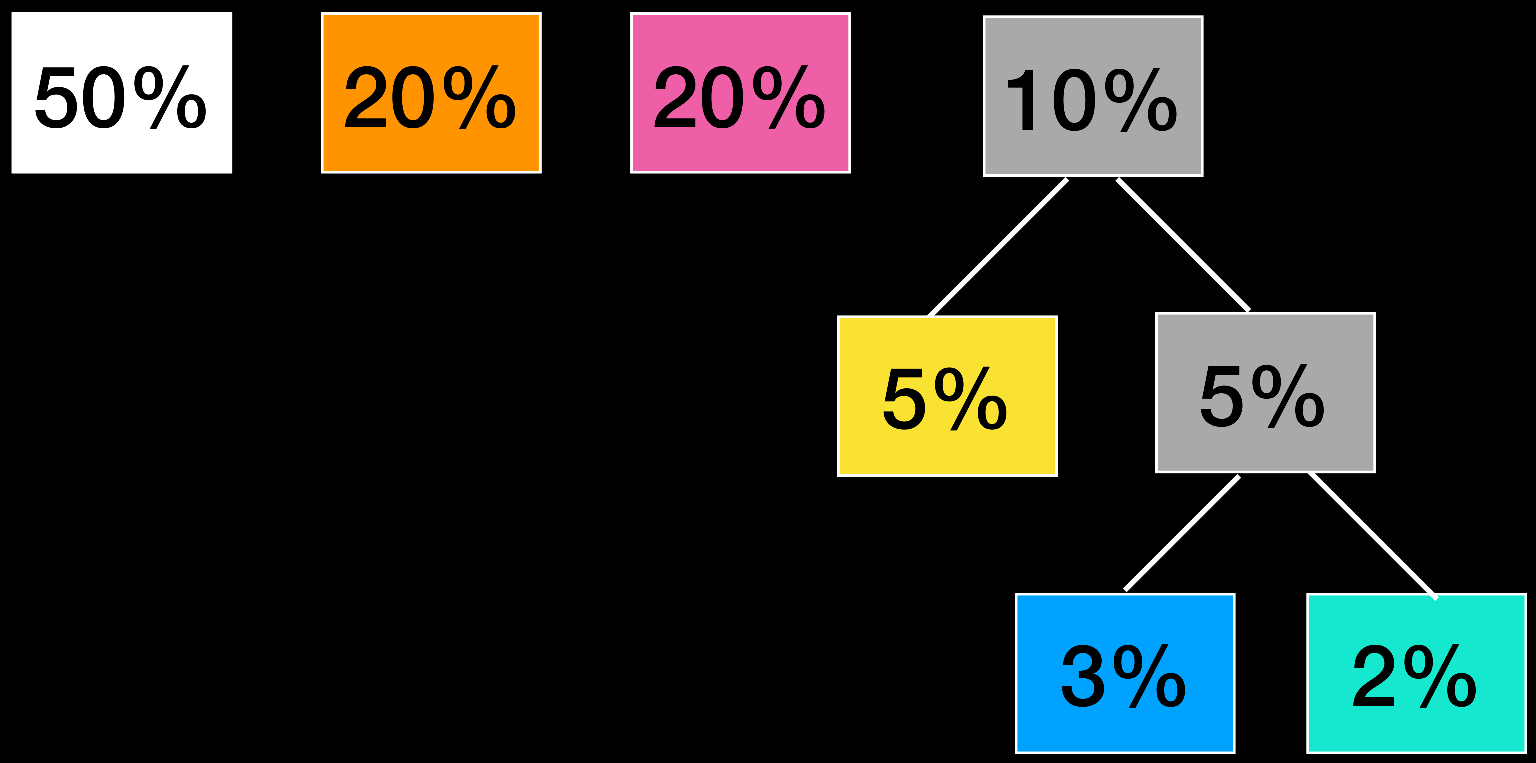
No codeword is prefix to another codeword (i.e. if a symbol is encoded as 00 no other codeword can start with 00)

Huffman Tree

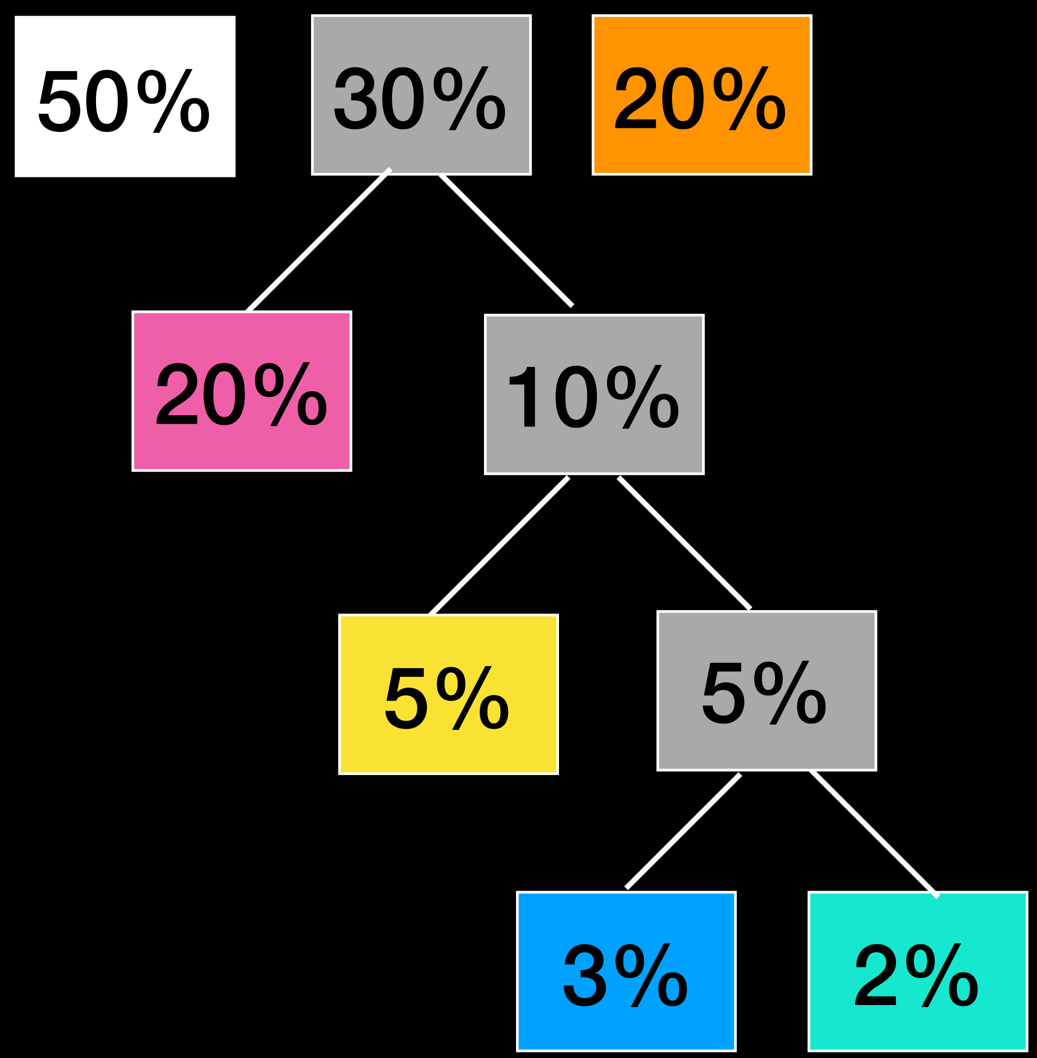
50%	20%	20%	5%	3%	2%

Huffman Tree

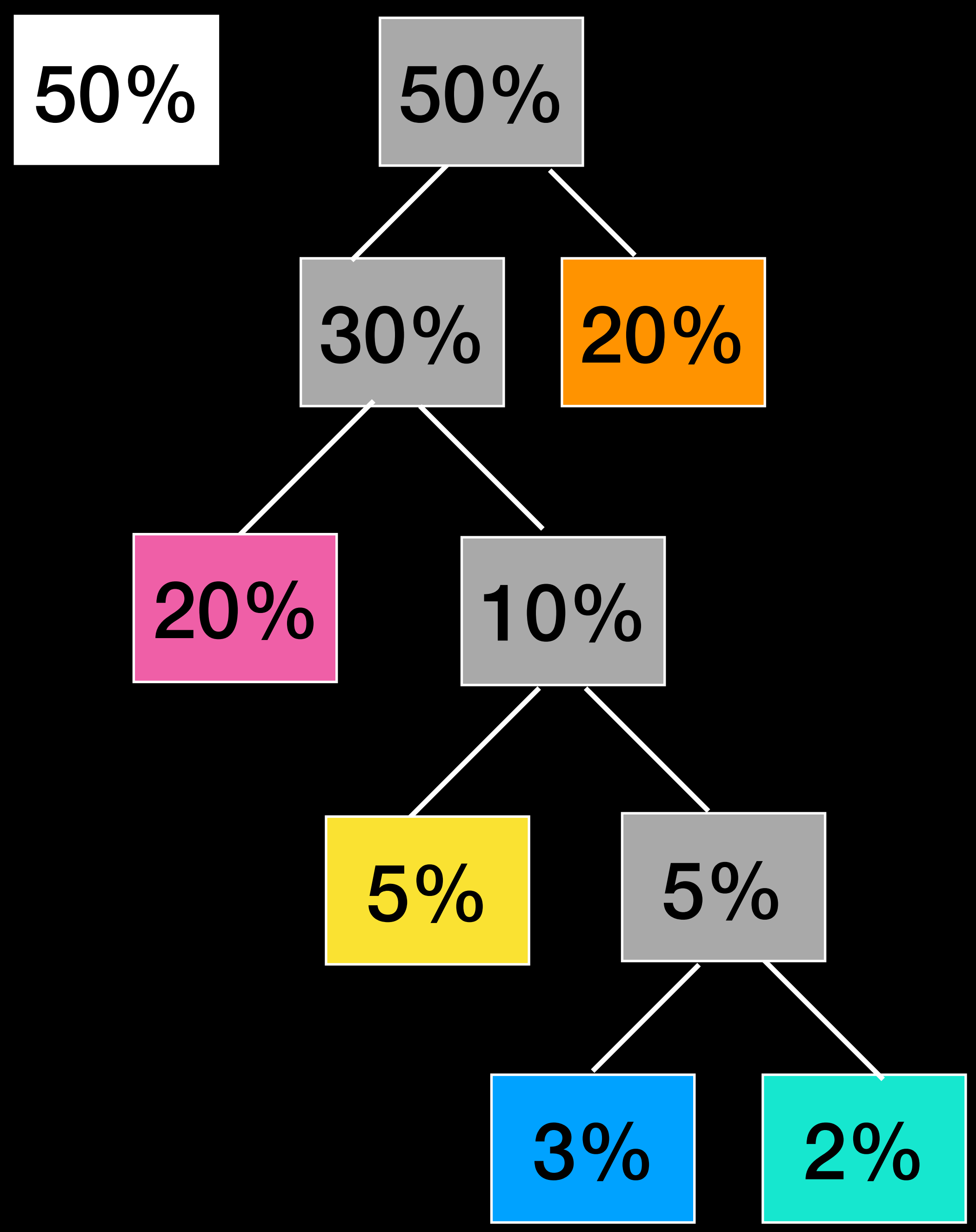
Huffman Tree



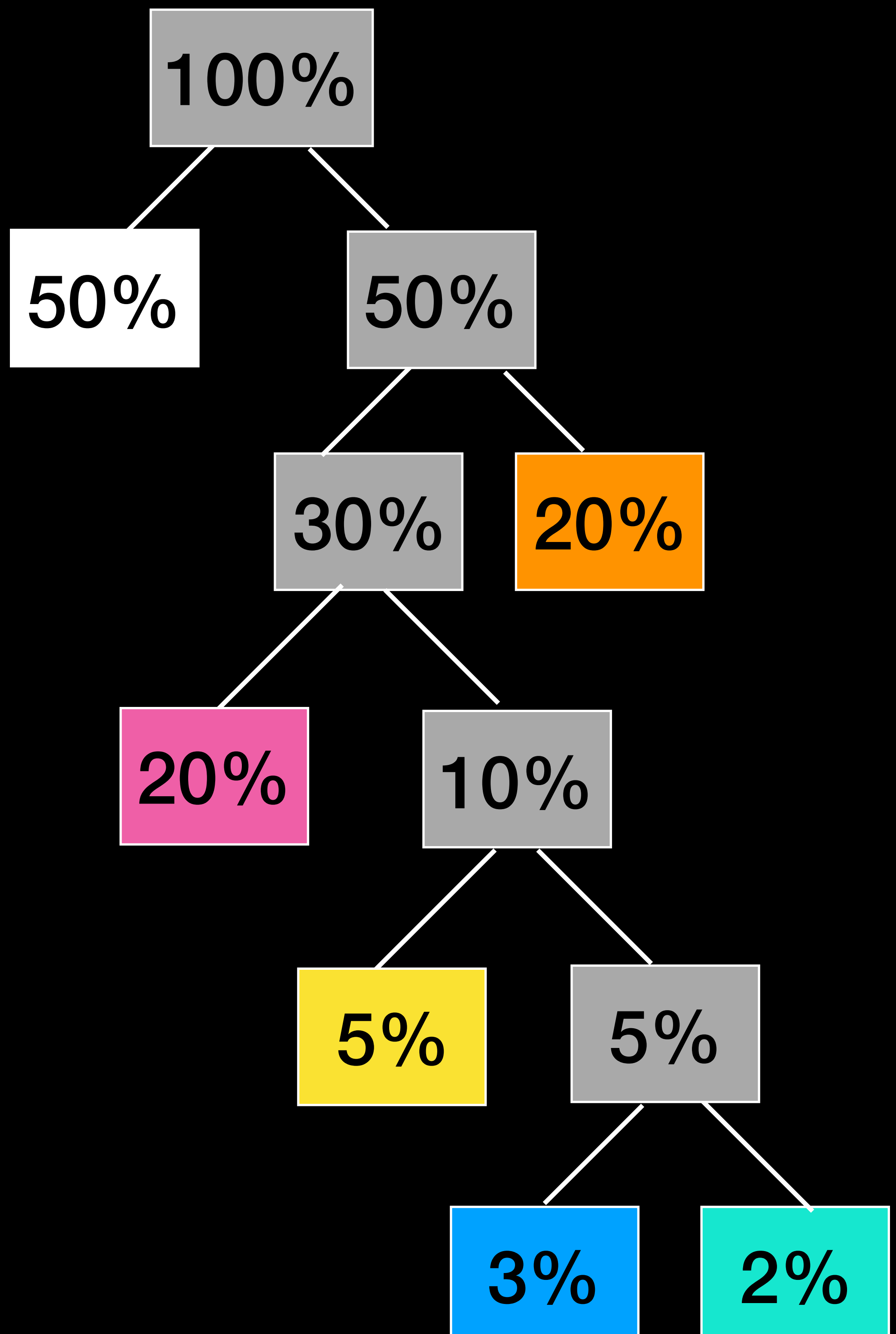
Huffman Tree



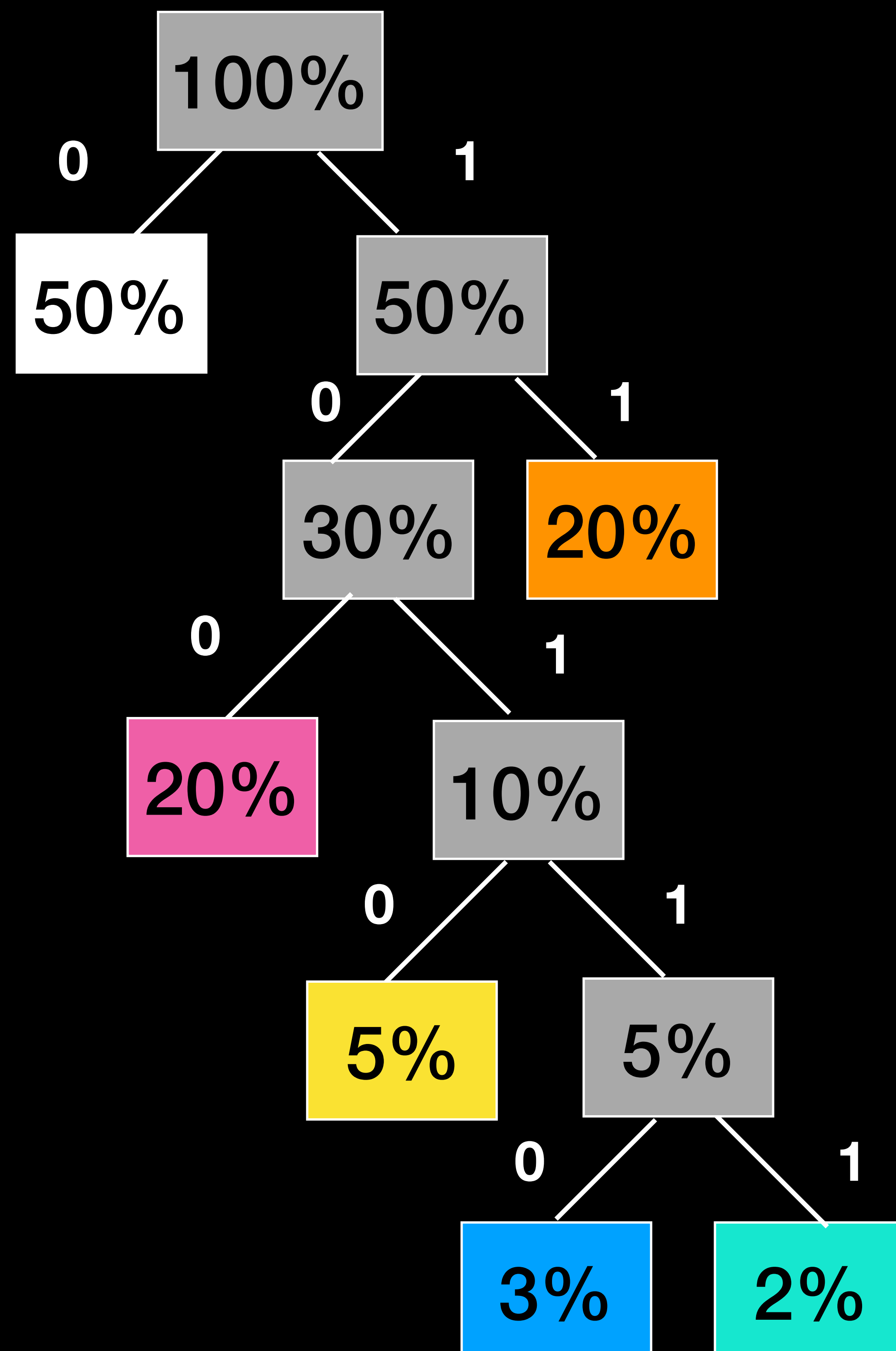
Huffman Tree



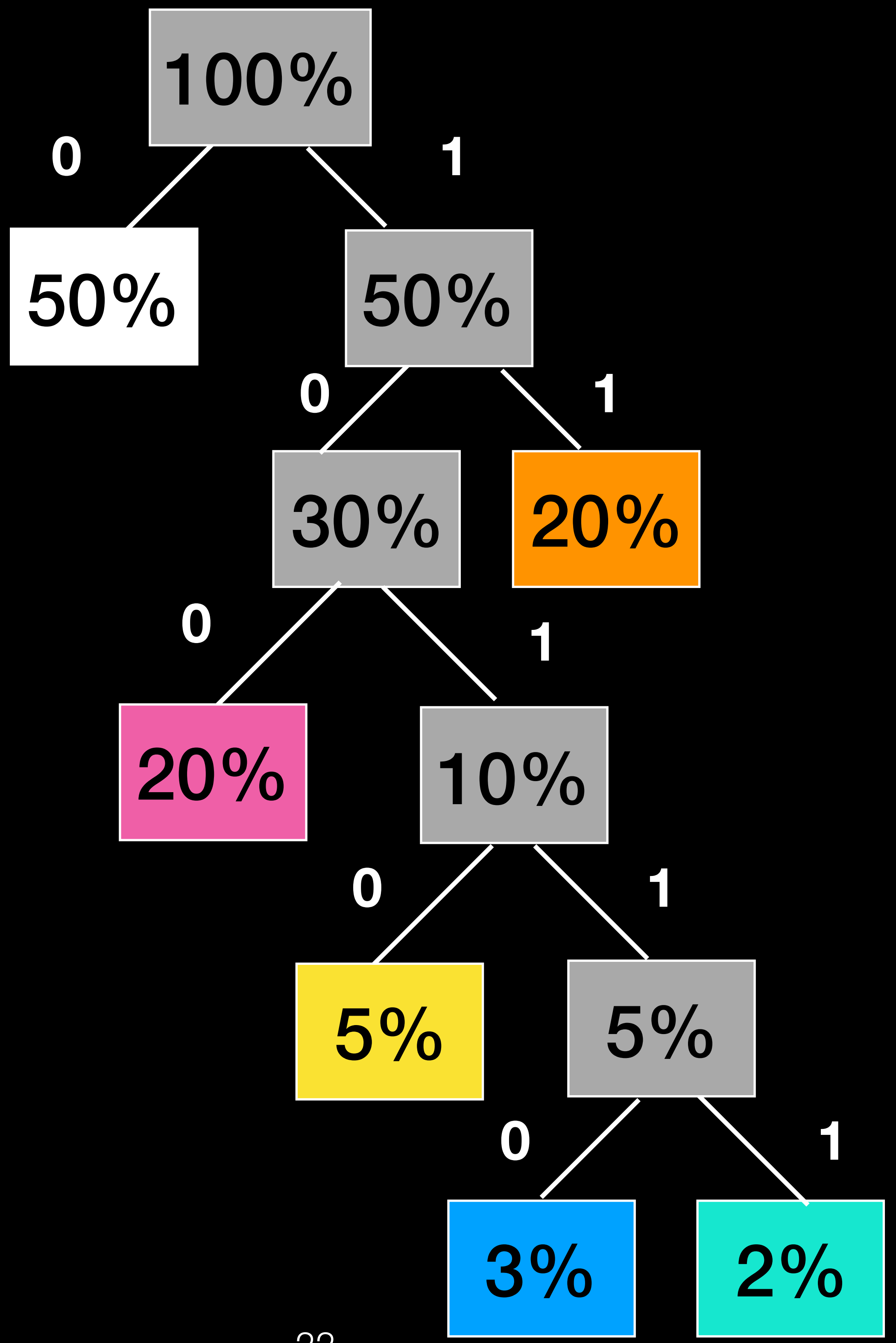
Huffman Tree



Huffman Tree

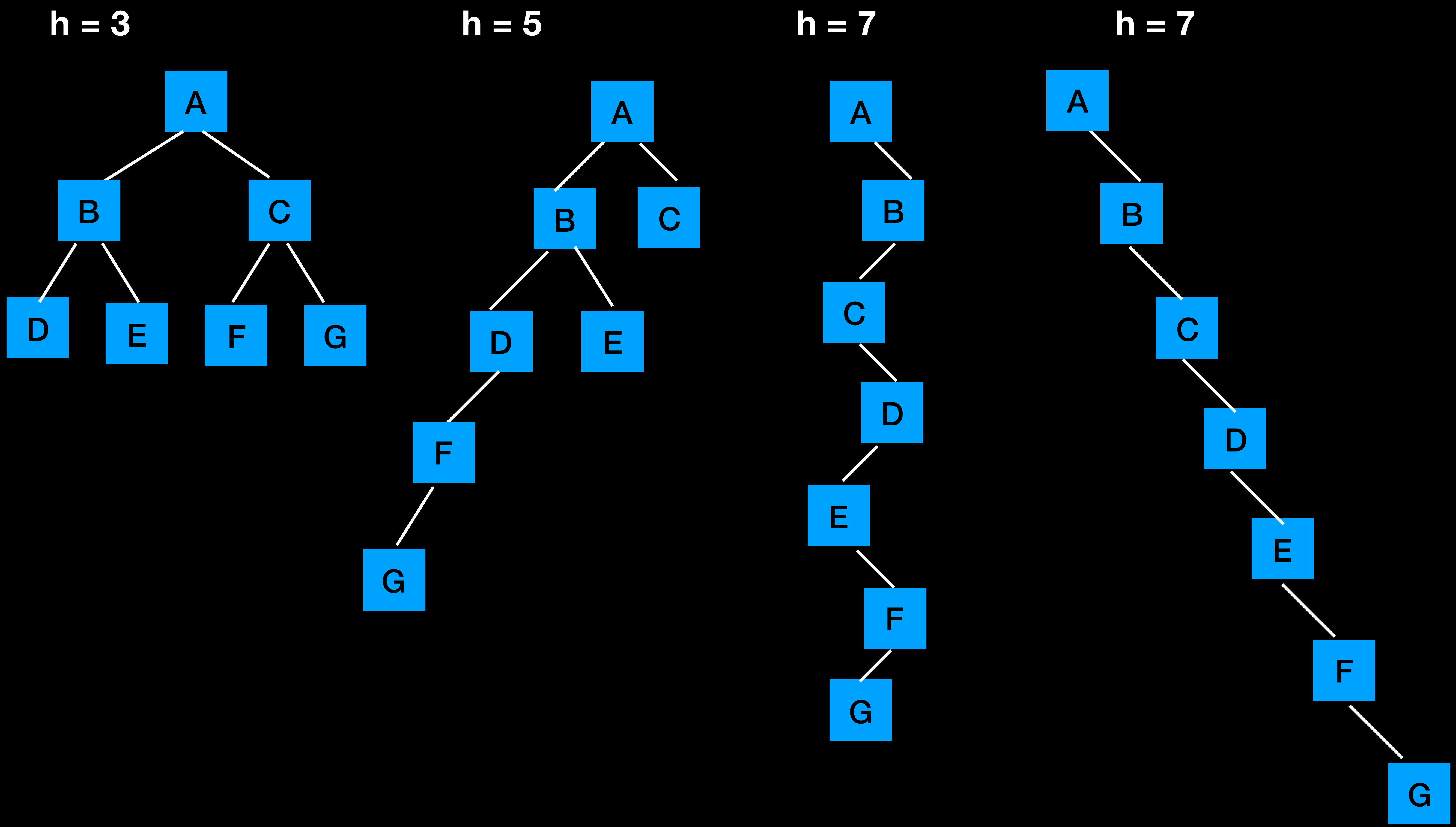


Huffman Tree

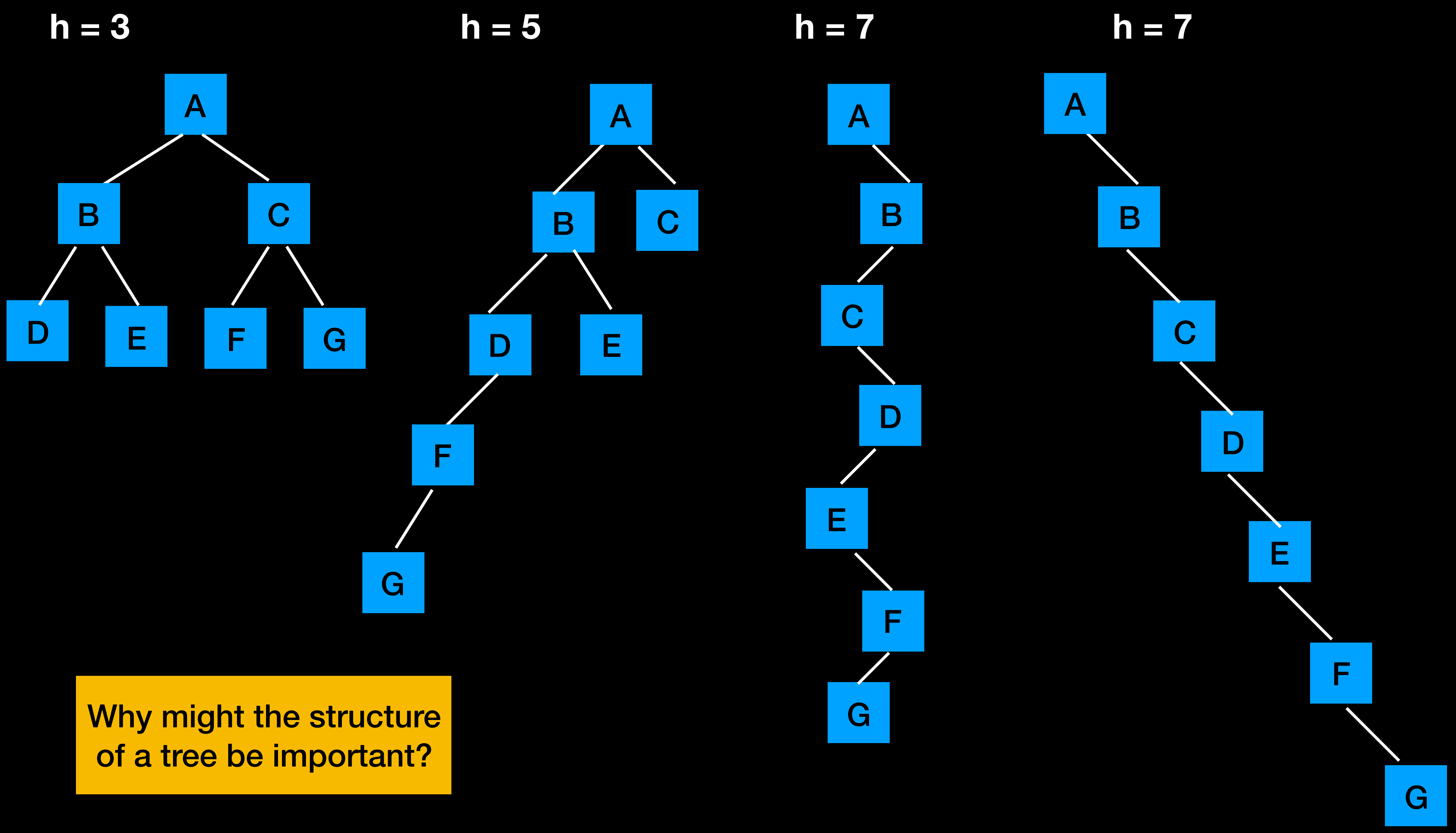


0
100
11
1010
10110
10111

Tree Structure



Tree Structure



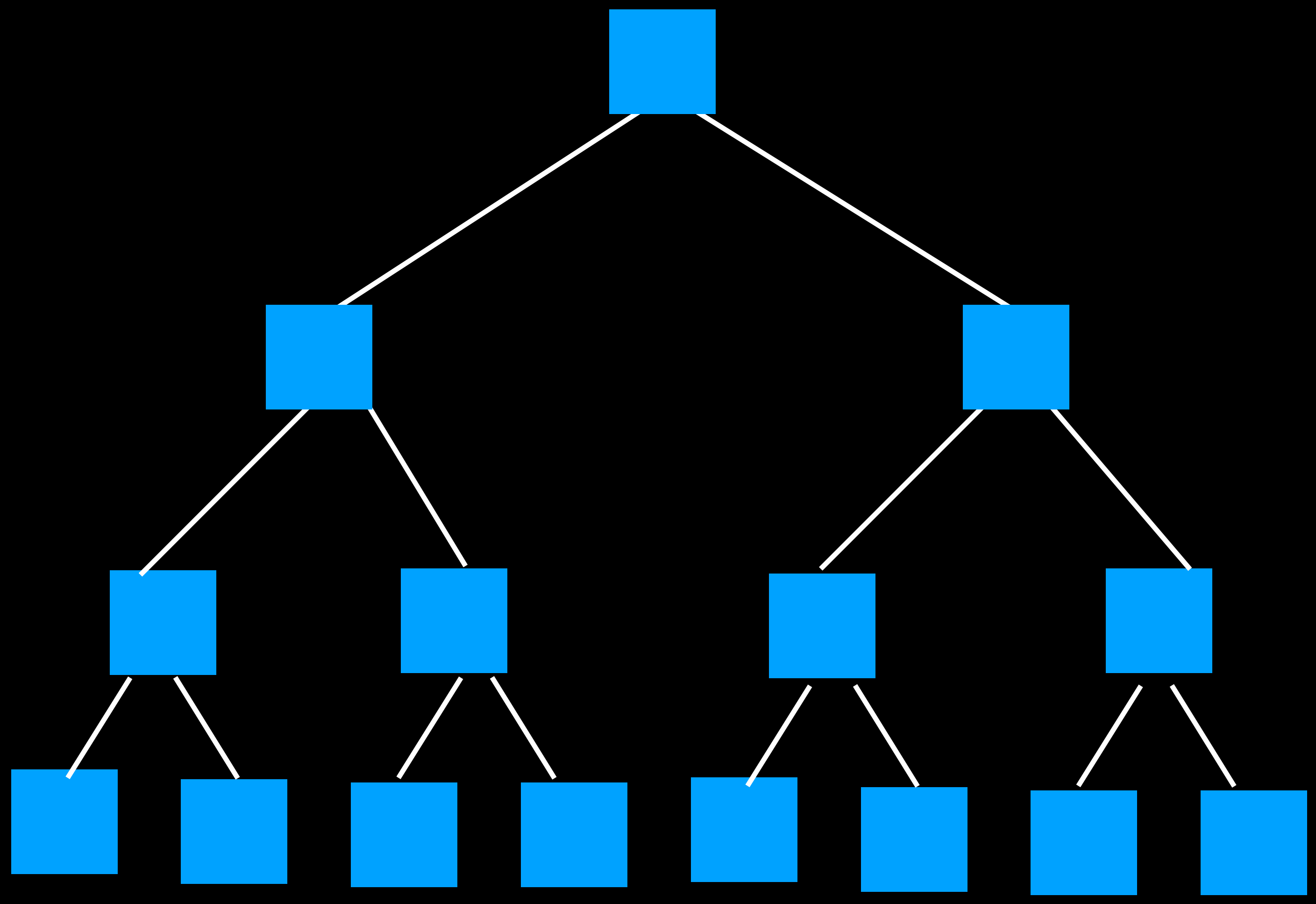
Why might the structure of a tree be important?

Full Binary Tree

Every node that is not a leaf has exactly 2 children

Every node has left and right subtrees of same size

All leaves are at same level *h*



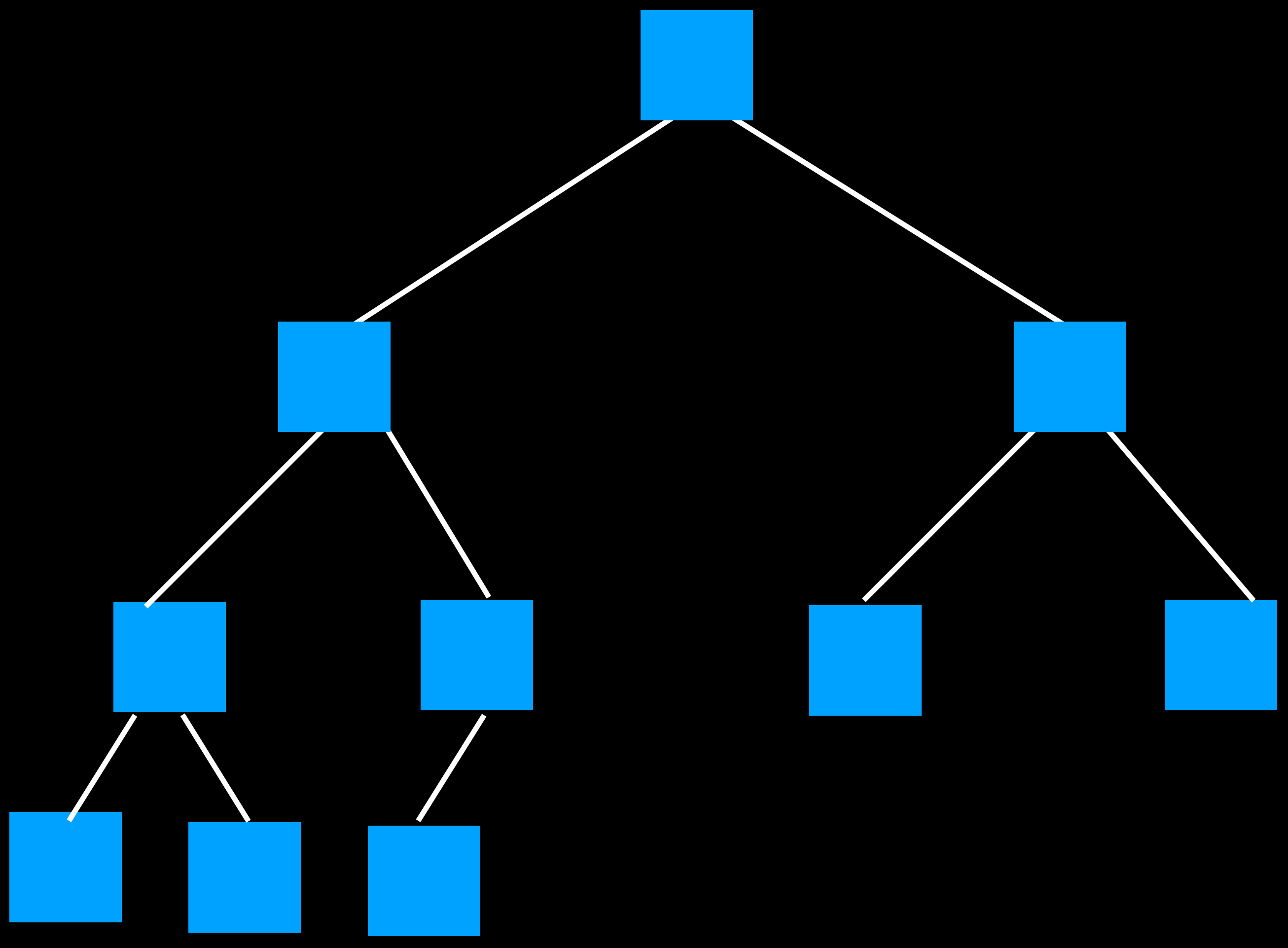
Complete Binary Tree

A three 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

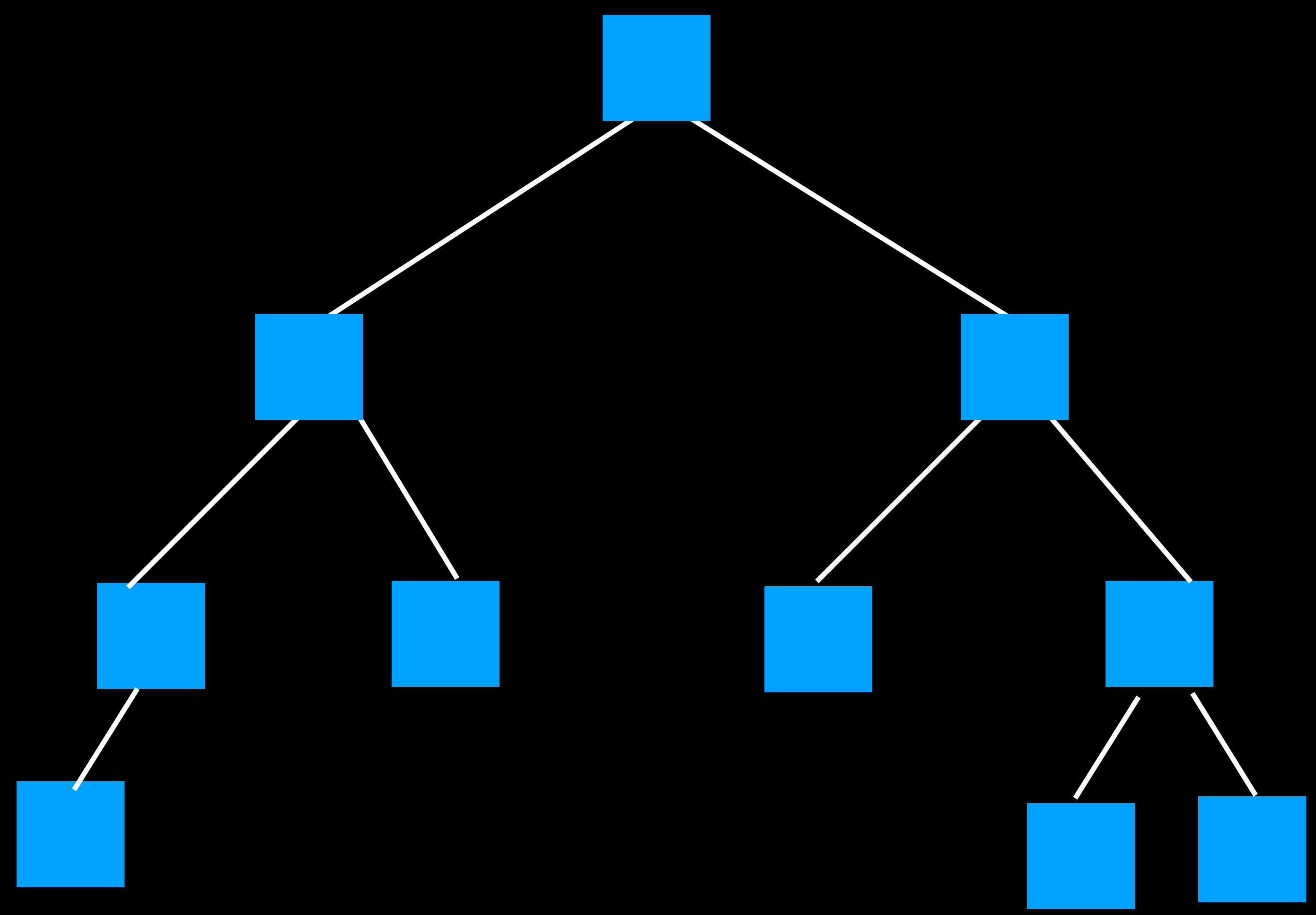
When a node at level *h-1* has one child, it is a left child



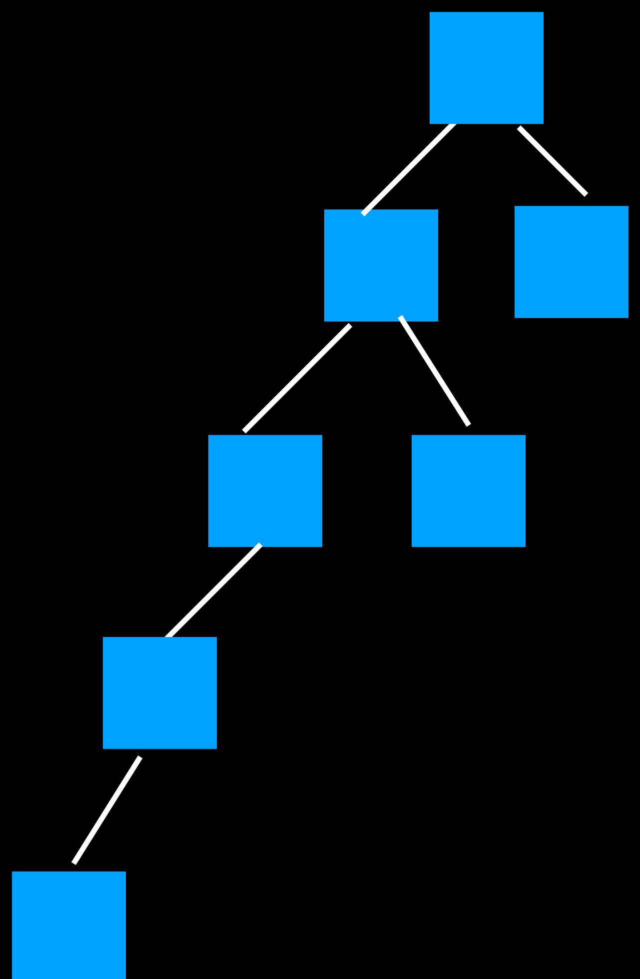
(Height) Balanced Binary Tree

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

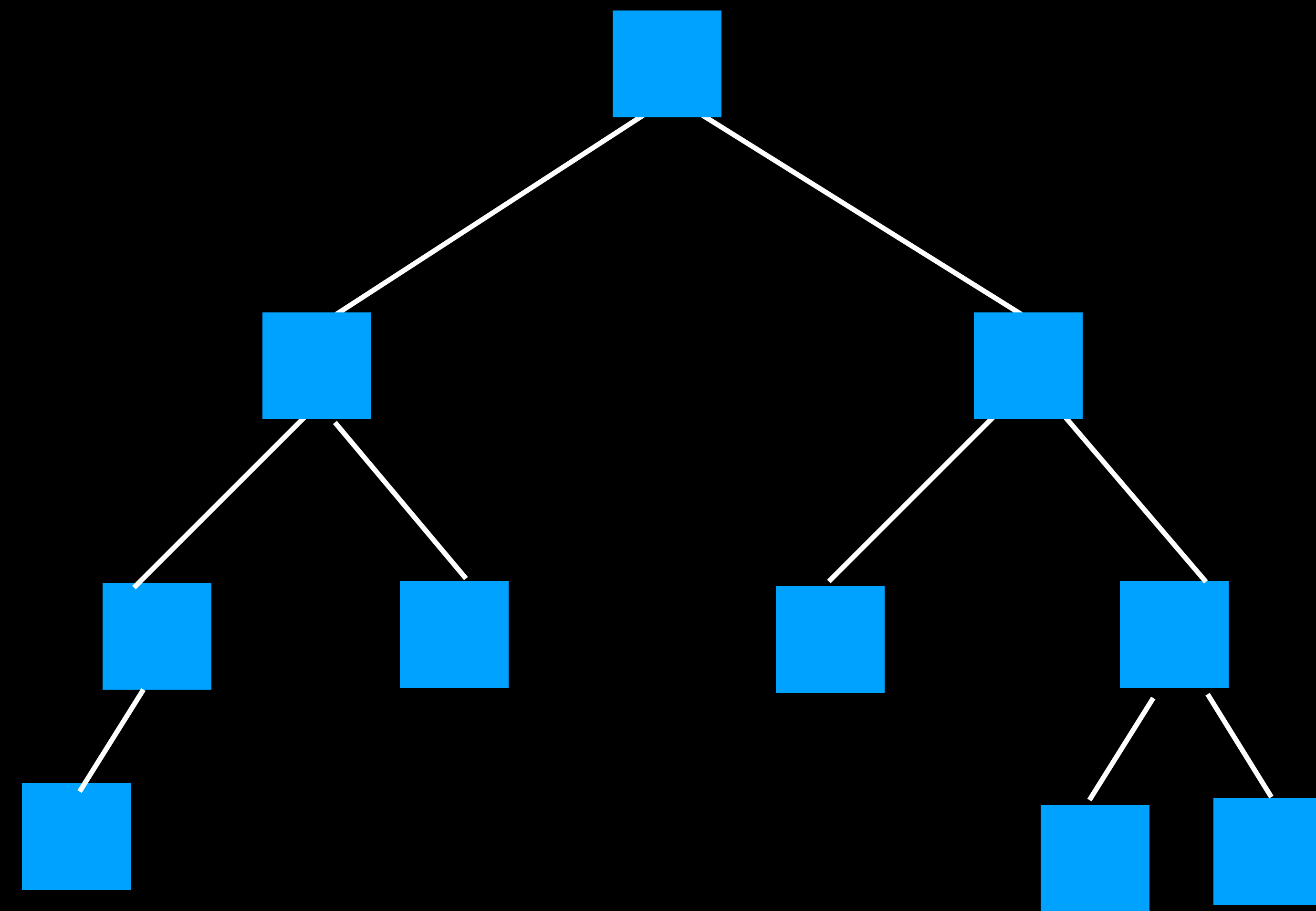
All paths from root to leaf differ in length by at most 1



Unbalanced



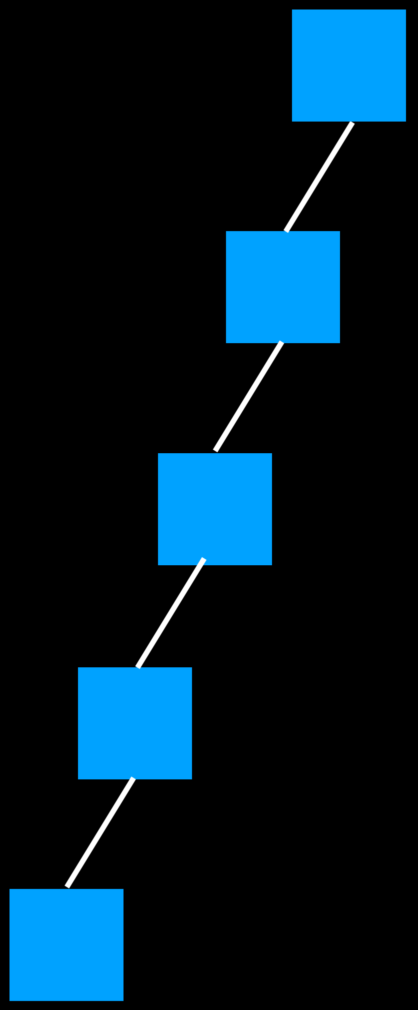
Balanced



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

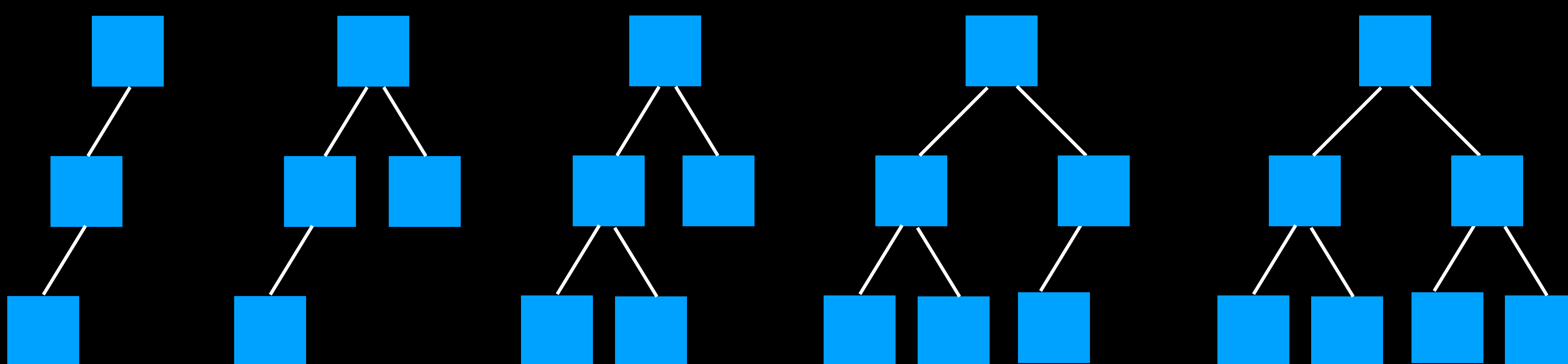
$h \approx \log_2 n$ for a balanced binary tree

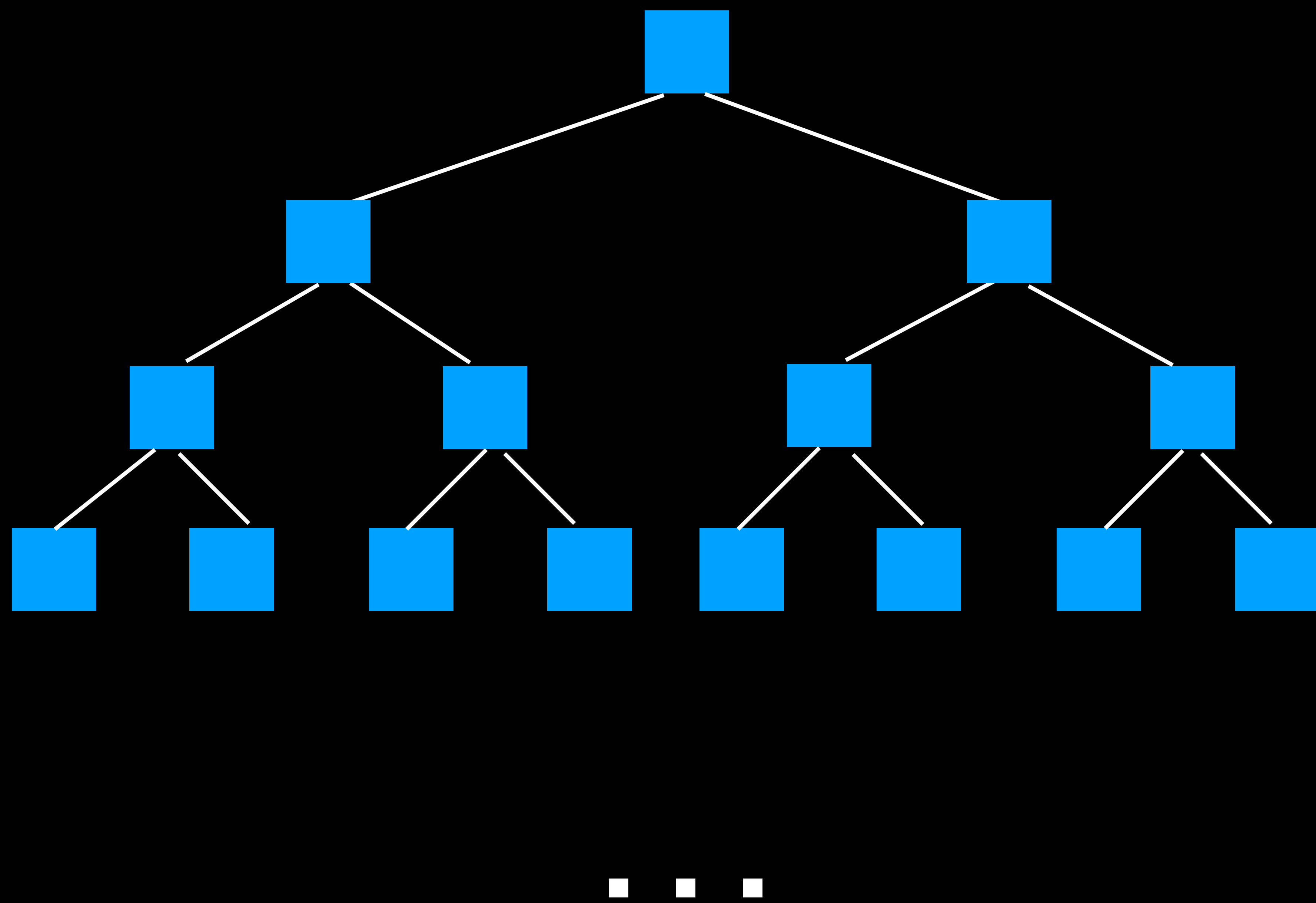
For example:

1000 nodes h ≈ 10 ($1000 \approx 1014 \approx 2^{10}$)

1000000 nodes h ≈ 20 ($10^6 \approx 2^{20}$)

Important when we will be looking for things in trees!!!





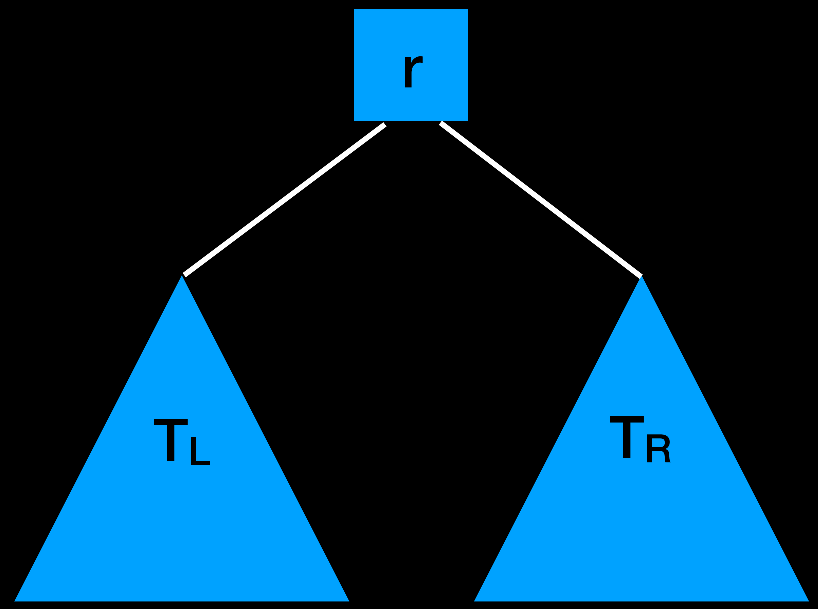
h	n @ level	Total n
1	$1 = 2^0$	$1 = 2^1 - 1$
2	$2 = 2^1$	$3 = 2^2 - 1$
3	$4 = 2^2$	$7 = 2^3 - 1$
4	$8 = 2^3$	$15 = 2^4 - 1$
h	2^{h-1}	$2^h - 1$

Binary Tree Traversals

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

Essentially visit the root as well as it’s subtrees

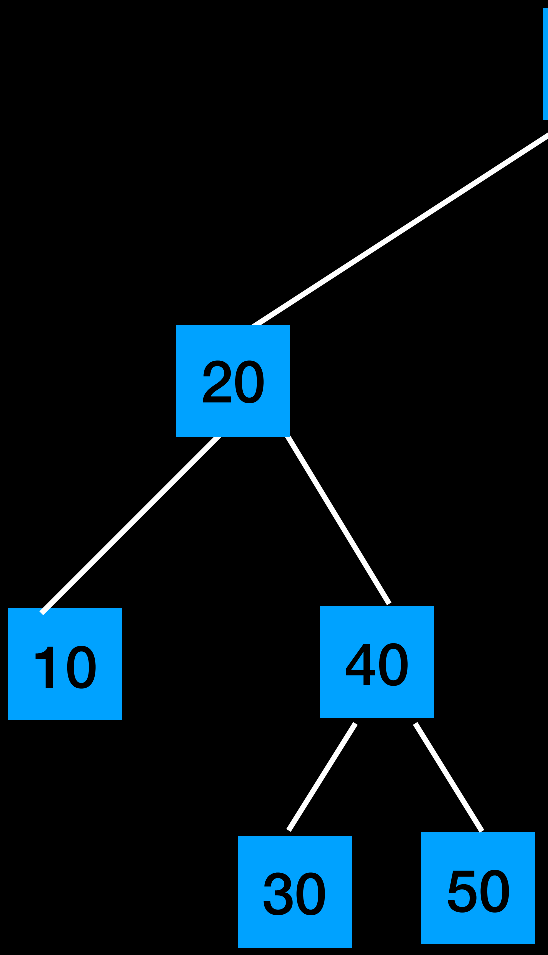
Order matters!!!

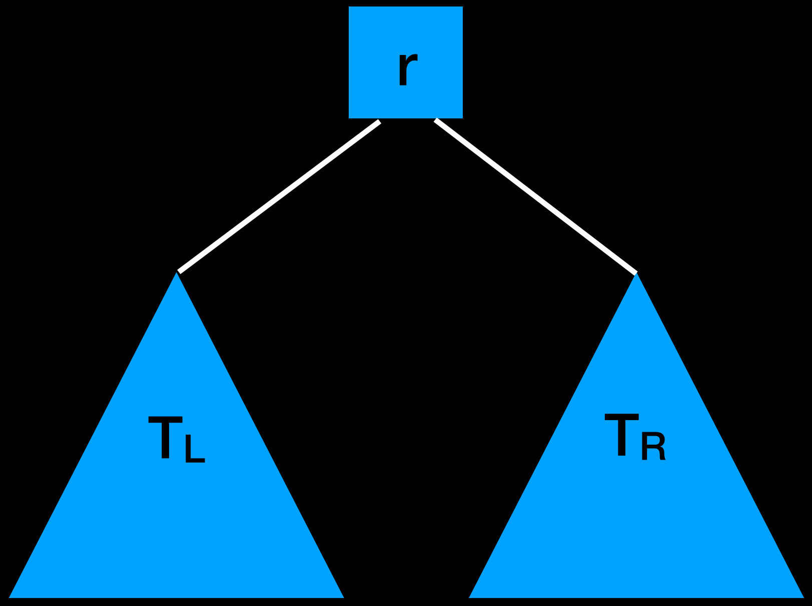


Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

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



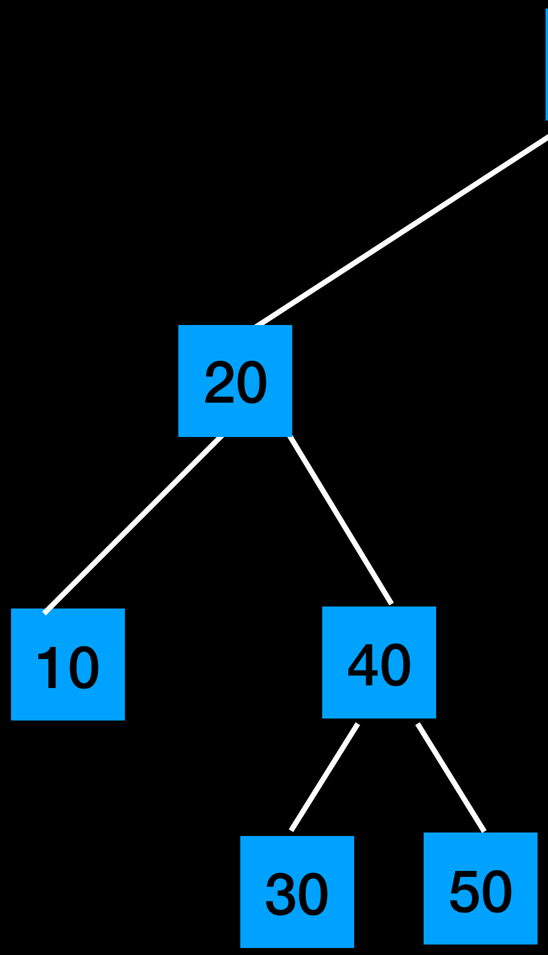


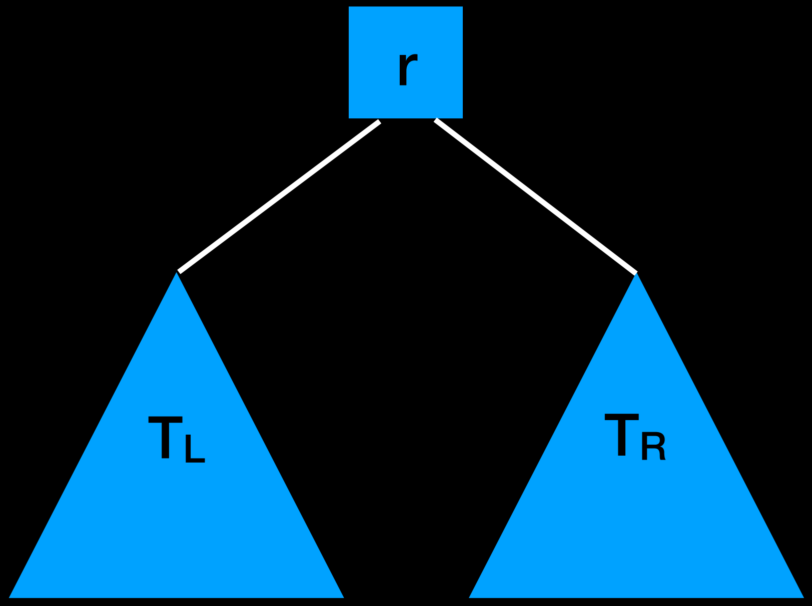
Preorder: 60, 20, 10, 40, 30, 50, 70

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
}
```



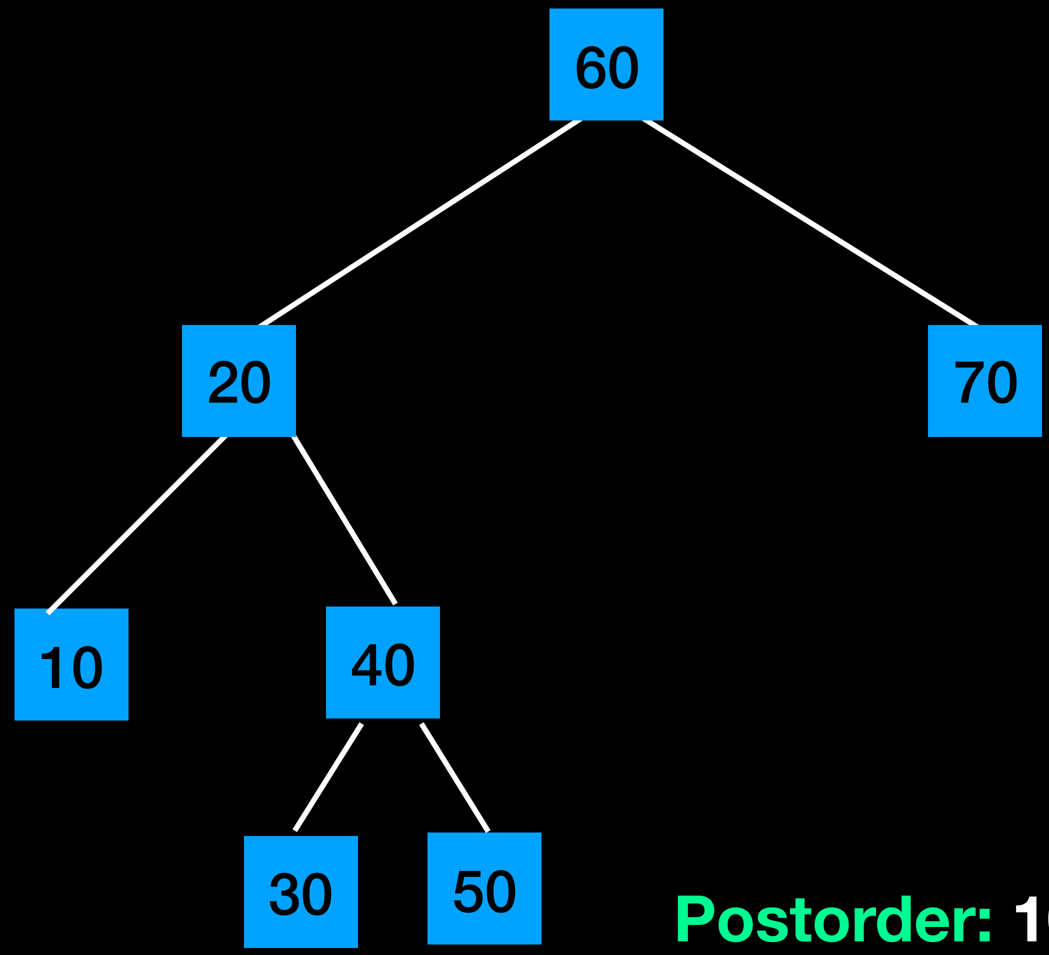


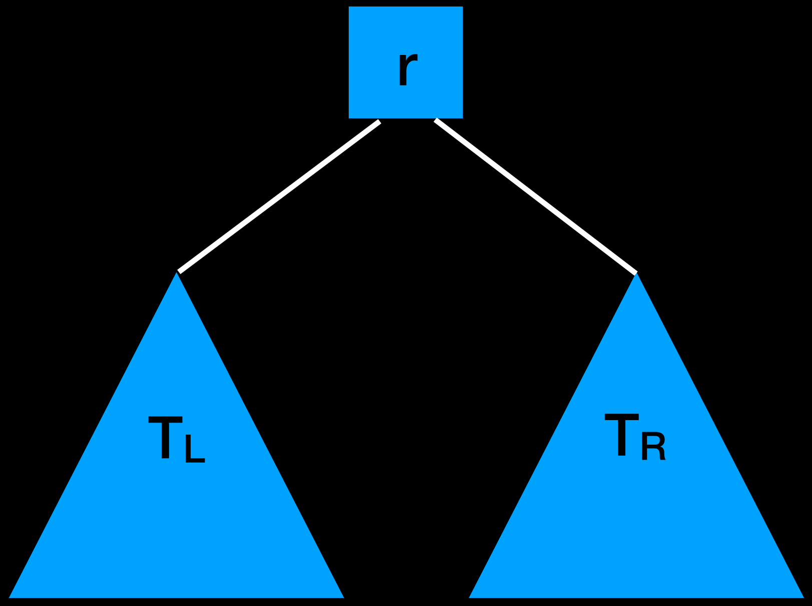
Inorder: 10, 20, 30, 40, 50, 60, 70

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

Postorder Traversal:

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





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

BinaryTree Operations

```
#ifndef BinaryTree_H_
#define BinaryTree_H_

template<class 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(void (*visit)(ItemType&))const;
    void inorderTraverse(void (*visit)(ItemType&))const;
    void postorderTraverse(void (*visit)(ItemType&))const;

    BinaryTree& operator= (const BinaryTree& rhs);

private: // implementation details here

}; // end BST

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

How might you do this?

You should implement this!

We will talk about implementation next time.
You should play around with it in the mean time