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

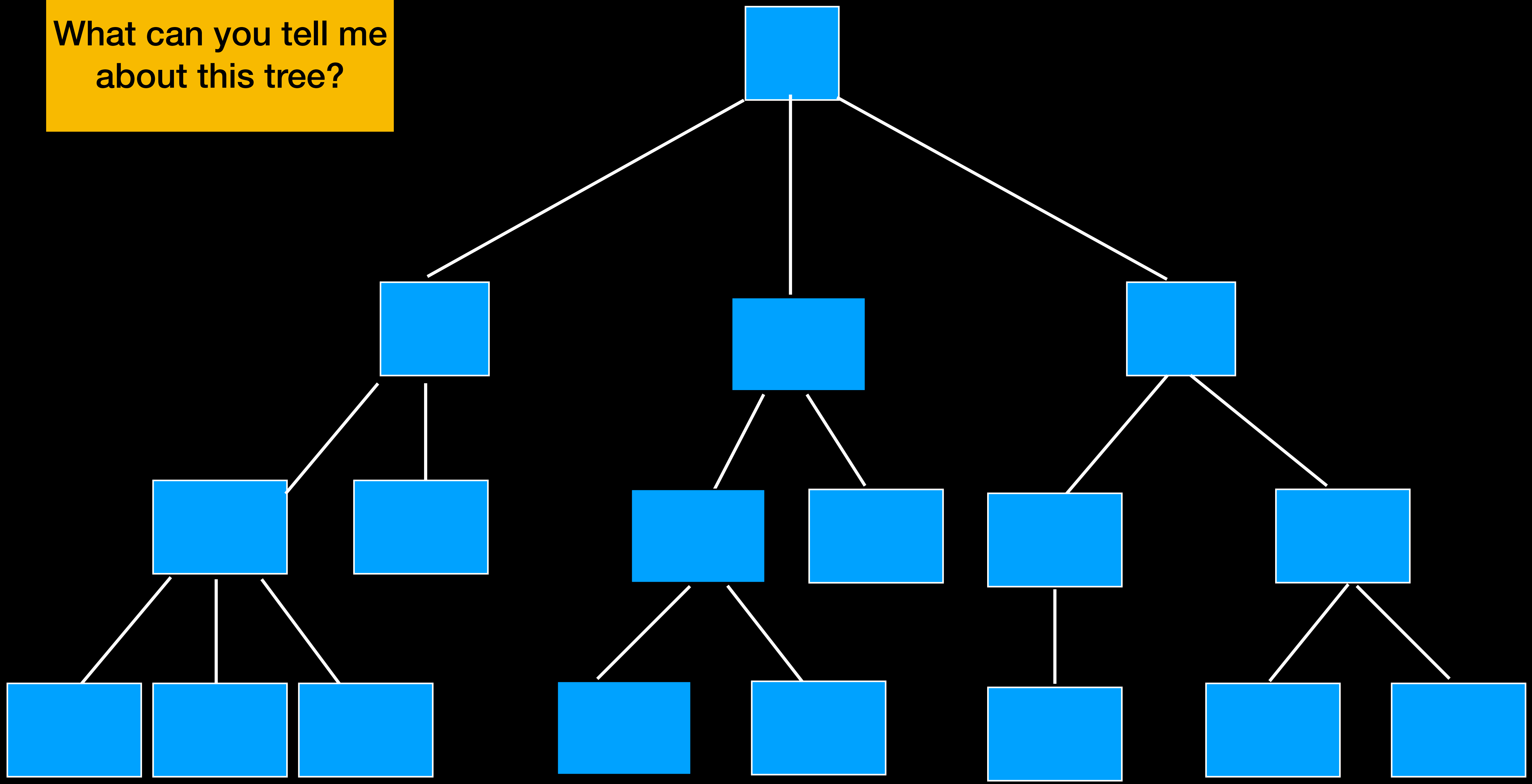
Data organization is linear

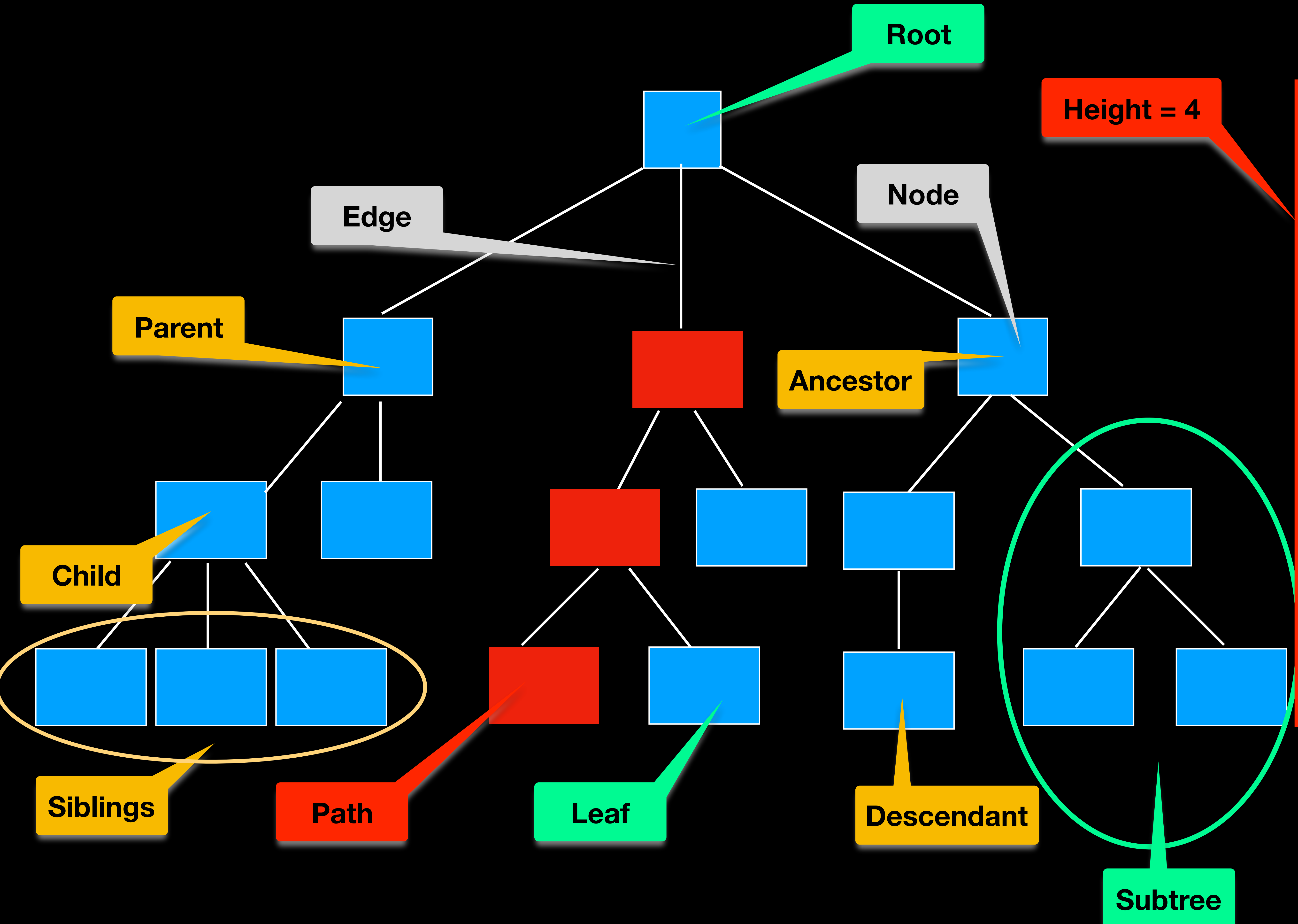
Tree

Represent relationships

Hierarchical (directional) organization

What can you tell me about this tree?




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

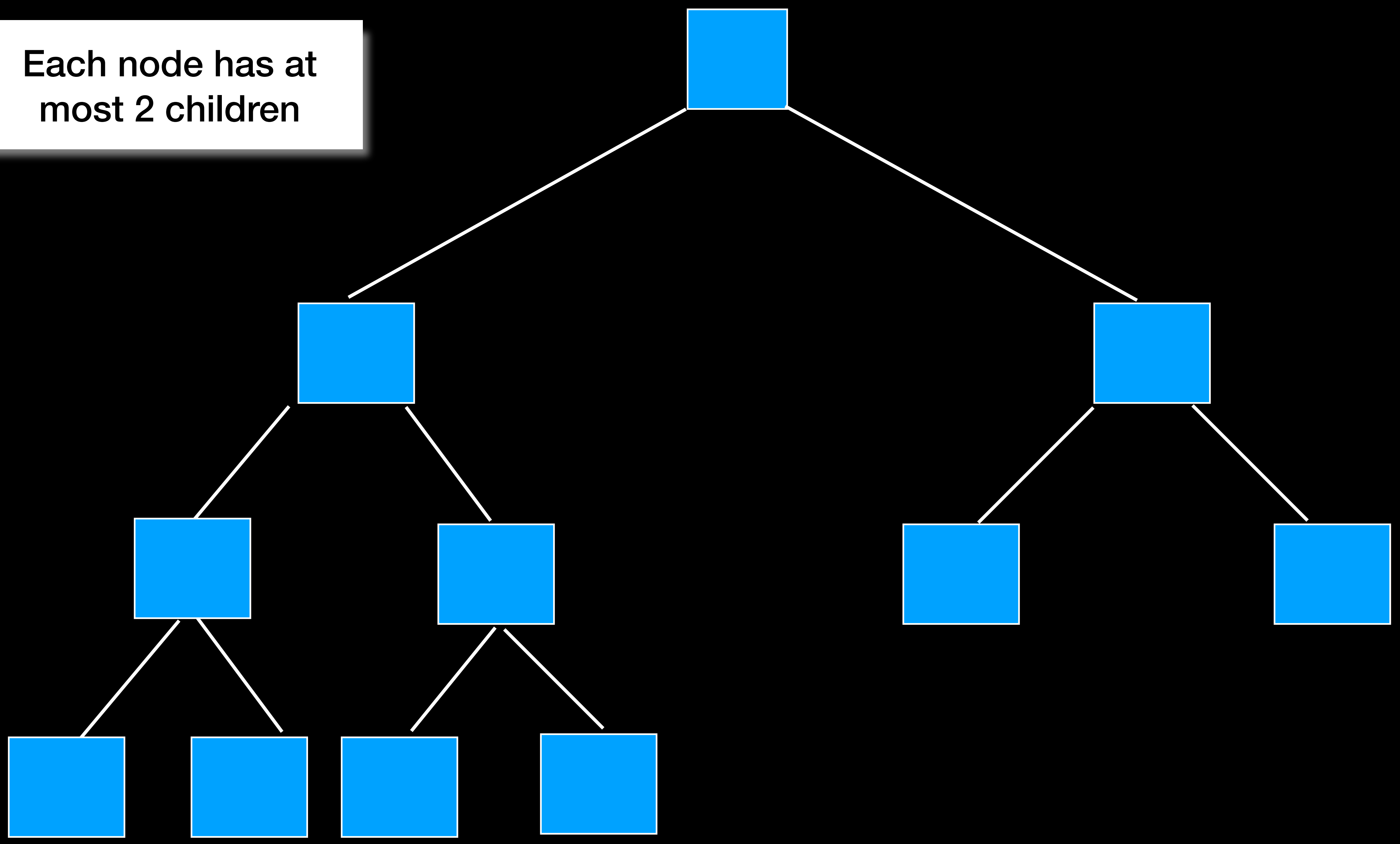
Path: a sequence of nodes c_1, c_2, ..., c_k where c_{i+1} is a child of c_i.

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.

Binary Tree

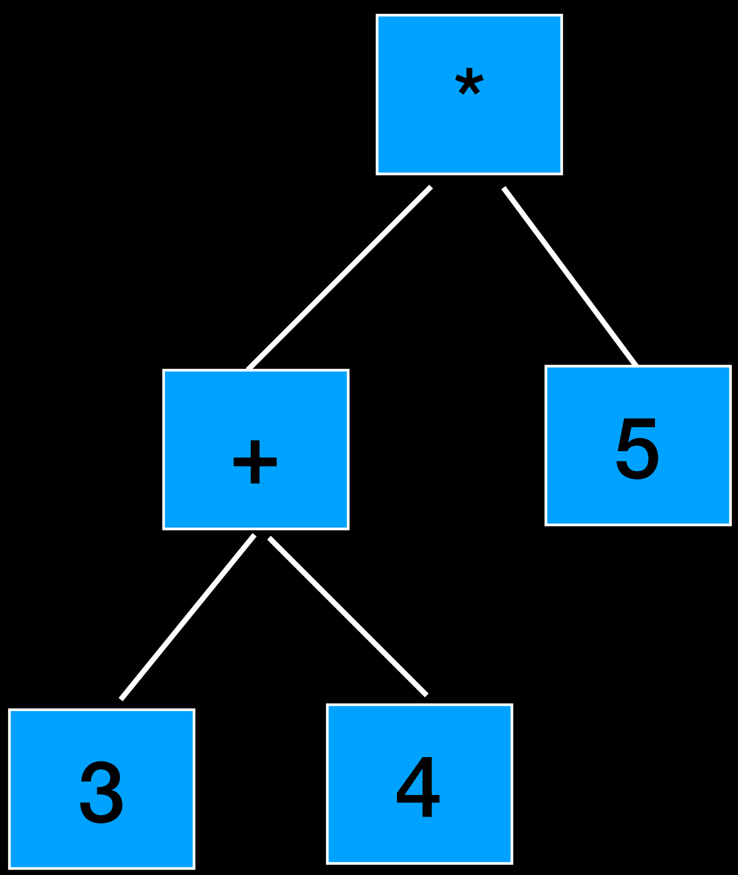
Each node has at most 2 children



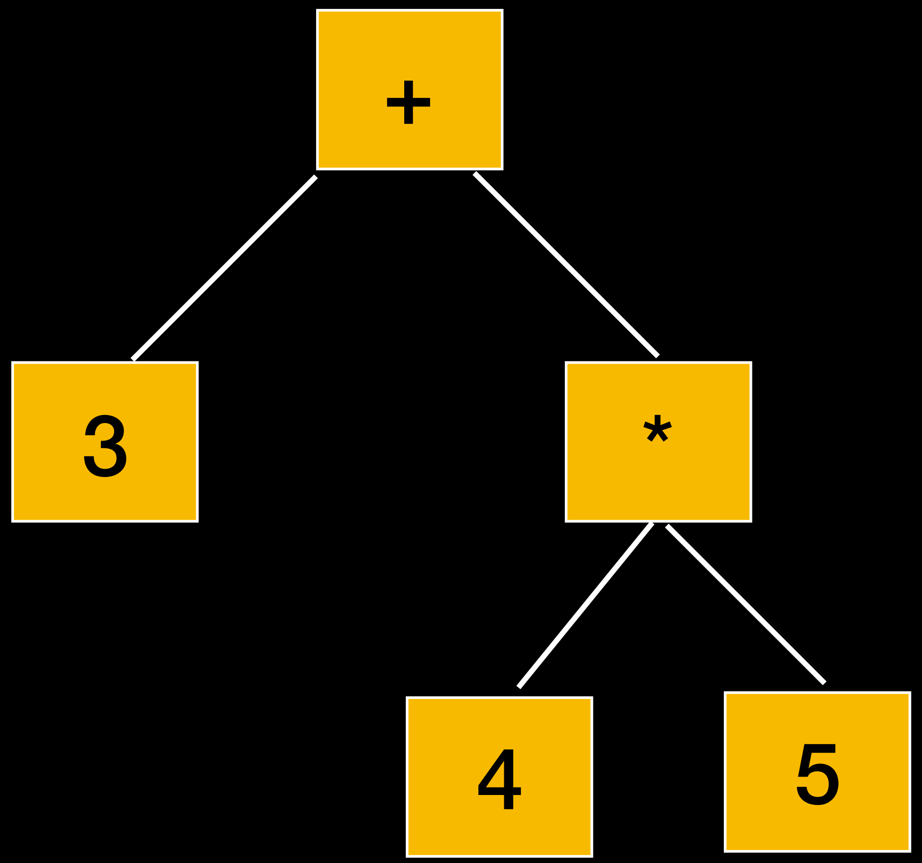
Binary Tree Applications

Algebraic Expressions

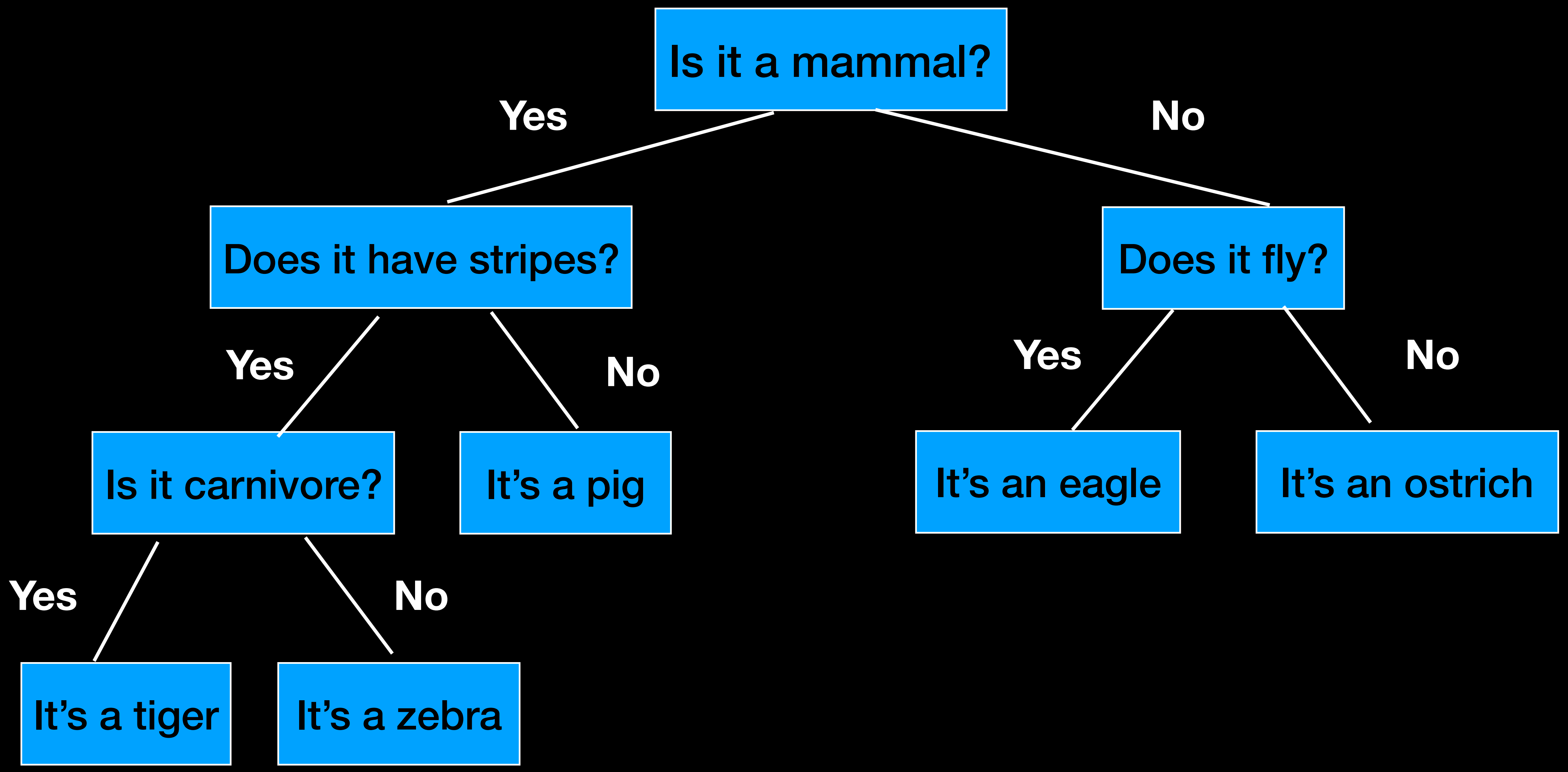
(3 + 4) * 5



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

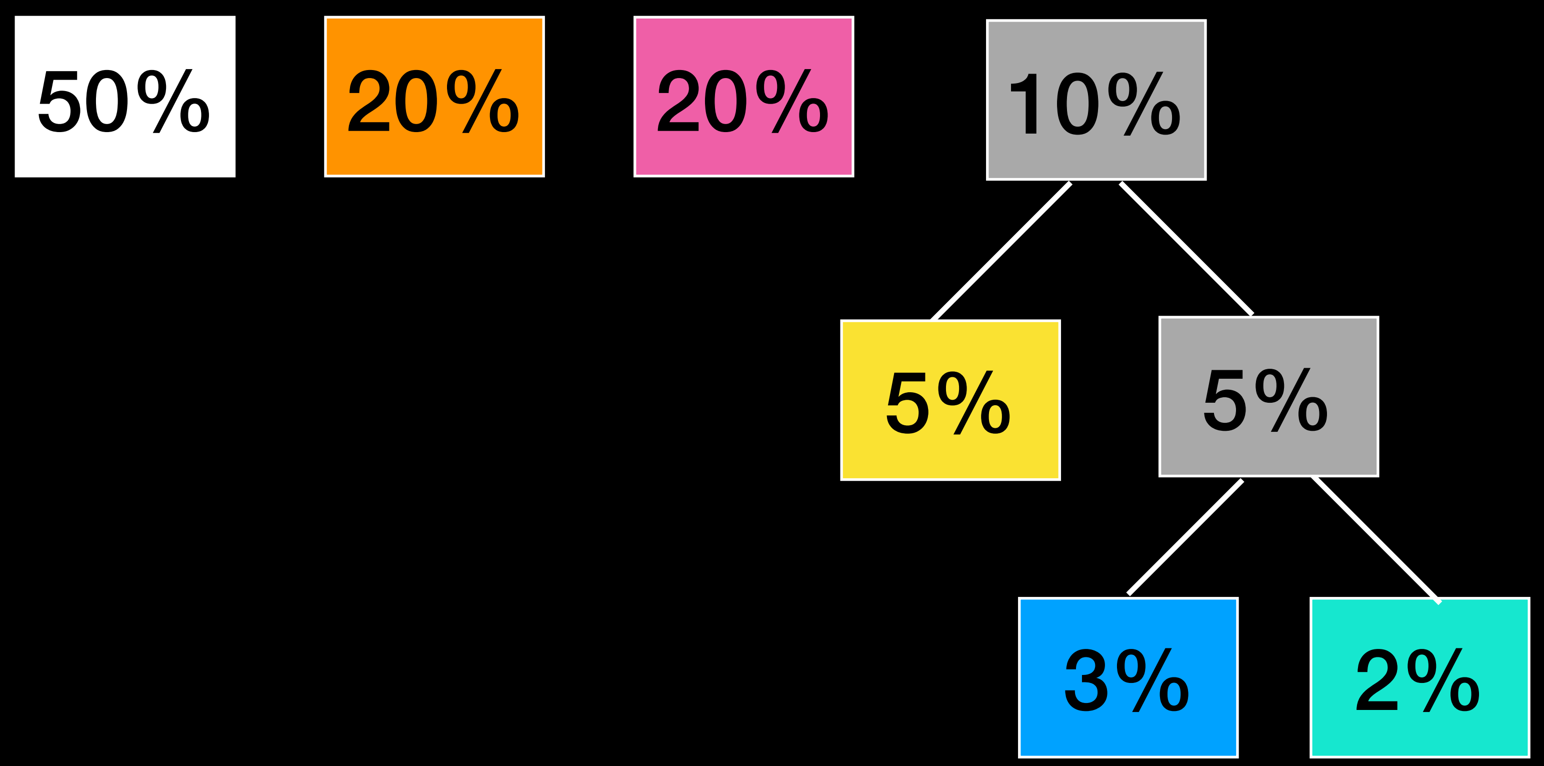
No codeword is prefix to another codeword (i.e. if a symbol is encoded as 00 no other codeword can start with 00)

Huffman Tree

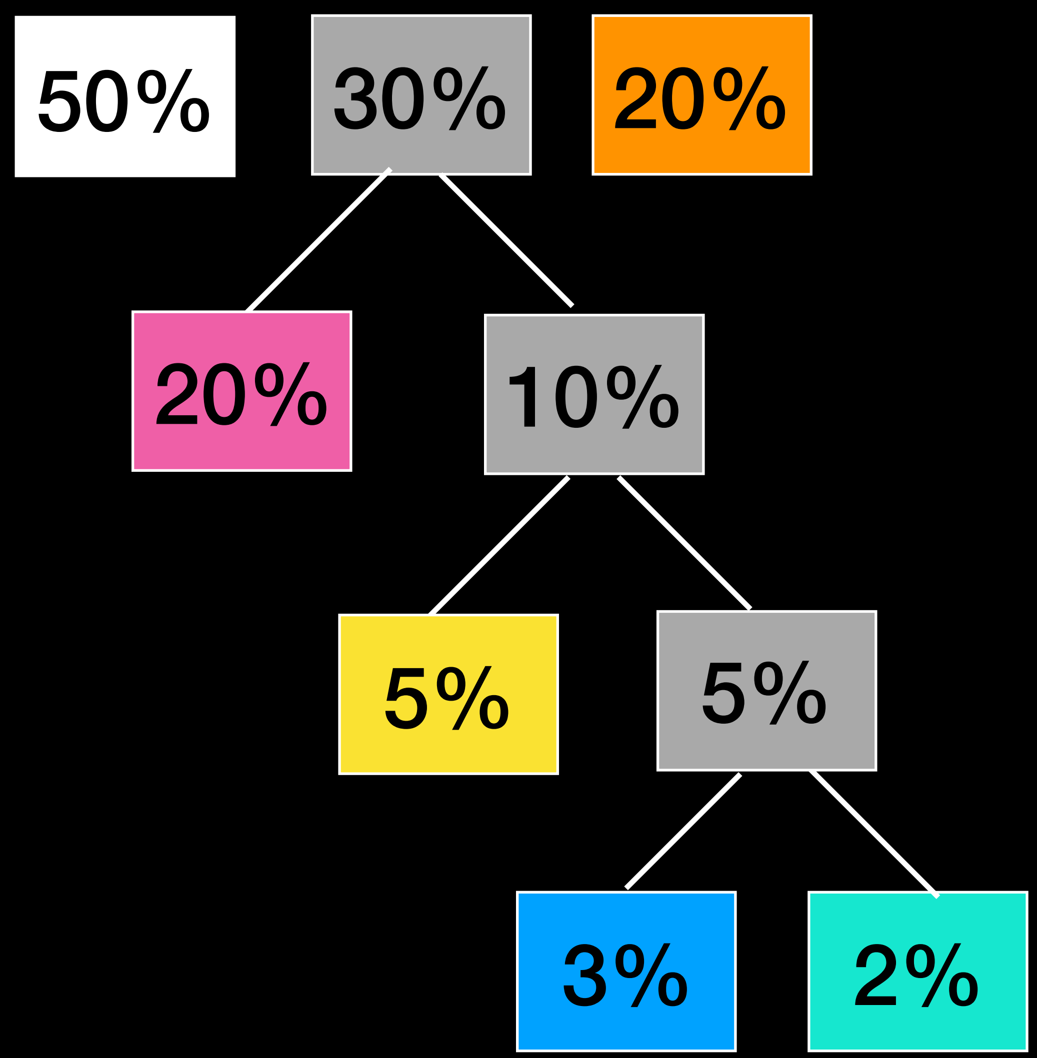
50%	20%	20%	5%	3%	2%

Huffman Tree

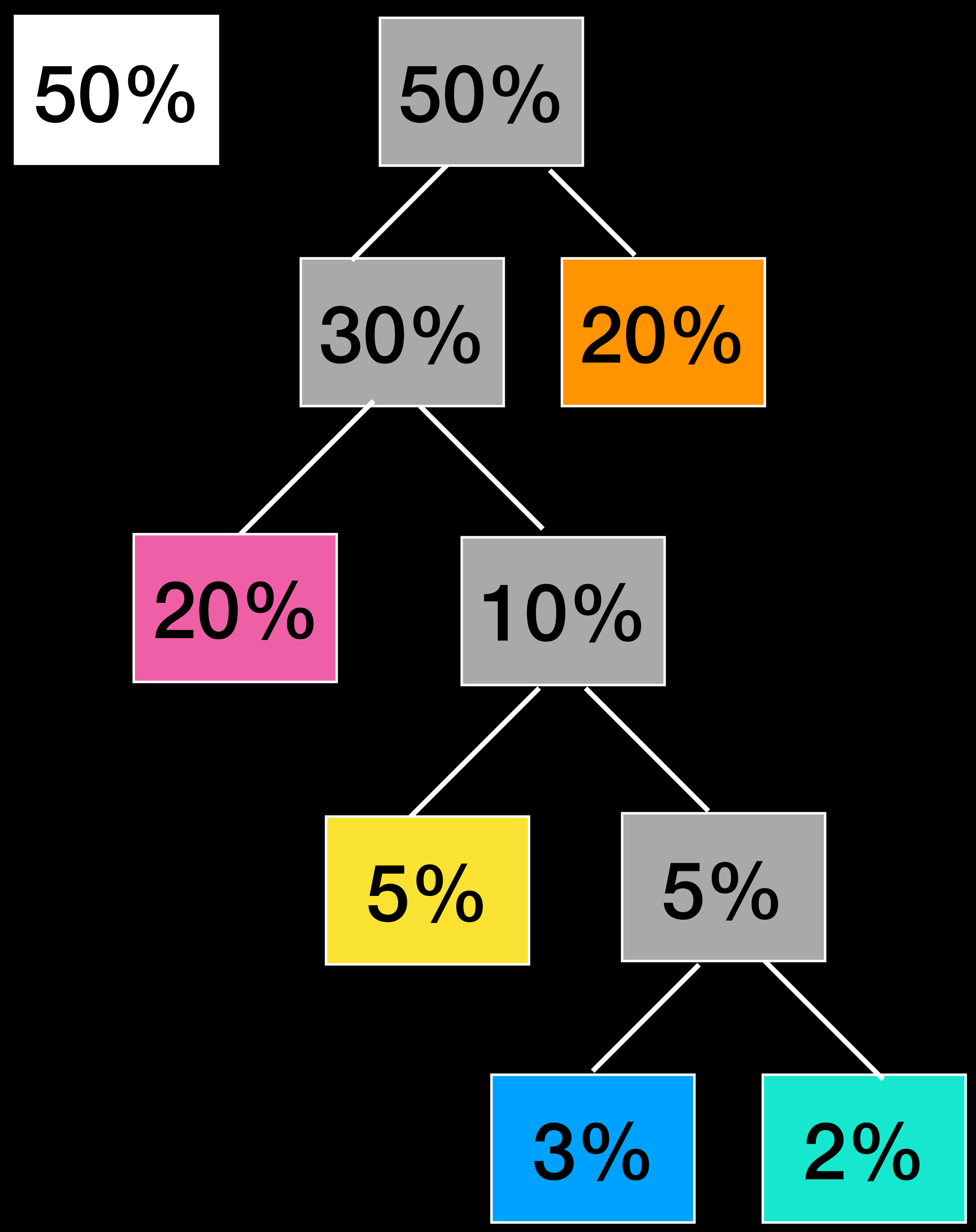
Huffman Tree



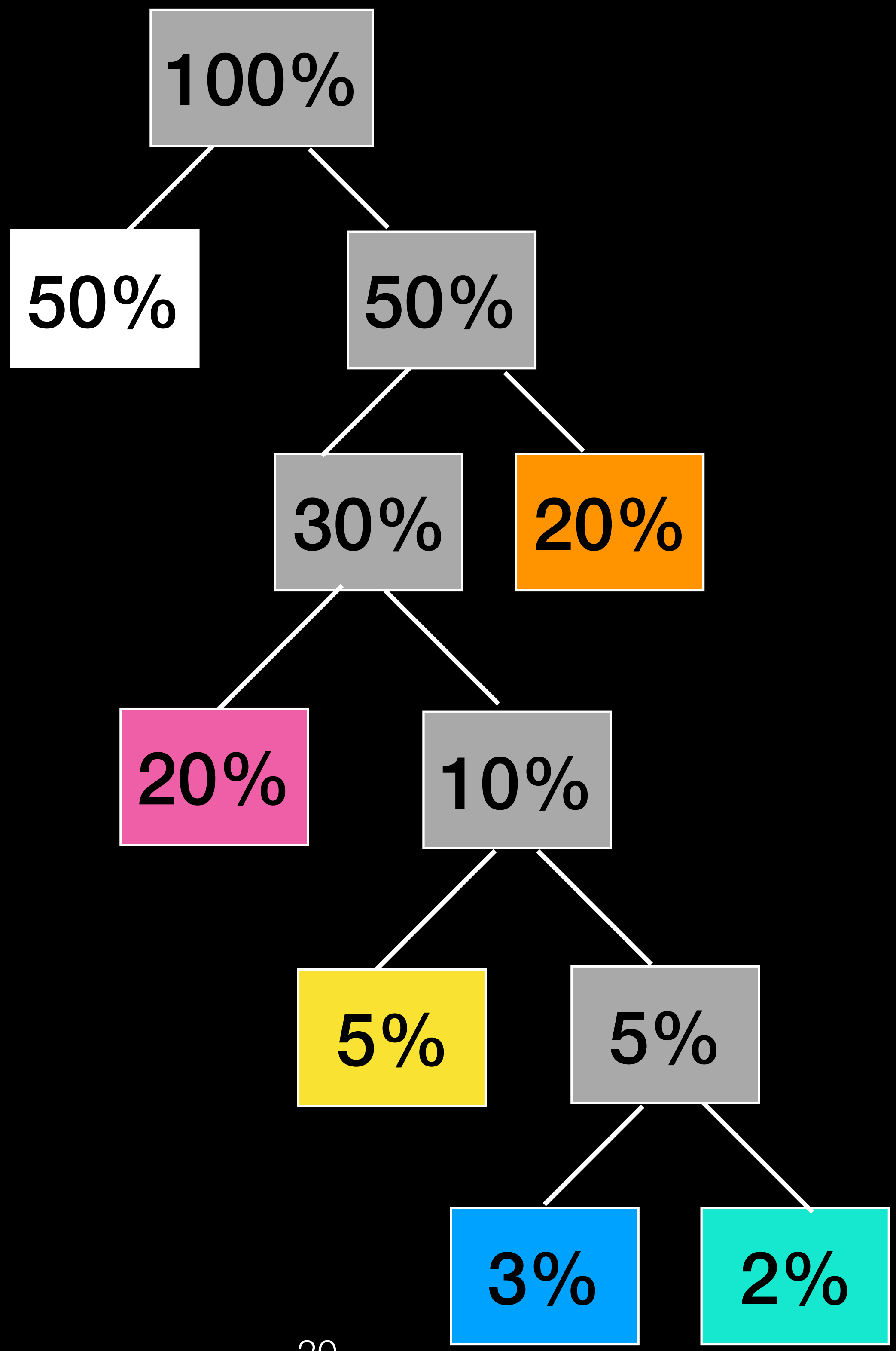
Huffman Tree



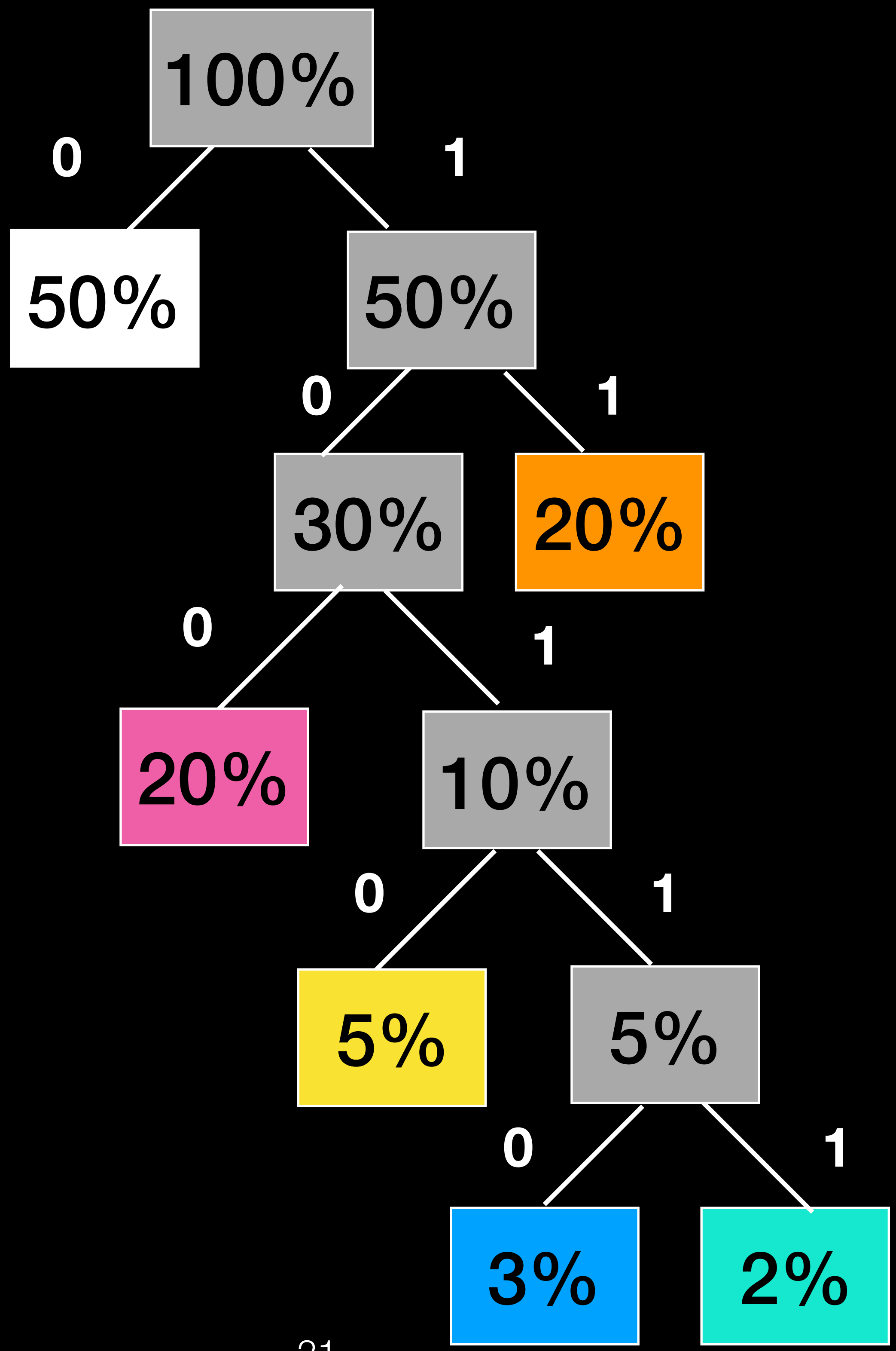
Huffman Tree



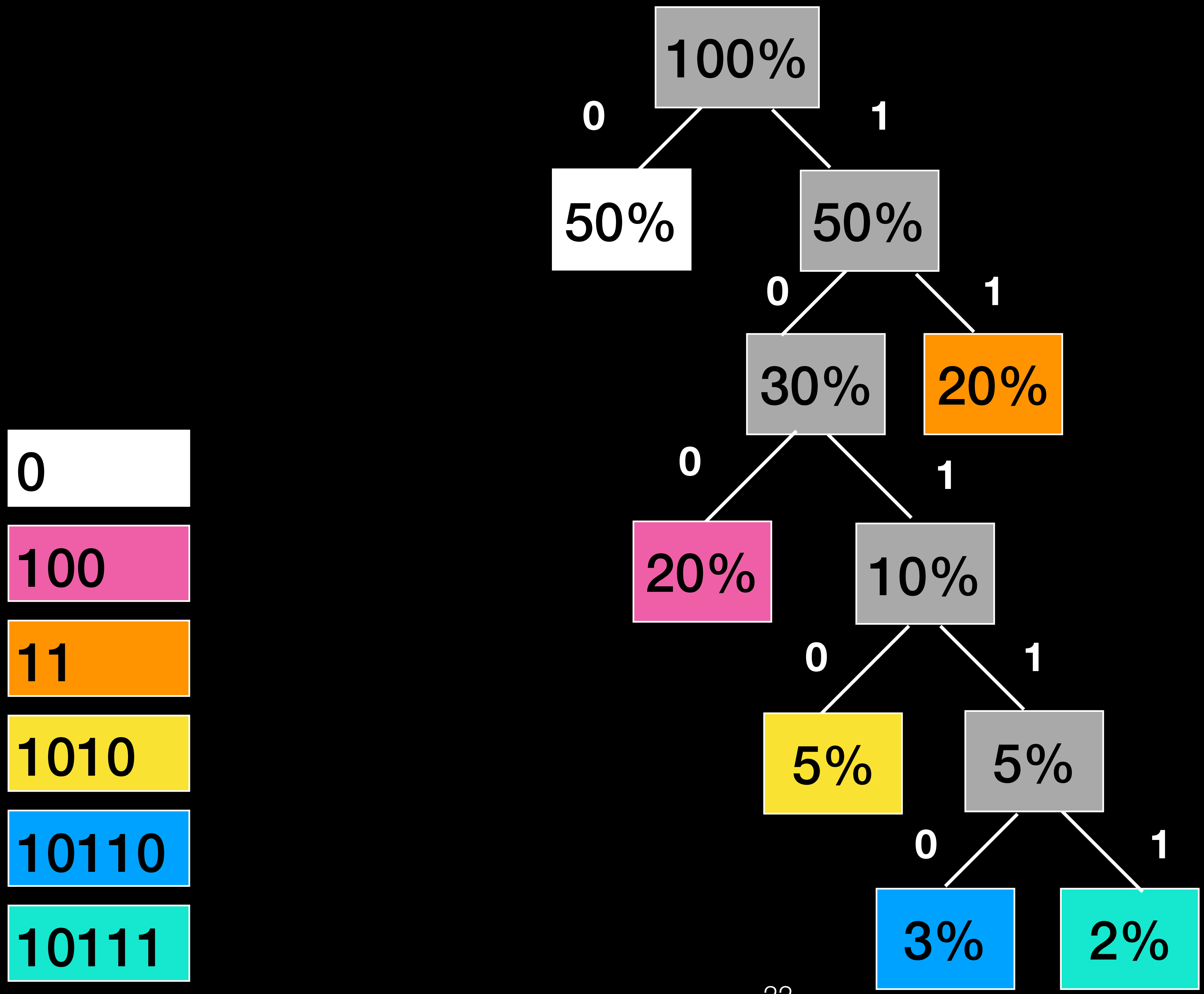
Huffman Tree



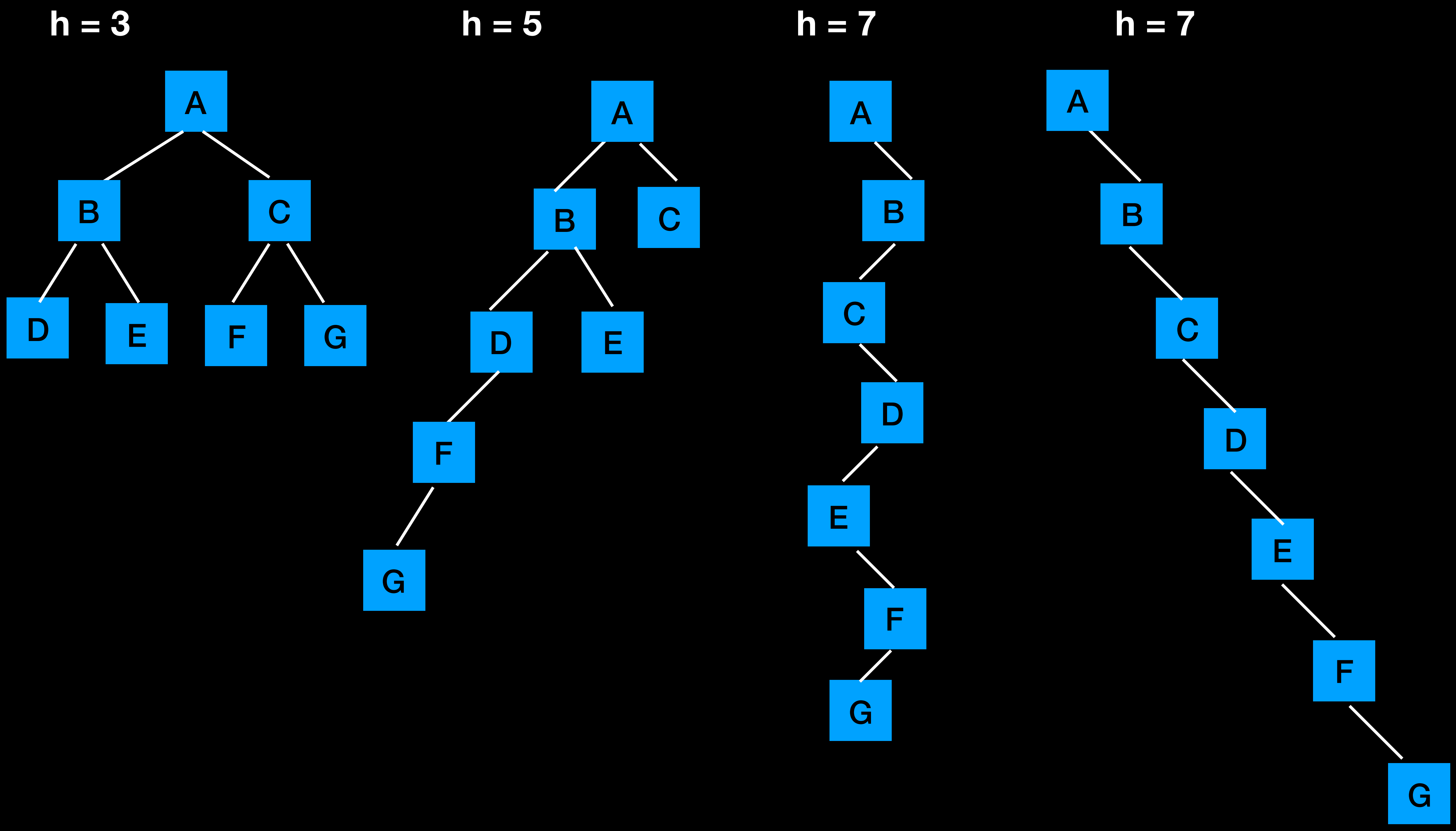
Huffman Tree



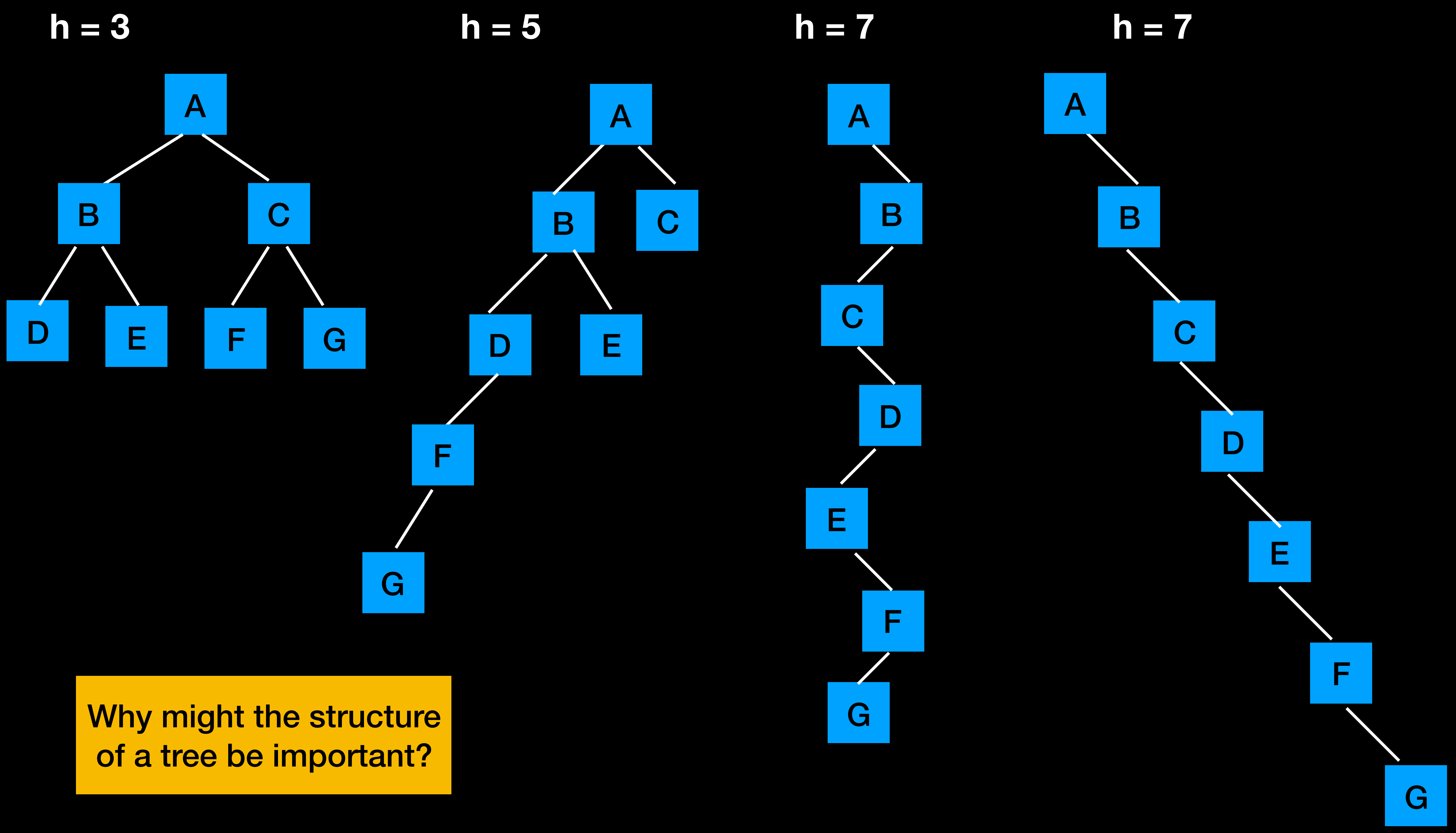
Huffman Tree



Tree Structure



Tree Structure



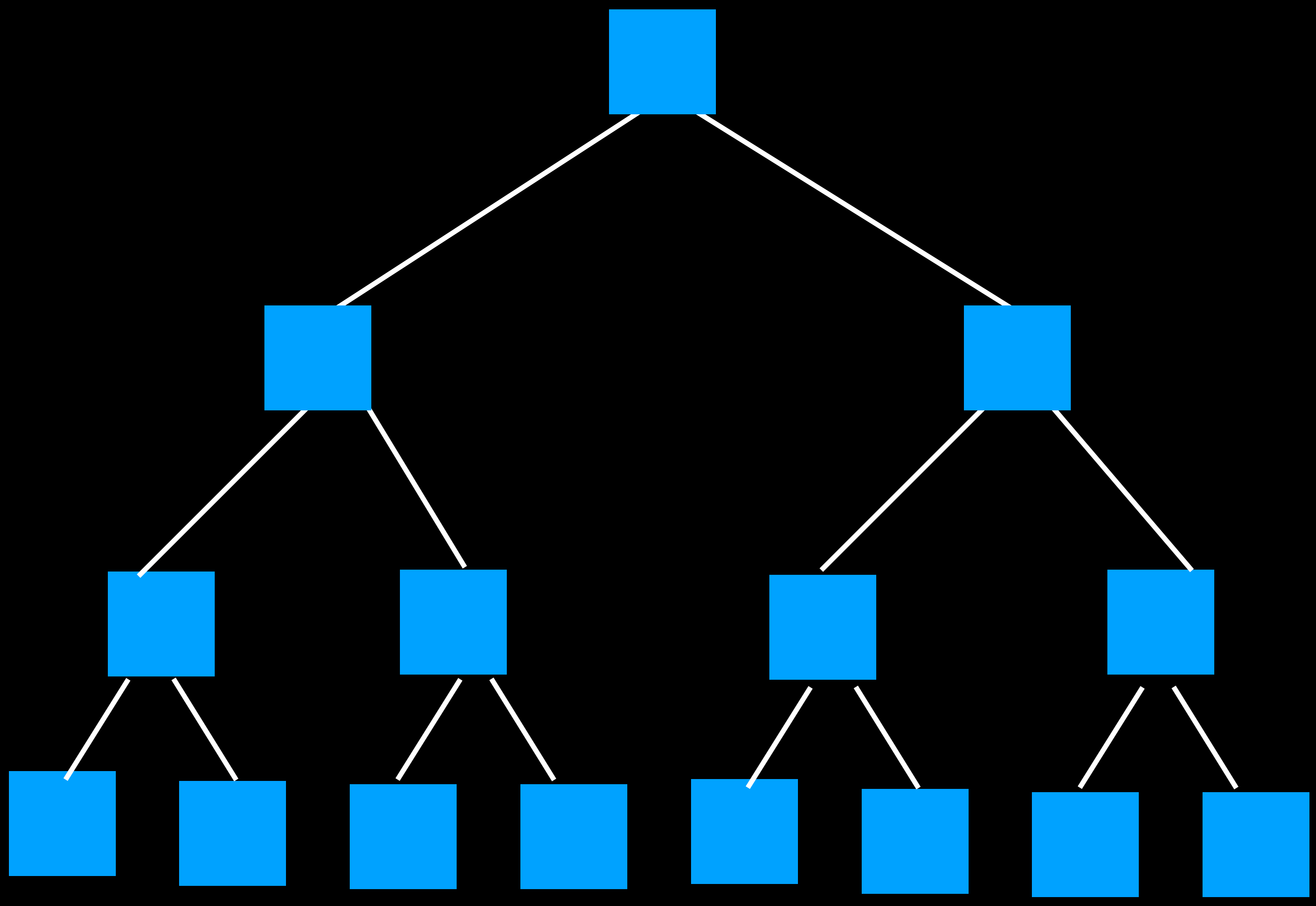
Why might the structure of a tree be important?

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*



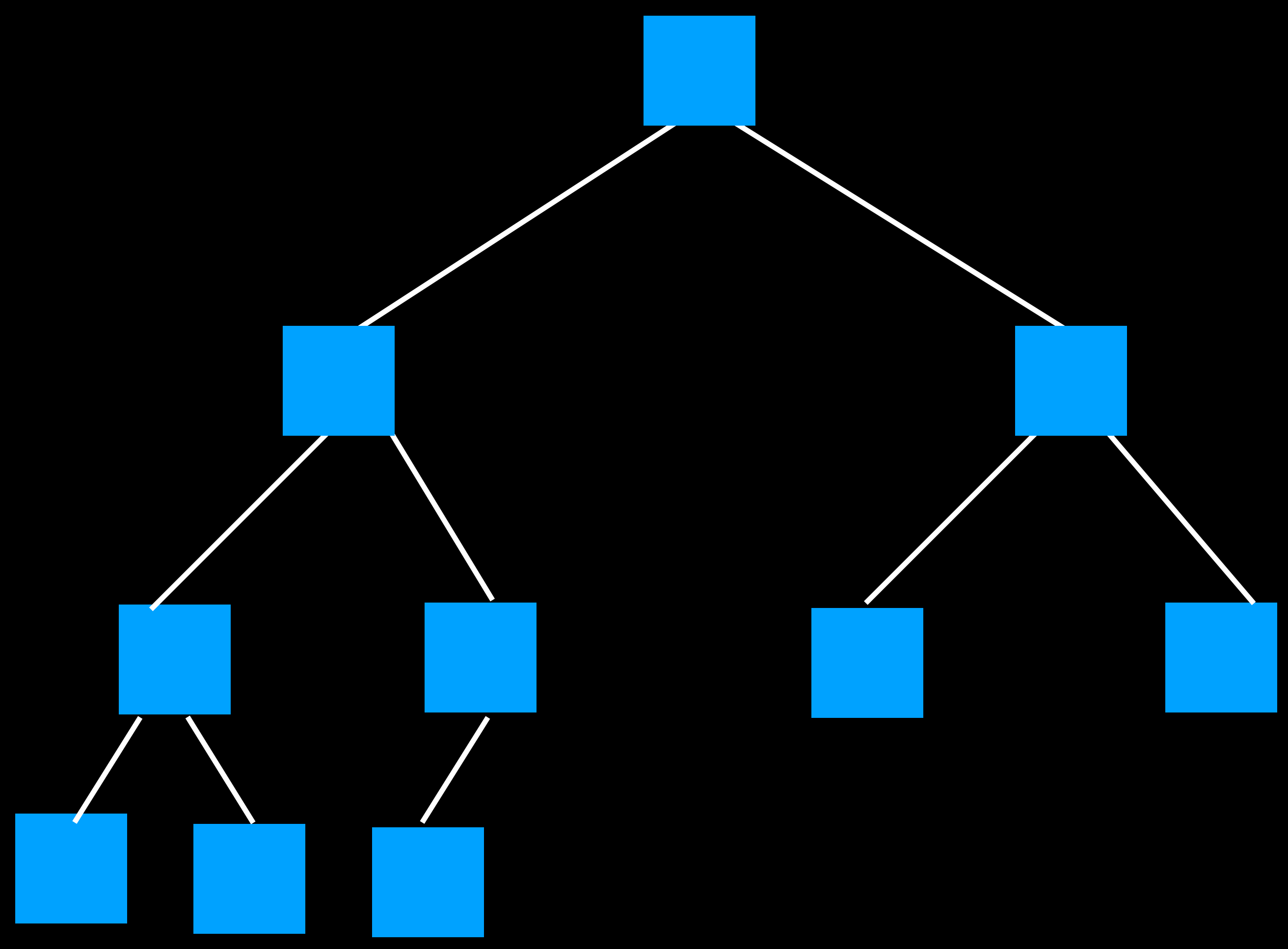
Complete Binary Tree

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

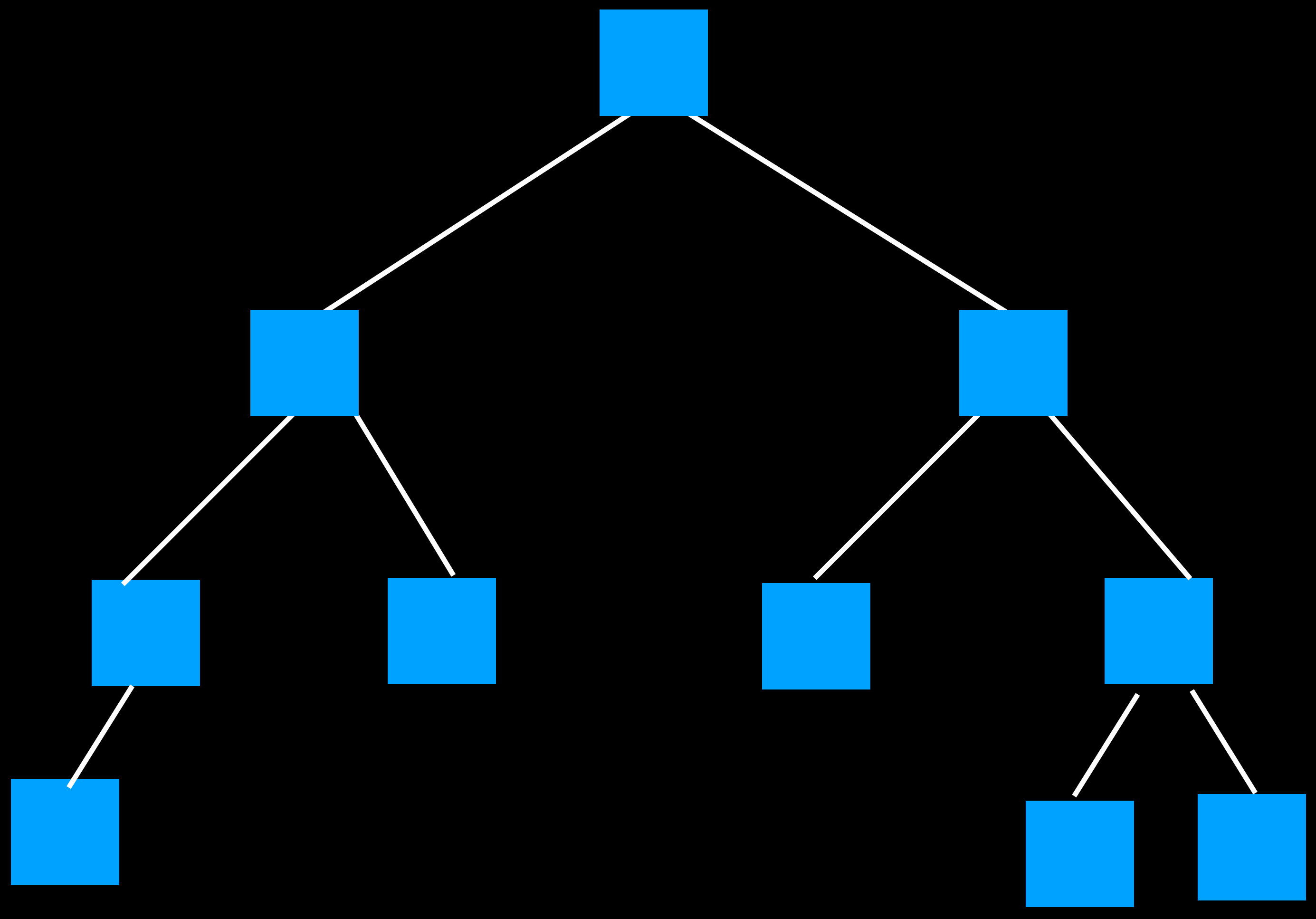
When a node at level *h-1* has one child, it is a left child



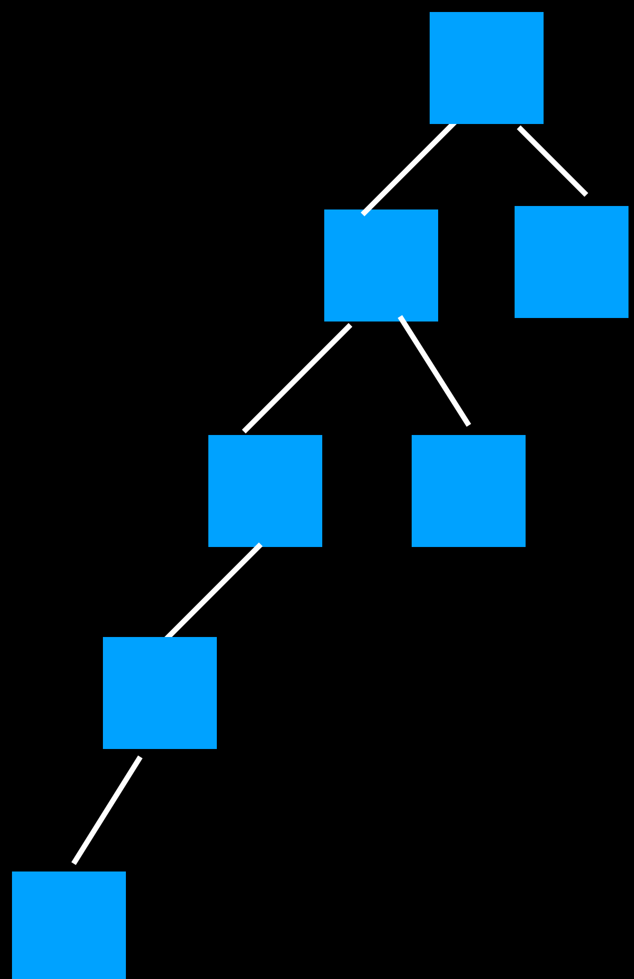
(Height) Balanced Binary Tree

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

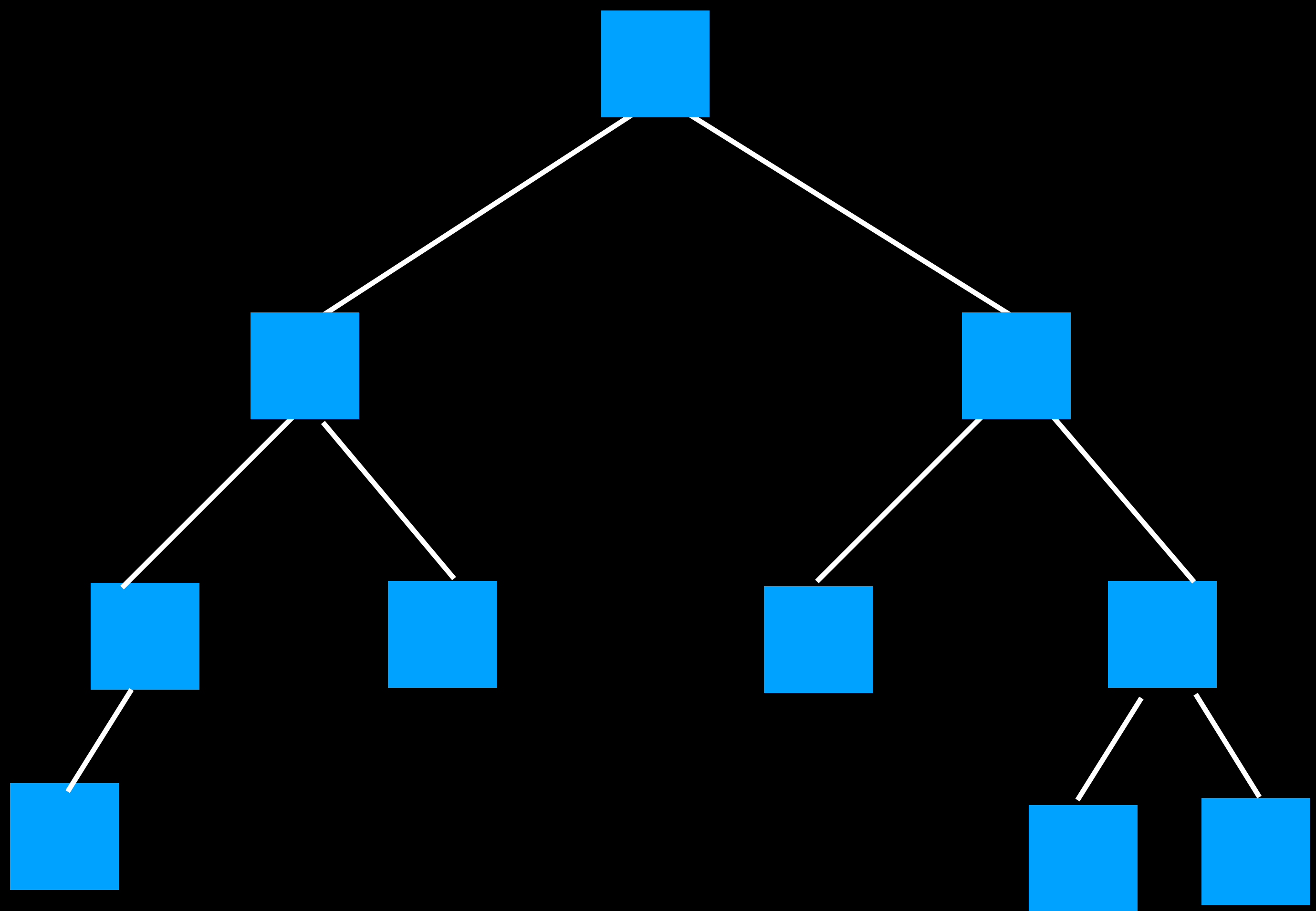
All paths from root to leaf differ in length by at most 1



Unbalanced



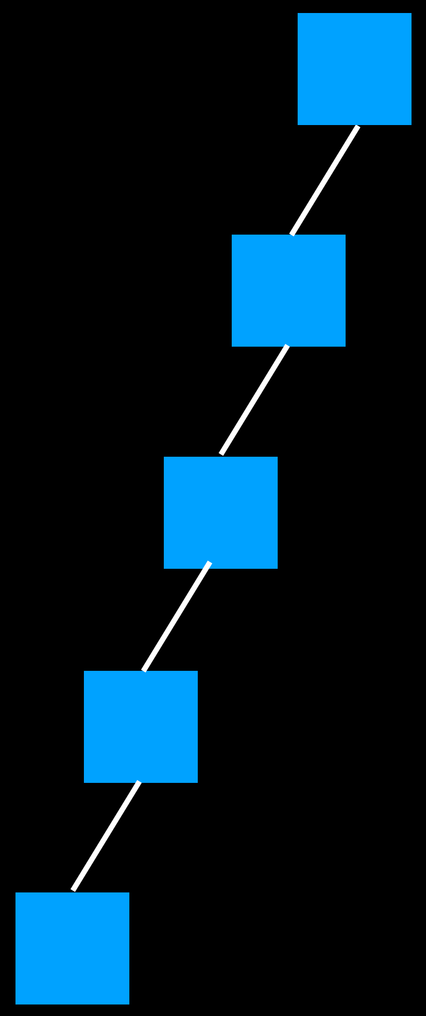
Balanced



Maximum Height

n nodes
every node 1 child
h = n

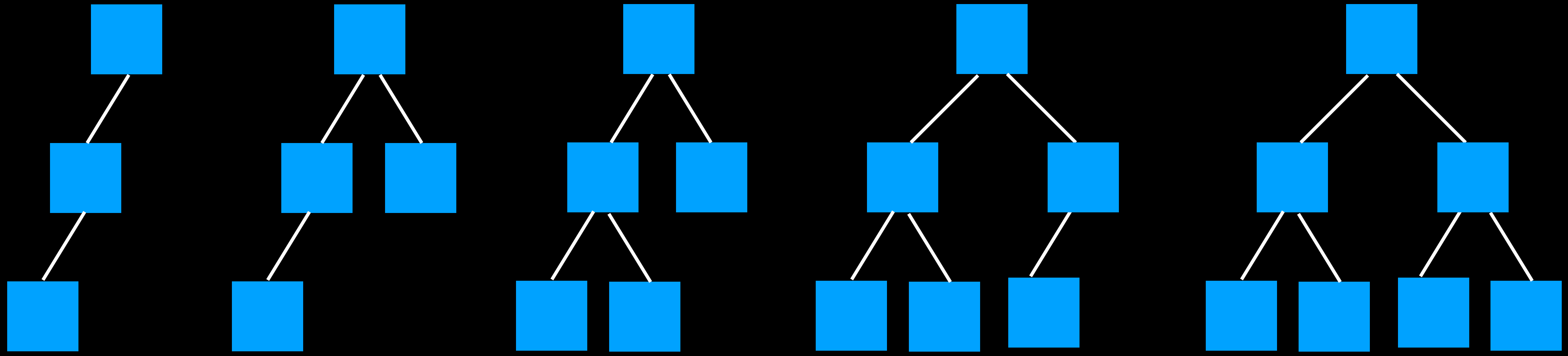
Essentially a chain

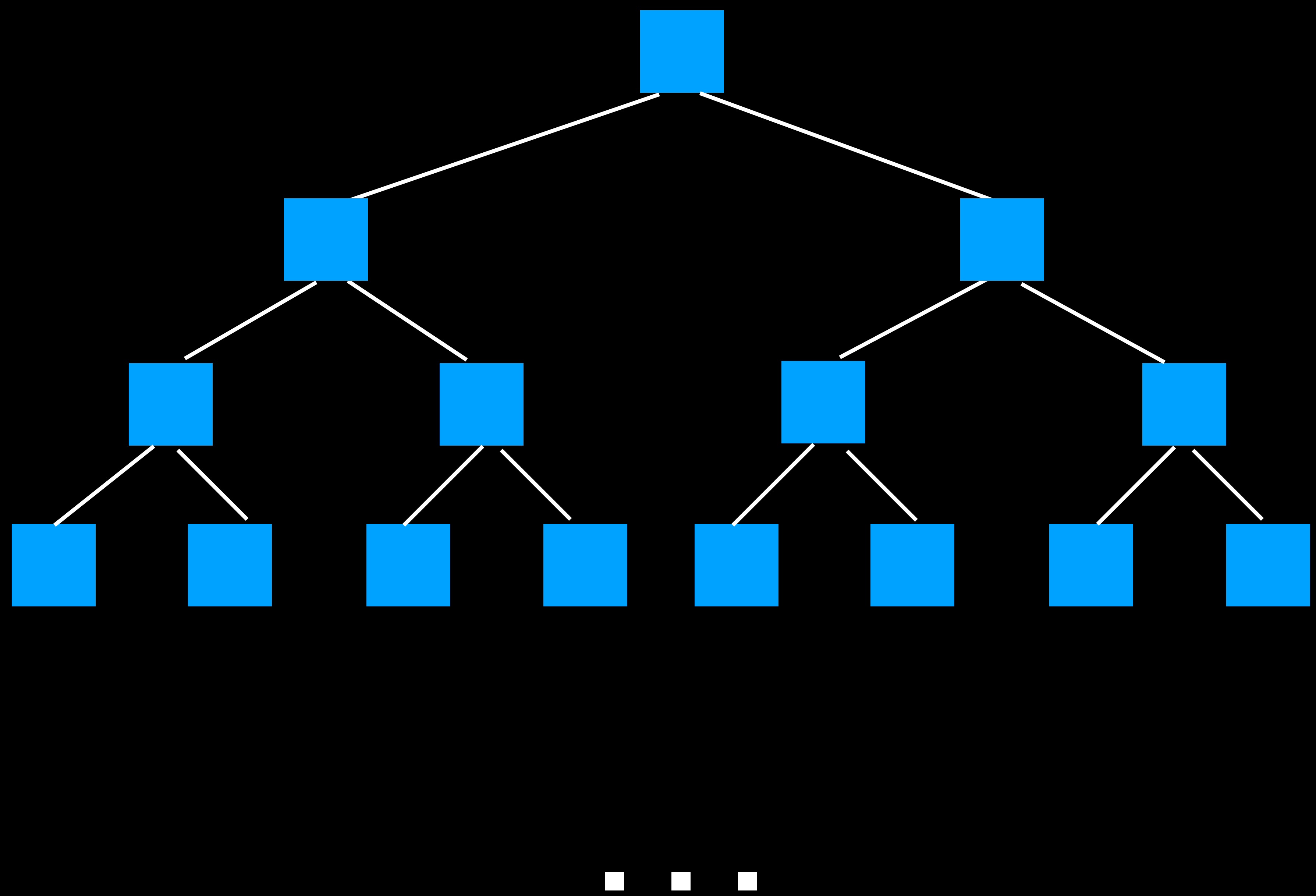


Minimum Height

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 $h \approx 10$ ($1000 \approx 1014 \approx 2^{10}$)
1000000 nodes $h \approx 20$ ($10^6 \approx 2^{20}$)

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





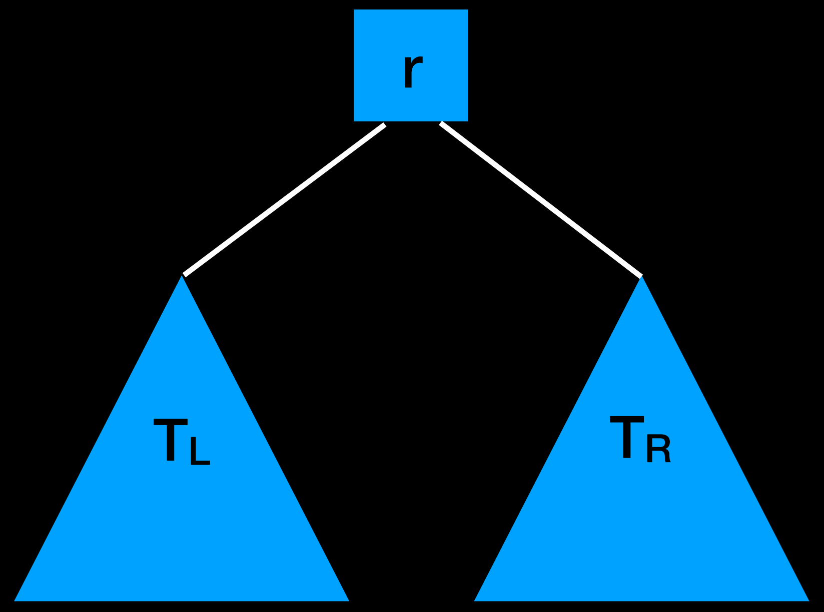
h	n @ level	Total n
1	$1 = 2^0$	$1 = 2^1 - 1$
2	$2 = 2^1$	$3 = 2^2 - 1$
3	$4 = 2^2$	$7 = 2^3 - 1$
4	$8 = 2^3$	$15 = 2^4 - 1$
h	2^{h-1}	$2^h - 1$

Binary Tree Traversals

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

Essentially visit the root as well as it's subtrees

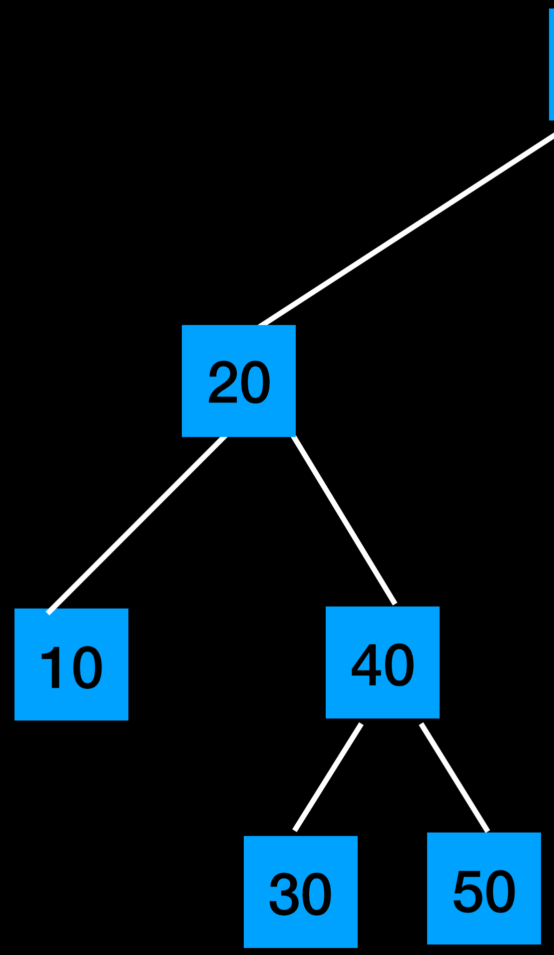
Order matters!!!

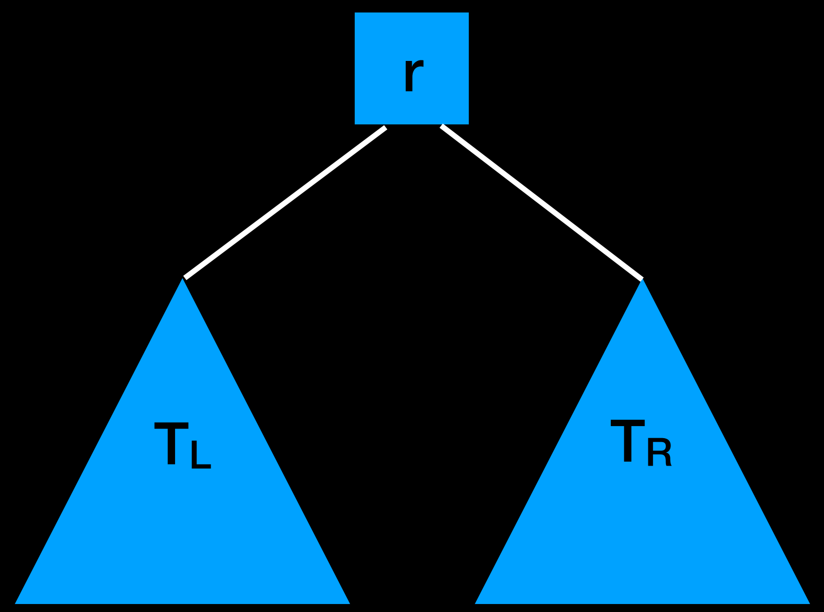


Visit (retrieve, print, modify …) every node in the tree

Preorder Traversal:

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



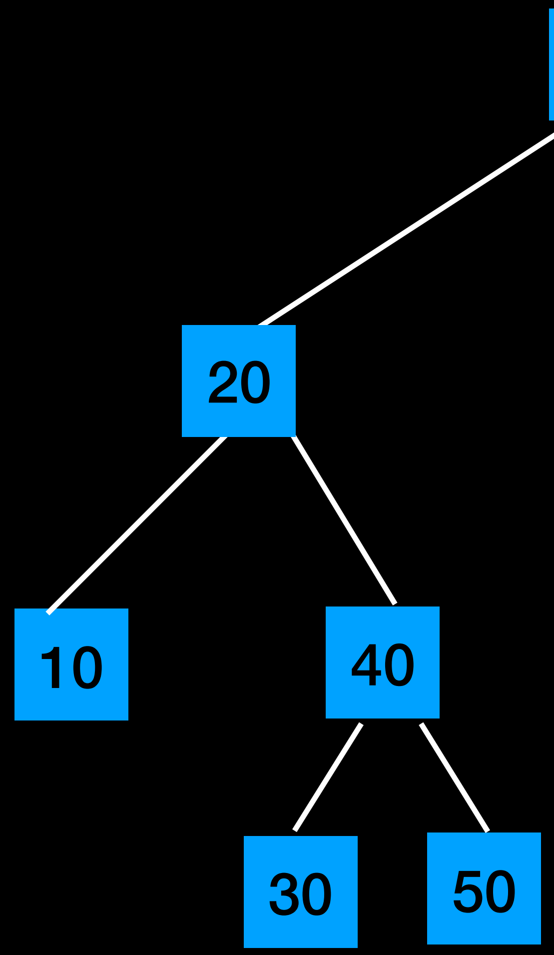


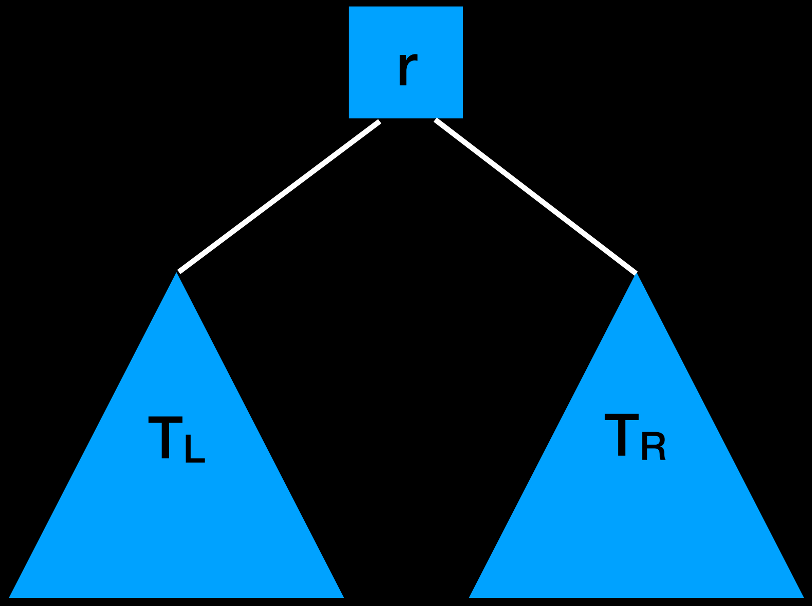
Preorder: **60, 20, 10, 40, 30, 50, 70**

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

Inorder Traversal:

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



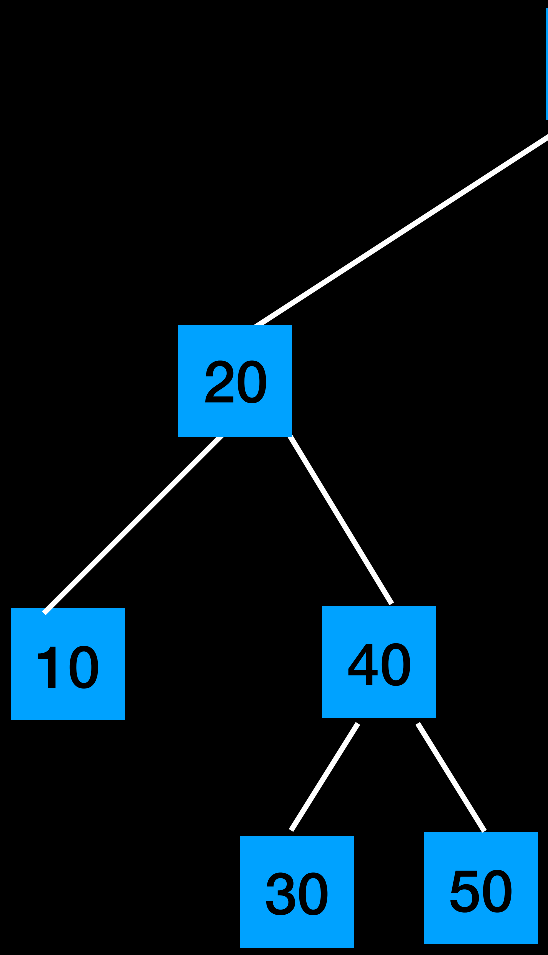


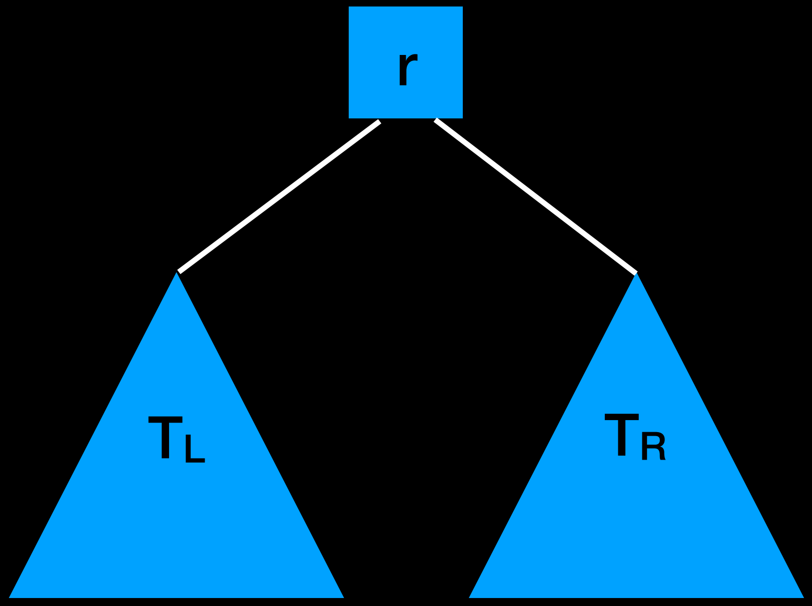
Inorder: 10, 20, 30, 40, 50, 60, 70

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

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<ItemType>& 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

Considerations

Recall

Remember our Set ADT?

- Array implementation
- Linked Chain implementation

Find an element: O(n)

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

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

Recall

Can we do better?

Remember our Set ADT?

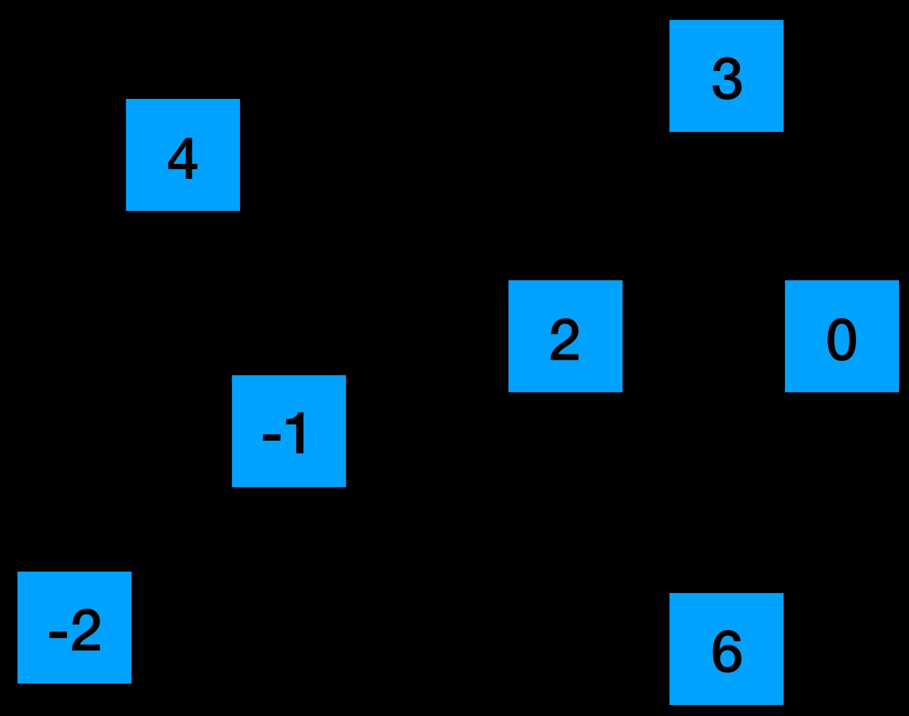
- Array implementation
- Linked Chain implementation

Find an element: O(n)

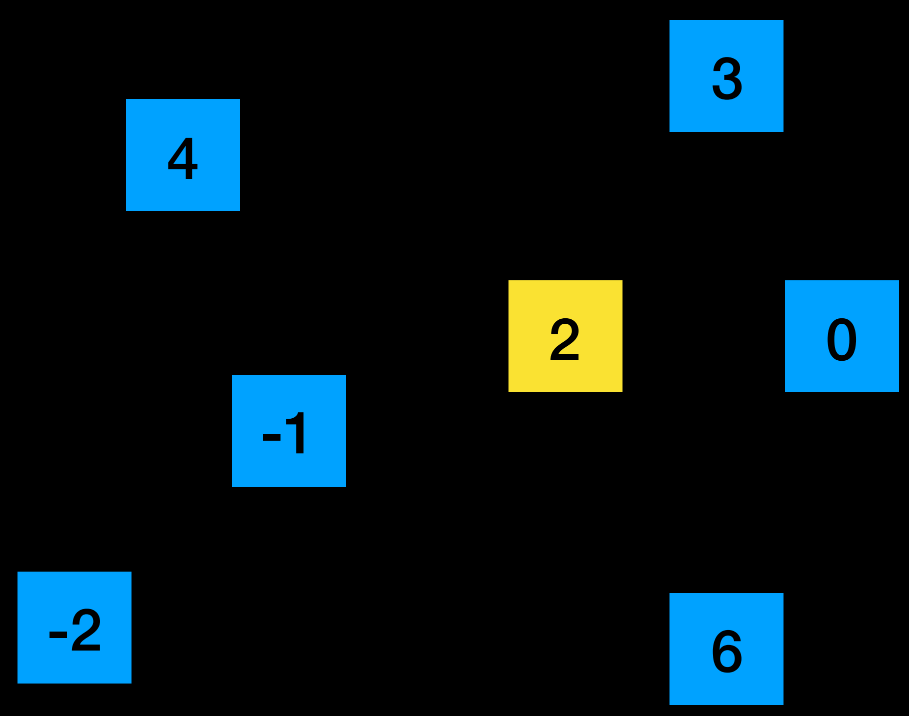
Add: Check if element is there and if not add it O(n)

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

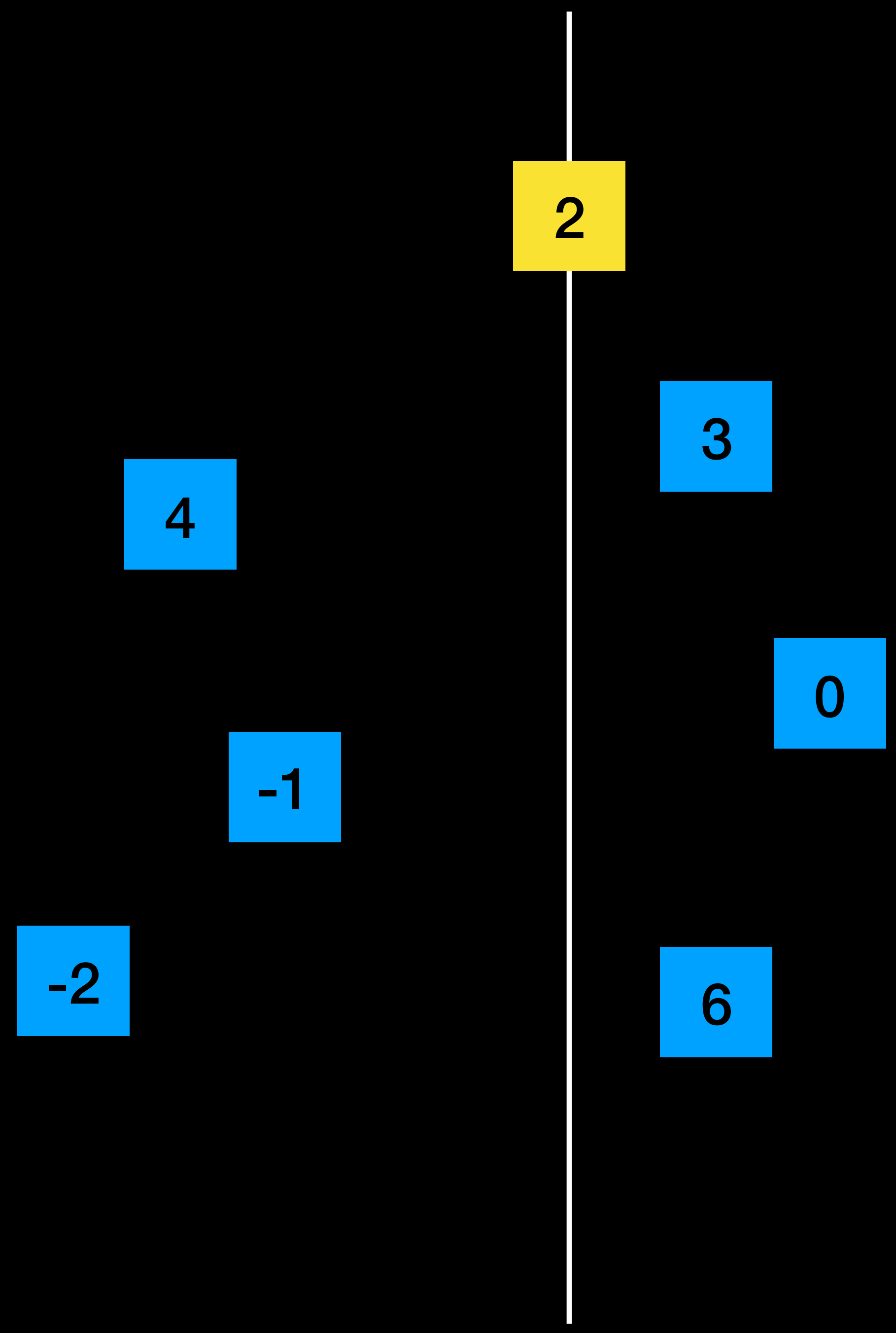
A Different Approach



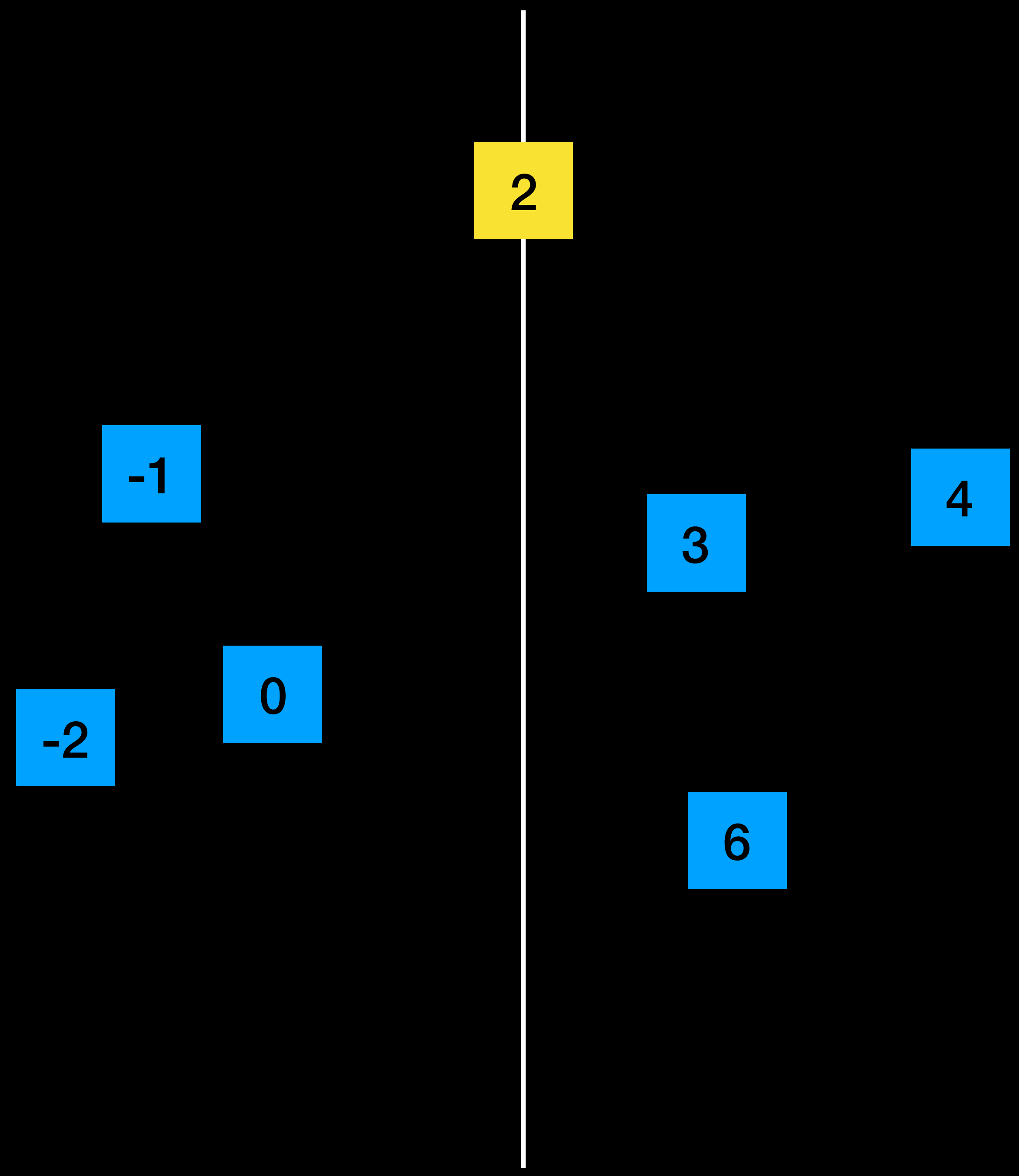
A Different Approach



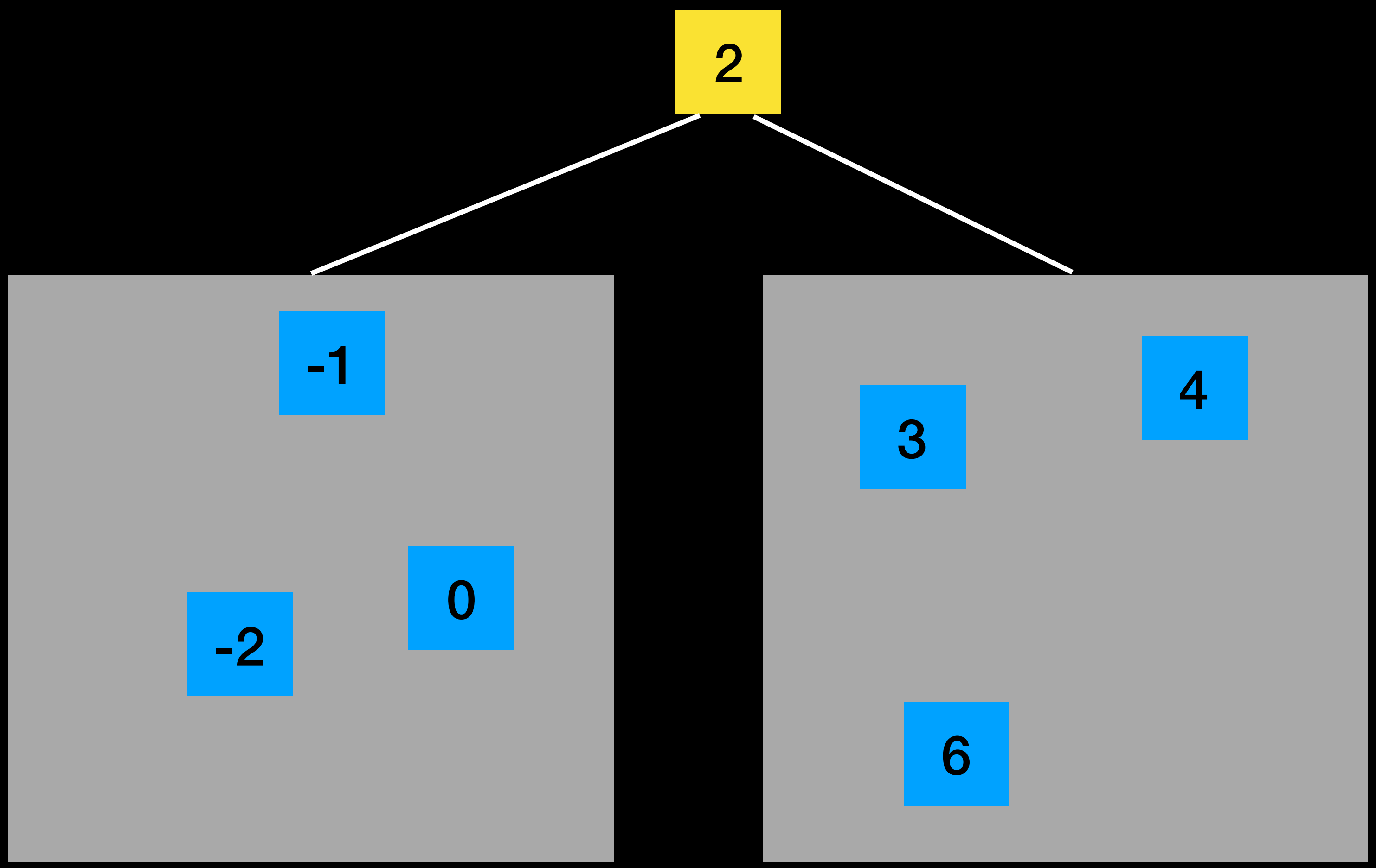
A Different Approach



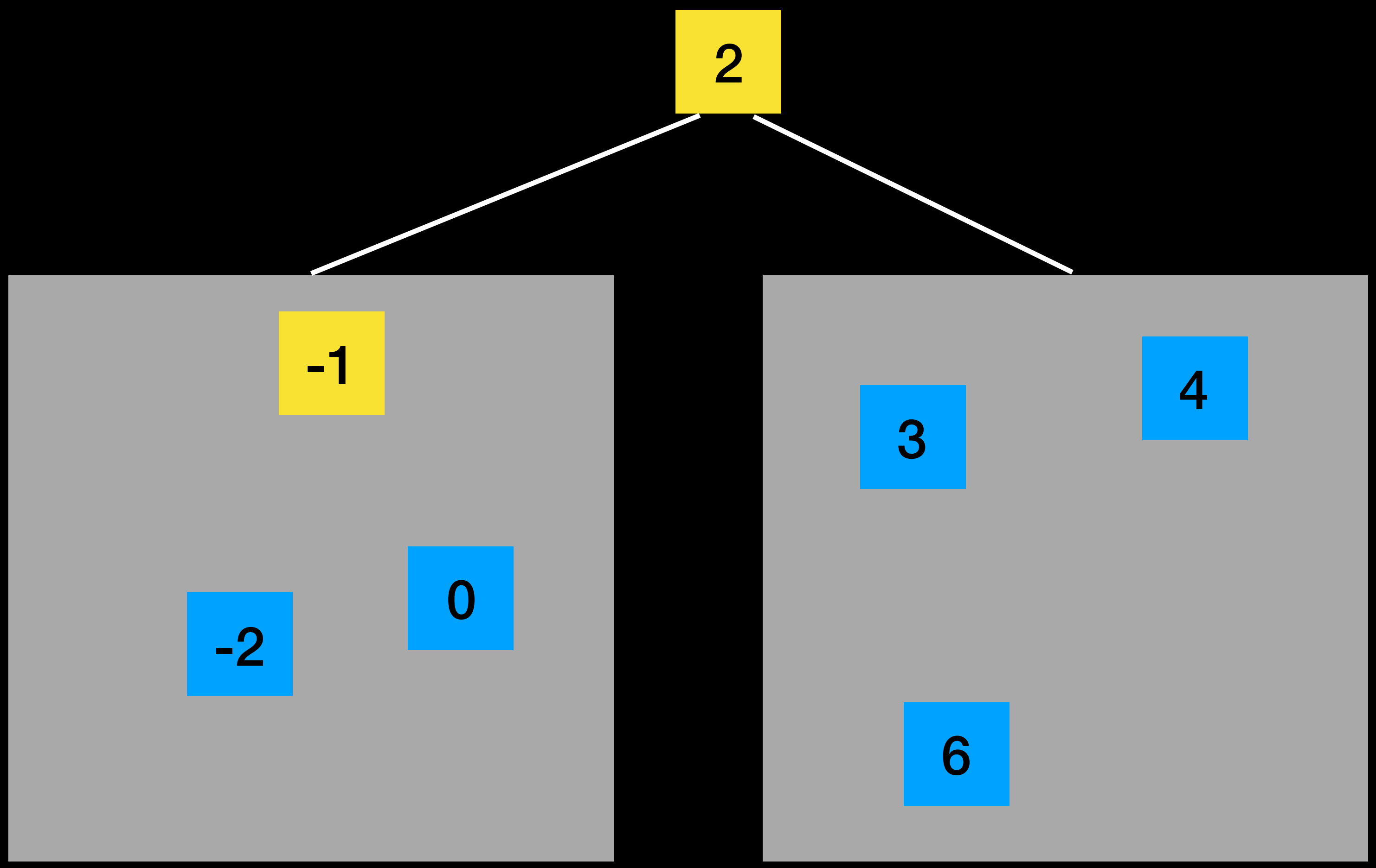
A Different Approach



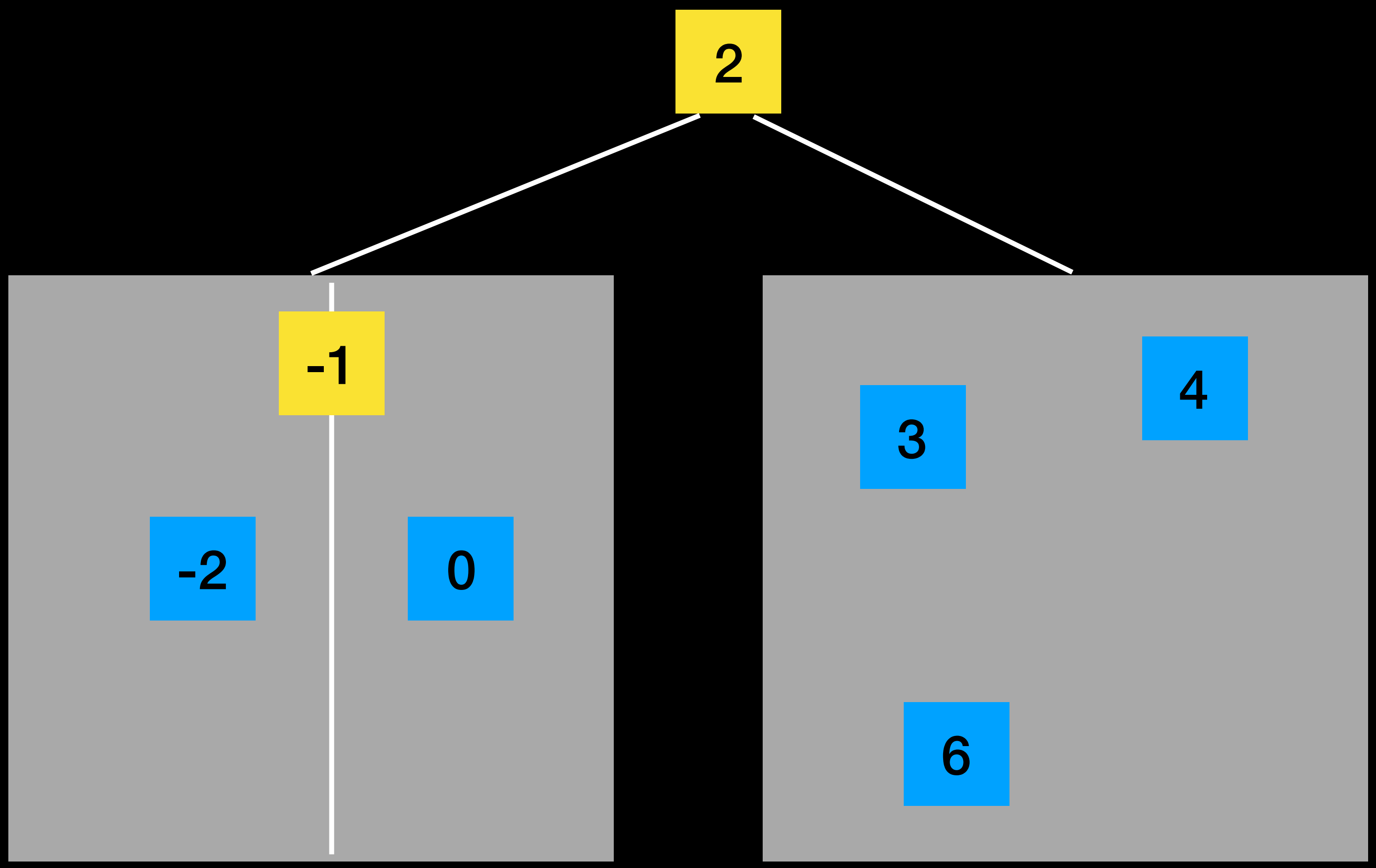
A Different Approach



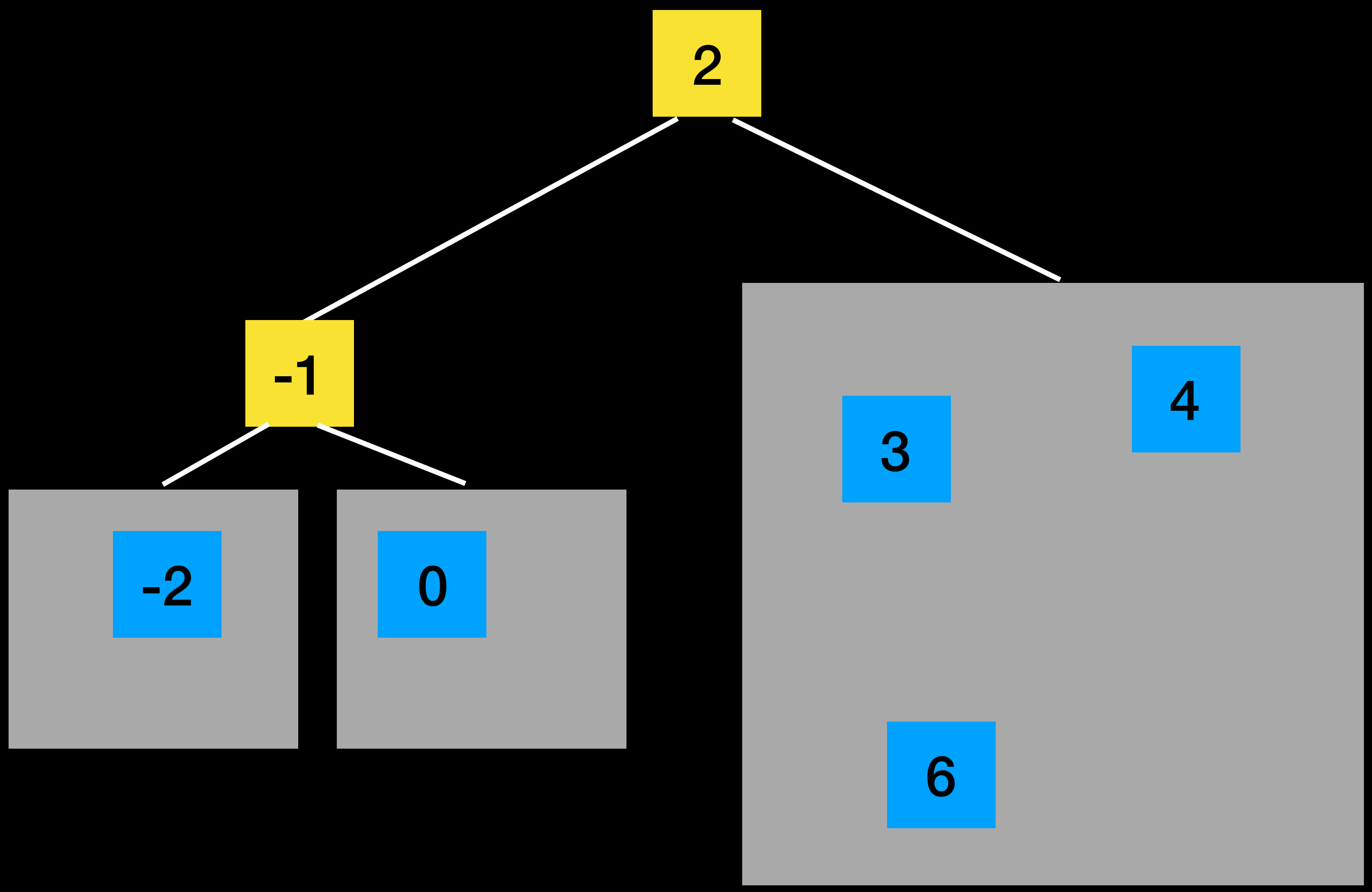
A Different Approach



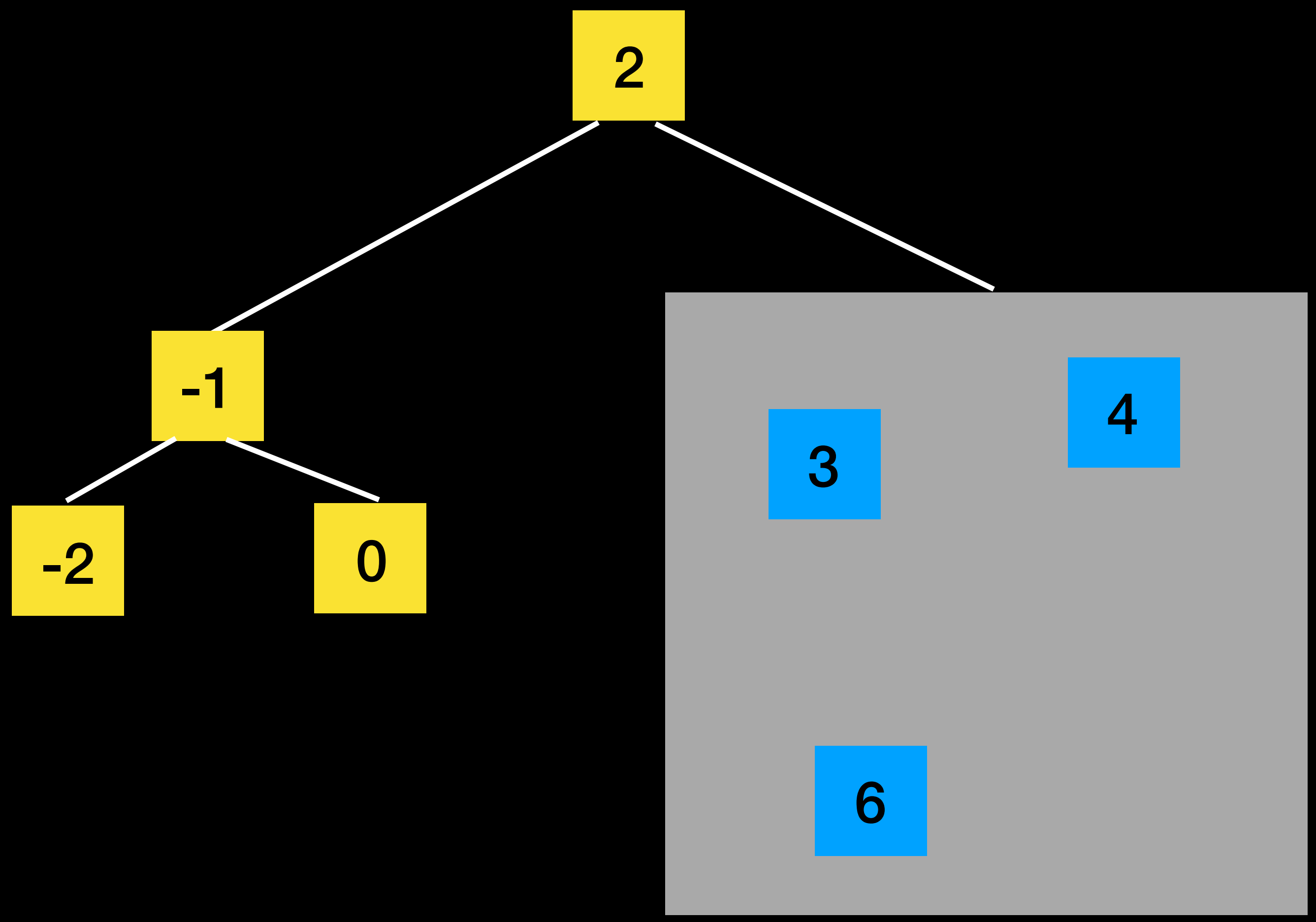
A Different Approach



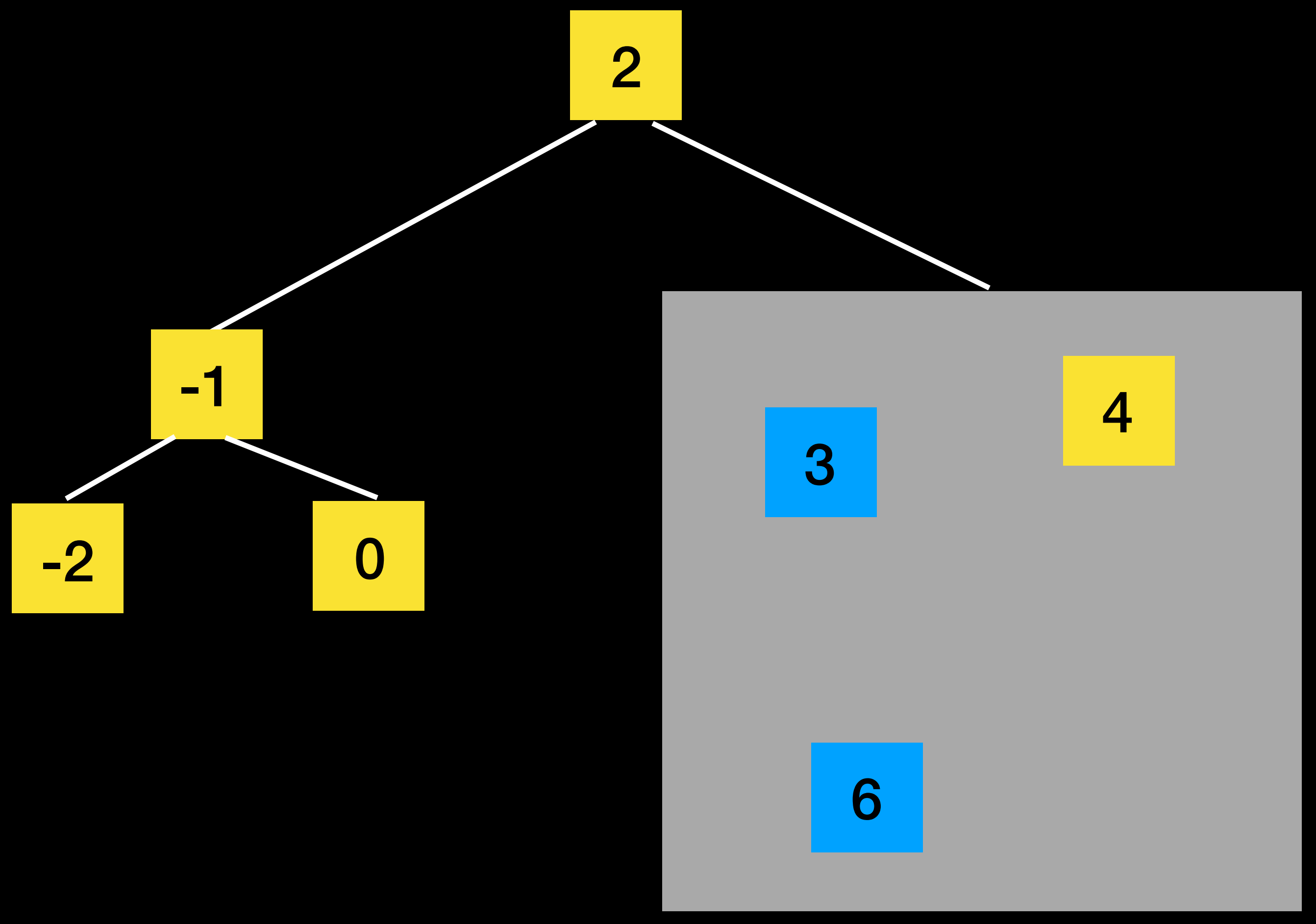
A Different Approach



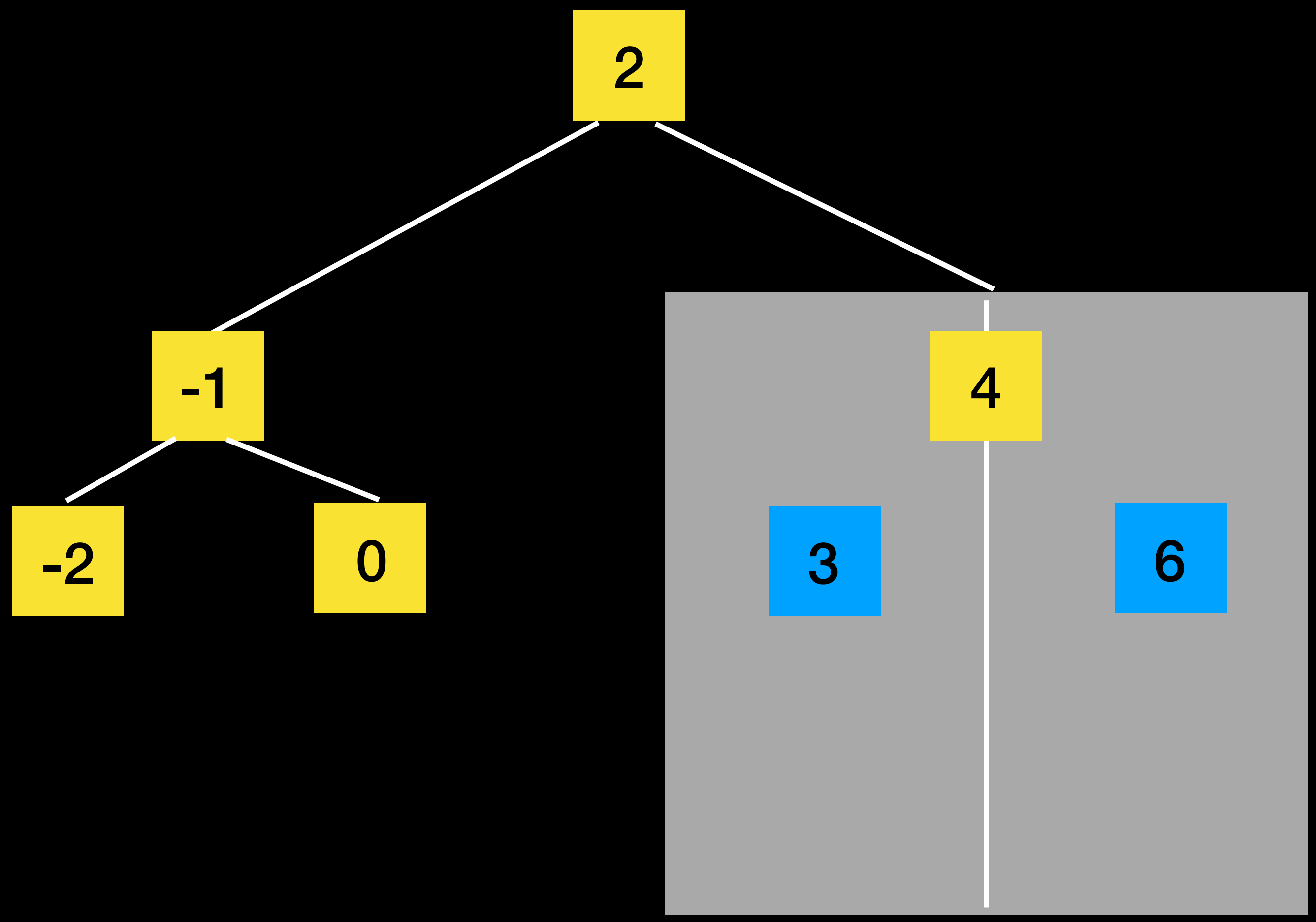
A Different Approach



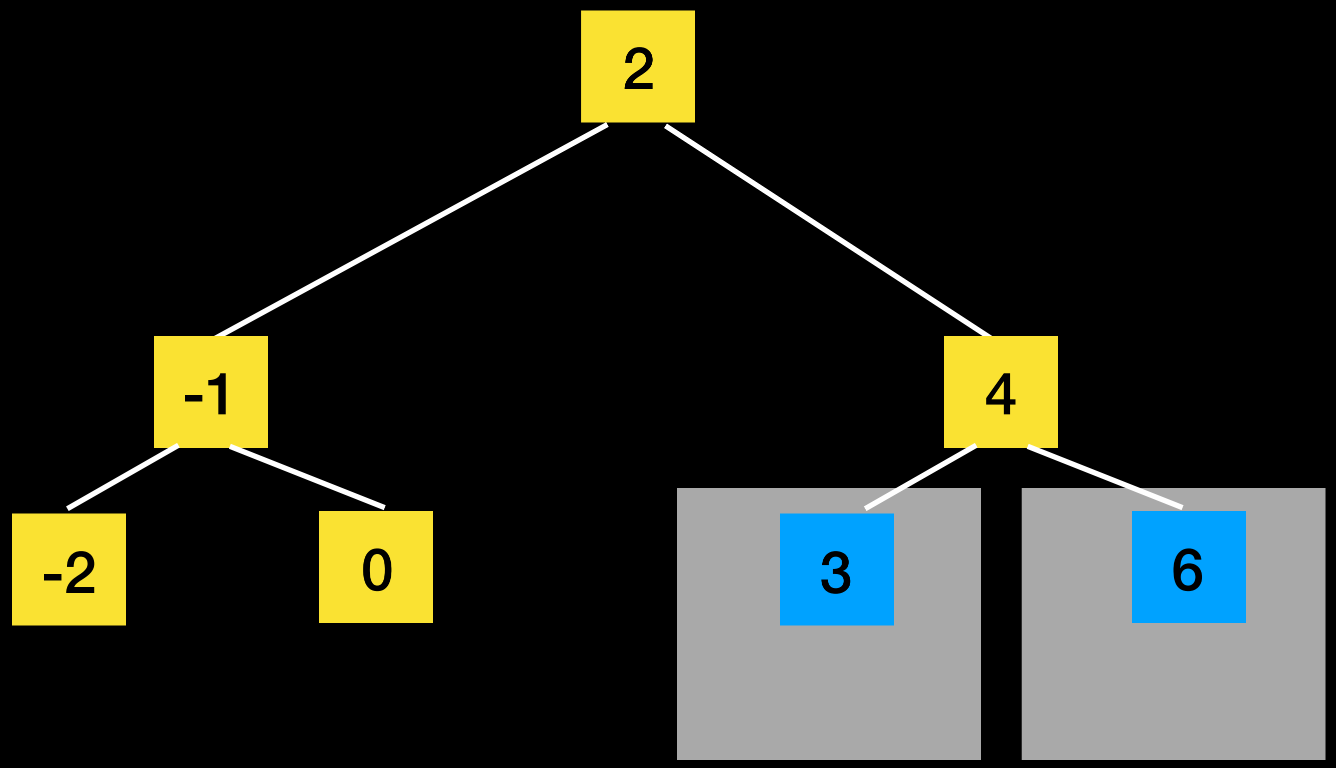
A Different Approach



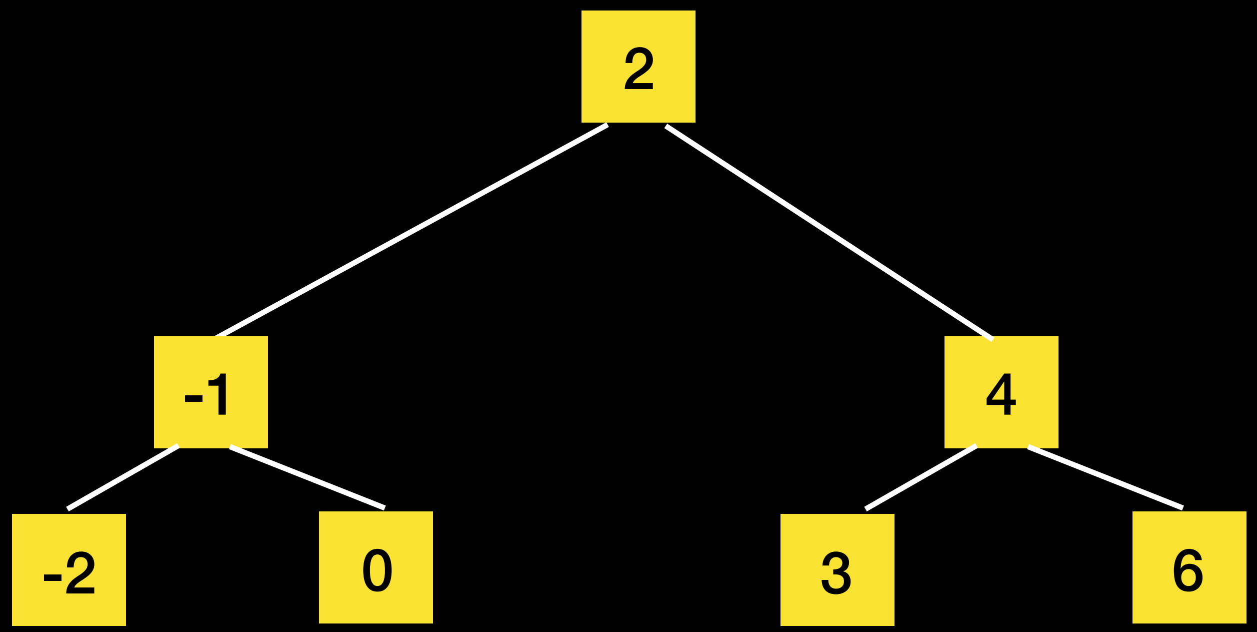
A Different Approach



A Different Approach

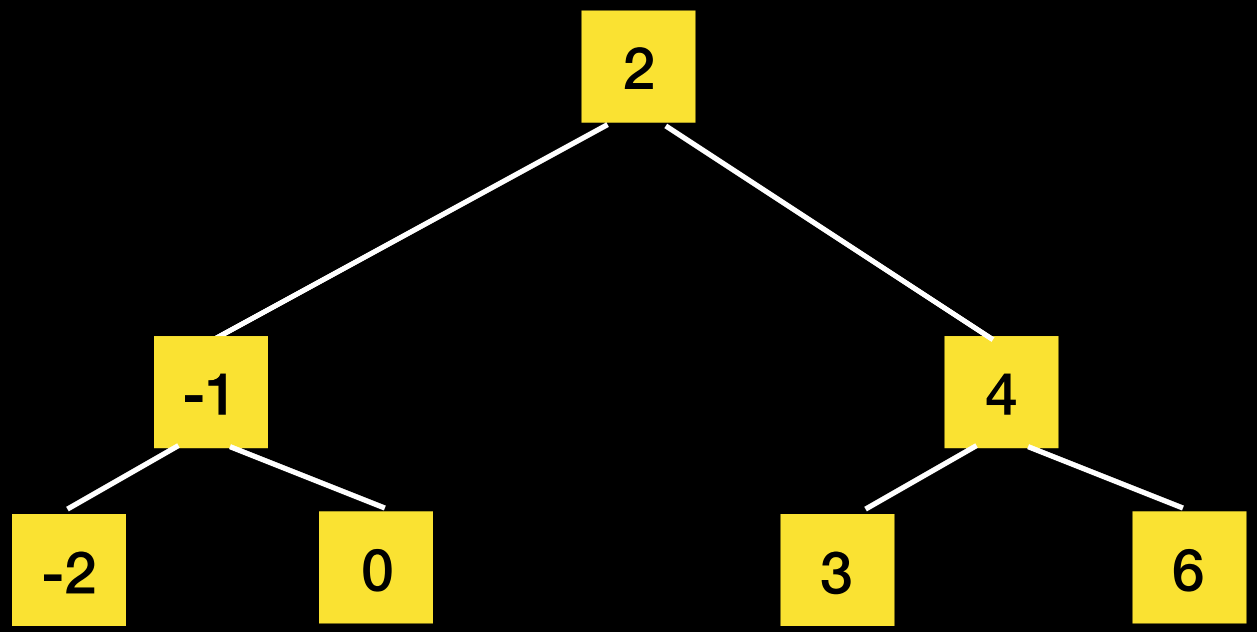


A Different Approach



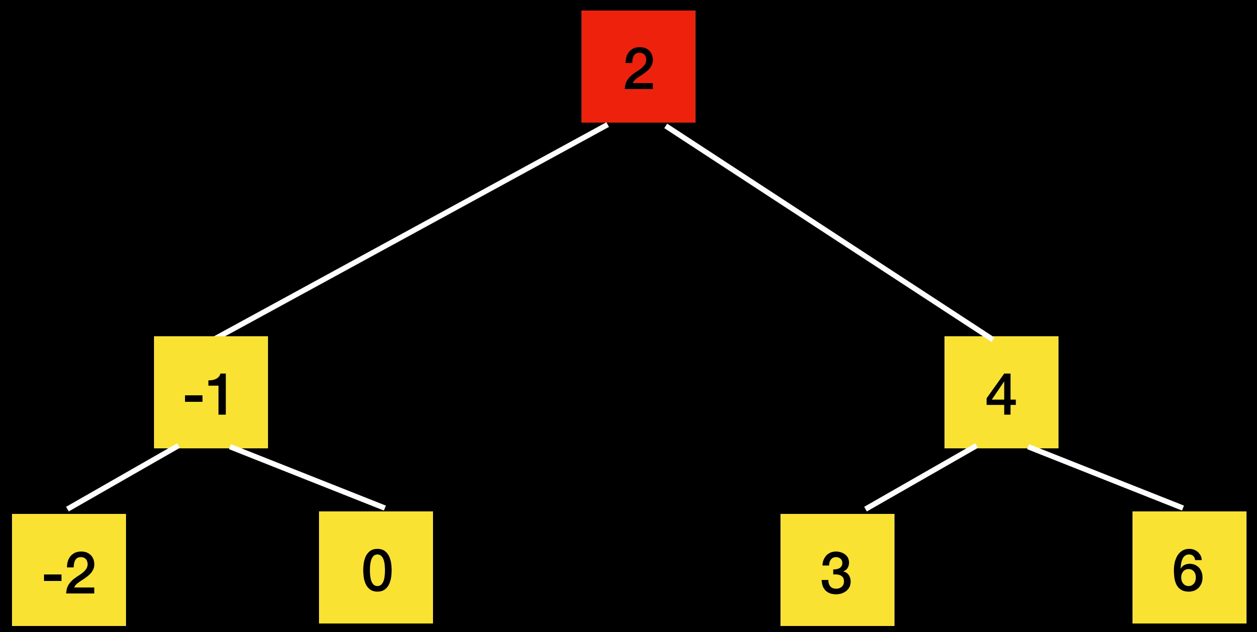
A Different Approach

Find 5



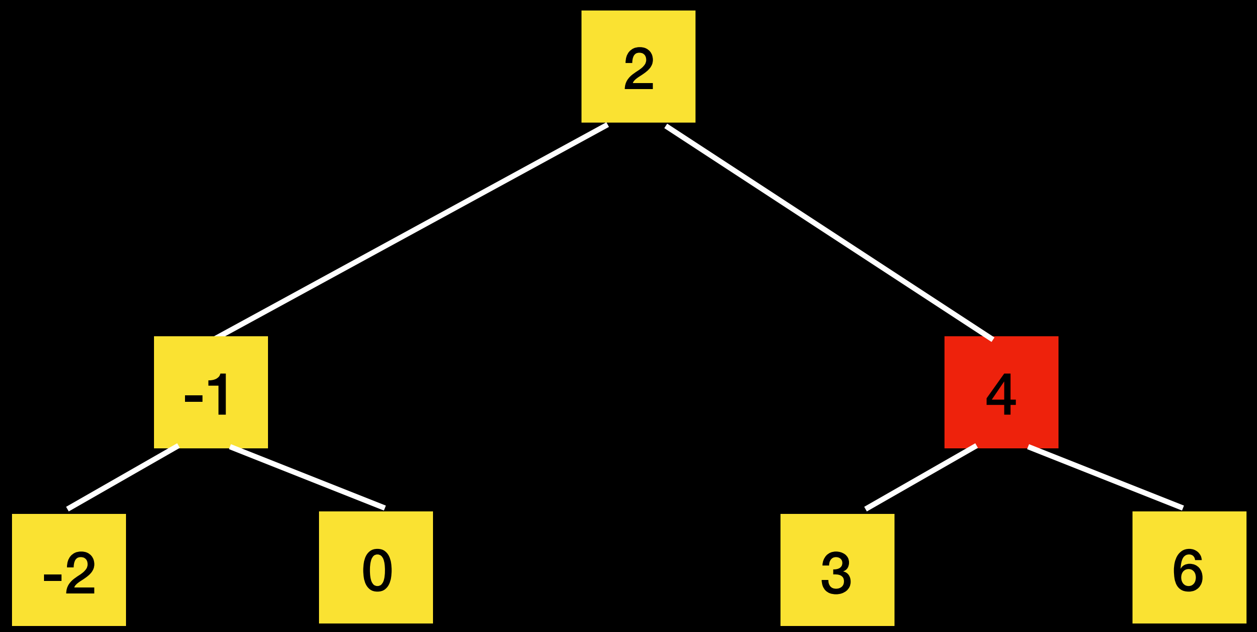
A Different Approach

Find 5



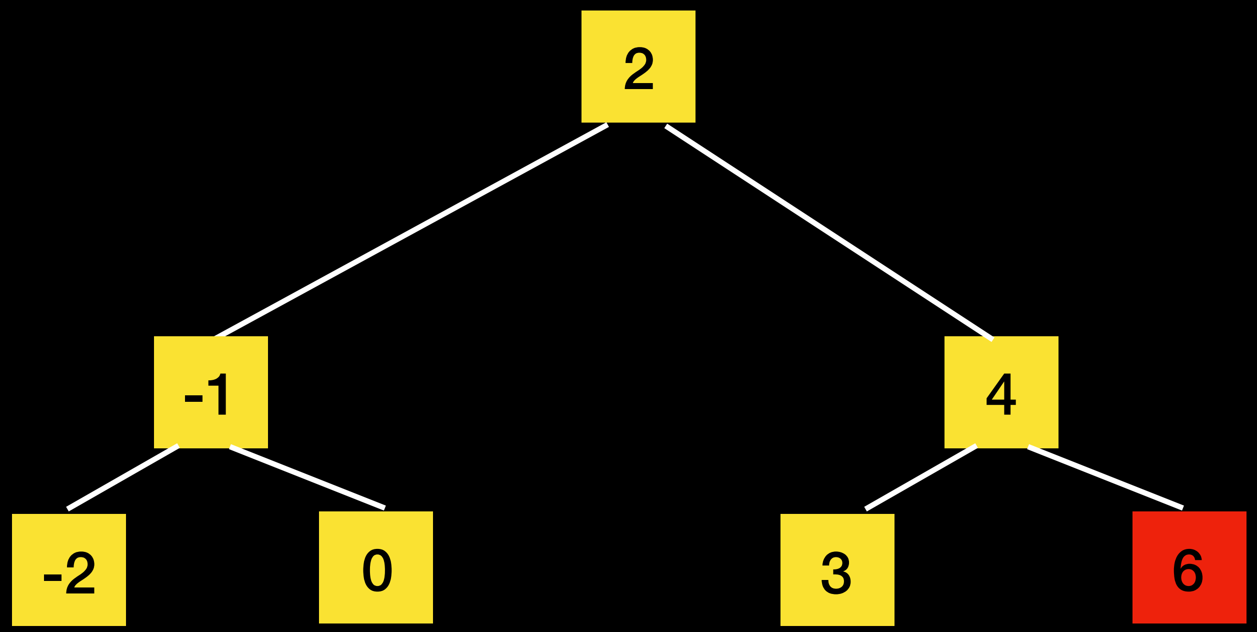
A Different Approach

Find 5



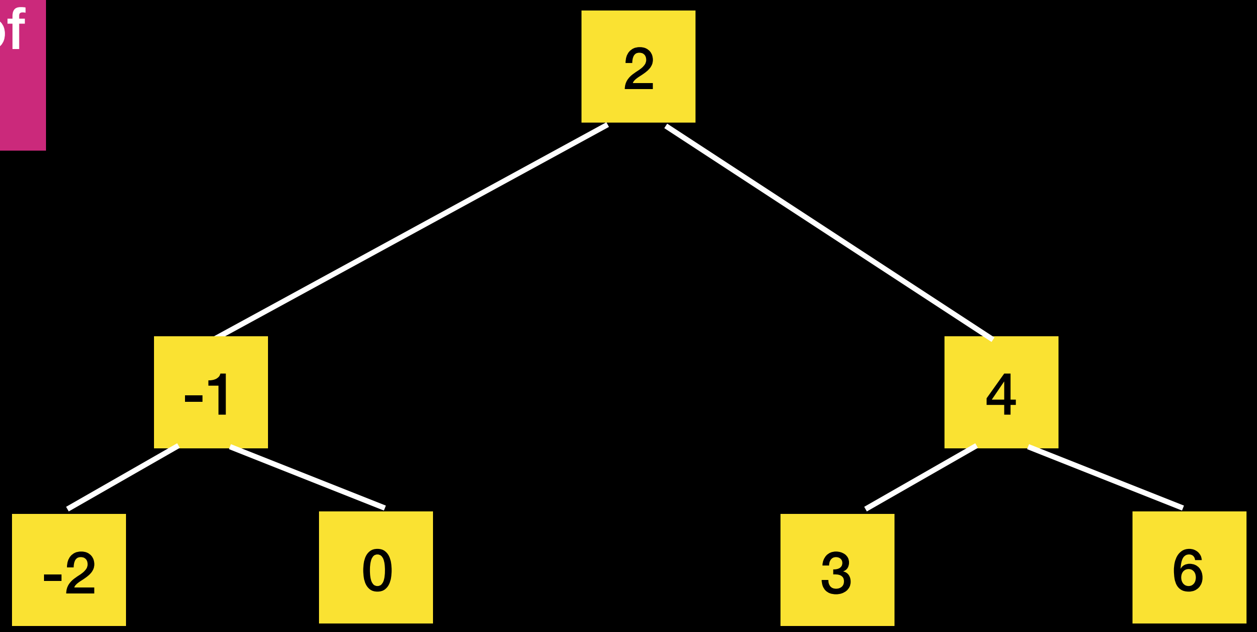
A Different Approach

Find 5



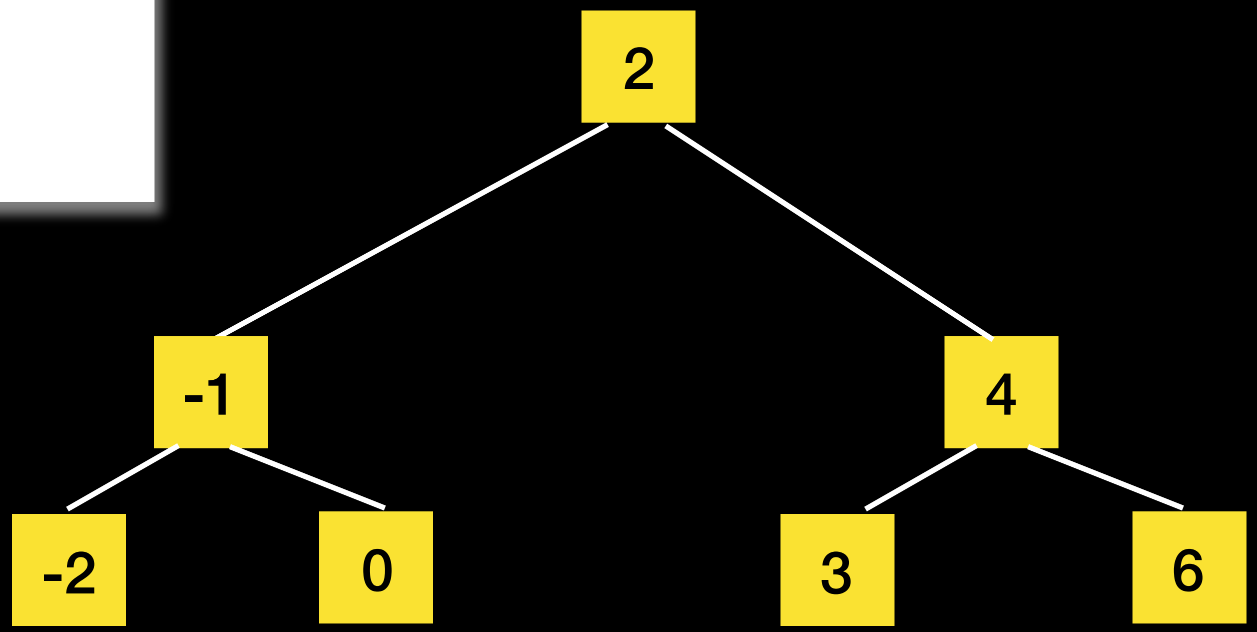
A Different Approach

What's special about the shape of this tree?



Binary Search Tree

Structural Property:
For each node n
n > all values in T_L
n < all values in T_R



BST Formally

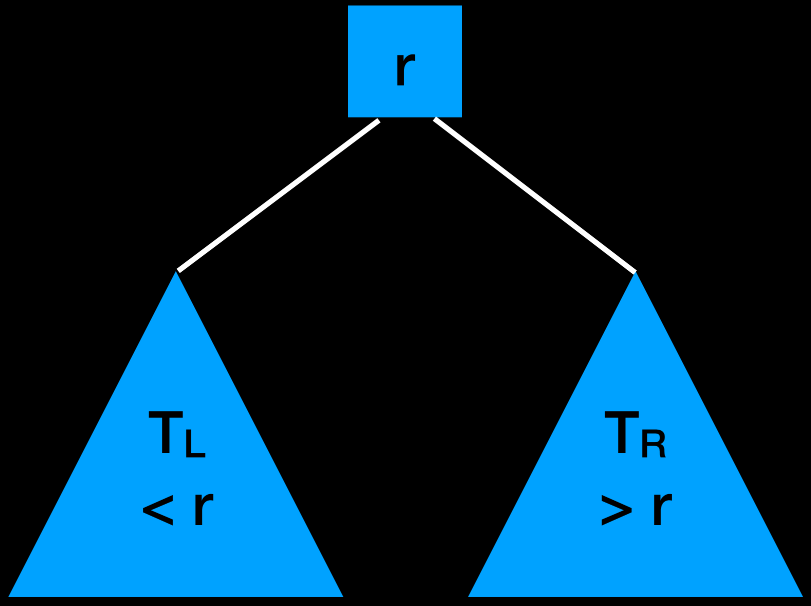
Let S be a set of values upon which a total ordering relation $<$, is defined. For example, S can be a set of numbers.
A **binary search tree** (**BST**) T for the ordered set $(S,<)$ is a binary tree with the following properties:

- 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 $s = n$.

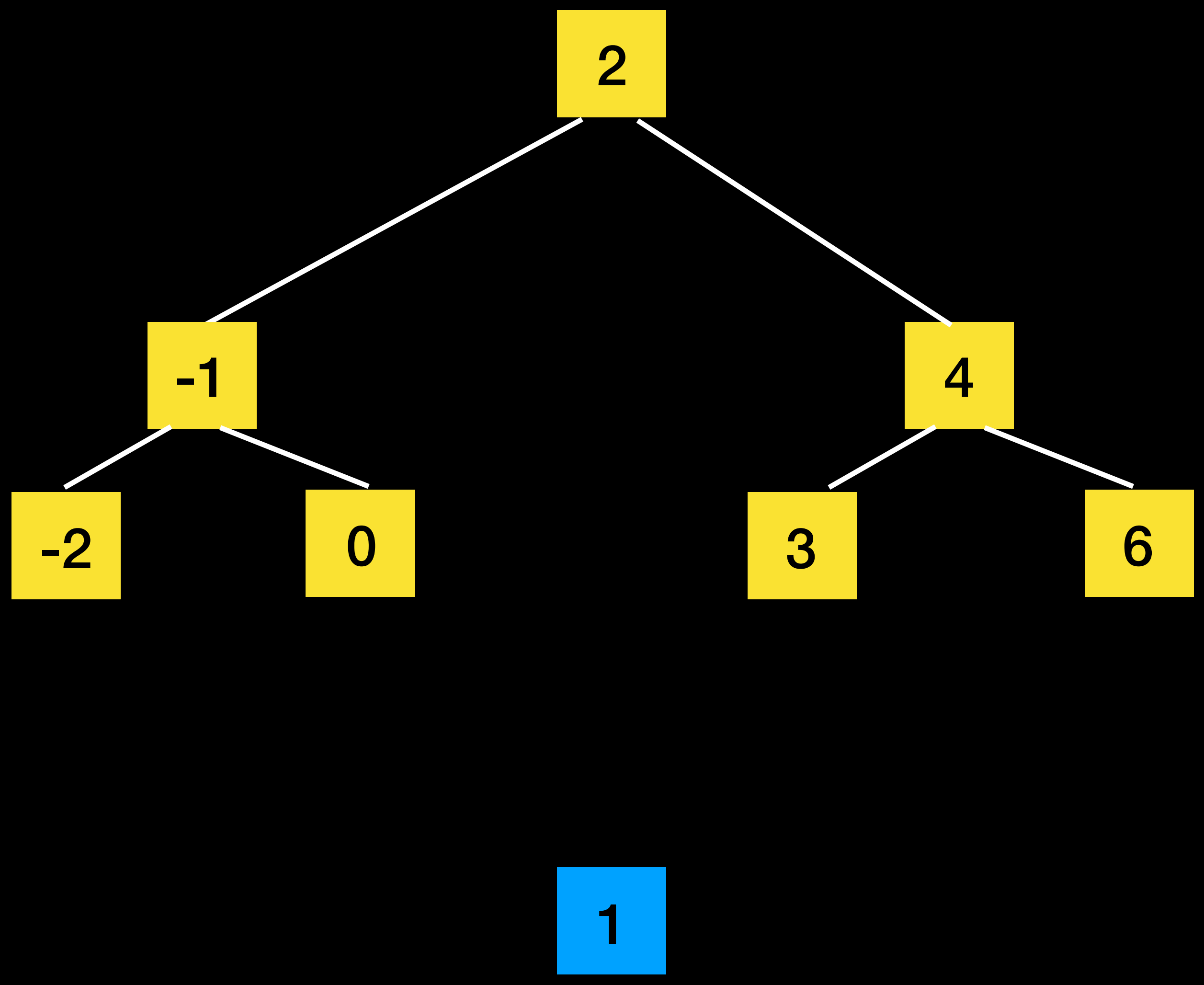
Binary Search Tree

Structural Property:
For each node n
n > all values in T_L
n < all values in T_R

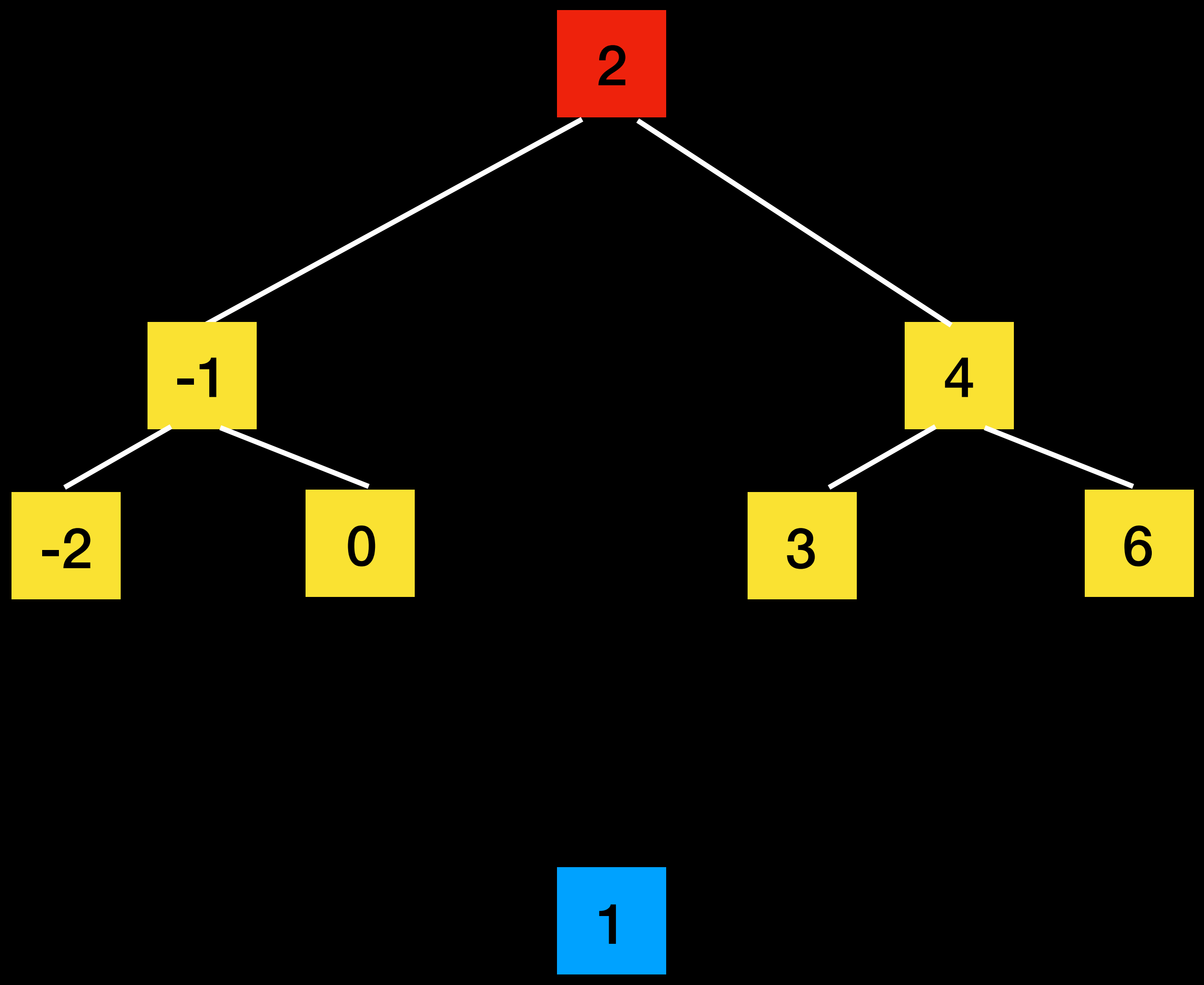
```
search(bs_tree, item)
{
    if (bs_tree is empty) //base case
        item not found
    else if (item == root)
        return root
    else if (item < root)
        search(TL , item)
    else // item >= root
        search(TR , item)
}
```



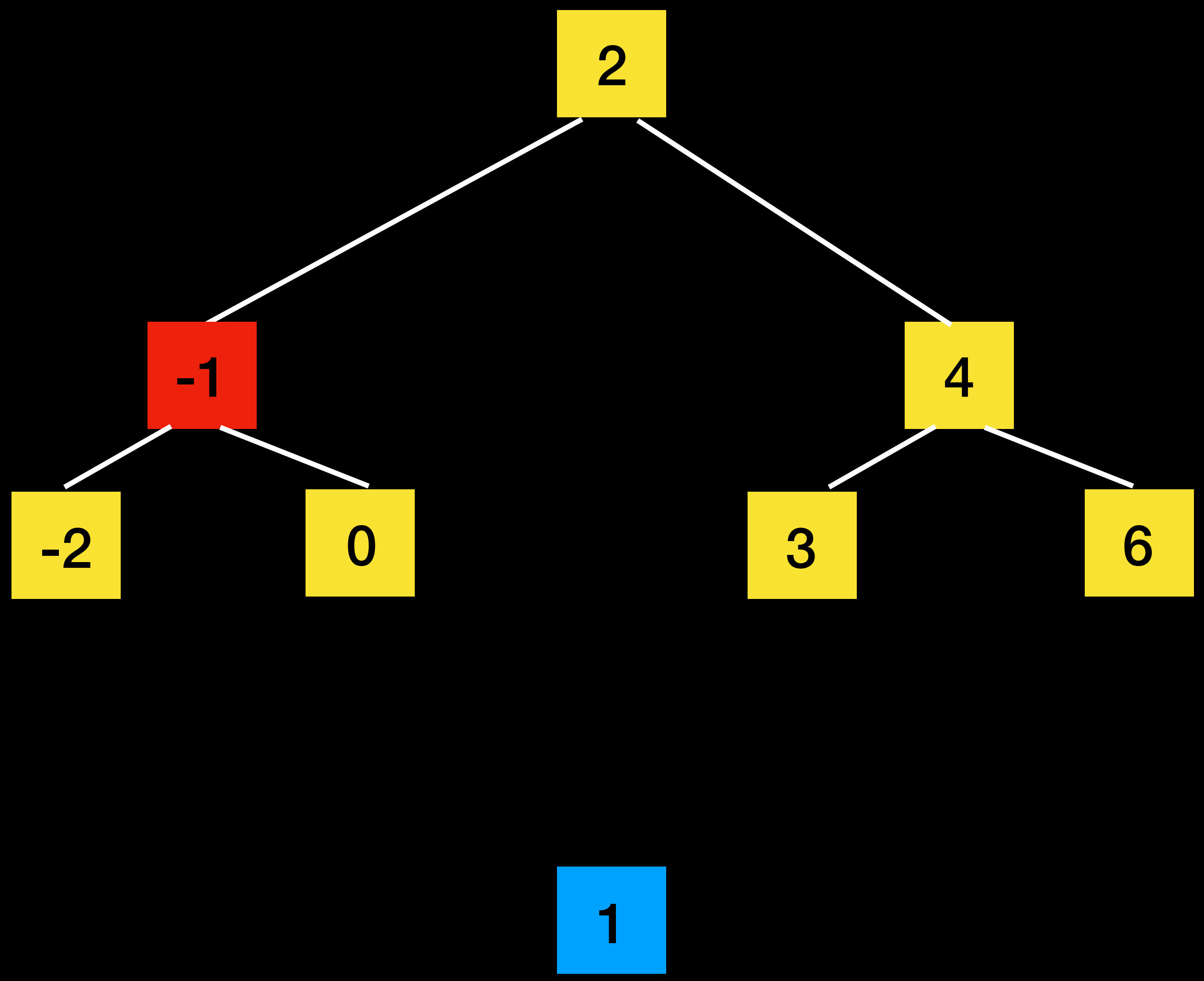
Inserting into a BST



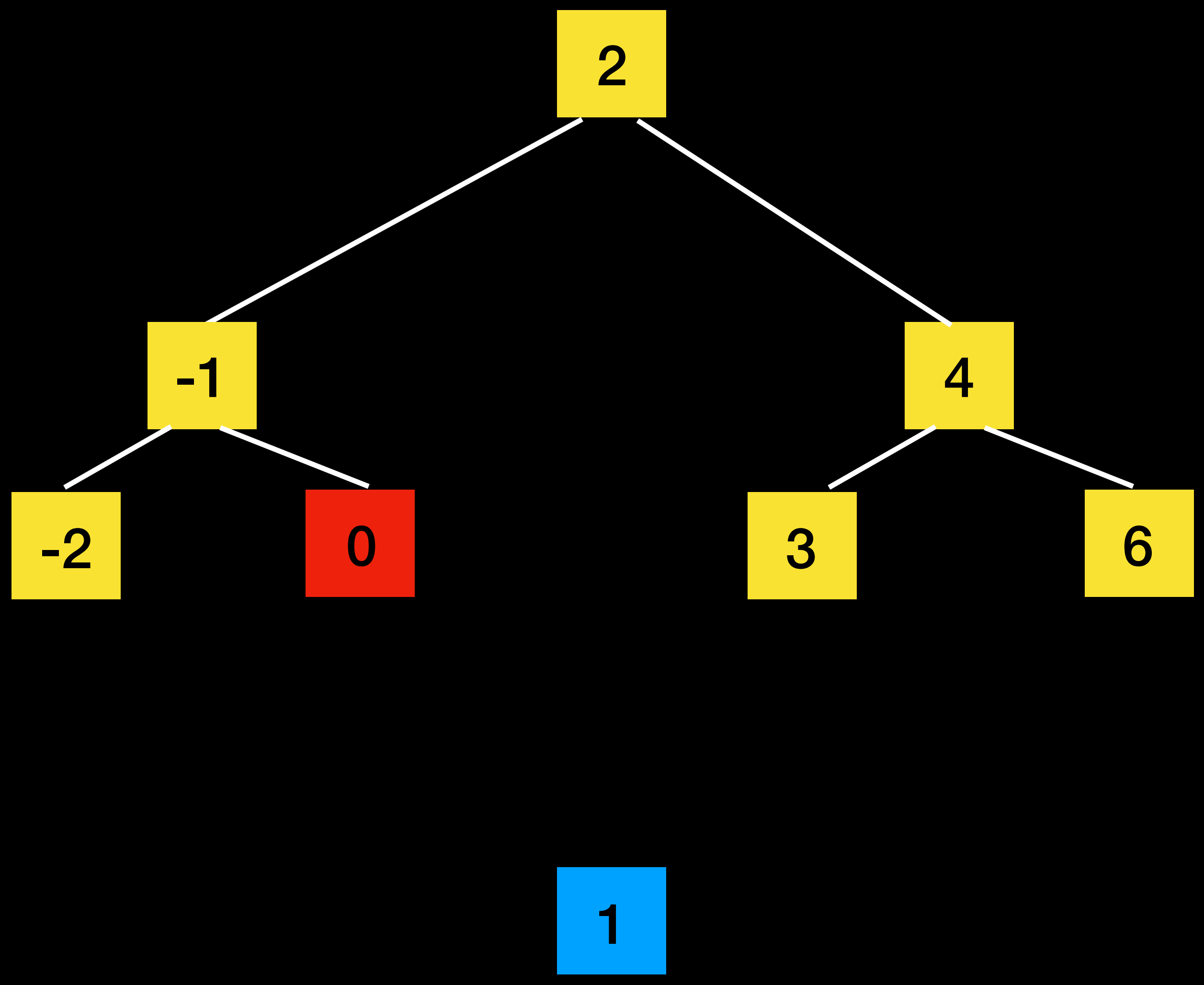
Inserting into a BST



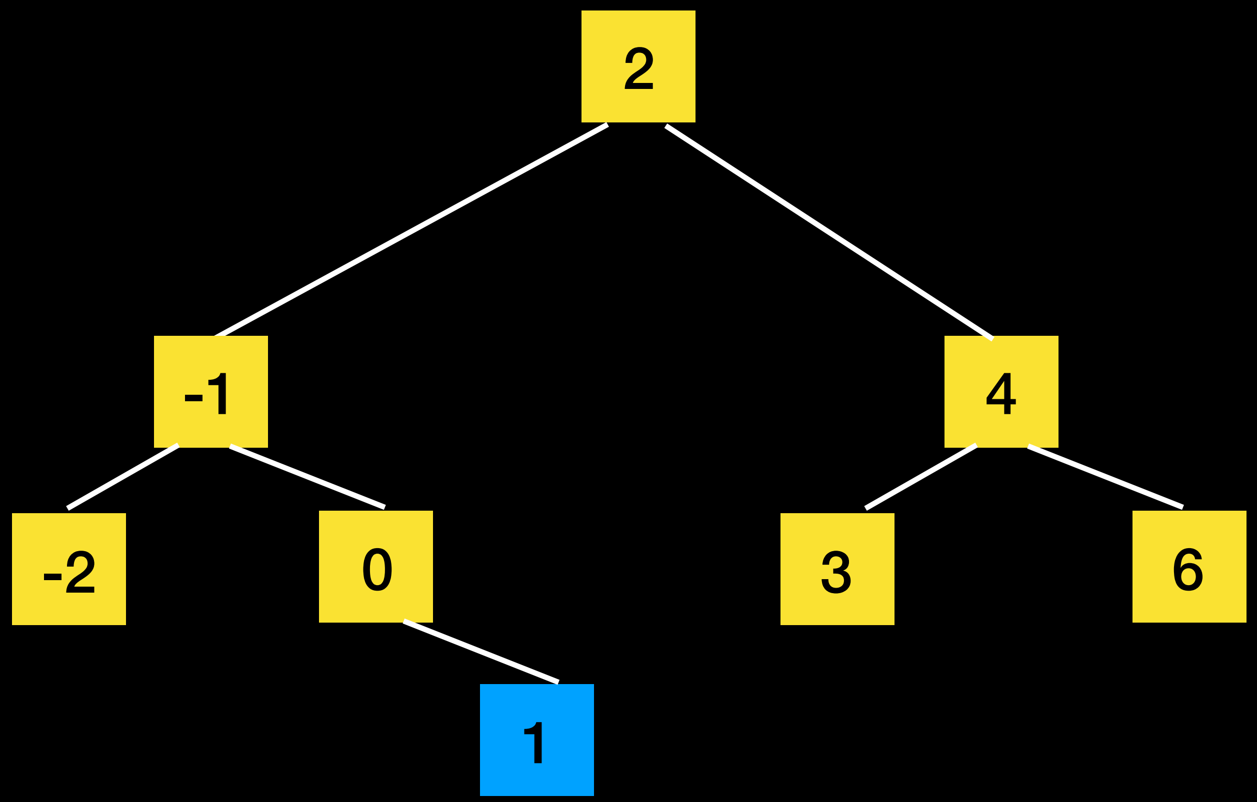
Inserting into a BST



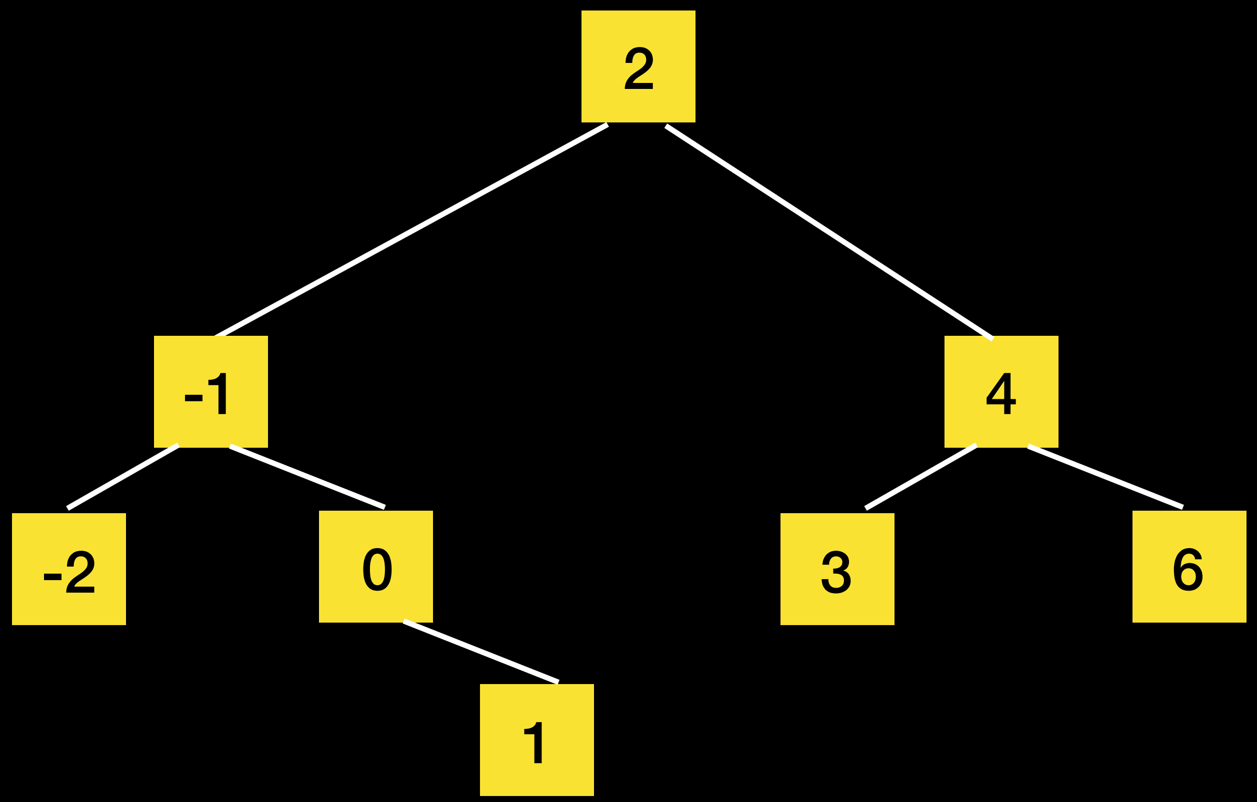
Inserting into a BST



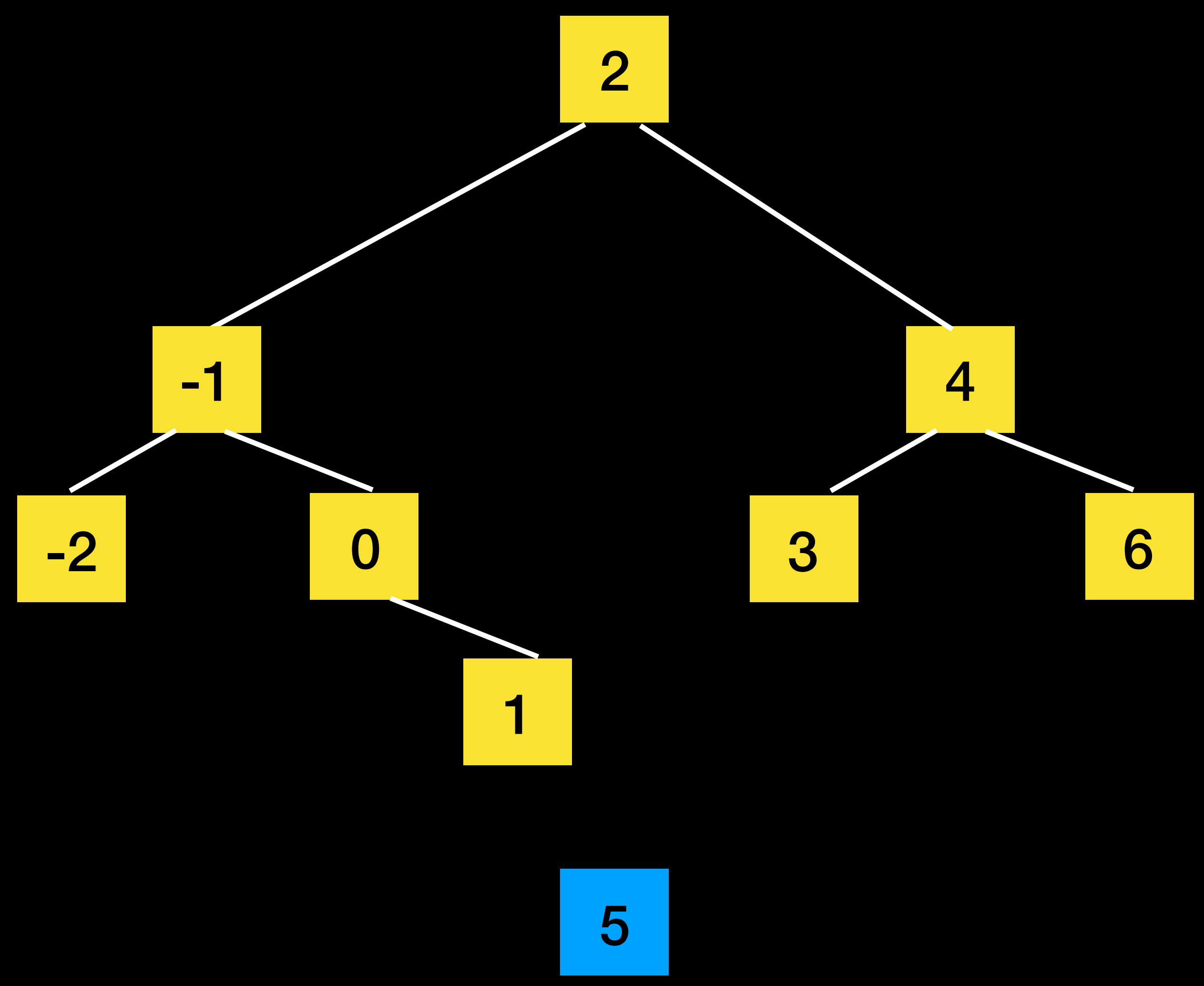
Inserting into a BST



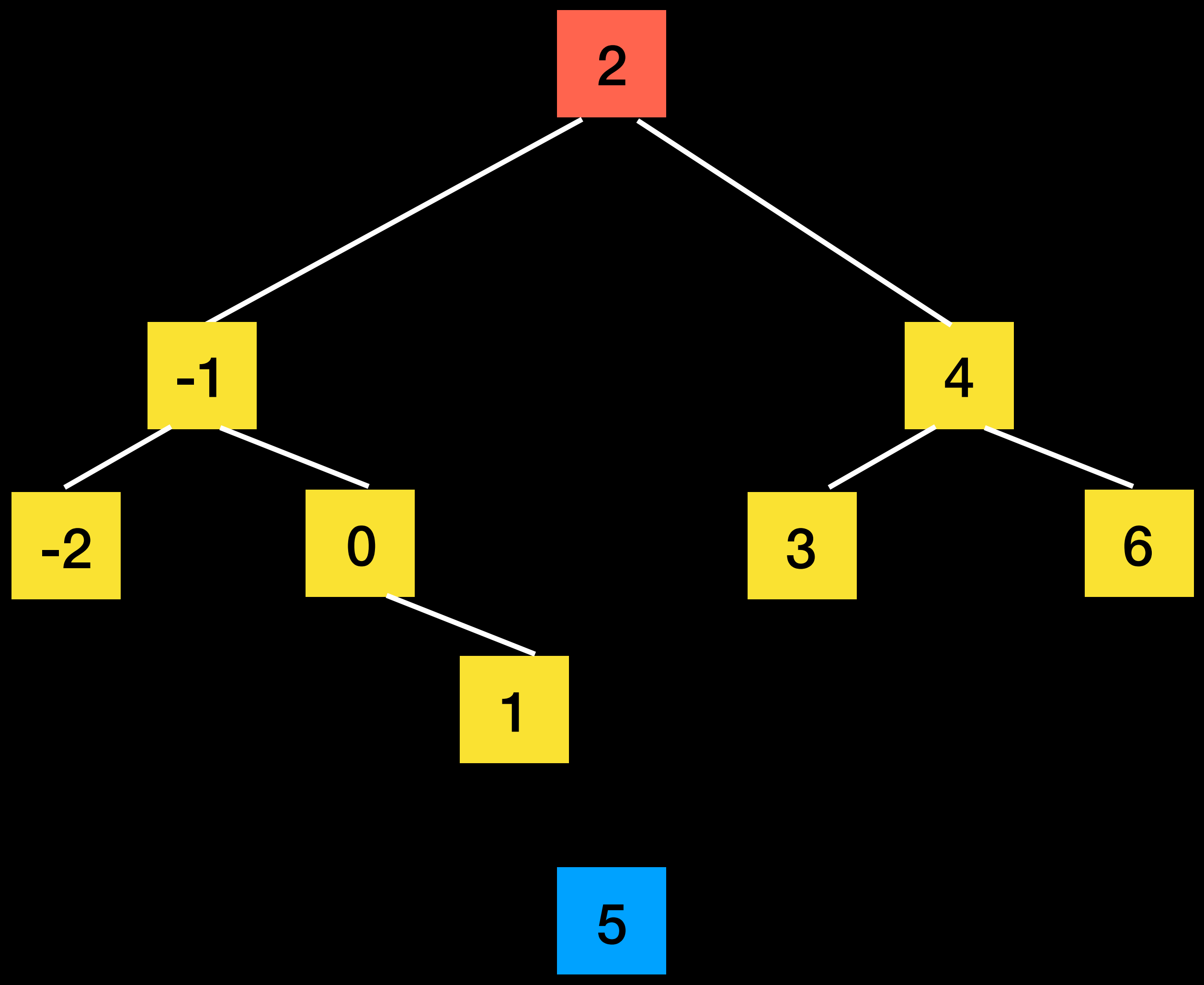
Inserting into a BST



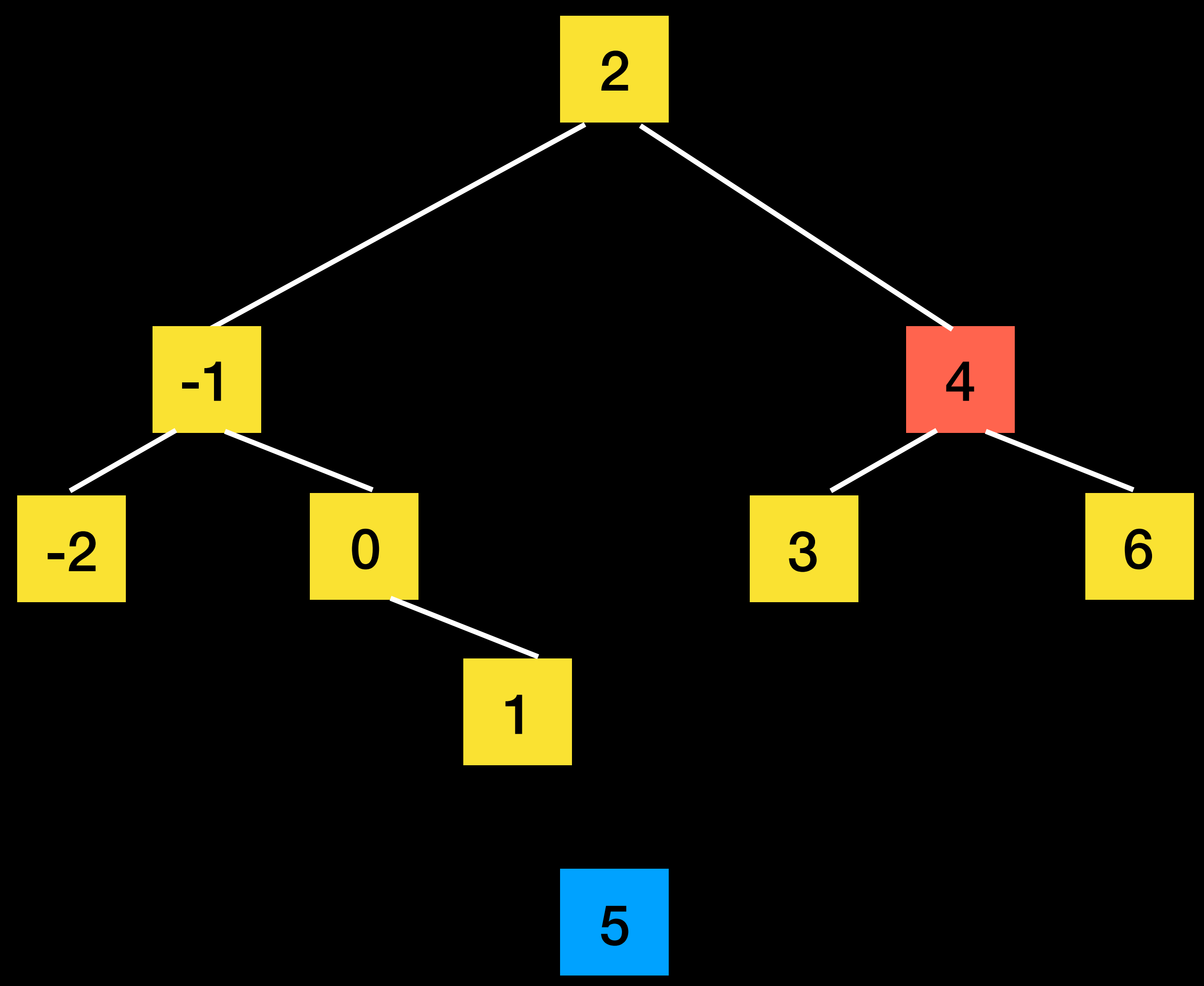
Inserting into a BST



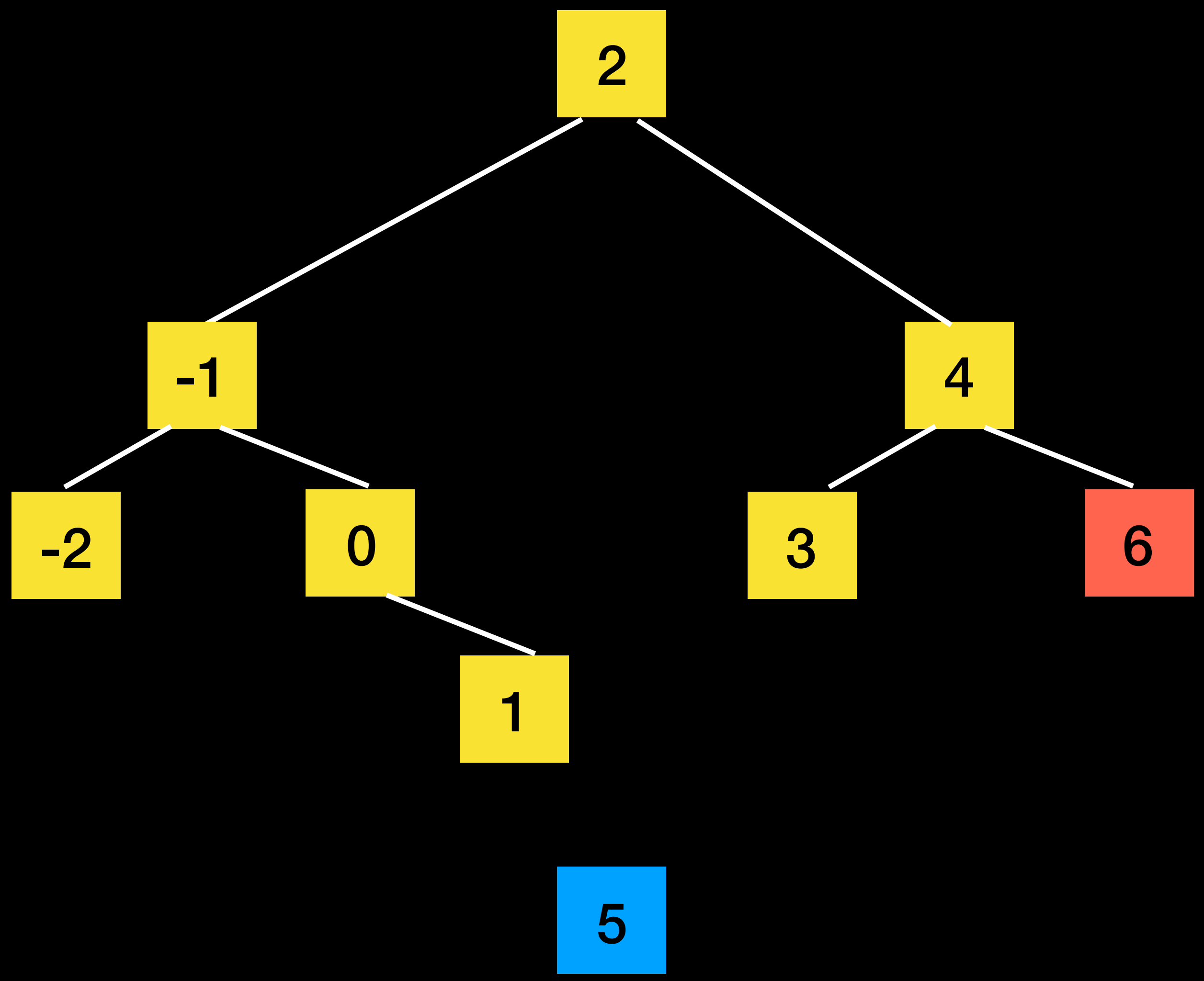
Inserting into a BST



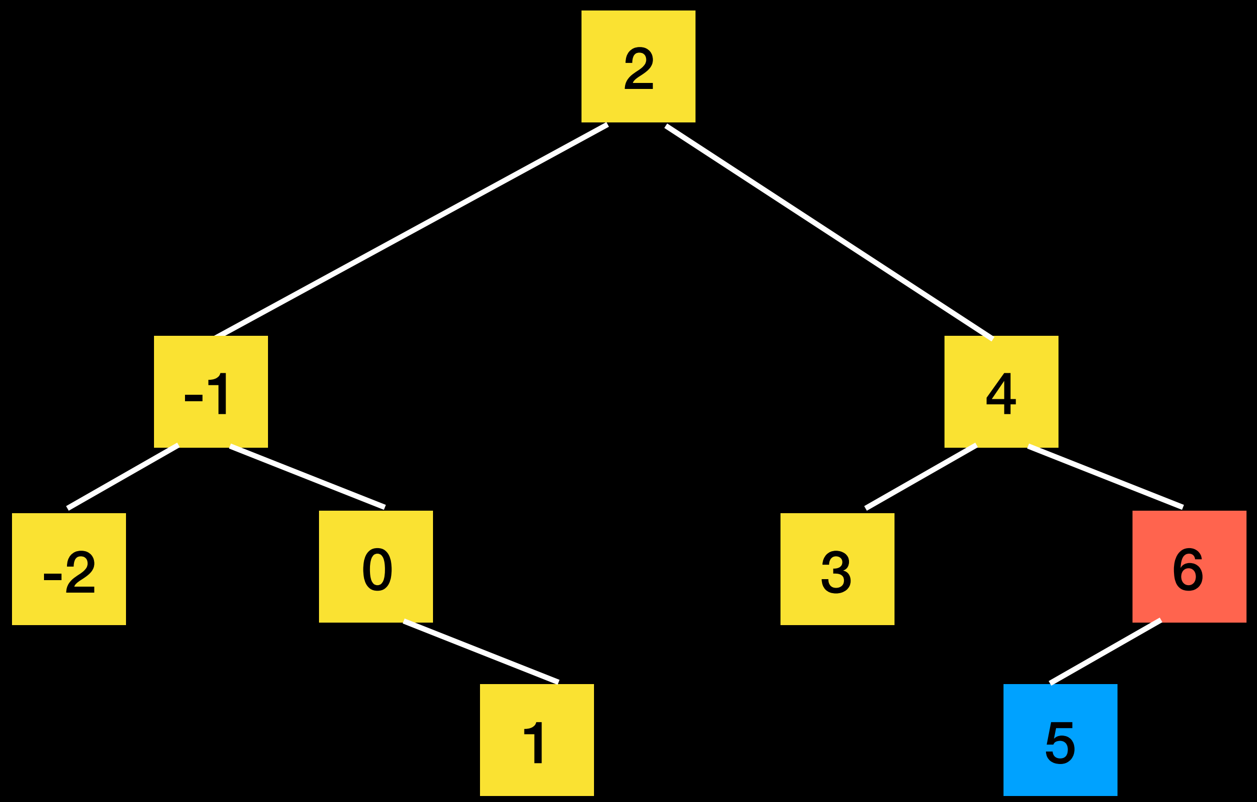
Inserting into a BST



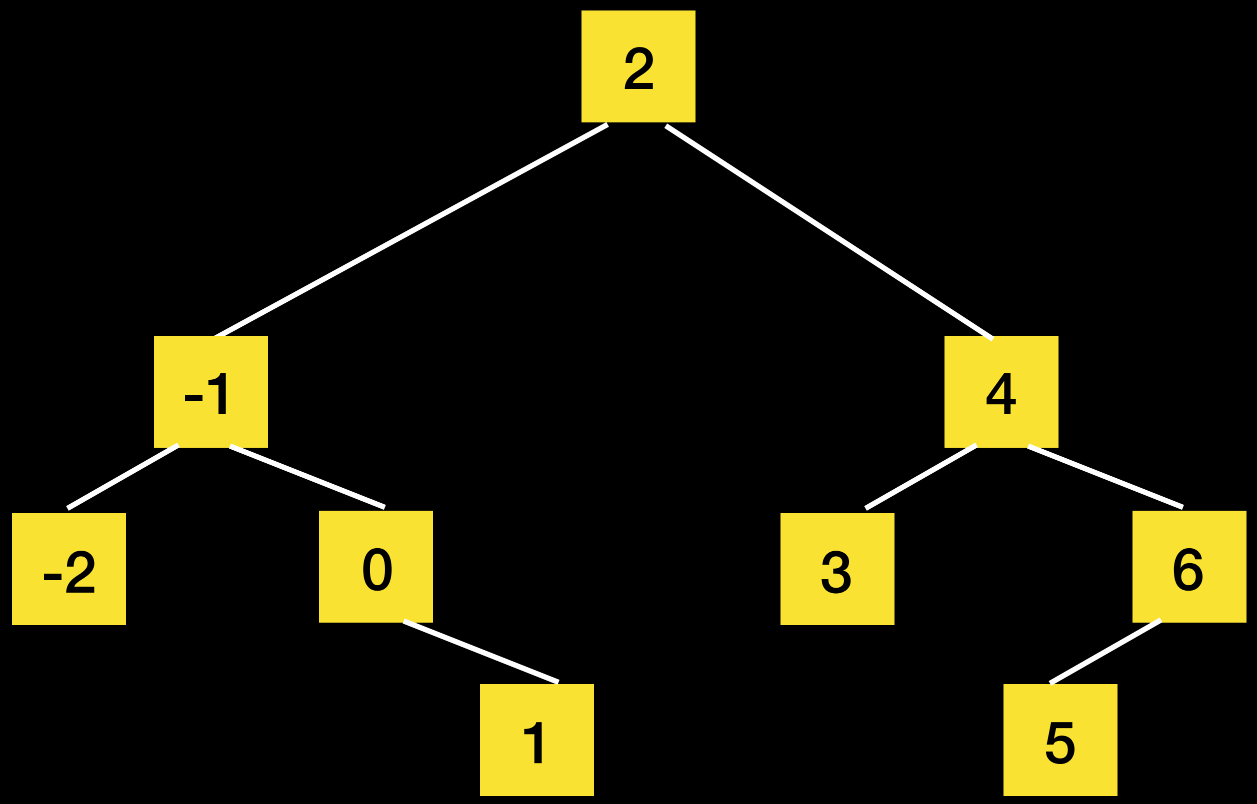
Inserting into a BST



Inserting into a BST

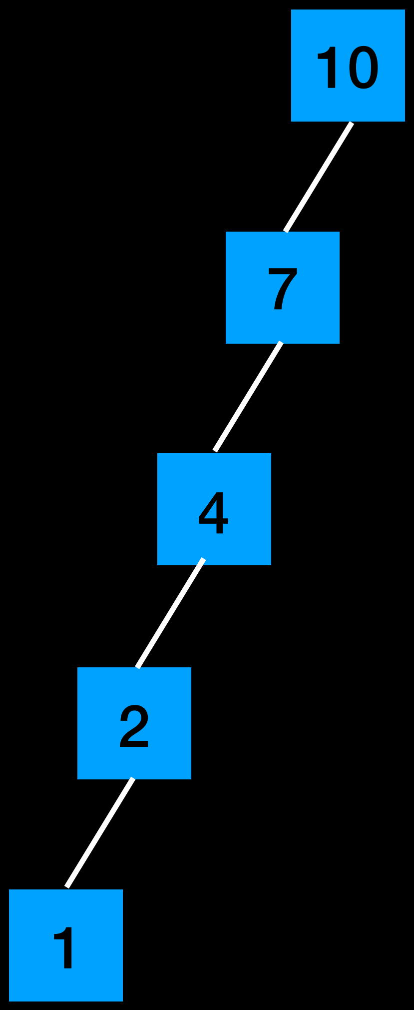


Inserting into a BST

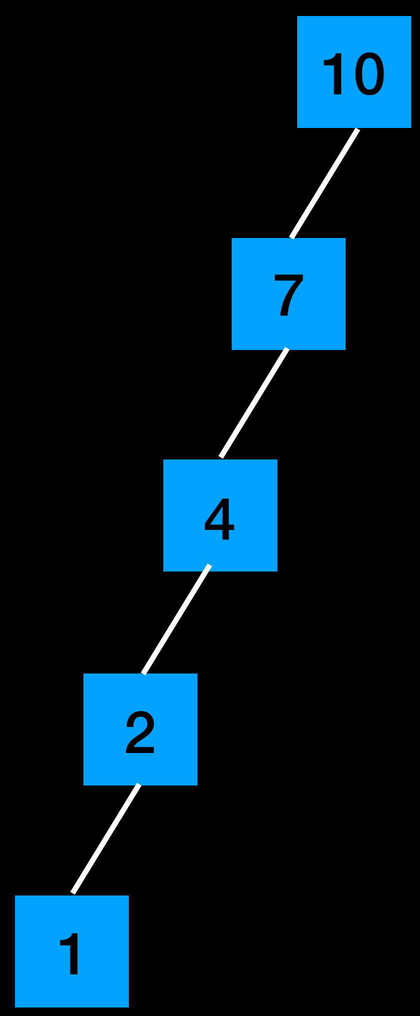


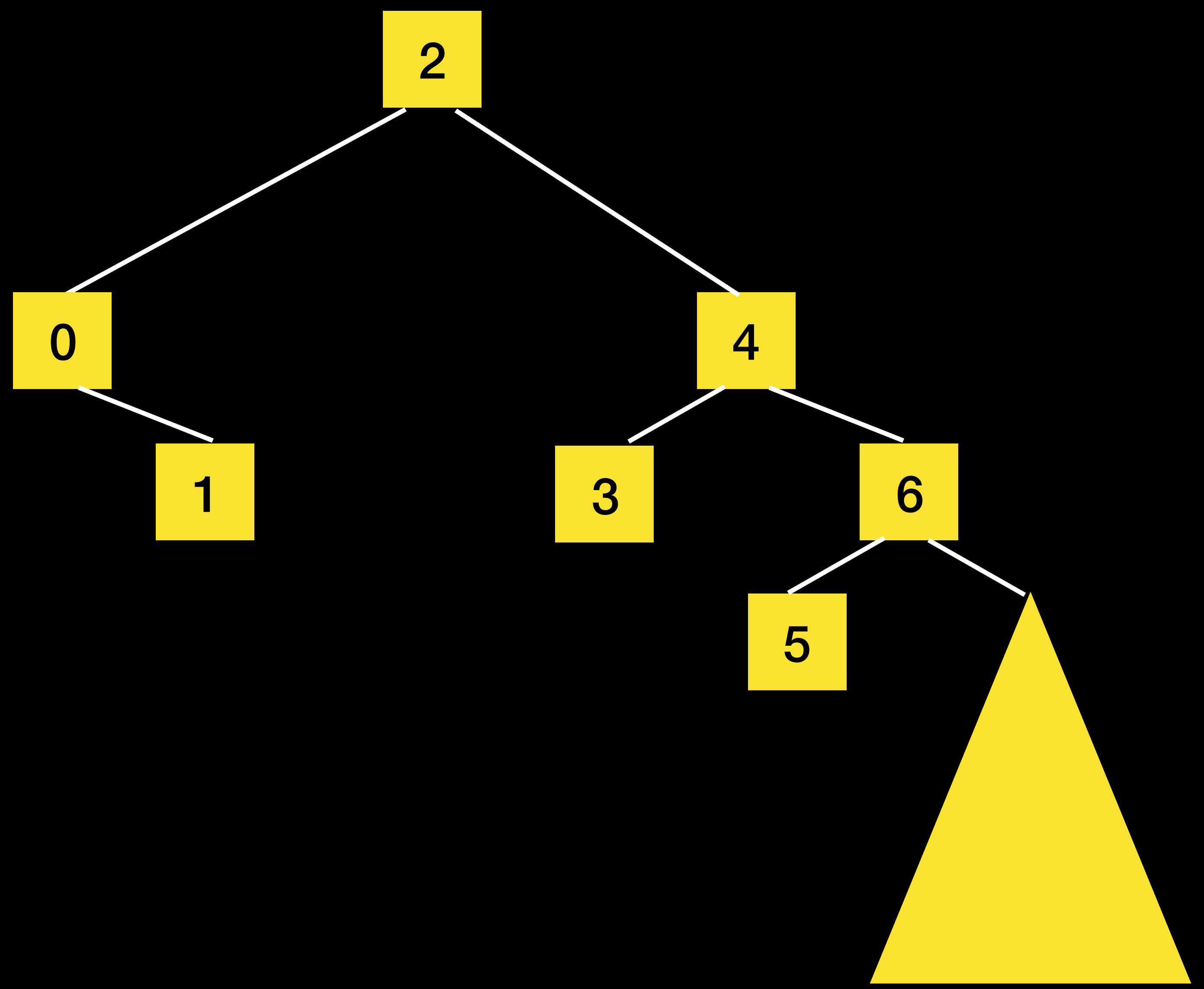
You **Grow** a tree with BST property, you don’t get to restructure it
(Self-balancing BST and AVL trees will do that, perhaps in CSCI 335)

Growing a BST



Growing a BST

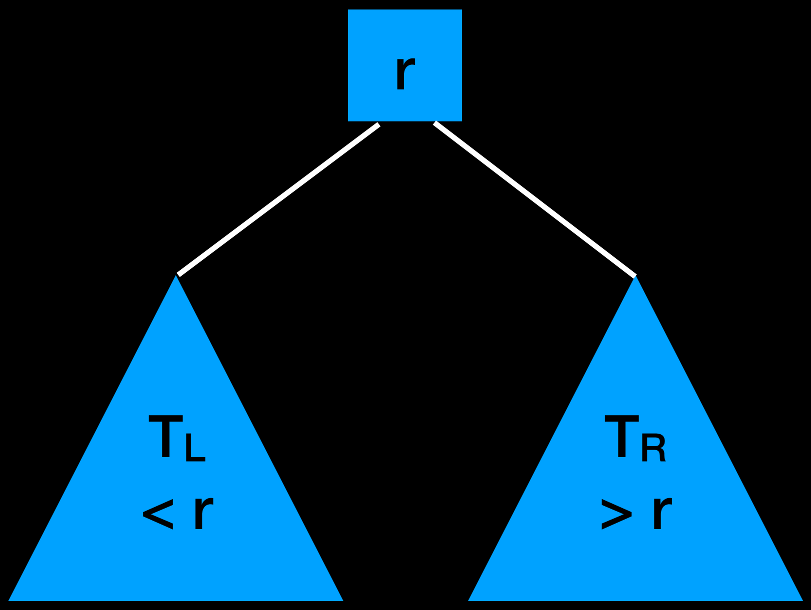




In-Class Task

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

Inserting into a BST

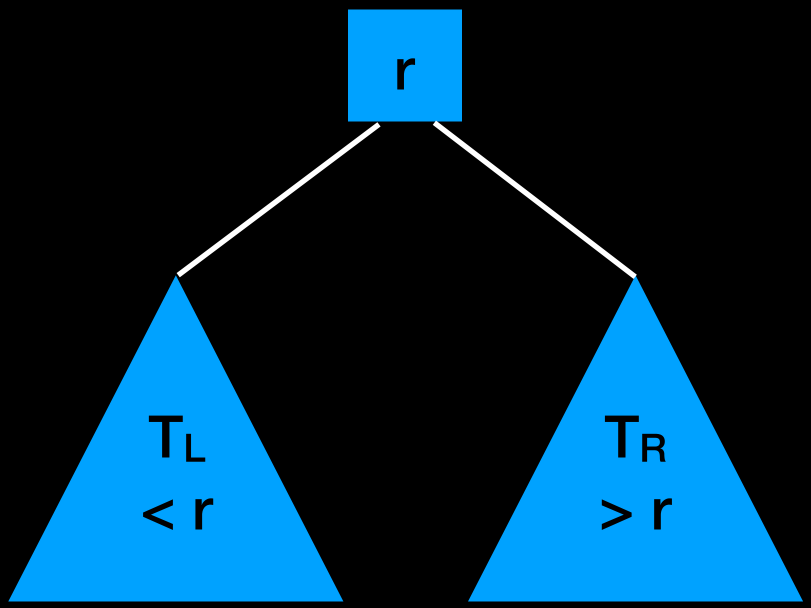



```
add(bs_tree, item)
{
    if (bs_tree is empty) //base case
        make item the root
    else if (item < root)
        add(T_L , item)
    else // item >= root
        add(T_R , item)
}
```

Traversing a BST

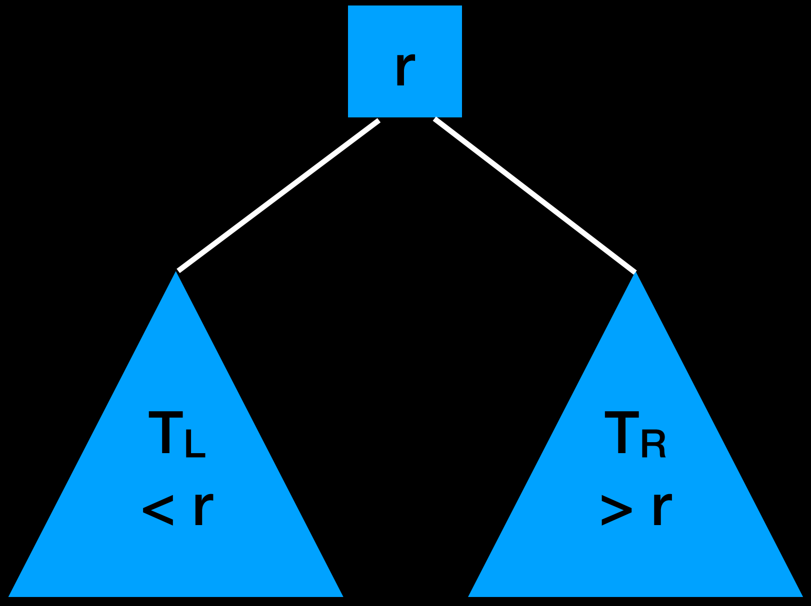
Same as traversing any binary tree

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

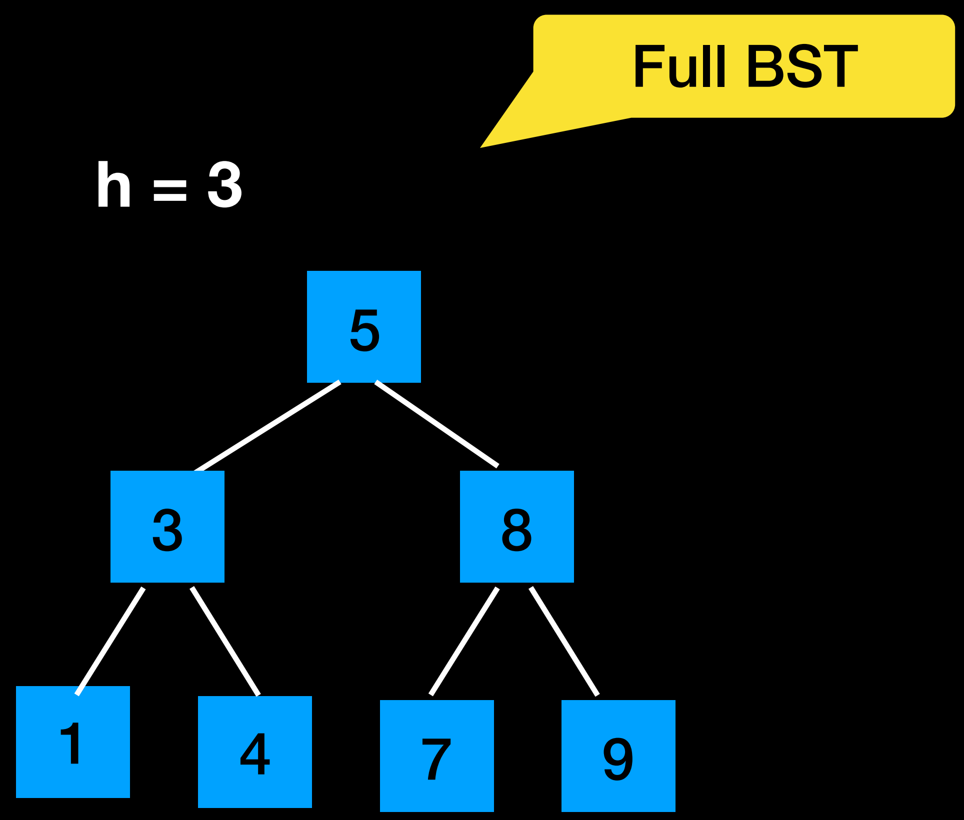
Think about the structure and height of the tree

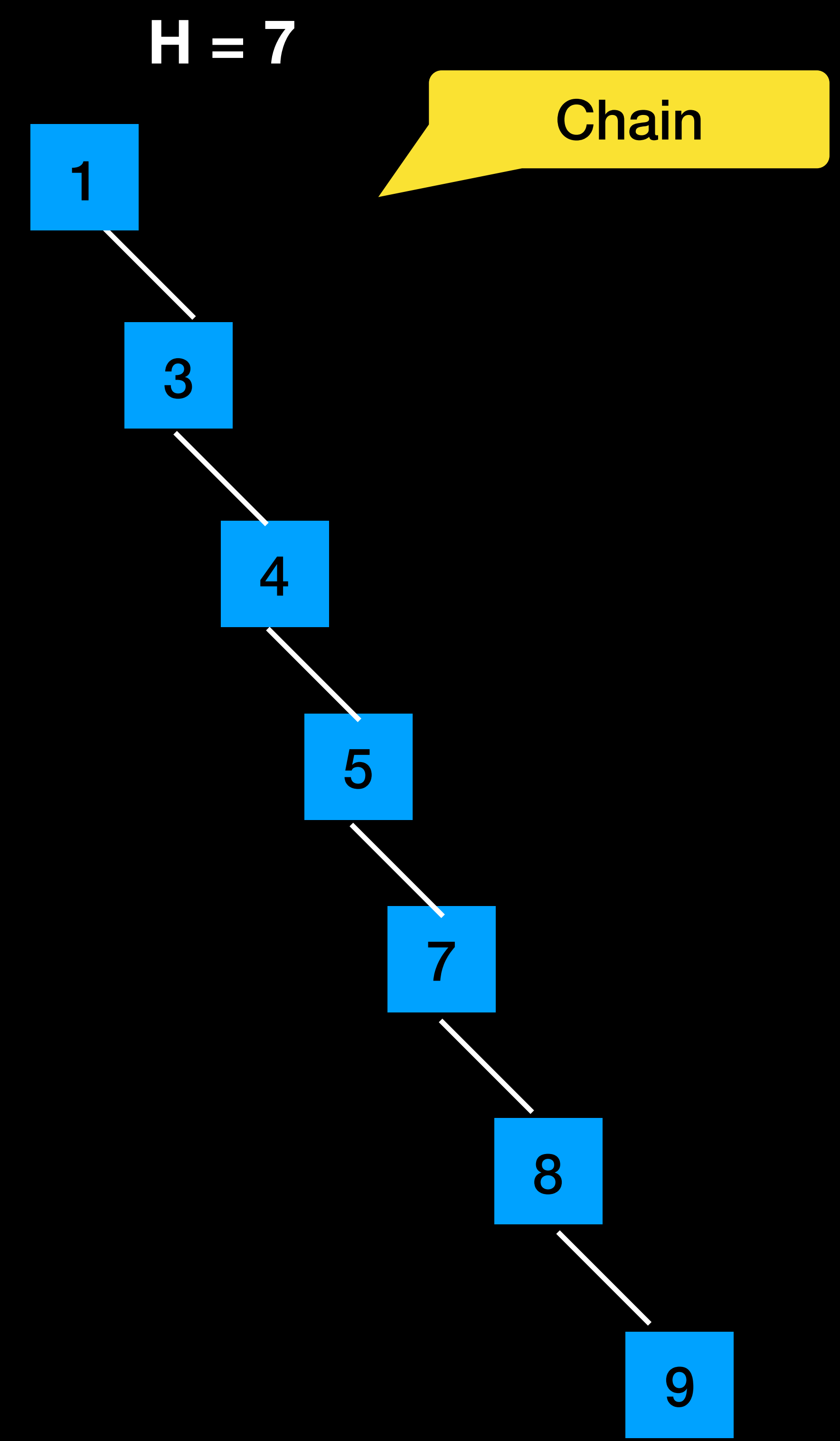
O(h)

What is the maximum height?

What is the minimum height?

Tree Structure





n nodes

$\log_2 (n+1) <= h <= n$

Operation	$\Theta(n)$	$O(n)$
Search	$\log_2 n$	n
Add	$\log_2 n$	n
Remove	$\log_2 n$	n
Traverse	n	n

BST Operations

```cpp
#ifndef BST_H_
#define BST_H_

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