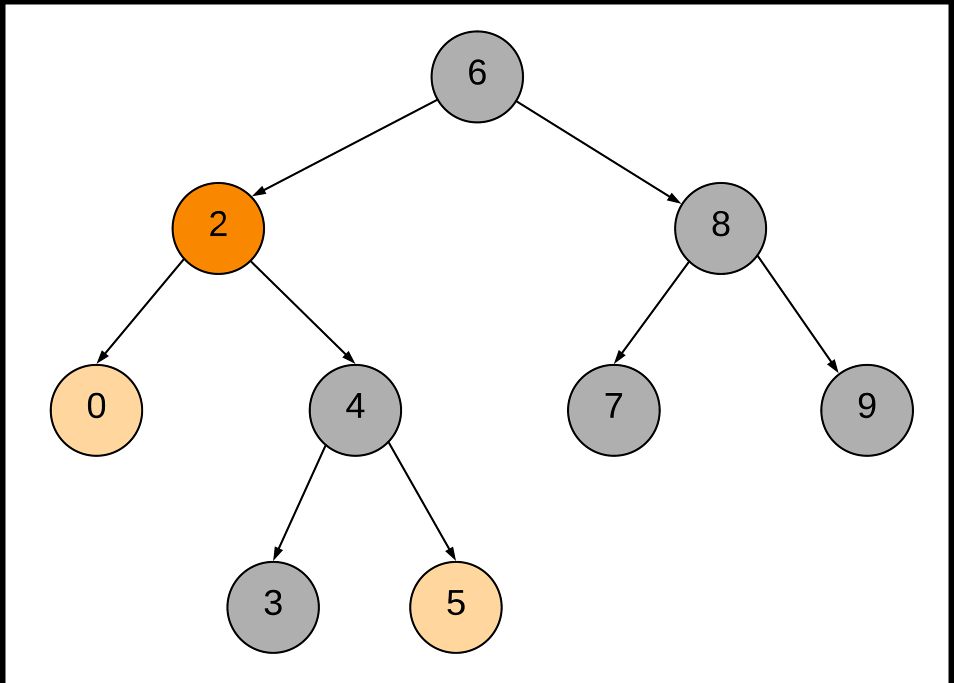
Binary Search Tree (BST)



Tiziana Ligorio
Hunter College of The City University of New York

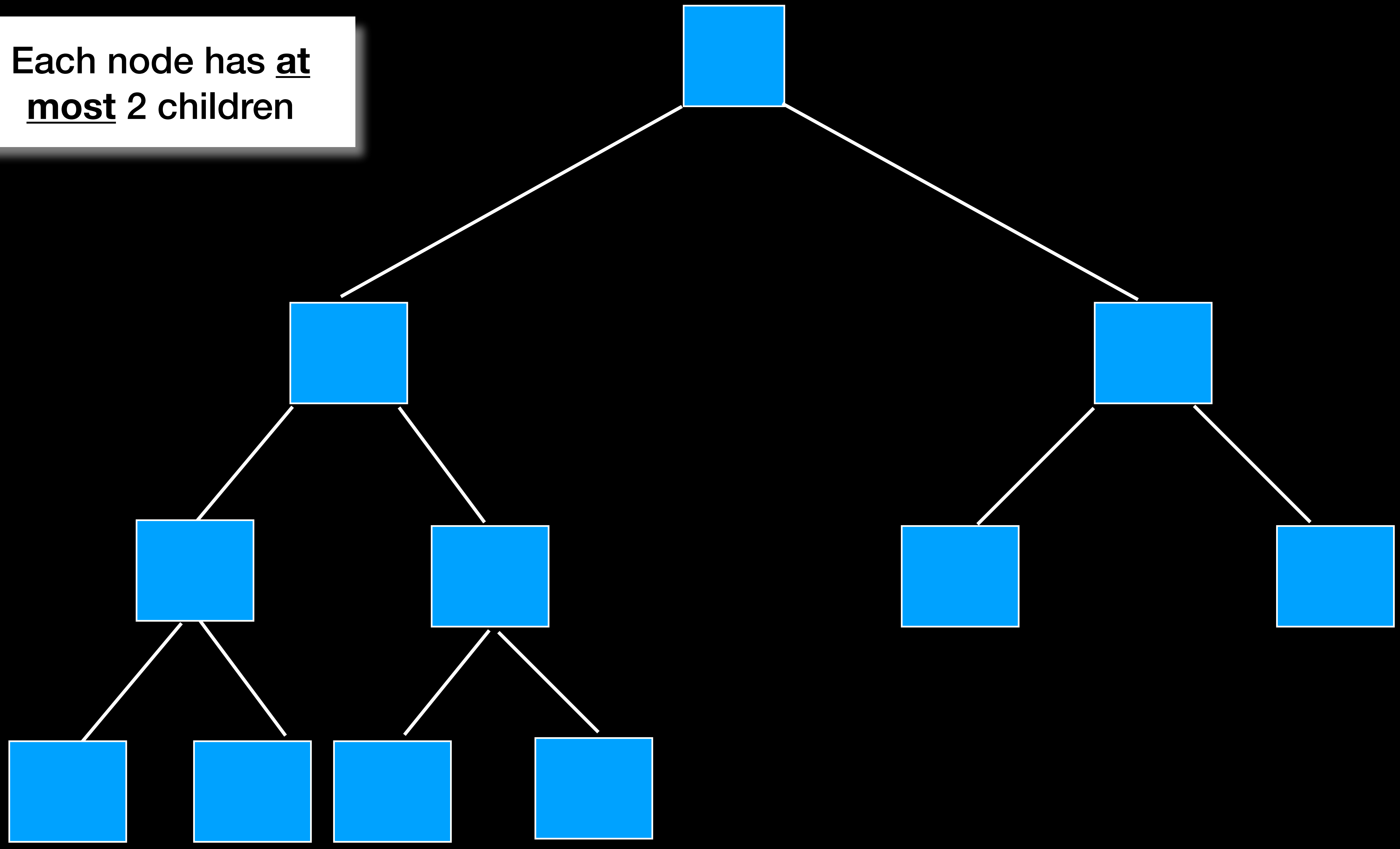
Today's Plan

Recap

Binary Search Tree ADT

Recap: Binary Tree

Each node has **at most** 2 children



Recap: Structure

Full:
- Non-leaves have exactly 2 children
- Each node has left and right subtree of same h
- All leaves at level h

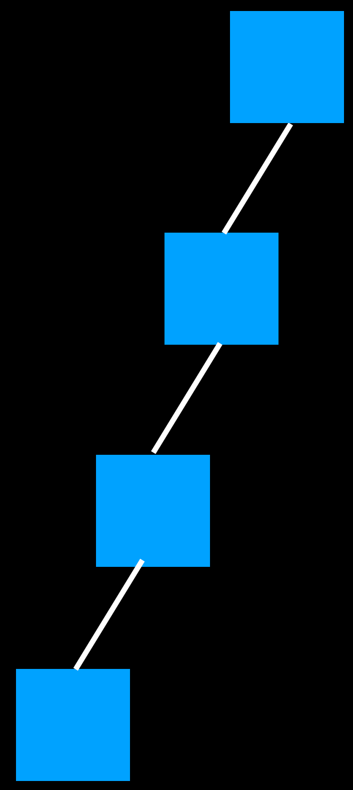
Complete:
- Full up to level h-1
- Level h filled from left to right
- All nodes at h-2 and above have exactly 2 children

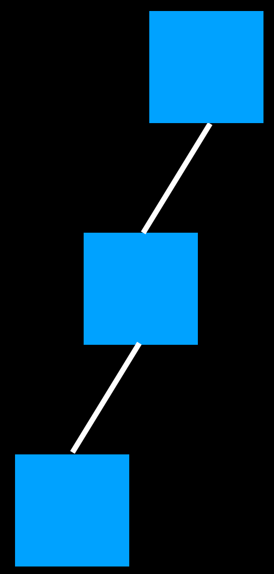
Balanced:
- For each node, left and right subtree height differ by at most 1

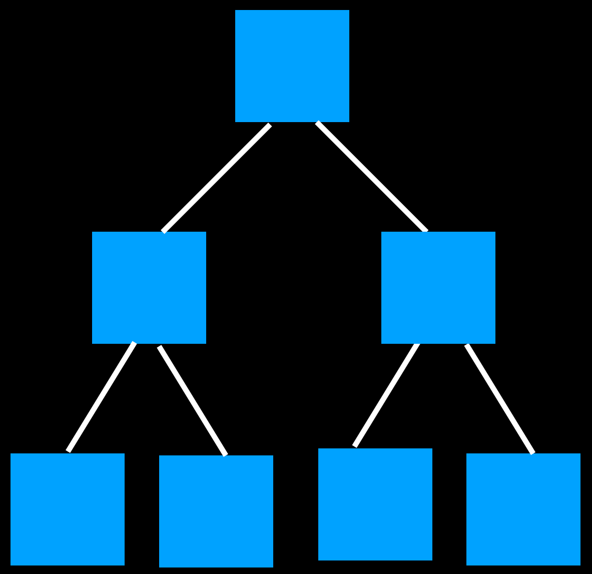
Recap: Max/Min Height

n nodes
every node 1 child
$h = n$
Essentially a chain

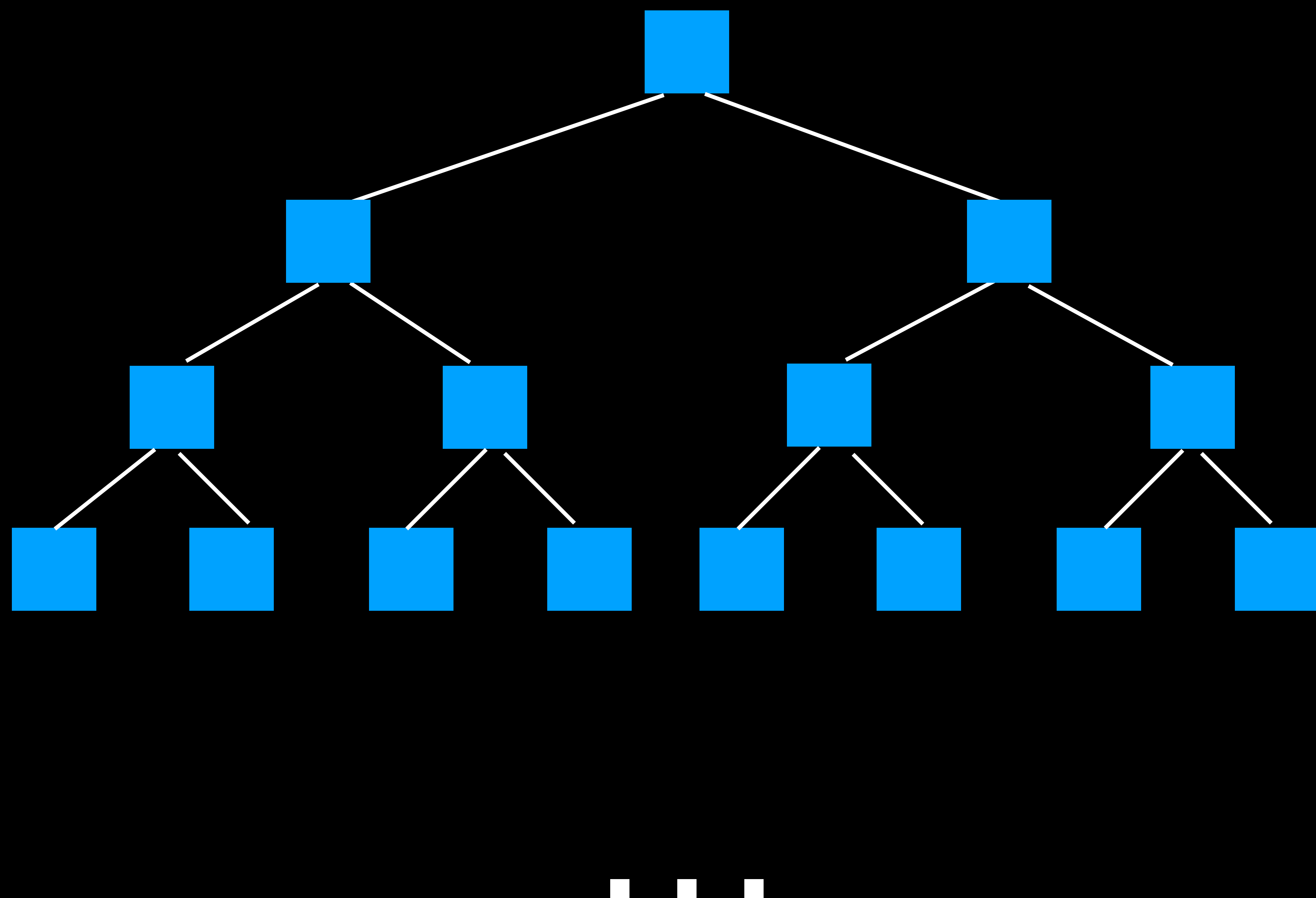
Binary tree of height h can have up to $n = 2^h - 1$
For example for $h = 3$, $1 + 2 + 4 = 7 = 2^3 - 1$
$h = \log(n+1)$ for a **full binary tree**







Recap



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}	$2^h - 1$

Recap

```
#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

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

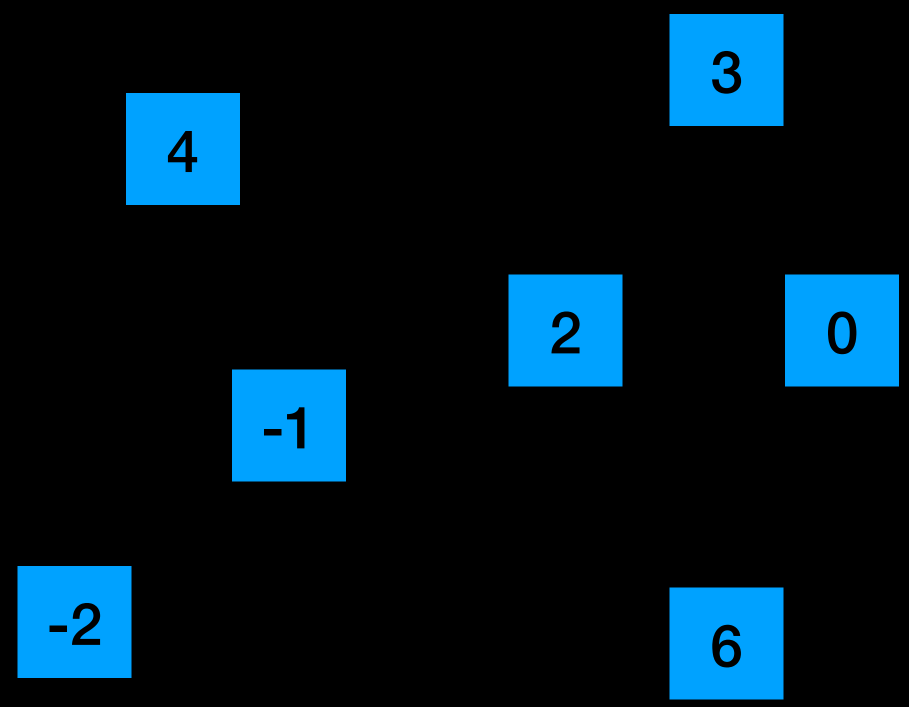
Can we do better?

Find an element: O(n)

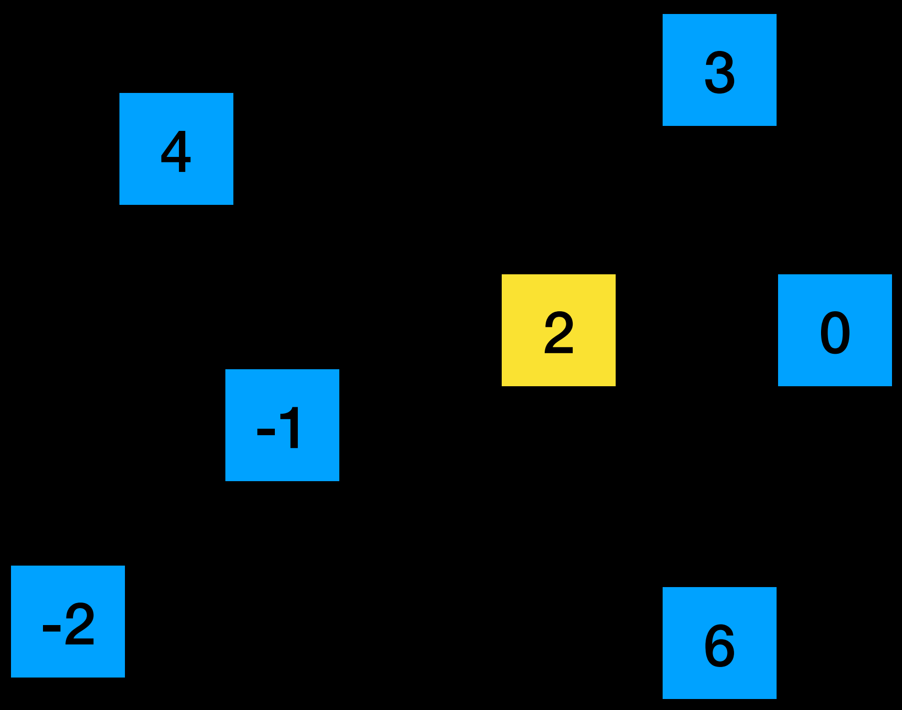
Remove: Find element and if there remove it O(n)

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

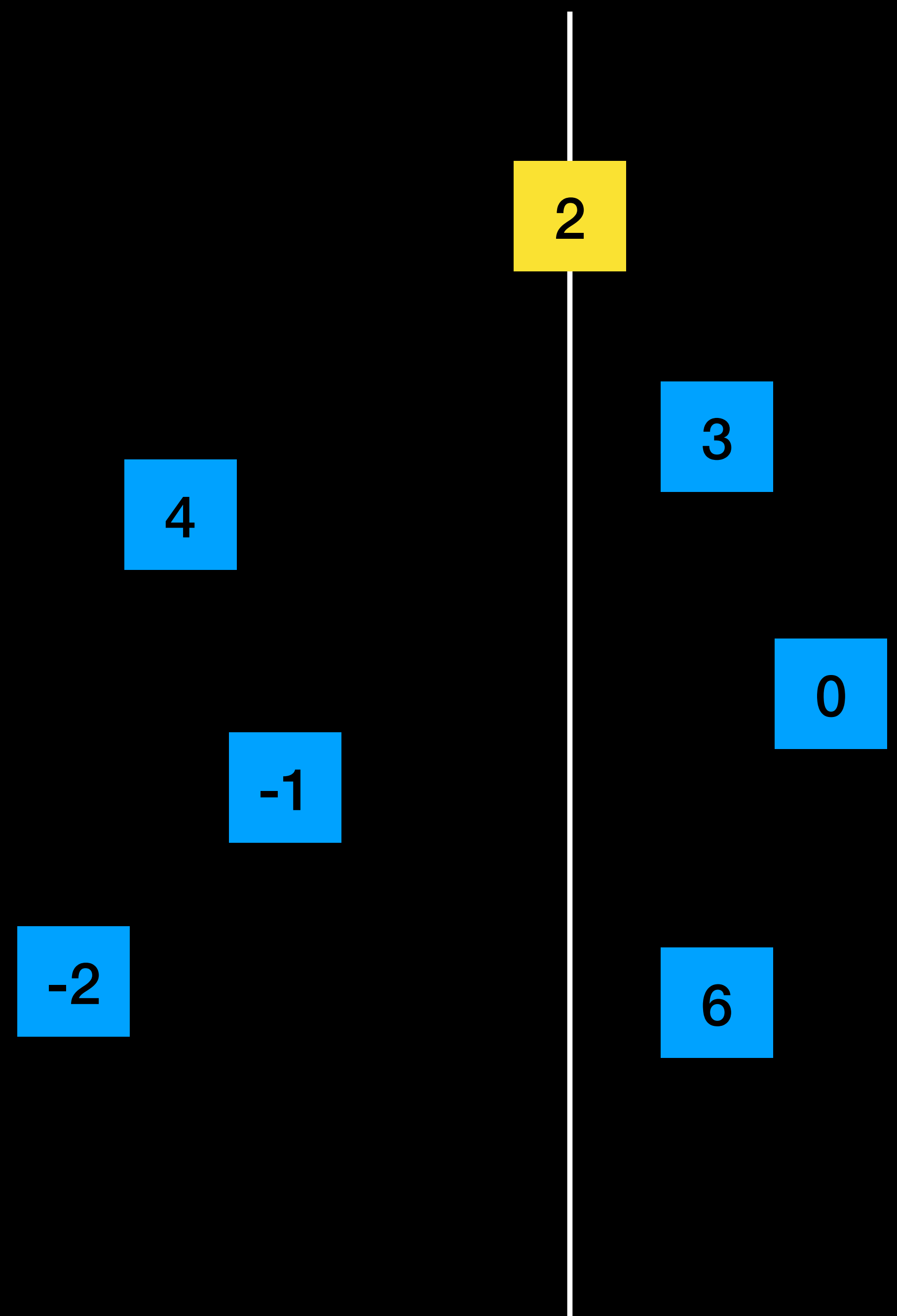
A Different Approach



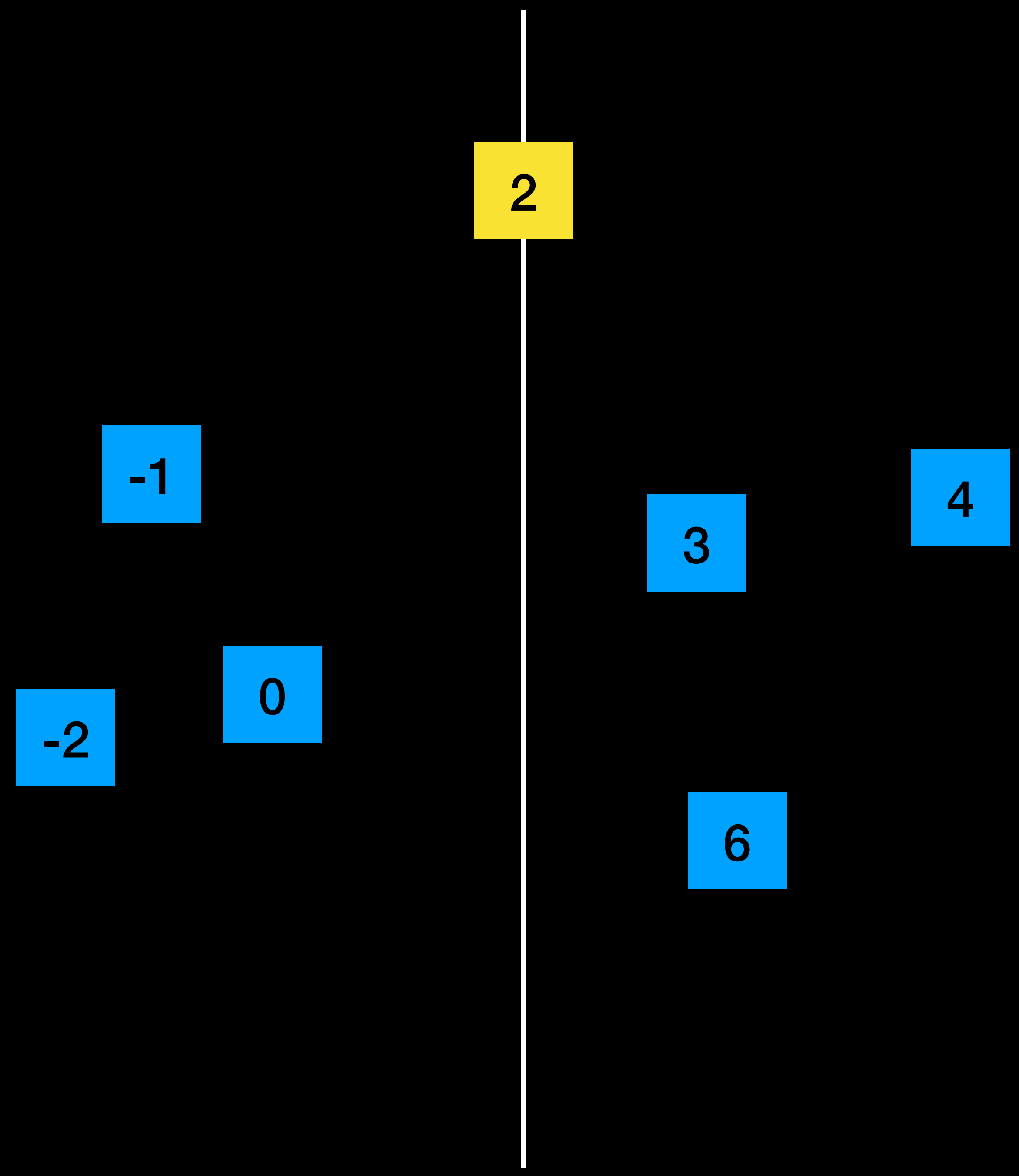
A Different Approach



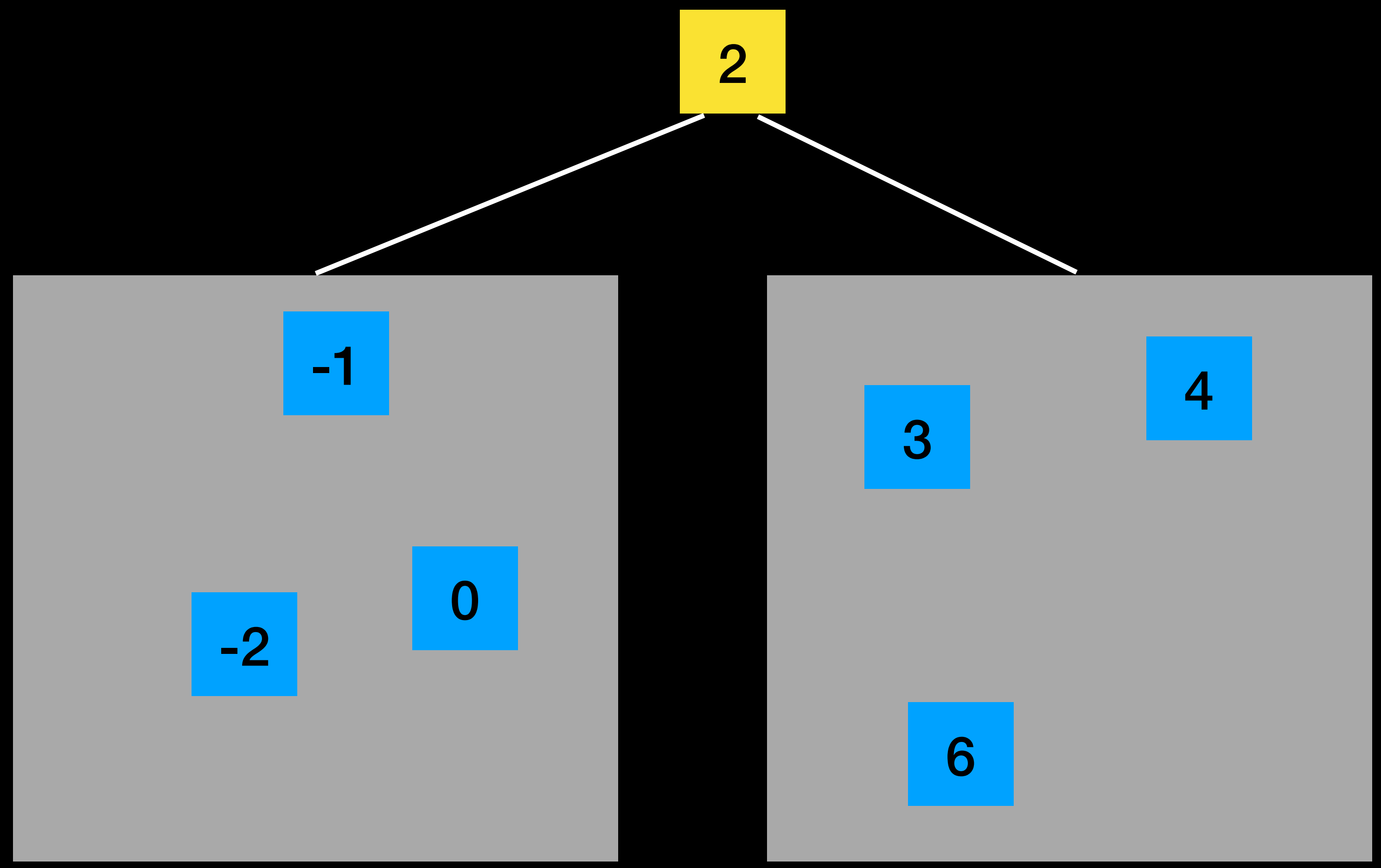
A Different Approach



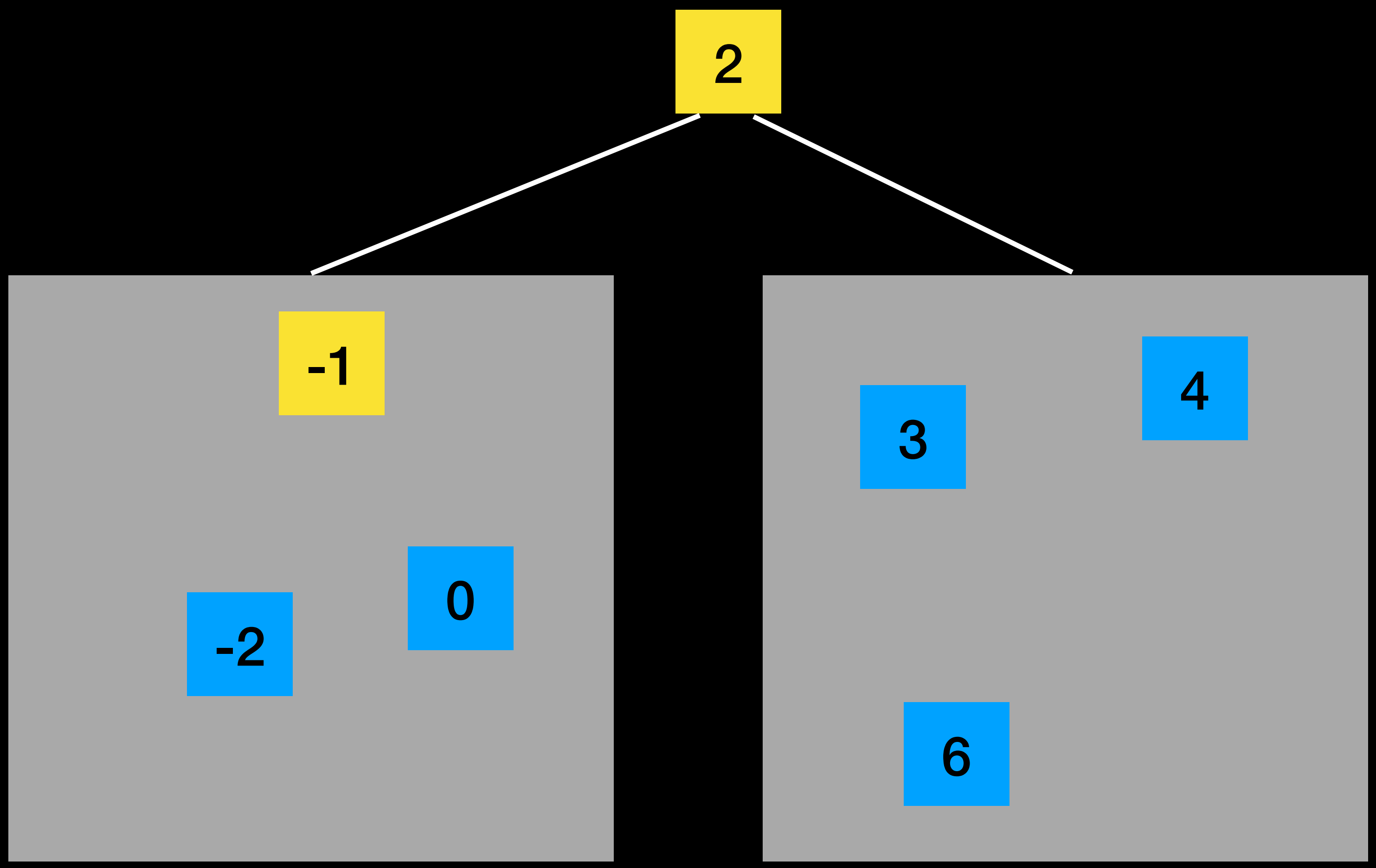
A Different Approach



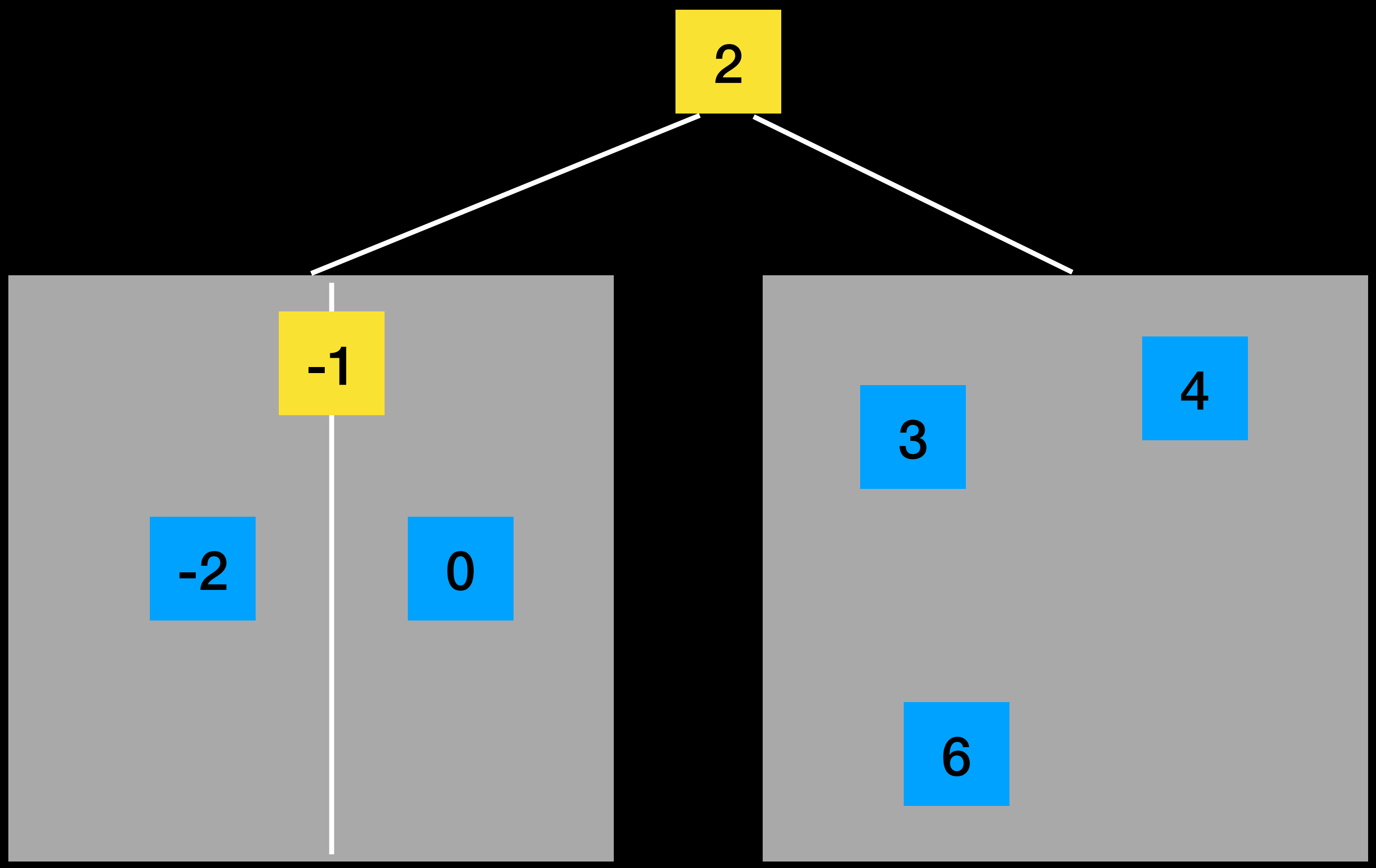
A Different Approach



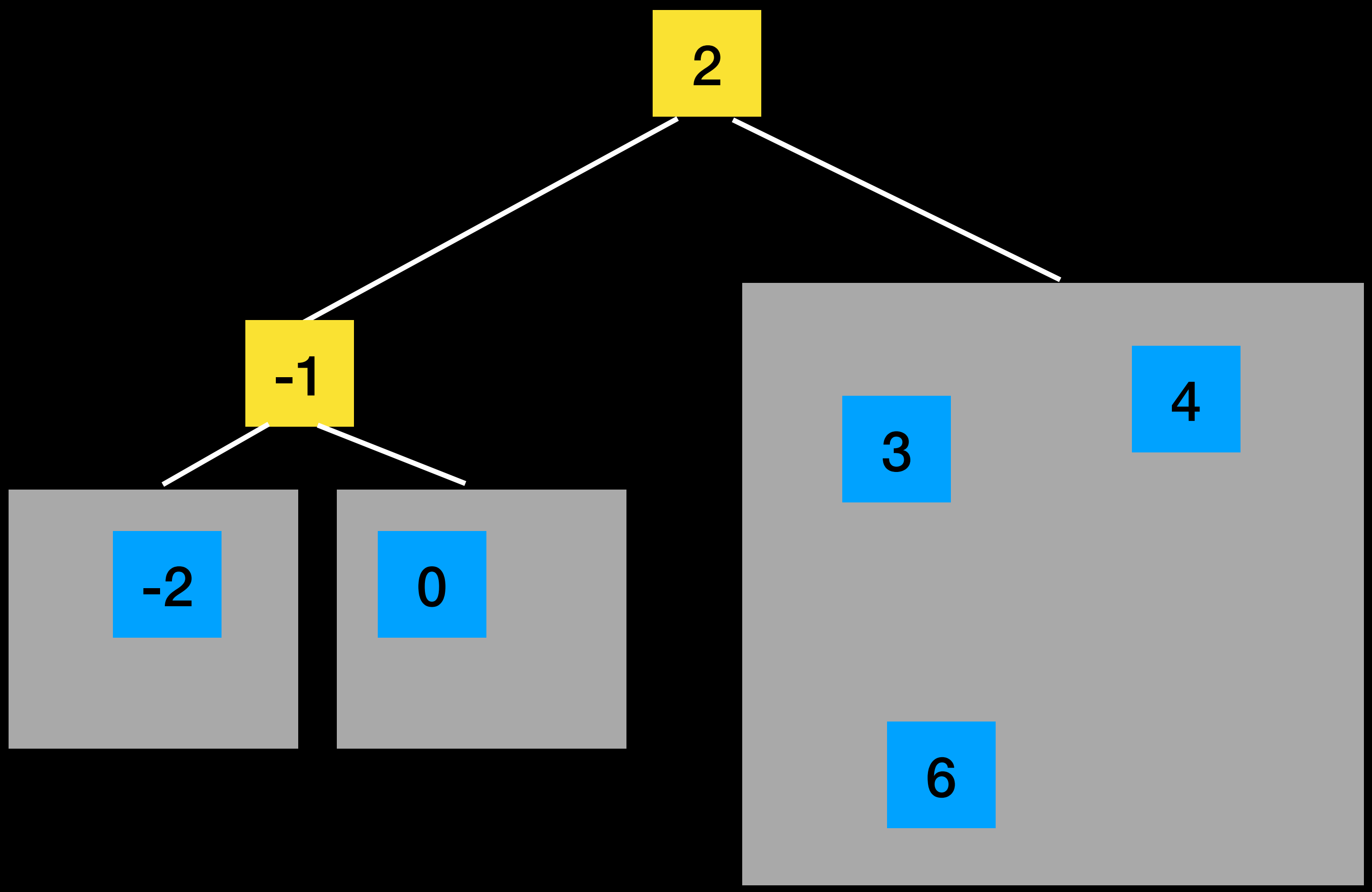
A Different Approach



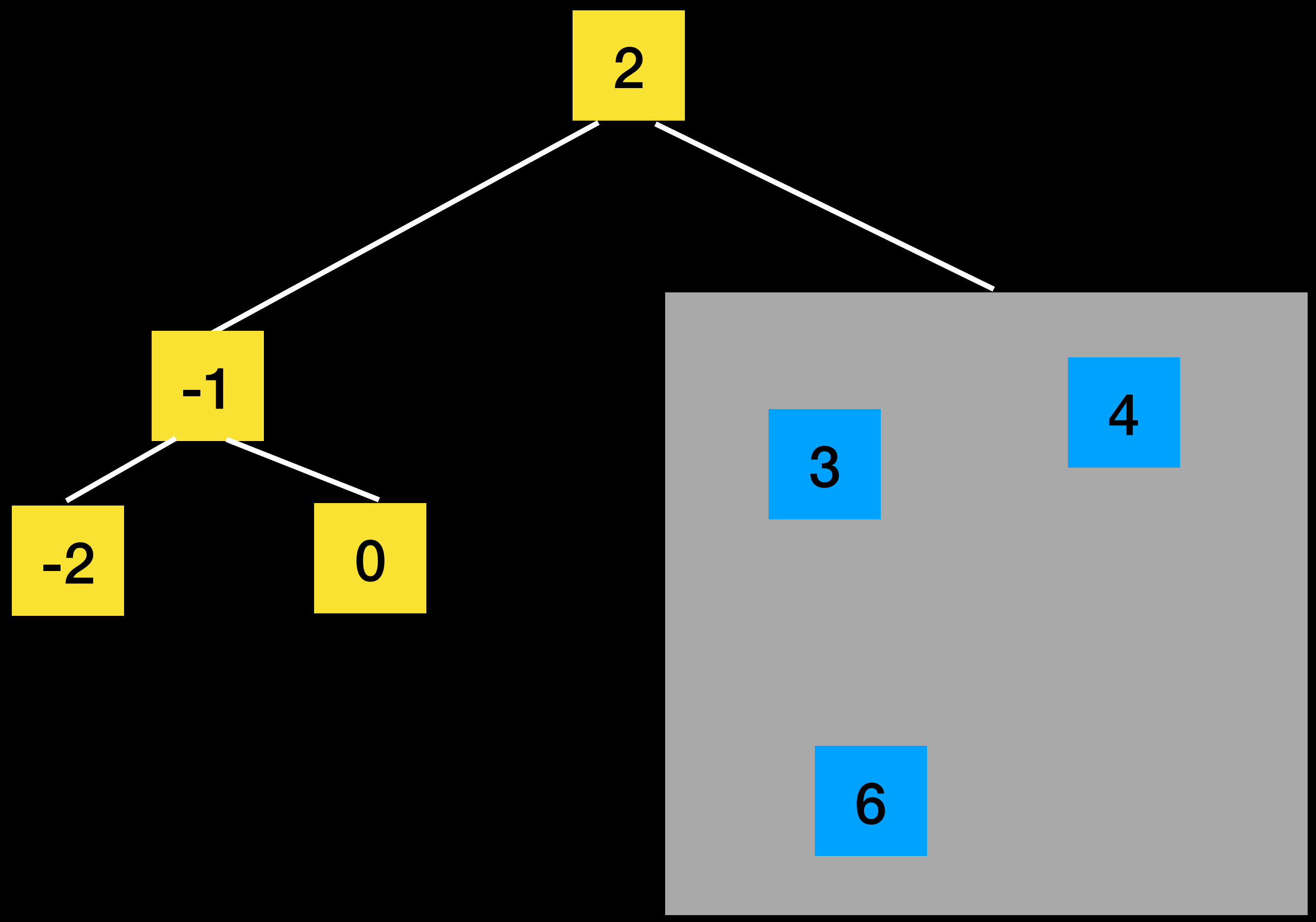
A Different Approach



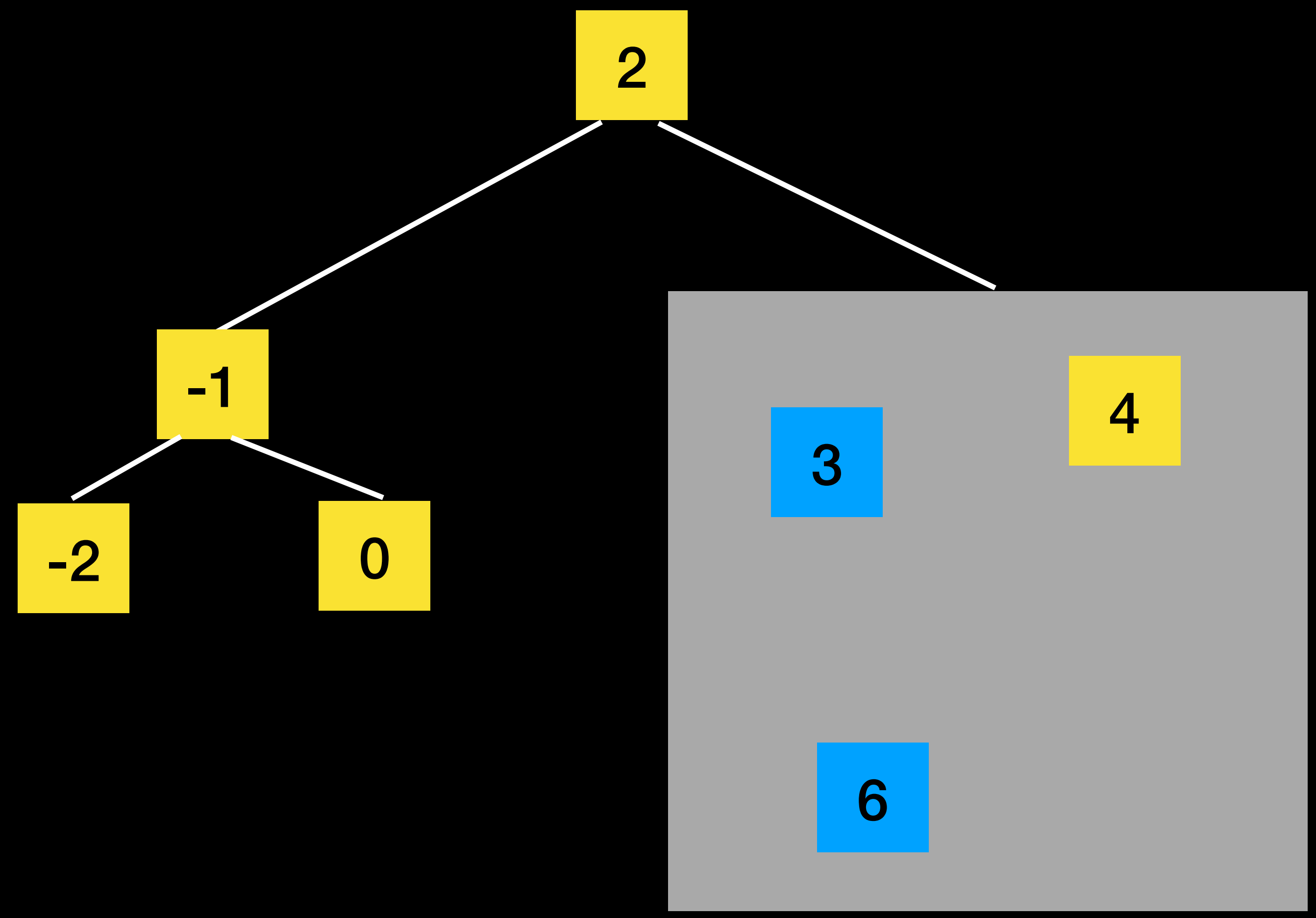
A Different Approach



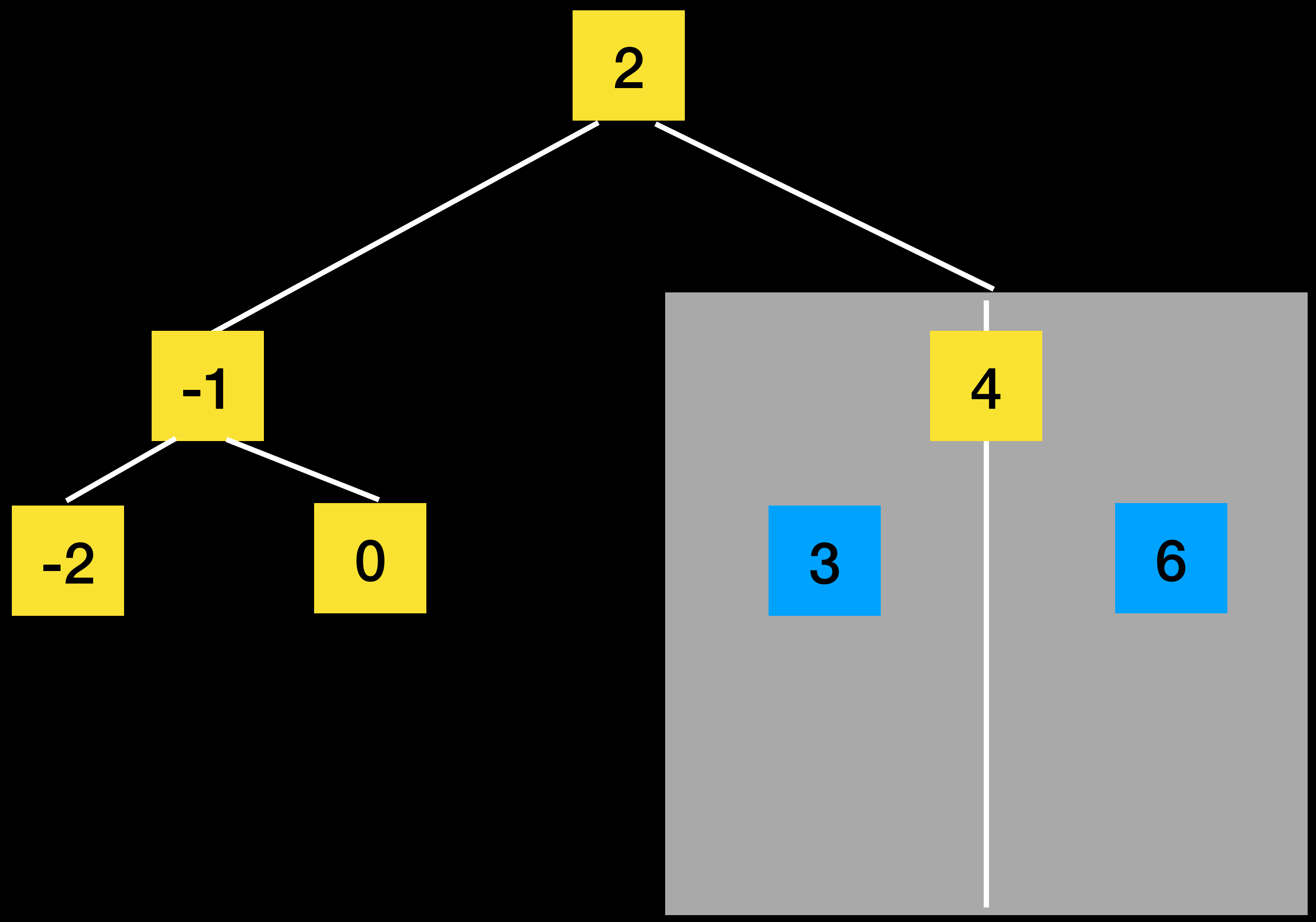
A Different Approach



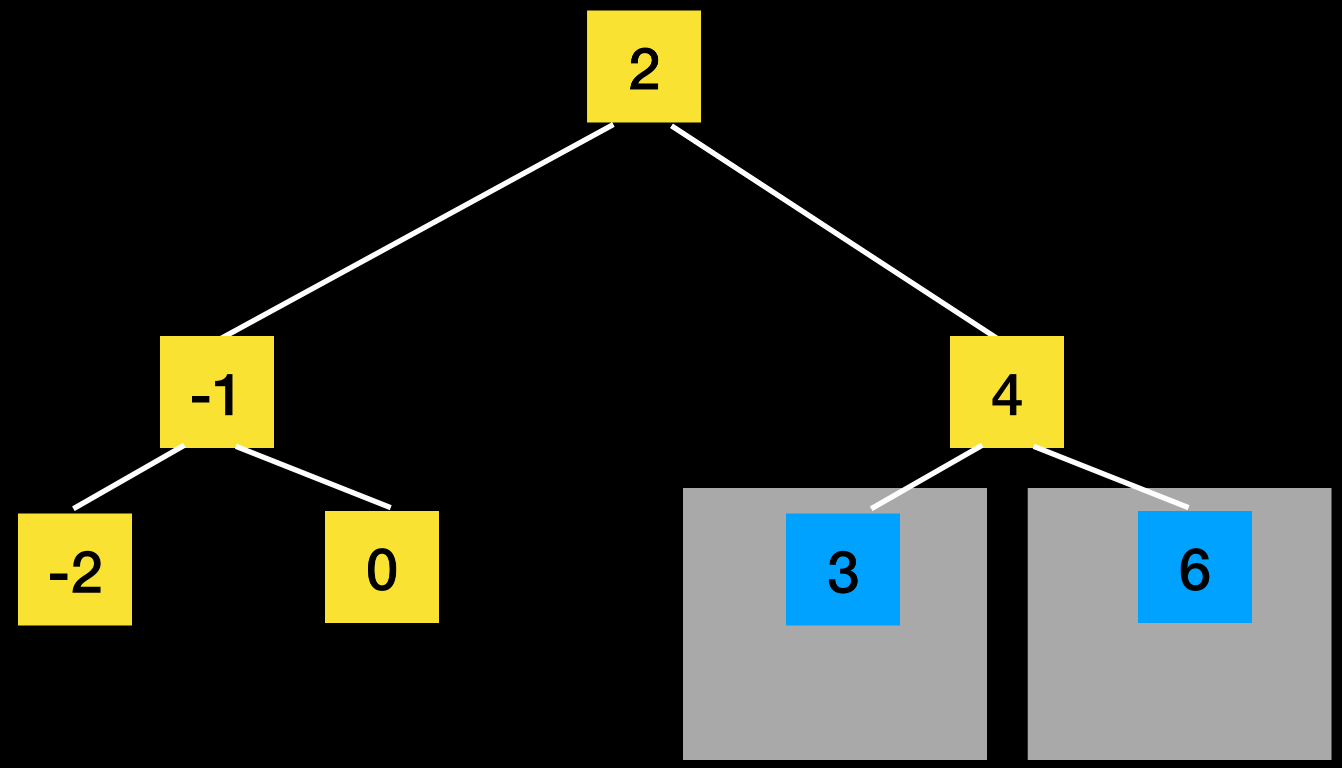
A Different Approach



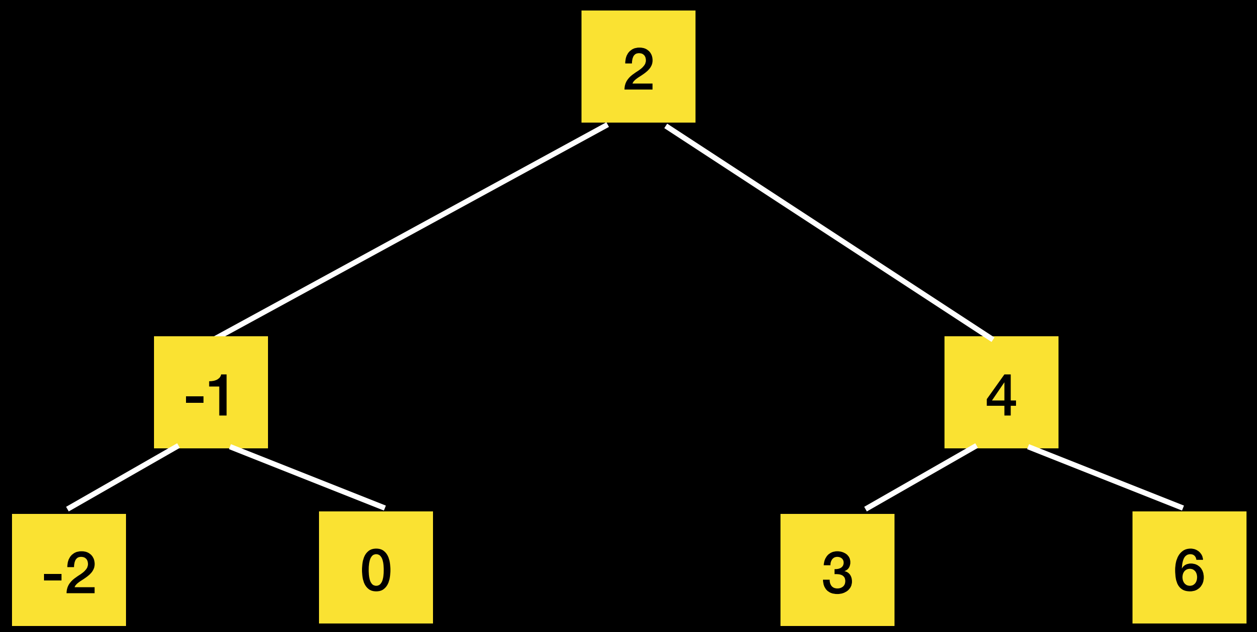
A Different Approach



A Different Approach

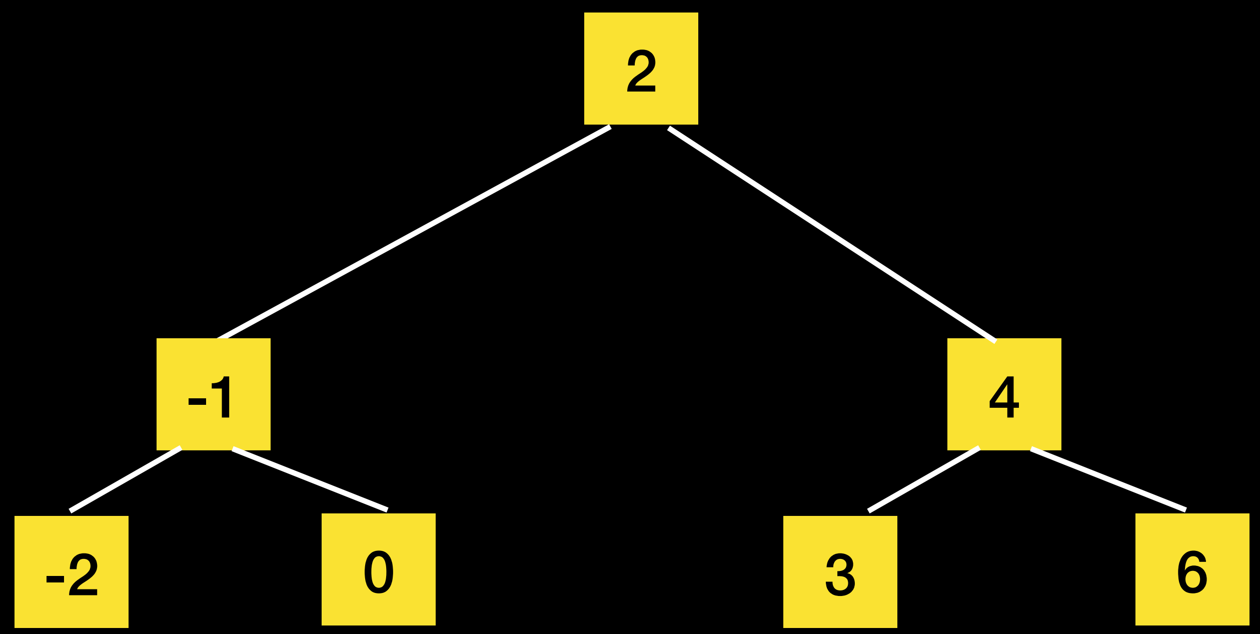


A Different Approach



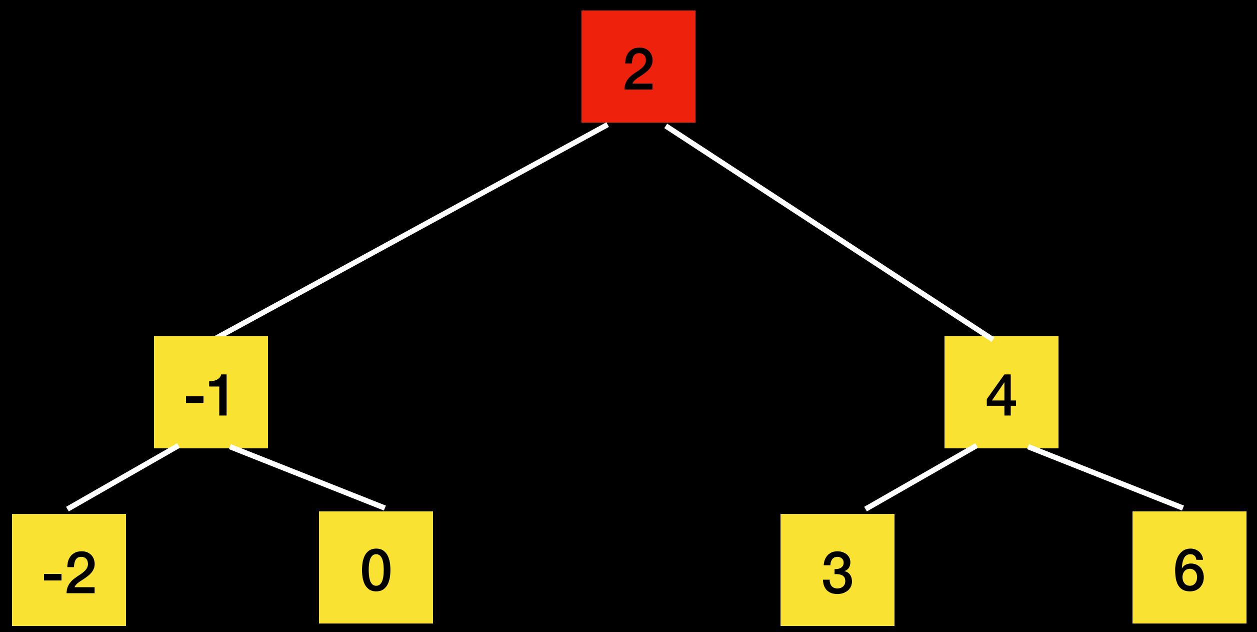
A Different Approach

Find 5



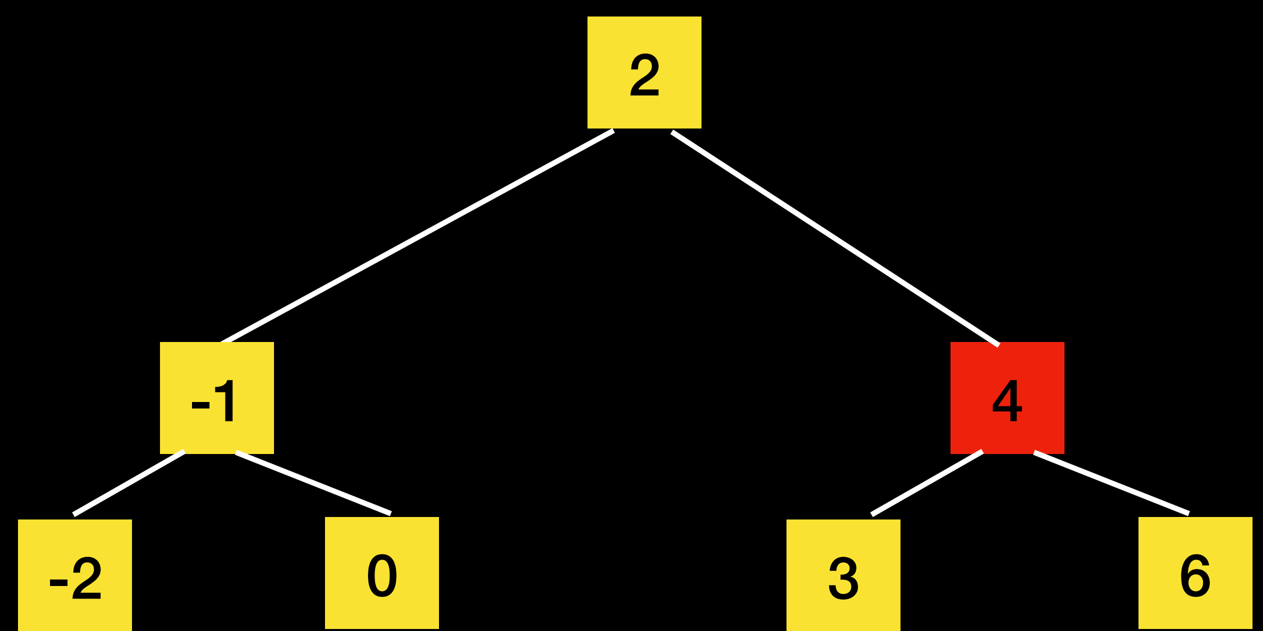
A Different Approach

Find 5



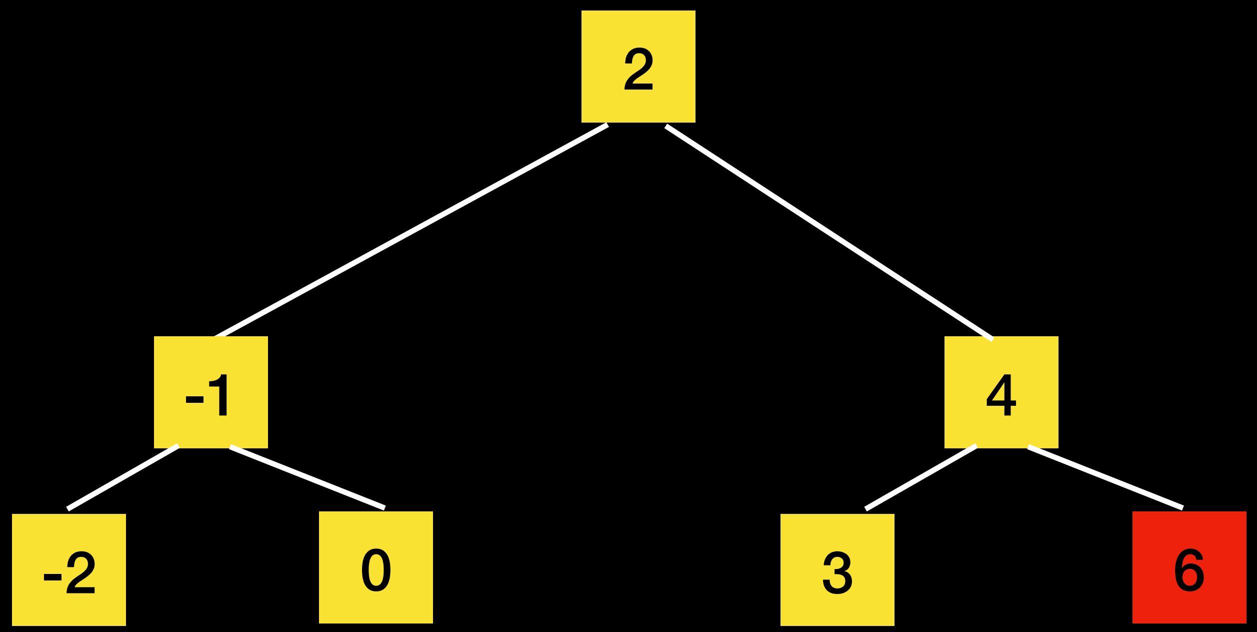
A Different Approach

Find 5



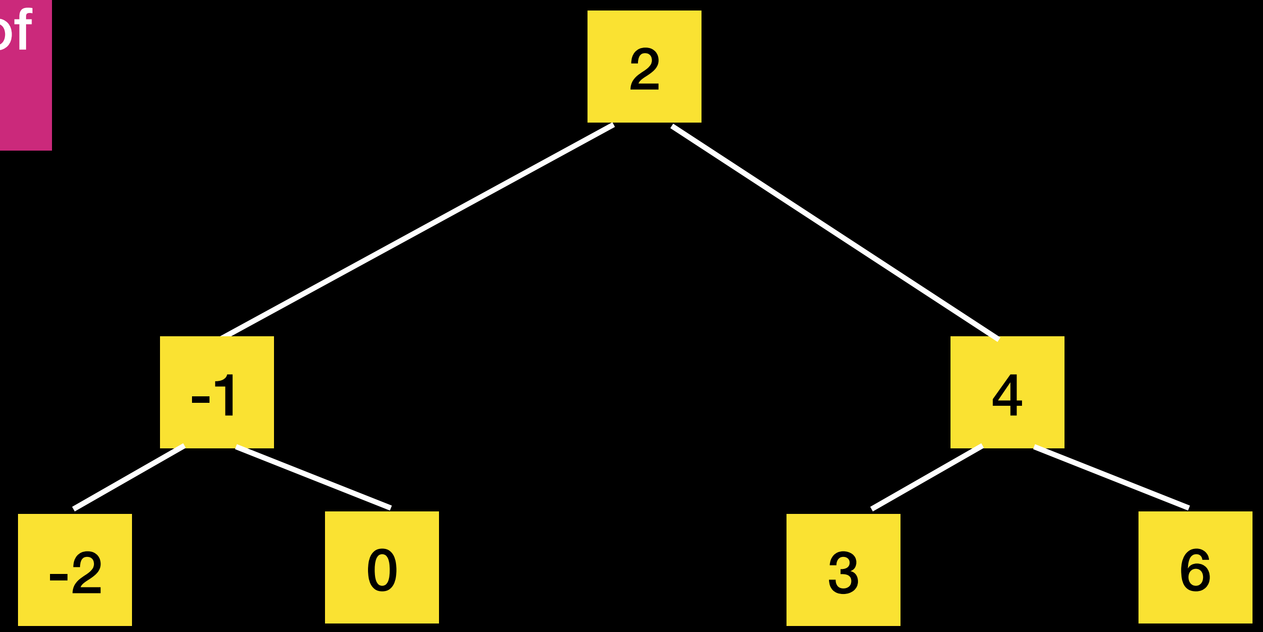
A Different Approach

Find 5



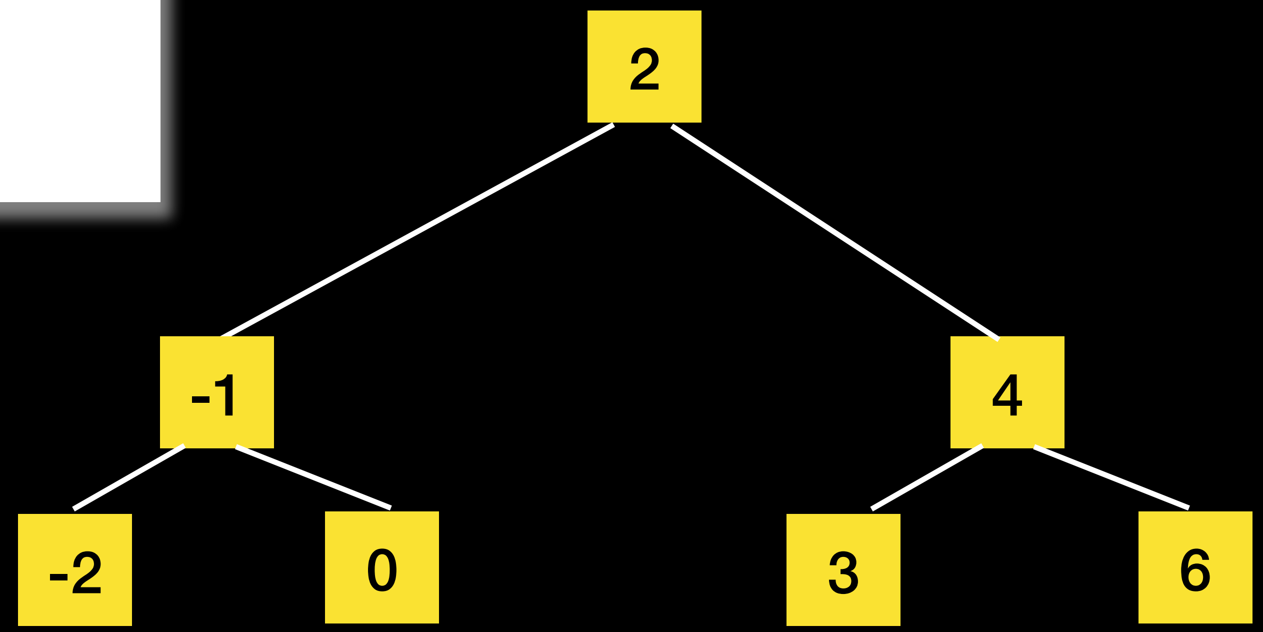
A Different Approach

What's special about the shape of this tree?



Binary Search Tree

Structural Property:
For each node n
n > 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 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 $p < n$.

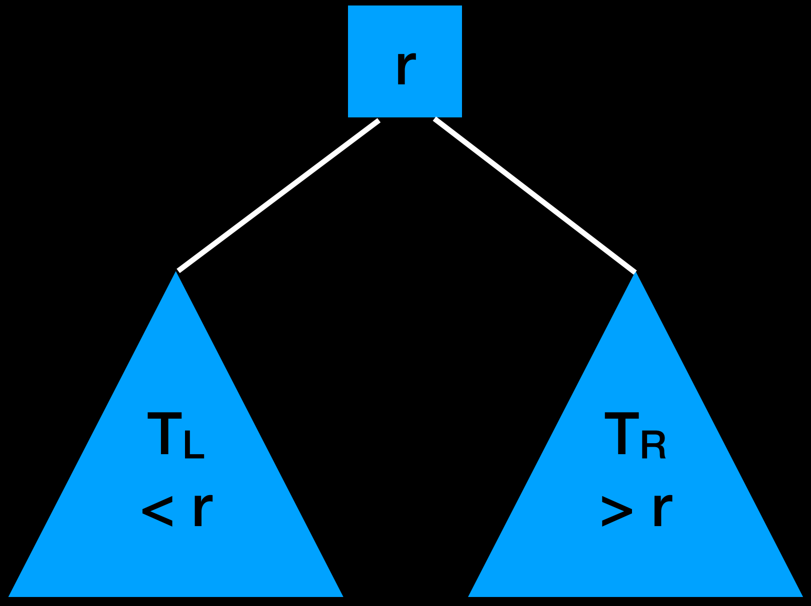
- For each node $n \in T$, if p is a node in the right subtree of n, then $n < p$.

- For each element $s \in S$ there exists a node $n \in T$ such that $s = n$.

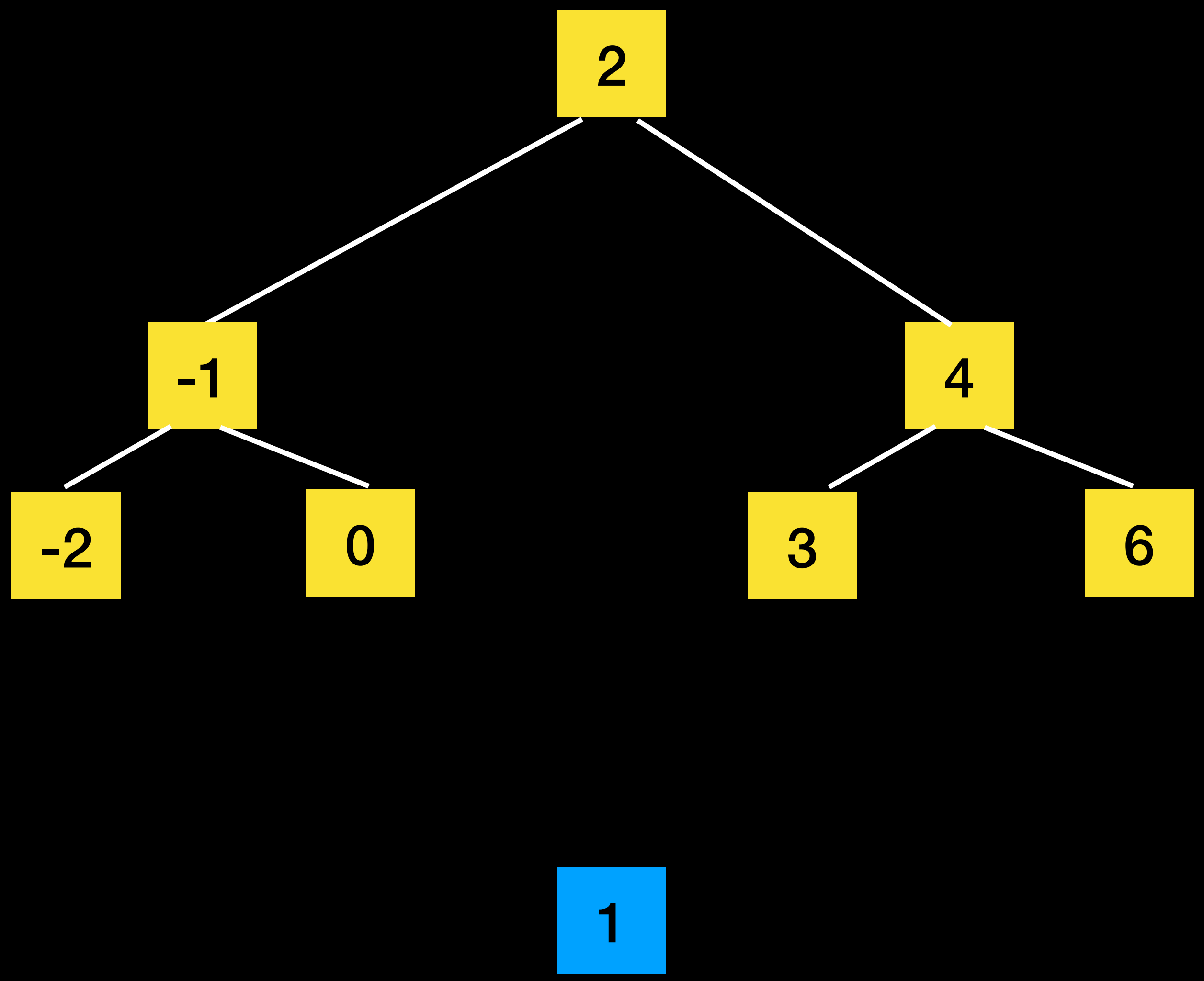
Binary Search Tree

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

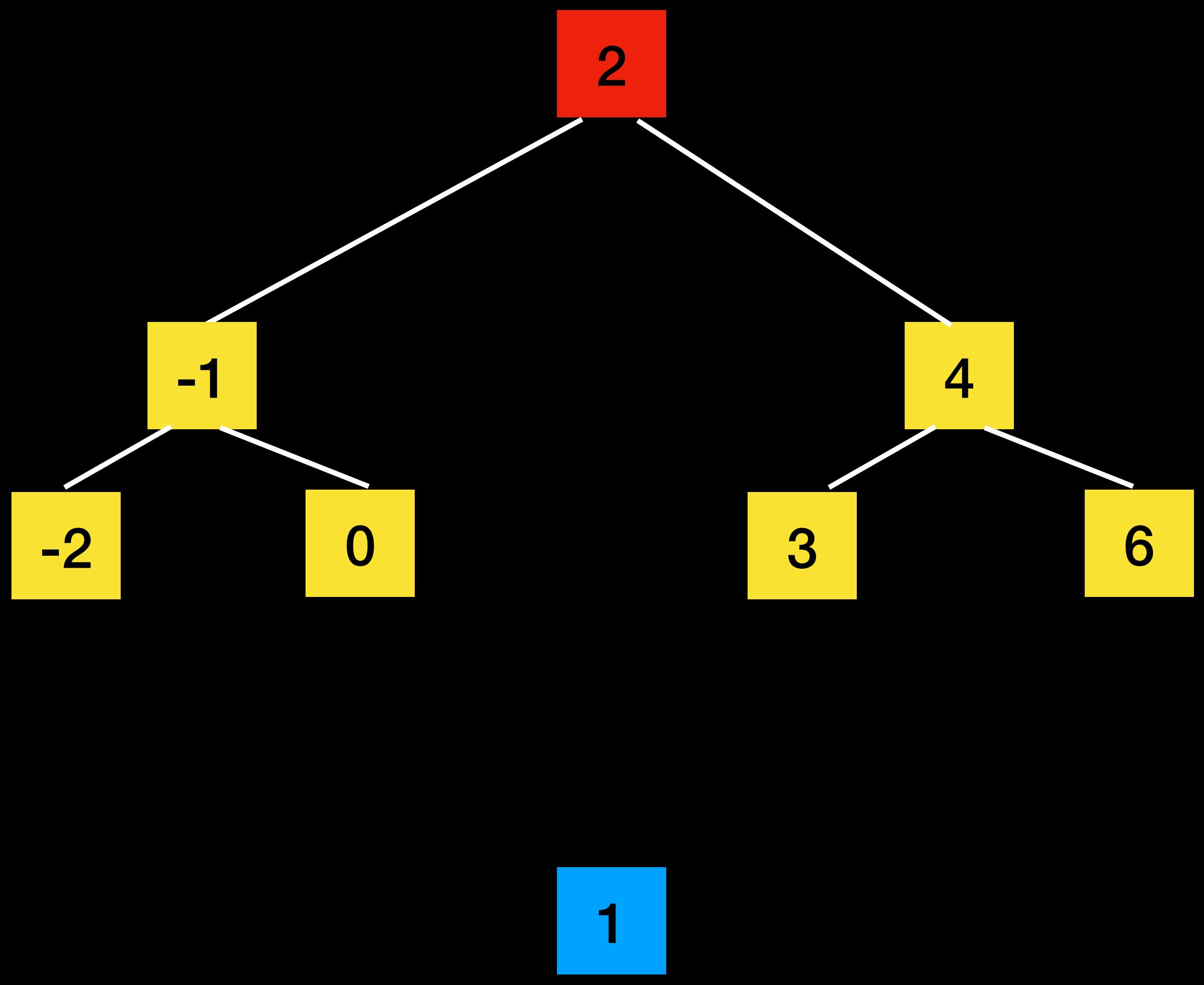
```
search(bs_tree, item)
{
    if (bs_tree is empty) //base case
        item not found
    else if (item == root)
        return root
    else if (item < root)
        search(TL , item)
    else // item > root
        search(TR , item)
}
```



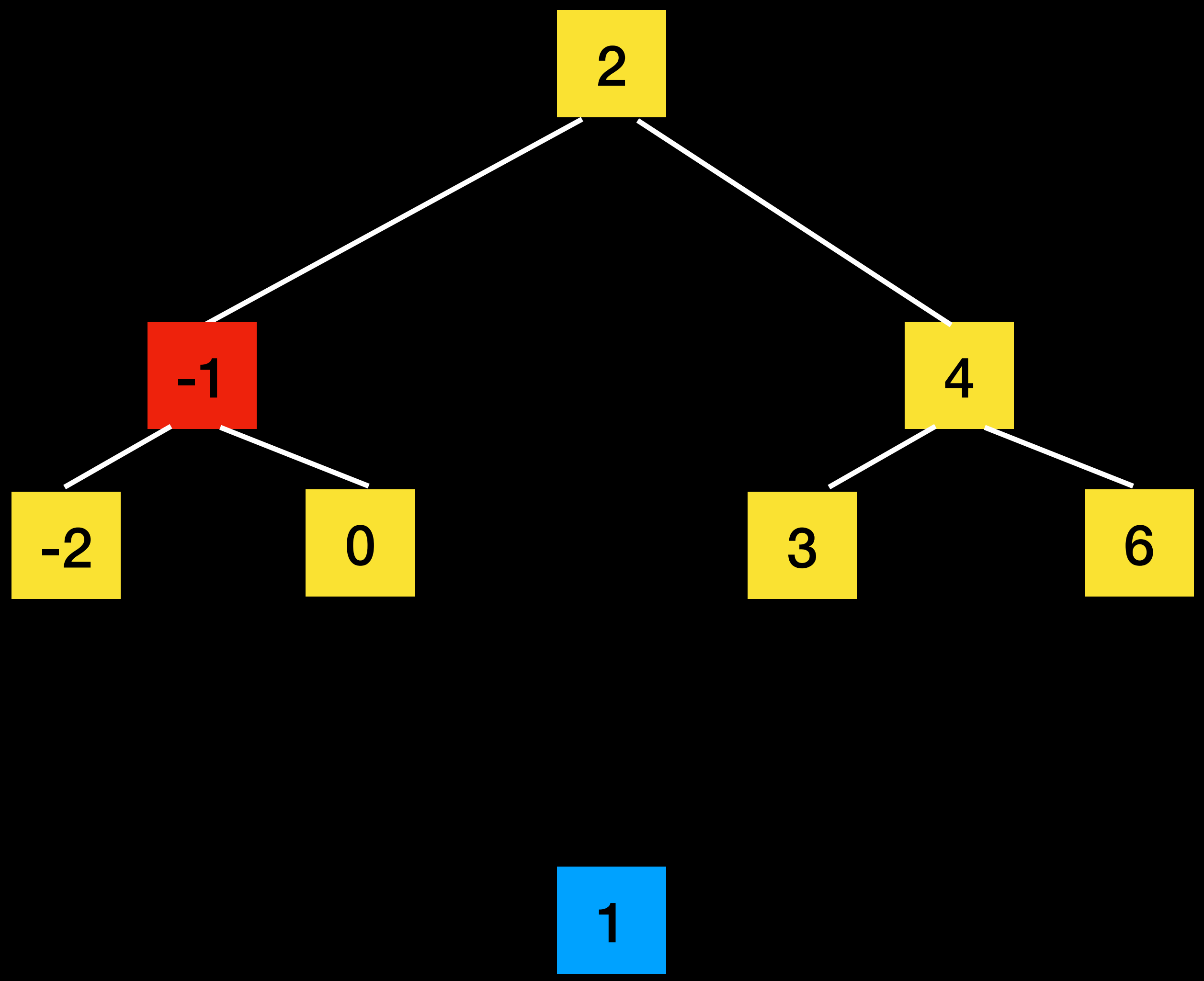
Inserting into a BST



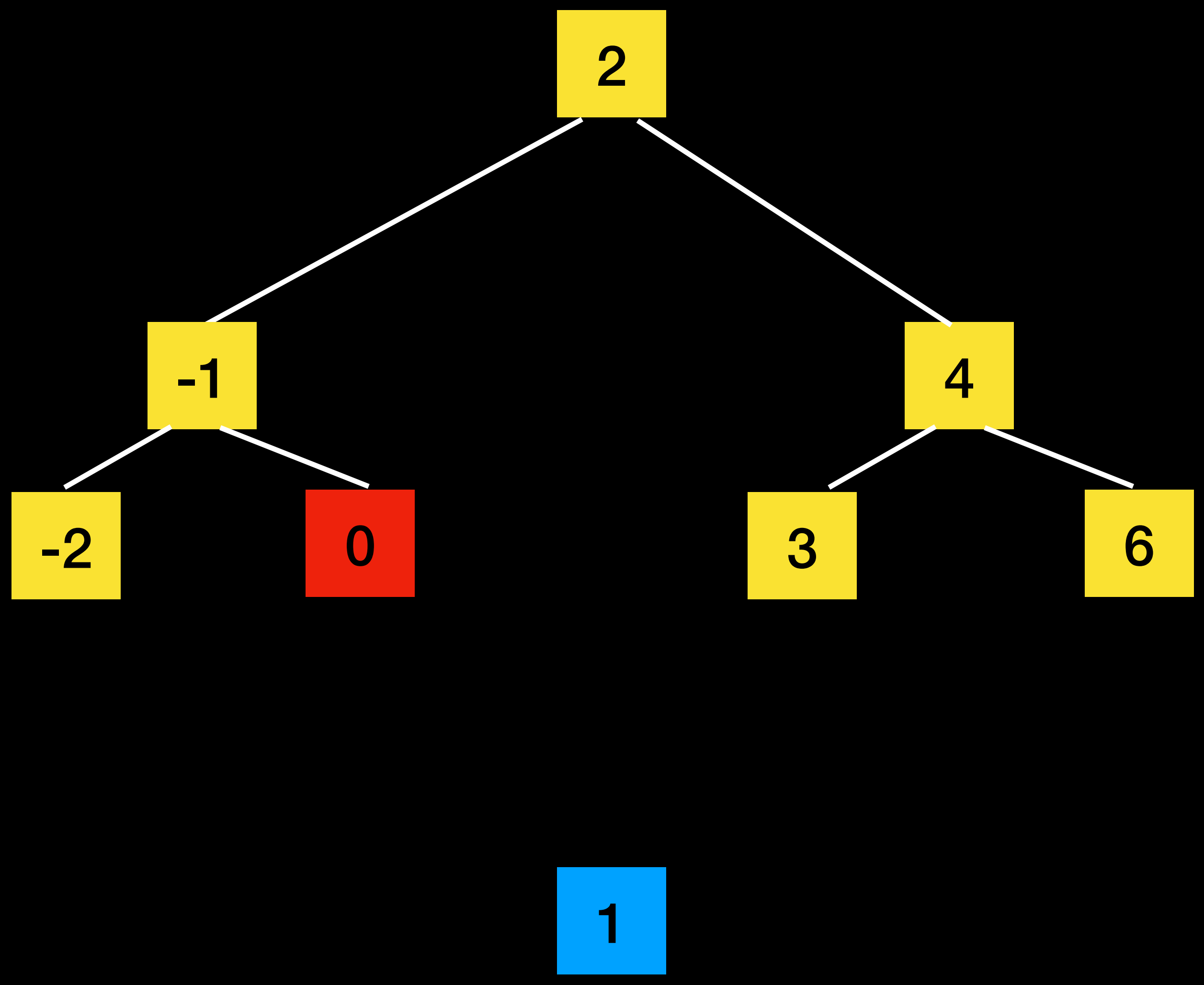
Inserting into a BST



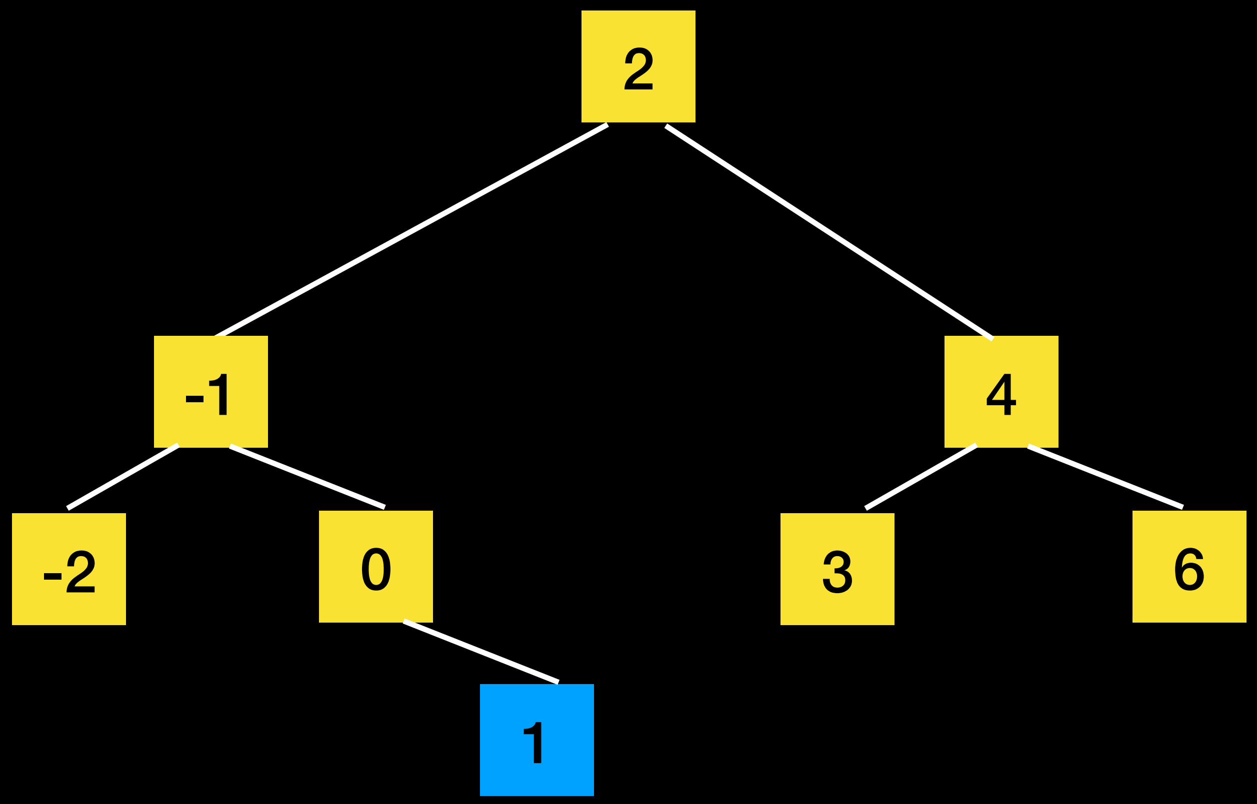
Inserting into a BST



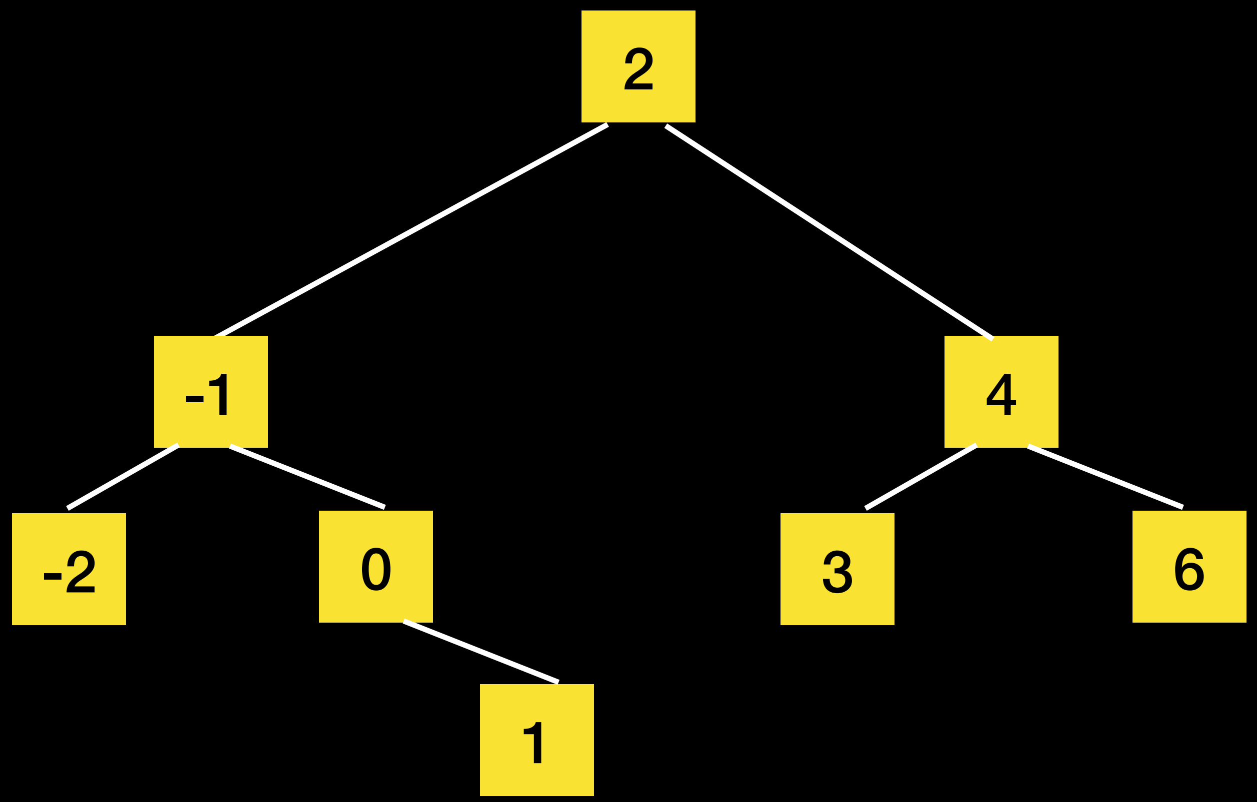
Inserting into a BST



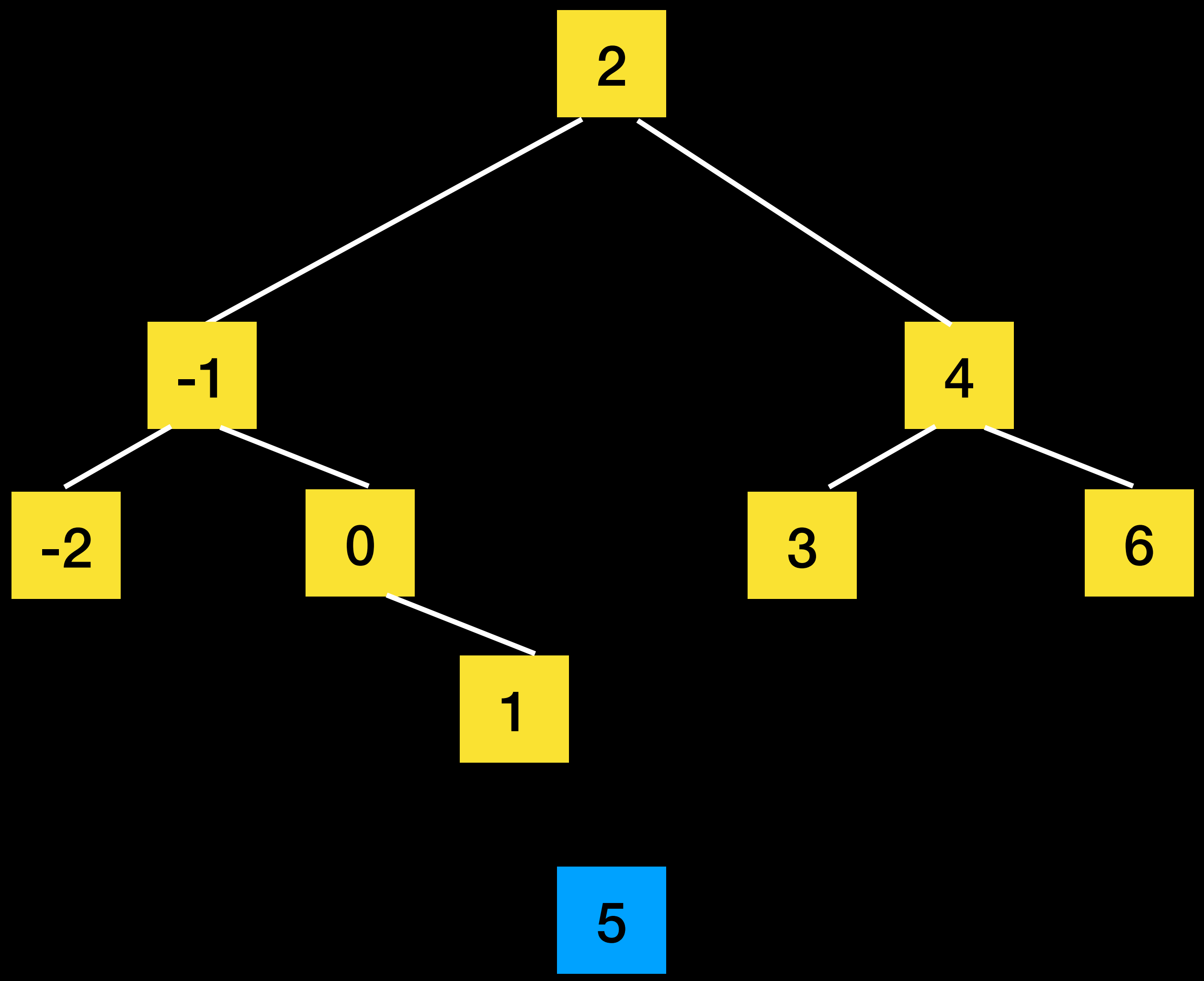
Inserting into a BST



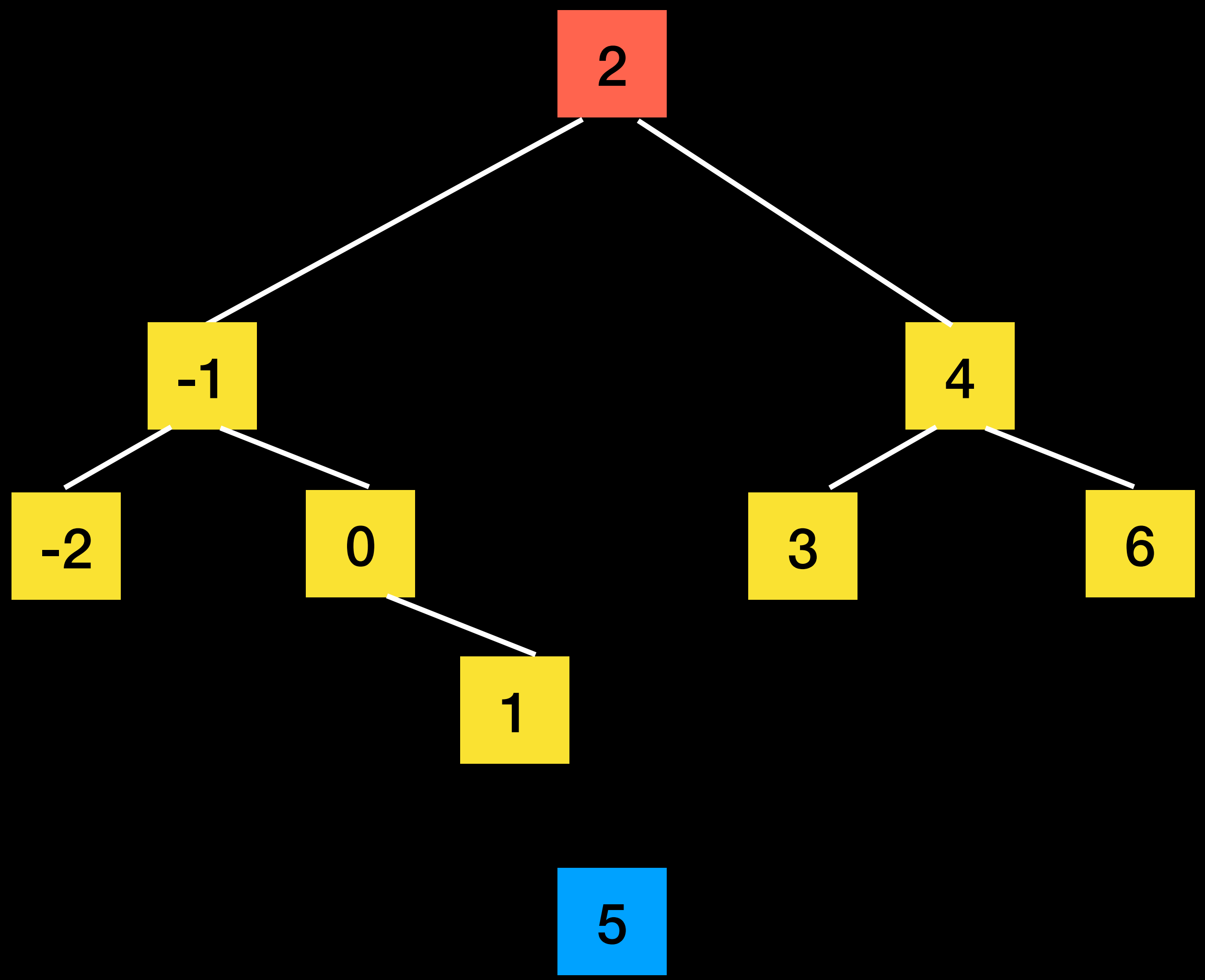
Inserting into a BST



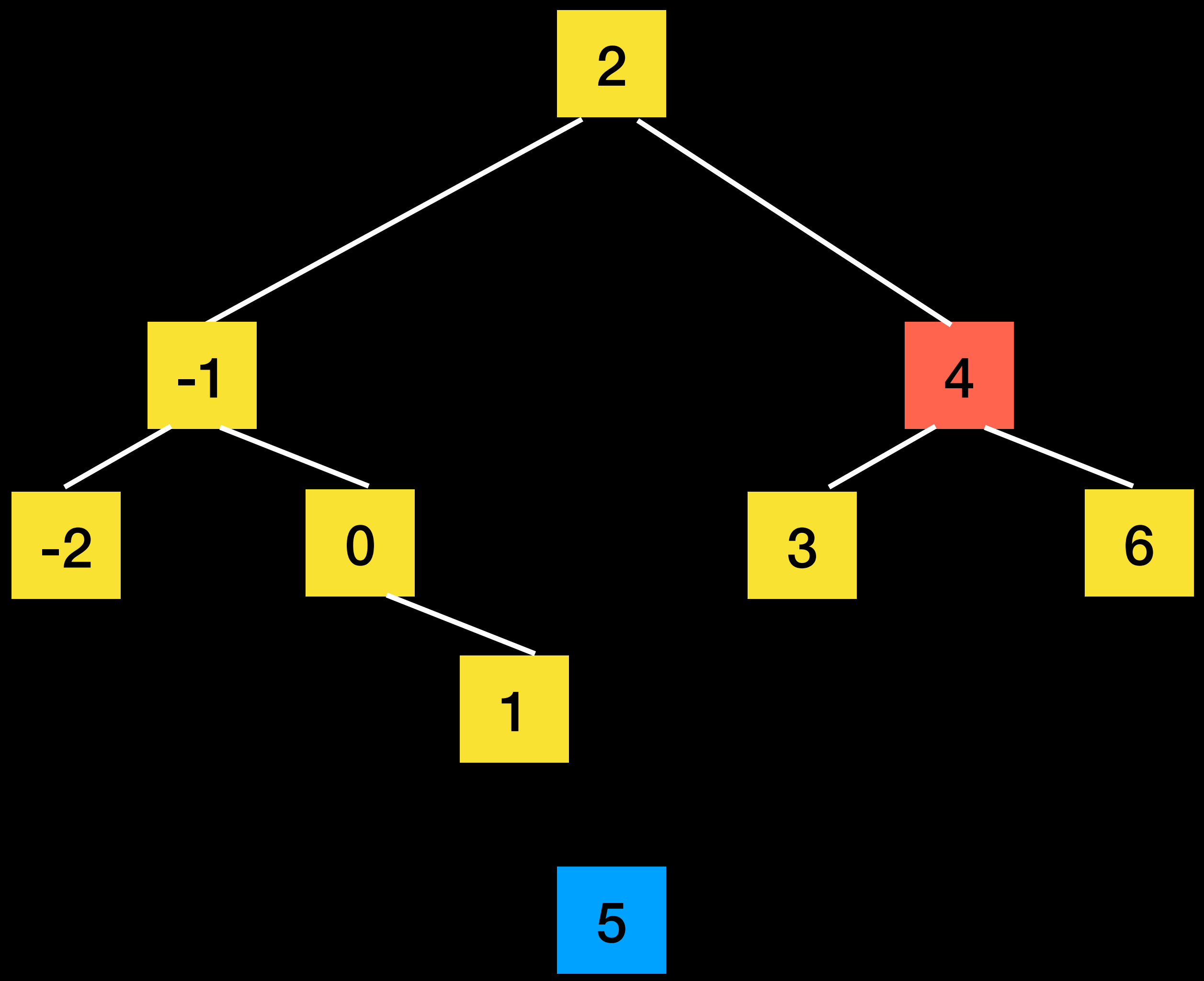
Inserting into a BST



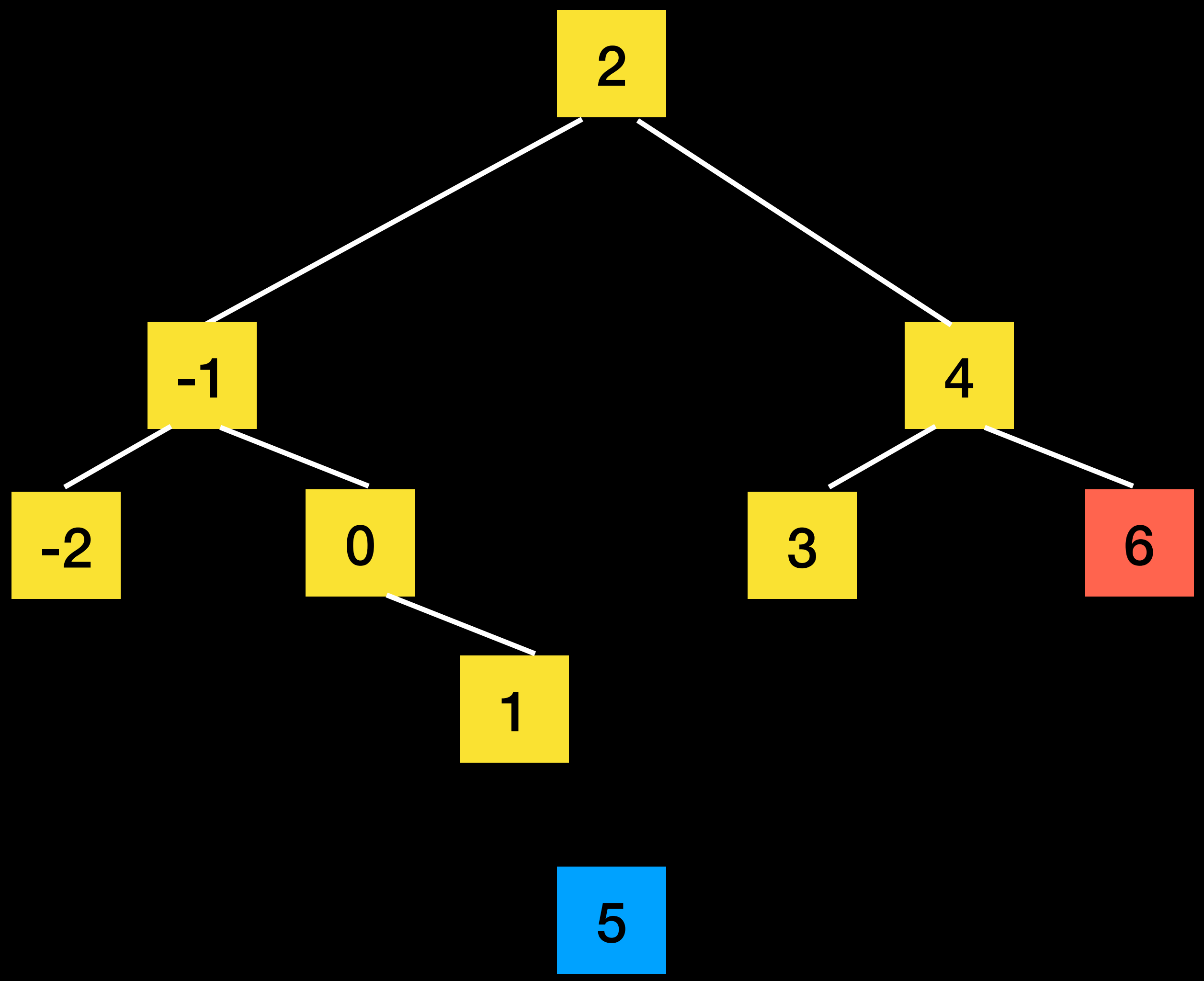
Inserting into a BST



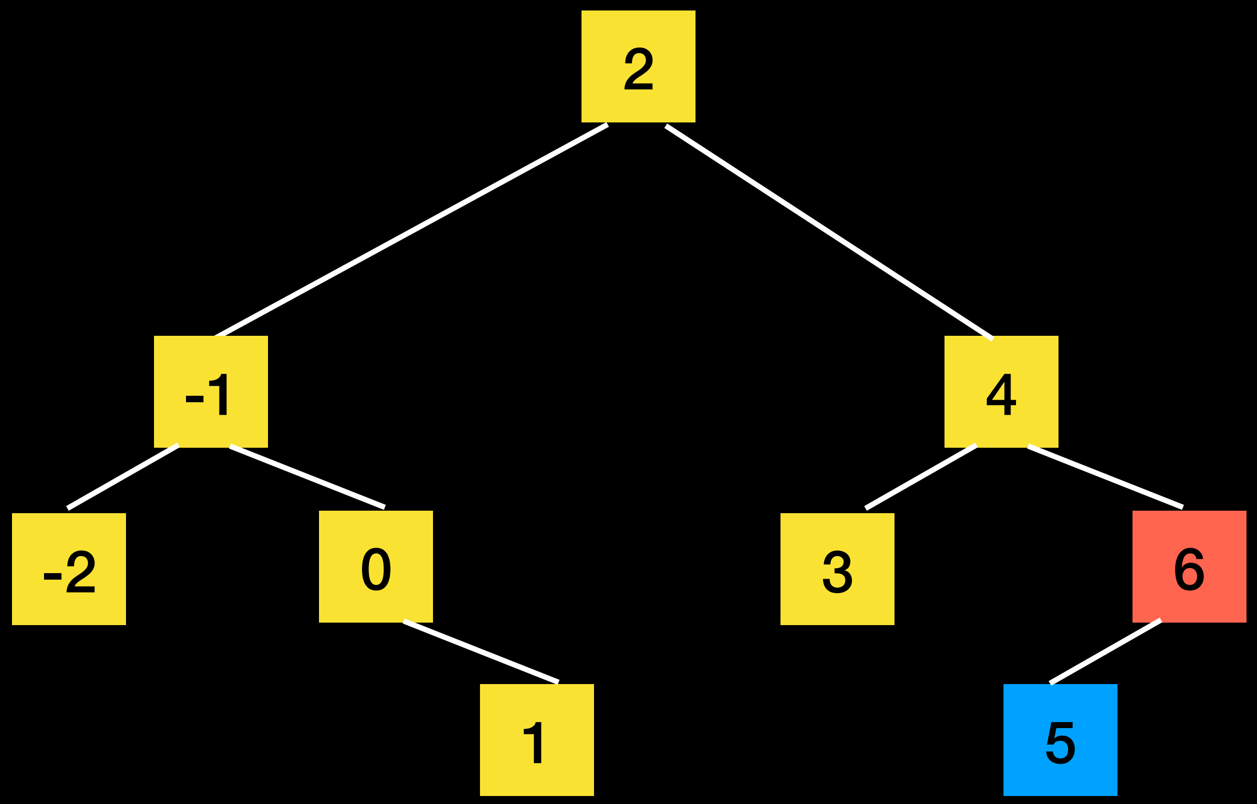
Inserting into a BST



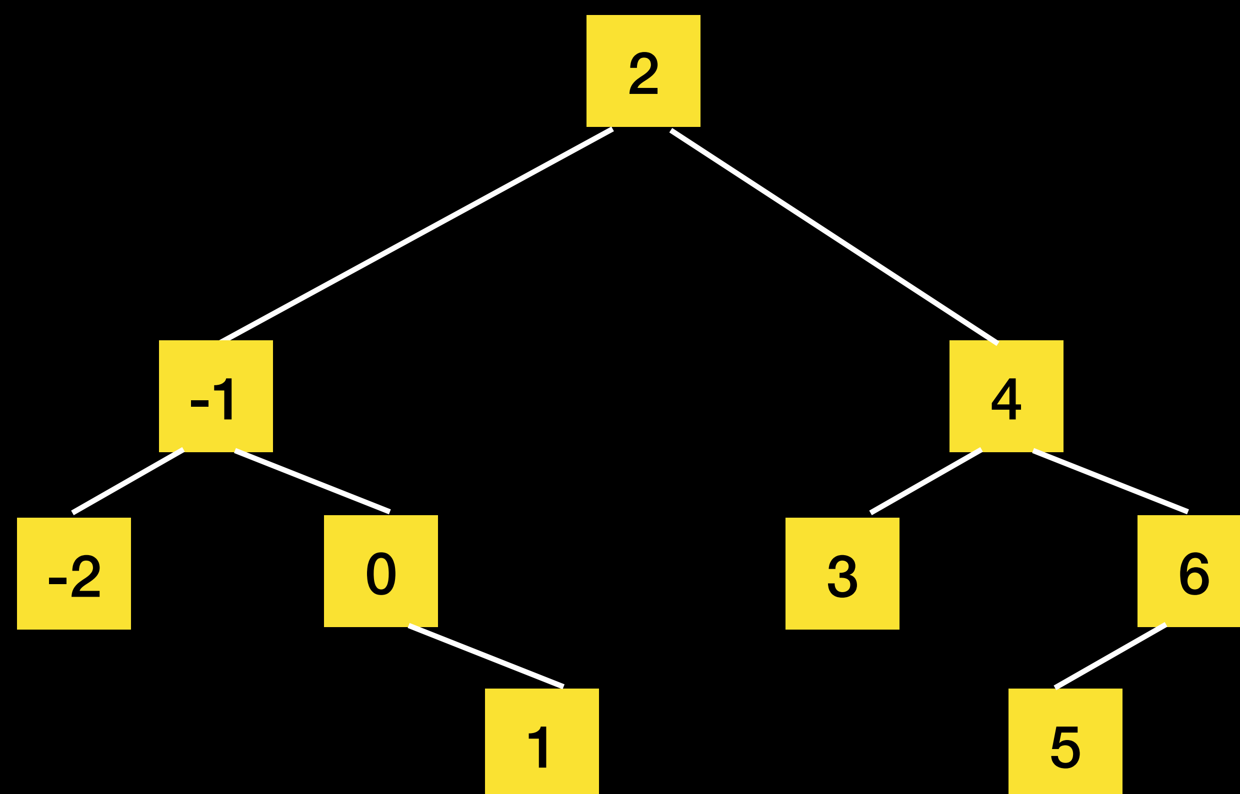
Inserting into a BST



Inserting into a BST

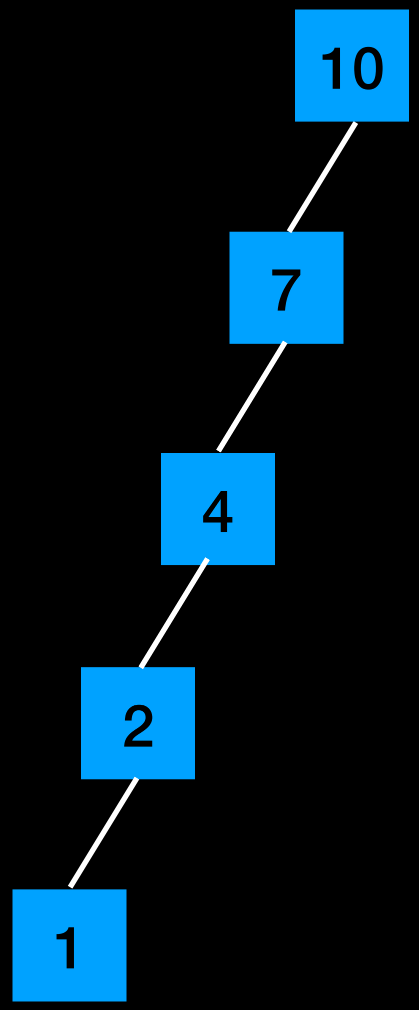


Inserting into a BST

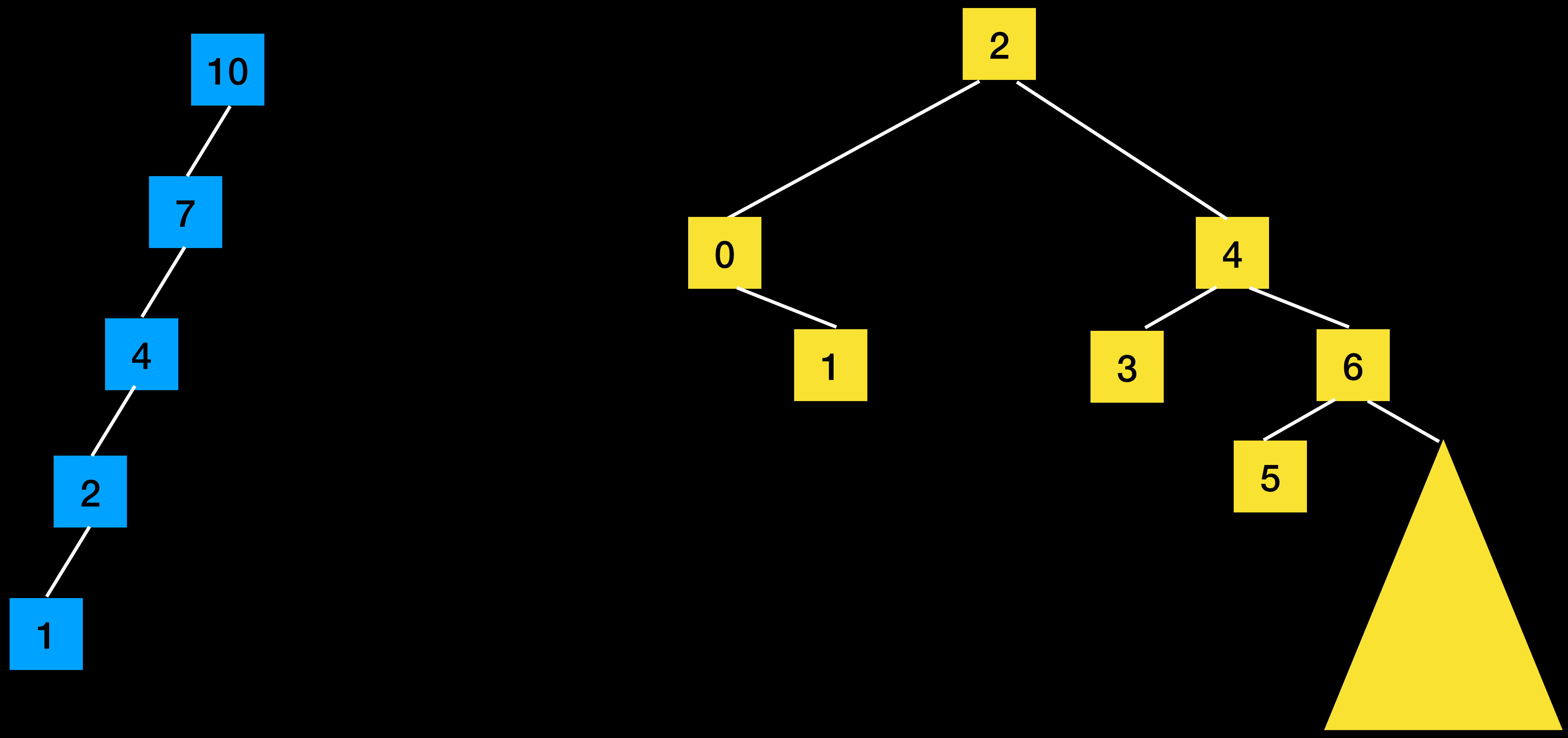


You **Grow** a tree with BST property, you **don't get to restructure it**
(Self-balancing trees (e.g.Red-Black trees) will do that, perhaps in CSCI 335)

Growing a BST



Growing a BST



Lecture Activity

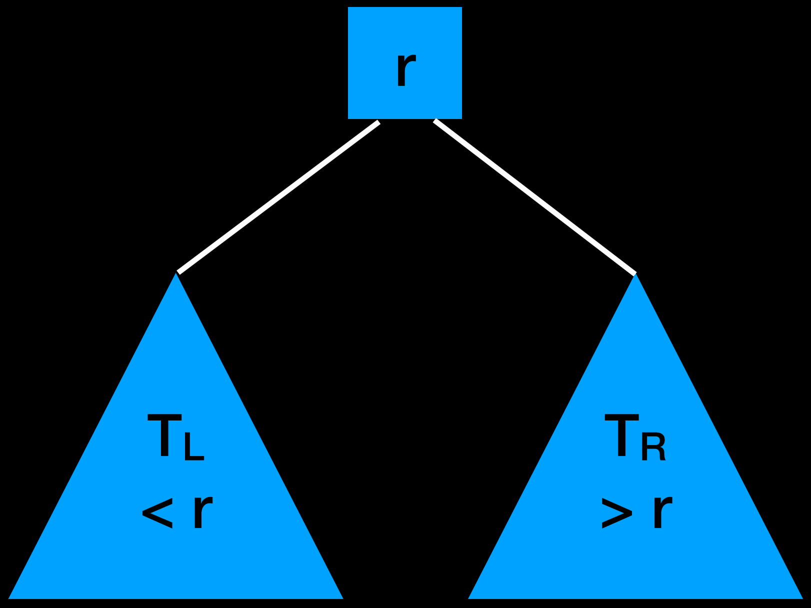
Write **pseudocode** to insert an item into a BST

Lecture Activity

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

How did you go about it?
What programming construct/approach did you use?

Inserting into a BST

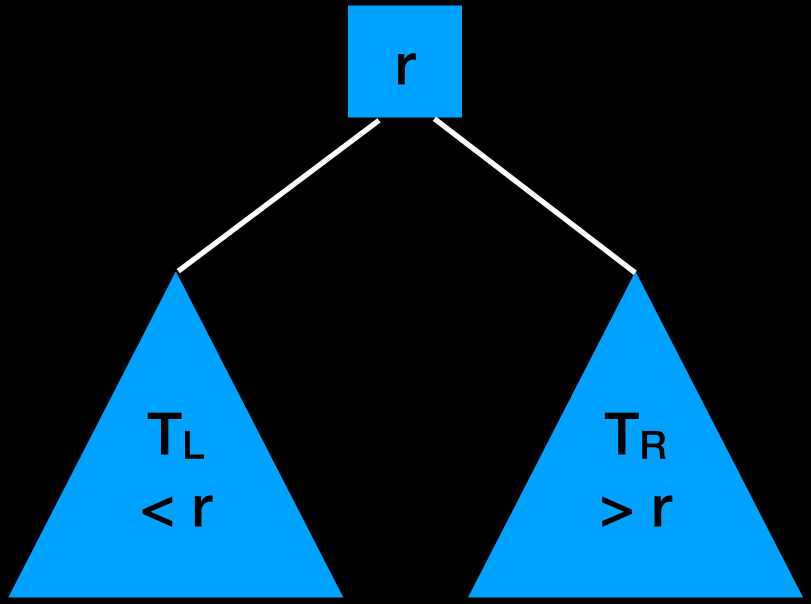



```
add(bs_tree, item)
{
    if (bs_tree is empty) //base case
        make item the root
    else if (item < root)
        add(TL , item)
    else // item > root
        add(TR , item)
}
```

Traversing a BST

Same as traversing any binary tree

Which type of traversal is special for a BST?

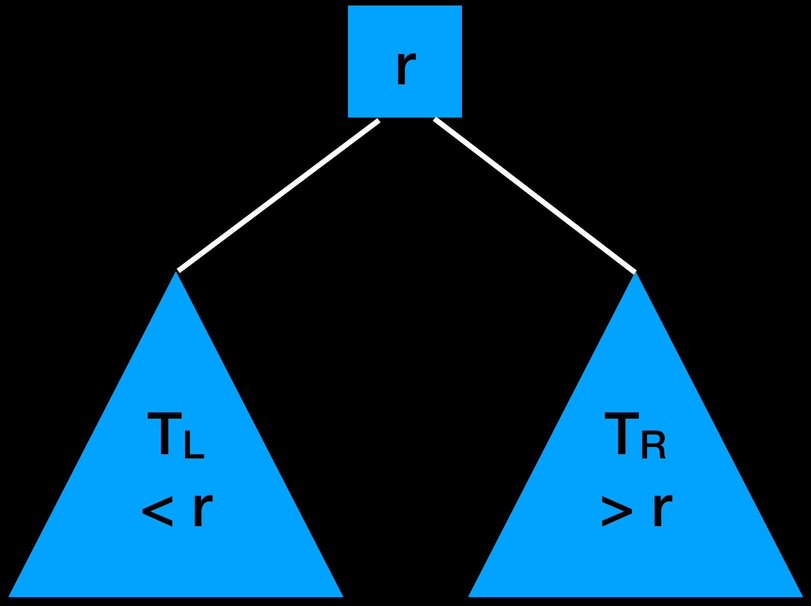


Traversing a BST

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?

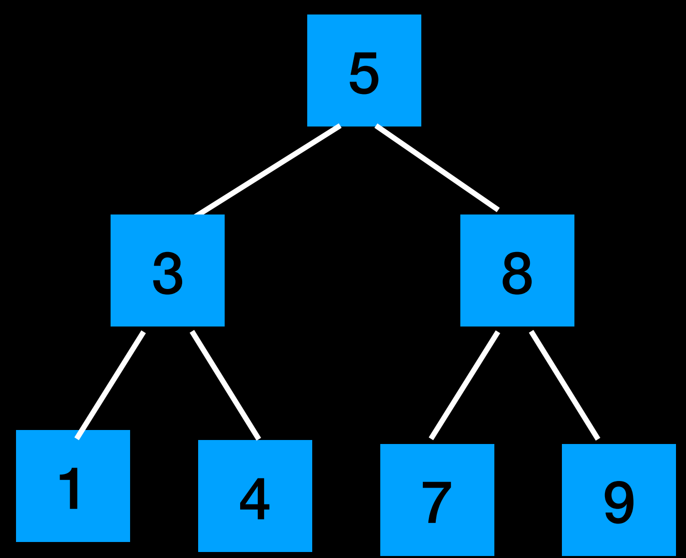
What is the minimum height?

Tree Structure

n = 7

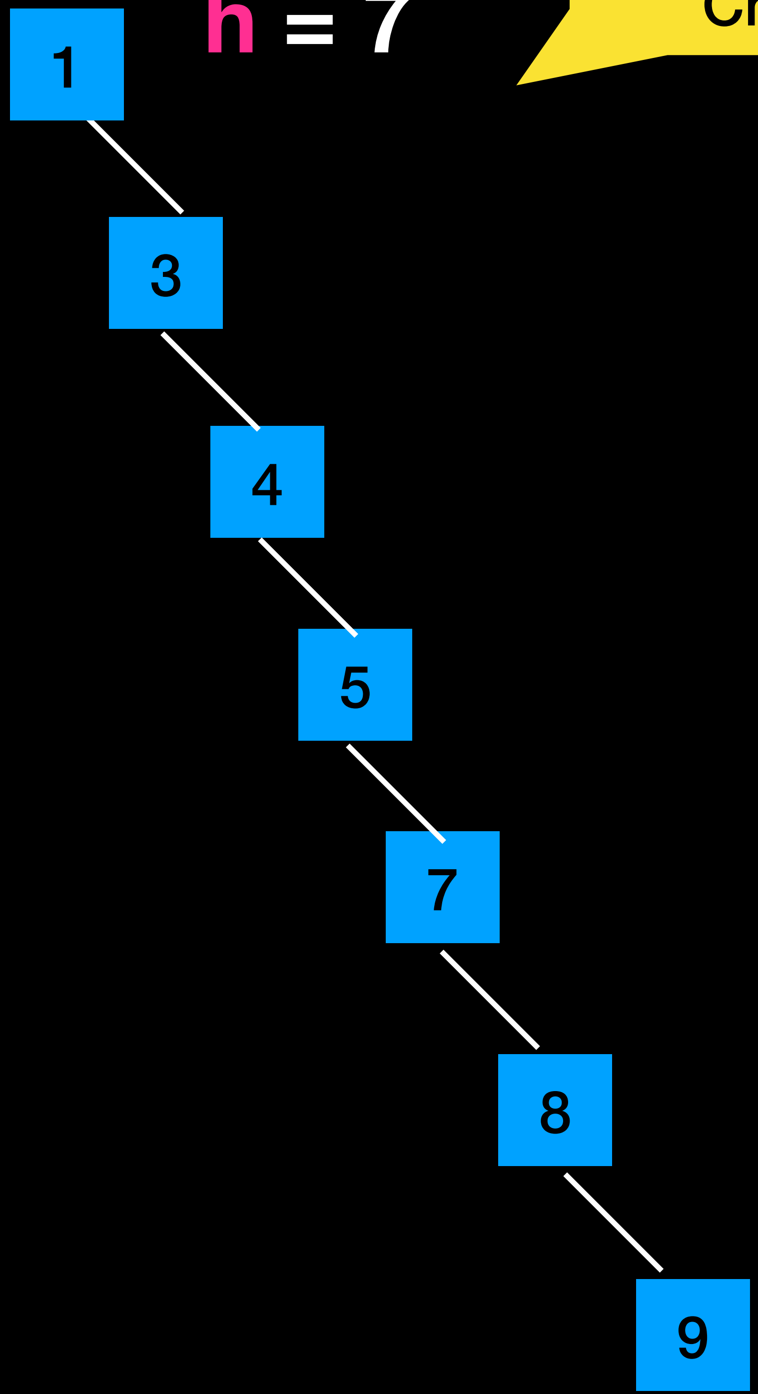
h = 3

Full BST



h = 7

Chain



n nodes

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

Operation	In Full Tree	Worst-case
Search	O(logn)	$O(h)$
Add	O(logn)	$O(h)$
Remove	O(logn)	$O(h)$
Traverse	O(n)	$O(n)$

BST Operations

```cpp
#ifndef BST_H_
#define BST_H_

template<class T>
class BST
{

public:
     BST(); // constructor
     BST(const BST<T>& tree); // copy constructor
     ~ BST(); // 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;

     BST& operator= (const BST<T>& 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<class T>
class BST
{

public:
     BST(); // constructor
     BST(const BST<T>& tree); // copy constructor
     ~ BST(); // 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;

     BST& operator= (const BST<T>& 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