Binary Search Trees

Friday Four Square!
4:15PM, Outside Gates

Implementing **`Set`**

- On Monday and Wednesday, we saw how to implement the **`Map`** and **`Lexicon`**, respectively.
- Let's now turn our attention to the **`Set`**.
- Major operations:
  - Insert
  - Remove
  - Contains

An Inefficient Implementation

- We could implement the `Set` as a list of all the values it contains.
- To add an element:
  - Check if the element already exists.
  - If not, append it.
- To remove an element:
  - Find and remove it from the list.
- To see if an element exists:
  - Search the list for the element.

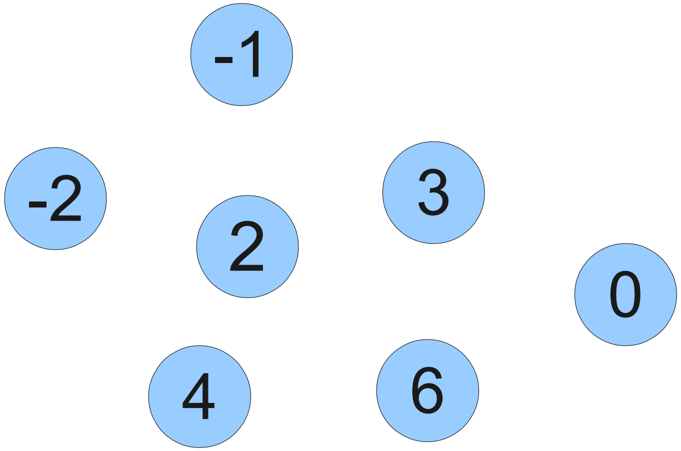
Using Hashing

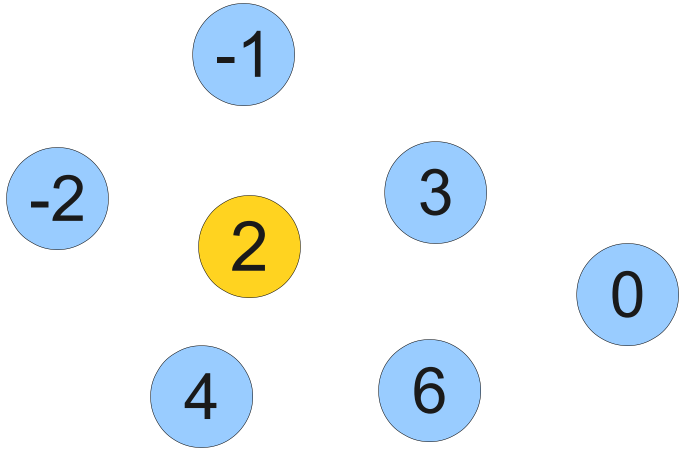
- If we have a hash function for the elements being stored, we can implement a **`Set`** using a hash table.
- What is the expected time to insert a value?
- Answer: **O(1)**.
- What is the expected time to remove a value?
- Answer: **O(1)**.
- What is the expected time to check if a value exists?
- Answer: **O(1)**.
- However, writing a good hash function for a set of elements can be very tricky!

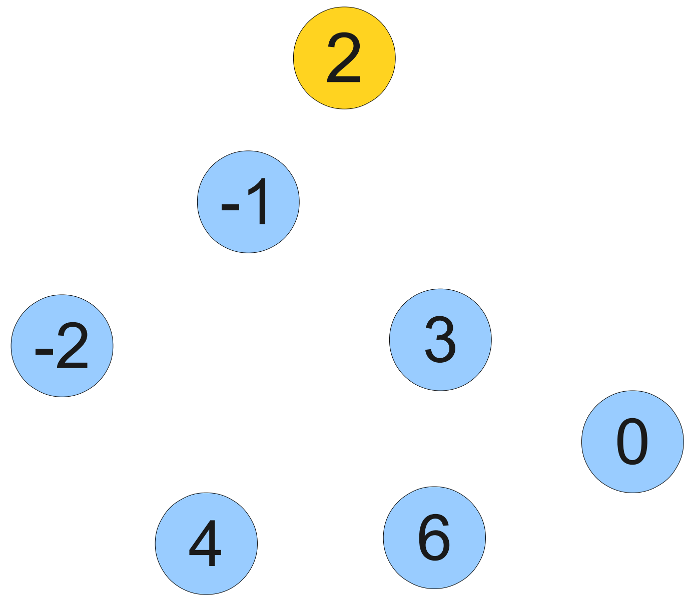
Using Tries

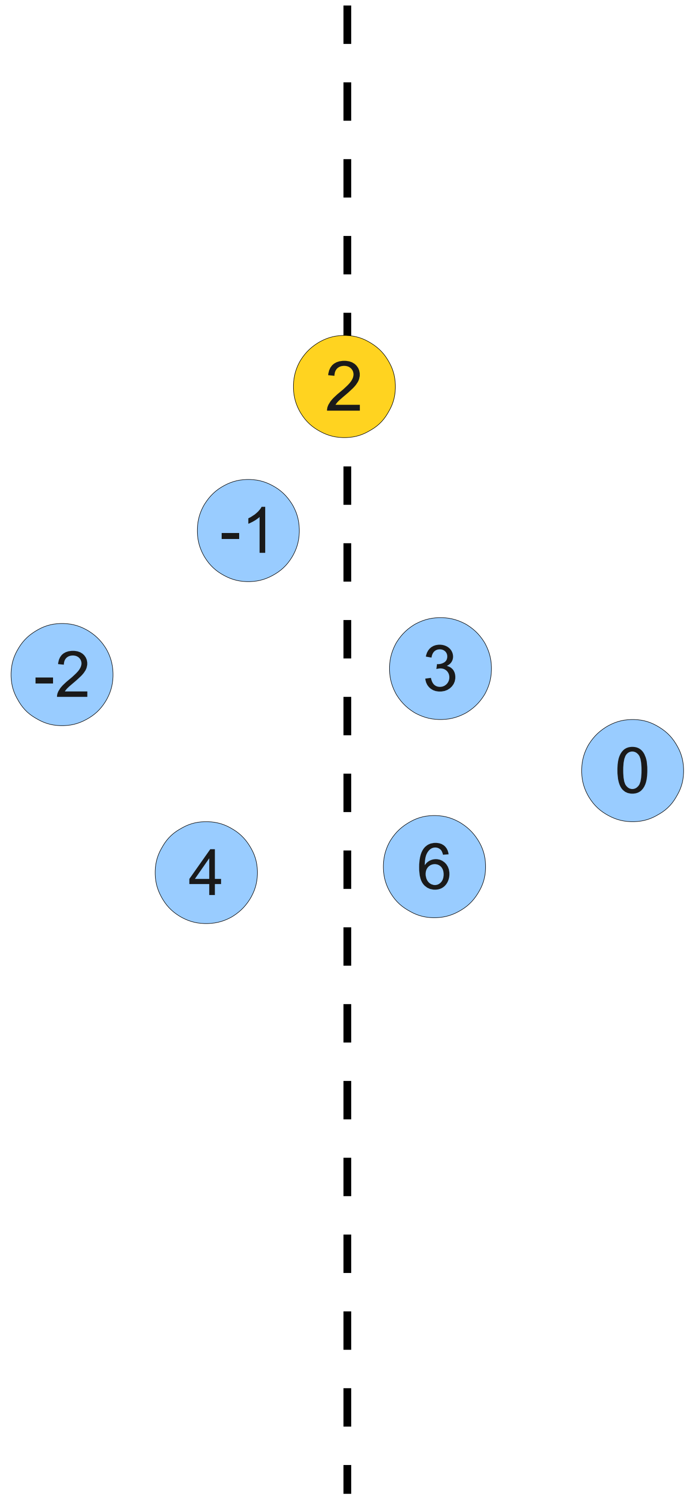
- If our keys are strings, we can store the set using a trie.
- Looking up or inserting a string with L letters takes time $O(L)$.
- Doesn't work for arbitrary values.

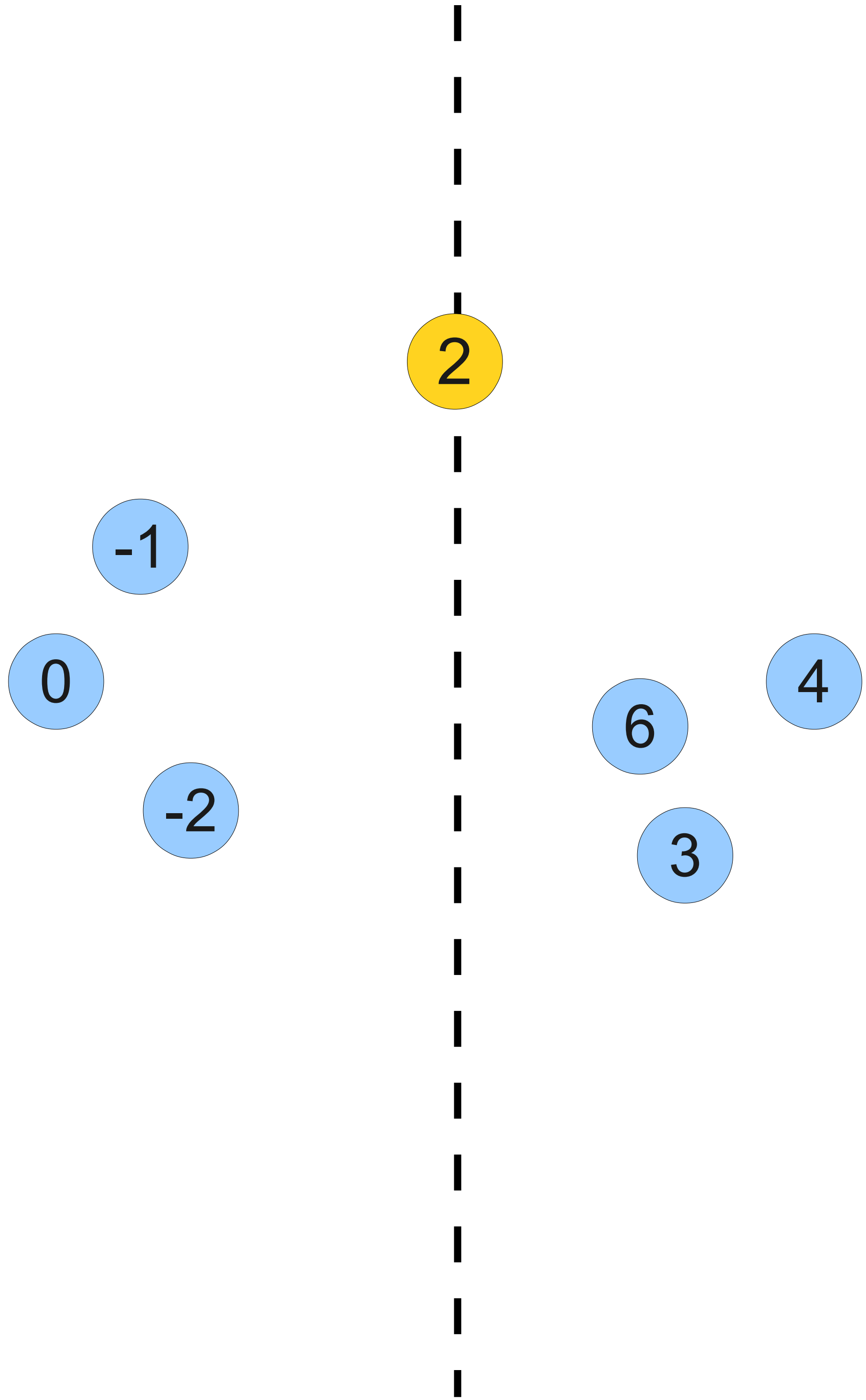
An Entirely Different Approach

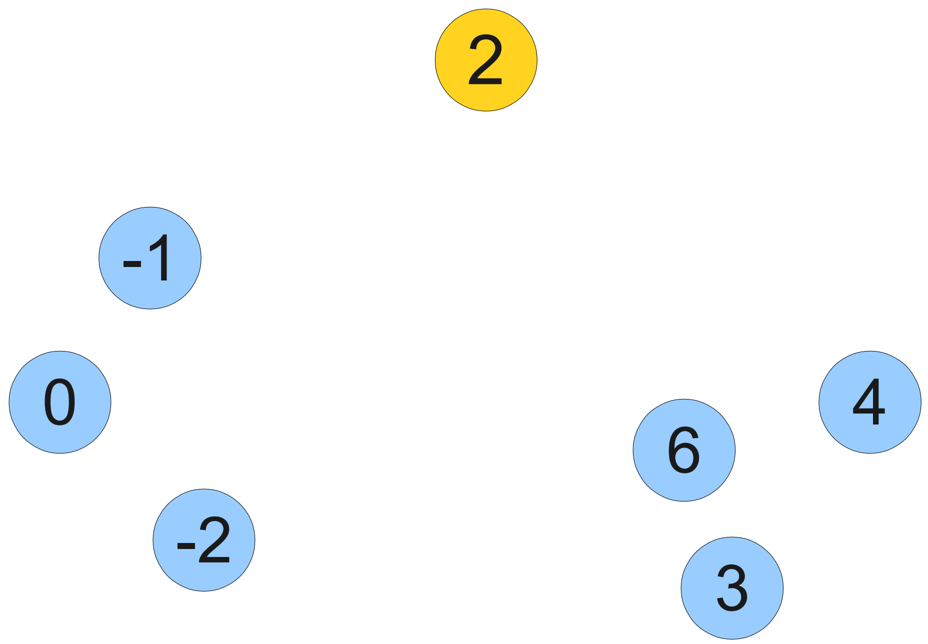

-1
-2
3
2
0
4
6

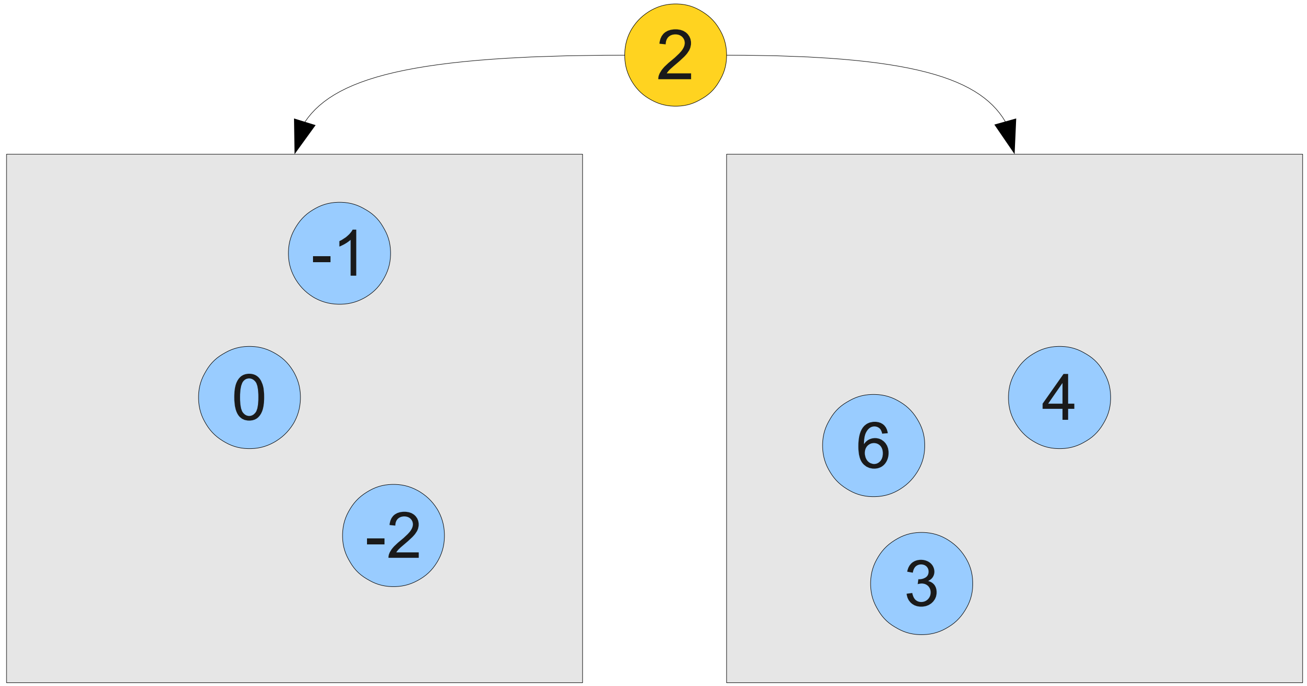

-1
-2
3
2
0
4
6

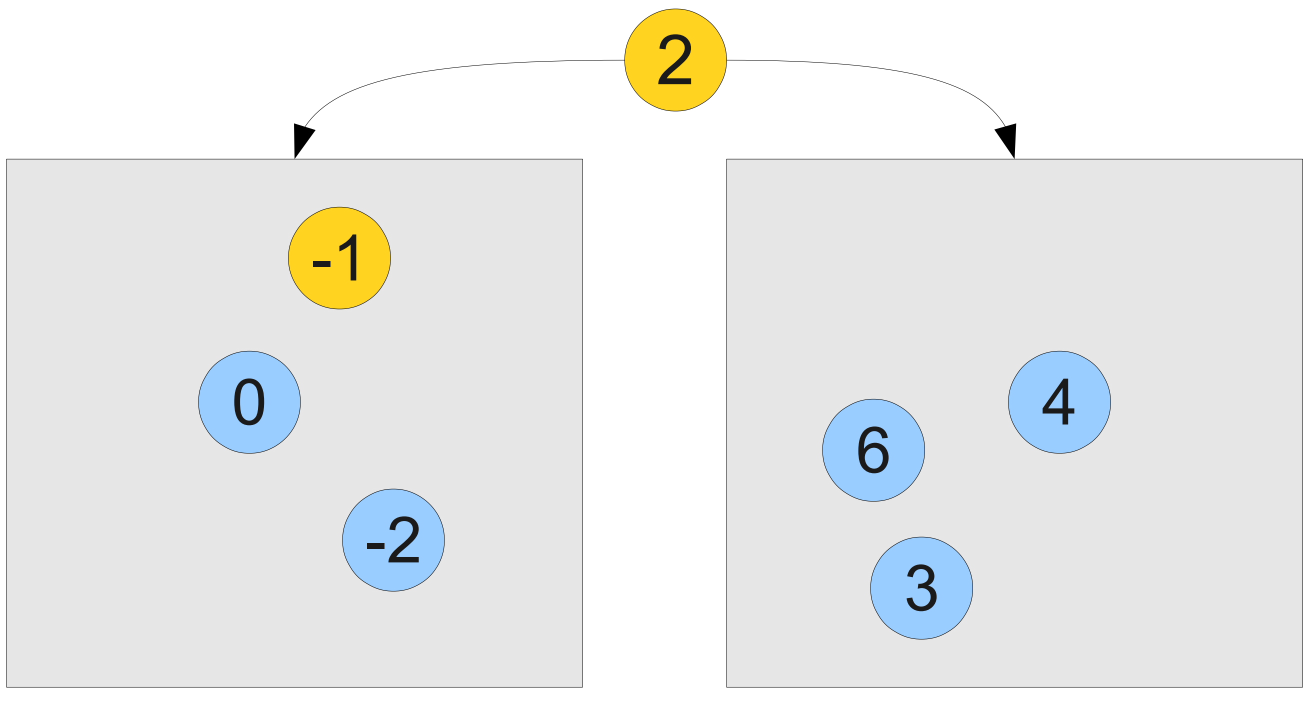

2
-1
-2
3
0
4
6

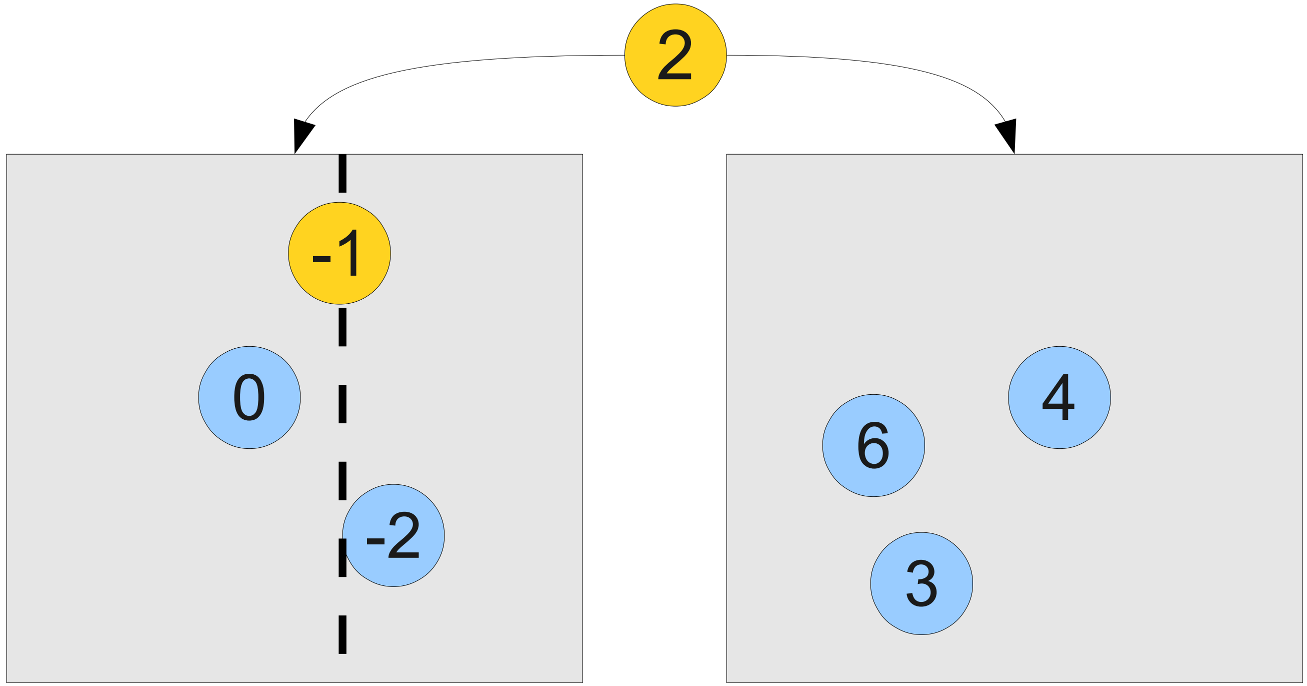

2
-1
-2
3
0
4
6

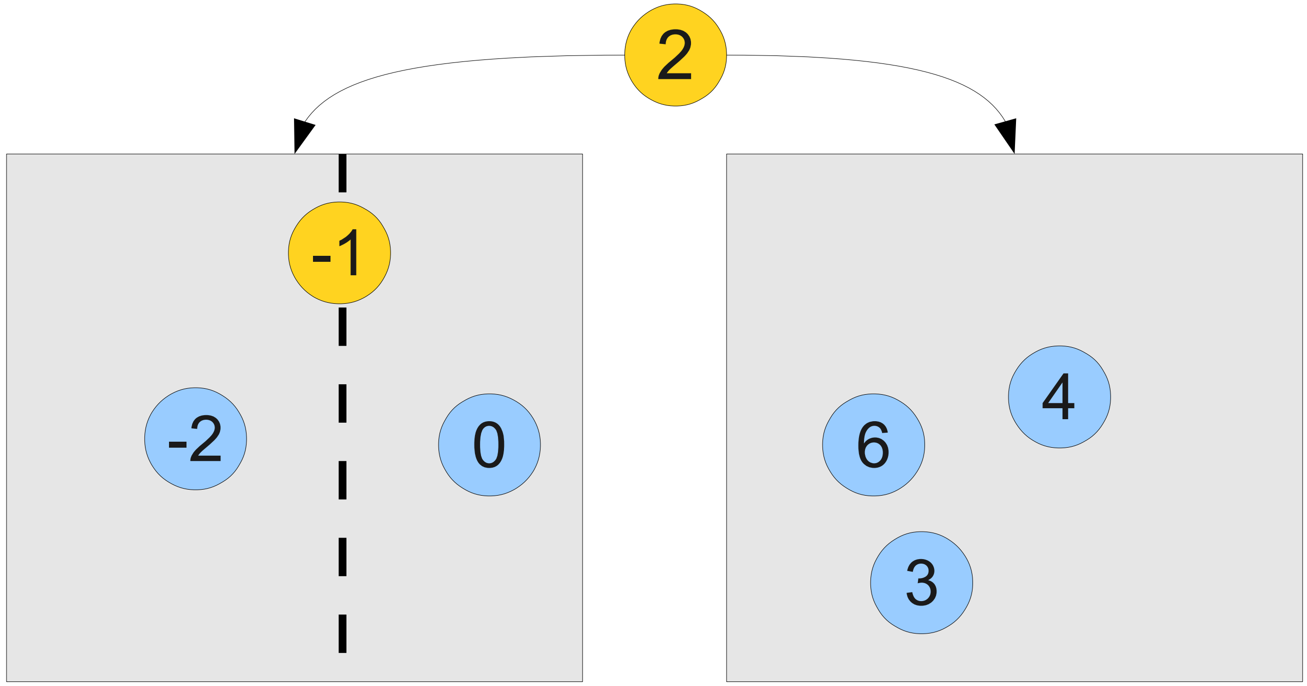

2
-1
0
6
4
-2
3

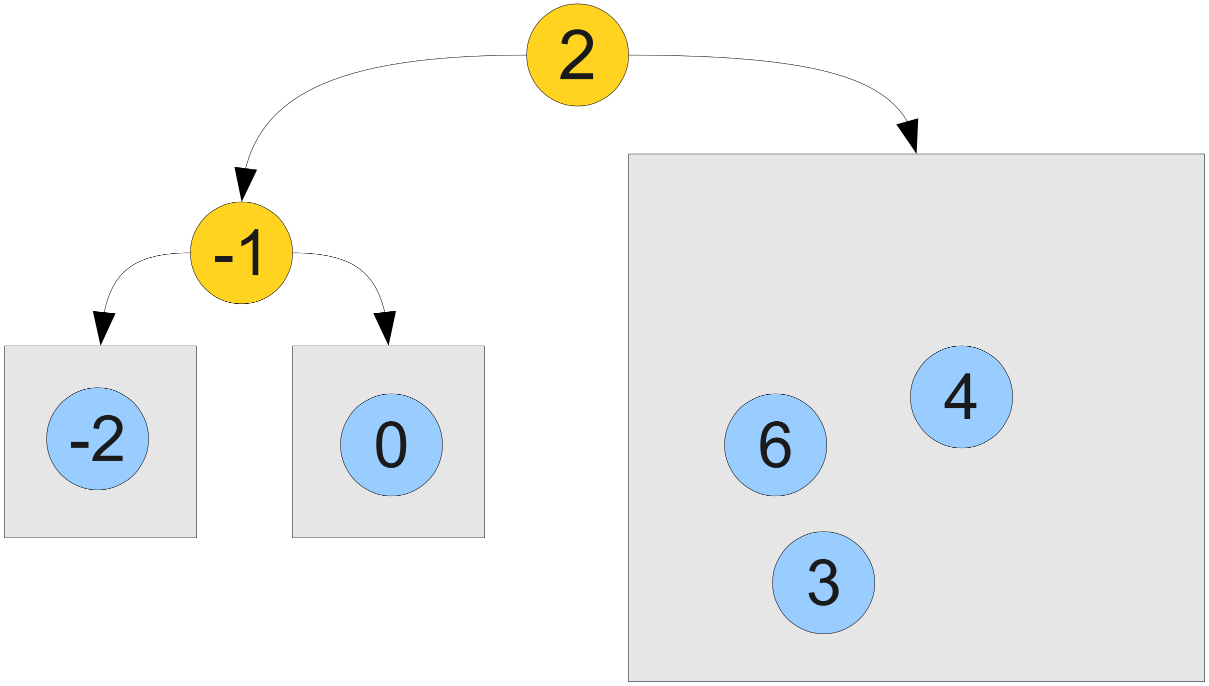

2
-1
0
6
4
-2
3

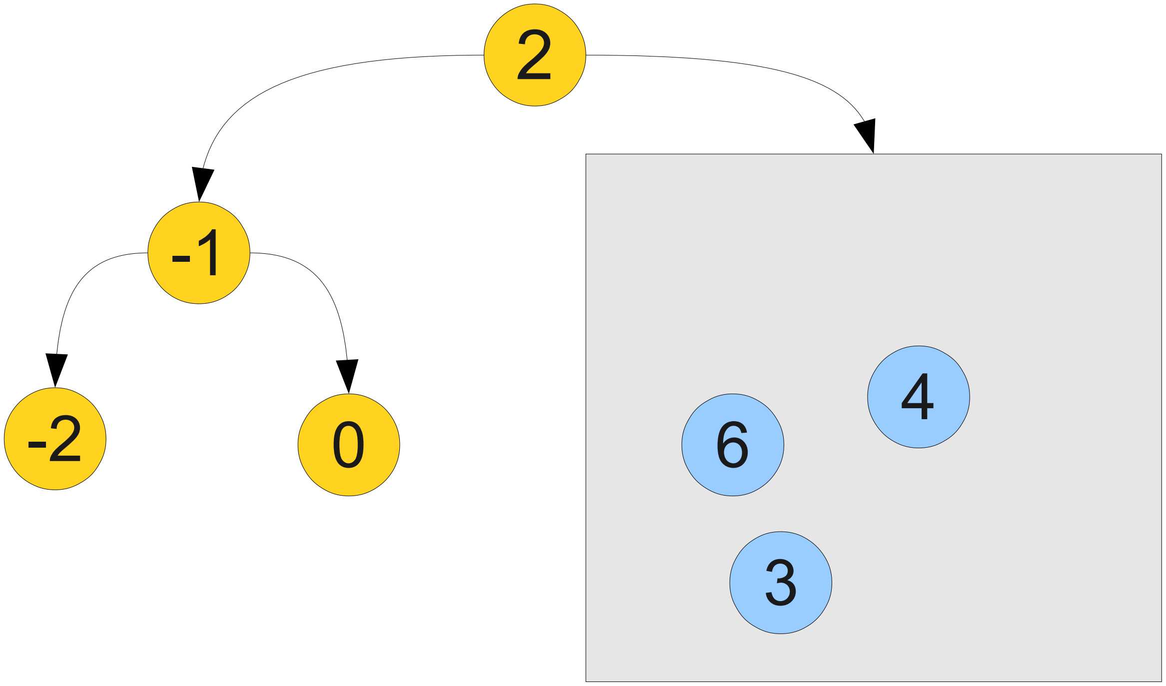

2
-1
0
-2
4
6
3

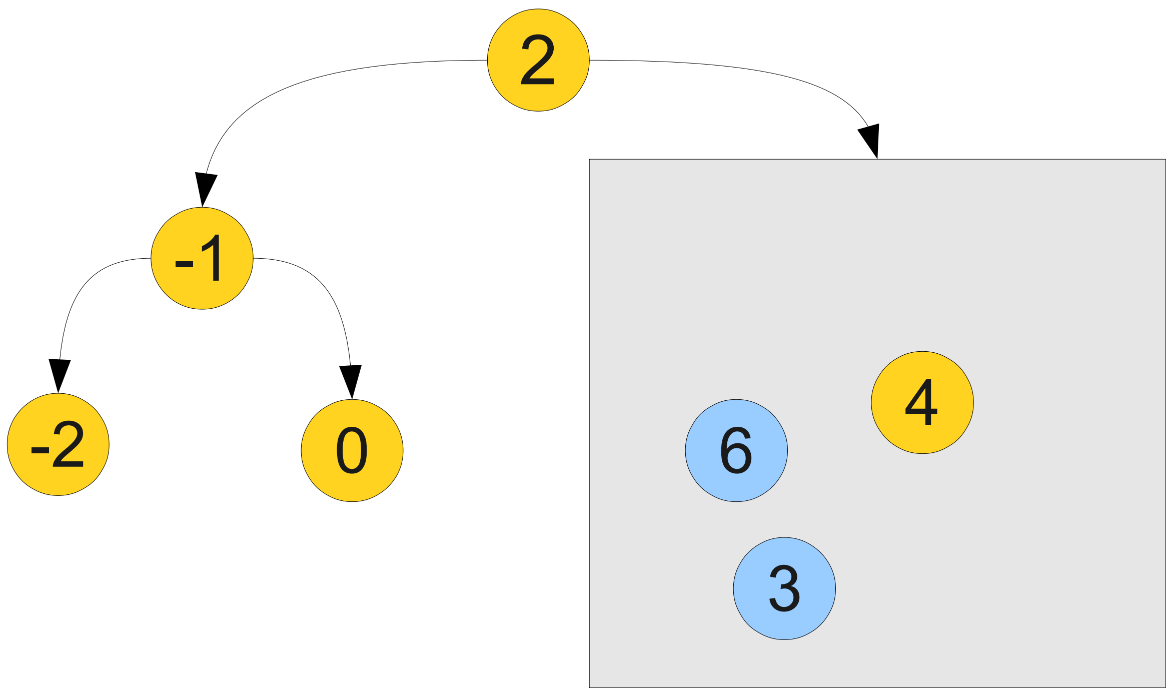

2
-1
0
-2
4
6
3

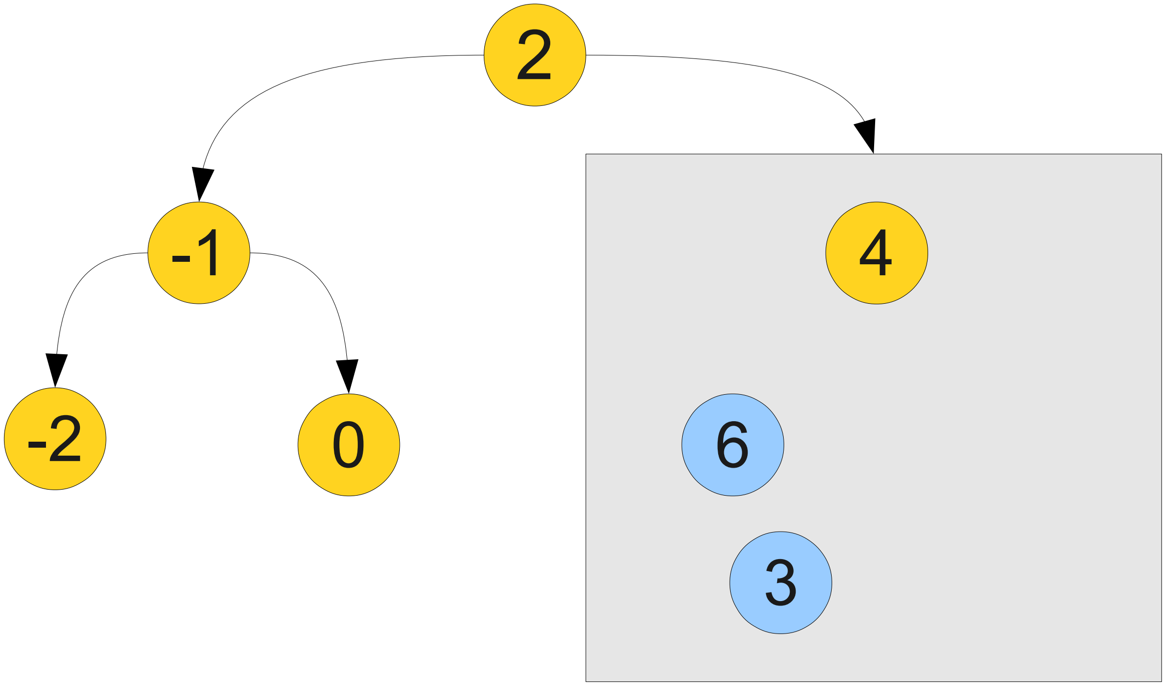

2
-1
0
-2
6
4
3

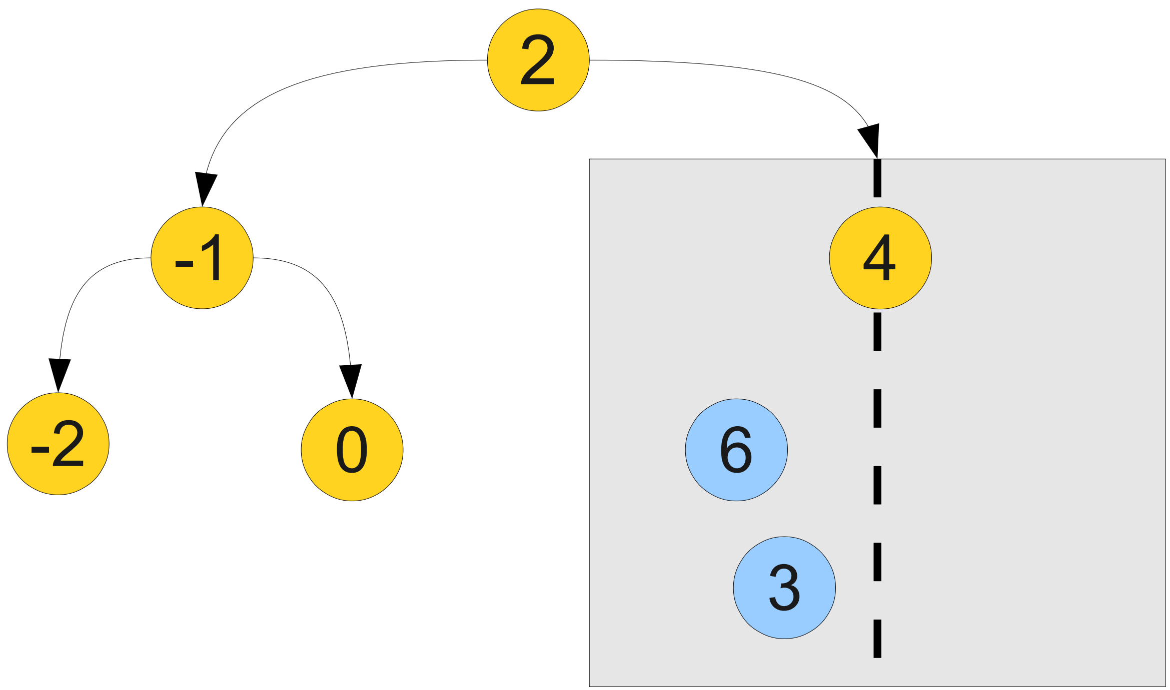

2
-1
-2
0
4
6
3

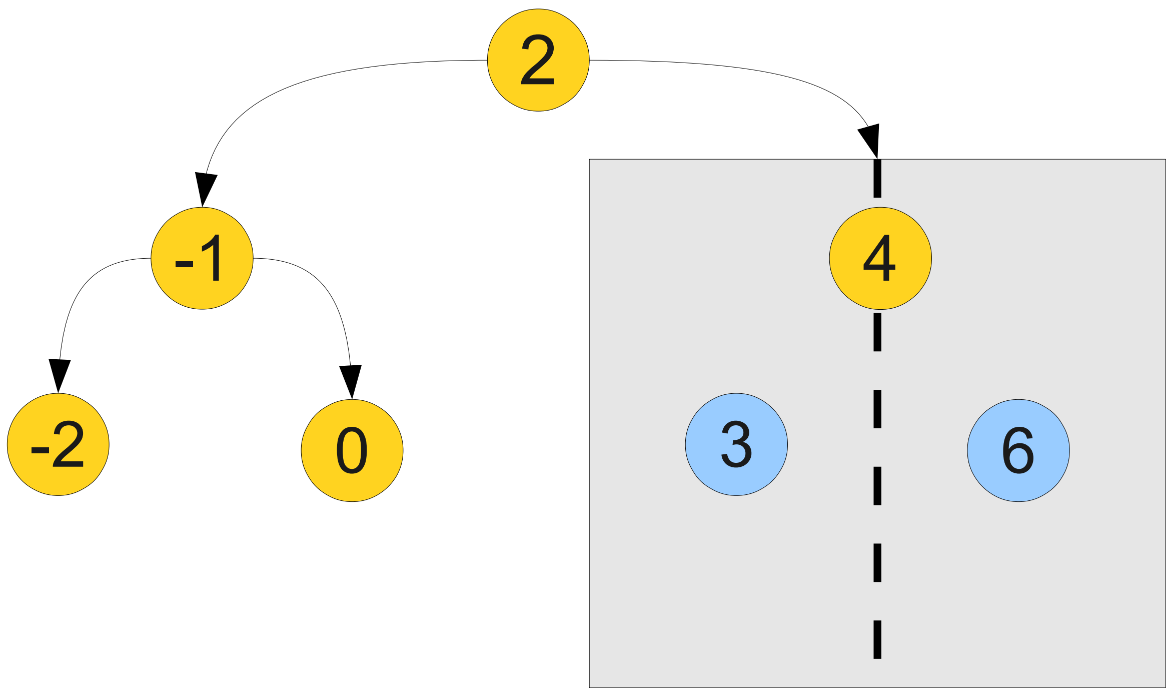

2
-1
-2
0
4
6
3

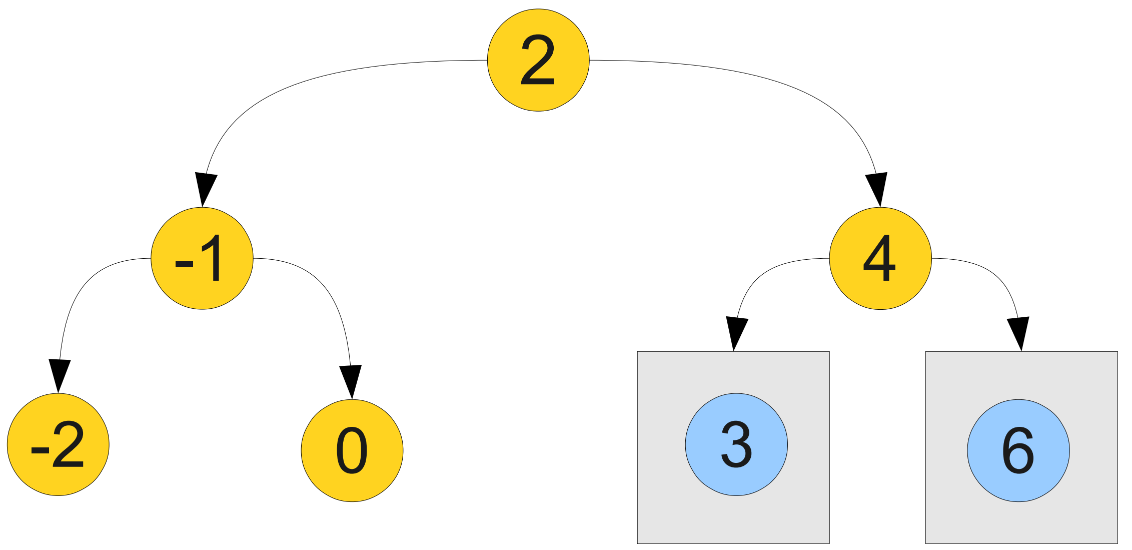

2
-1
-2
0
6
4
3

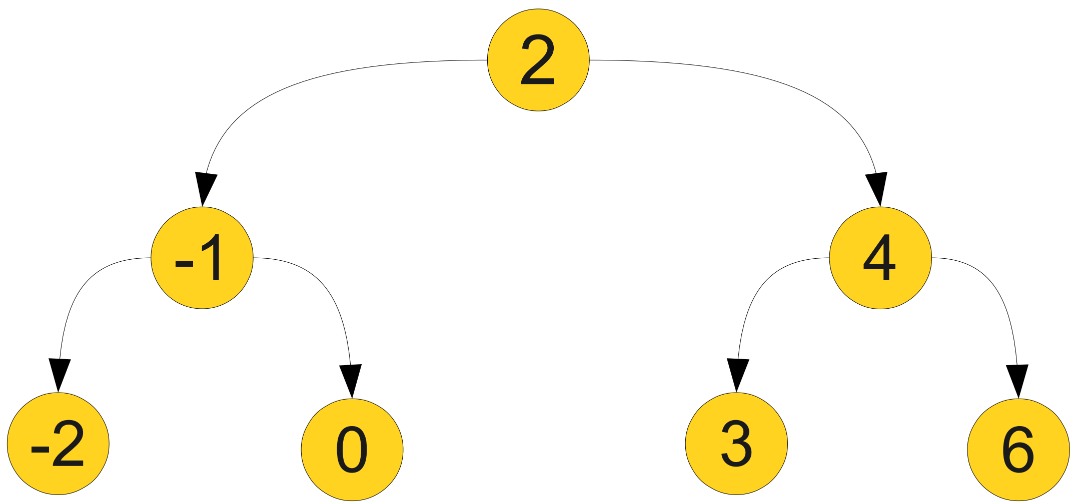

2
-1
-2
0
6
4
3

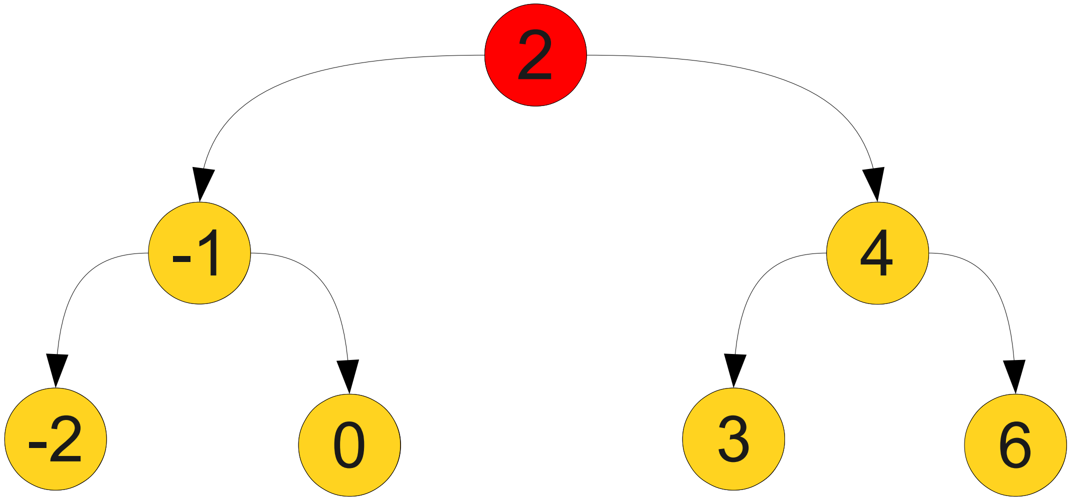

2
-1
4
-2
0
6
3

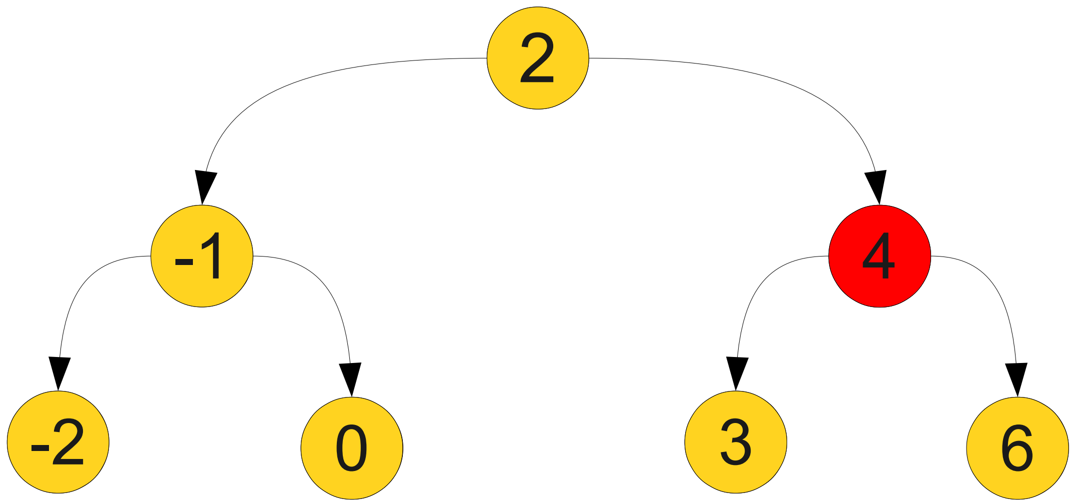

2
-1
4
-2
0
6
3

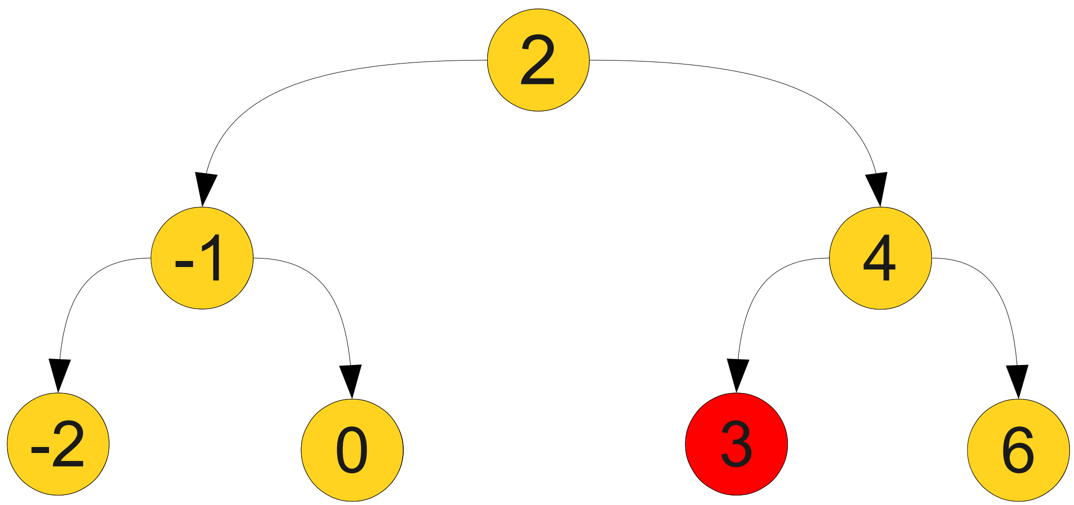

2
-1
4
-2
0
3
6

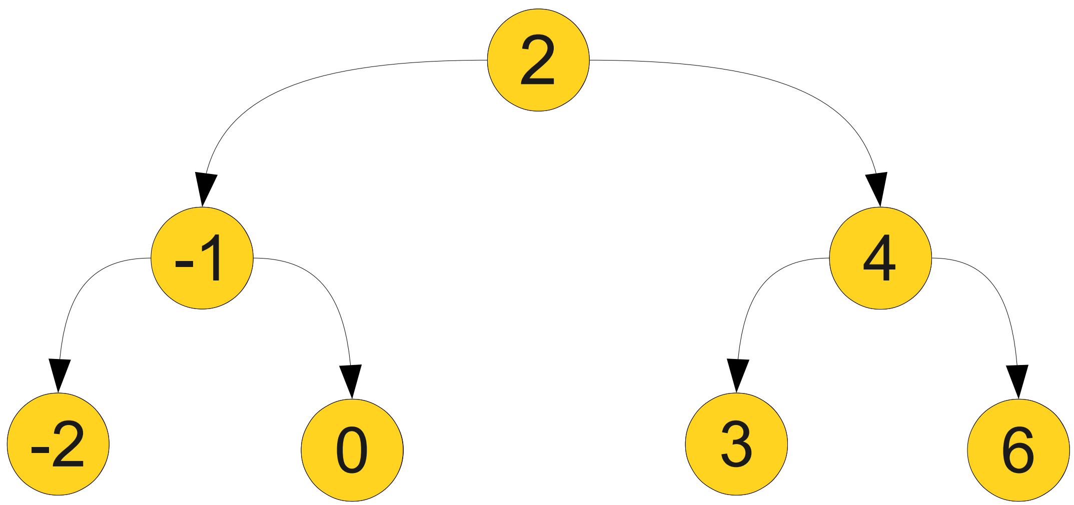

2
-1
4
-2
0
3
6

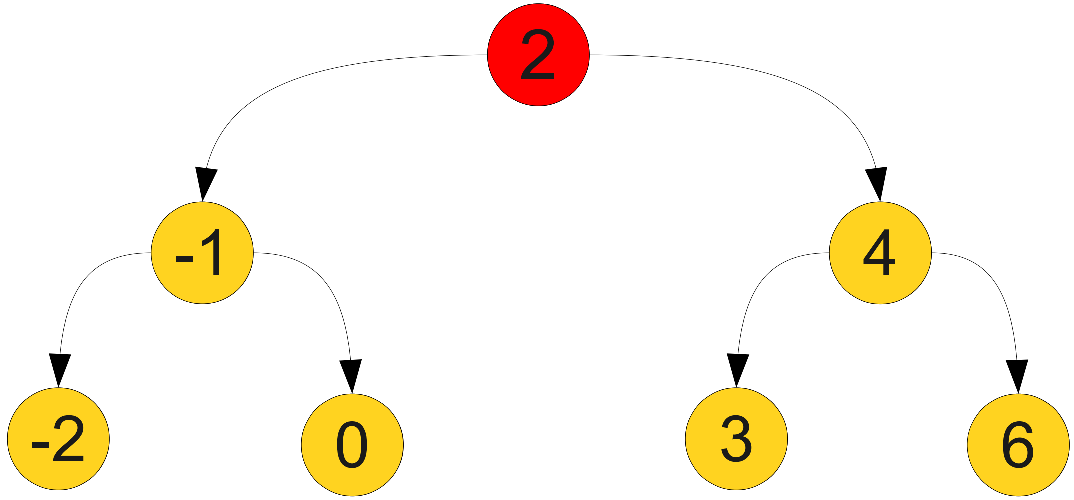

2
-1
4
-2
0
3
6

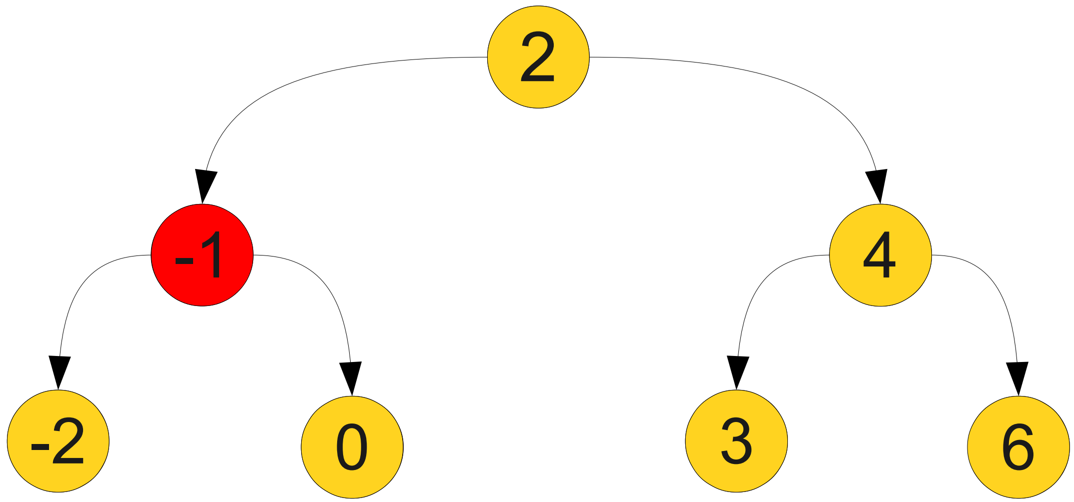

2
-1
4
-2
0
3
6

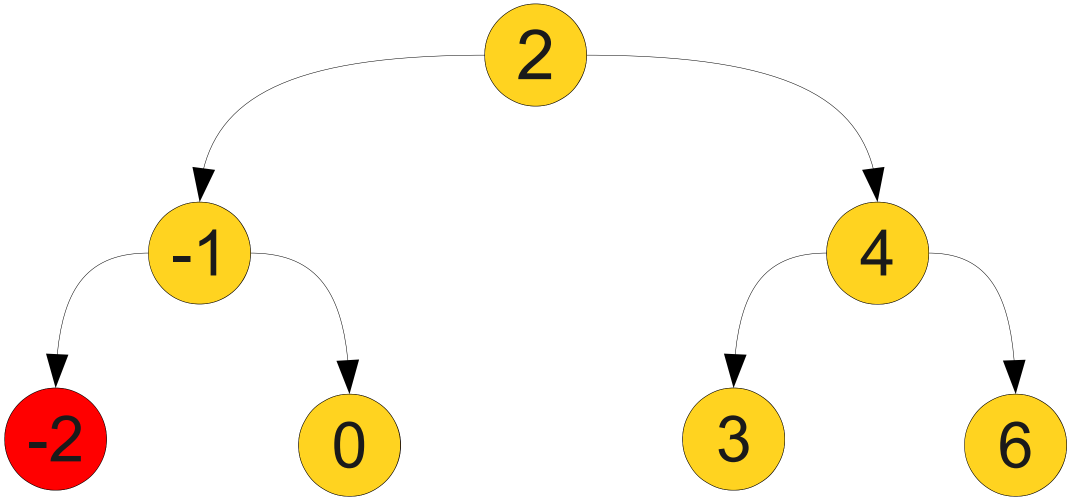

2
-1
4
-2
0
3
6

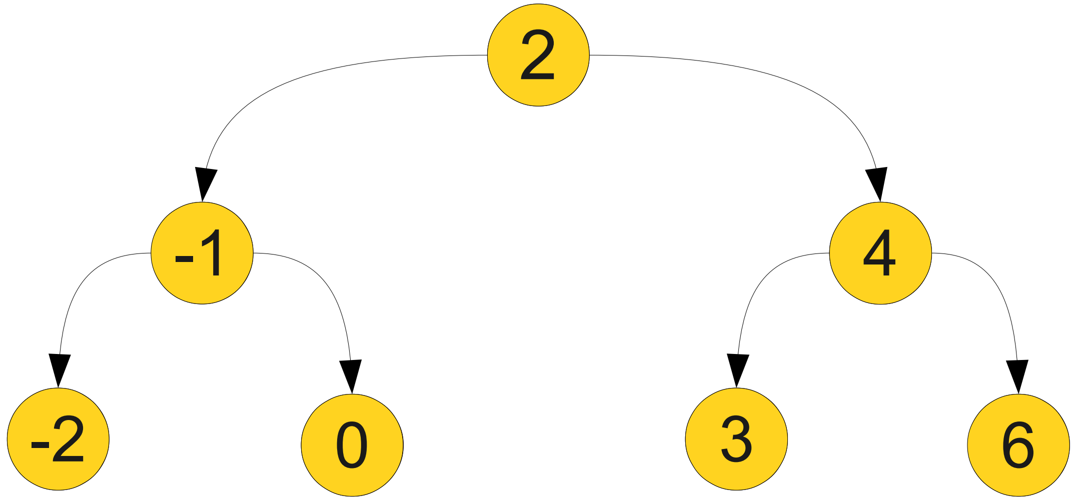

2
-1
4
-2
0
3
6


2
-1
4
-2
0
3
6


2
-1
4
-2
0
3
6


2
-1
4
-2
0
3
6


2
-1
4
-2
0
3
6

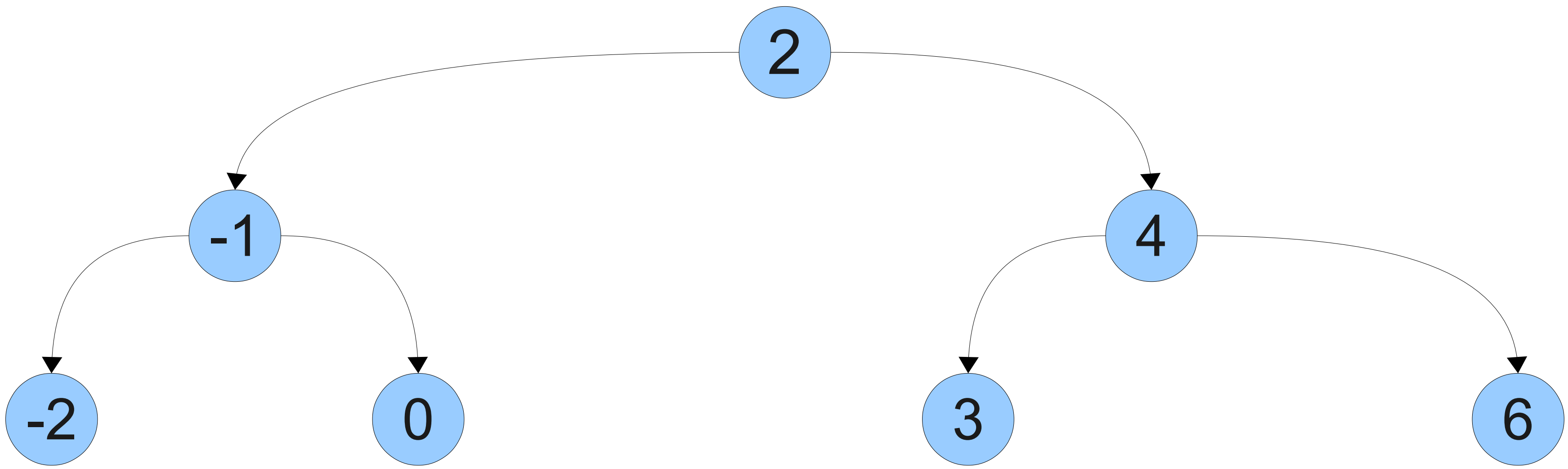

2
-1
4
-2
0
3
6

Binary Search Trees

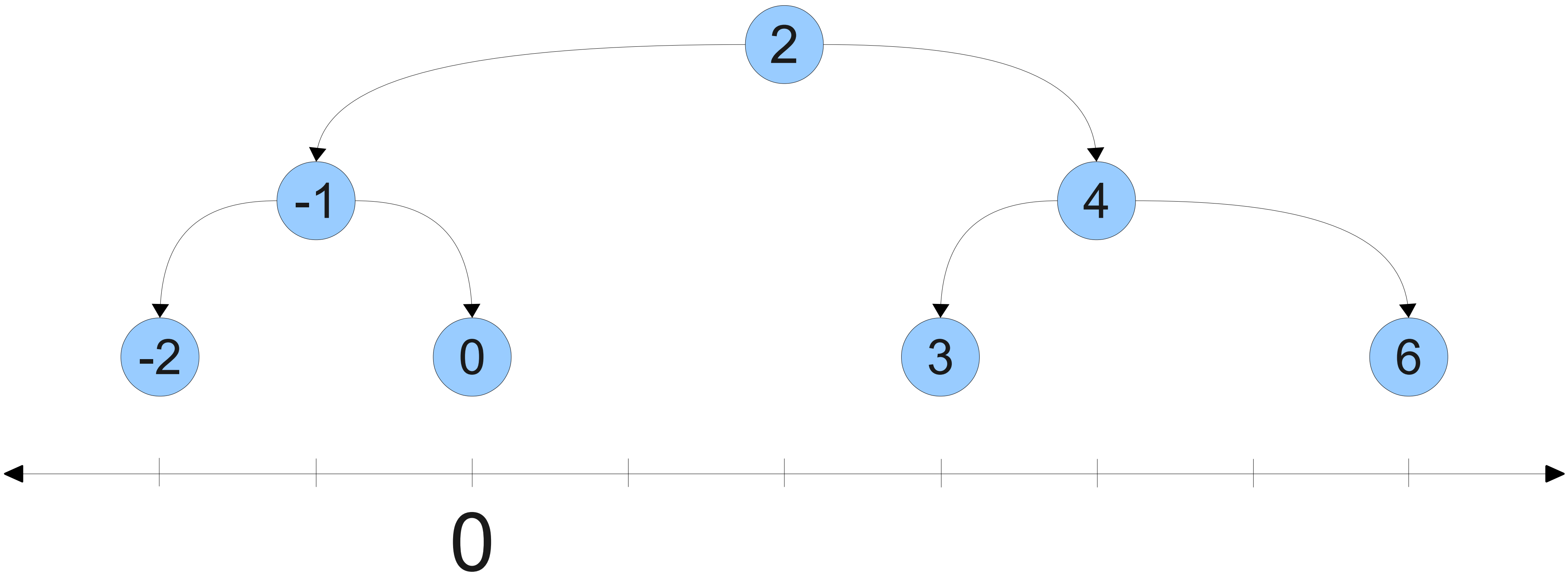
- The data structure we have just seen is called a **binary search tree** (or **BST**).
- Uses comparisons between elements to store elements efficiently.
- No need for a complex hash function, or the ability to work one symbol at a time.

The Intuition

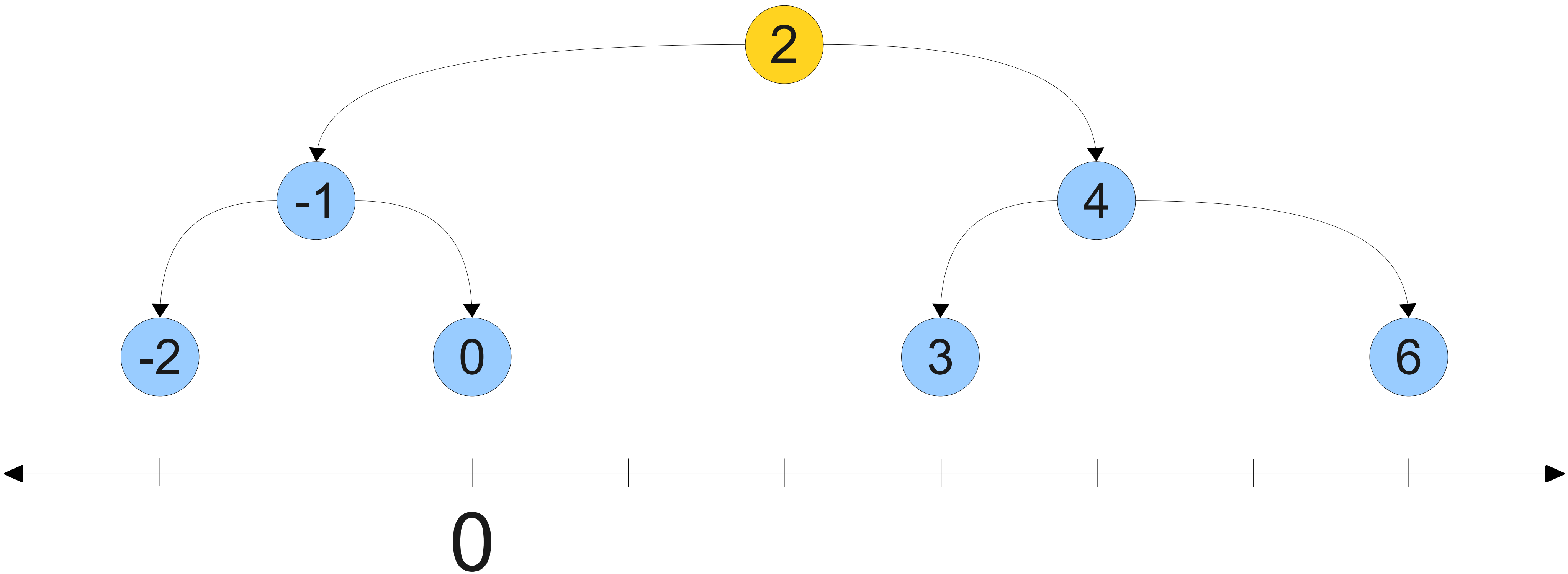
The Intuition



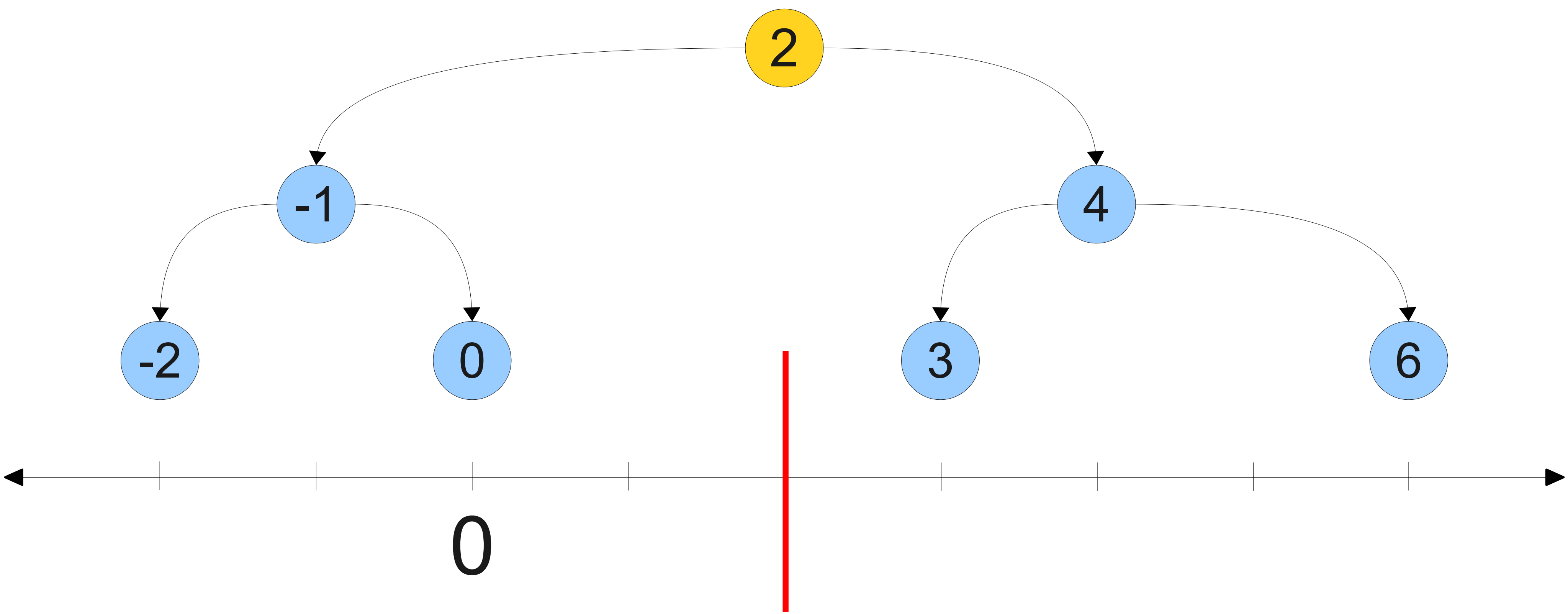
The Intuition



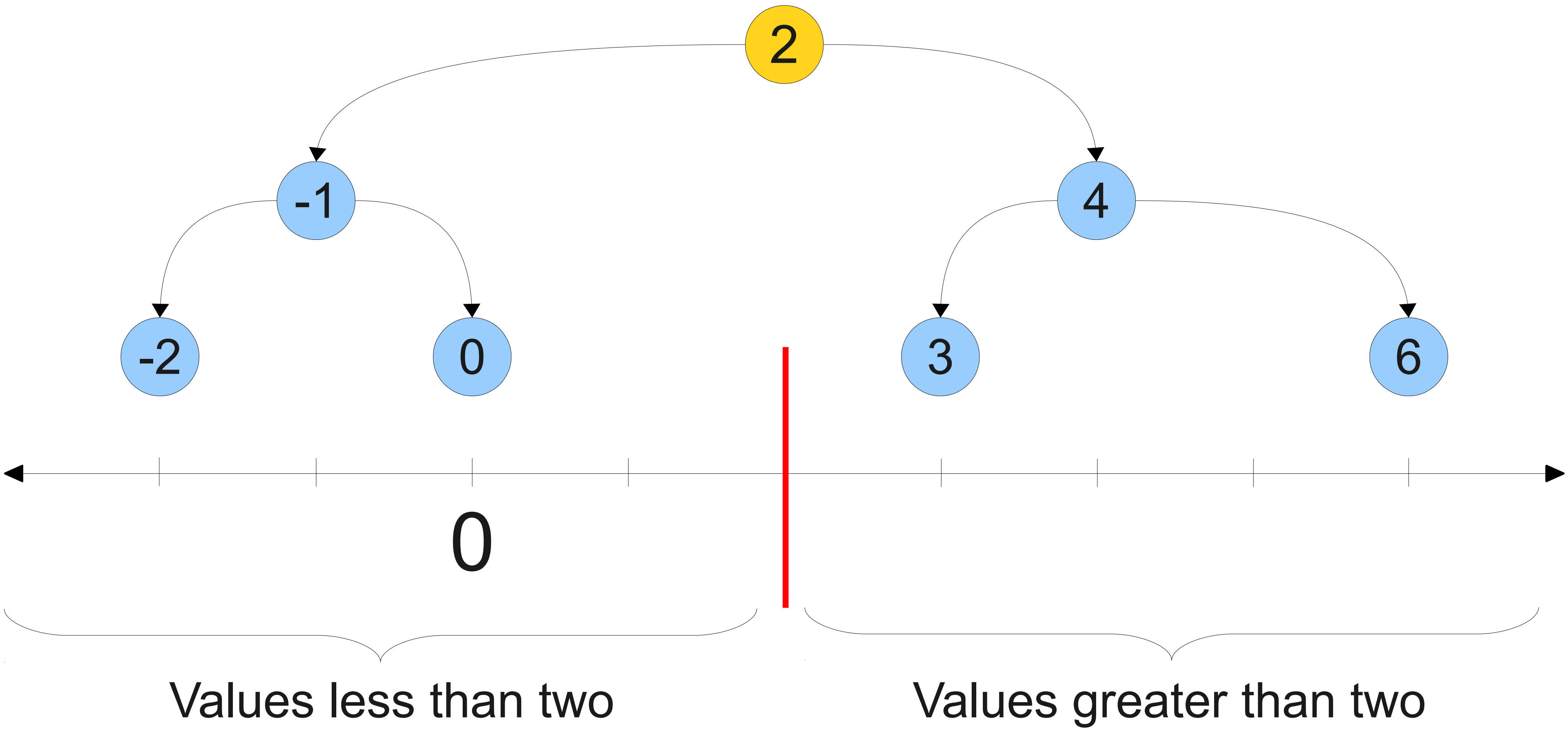
The Intuition



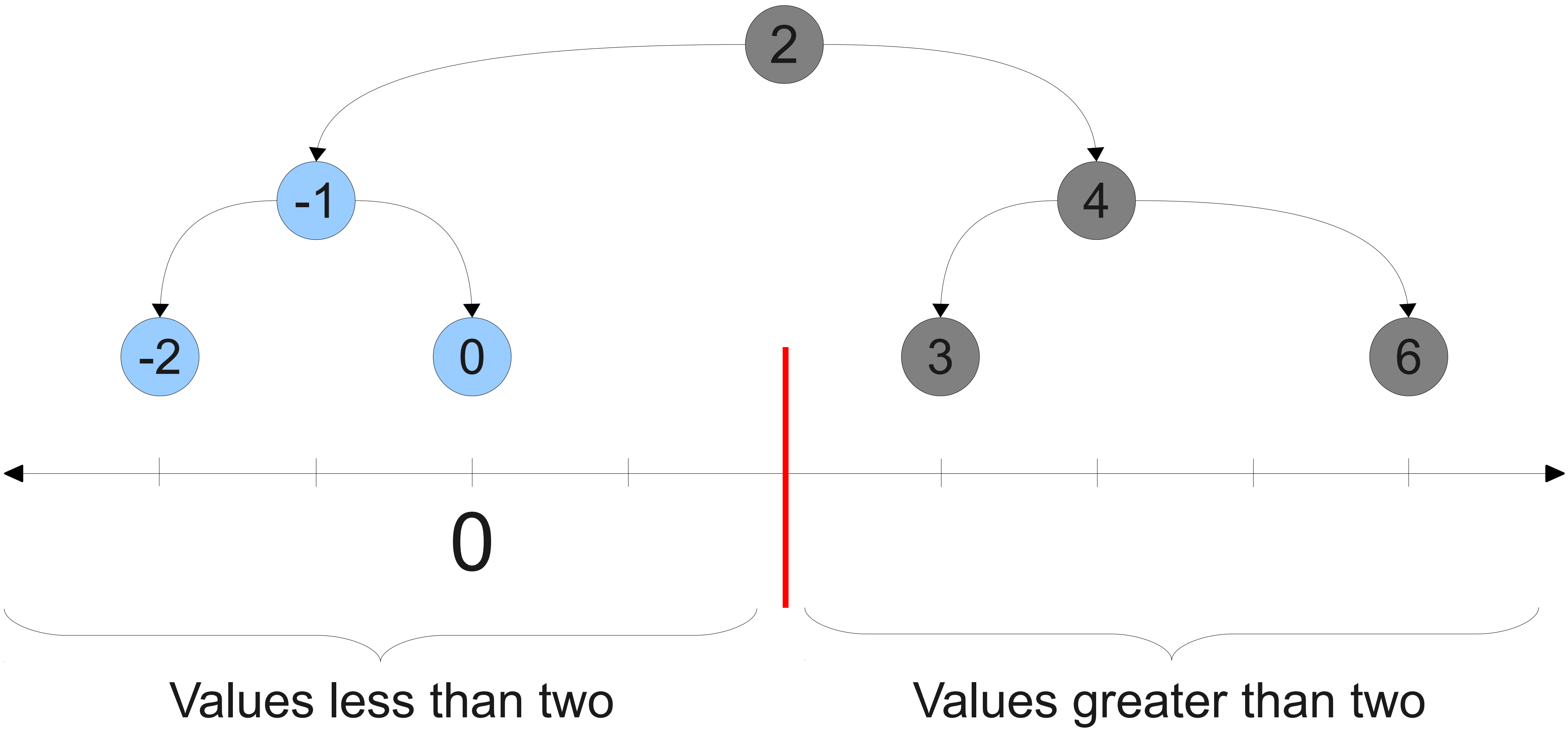
The Intuition



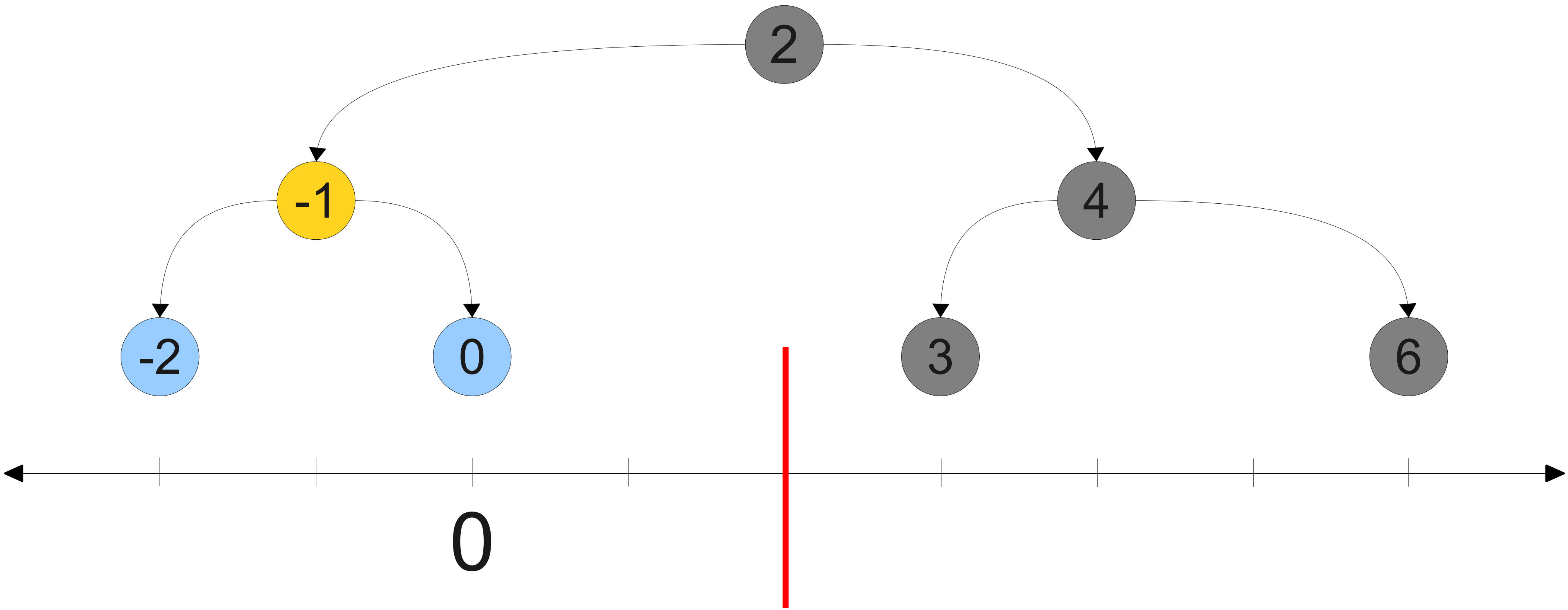
The Intuition



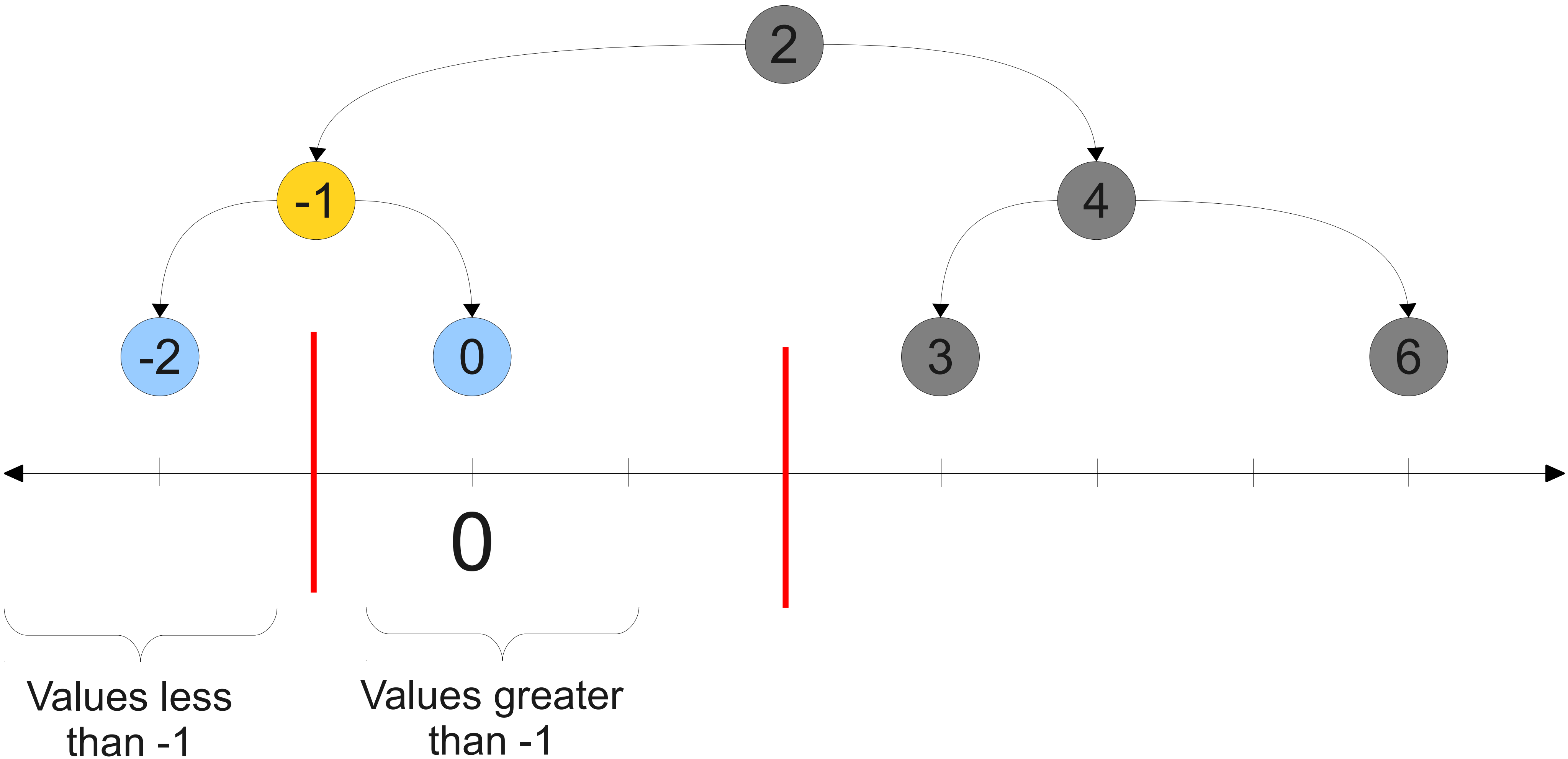
The Intuition



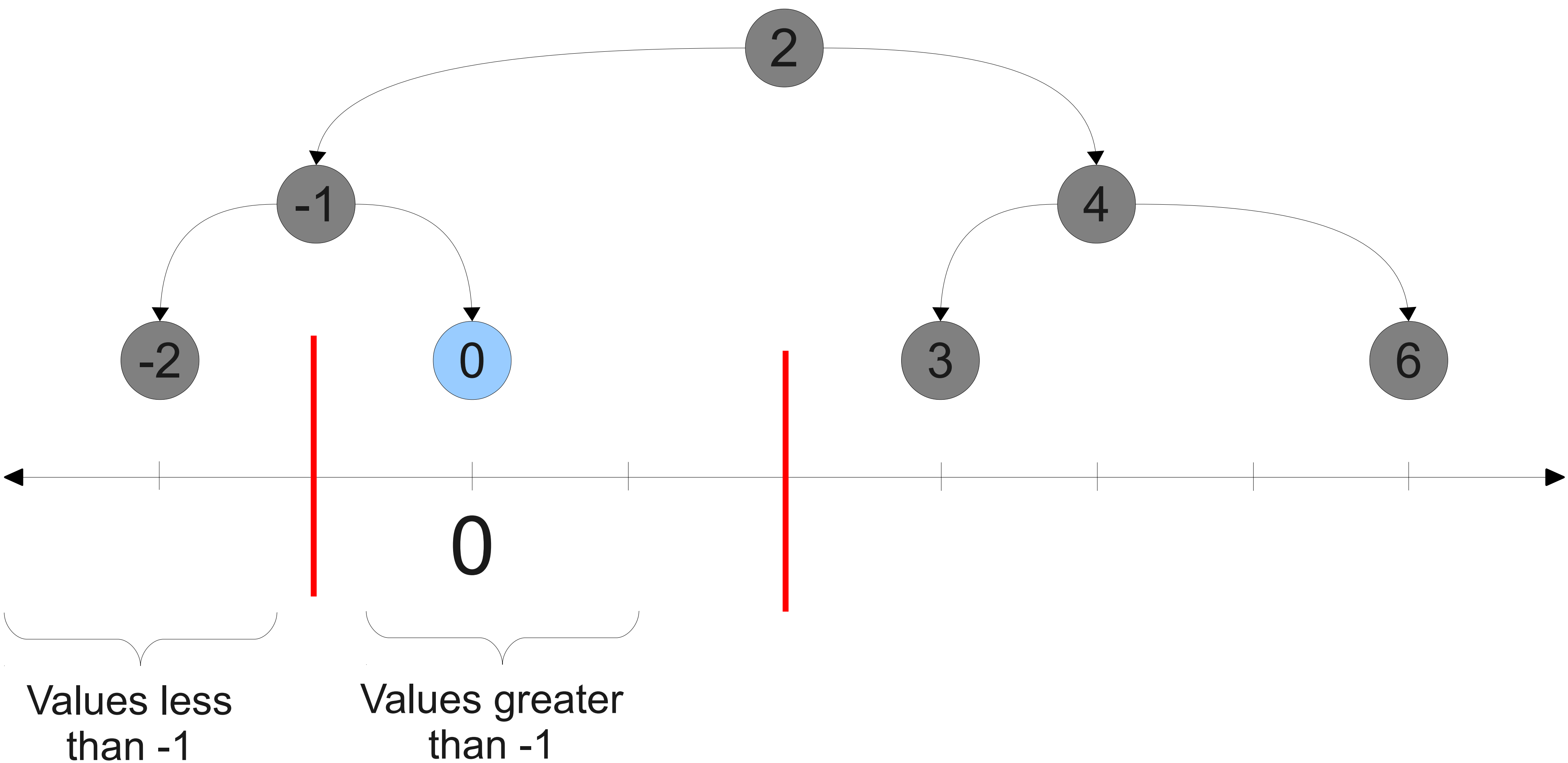
The Intuition



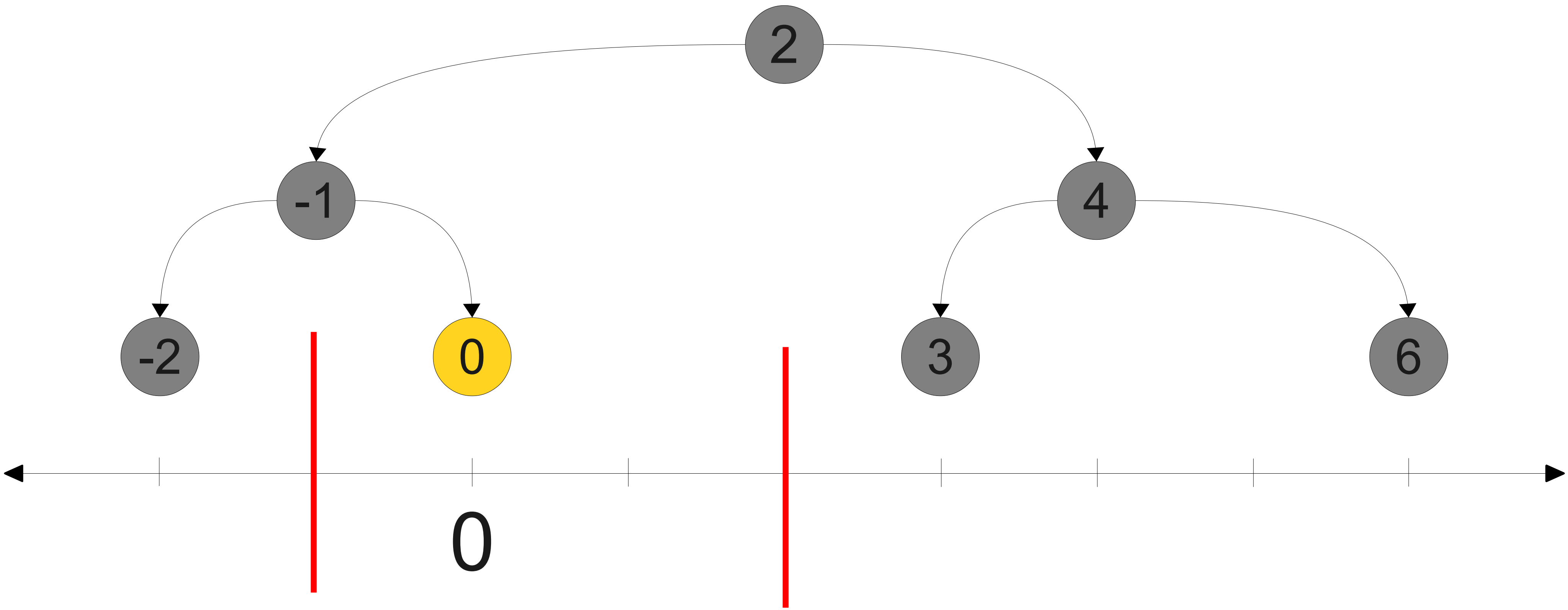
The Intuition



The Intuition

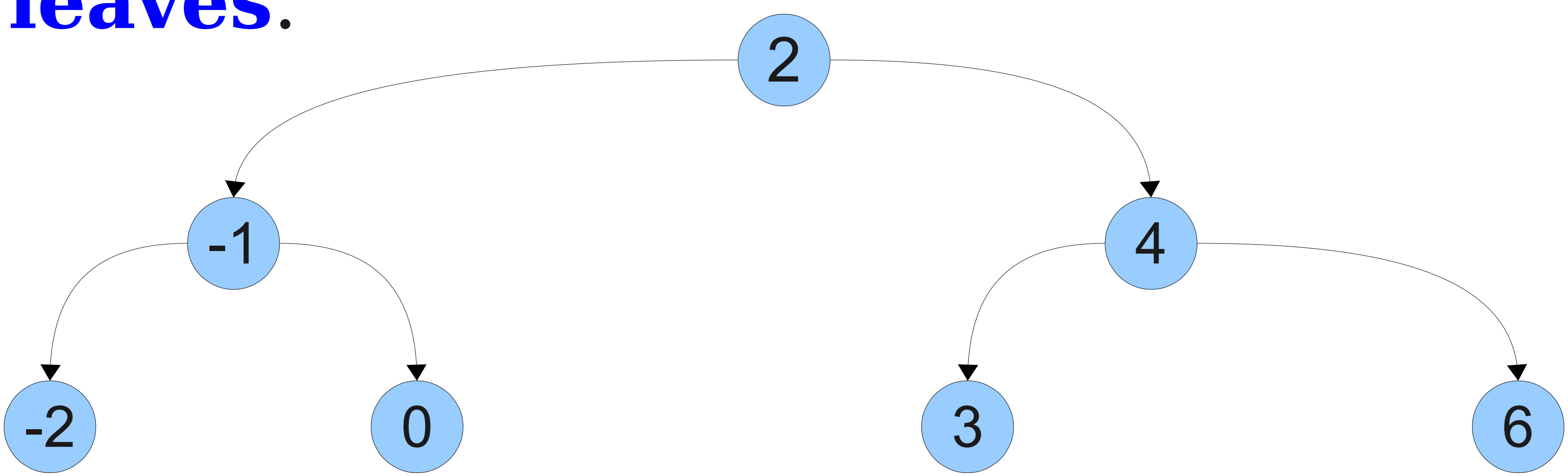


The Intuition

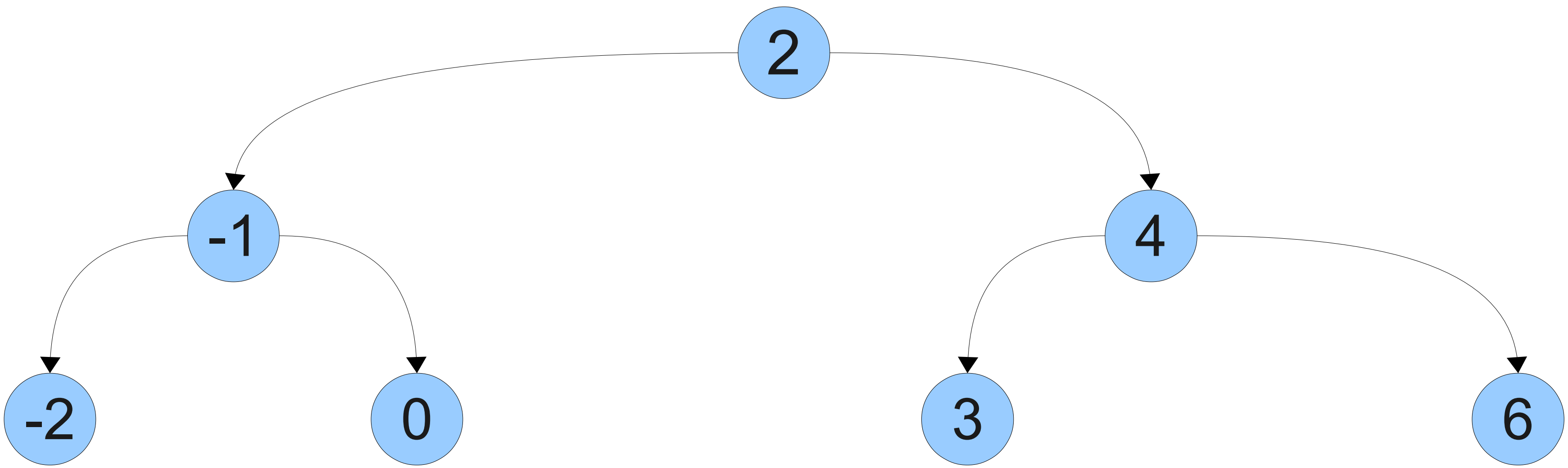


Tree Terminology

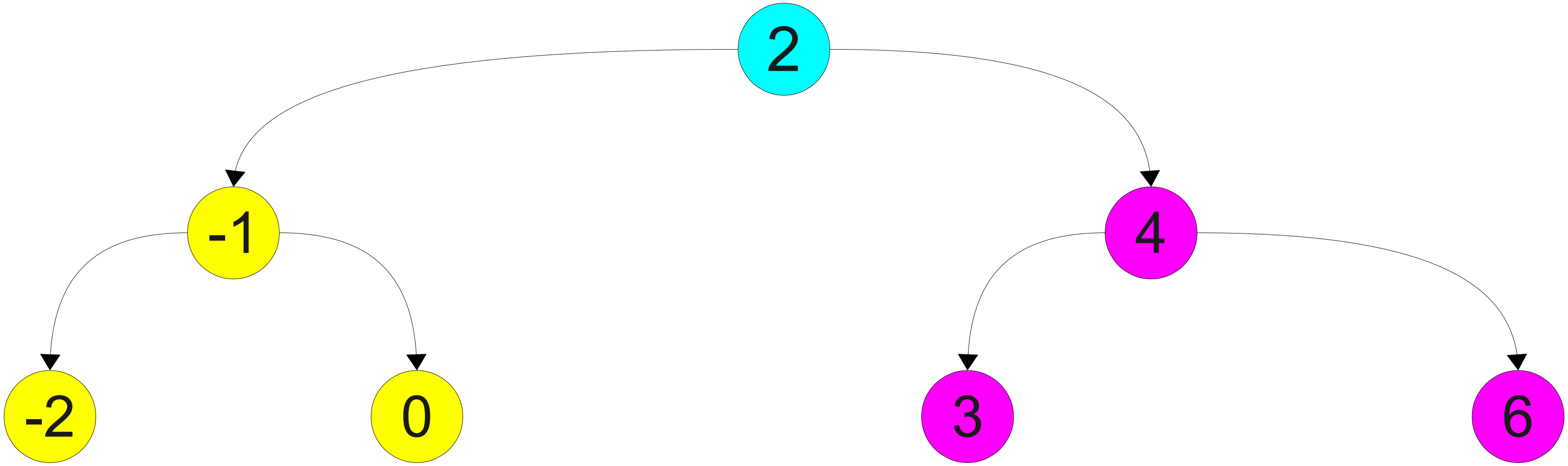
- As with a trie, a BST is a collection of **nodes**.
- The top node is called the **root node**.
- Nodes with no children are called **leaves**.



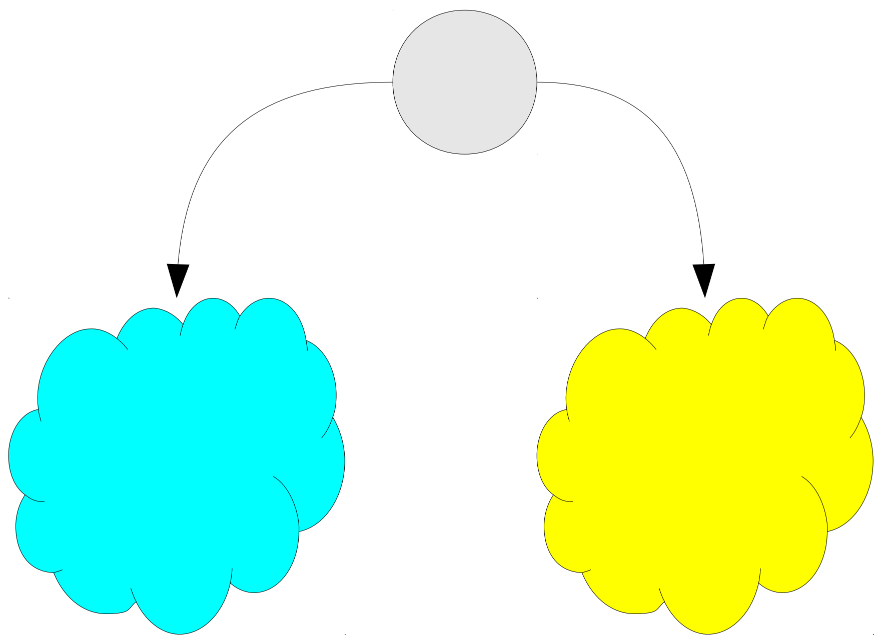
A Recursive View of BSTs



A Recursive View of BSTs

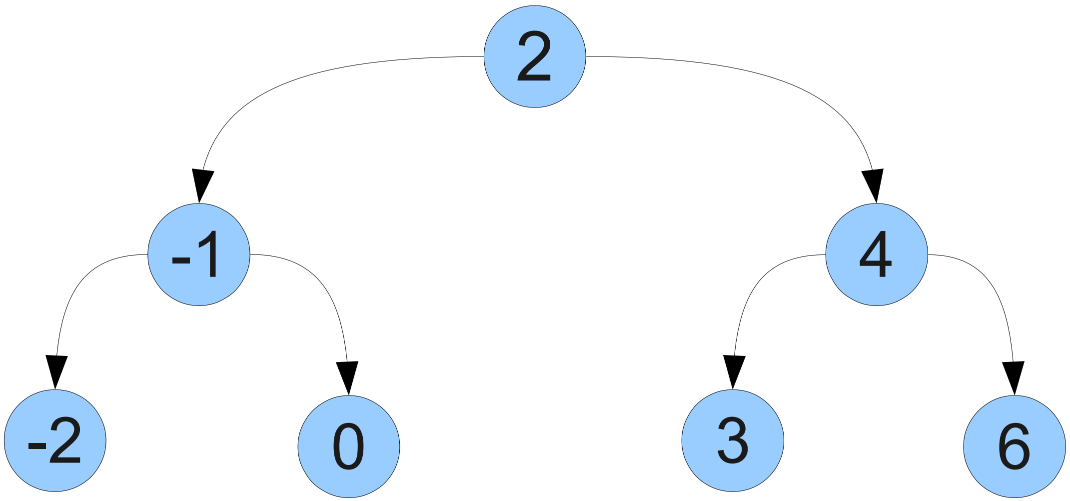


A Recursive View of BSTs

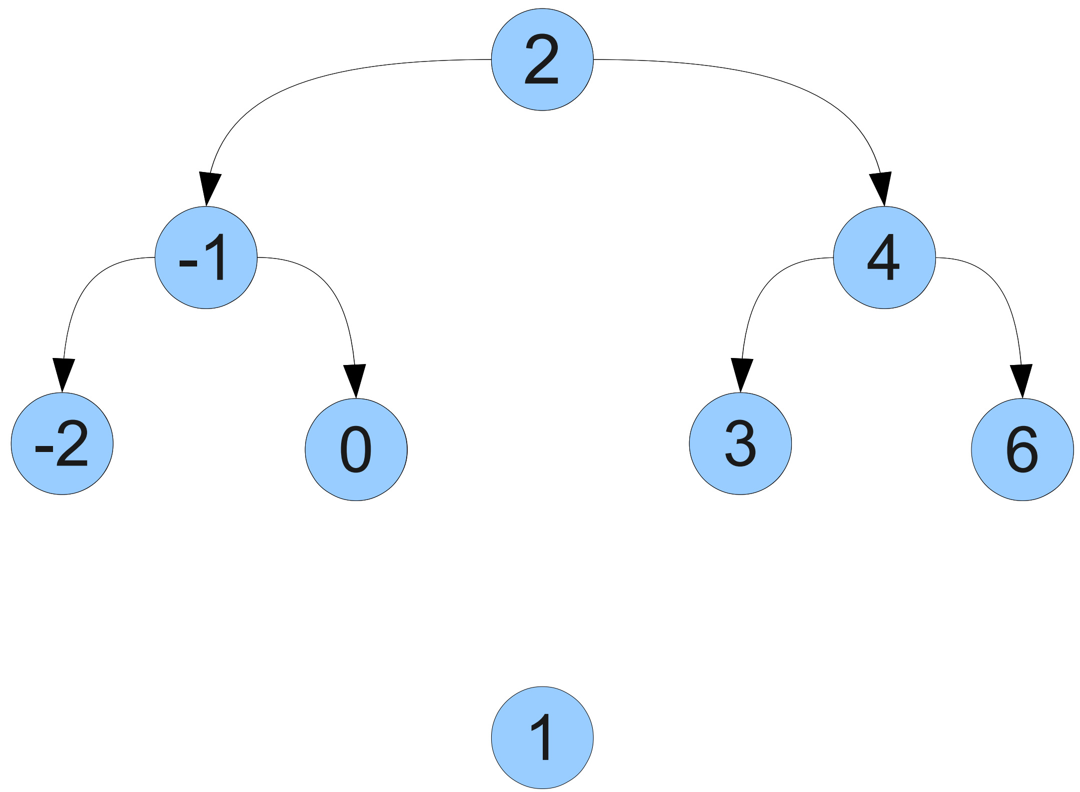


Implementing Lookups

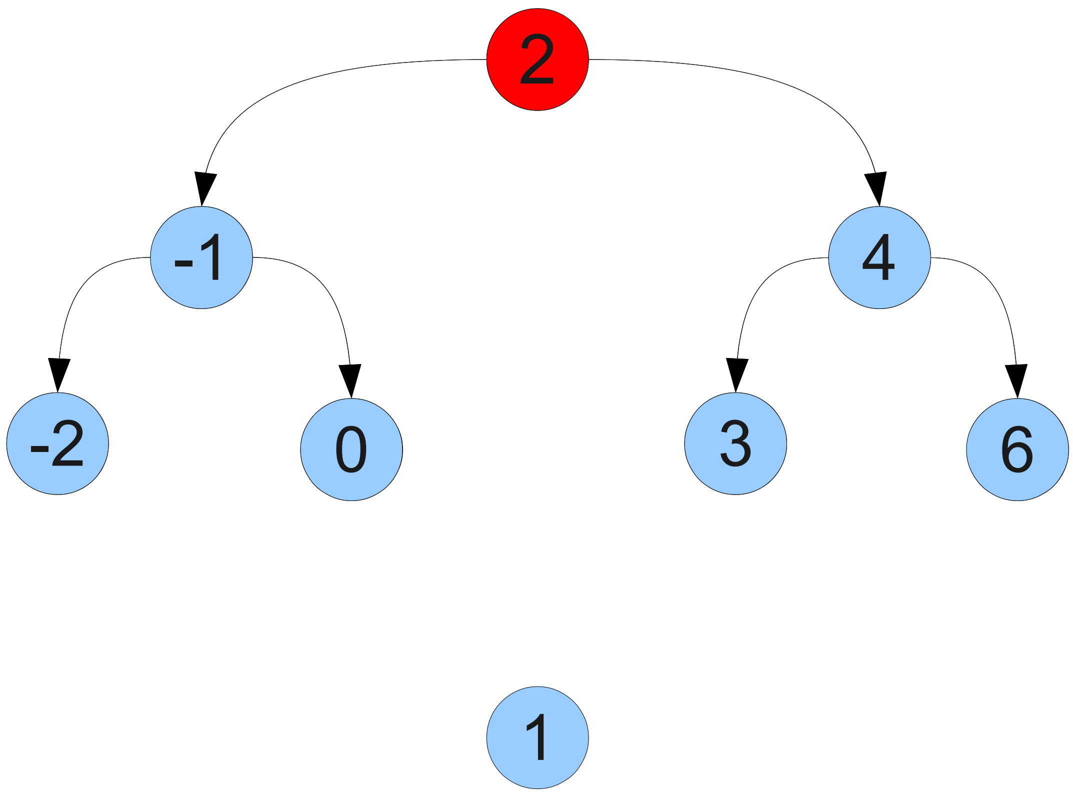
Inserting into a BST



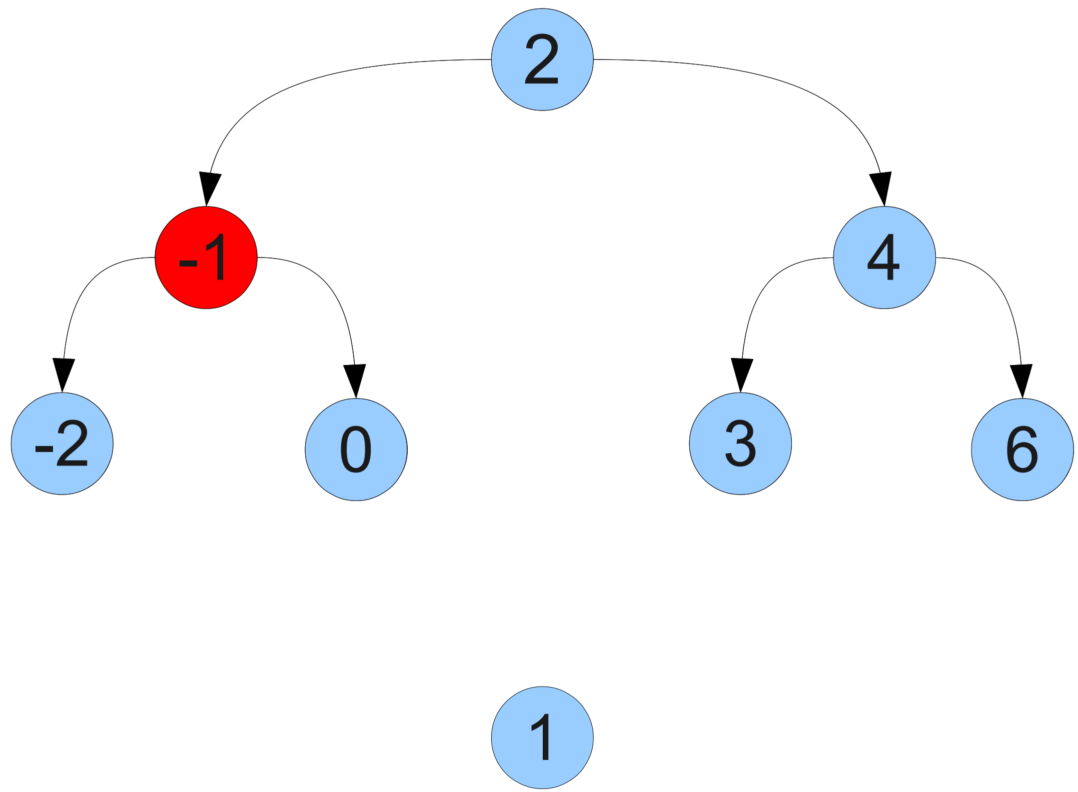
Inserting into a BST



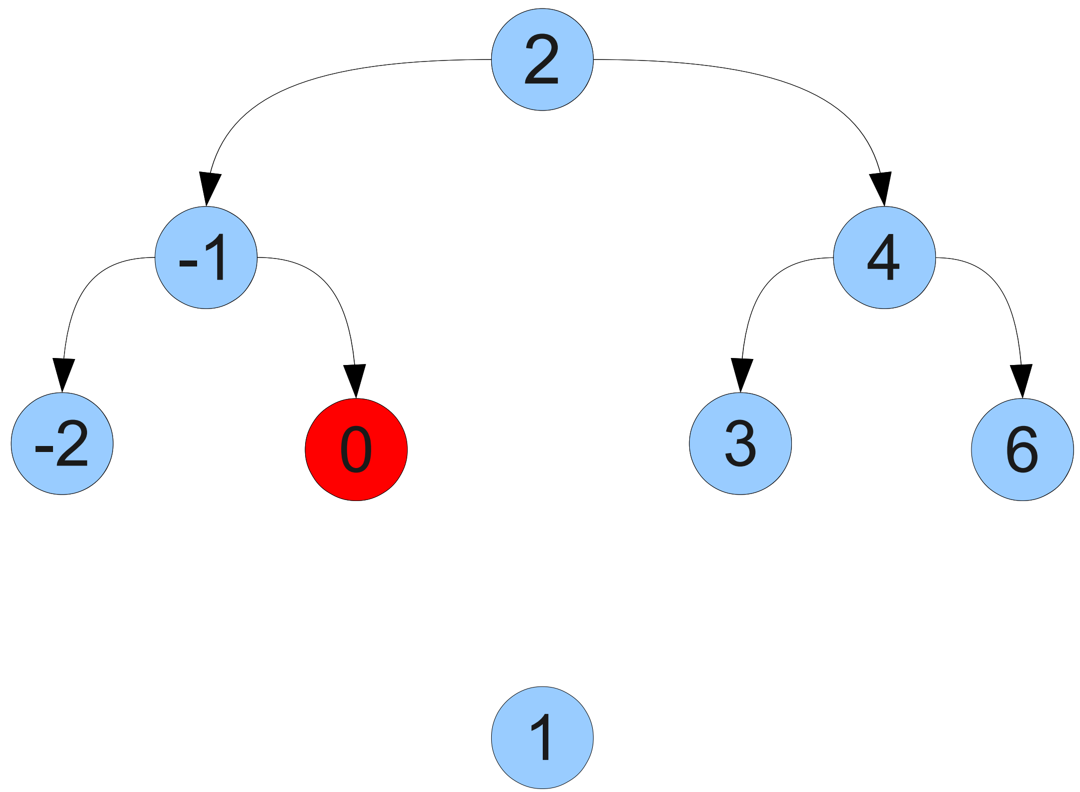
Inserting into a BST



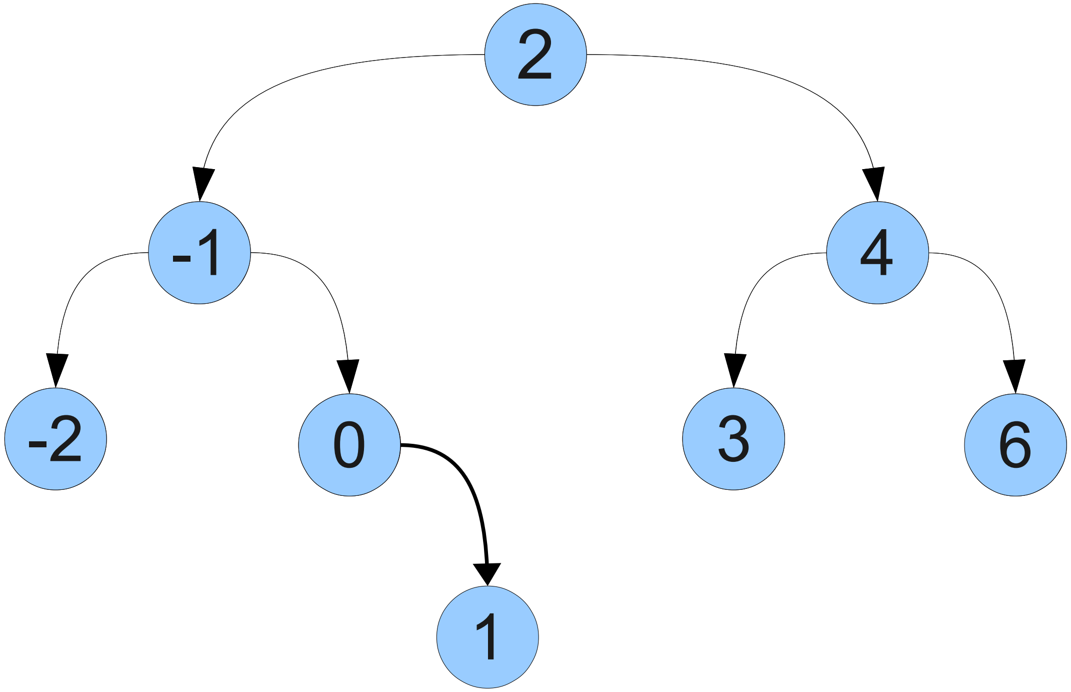
Inserting into a BST



Inserting into a BST



Inserting into a BST

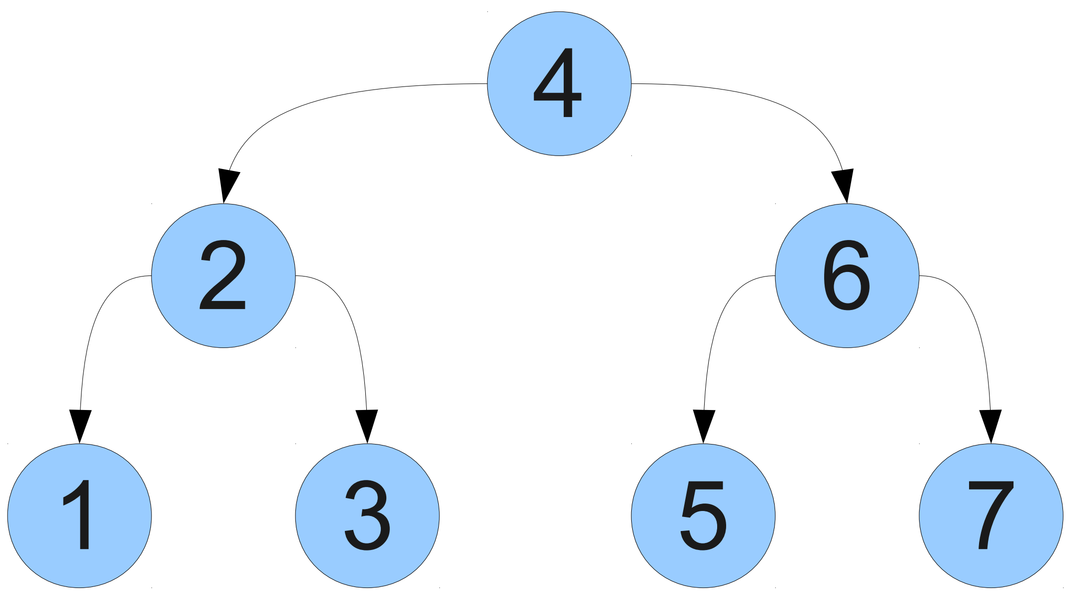


Let's Code it Up!

Insertion Order Matters

- Suppose we create a BST of numbers in this order:

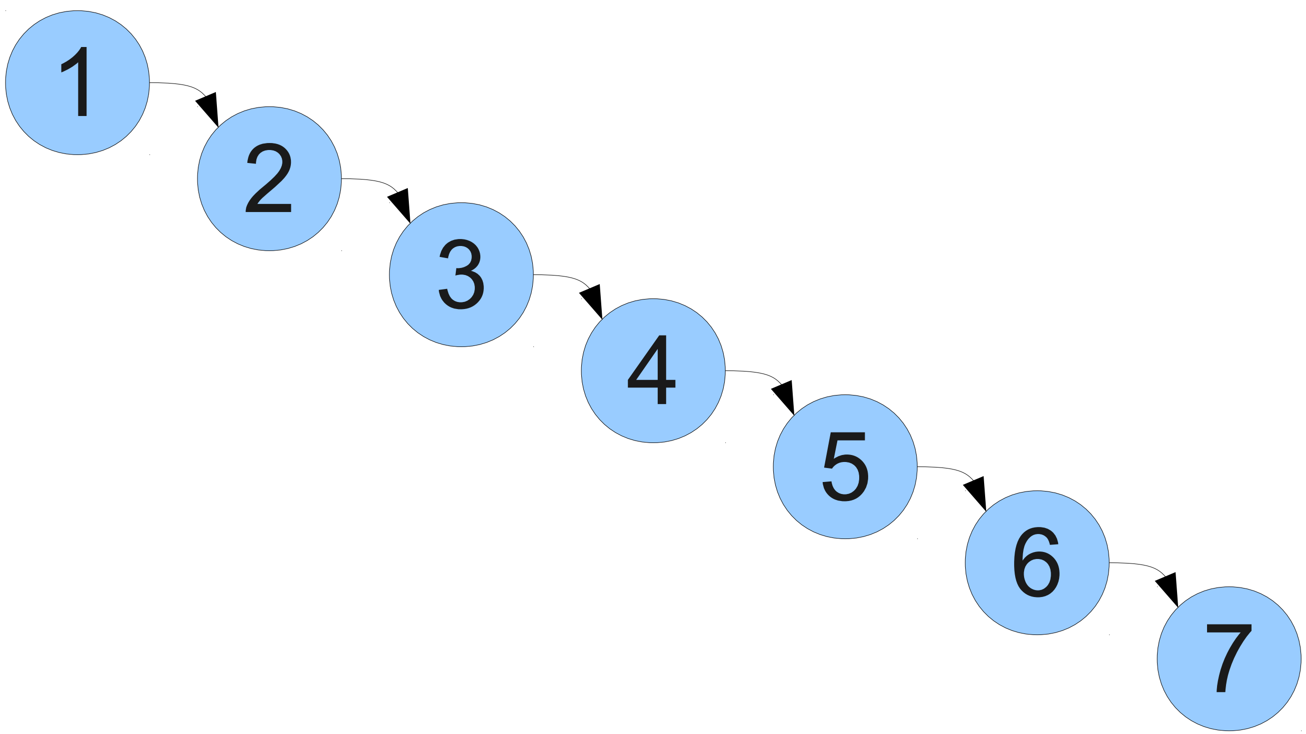
4, 2, 1, 3, 6, 5, 7



Insertion Order Matters

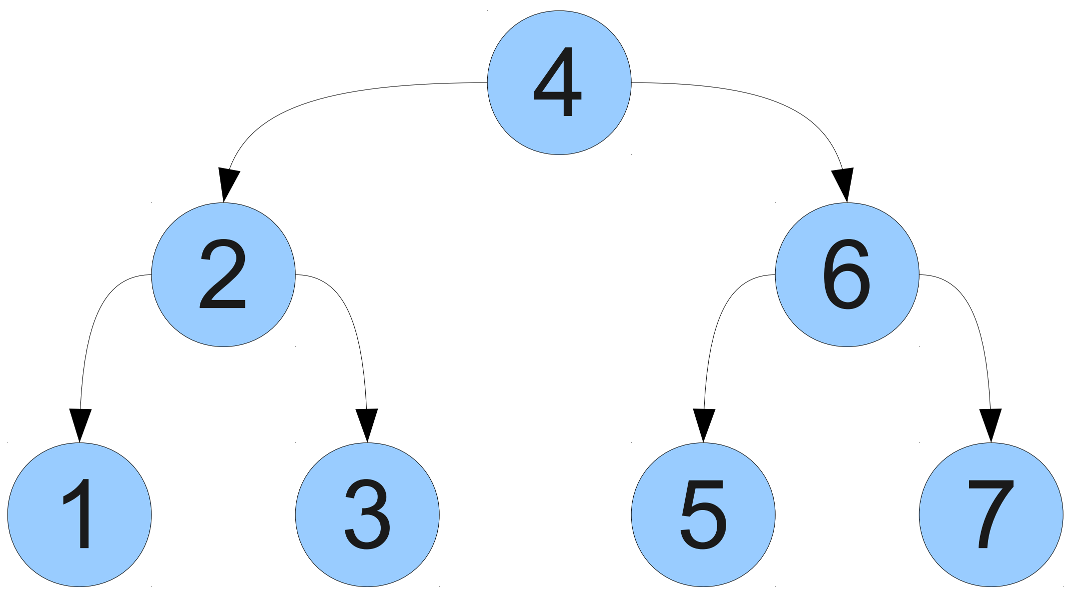
- Suppose we create a BST of numbers in this order:

1, 2, 3, 4, 5, 6, 7



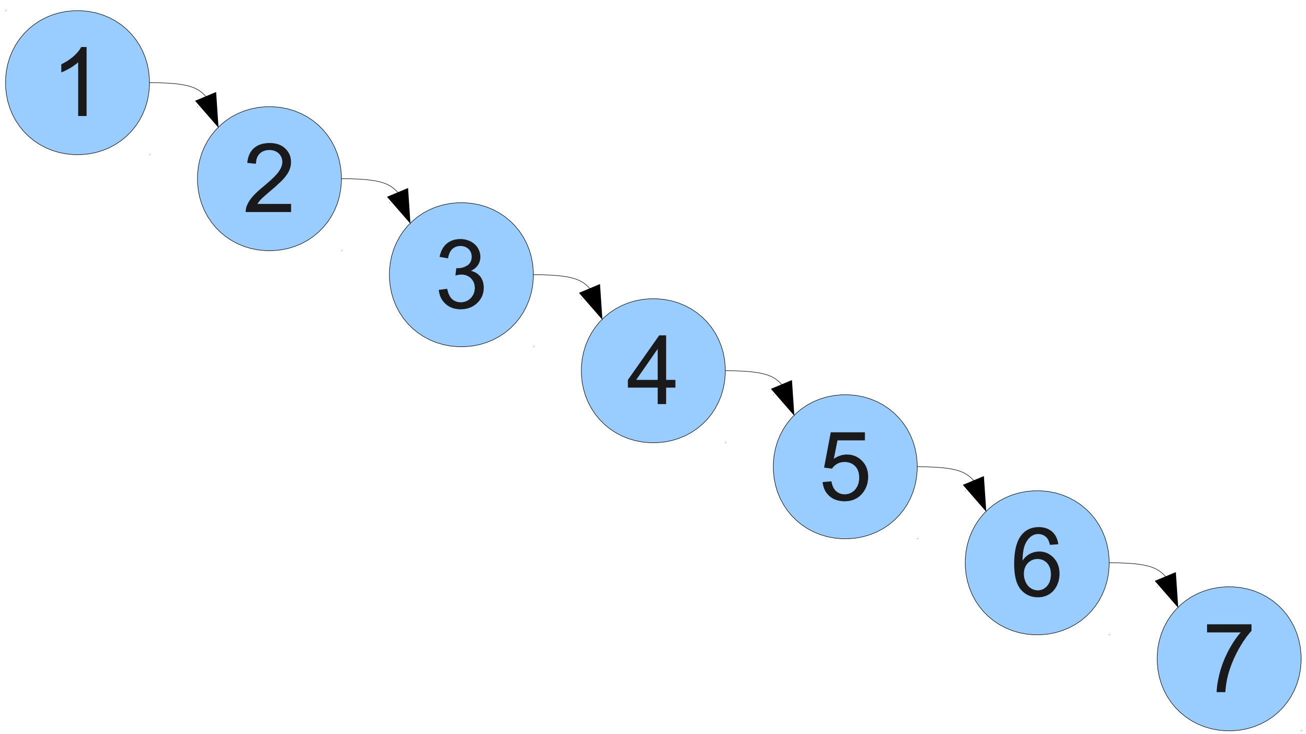
Tree Terminology

- The **height** of a tree is the number of nodes in the longest path from the root to a leaf.



Tree Terminology

- The **height** of a tree is the number of nodes in the longest path from the root to a leaf.



Efficiency of Insertion

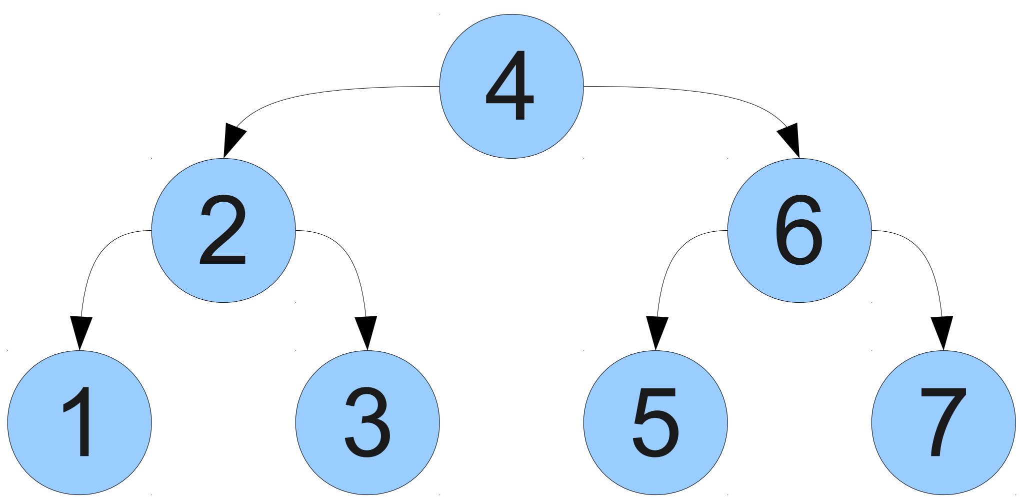
- What is the big-O complexity of adding a node to a tree?
- Depends on the height of a tree!
- Worst-case: have to take the longest path down to find where the node goes.
- Time is $O(h)$, where h is the height of the tree.

Tree Heights

- What are the maximum and minimum heights of a tree with n nodes?
- Maximum height: all nodes in a chain. Height is $O(n)$.

Tree Heights

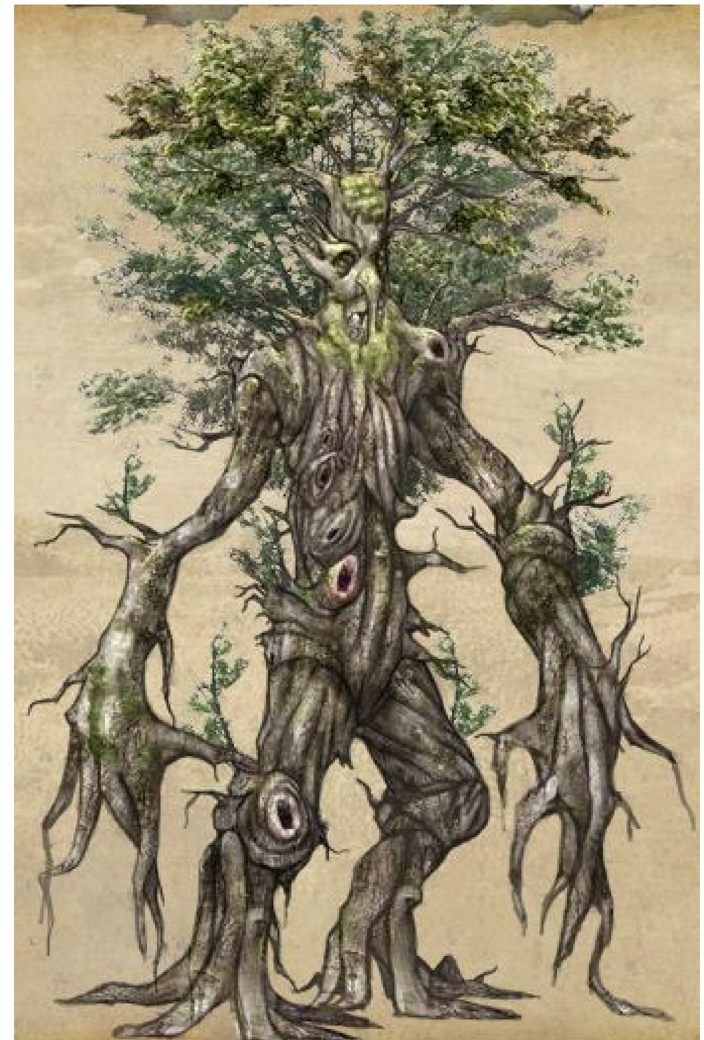
- What are the maximum and minimum heights of a tree with *n* nodes?
- Maximum height: all nodes in a chain. Height is O(*n*).
- Minimum height: Tree is as complete as possible. Height is O(log *n*).



Keeping the Height Low

- There are many modifications of the binary search tree designed to keep the height of the tree low (usually O(log n)).
- A **self-balancing binary search tree** is a binary search tree that automatically adjusts itself to keep the height low.
- Details next time.

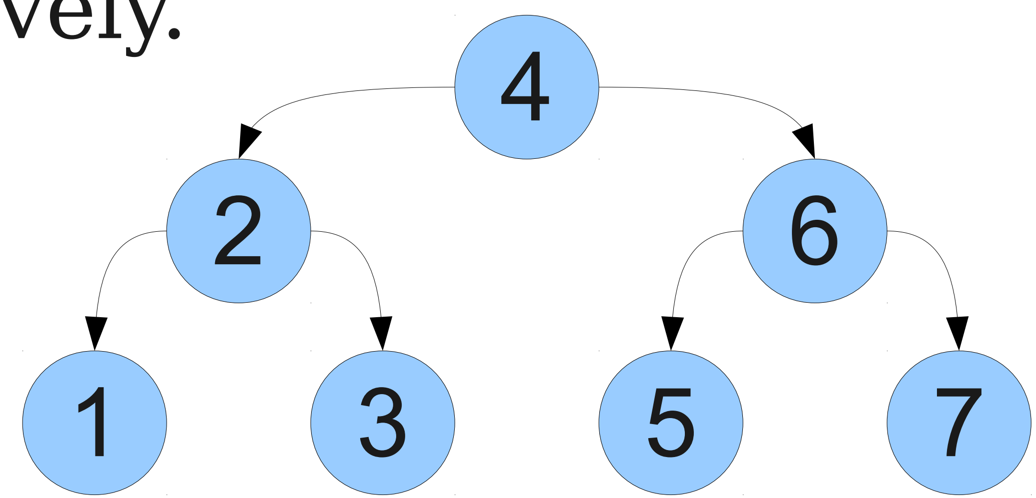
Walking Trees



http://blog.pro-ecuador.com/wp-content/uploads/2010/03/floating-tree.jpg
http://media.tumblr.com/tumblr_lp2fs6CgA11qejdya.jpg

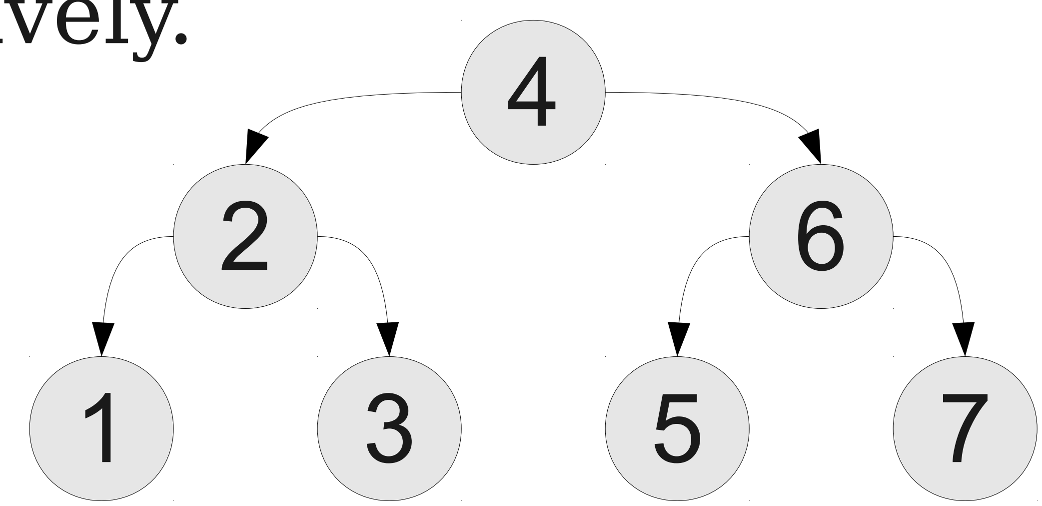
Walking a BST

- One advantage of a BST is that elements are stored in sorted order.
- We can iterate over the elements of a BST in sorted order by walking the tree recursively.



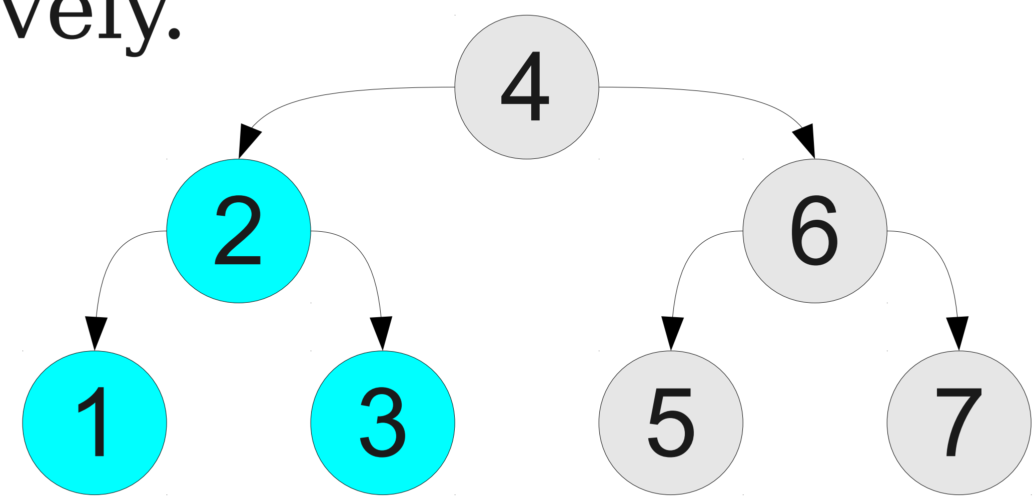
Walking a BST

- One advantage of a BST is that elements are stored in sorted order.
- We can iterate over the elements of a BST in sorted order by walking the tree recursively.



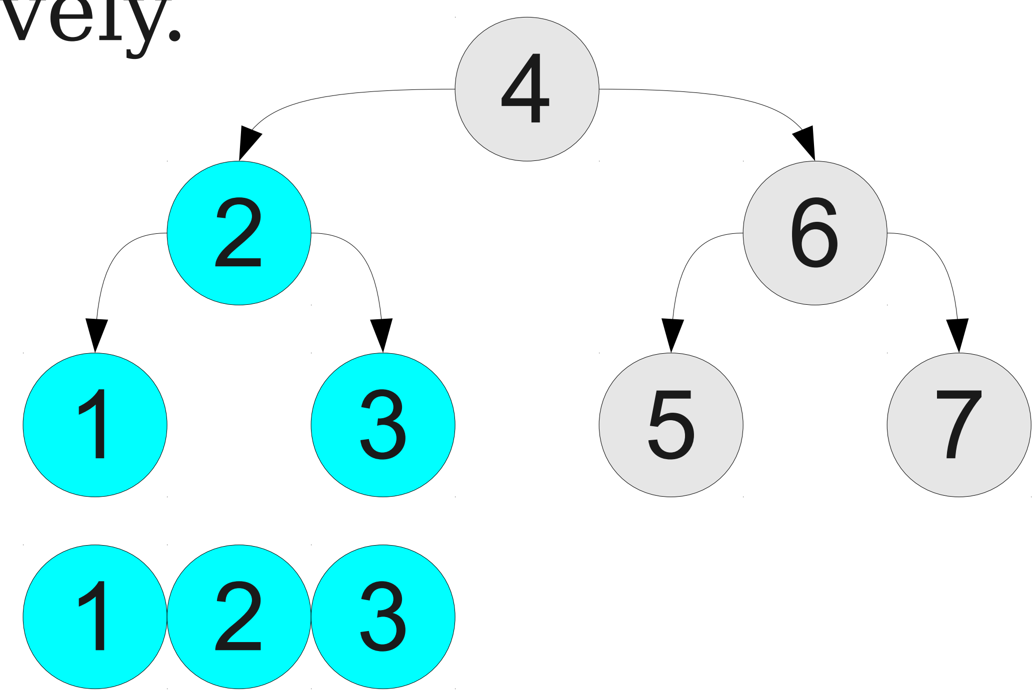
Walking a BST

- One advantage of a BST is that elements are stored in sorted order.
- We can iterate over the elements of a BST in sorted order by walking the tree recursively.



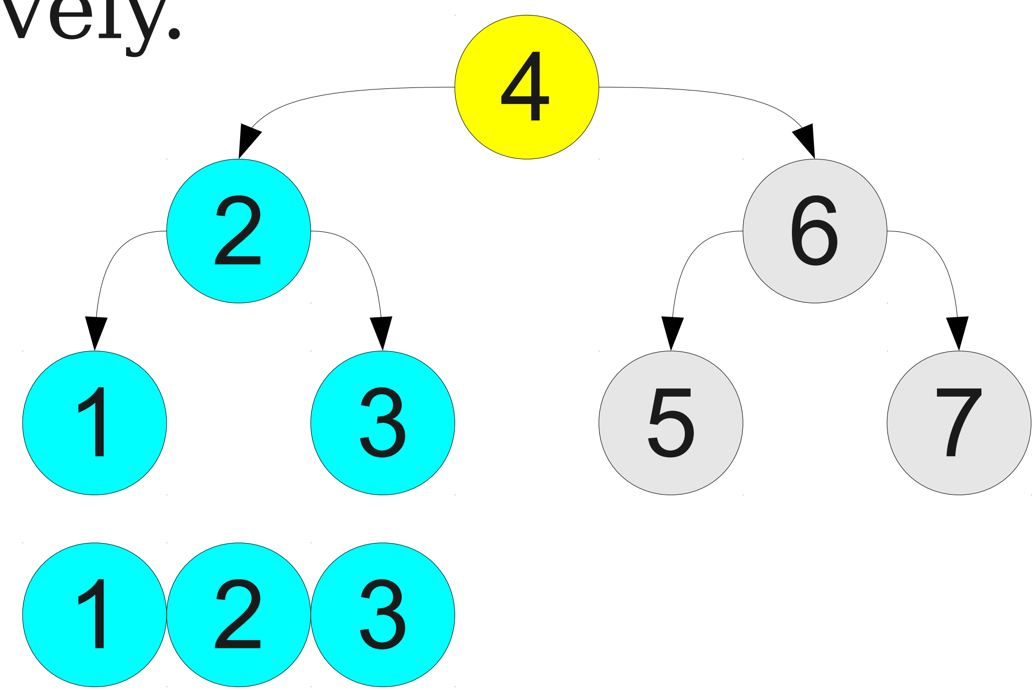
Walking a BST

- One advantage of a BST is that elements are stored in sorted order.
- We can iterate over the elements of a BST in sorted order by walking the tree recursively.



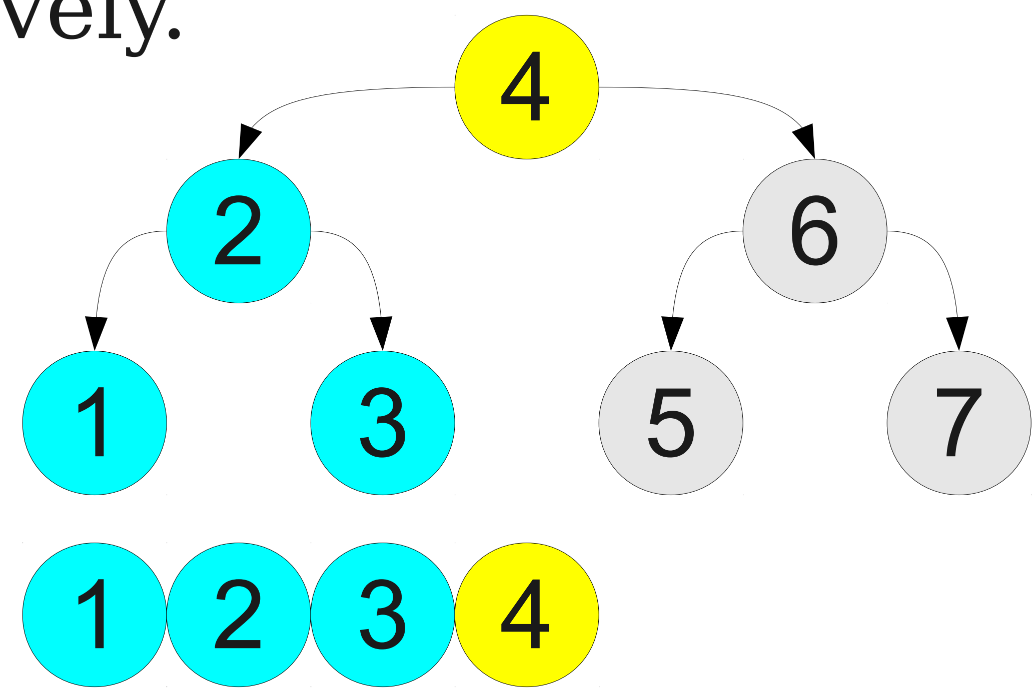
Walking a BST

- One advantage of a BST is that elements are stored in sorted order.
- We can iterate over the elements of a BST in sorted order by walking the tree recursively.



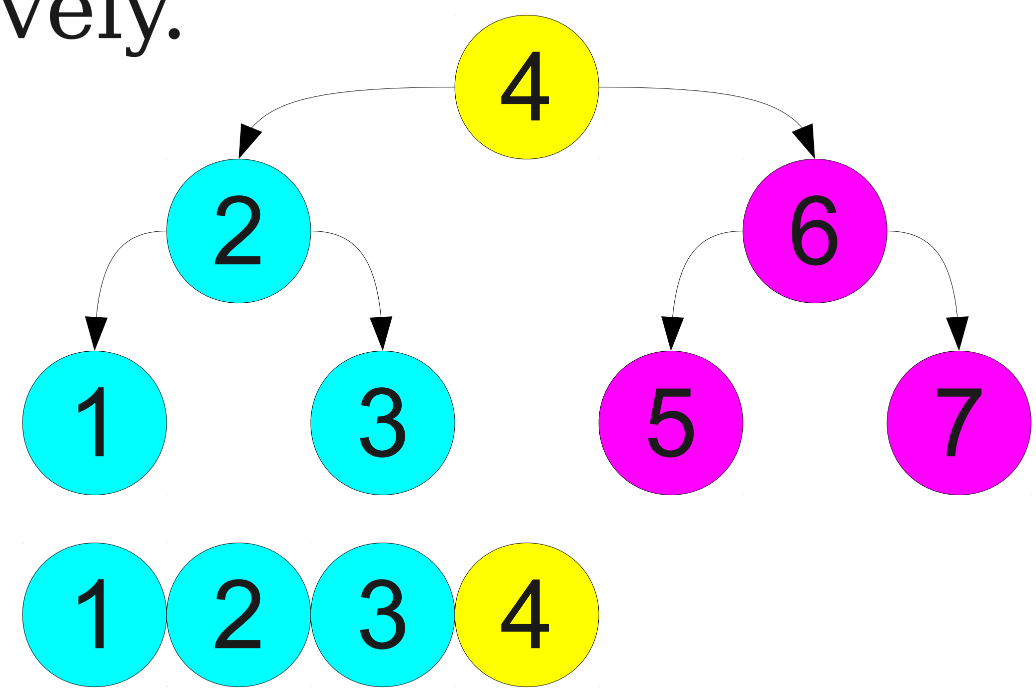
Walking a BST

- One advantage of a BST is that elements are stored in sorted order.
- We can iterate over the elements of a BST in sorted order by walking the tree recursively.



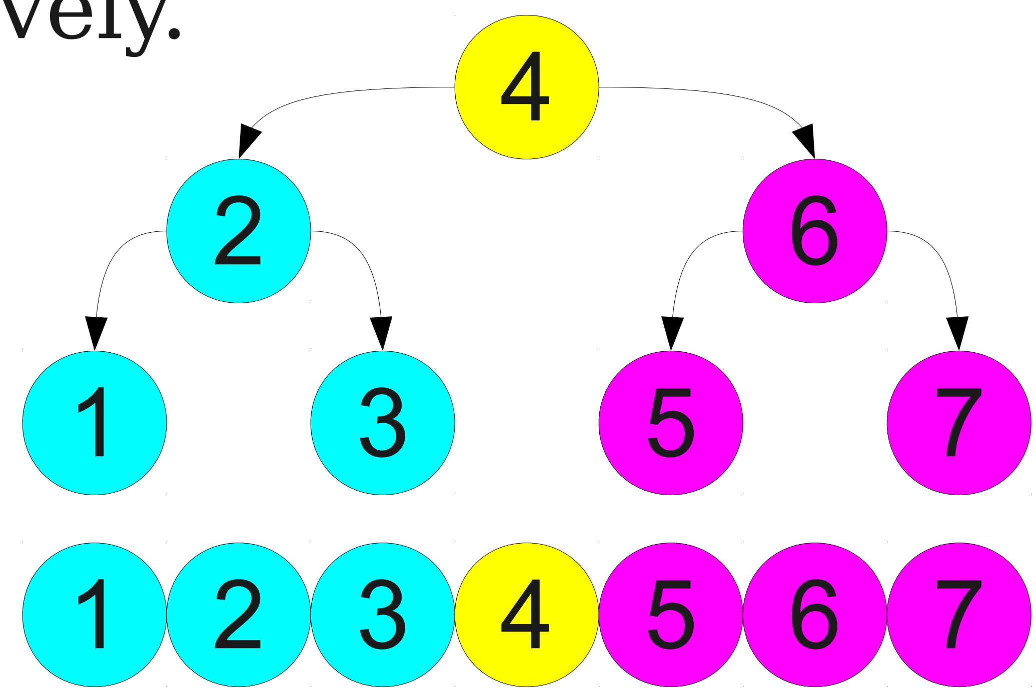
Walking a BST

- One advantage of a BST is that elements are stored in sorted order.
- We can iterate over the elements of a BST in sorted order by walking the tree recursively.



Walking a BST

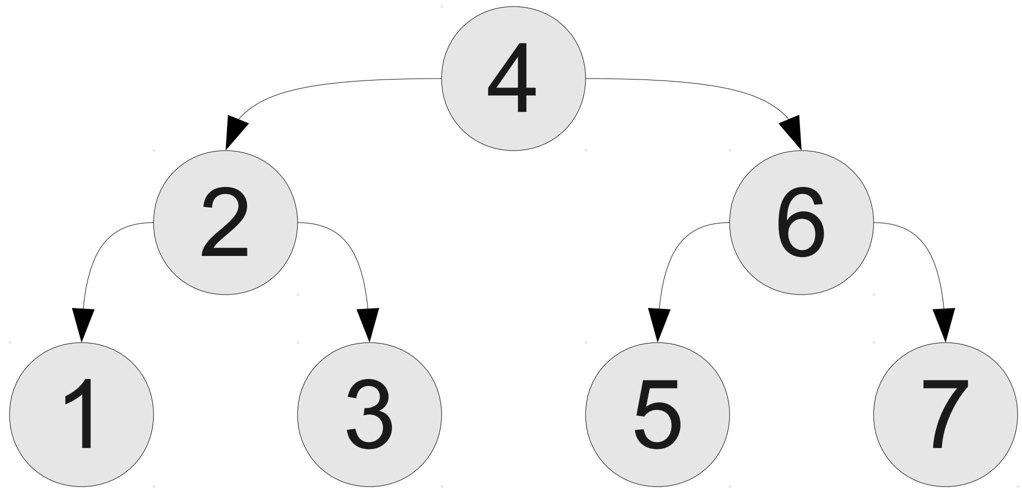
- One advantage of a BST is that elements are stored in sorted order.
- We can iterate over the elements of a BST in sorted order by walking the tree recursively.



Tree Traversals

- There are three general types of tree traversals:
- **Preorder**: Visit the node, then visit the children.
- **Inorder**: Visit the left child, then the node, then the right child.
- **Postorder**: Visit the children, then visit the node.

Walking a Tree



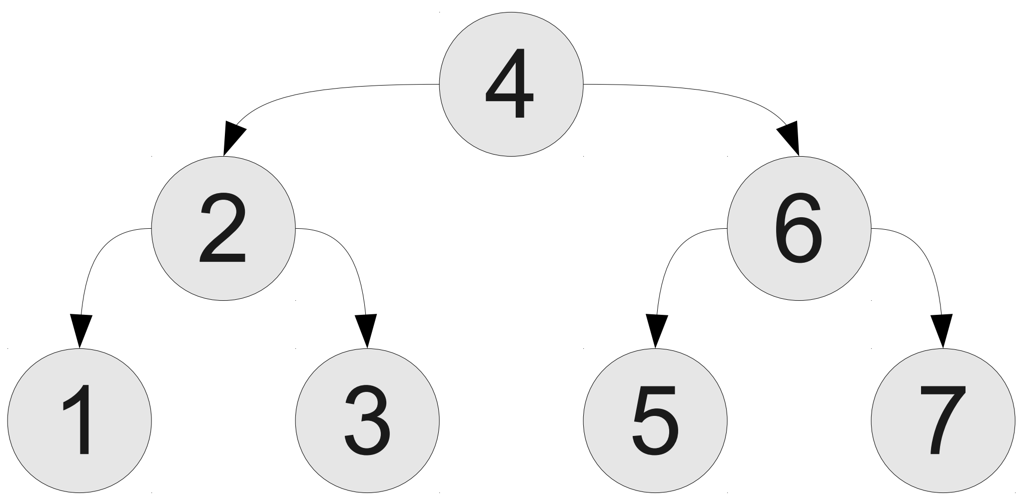
Inorder	1	2	3	4	5	6	7
Preorder	4	2	1	3	6	5	7
Postorder	1	3	2	5	7	6	4

Getting Rid of Trees

http://www.tigersheds.com/garden-resources/image.axd?picture=2010%2F6%2Fdeforestation1.jpg

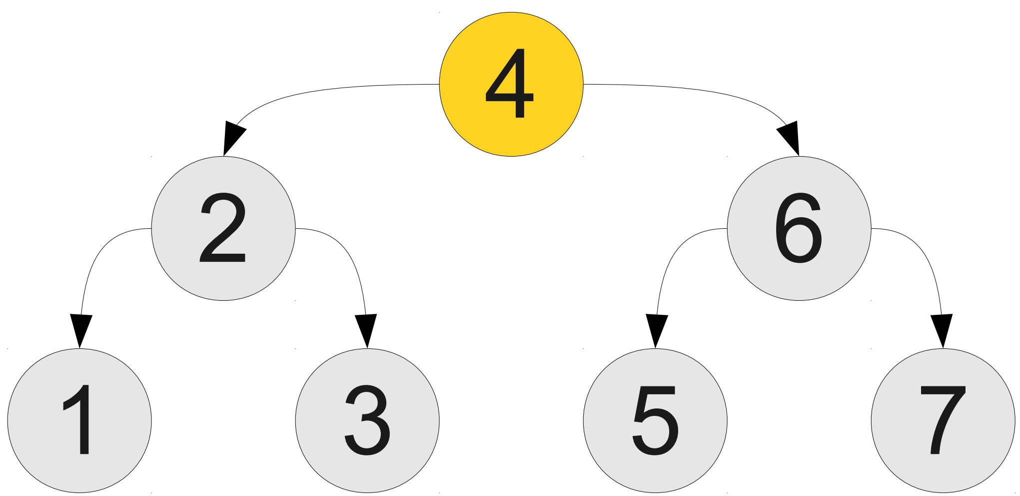
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



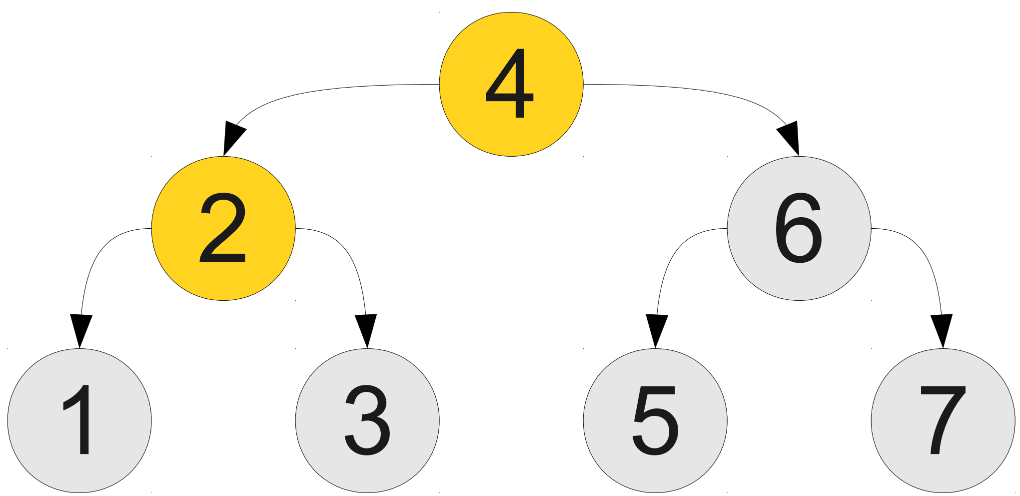
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



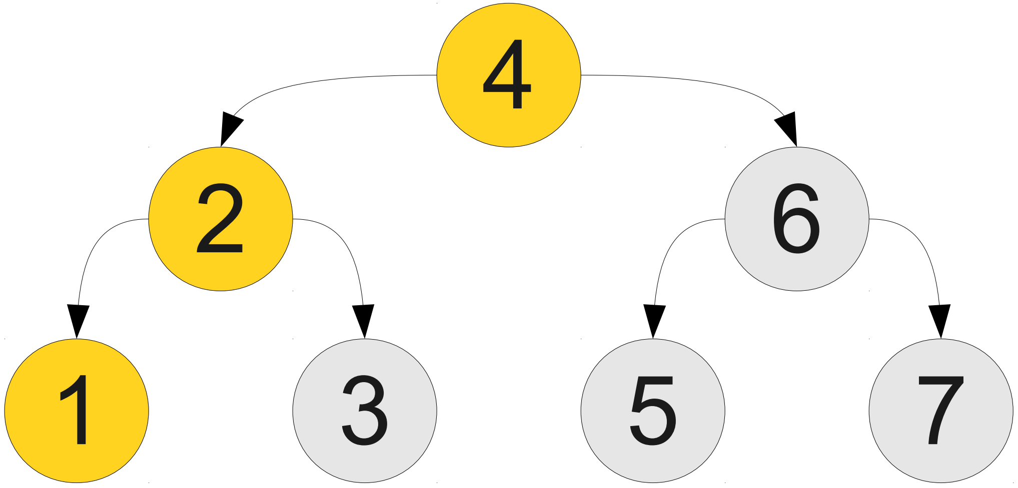
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



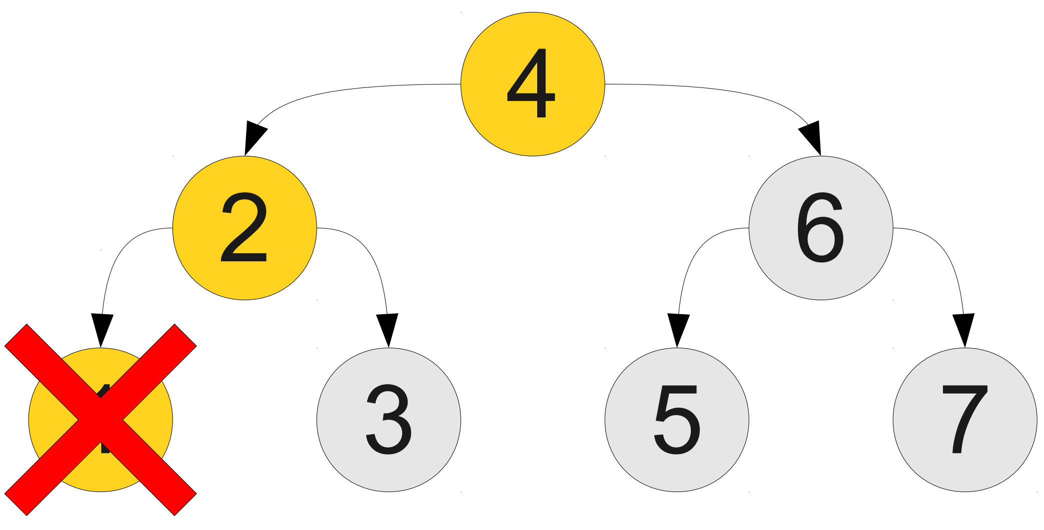
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



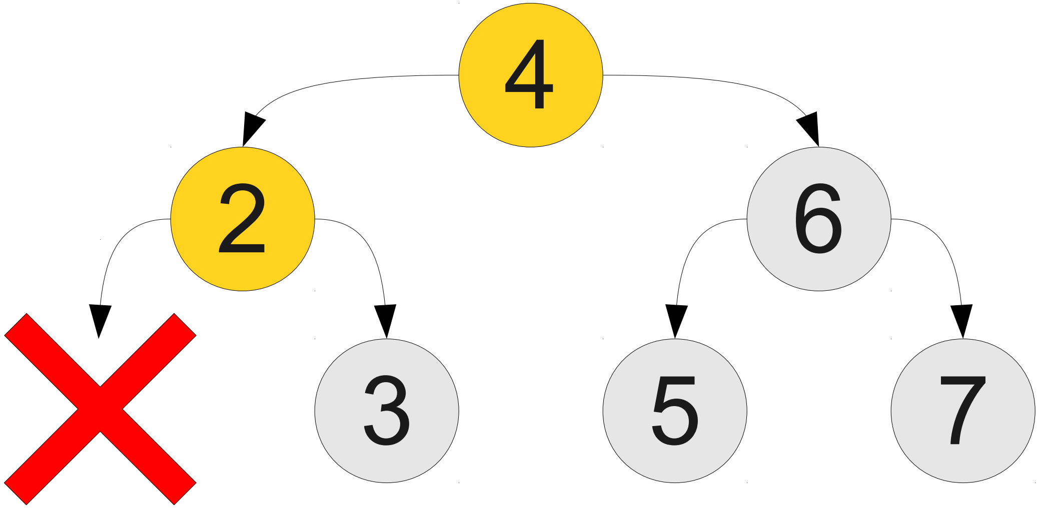
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



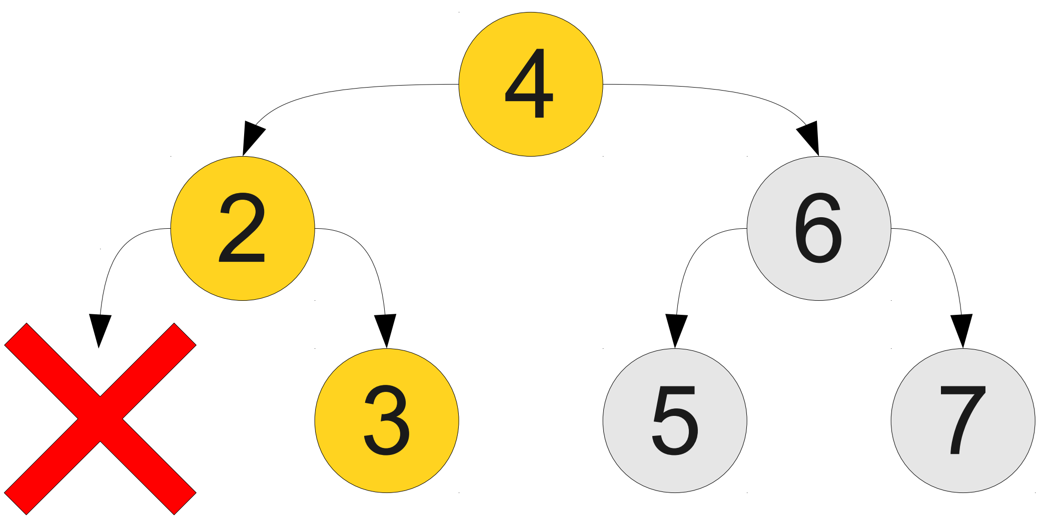
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



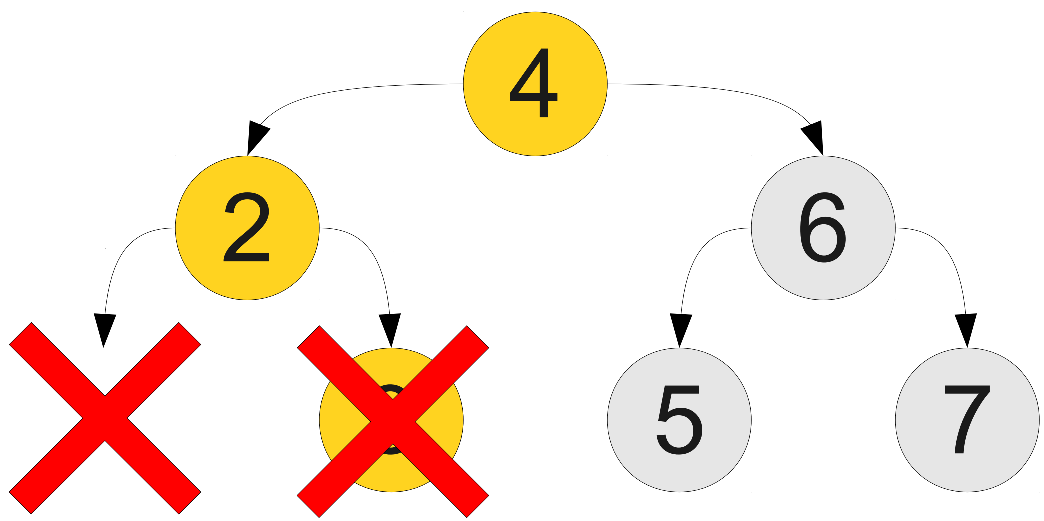
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



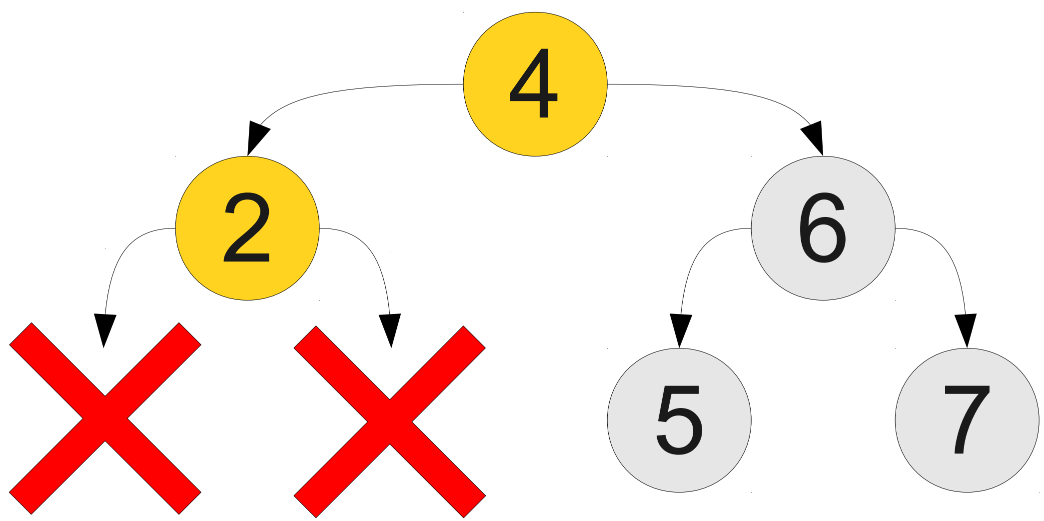
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



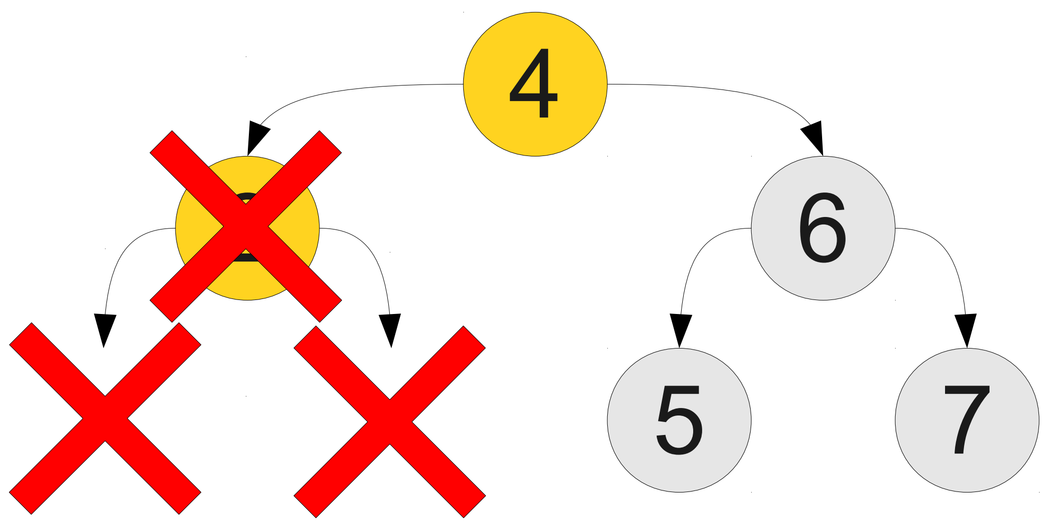
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



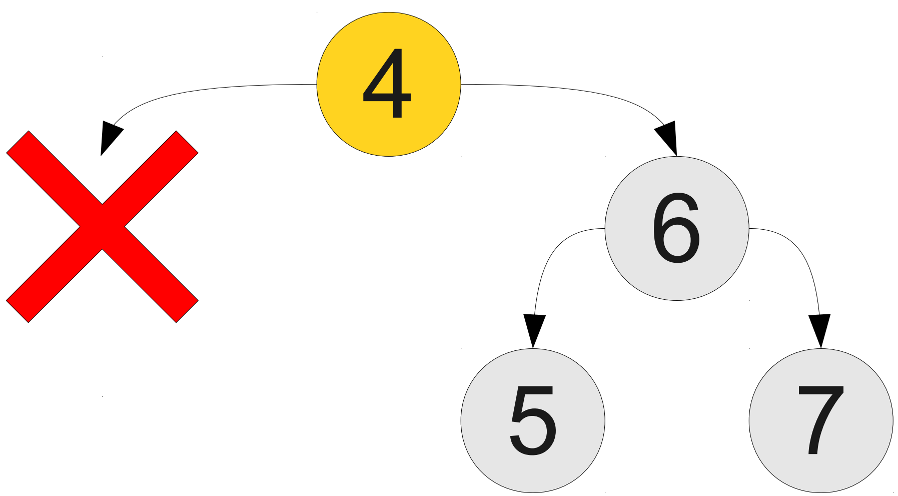
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



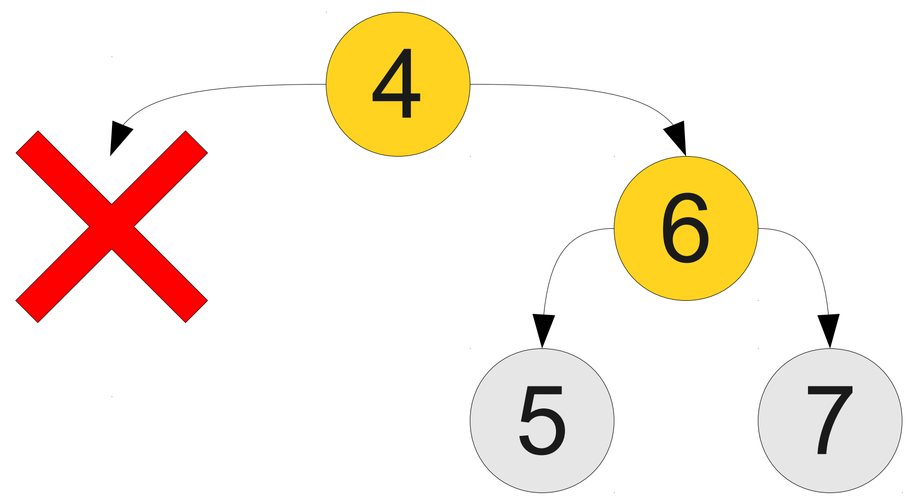
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



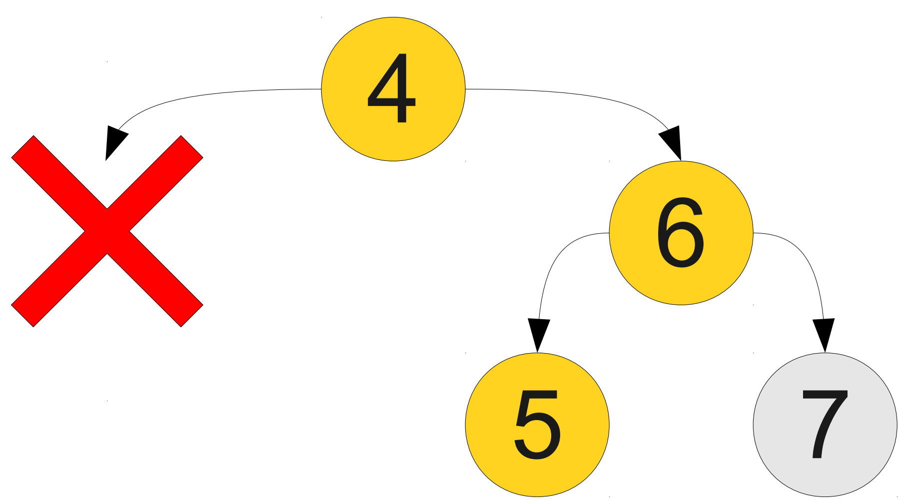
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



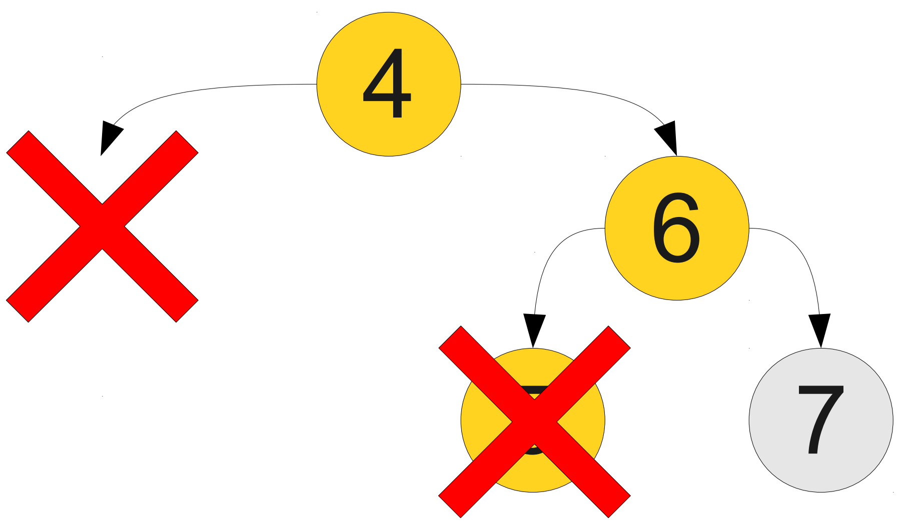
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



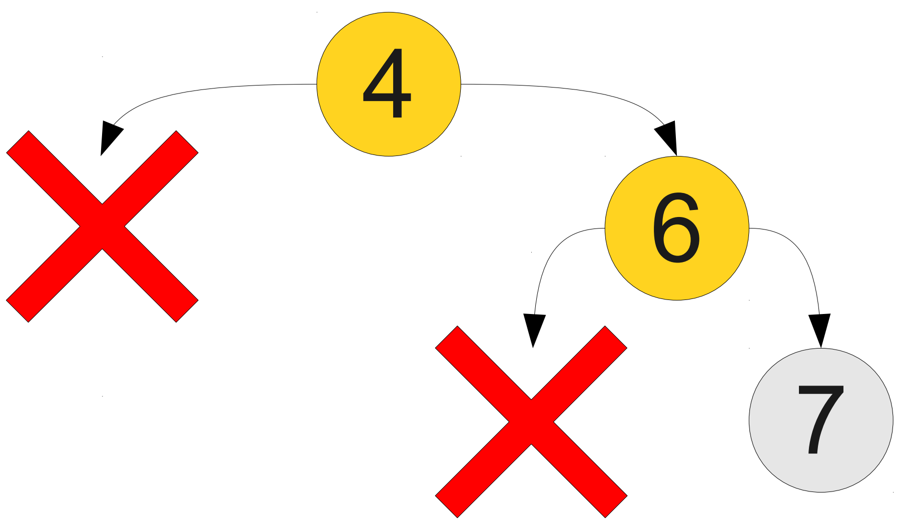
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



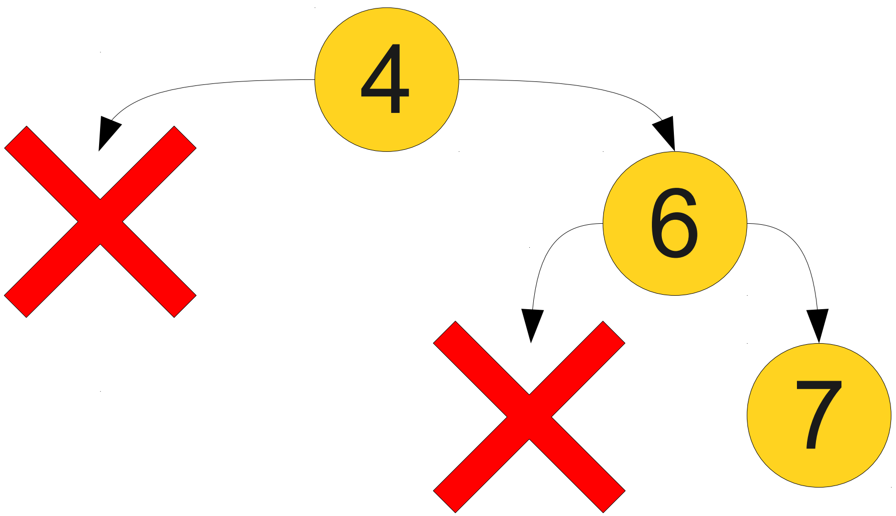
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



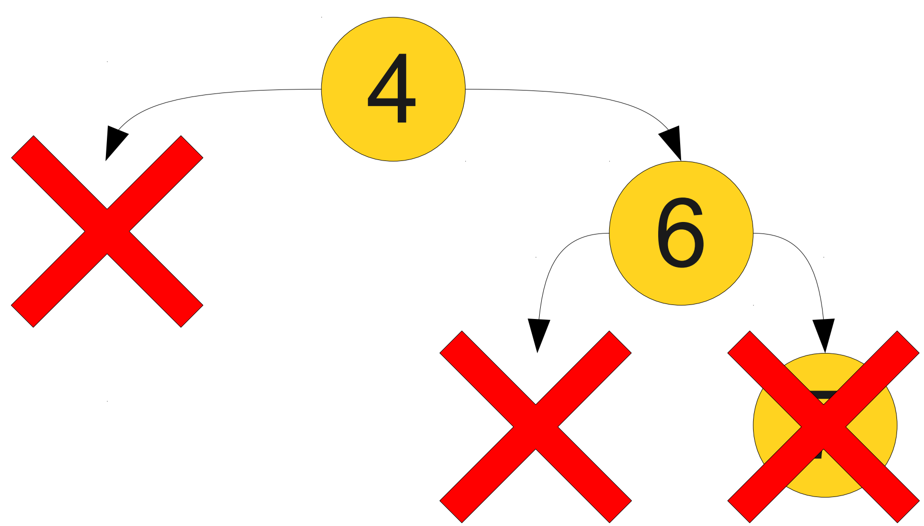
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



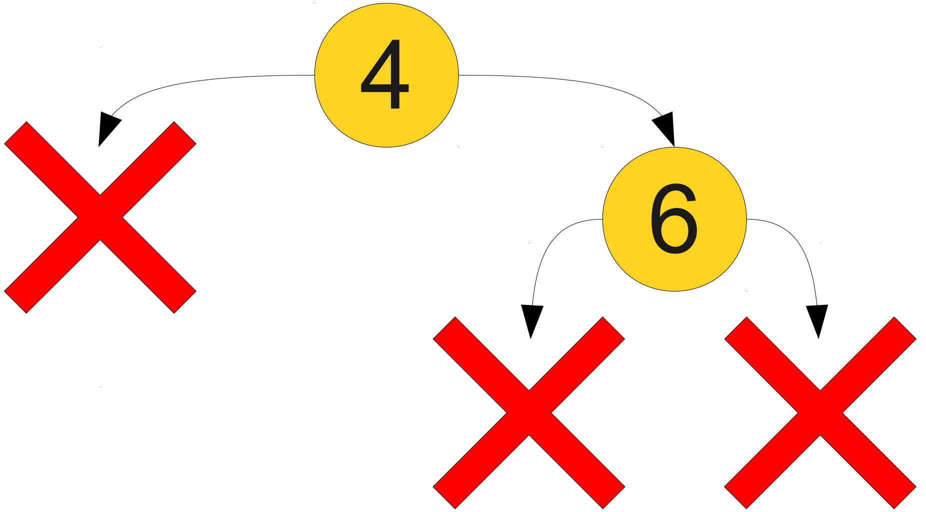
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



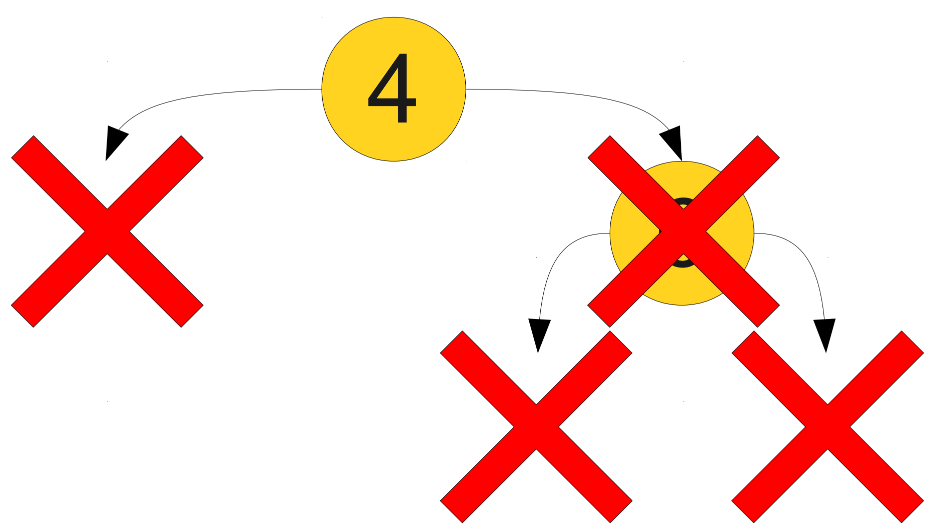
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



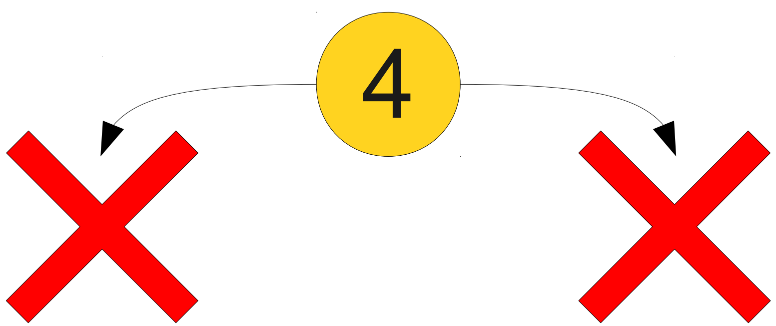
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



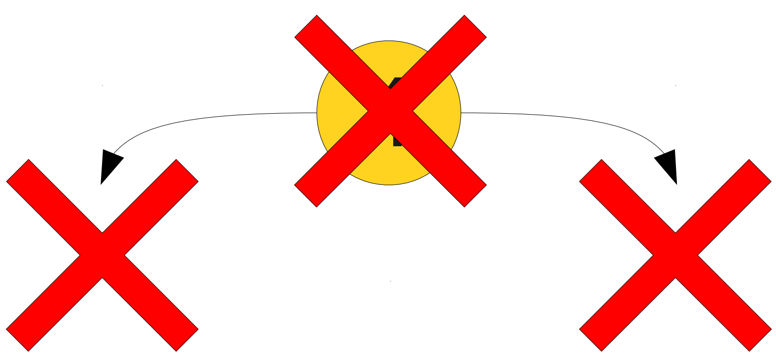
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



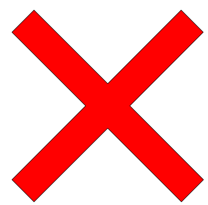
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.



Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.

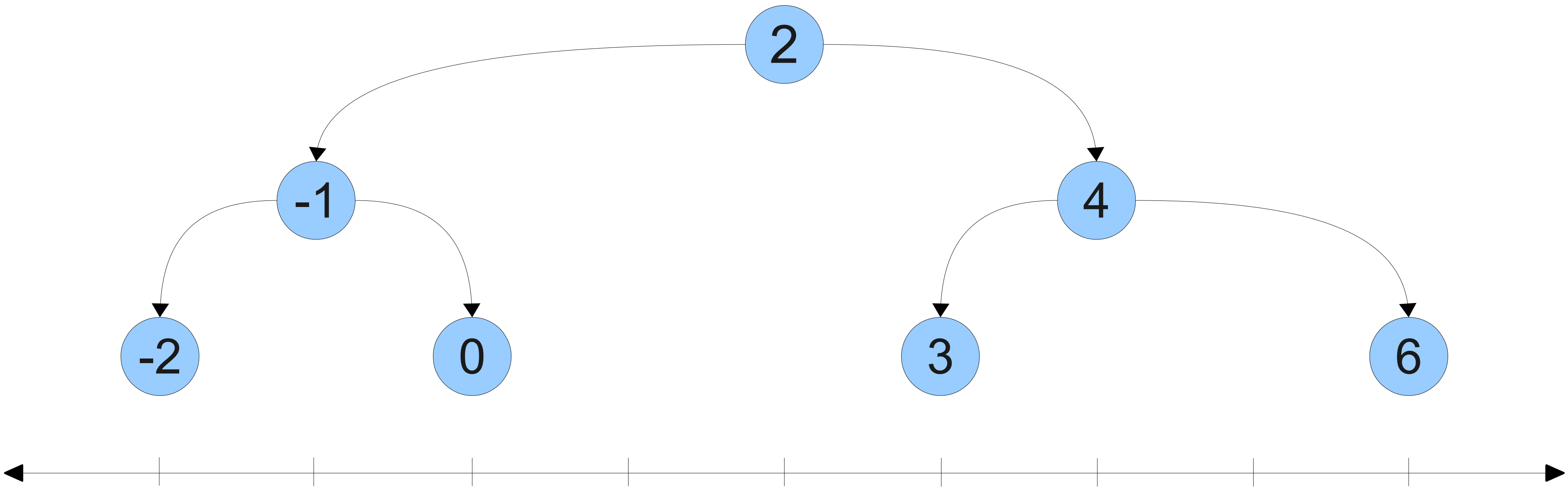
Freeing a Tree

- Once we're done with a tree, we need to free all of its nodes.
- As with a linked list, we have to be careful not to use any nodes after freeing them.
- This is done as follows:
  - **Base case**: There is nothing to delete in an empty tree.
  - **Recursive step**: Delete both subtrees, then delete the current node.
- What kind of tree traversal is this?

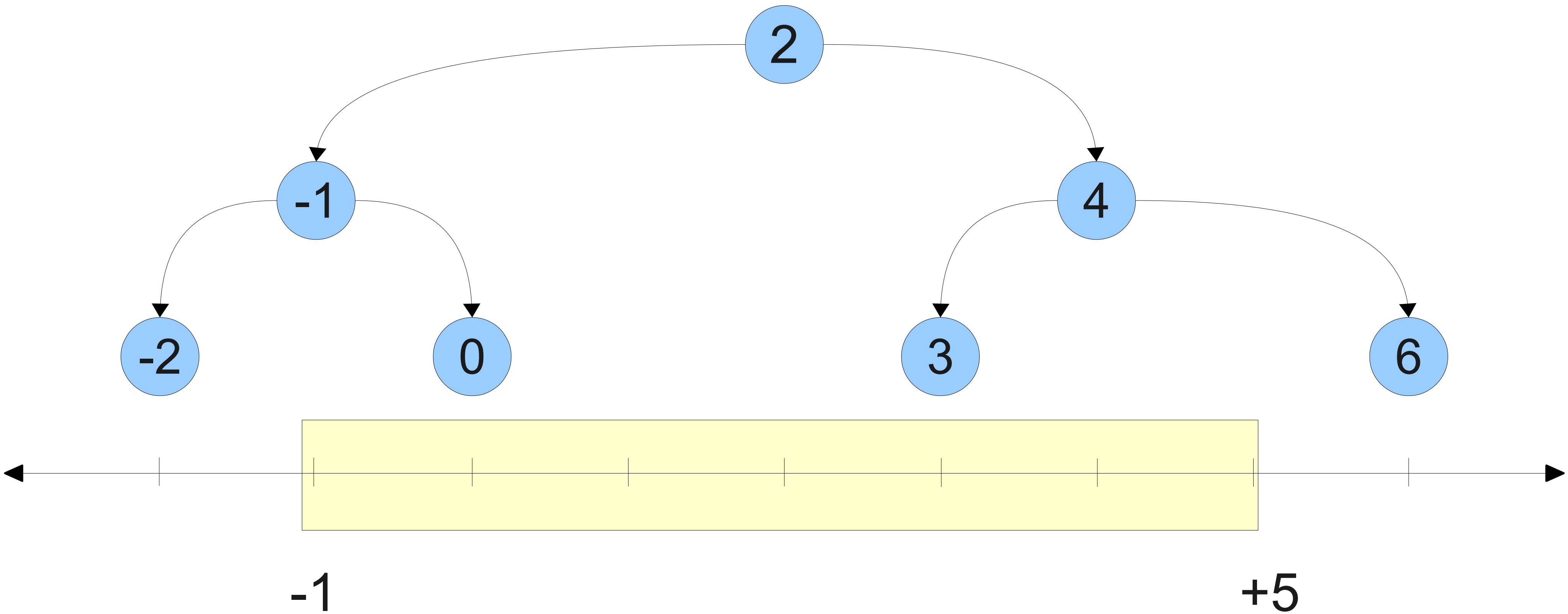
Range Searches

- We can use BSTs to do **range searches**, in which we find all values in the BST within some range.
- For example:
  - If values in a BST are dates, can find all events that occurred within some time window.
  - If values in a BST are samples of a random variable, can find everything within one and two standard deviations above the mean.

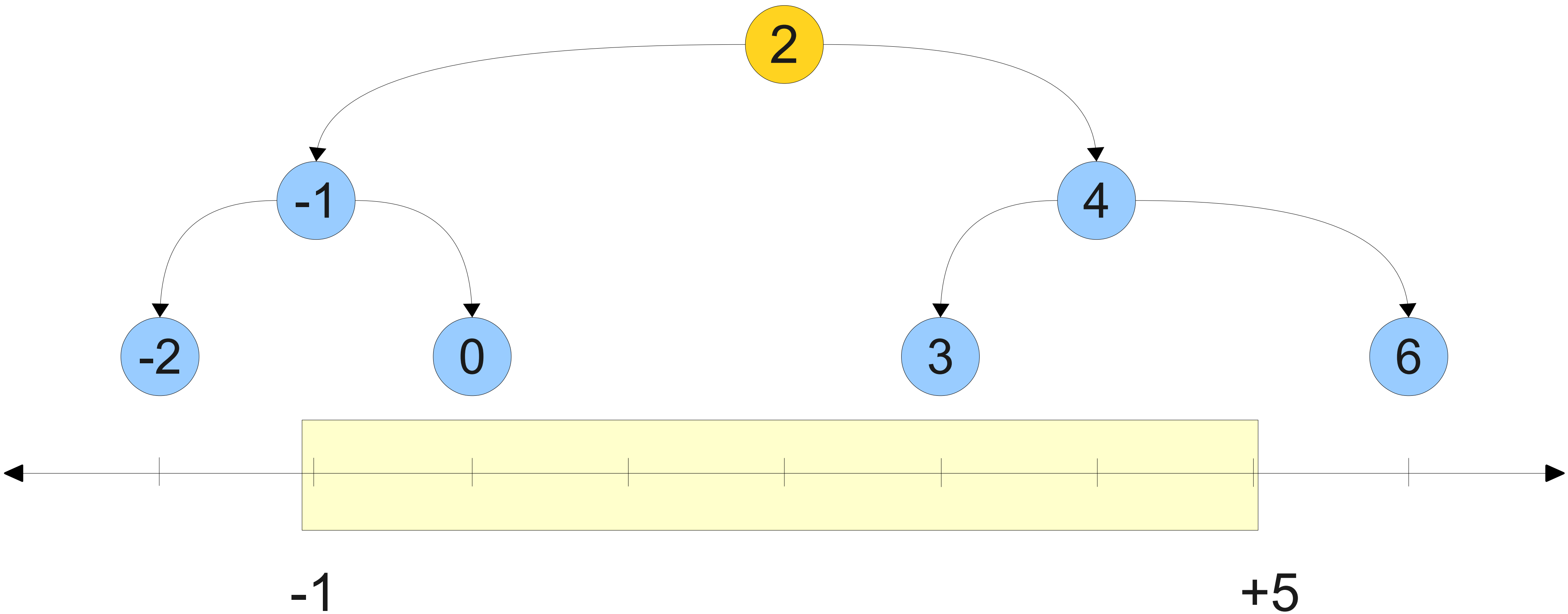
The Intuition



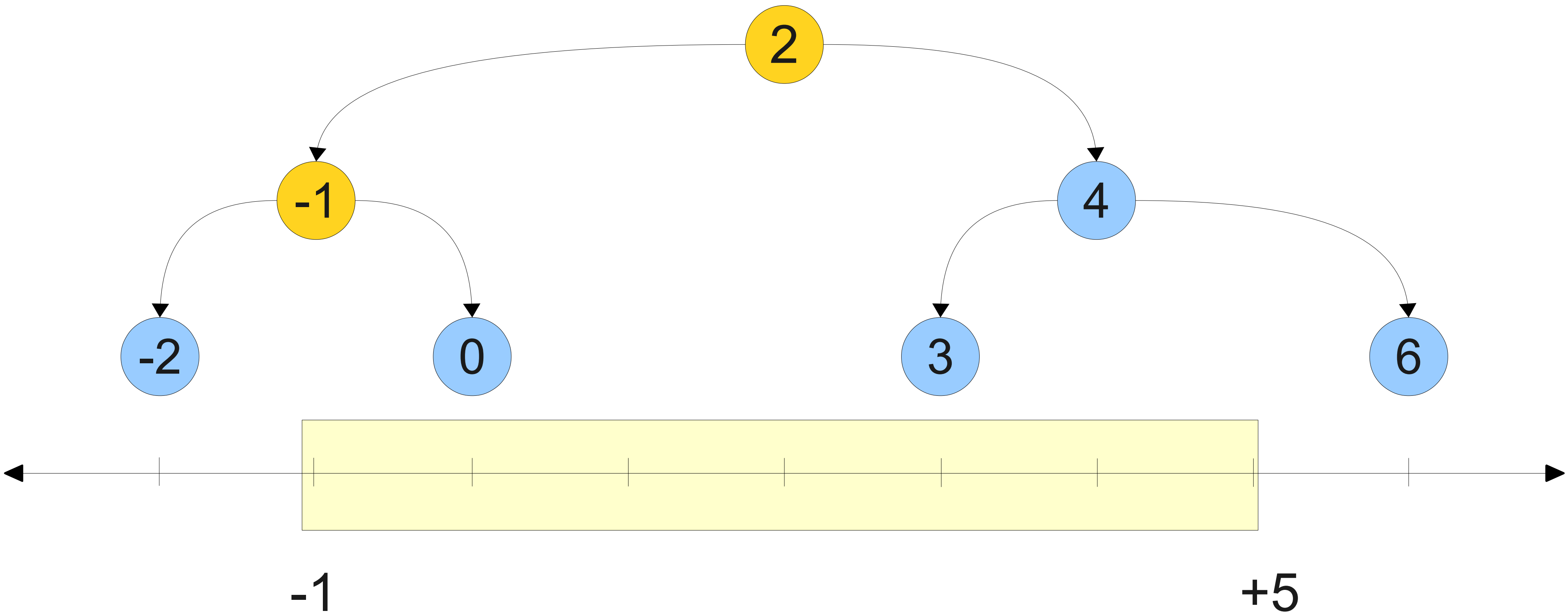
The Intuition



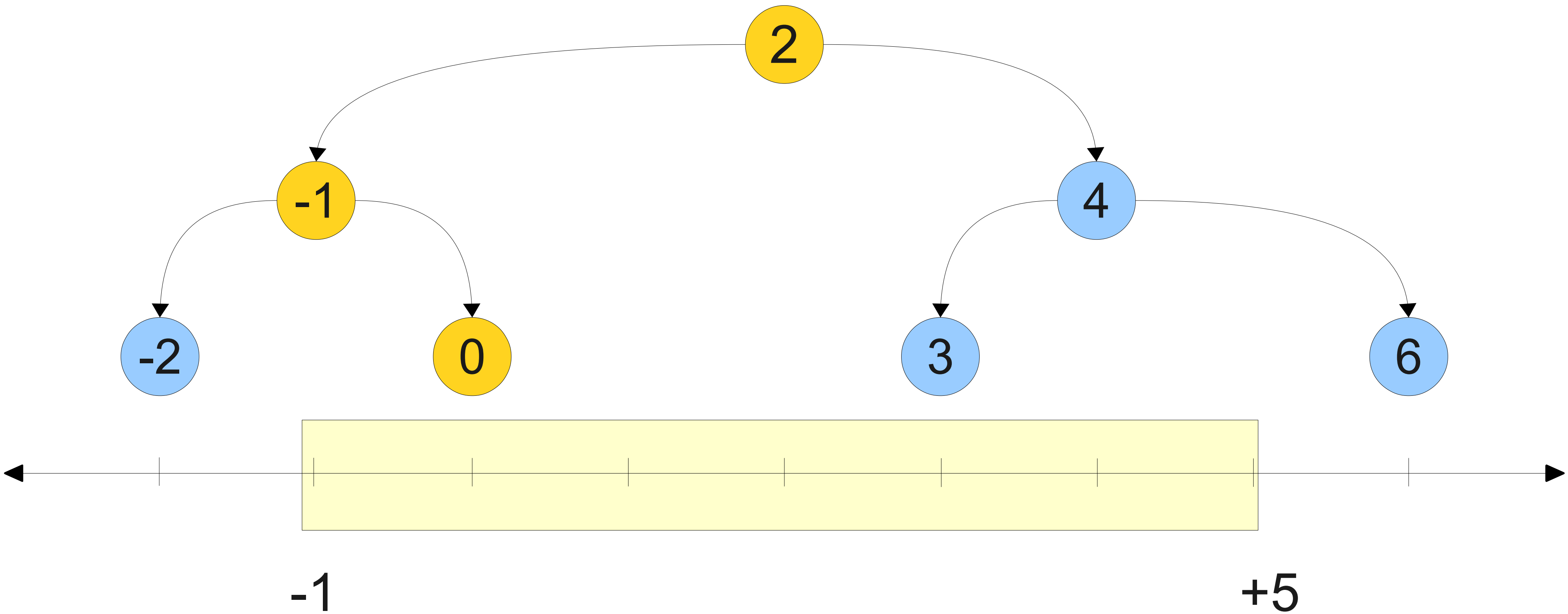
The Intuition



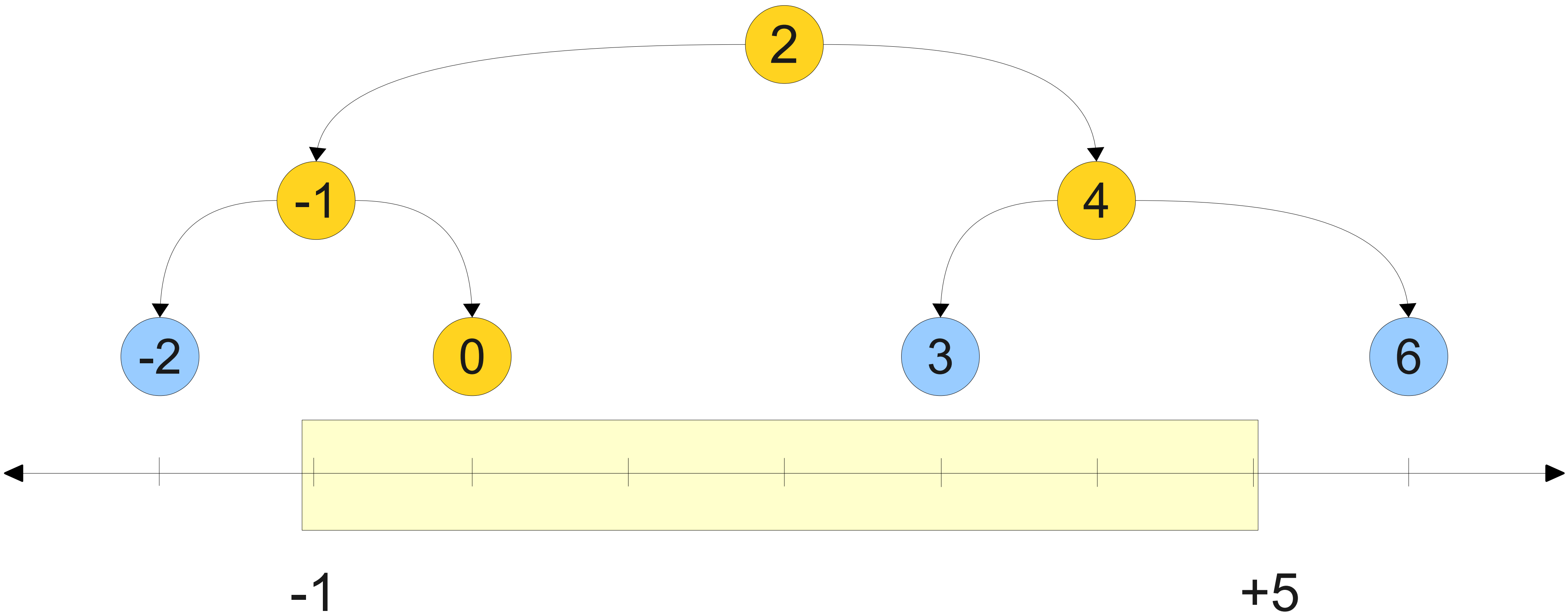
The Intuition



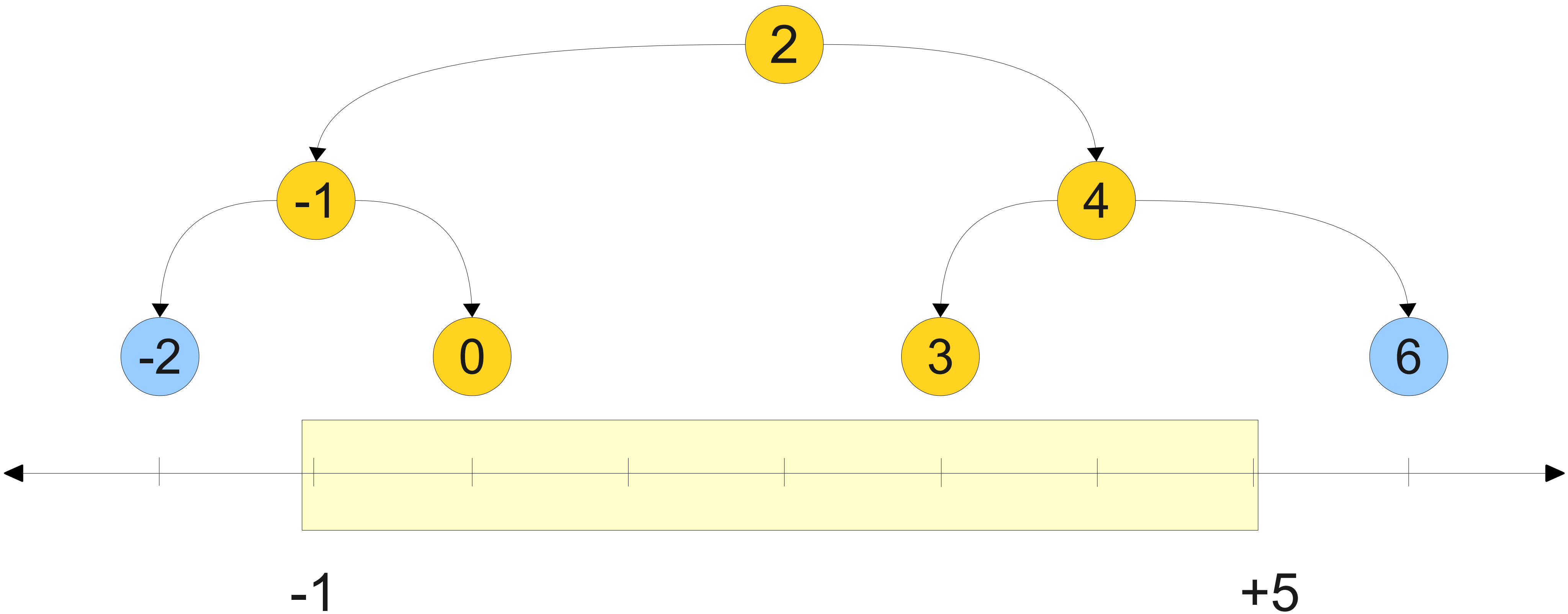
The Intuition



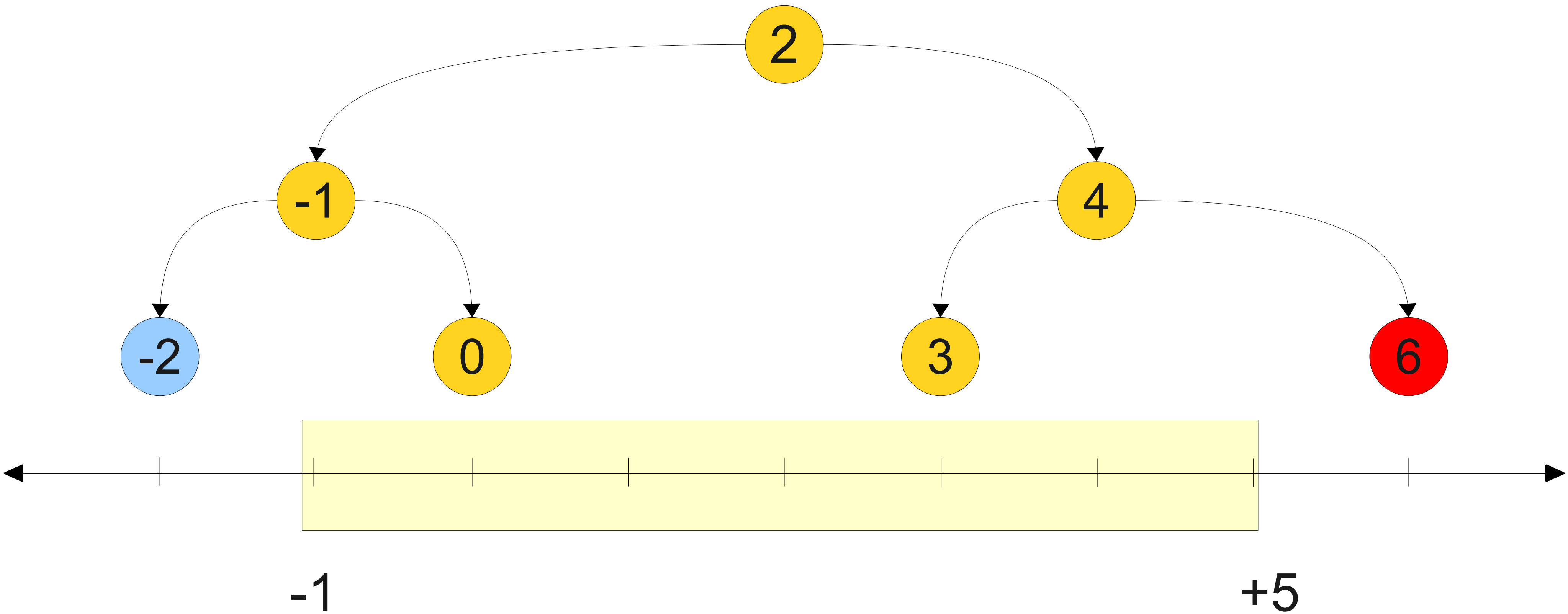
The Intuition



The Intuition



The Intuition

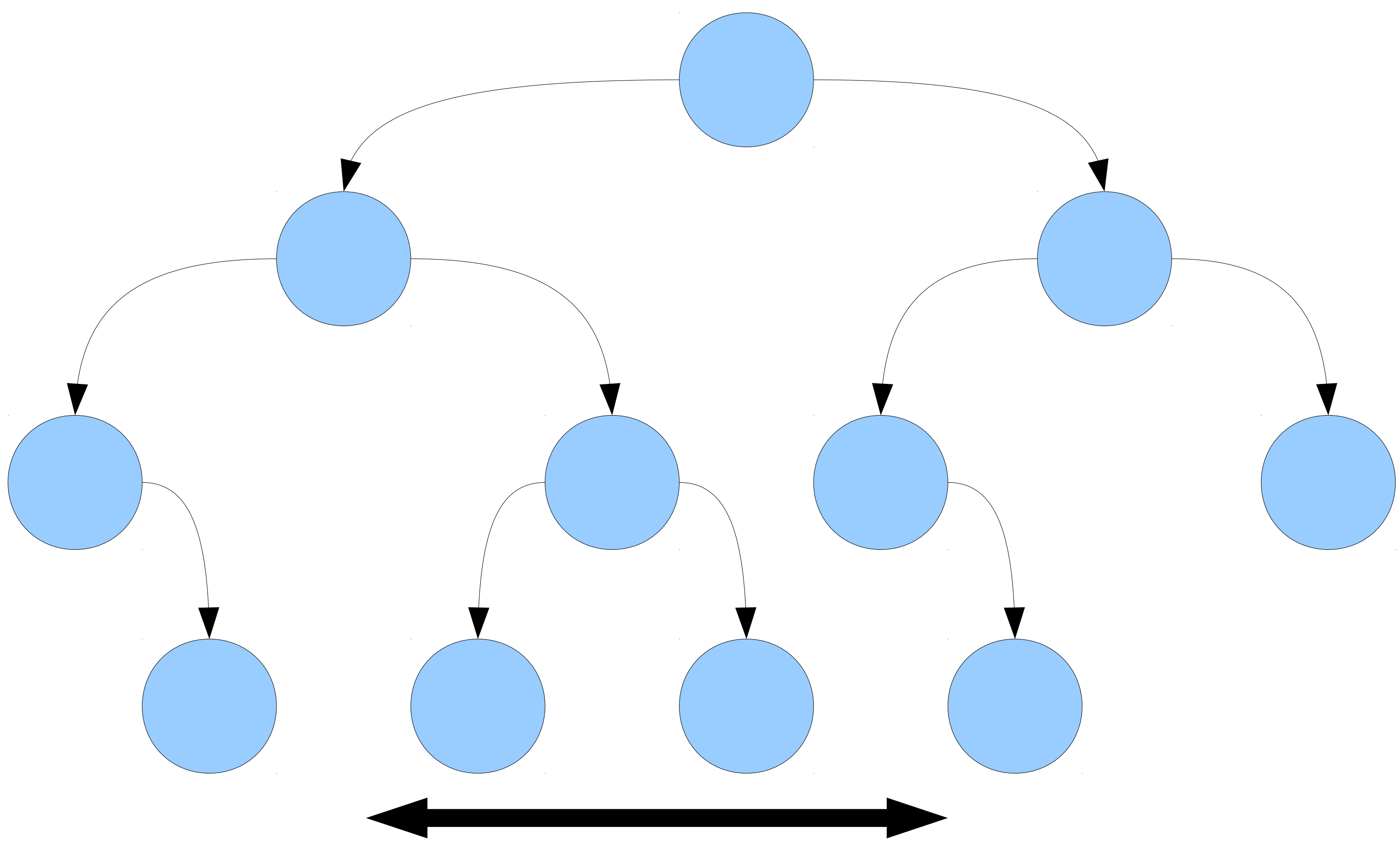


The Logic

- **Base case:**
  - The empty tree has no nodes within any range.
- **Recursive step:**
  - If this node is below the lower bound, recursively search the right subtree.
  - If this node is above the upper bound, recursively search the left subtree.
  - If this node is within bounds:
    - Search the left subtree.
    - Add this node to the output.
    - Search the right subtree.

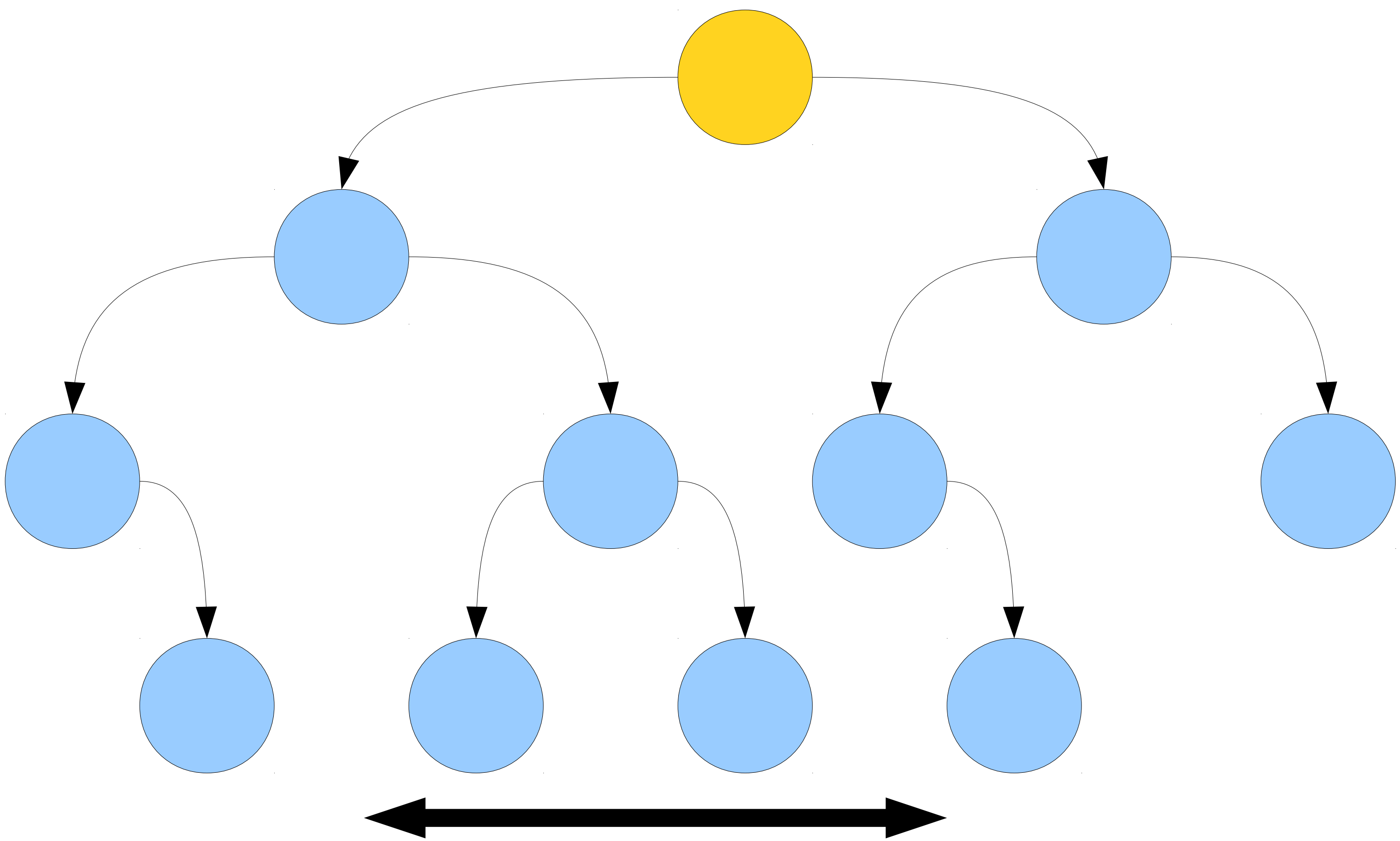
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



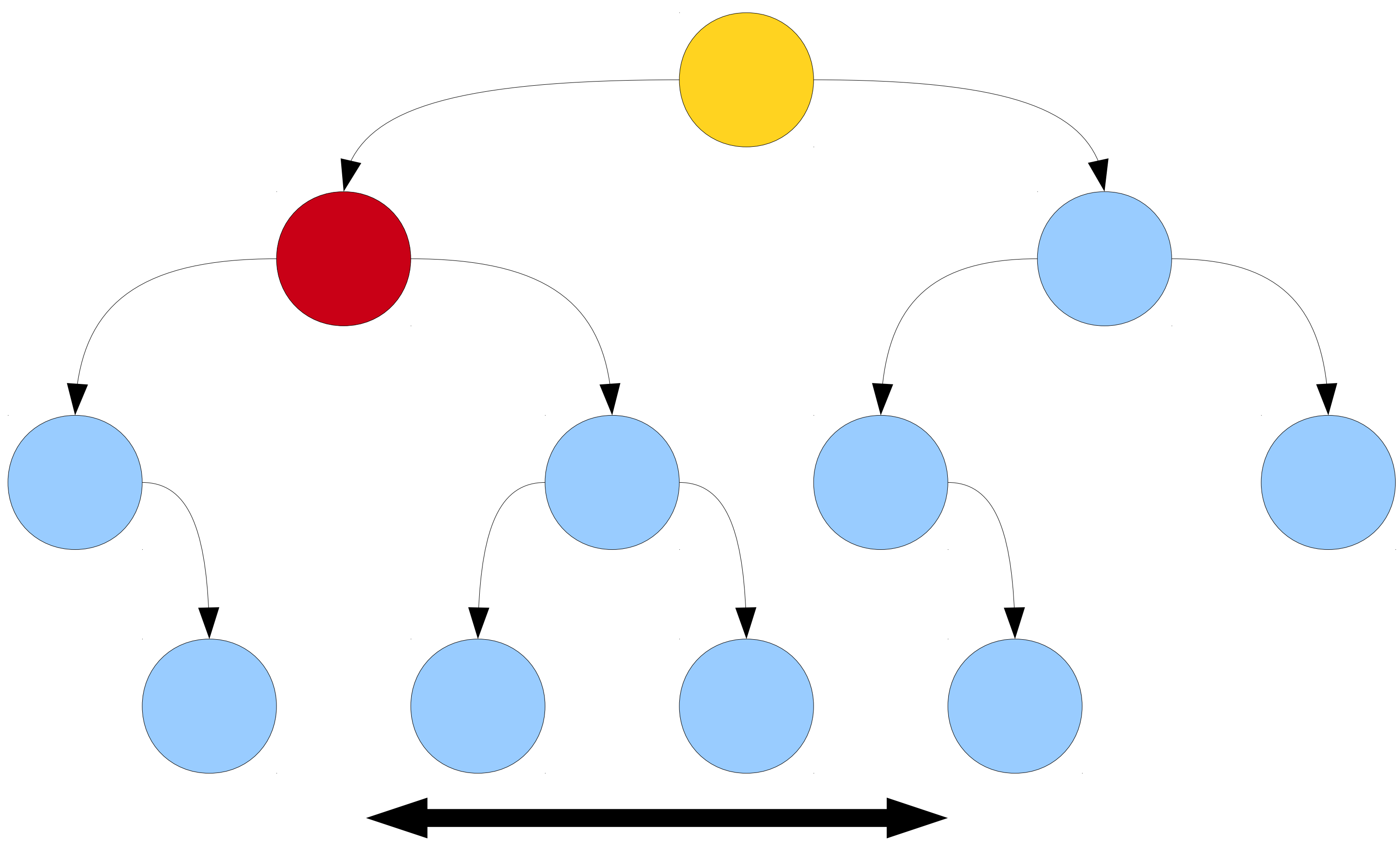
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



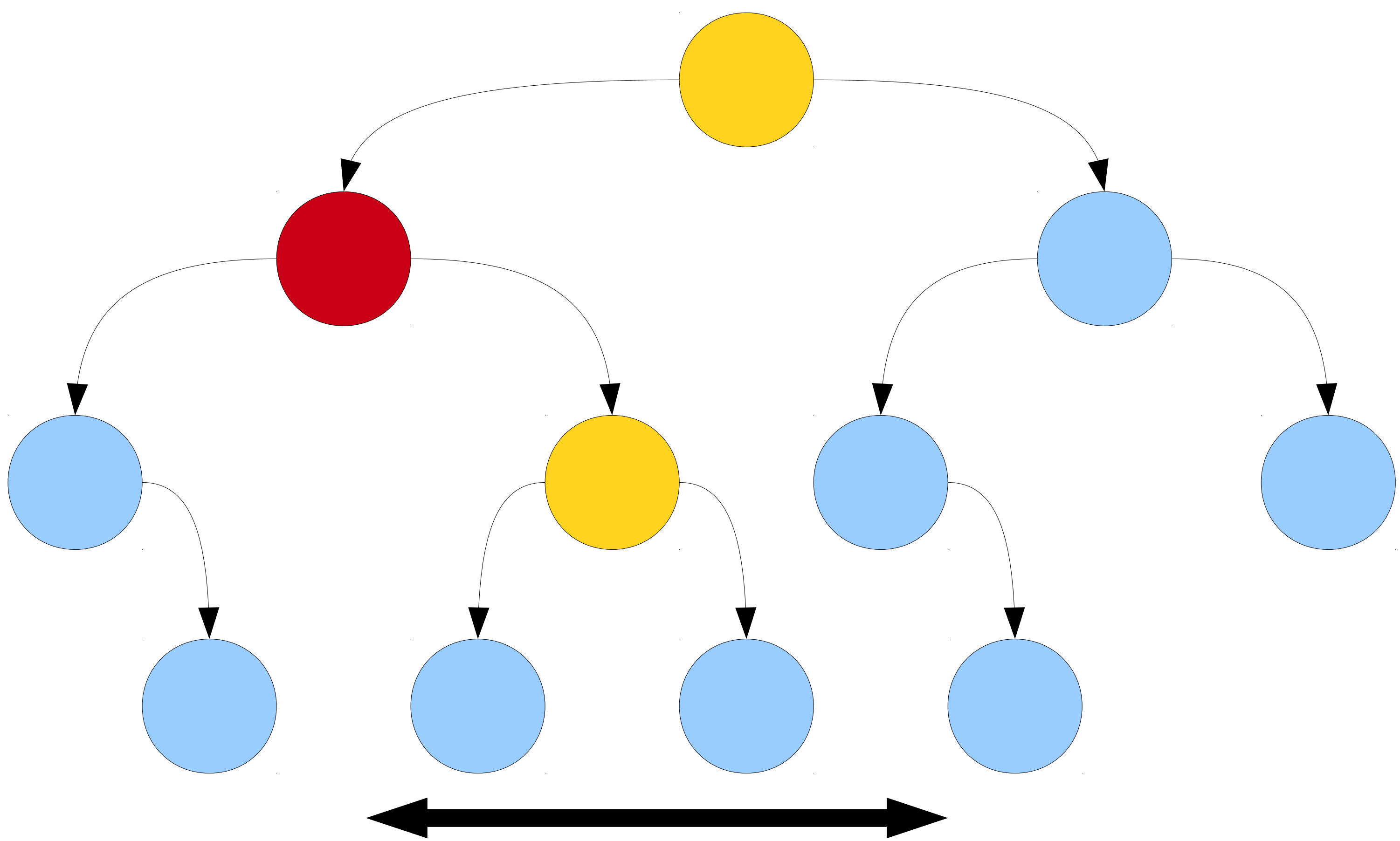
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



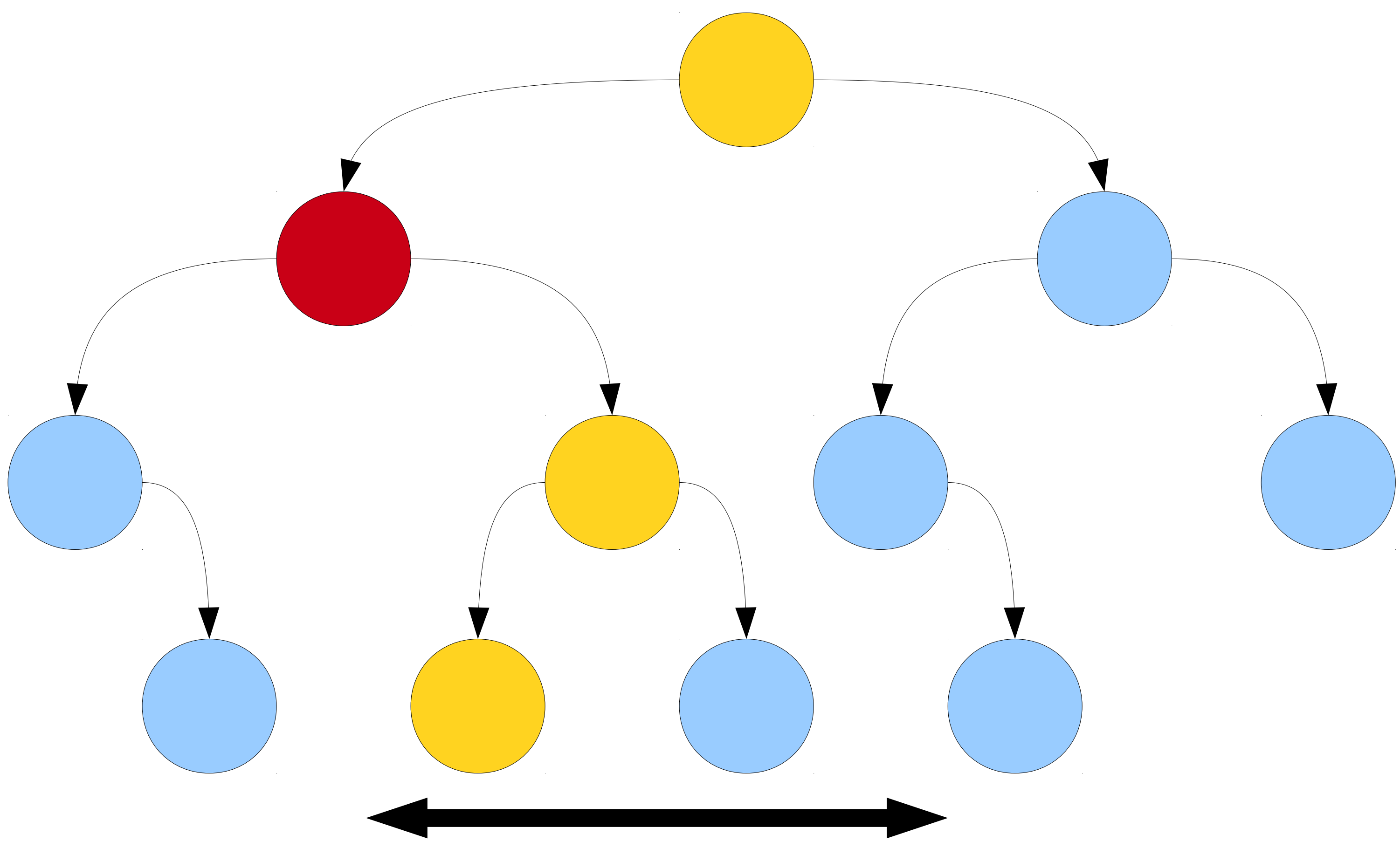
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



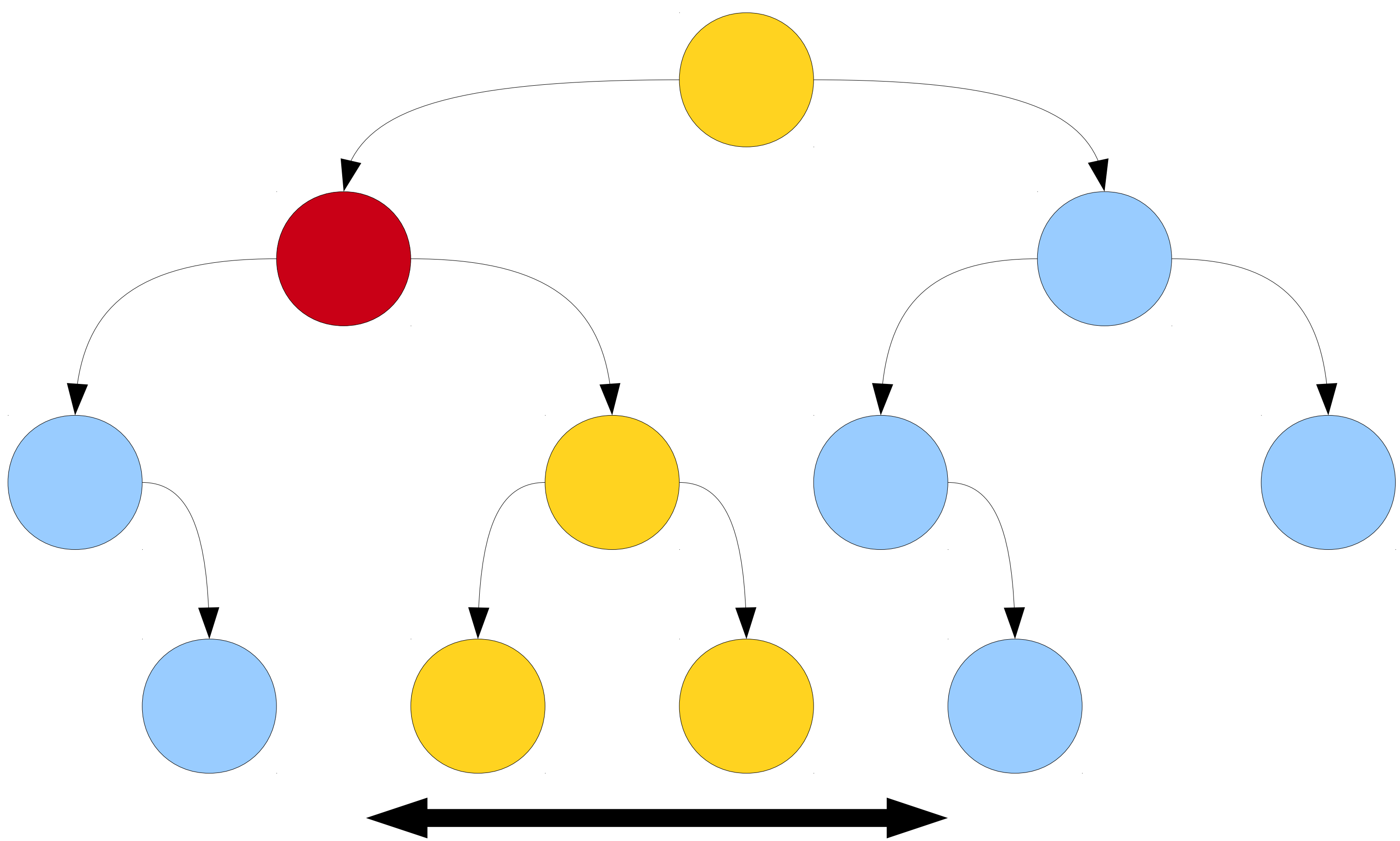
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



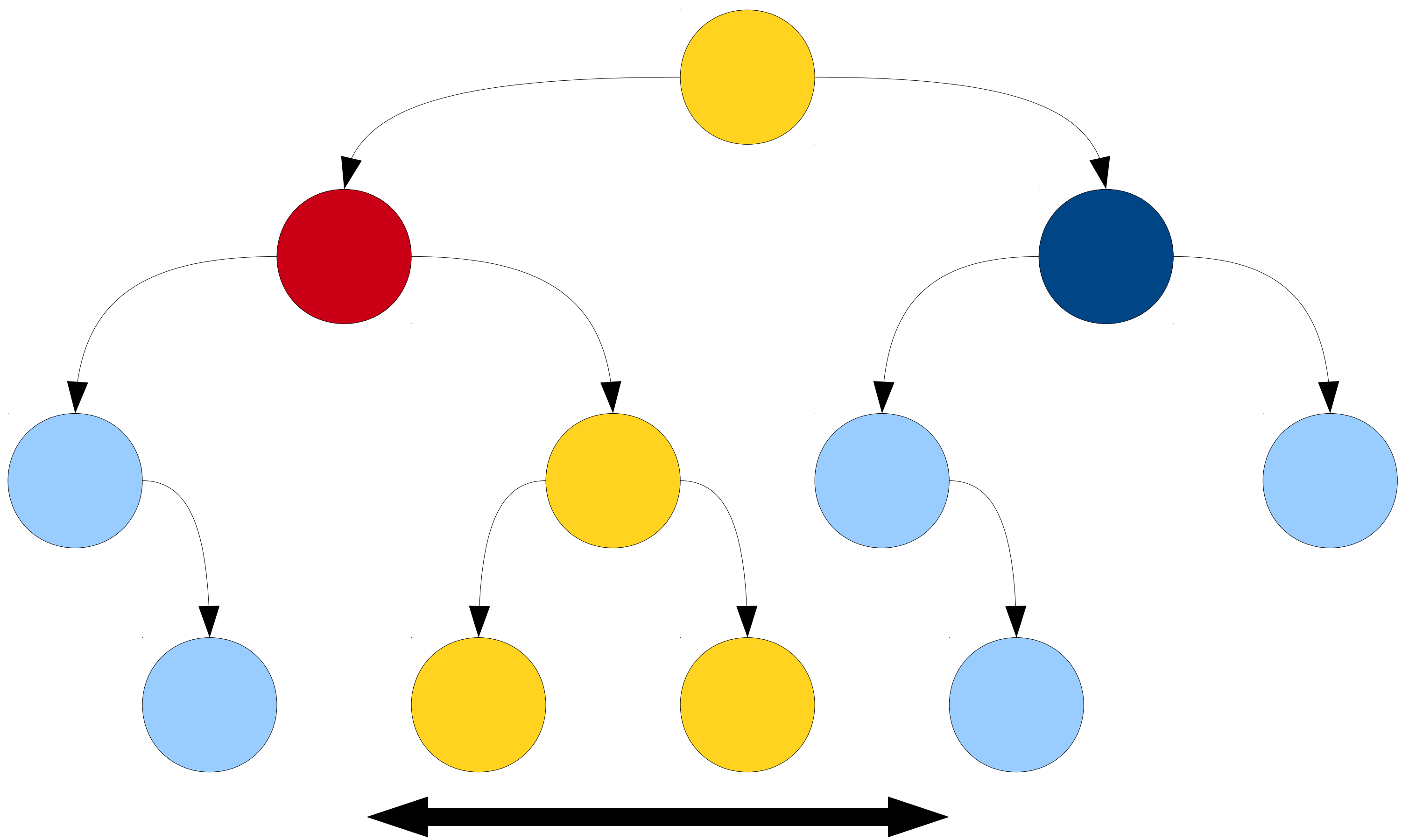
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



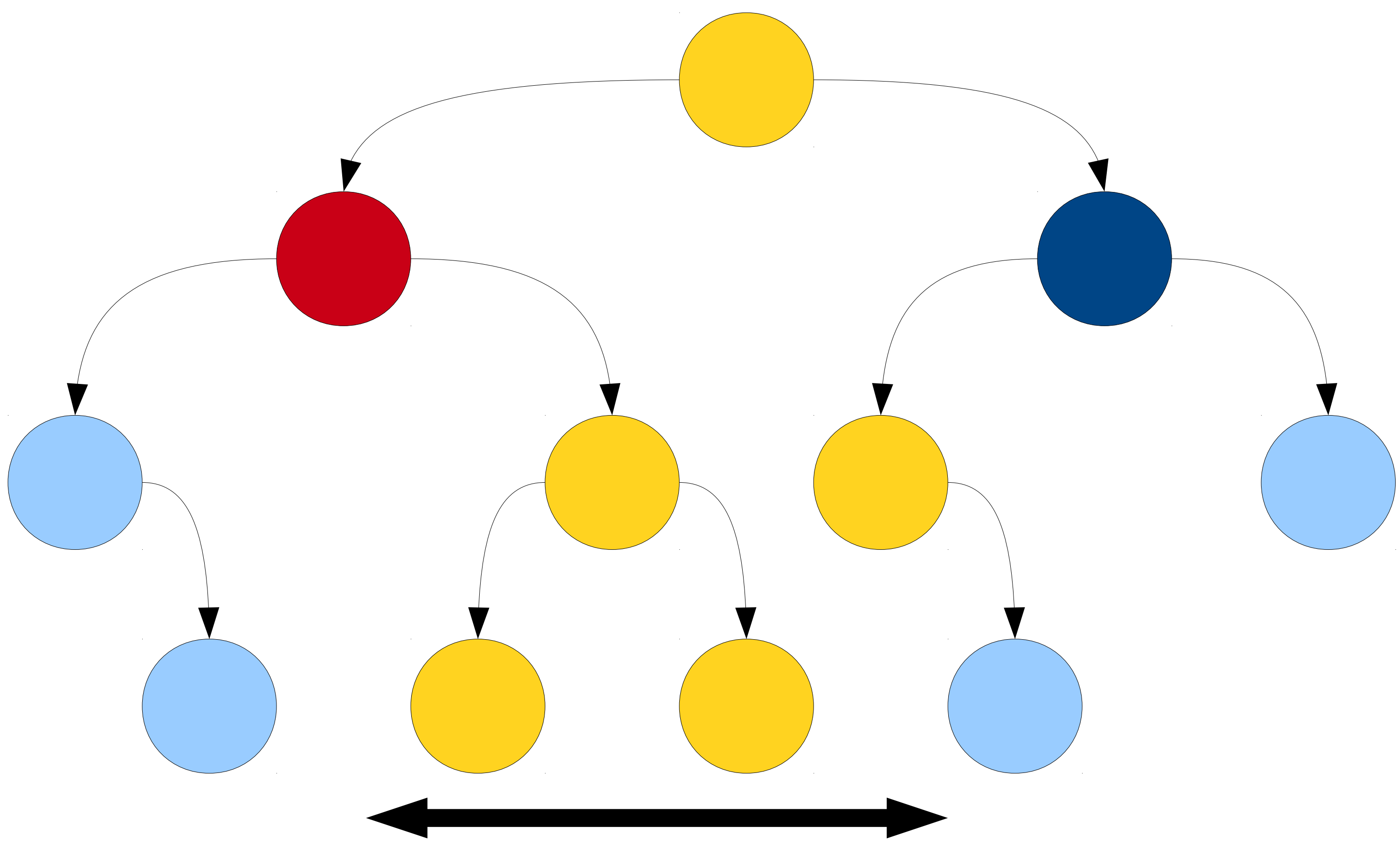
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



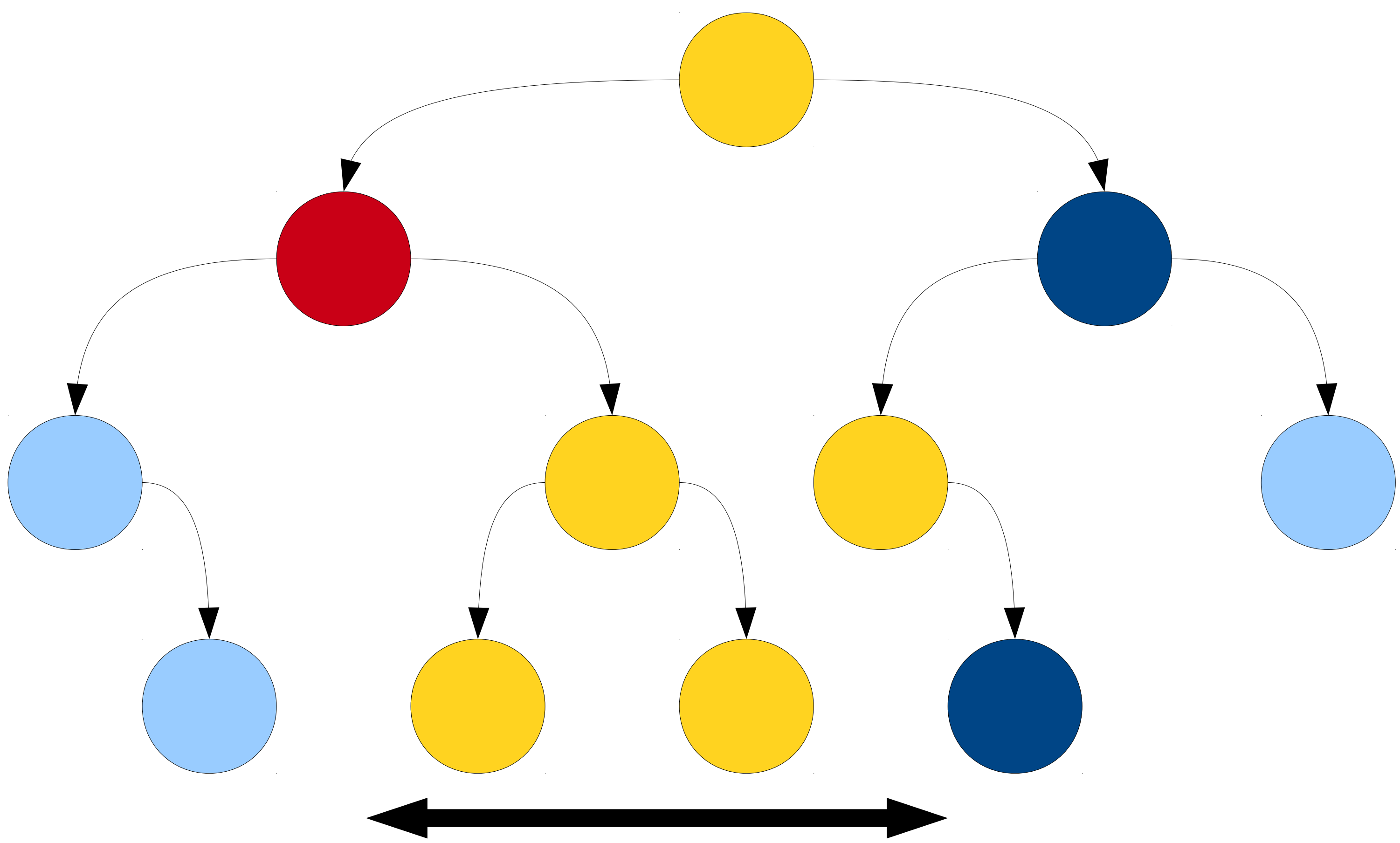
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



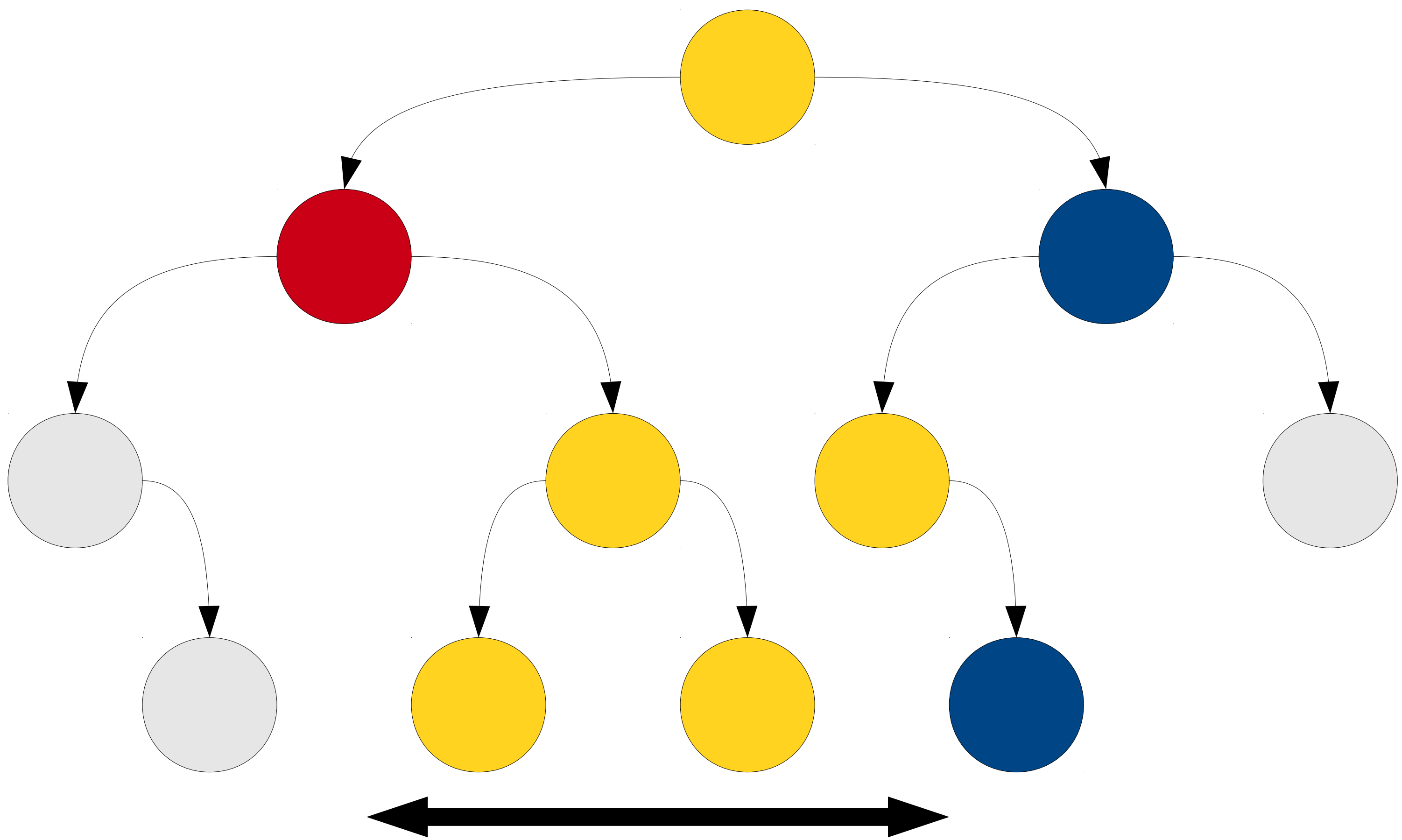
Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.



Complexity of Range Searches

- How do we get a runtime for a range search?
- Depends on how many nodes we find.
- If there are k nodes within the range, we do at least $O(k)$ work finding them.
- In addition, we have two "border sets" of nodes that are immediately outside that range. Each set has size $O(h)$, where h is the height of the tree.
- Total work done is $O(k + h)$.
- This is an **output-sensitive algorithm**.

Next Time

- **Fun With Data Structures.**
  - Balanced binary search trees.
  - Ternary search trees.
  - DAWGs.