Data Structures II

Trees

What are Trees?

An abstract hierarchical data structure of nodes.

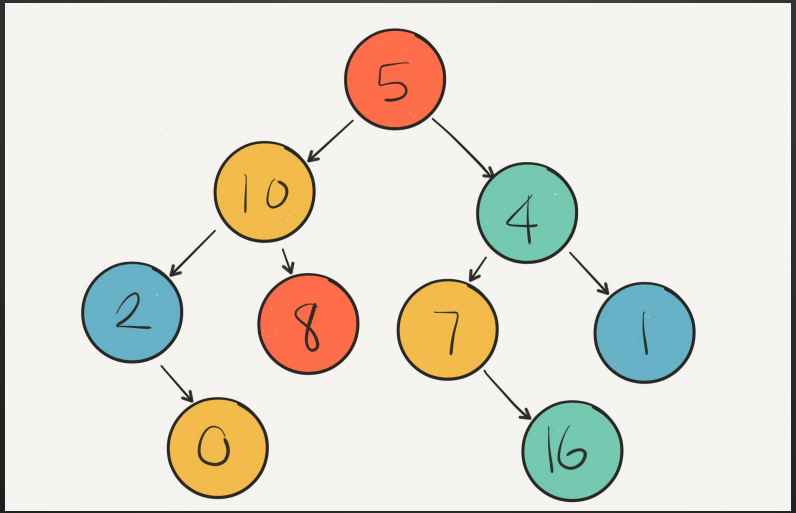
What are they used for?

Store data that are naturally organized in hierarchical structure

- File systems
- Databases
- Router systems
- Nested data: DOM

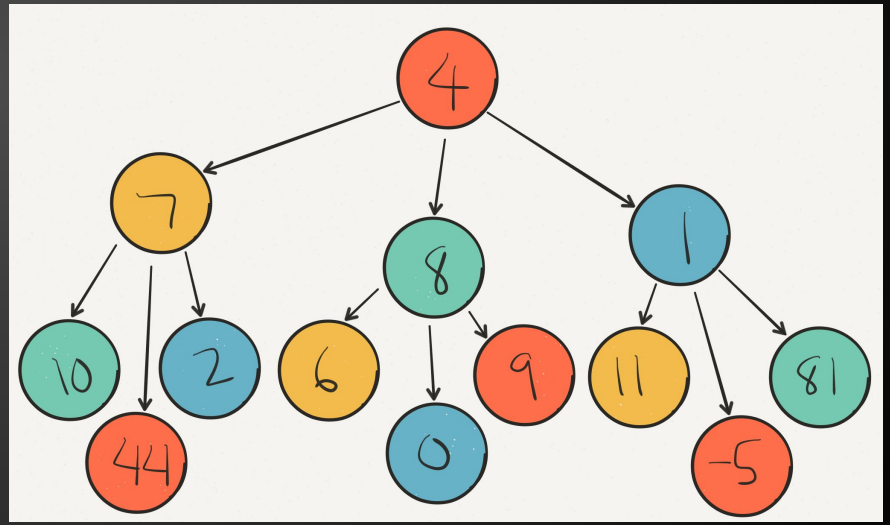
Types of Trees

- Binary trees
  - Up to 2 children nodes
  - Ex. Binary Search Trees, Binary Heaps



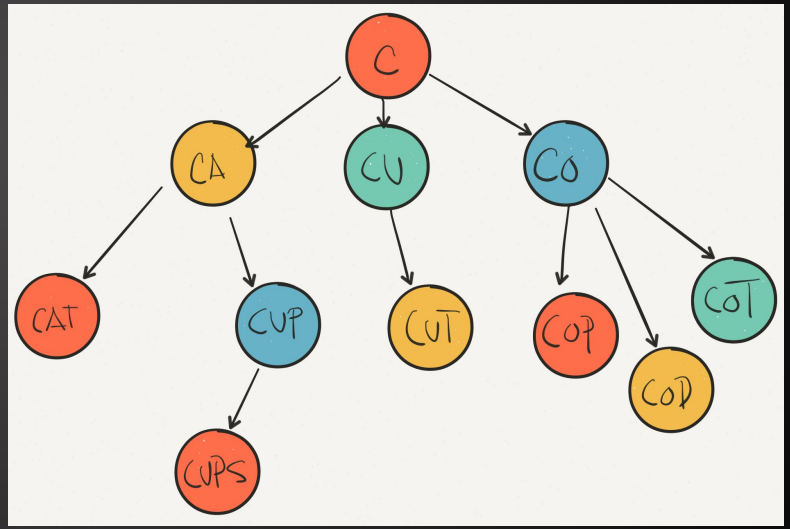
Types of Trees

- Ternary trees
  - Up to 3 children nodes
- K-nary trees
  - Up to K children nodes



Types of Trees

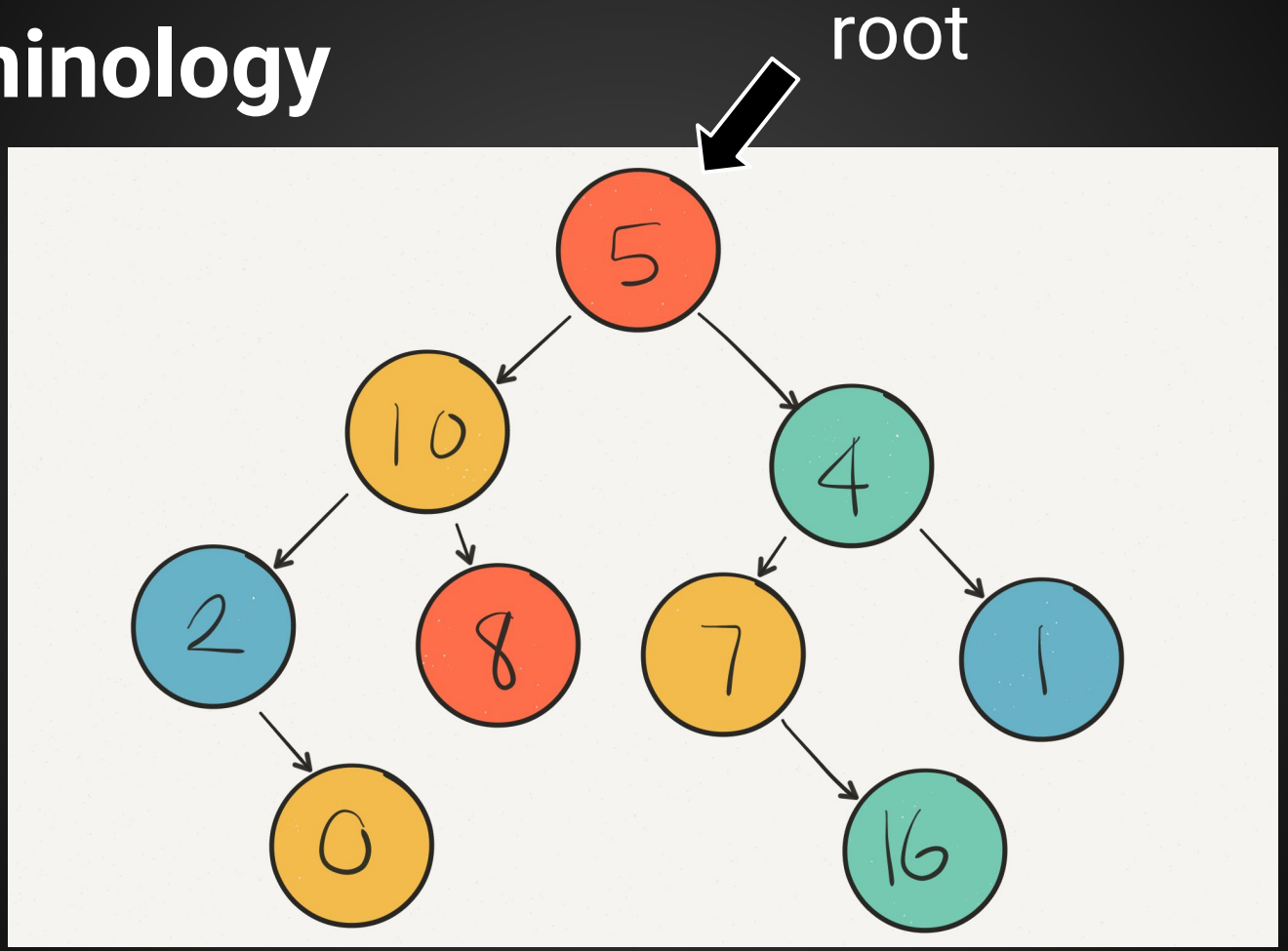
- Tries (digital/prefix tree)
  - Typically strings
  - Allows search by prefixes
- Balanced search tree
  - B-trees
  - AVL Trees
  - Red-black Trees



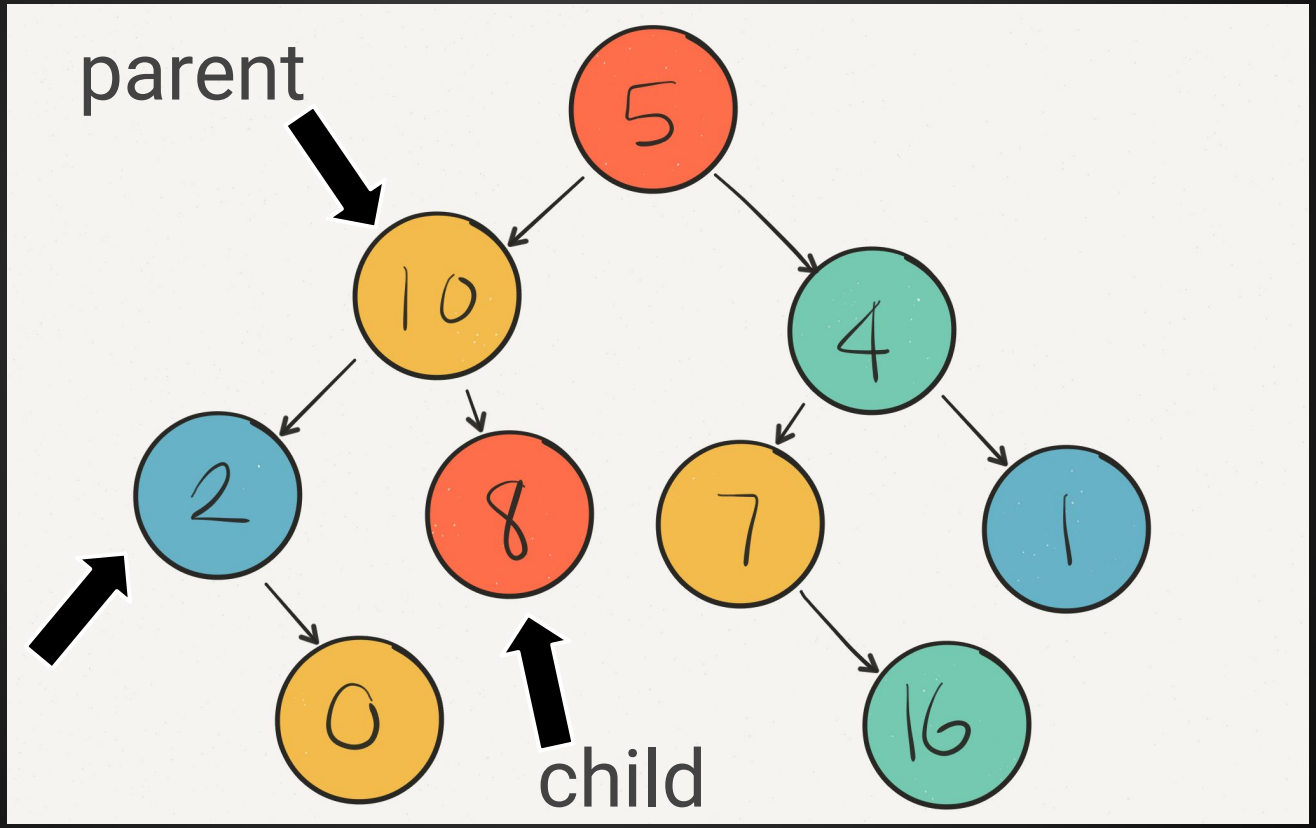
Focus on Binary Trees

- Most commonly used
- Provides foundational understanding trees
- In depth dive
  - Binary Trees
  - Binary Search Trees (BST)
  - Binary Heaps
- Know other types at a high-level

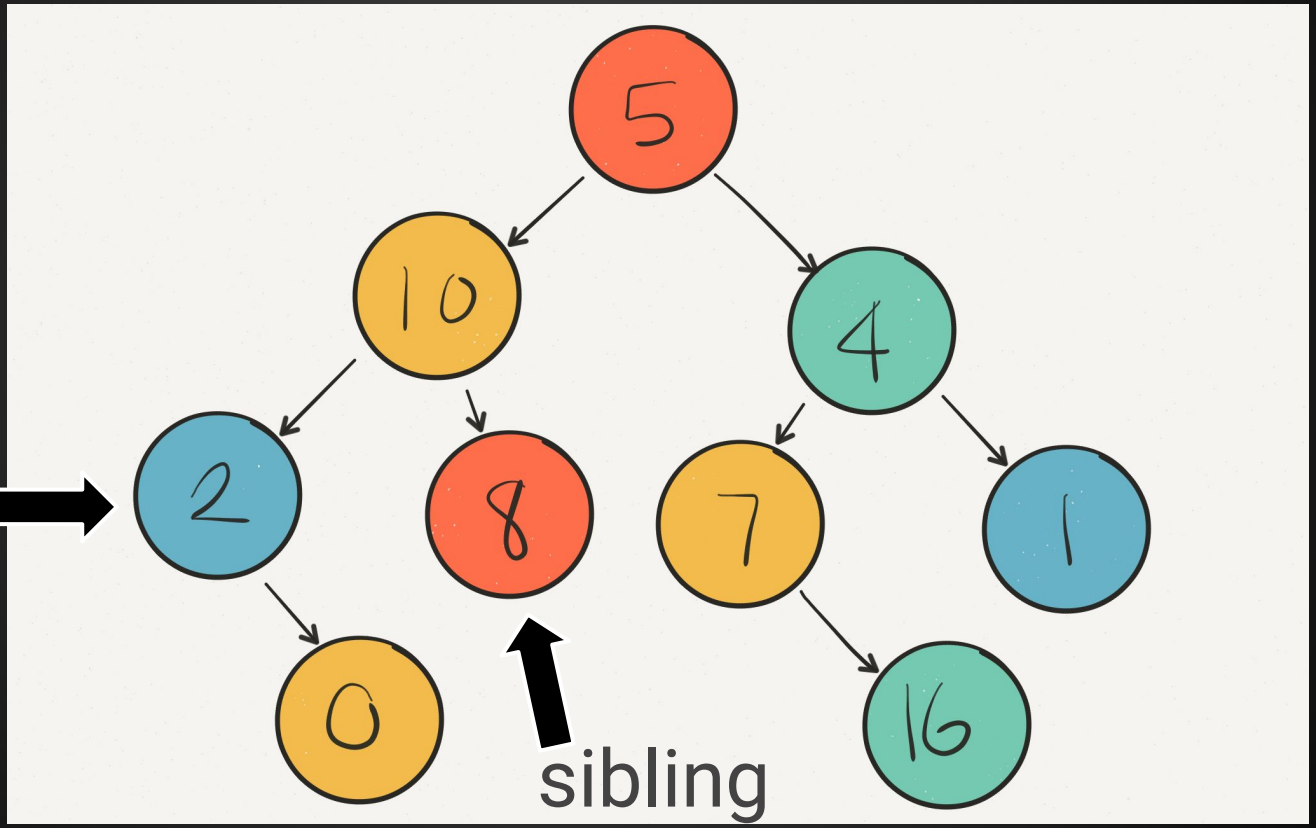
Terminology



Terminology

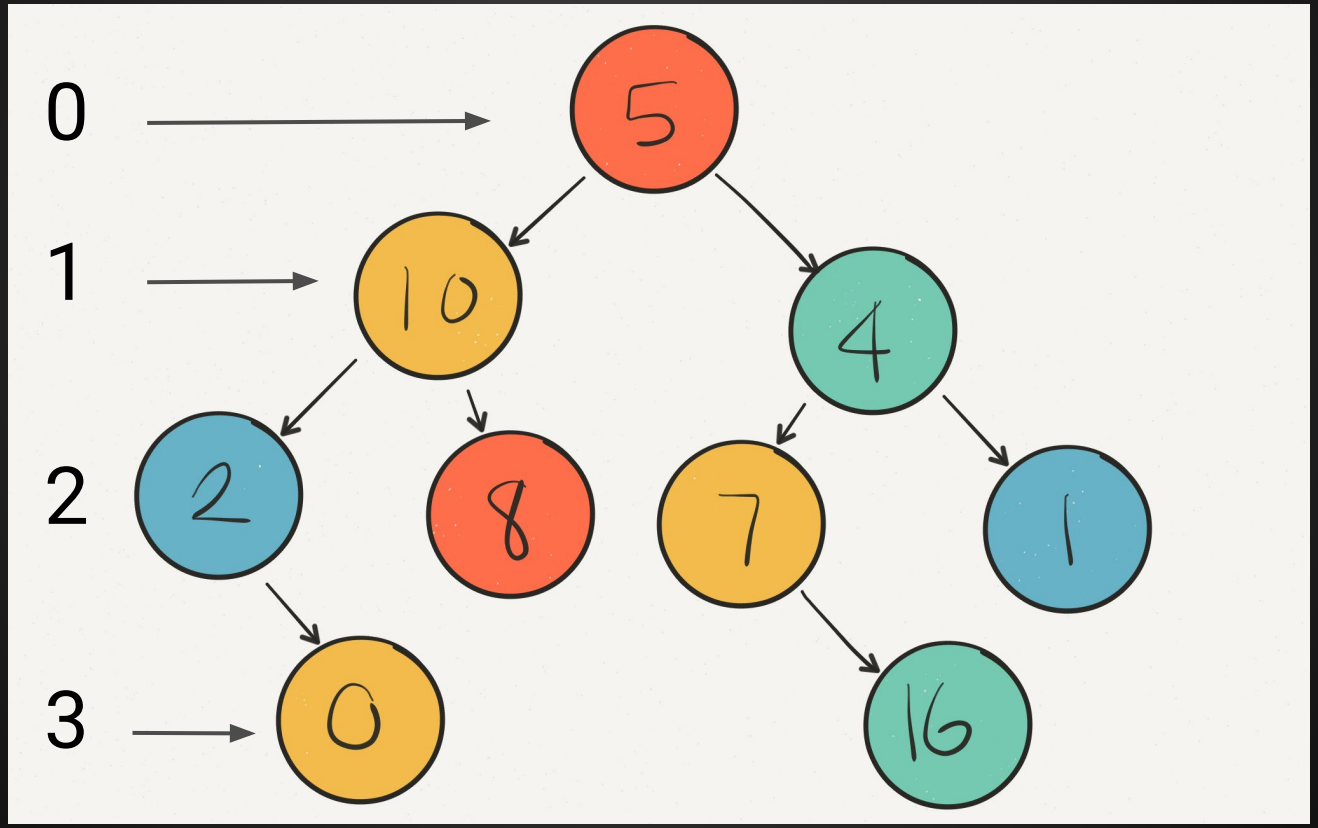


Terminology

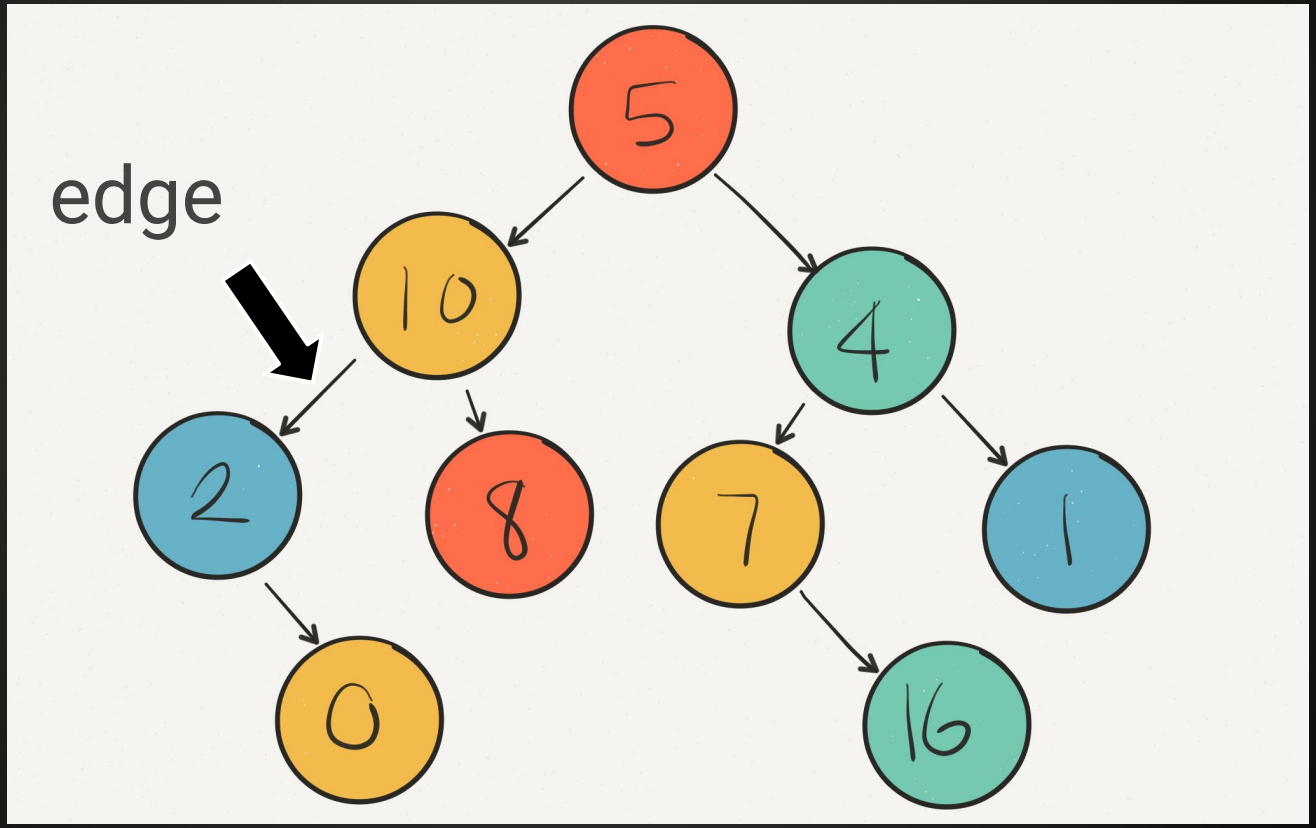


Terminology

depth



Terminology



degree:
no. of
edges

Binary Tree Attributes

- Nodes
  - Value
  - Children (up to 2 nodes for binary tree)
    - left child
    - right child
- Properties
  - Root
- Common methods
  - Insert, Search, Remove

Node class for binary tree

```ruby
class Node
  def initialize(val)
    @val = val
    @left_child = nil
    @right_child = nil
  end
  attr_accessor :val
  attr_accessor :left_child
  attr_accessor :right_child
end
```

initialize method
takes in a value

Node class for binary tree

```ruby
class Node
  def initialize(val)
    @val = val
    @left_child = nil
    @right_child = nil
  end
  attr_accessor :val
  attr_accessor :left_child
  attr_accessor :right_child
end
```

assigns value to a property called val

Node class for binary tree

```ruby
class Node
  def initialize(val)
    @val = val
    @left_child = nil
    @right_child = nil
  end
  attr_accessor :val
  attr_accessor :left_child
  attr_accessor :right_child
end
```

left and right children are empty

Node class for binary tree

```ruby
class Node
  def initialize(val)
    @val = val
    @left_child = nil
    @right_child = nil
  end
  attr_accessor :val
  attr_accessor :left_child
  attr_accessor :right_child
end
```

ruby specific, allows to be read and modified later

Tree class for binary tree

```
class Tree
  def initialize()
    @root = nil
  end
  attr_reader :root

  def insert()
  end
  def search()
  end
end
```

instantiate empty root

Tree class for binary tree

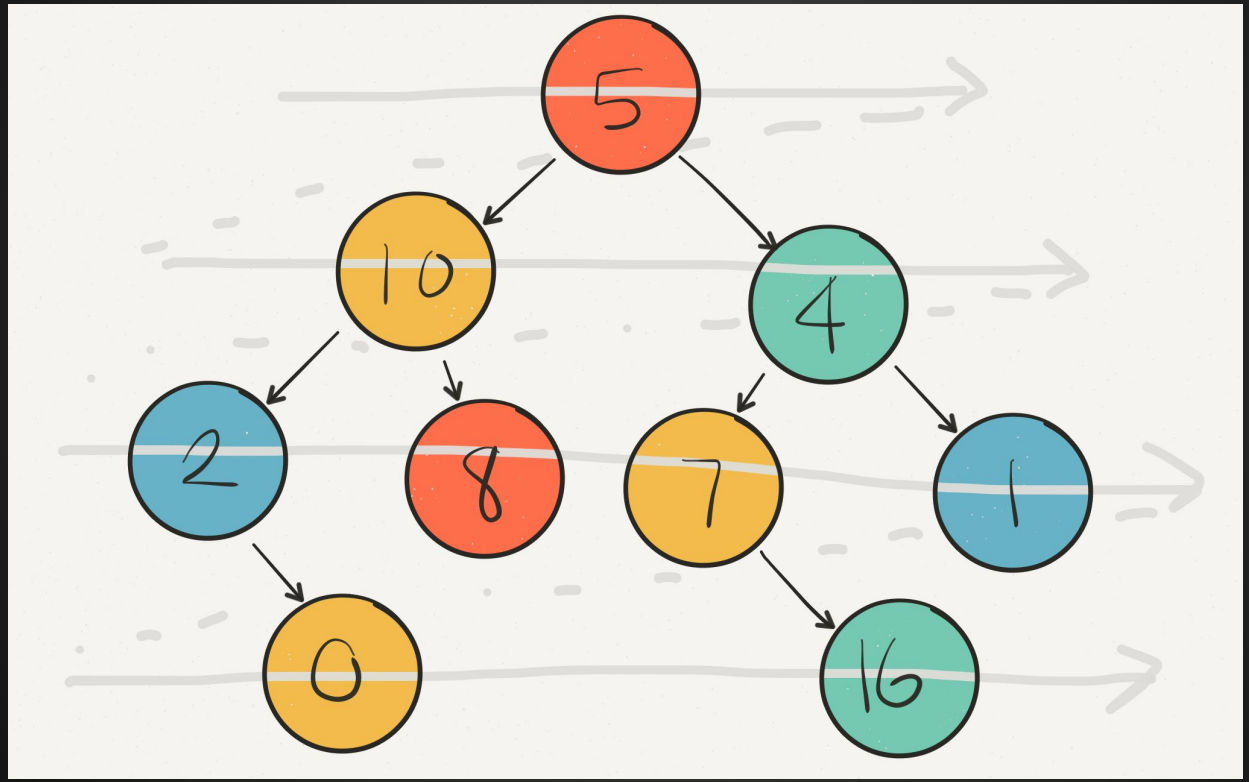
```
class Tree
  def initialize()
    @root = nil
  end
  attr_reader :root

  def insert()
  end
  def search()
  end
end
```

allows root to be read

Tree Traversal

- Breadth first search
- Depth first search
  - Pre-order
  - In-order
  - Post-order

Breadth First Search

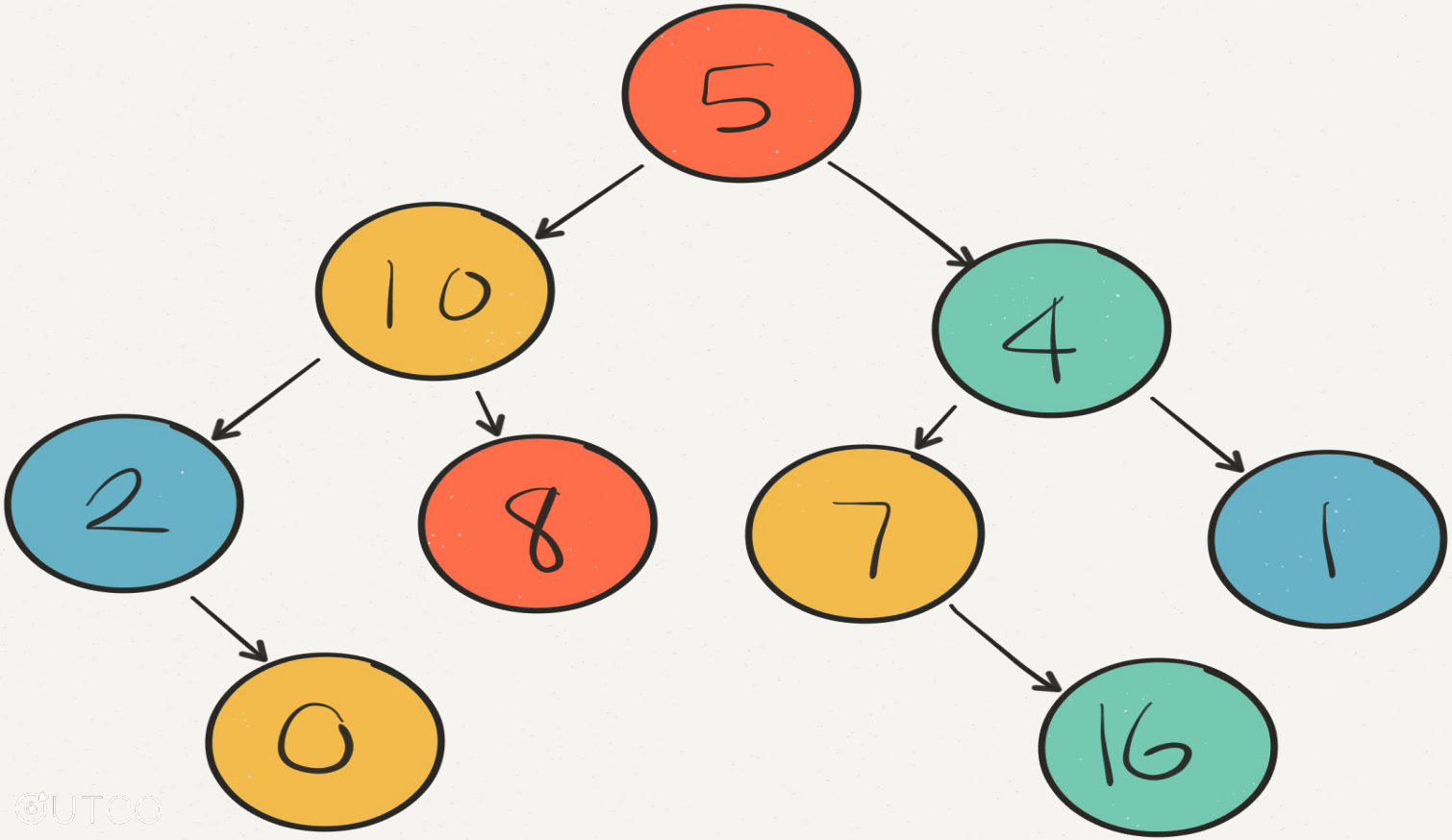


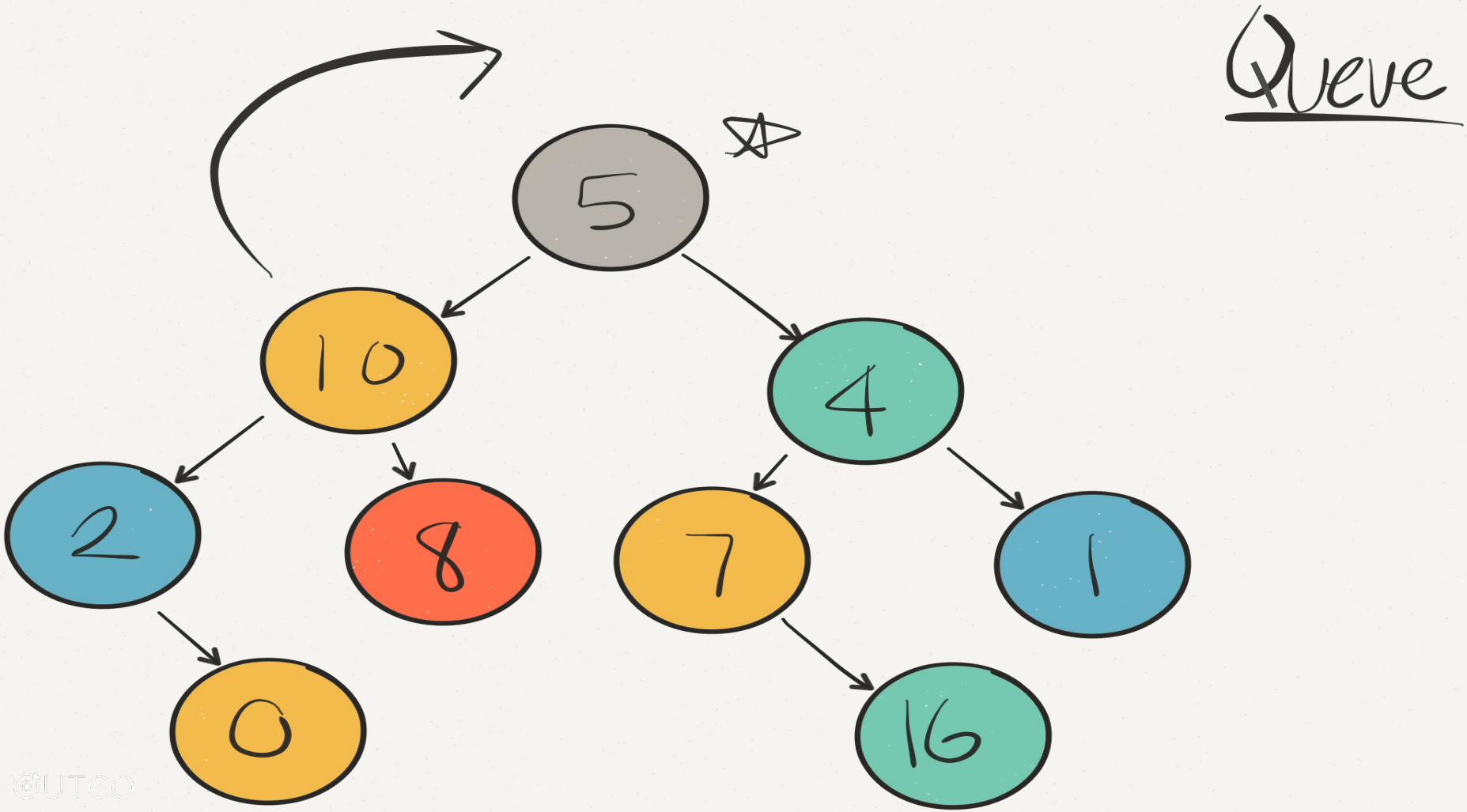
Breadth First Search

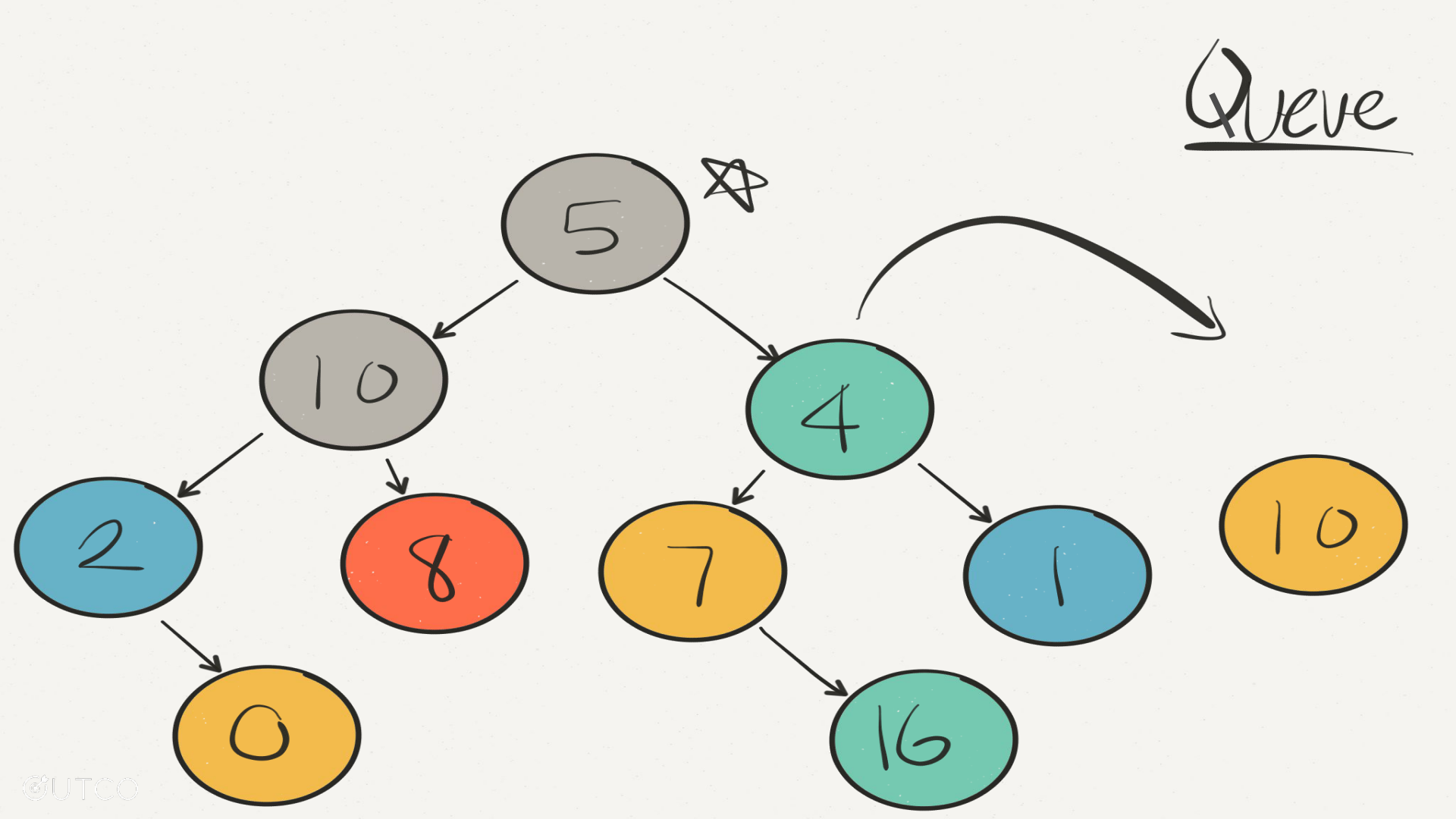
Use a Queue

- Start at root node
- Add all children into queue first
- Then execute code on the current node
- Then get the next item on the queue and repeat

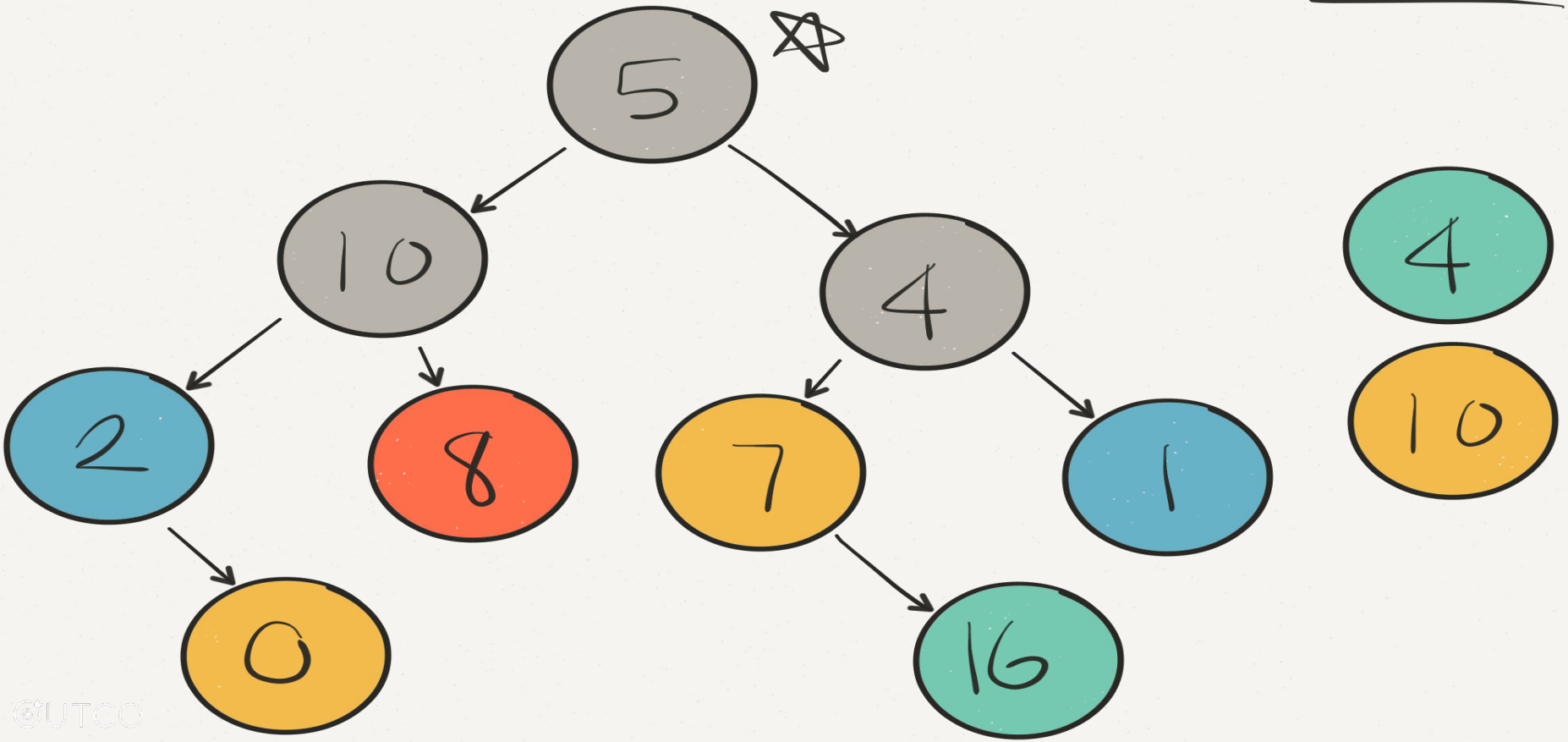
Queve



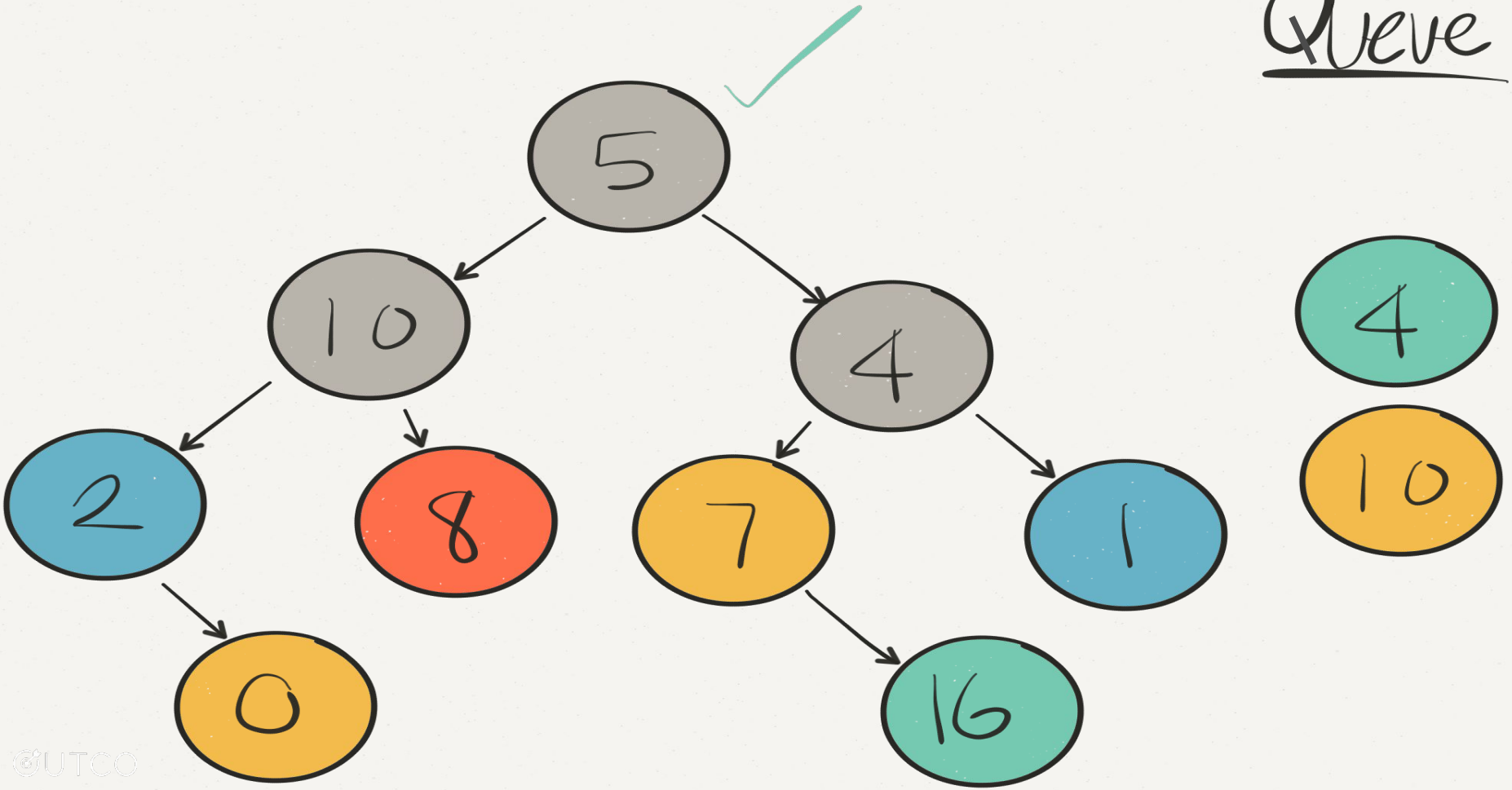

Queue
5
10
4
2
8
7
1
0
16

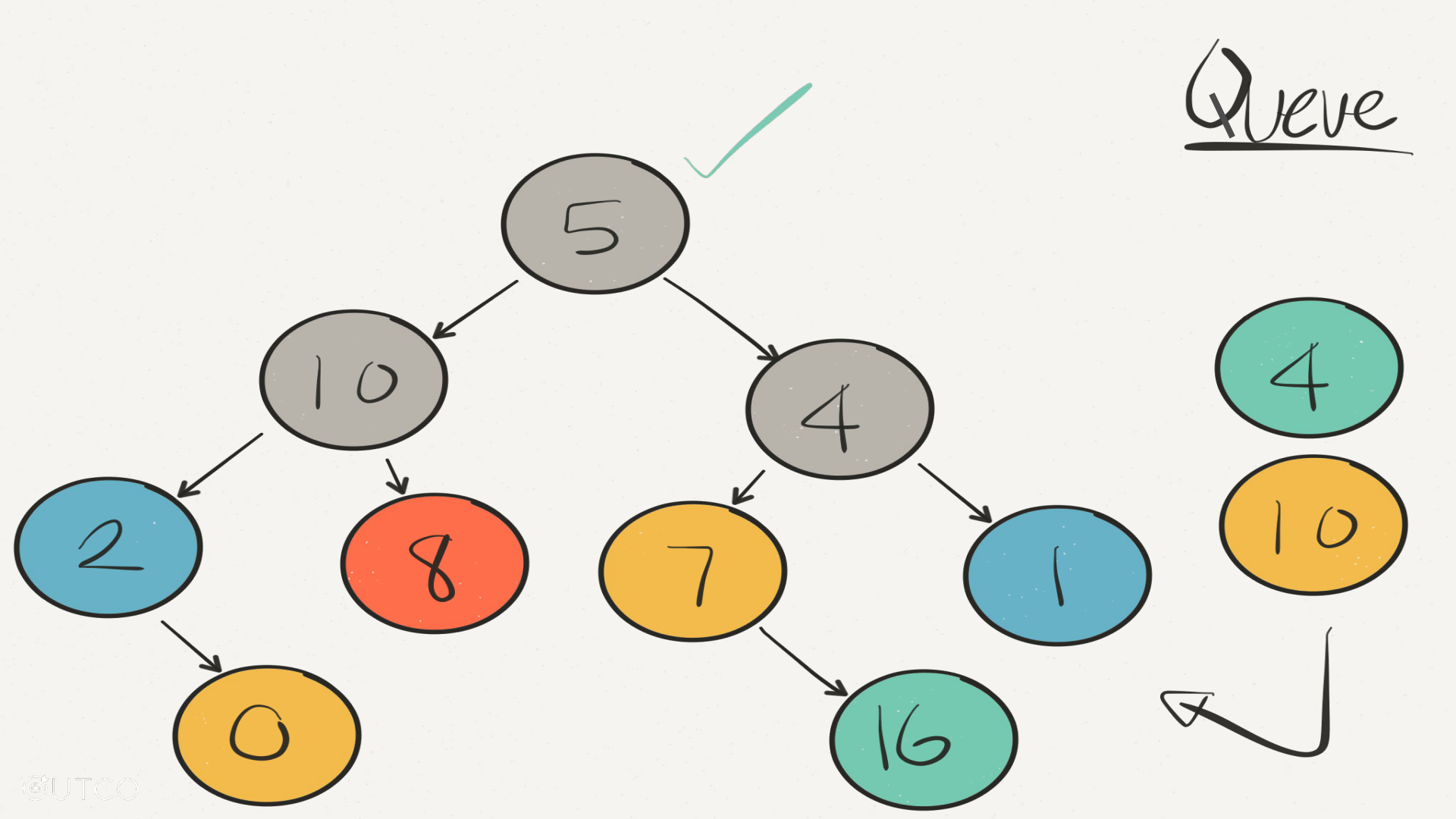

Queue
5
10
4
2
8
7
1
10
0
16

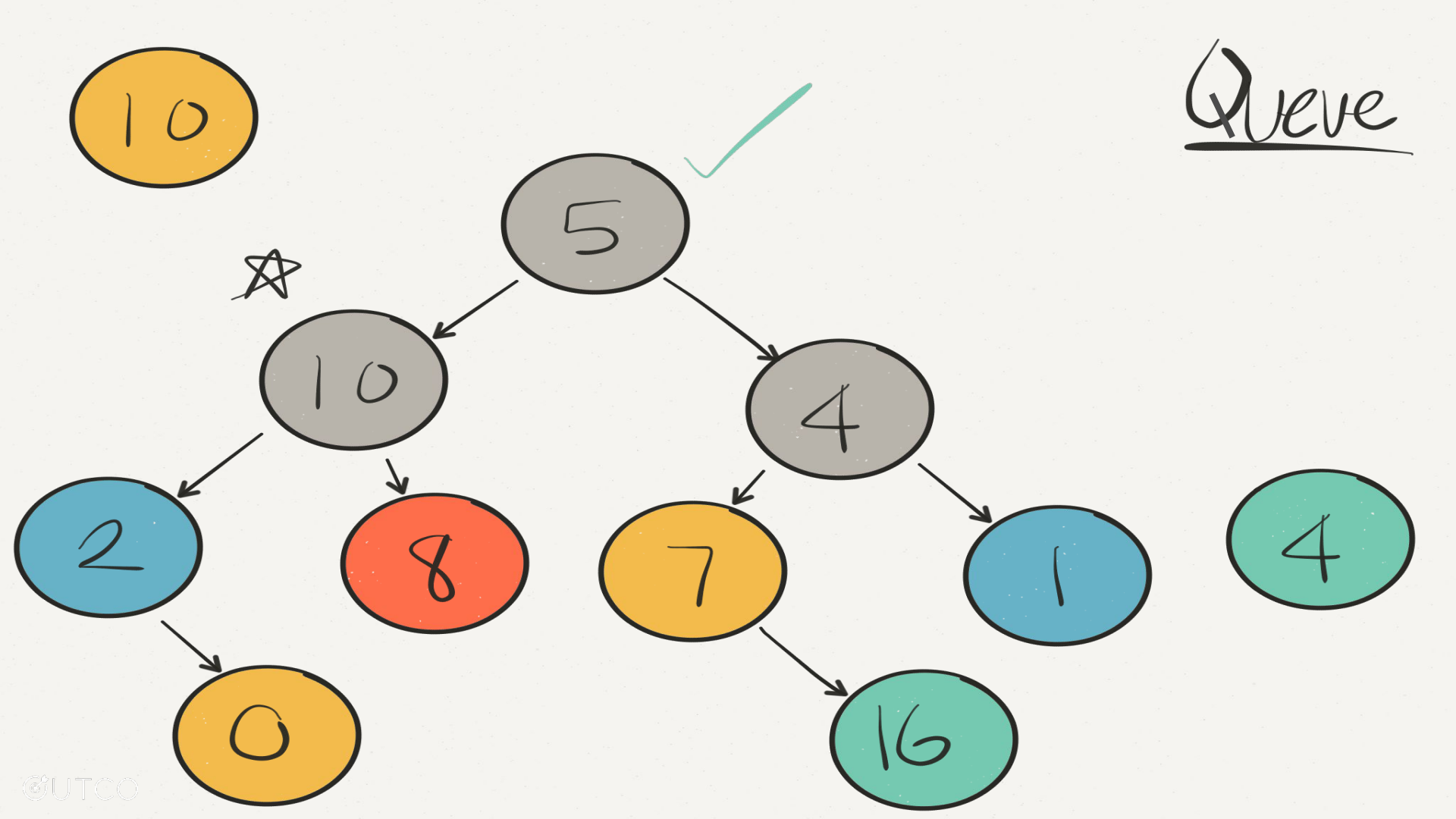
Queve

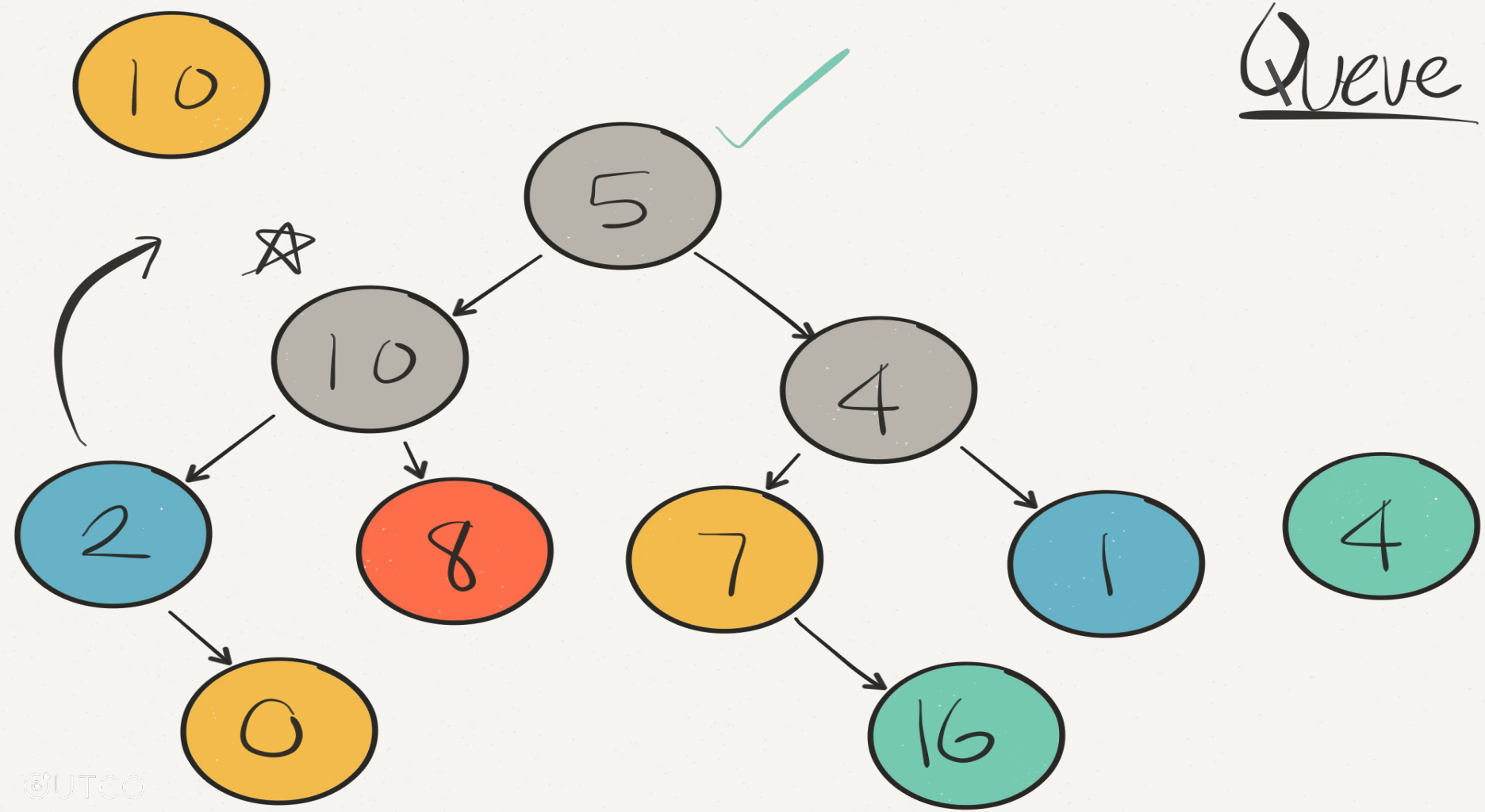


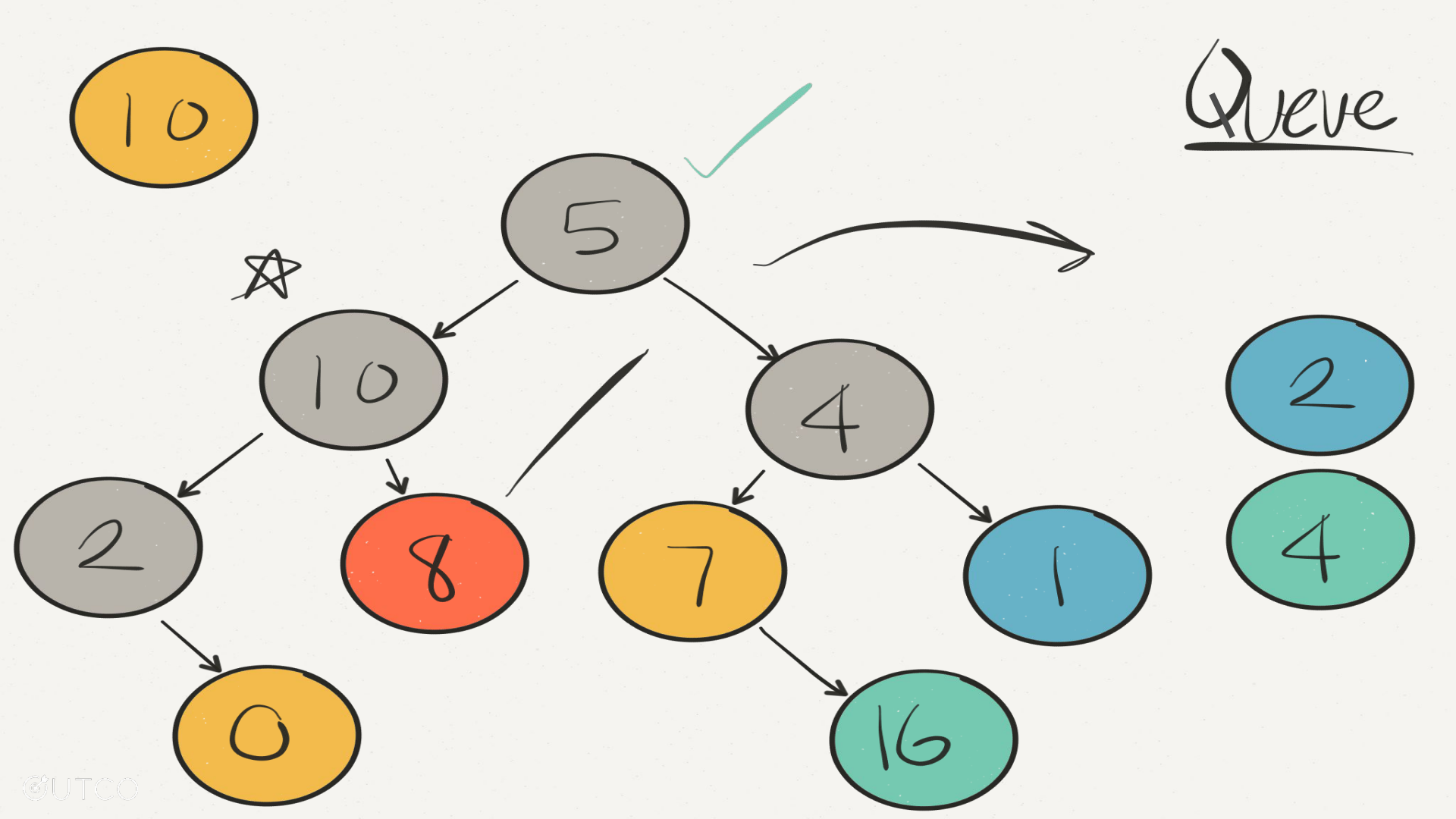
Queve

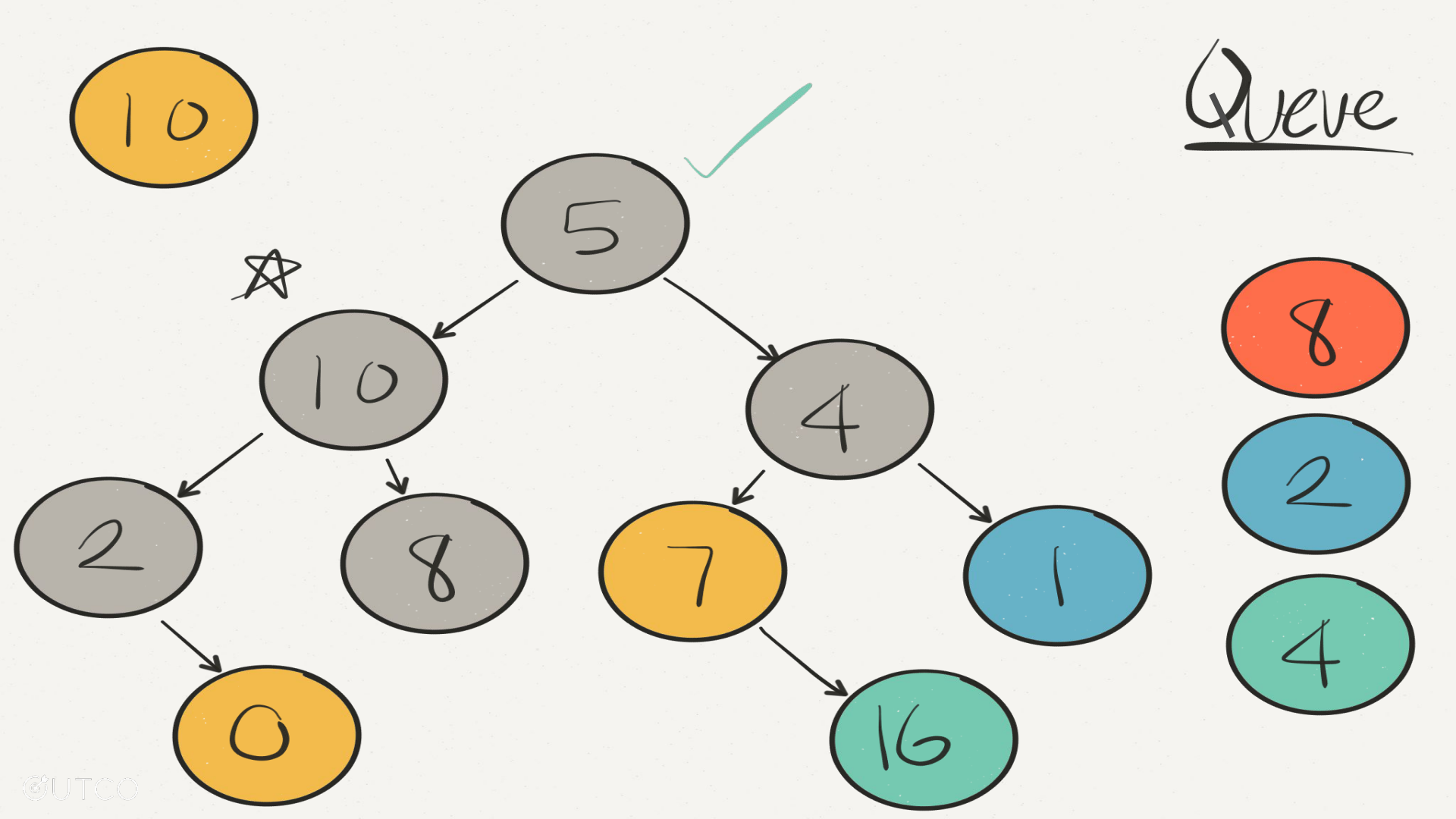


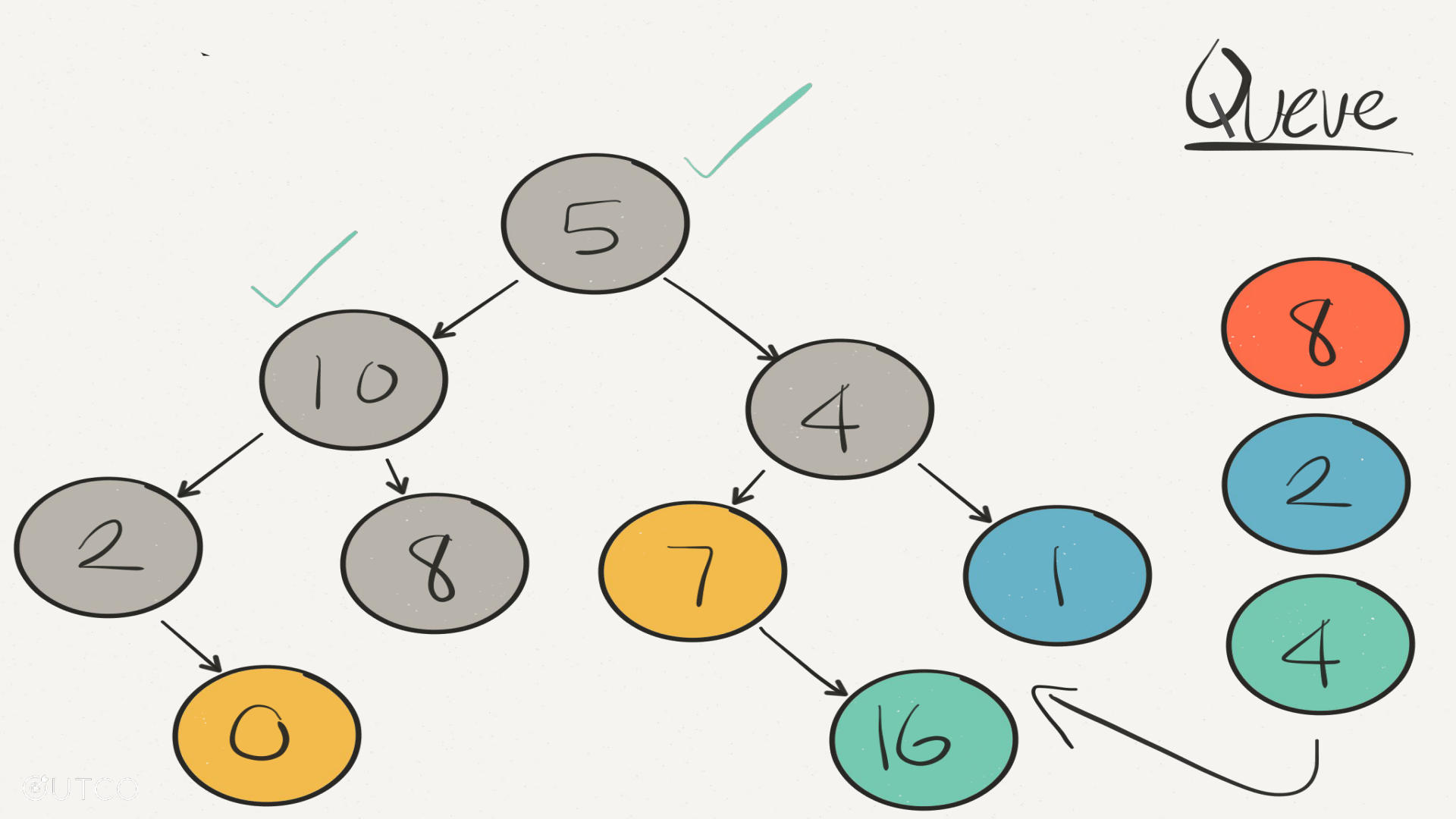

Queve
5
10
4
2
8
7
1
0
16
4
10

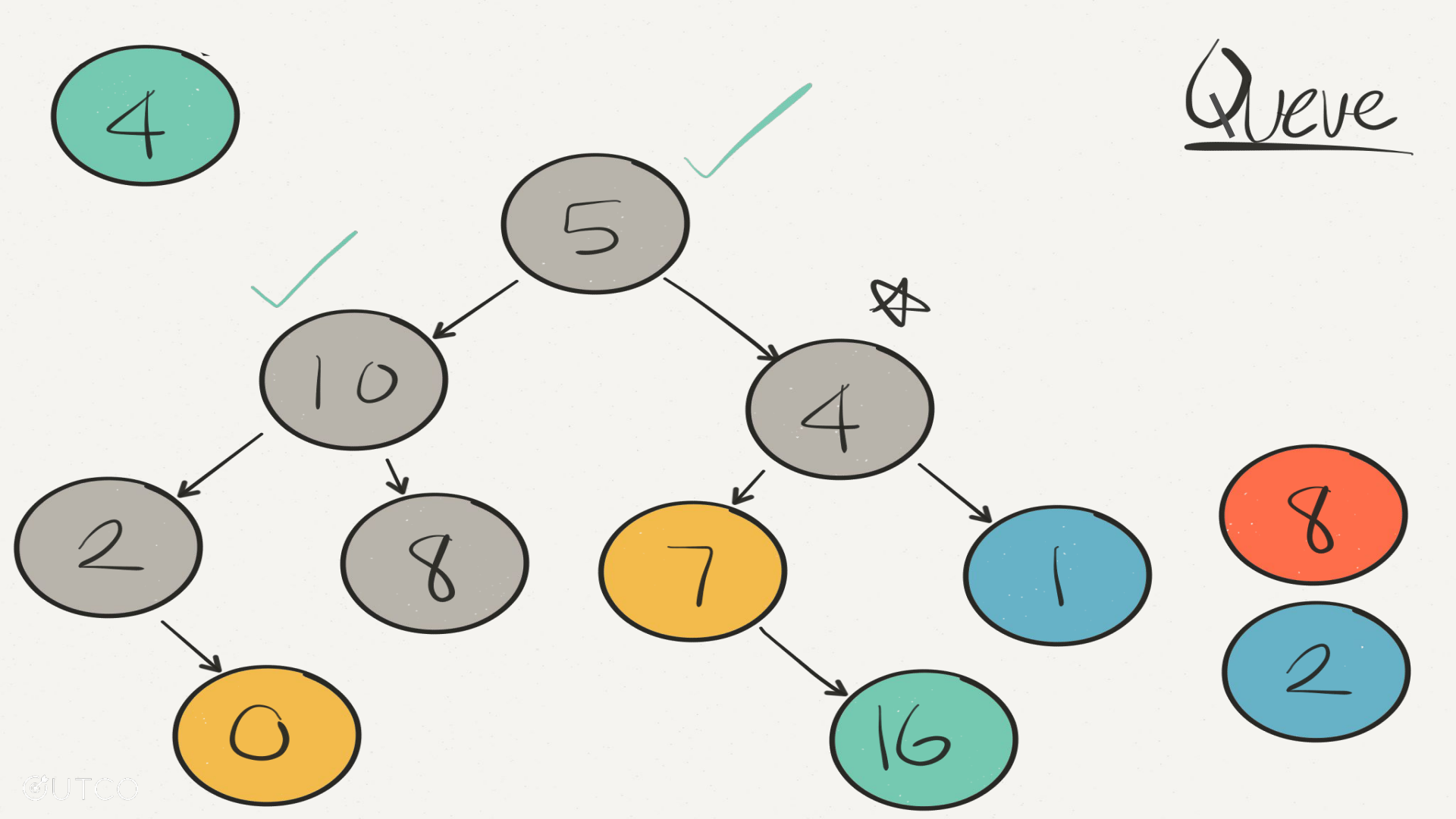

10
5
Queve
10
4
2
8
7
1
4
0
16

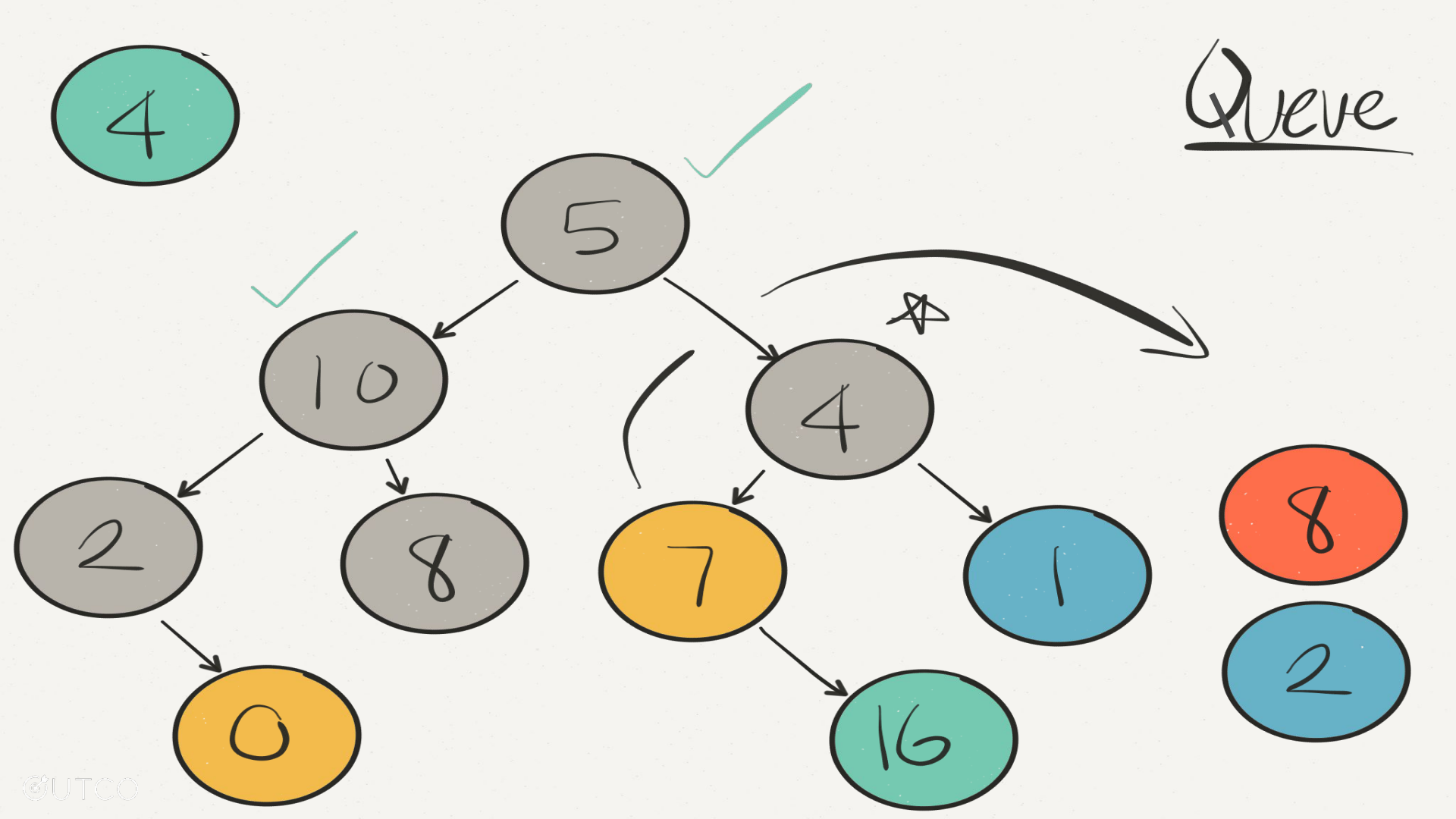

10
Queve
5
10
4
2
8
7
1
4
0
16

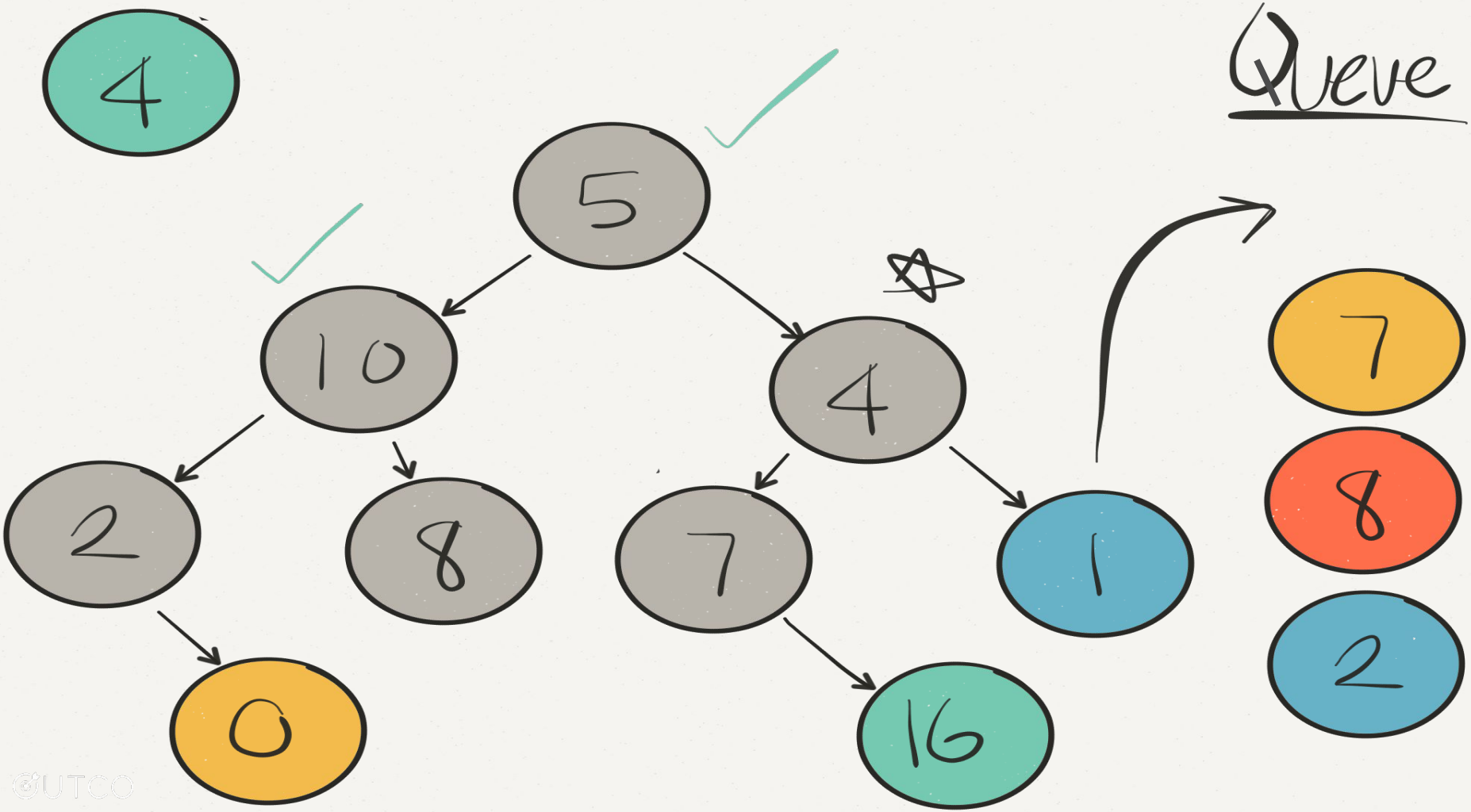

10
Queve
5
10
4
2
2
8
7
1
4
0
16

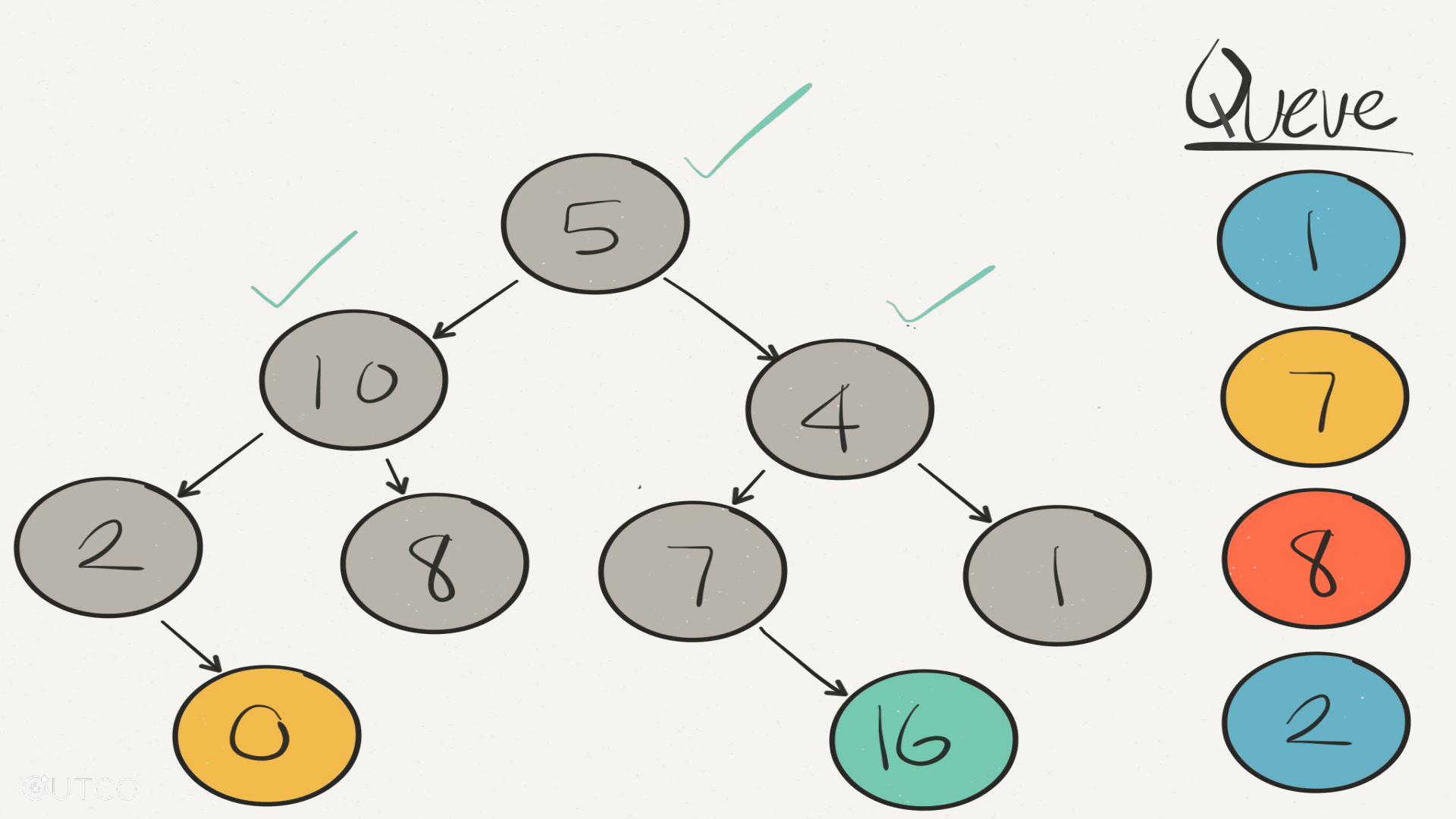

10
5
10
4
2
8
7
1
0
16
Queve
8
2
4

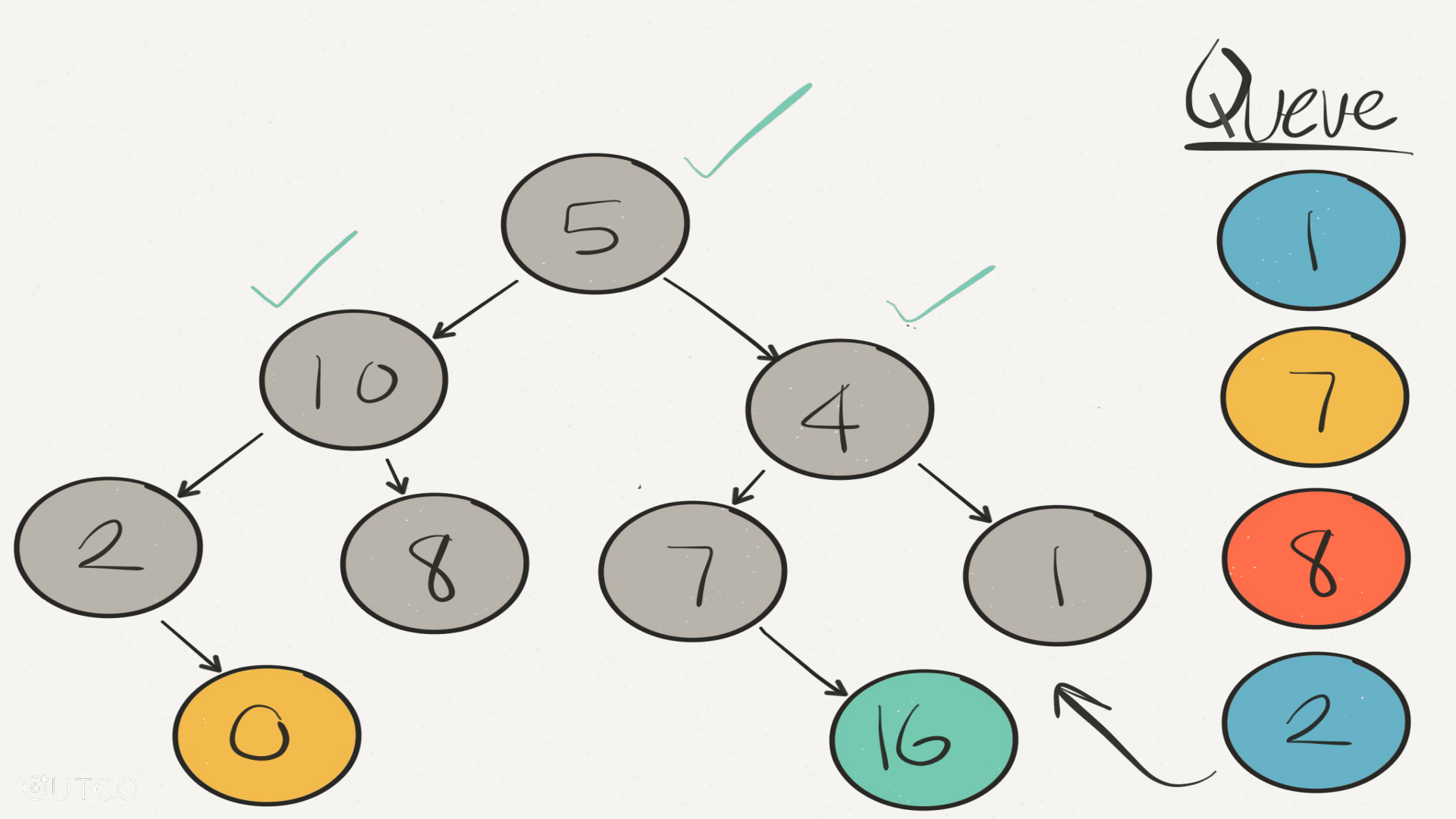

Queve
5
10
4
2
8
7
1
0
16
8
2
4

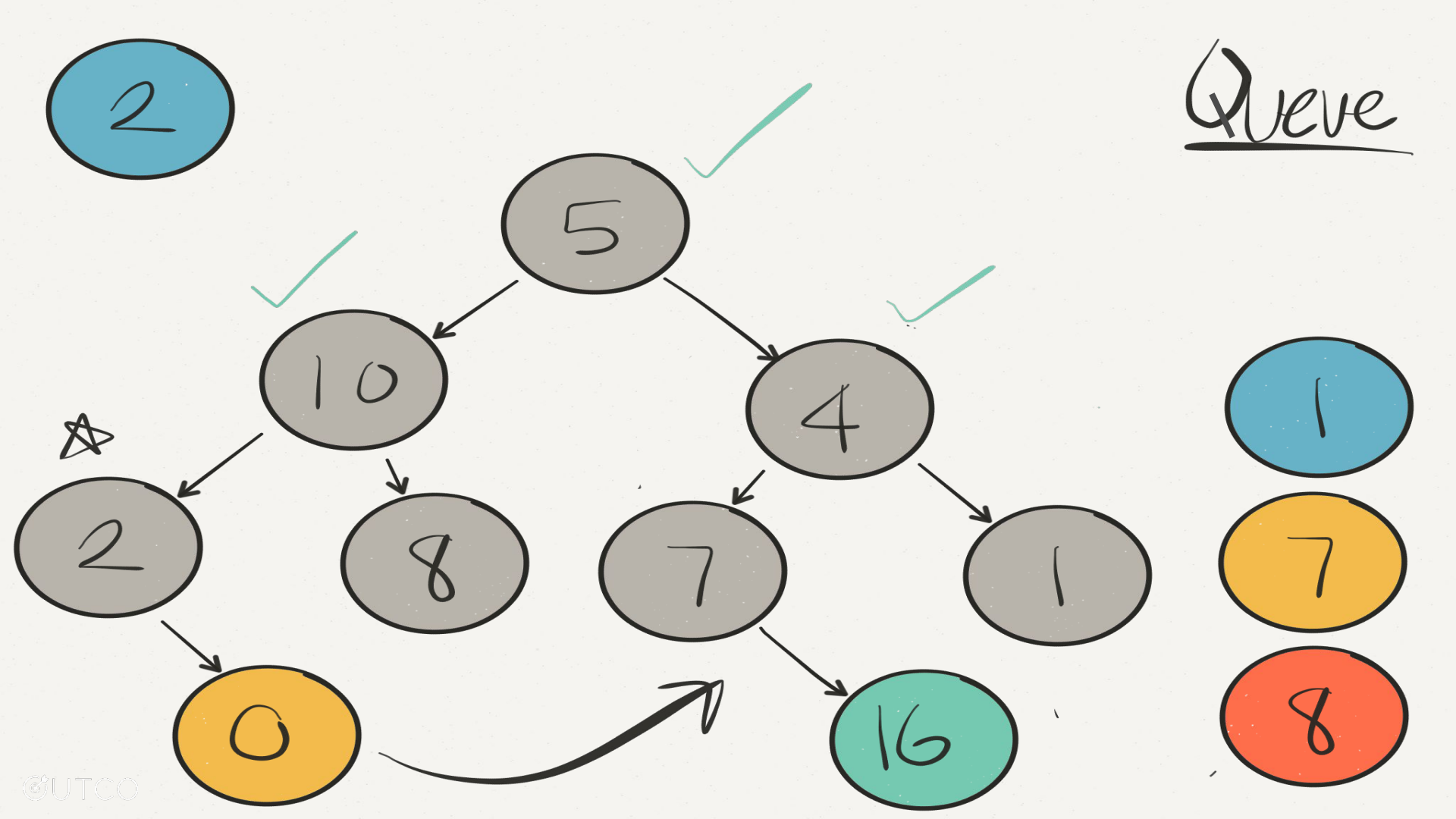

4
Queve
5
10
4
2
8
7
1
0
16
8
2

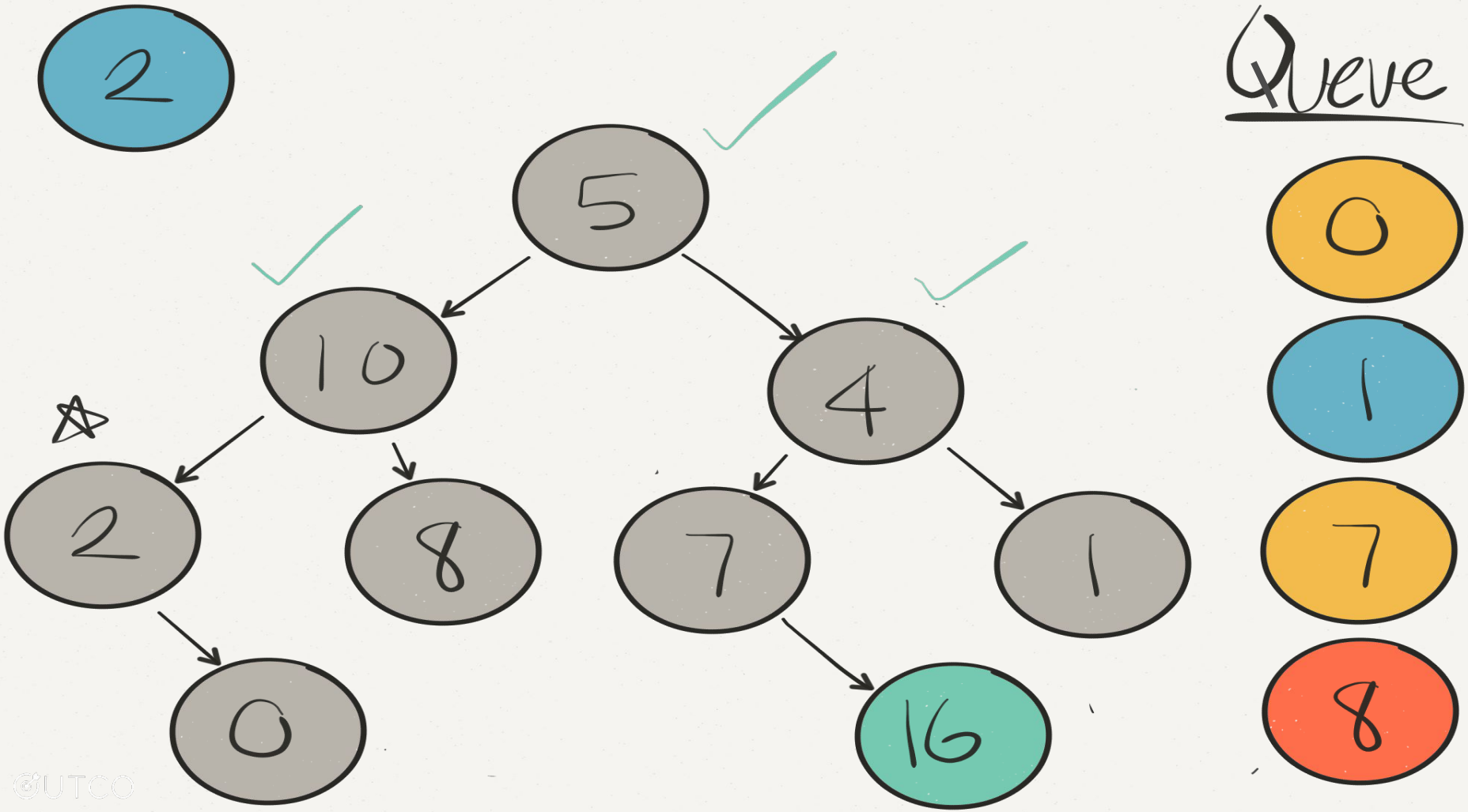

Queve
4
5
10
4
2
8
7
1
8
2
0
16

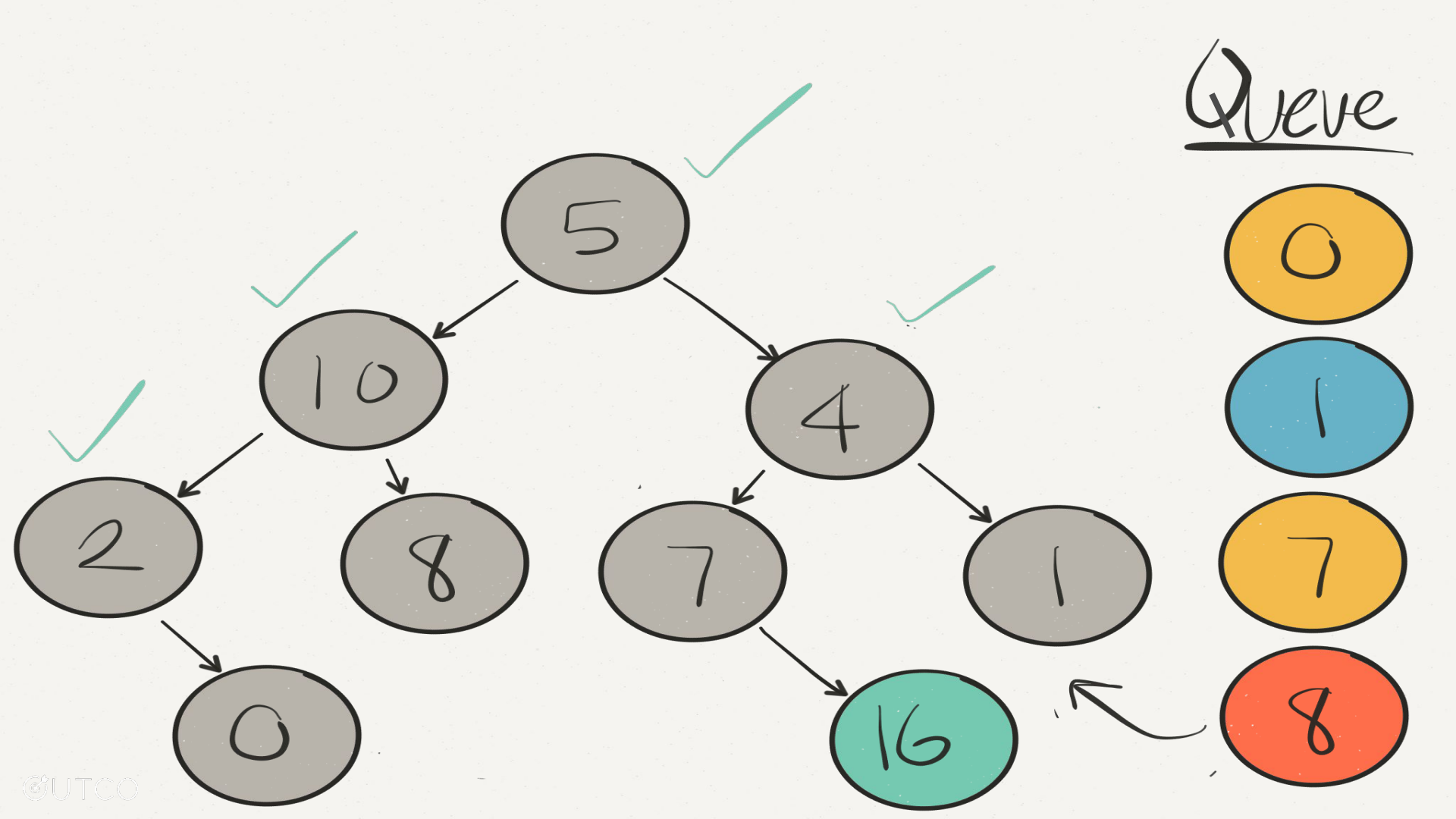

4
5
10
4
2
8
7
1
0
16
Queve
7
8
2

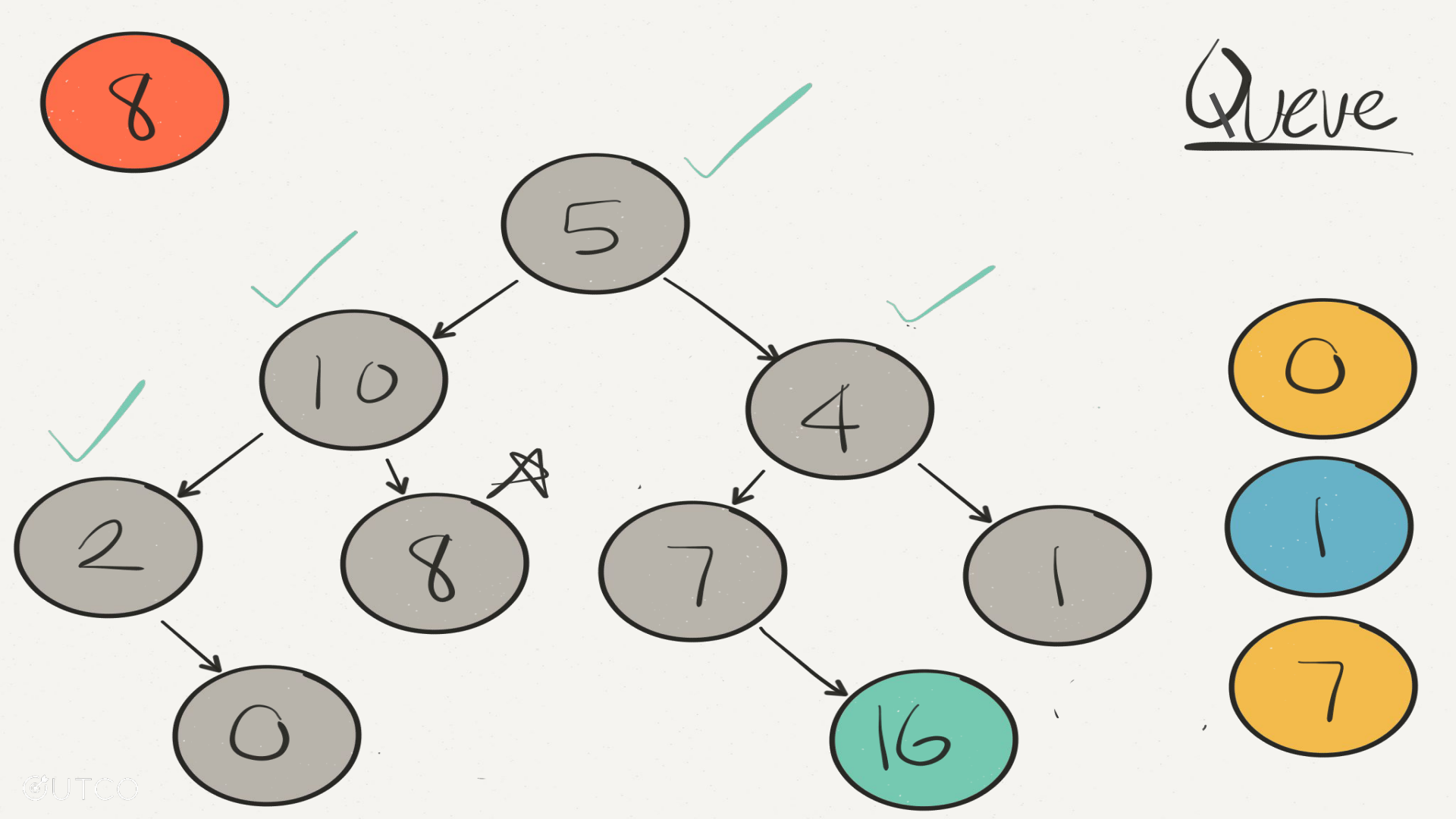

Queve
5
10
4
2
8
7
1
0
16
1
7
8
2

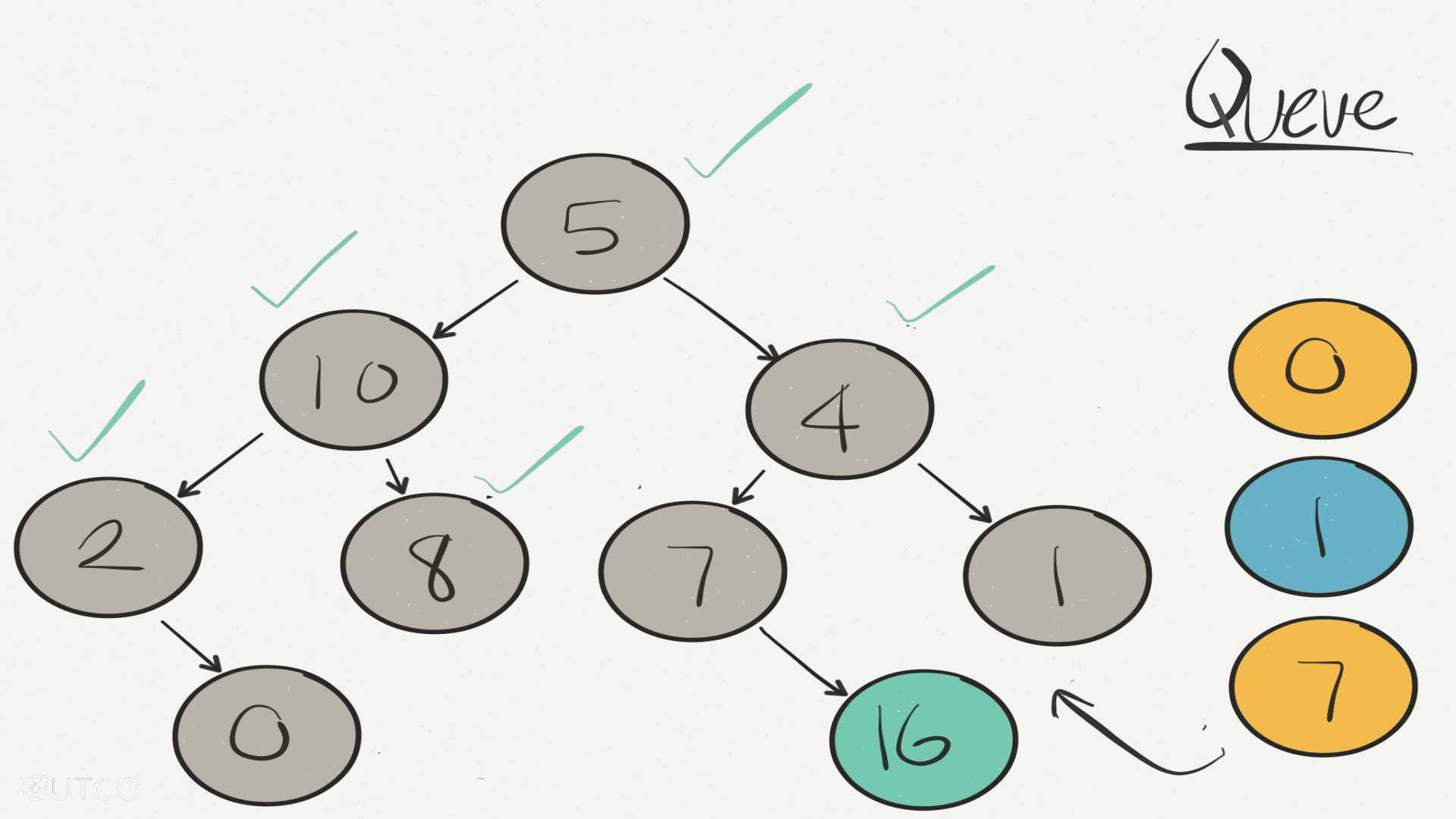

Queve
5
10
4
2
8
7
1
0
16
1
7
8
2

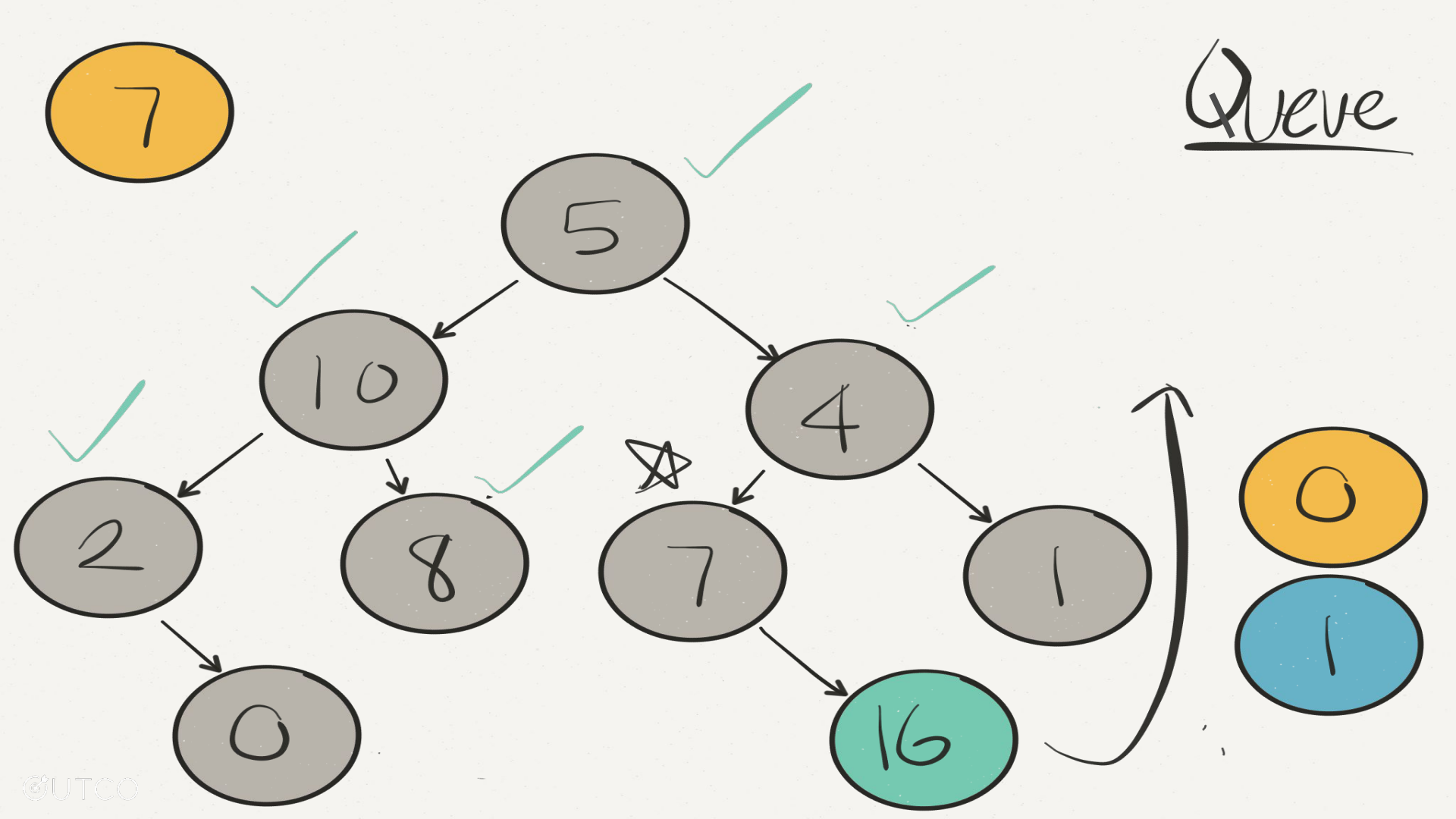

2
Queve
5
10
4
2
8
7
1
0
16
1
7
8

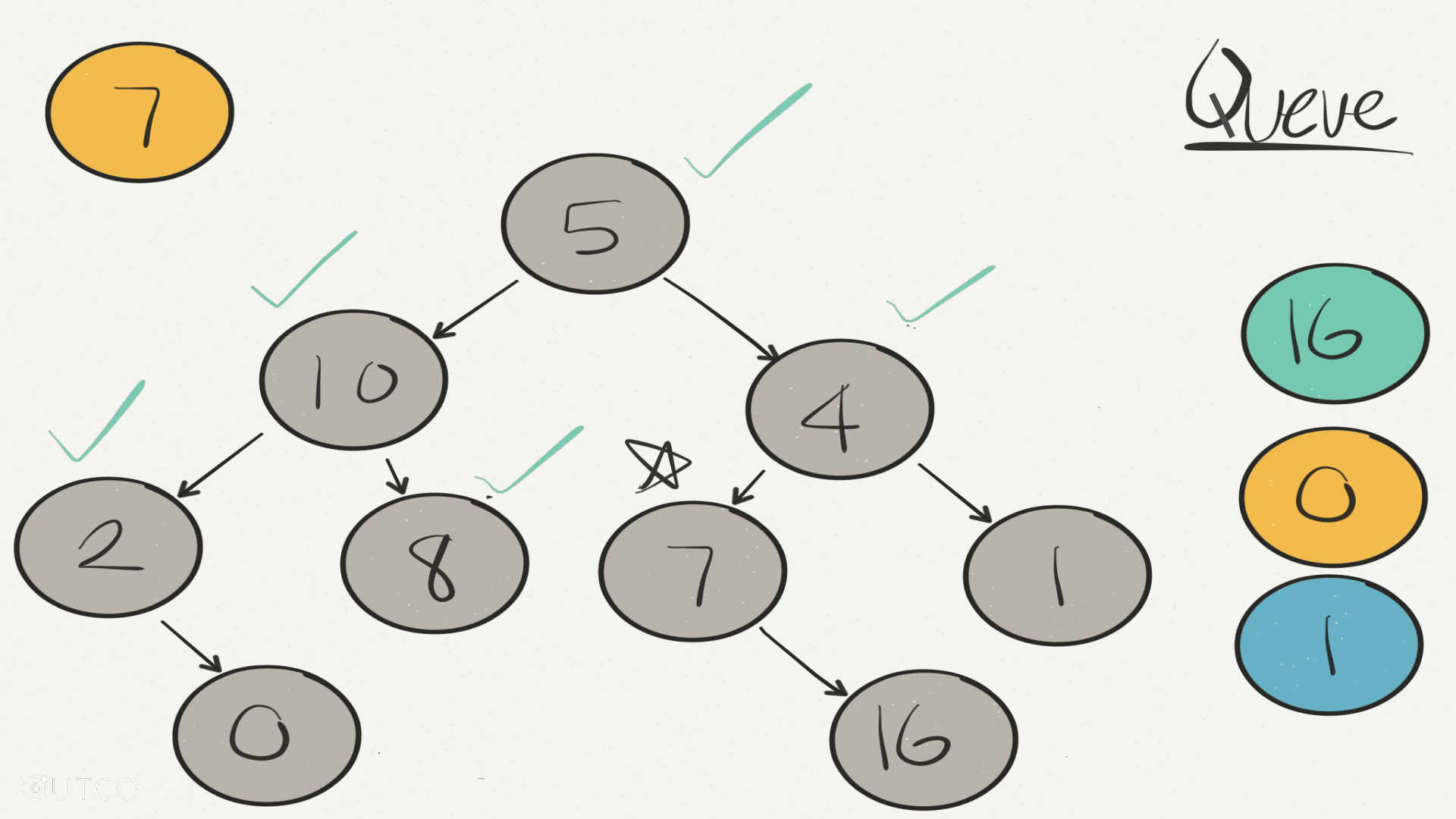

2
5
10
4
2
8
7
1
0
16
Queve
0
1
7
8

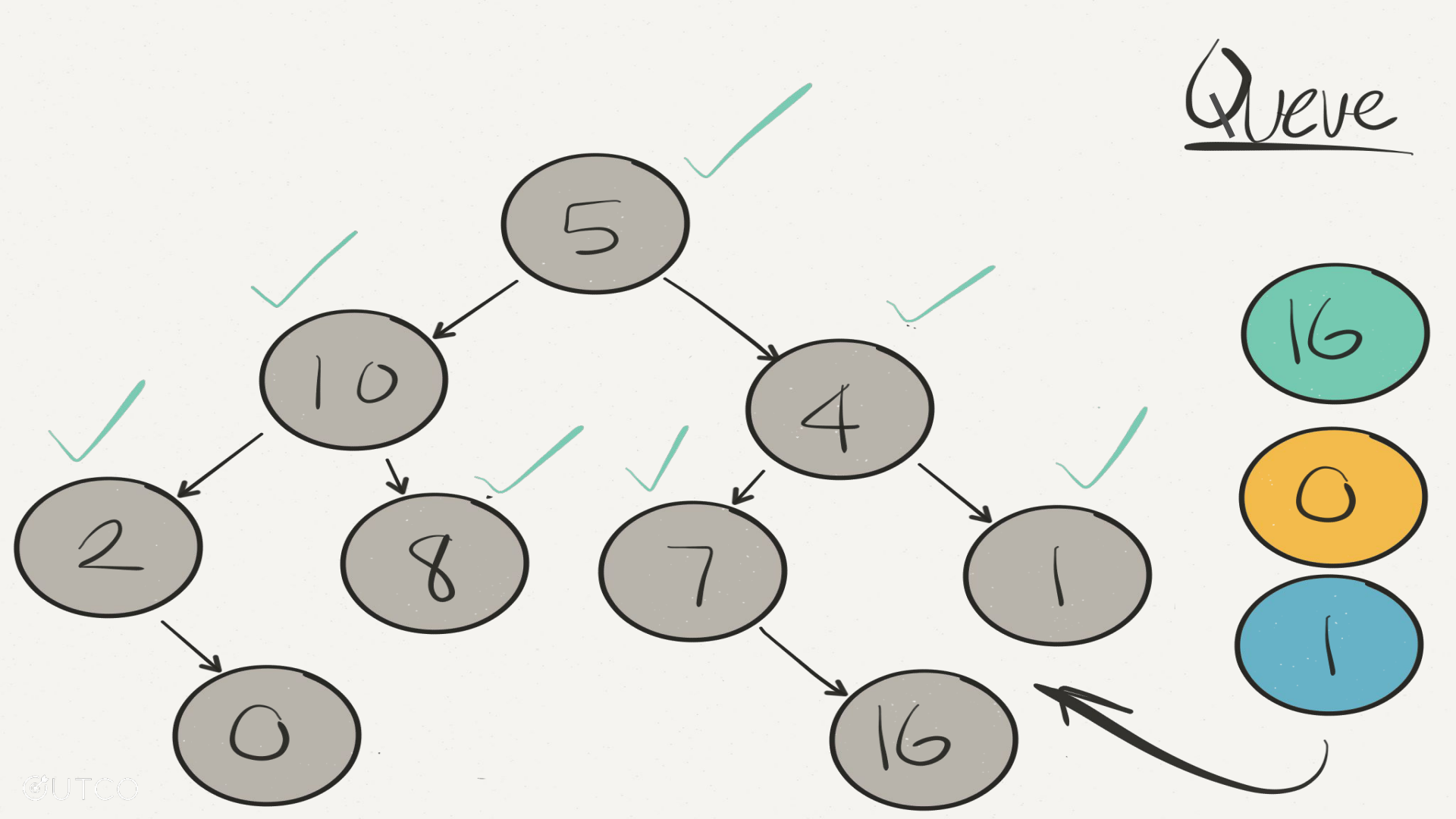

Queue
5
10
4
2
8
7
1
0
16
0
1
7
8

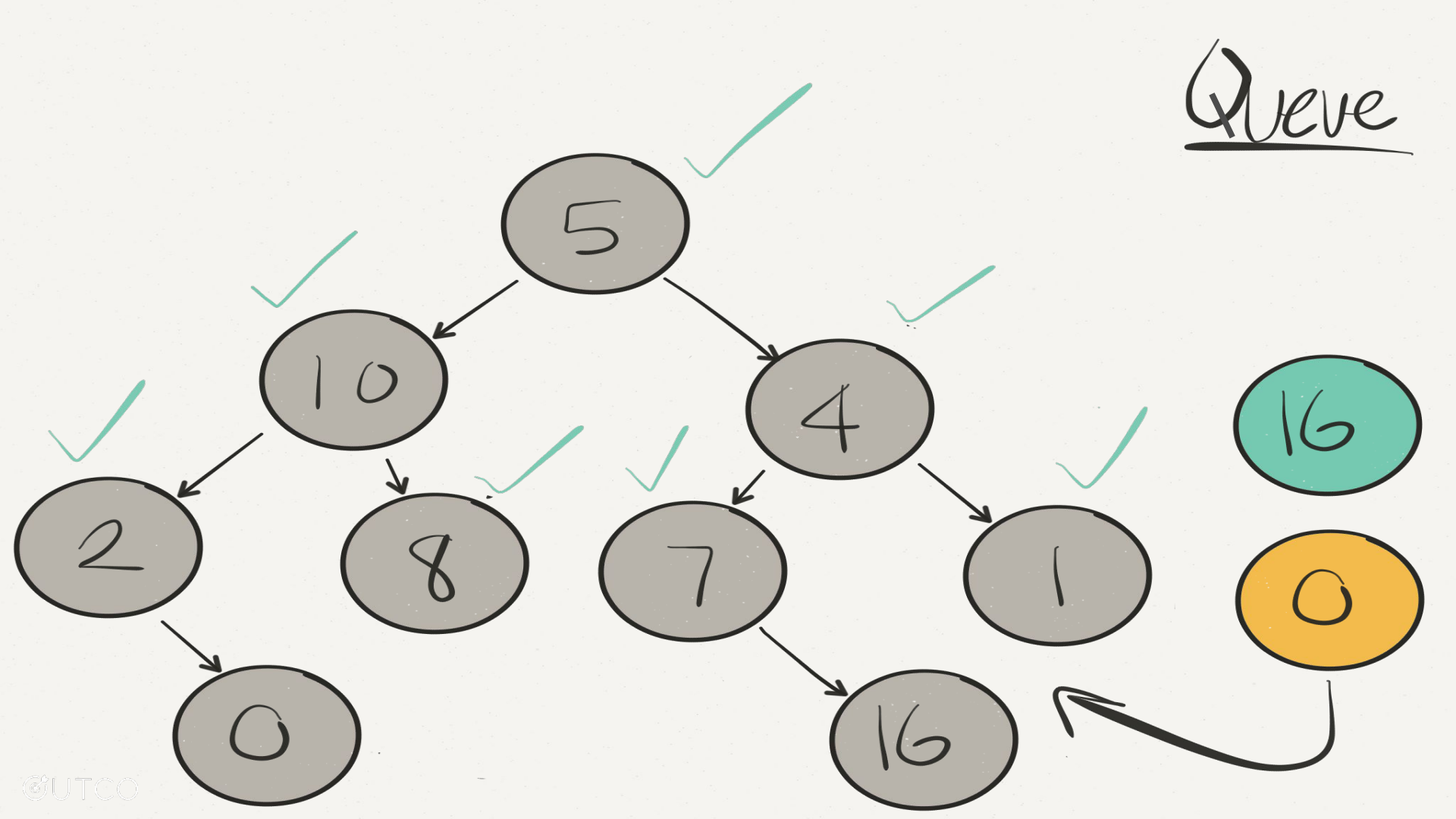

8
Queve
5
10
4
2
8
7
1
0
16
0
1
7


Queue
5
10
4
2
8
7
1
0
16
0
1
7

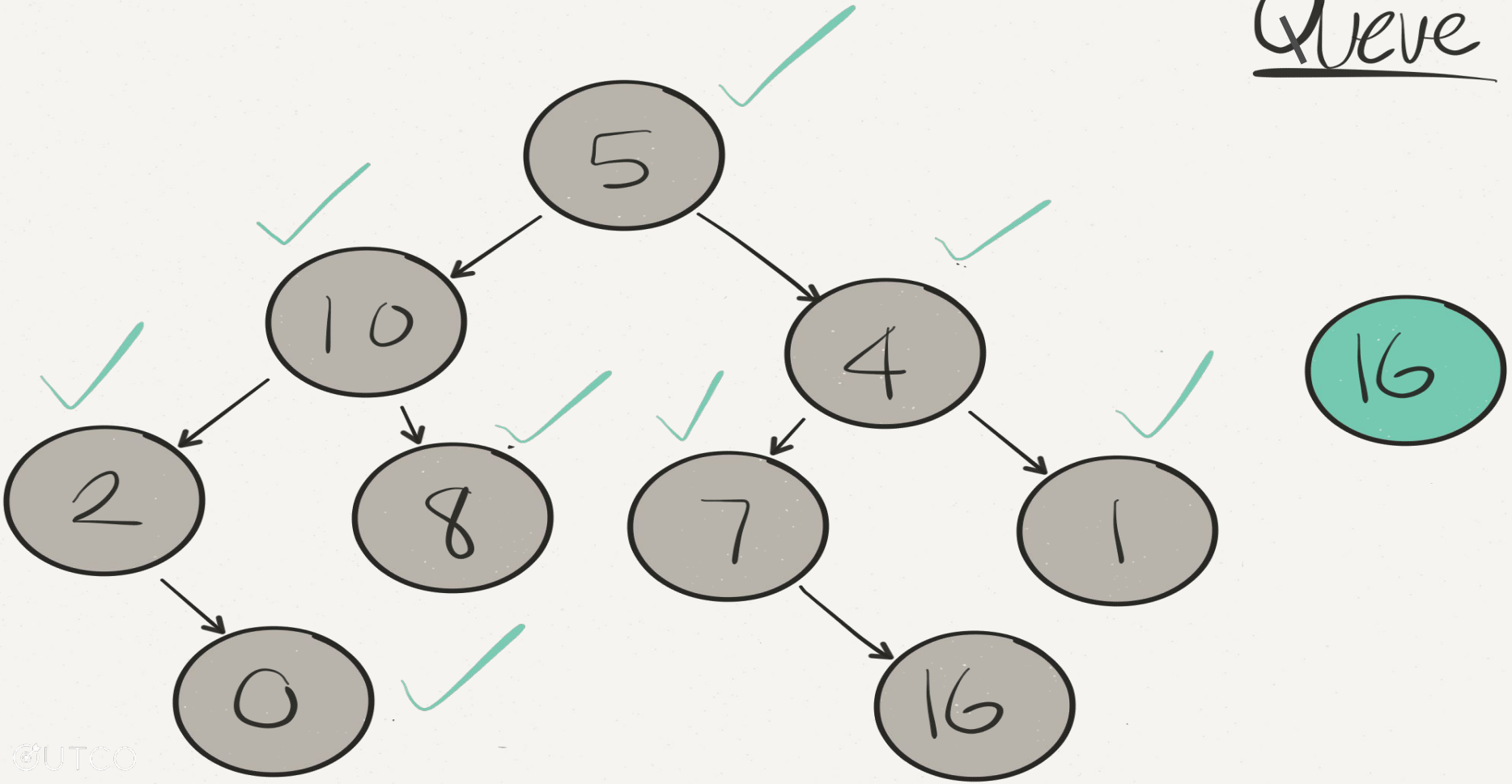

7
Queue
5
10
4
2
8
7
1
0
16
0
1

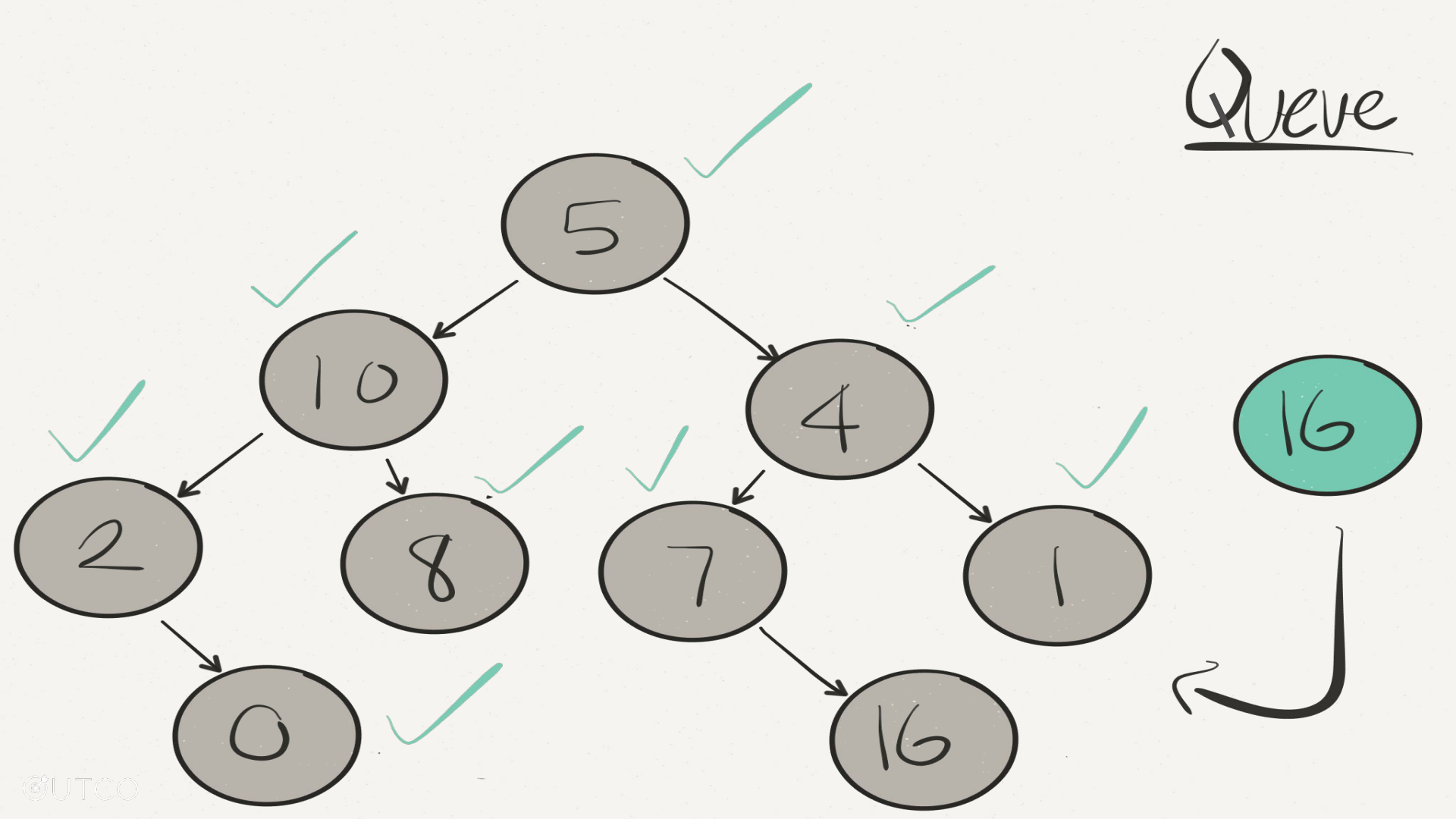

7
Queue
5
10
4
2
8
7
1
0
16
16
0
1


Queue
5
10
4
2
8
7
1
0
16
16
0
1

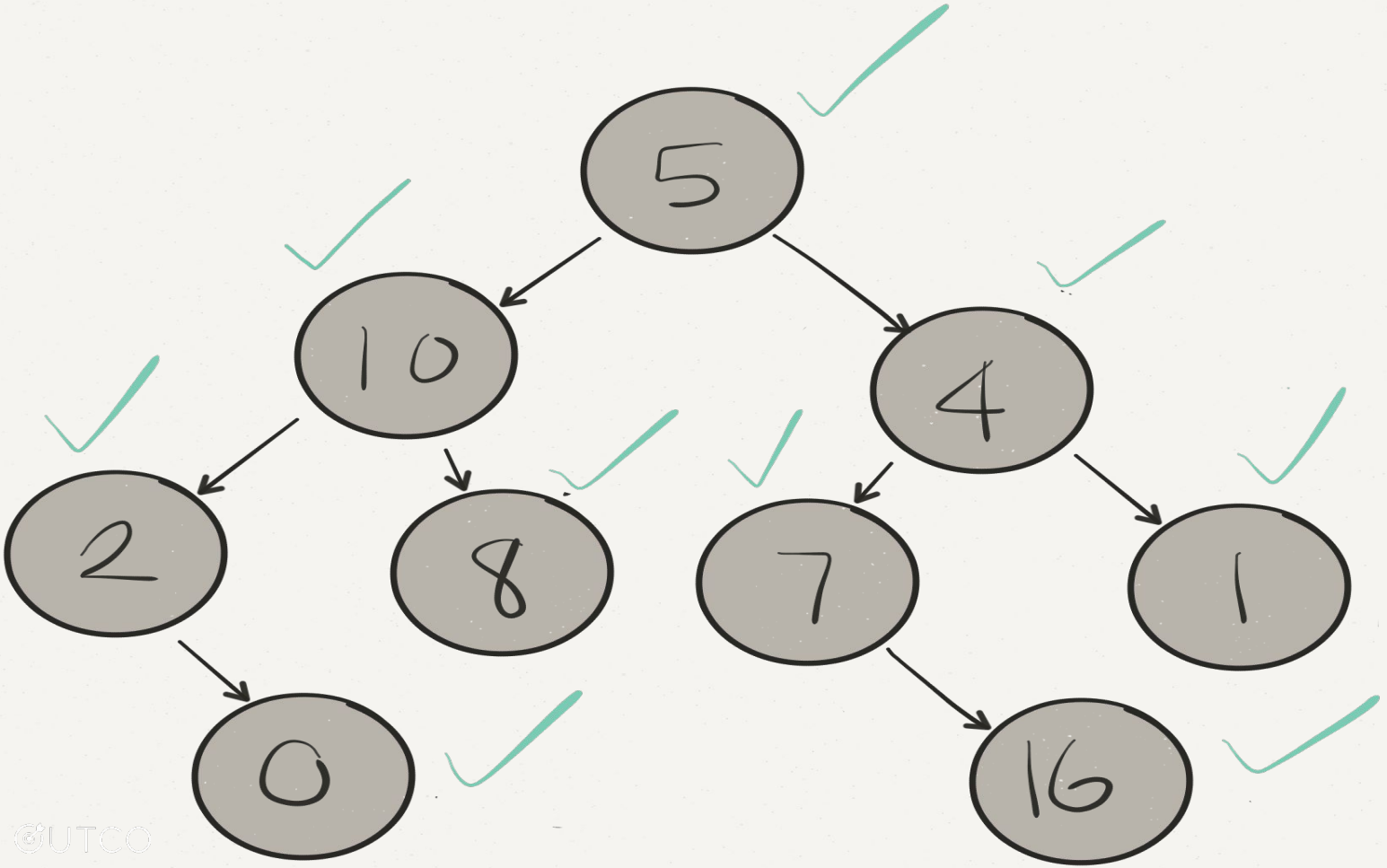

Queue
5
10
4
2
8
7
1
16
0
0
16

Queue




Queve
5
10
4
16
2
8
7
1
0
16

Queve



Insert method for tree class

```
def insert(val)
    new_node = Node.new(val)
    if @root == nil
      @root = new_node
      return
    end

    queue = [@root]
    while(queue.size)
      current = queue.shift
      if current.left_child == nil
        current.left_child = new_node
        return
      else
        queue.push(current.left_child)
      end
     ...
```

create a new node

Insert method for tree class

```
def insert(val)
    new_node = Node.new(val)
    if @root == nil
      @root = new_node
      return
    end

    queue = [@root]
    while(queue.size)
      current = queue.shift
      if current.left_child == nil
        current.left_child = new_node
        return
      else
        queue.push(current.left_child)
      end
     ...
```

if root is empty assign new node to root and return

Insert method for tree class

```
def insert(val)
    new_node = Node.new(val)
    if @root == nil
      @root = new_node
      return
    end

    queue = [@root]
    while(queue.size)
      current = queue.shift
      if current.left_child == nil
        current.left_child = new_node
        return
      else
        queue.push(current.left_child)
      end
     ...
```

create a queue to for breadth first search for an empty child position

Insert method for tree class

```
def insert(val)
    new_node = Node.new(val)
    if @root == nil
      @root = new_node
      return
    end

    queue = [@root]
    while(queue.size)
      current = queue.shift
      if current.left_child == nil
        current.left_child = new_node
        return
      else
        queue.push(current.left_child)
      end
     ...
```

as long as there are items in the queue

Insert method for tree class

```
def insert(val)
    new_node = Node.new(val)
    if @root == nil
      @root = new_node
      return
    end

    queue = [@root]
    while(queue.size)
      current = queue.shift
      if current.left_child == nil
        current.left_child = new_node
        return
      else
        queue.push(current.left_child)
      end
     ...
```

dequeue node and set to current

Insert method for tree class

```
def insert(val)
    new_node = Node.new(val)
    if @root == nil
      @root = new_node
      return
    end

    queue = [@root]
    while(queue.size)
      current = queue.shift
      if current.left_child == nil
        current.left_child = new_node
        return
      else
        queue.push(current.left_child)
      end
     ...
```

if left child is empty set new node to left child

Insert method for tree class

```
def insert(val)
    new_node = Node.new(val)
    if @root == nil
      @root = new_node
      return
    end

    queue = [@root]
    while(queue.size)
      current = queue.shift
      if current.left_child == nil
        current.left_child = new_node
        return
      else
        queue.push(current.left_child)
      end
     ...
```

otherwise add left child into queue

Insert method for tree class

```
    ...

    if current.right_child == nil
        current.right_child = new_node
        return
      else
        queue.push(current.right_child)
      end

    end
  end
```

then repeat for right child

Insert method for tree class

```
    ...

    if current.right_child == nil
        current.right_child = new_node
        return
      else
        queue.push(current.right_child)
      end

    end
  end
```

check if right child is empty

Insert method for tree class

```
    ...

    if current.right_child == nil
        current.right_child = new_node
        return
      else
        queue.push(current.right_child)
      end

    end
  end
```

if right child is empty set new node to right child

Insert method for tree class

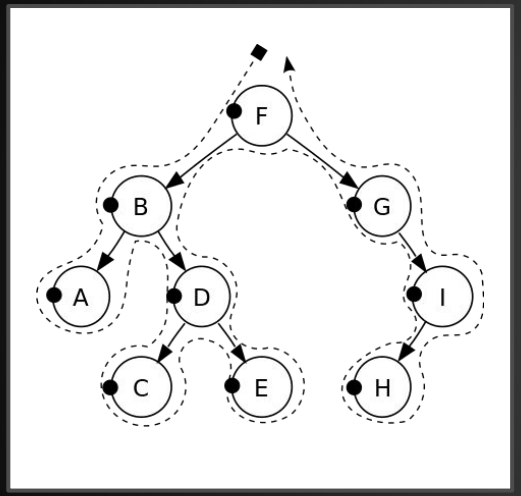
```
    ...

    if current.right_child == nil
        current.right_child = new_node
        return
      else
        queue.push(current.right_child)
      end

    end
  end
```

otherwise push right child into queue

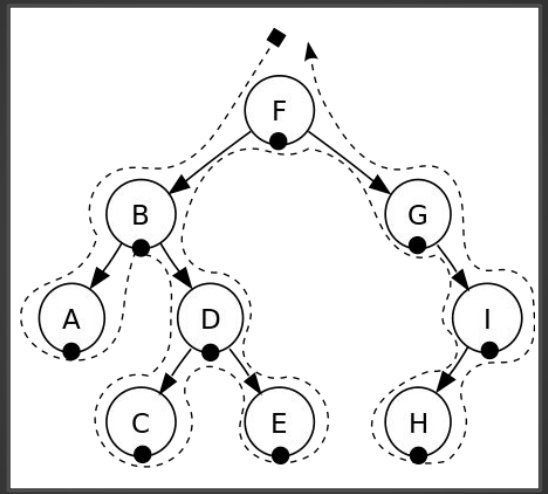
Insert method for tree class

```
def insert(val)
    new_node = Node.new(val)
    if @root == nil
      @root = new_node
      return
    end

    queue = [@root]
    while(queue.size)
      current = queue.shift
      if current.left_child == nil
        current.left_child = new_node
        return
      else
        queue.push(current.left_child)
      end
      if current.right_child == nil
          current.right_child = new_node
          return
        else
          queue.push(current.right_child)
        end

      end
    end
```

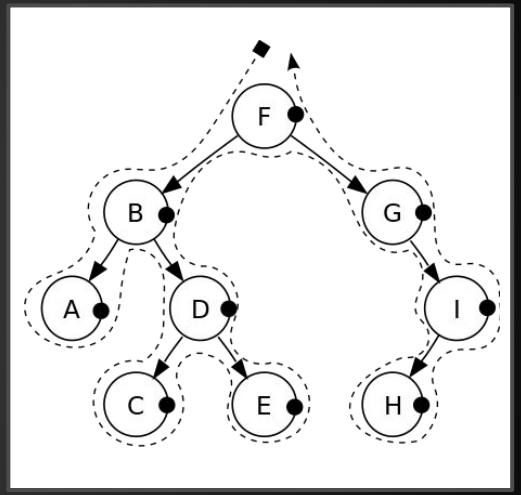
Depth First Search



Pre-order



In-order

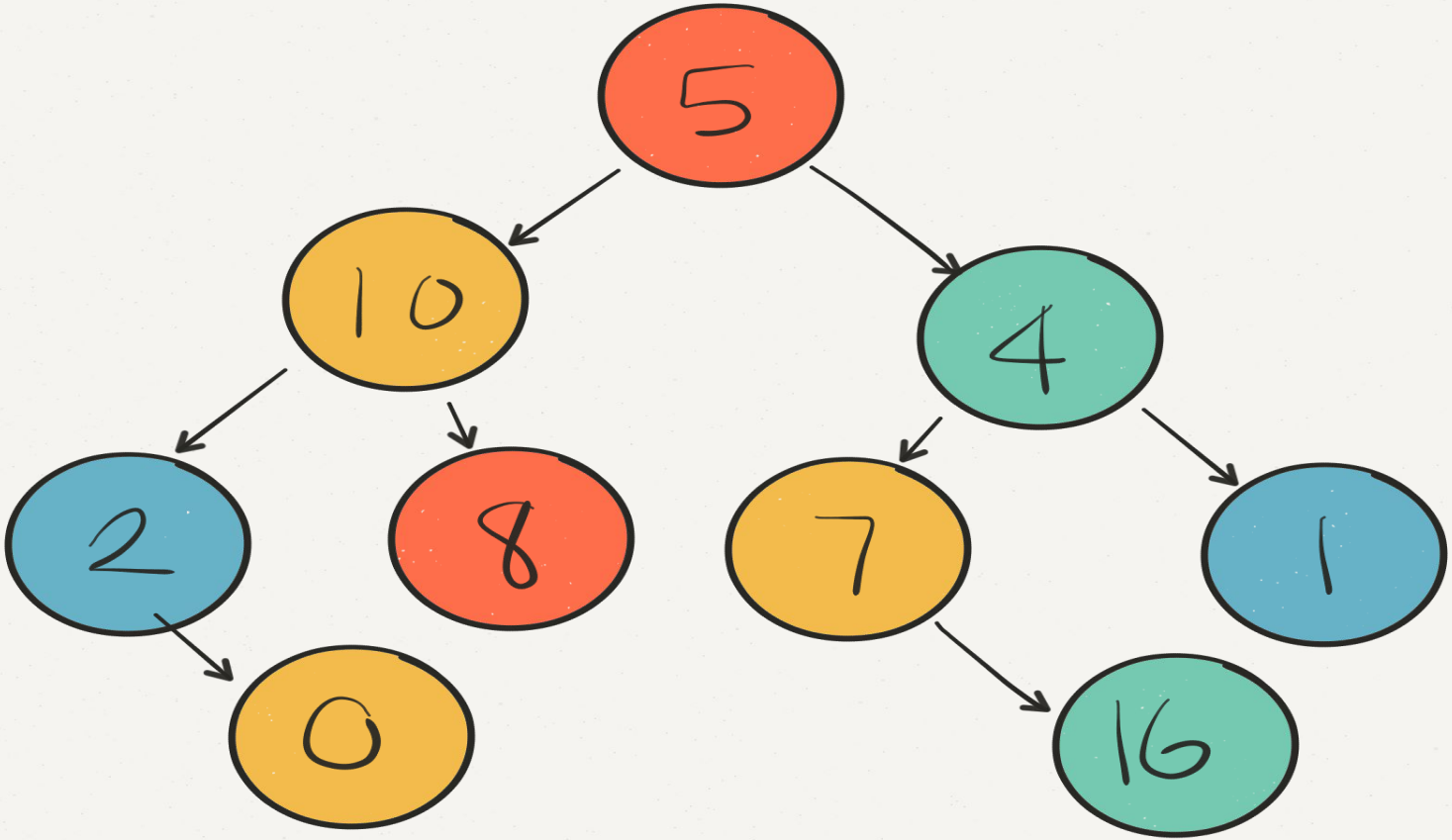


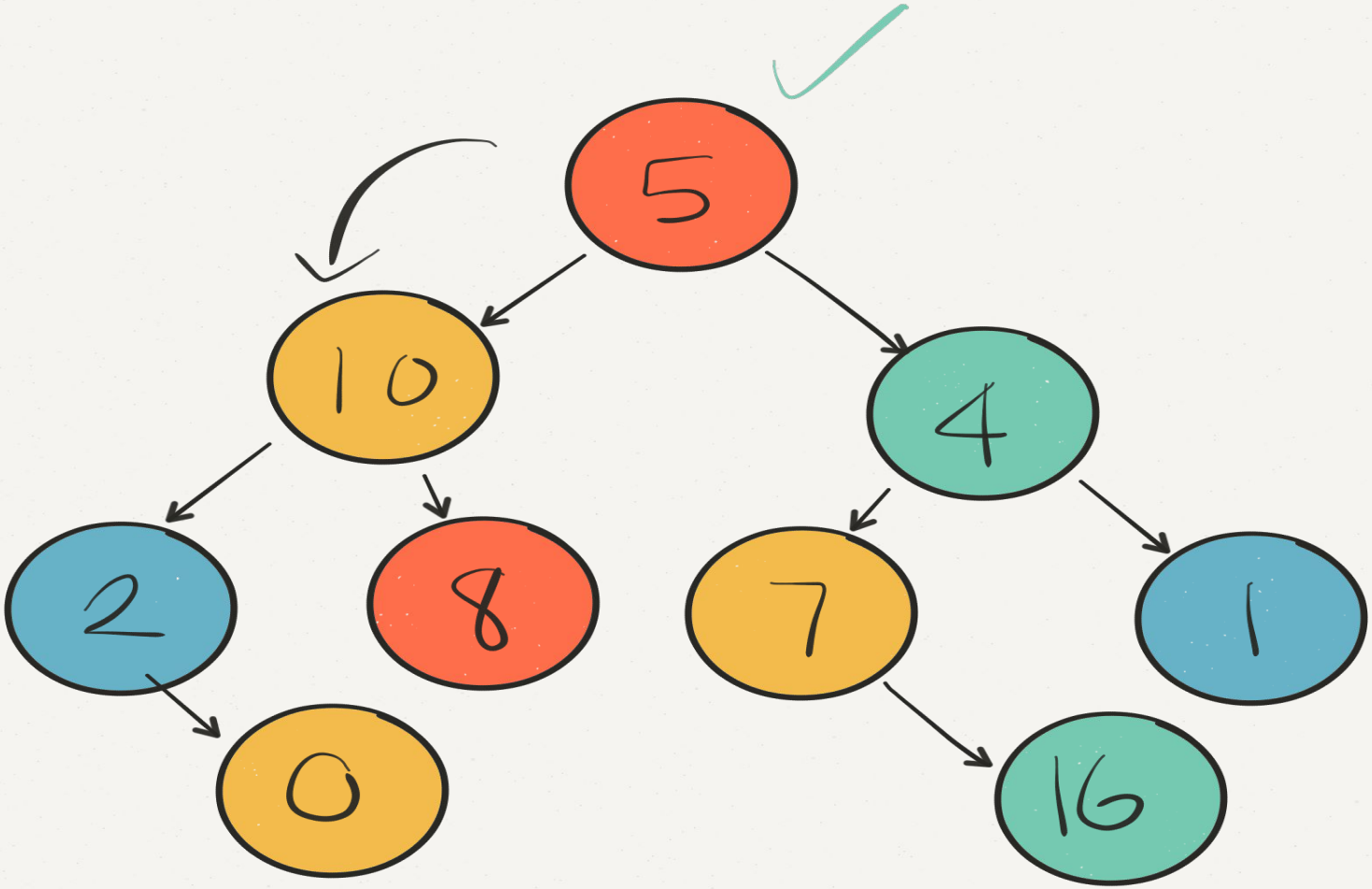
Post-order

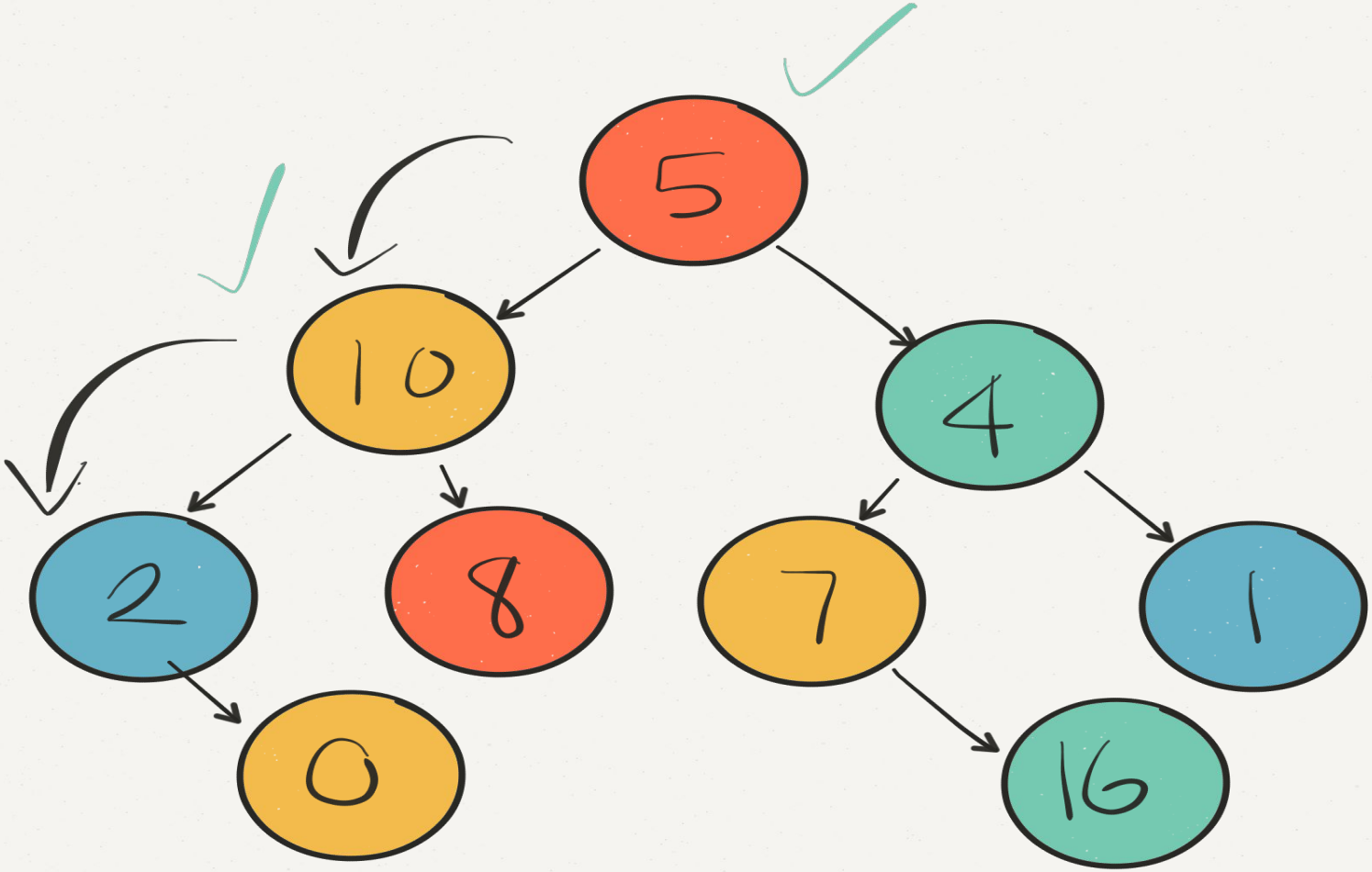
Depth First Search

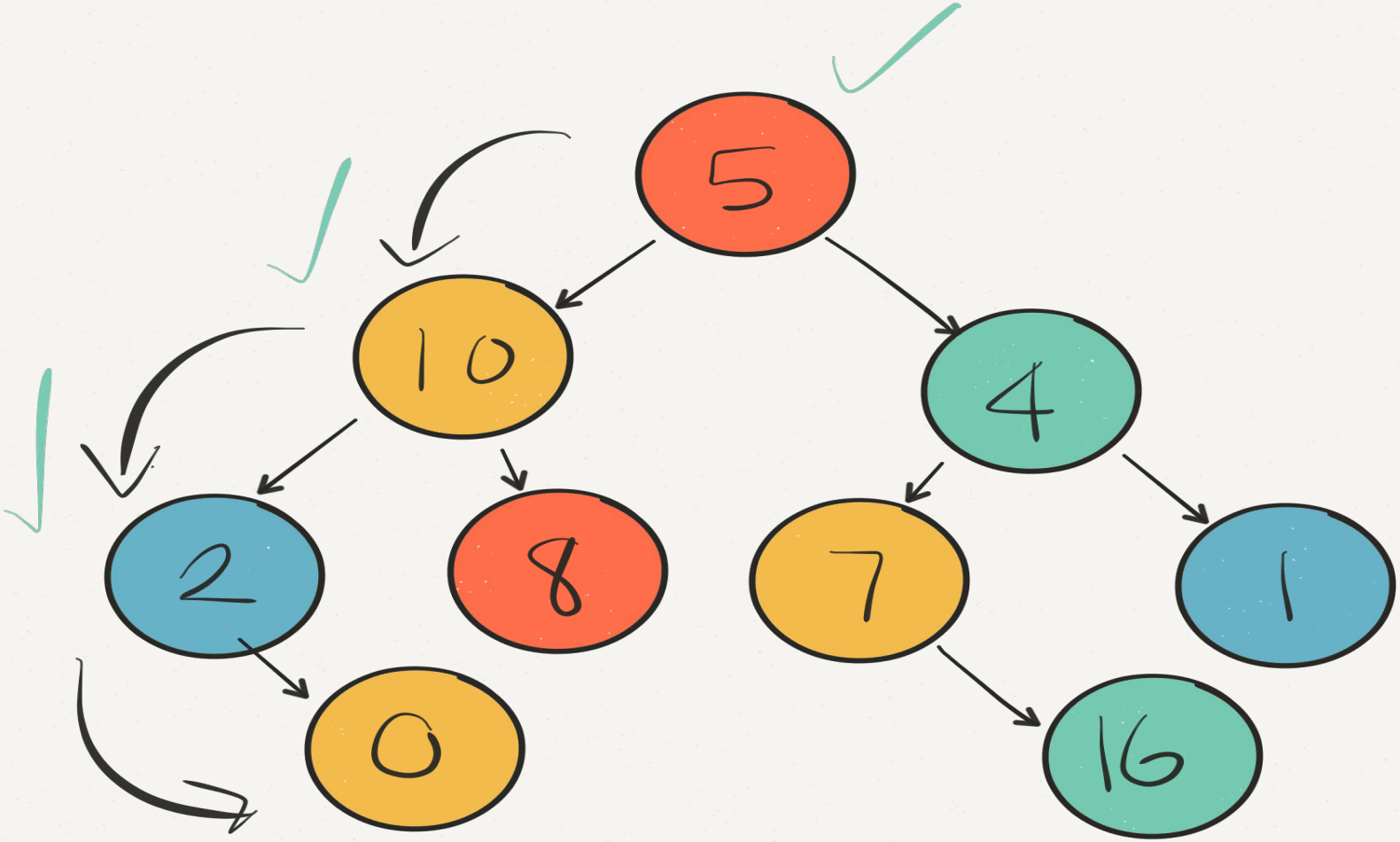
Can be implemented using recursion

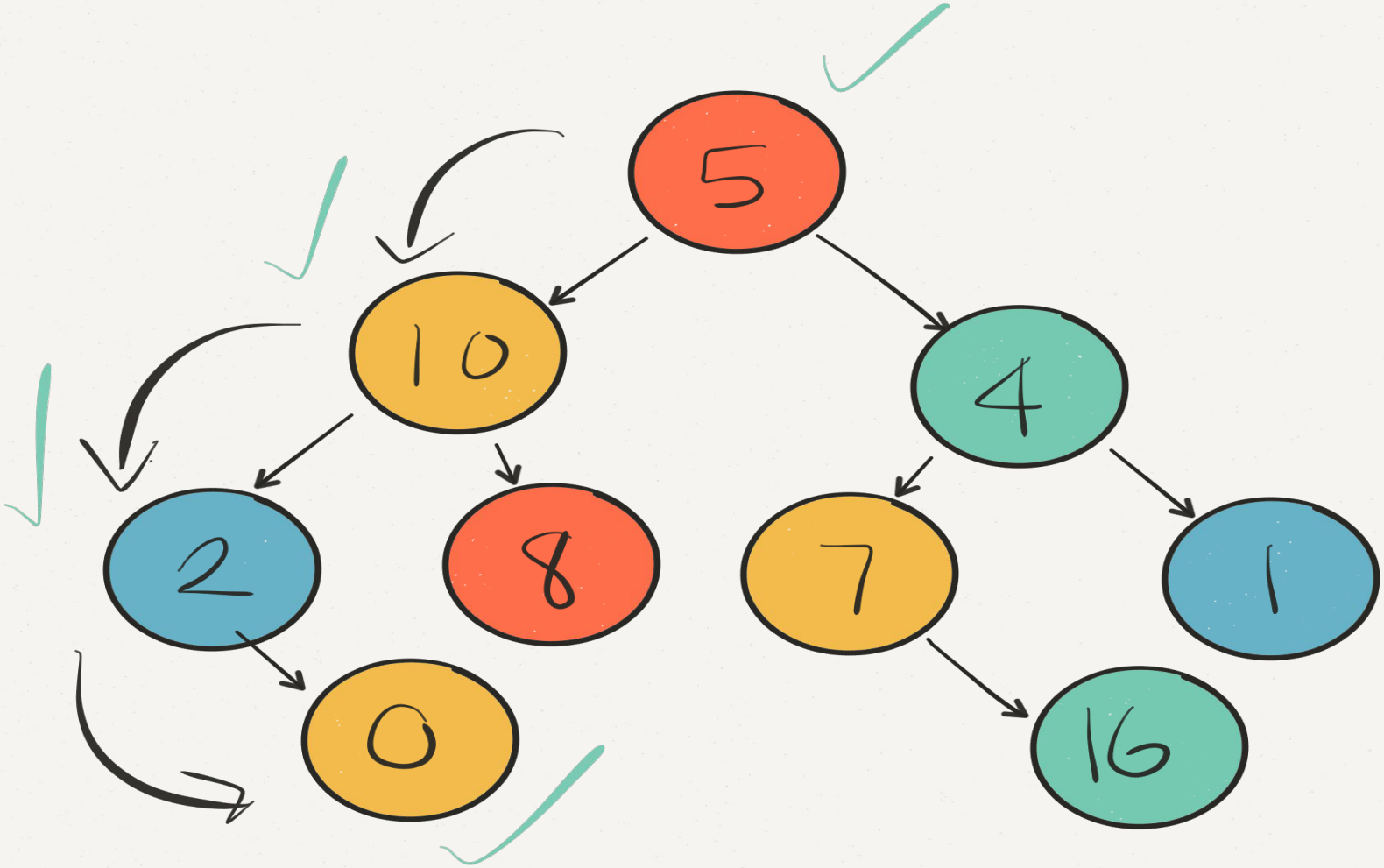
- Start at root node
- Perform some operation
  - this could be before, in-between, or after the recursive case
- Base case
  - No more children
- Recursive case
  - Loop through collection of children
  - Call recursive case on each child

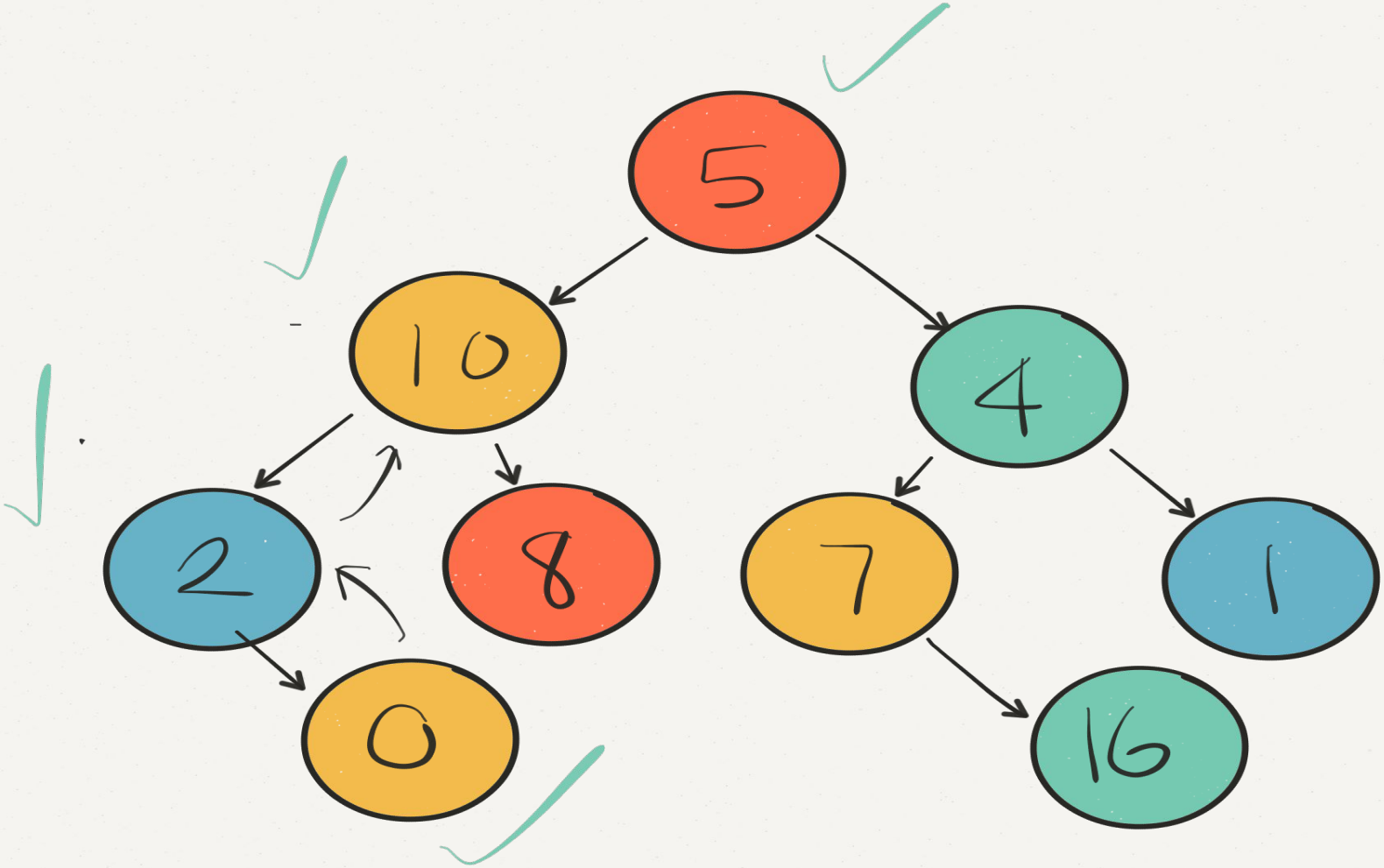

5
10
4
2
8
7
1
0
16

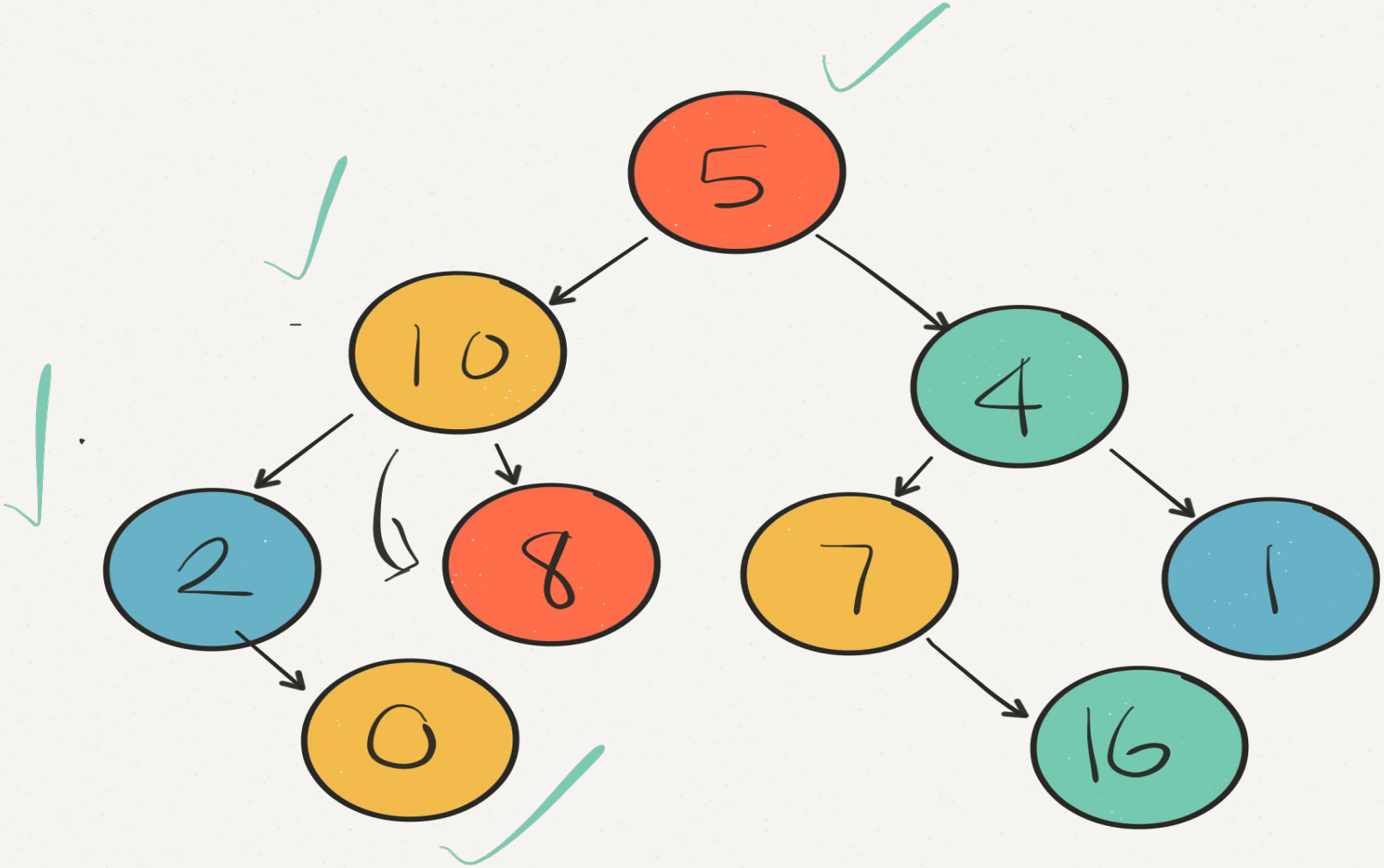

5
10
4
2
8
7
1
0
16

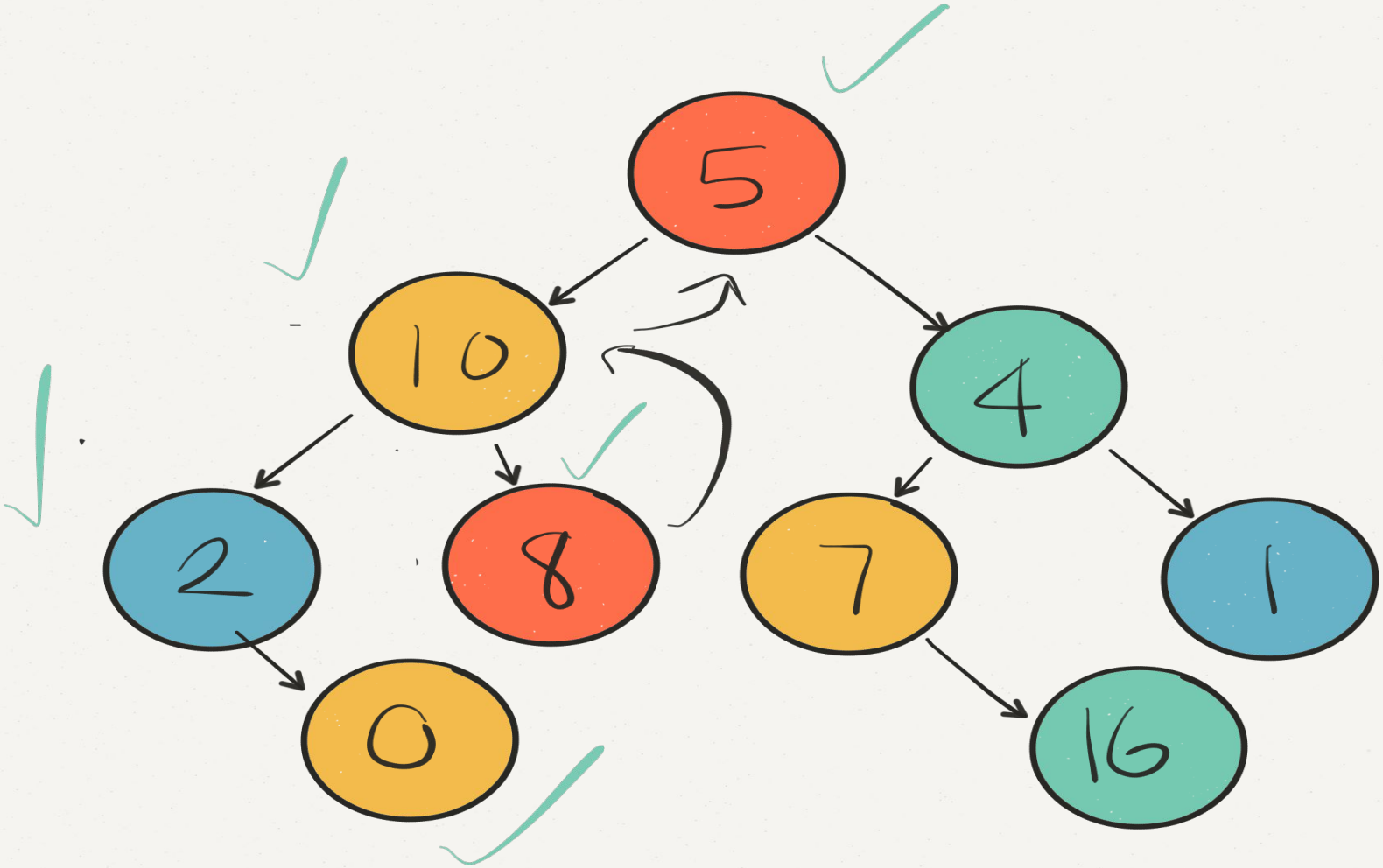

5
10
4
2
8
7
1
0
16

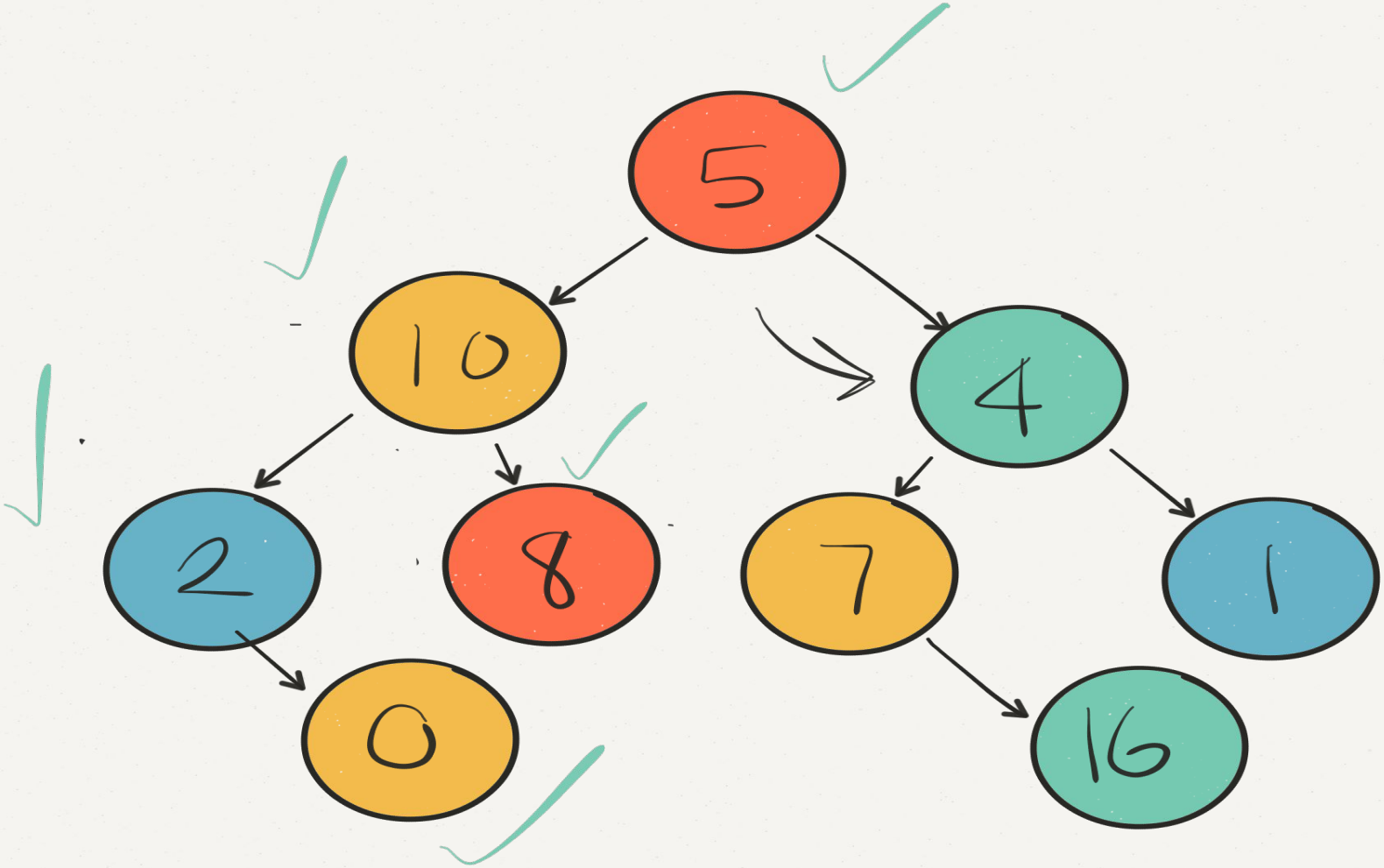

5
10
4
2
8
7
1
0
16

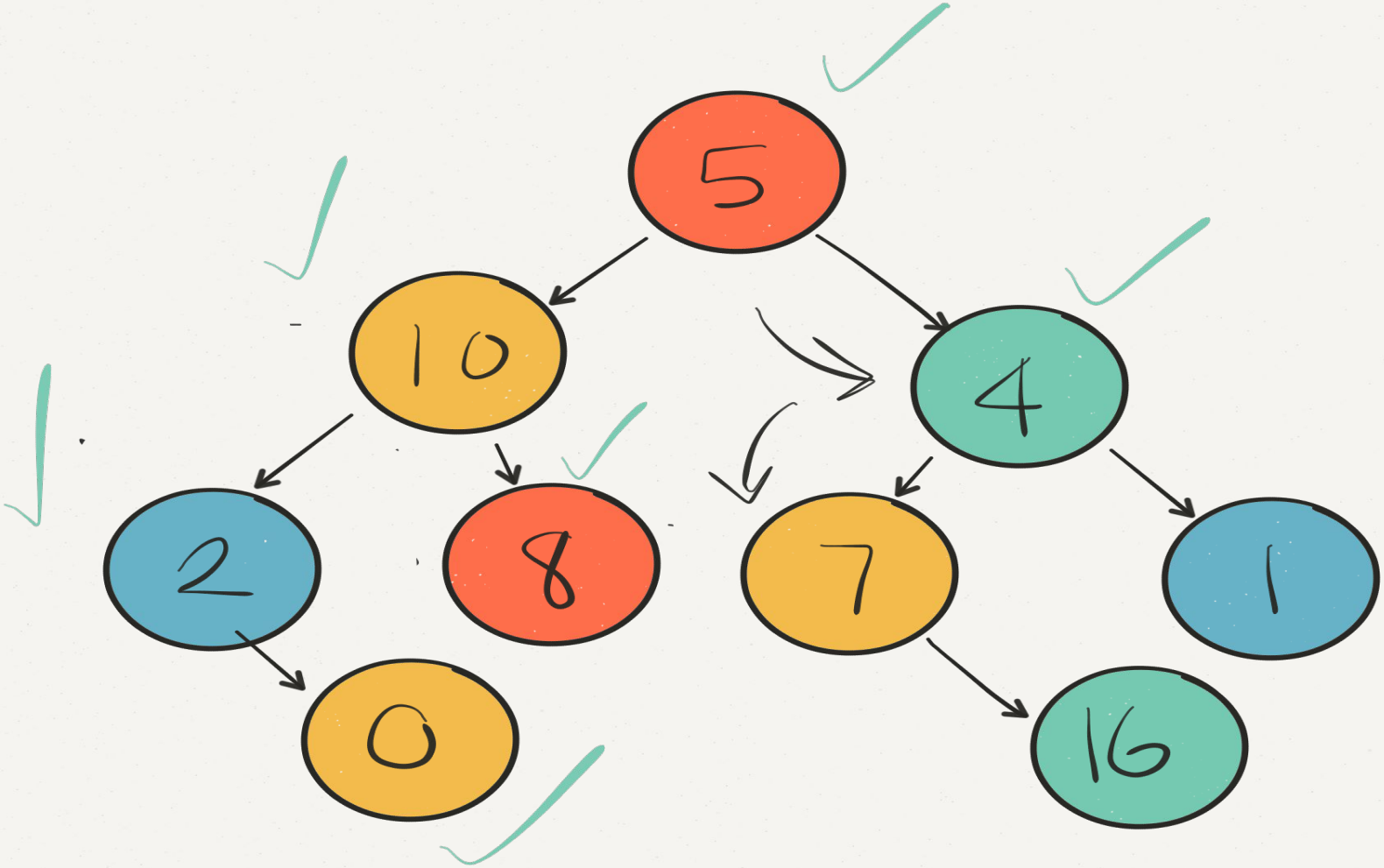

5
10
4
2
8
7
1
0
16

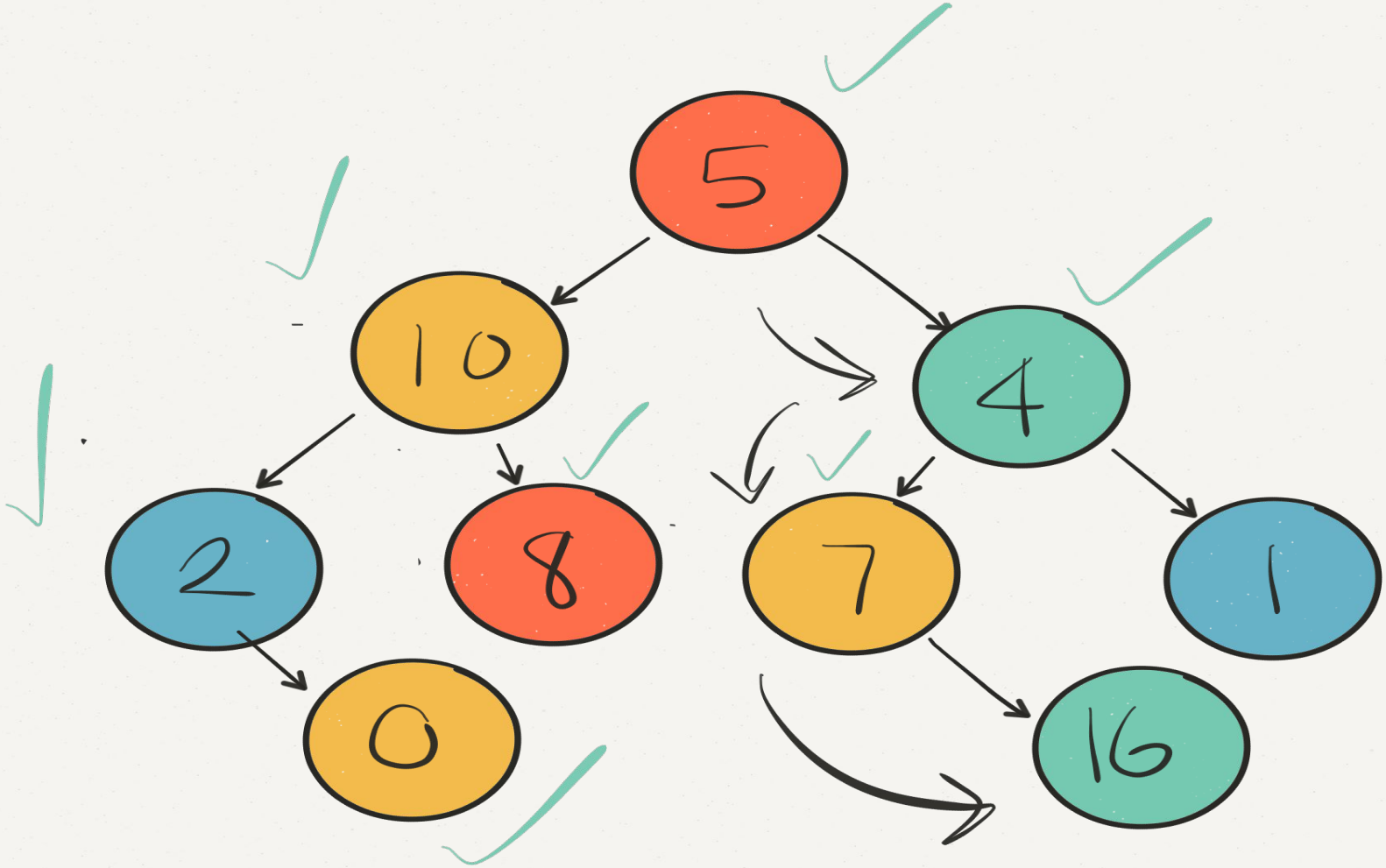

5
10
4
2
8
7
1
0
16

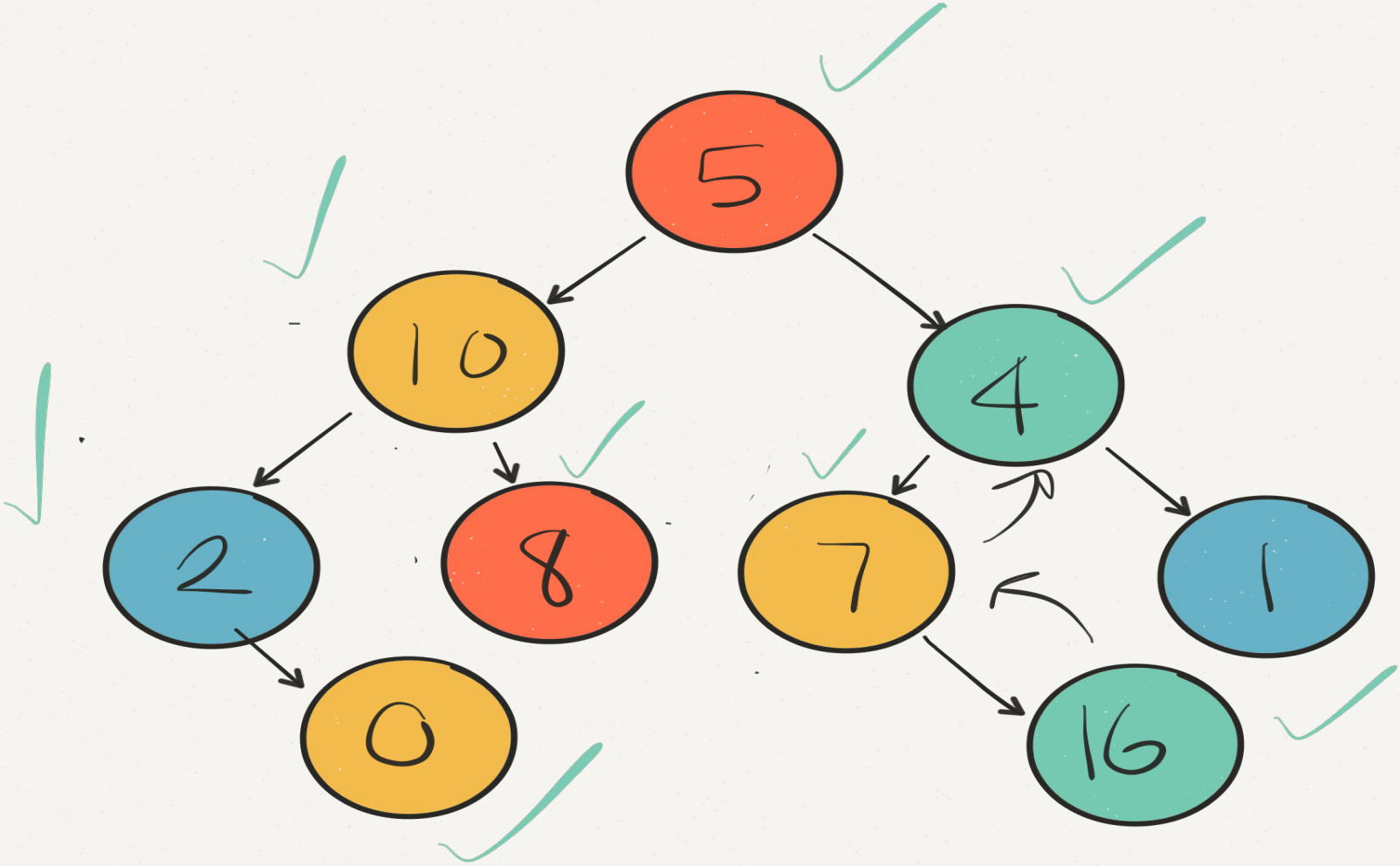

5
10
4
2
8
7
1
0
16

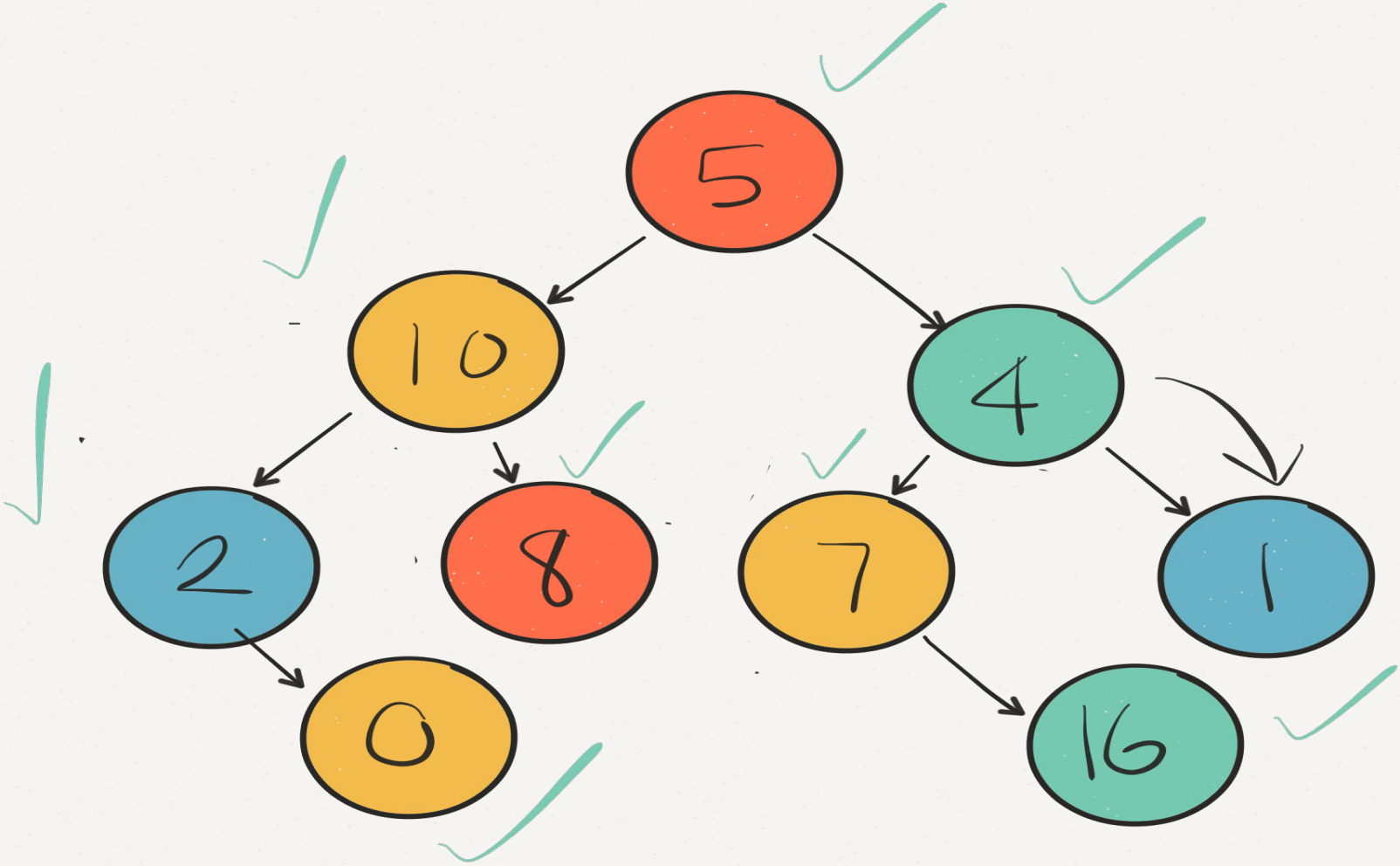

5
10
4
2
8
7
1
0
16

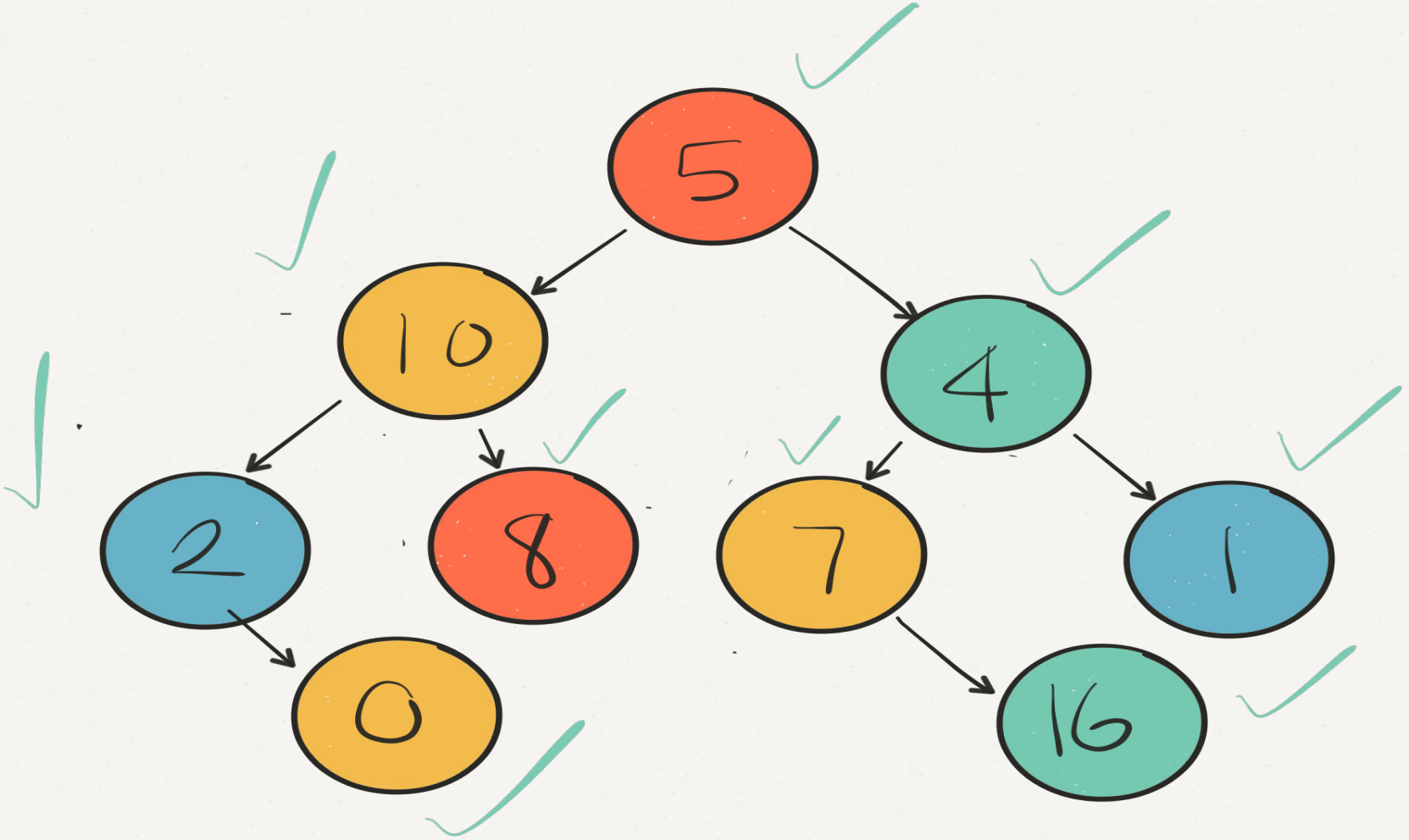

5
10
4
2
8
7
1
0
16


5
10
4
2
8
7
1
0
16


5
10
4
2
8
7
1
0
16


5
10
4
2
8
7
1
0
16


5
10
4
2
8
7
1
0
16


5
10
4
2
8
7
1
0
16

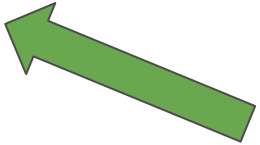
Search method for tree class

```
def search(val)
  @value = val




end
```

set scope variable

Search method for tree class

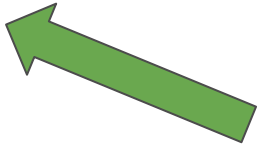
```
def search(val)
  @value = val

   def traverse(current)



  end


end
```



instantiate helper method

Search method for tree class

```
def search(val)
  @value = val

   def traverse(current)



  end

   return traverse(@root)
end
```



return a call to the helper method

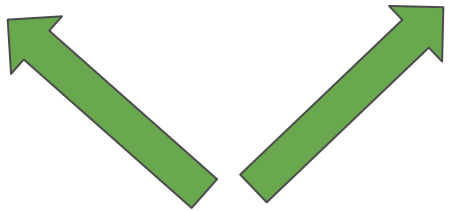
Search method for tree class

```
def search(val)
  @value = val

   def traverse(current)
    return false if current == nil
    return true if current.val == @value

  end

   return traverse(@root)
end
```

base cases: return false if current node is empty
return true if current value is a match

Search method for tree class

```
def search(val)
  @value = val

   def traverse(current)
    return false if current == nil
    return true if current.val == @value
    return traverse(current.left_child) || traverse(current.right_child)
  end

   return traverse(@root)
end
```



recursive case: call itself on both children

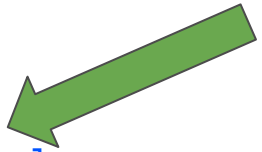
Search method for tree class

```ruby
def search(val)
  @value = val

   def traverse(current)
    return false if current == nil
    return true if current.val == @value
    return traverse(current.left_child) || traverse(current.right_child)
  end

   return traverse(@root)
end
```

pre-order, in-order, or post-order?

Search method for tree class

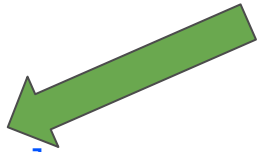
```ruby
def search(val)
  @value = val

   def traverse(current)
    return false if current == nil
    return true if current.val == @value
    return traverse(current.left_child) || traverse(current.right_child)
  end

   return traverse(@root)
end
```



operations done before recursive cases called on both children

Search method for tree class

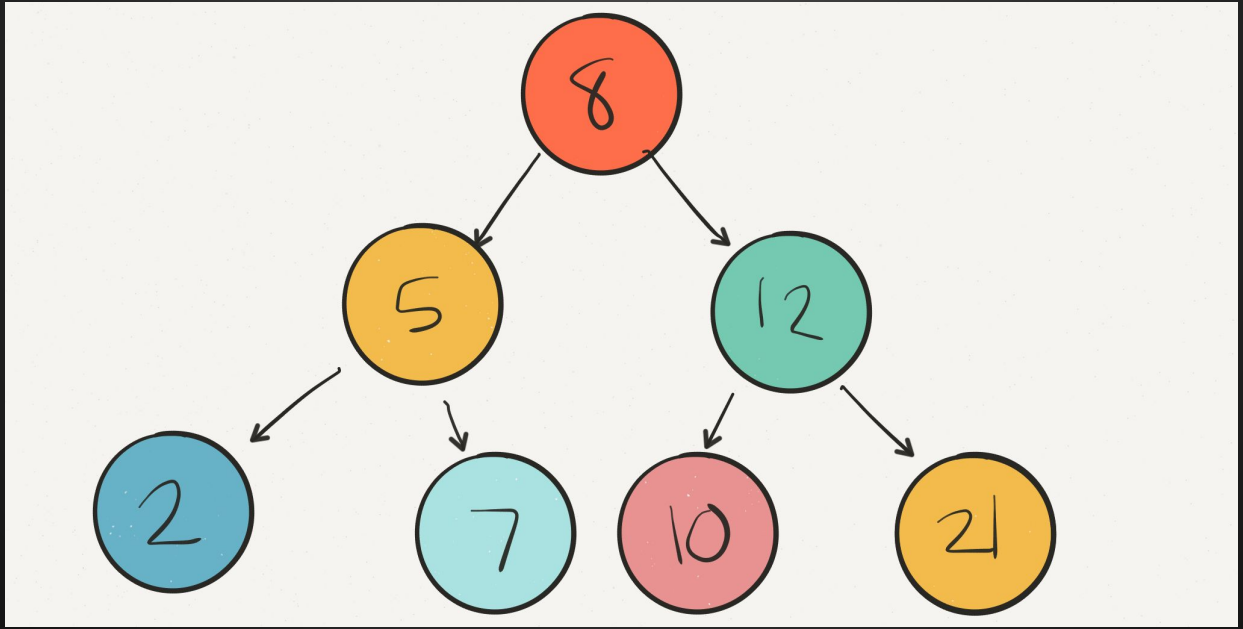
```ruby
def search(val)
  @value = val

   def traverse(current)
     return false if current == nil
     return true if current.val == @value
     return traverse(current.left_child) || traverse(current.right_child)
   end

   return traverse(@root)
end
```



pre-order!

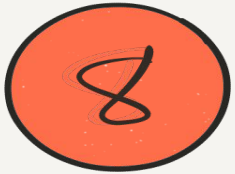
Binary Search Tree (BST)

Binary trees that keep nodes in sorted order

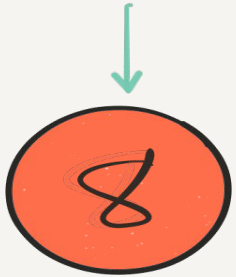


Binary Search Tree (BST)

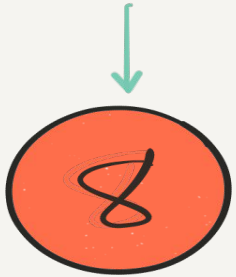
- Start at root node
- Place node with lower value to left
- Place node with higher value to right
- If there is already a node
  - recursing downward until open spot


8

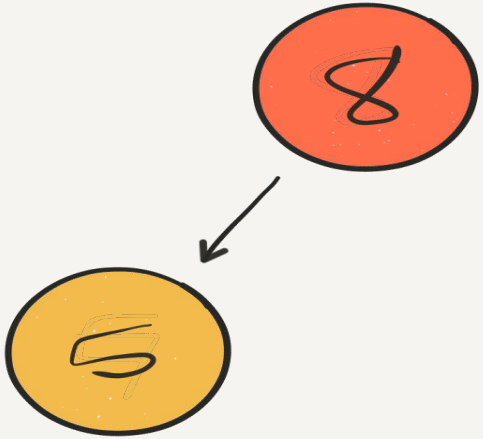
INSERT



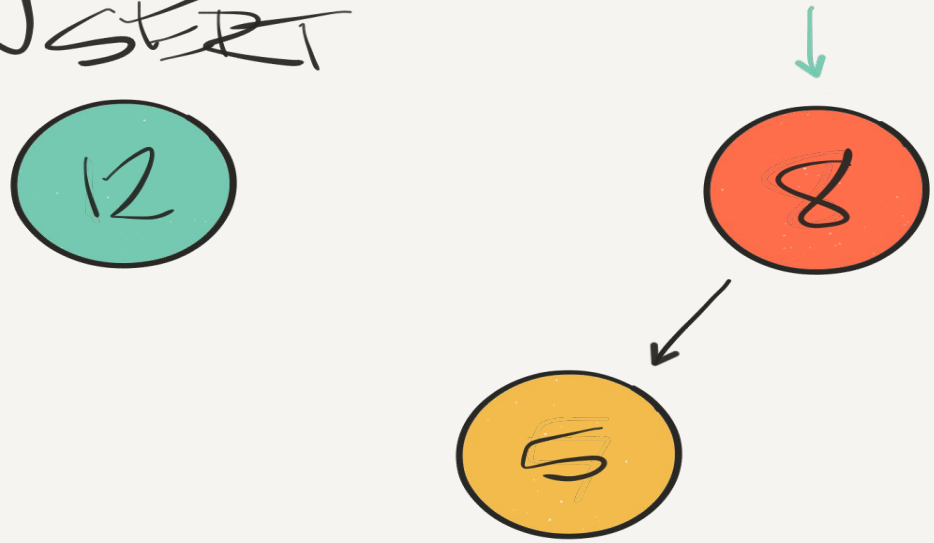
INSERT

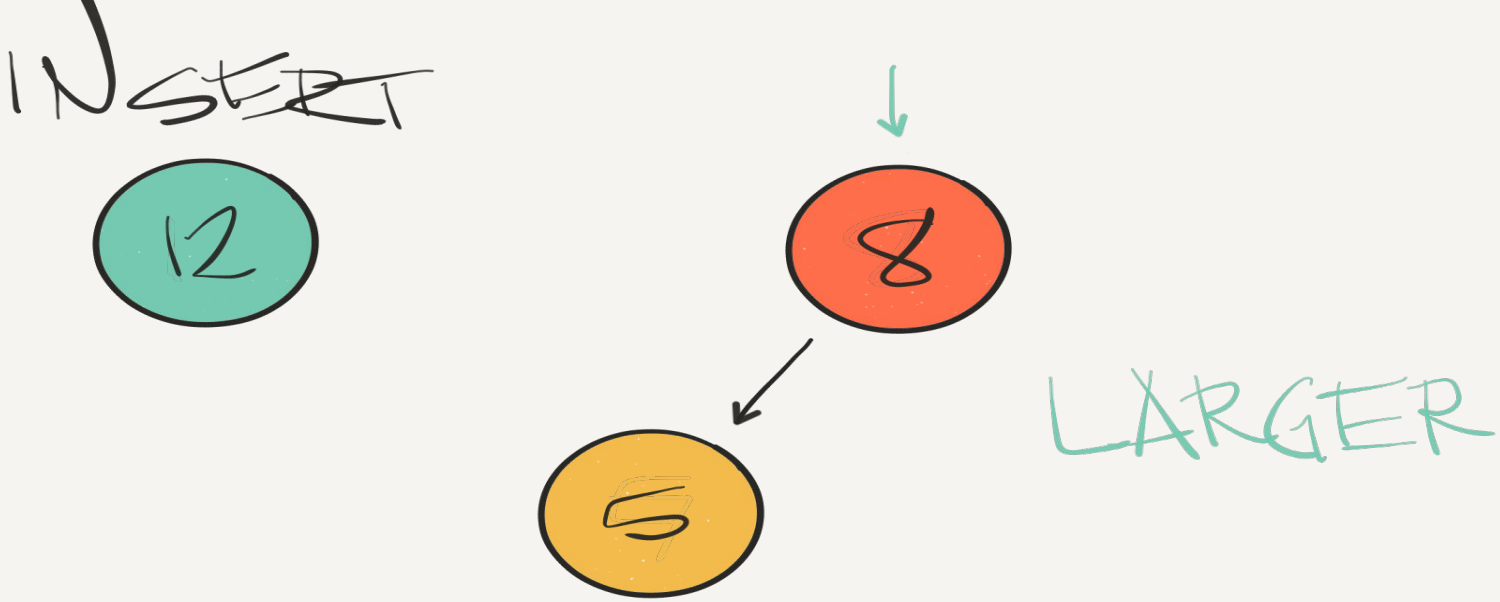


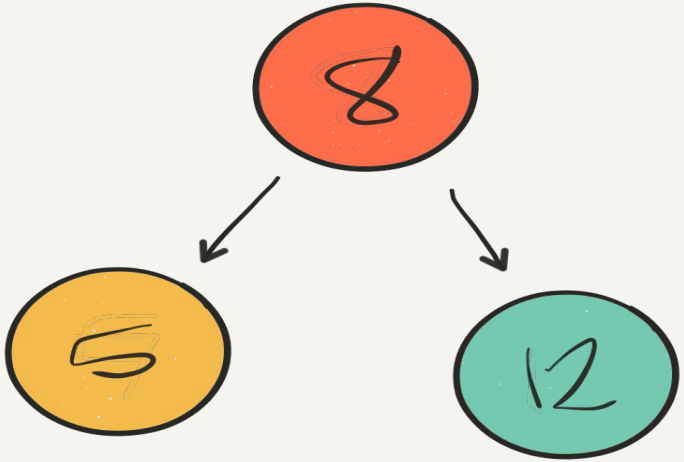
SMALLER


8
5

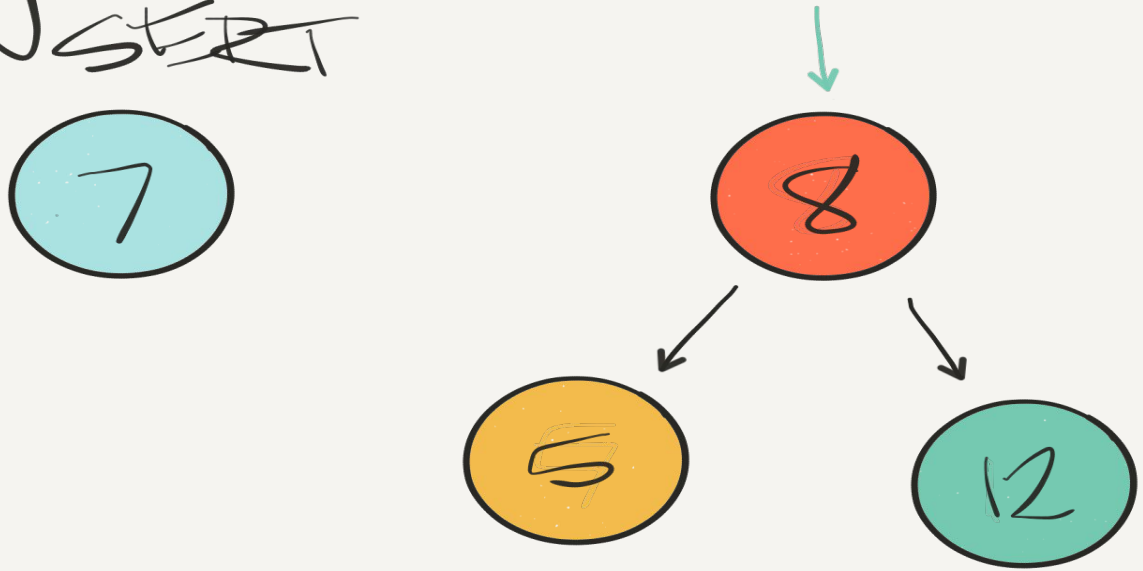
INSERT

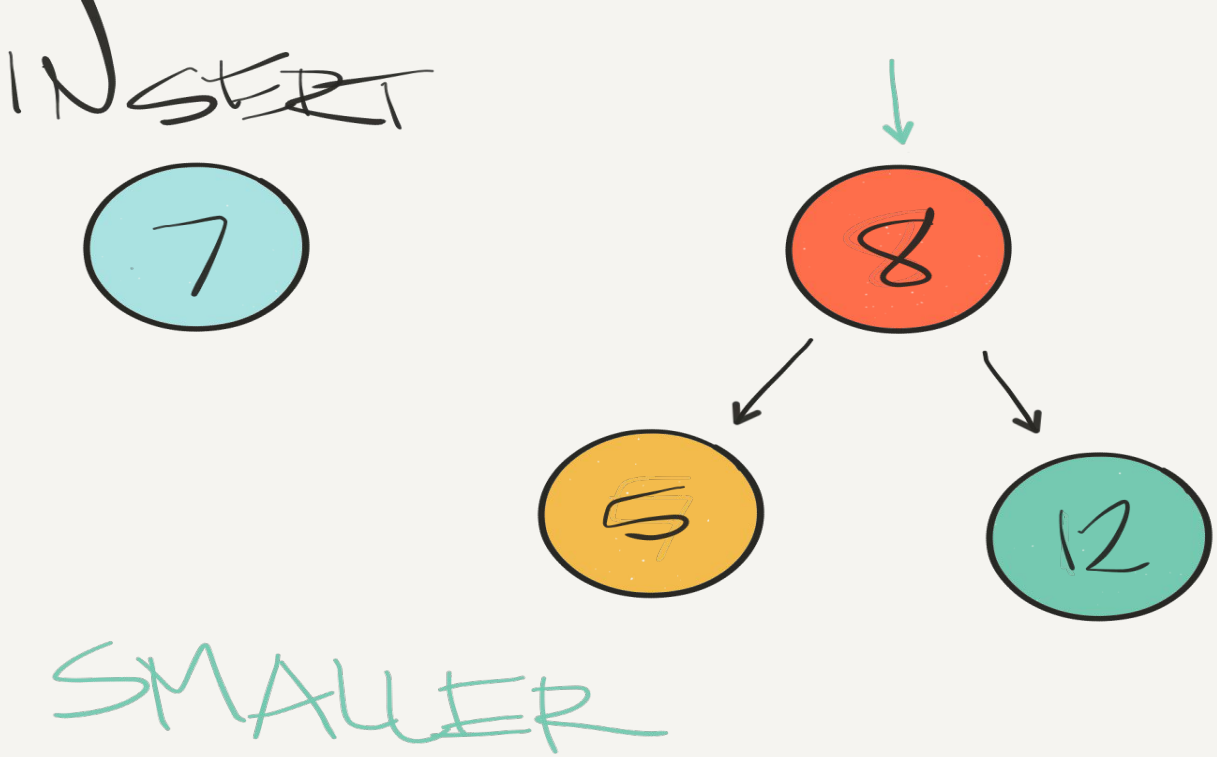




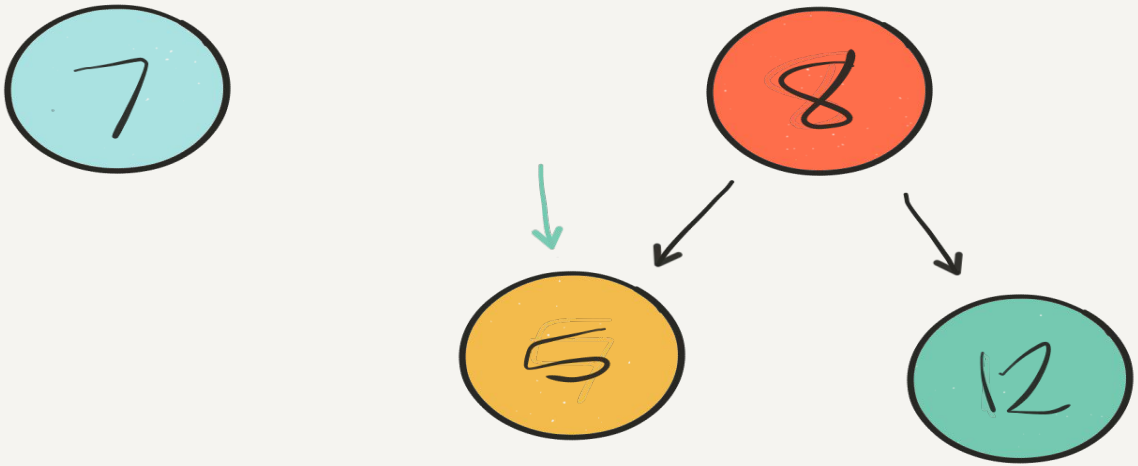

8
5
12

INSERT

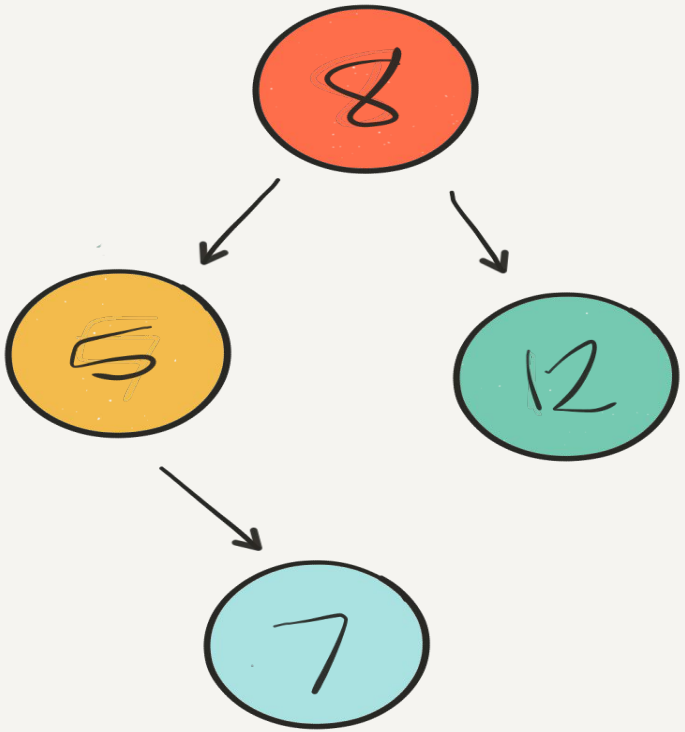


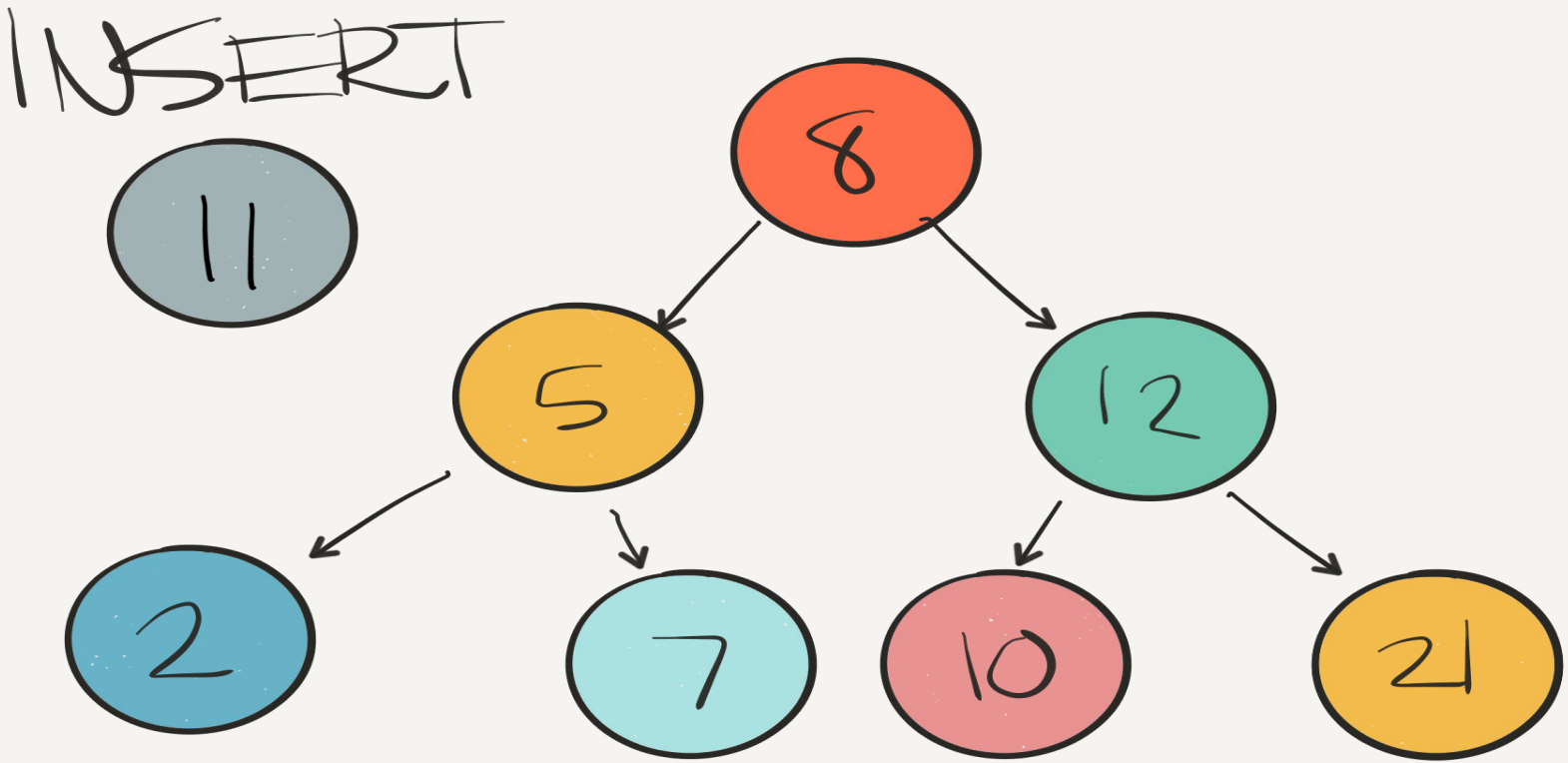


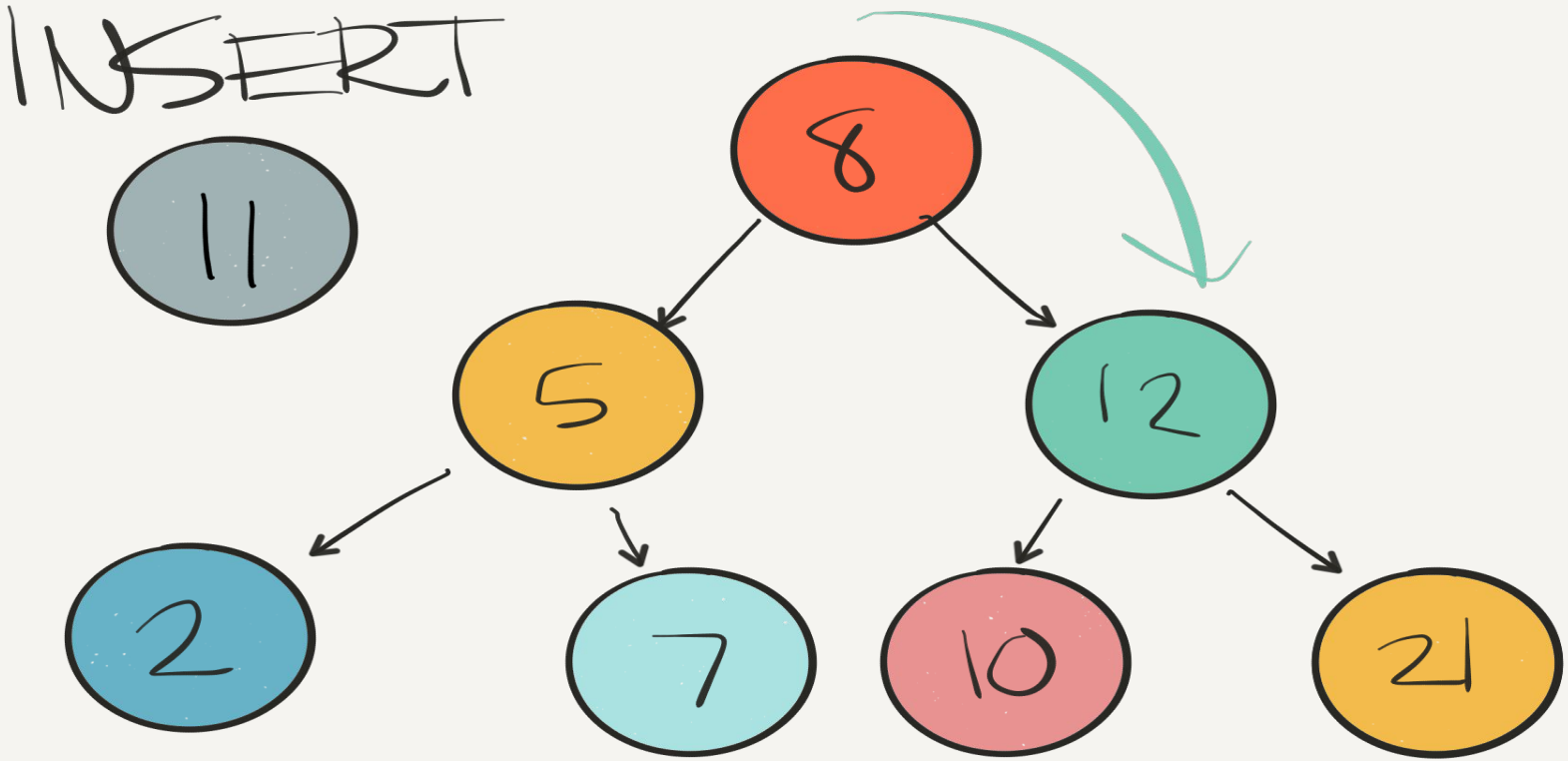
INSERT

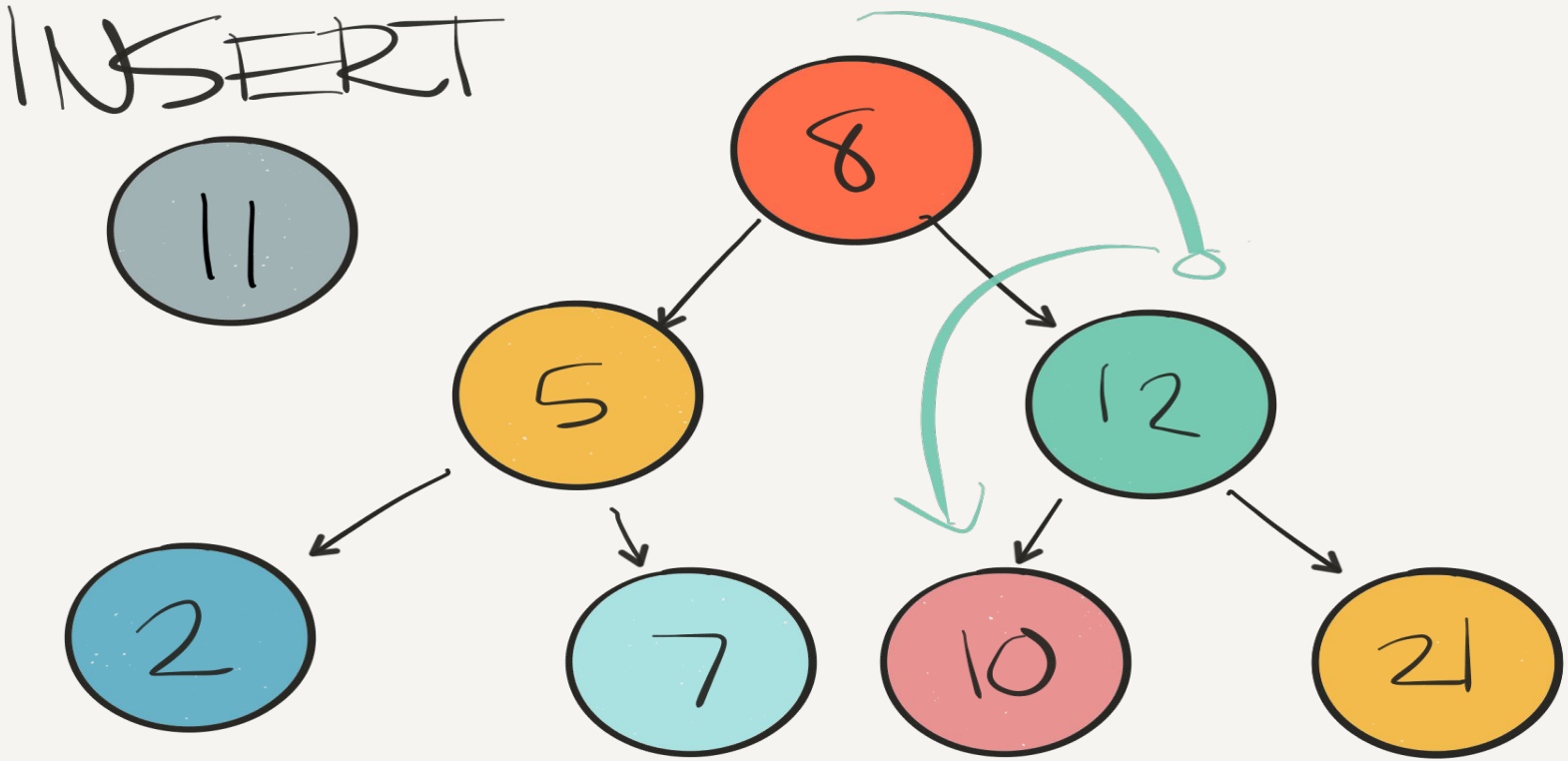


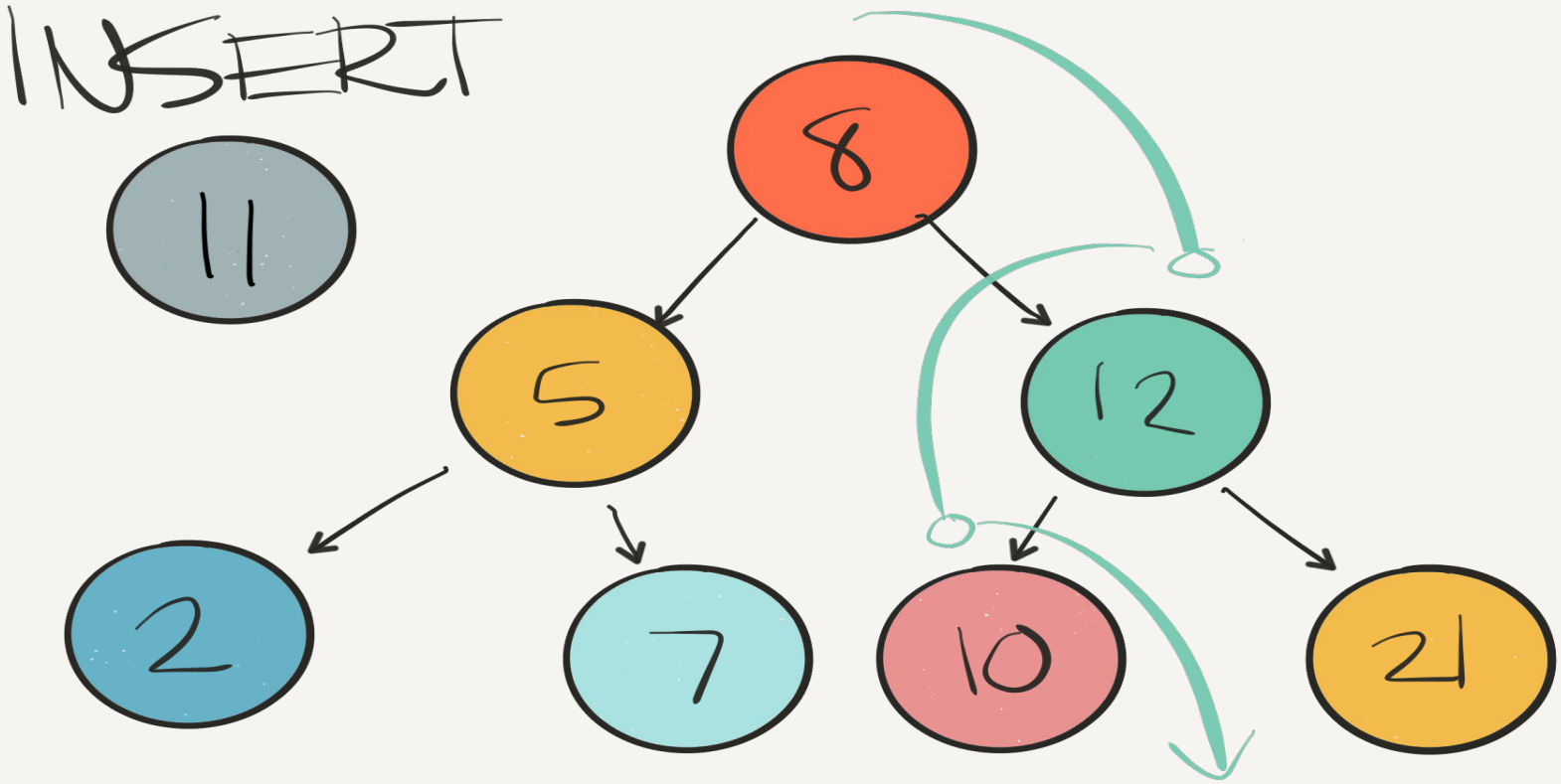
LARGER


8
5
12
7




INSERT
11
8
5
12
2
7
10
21

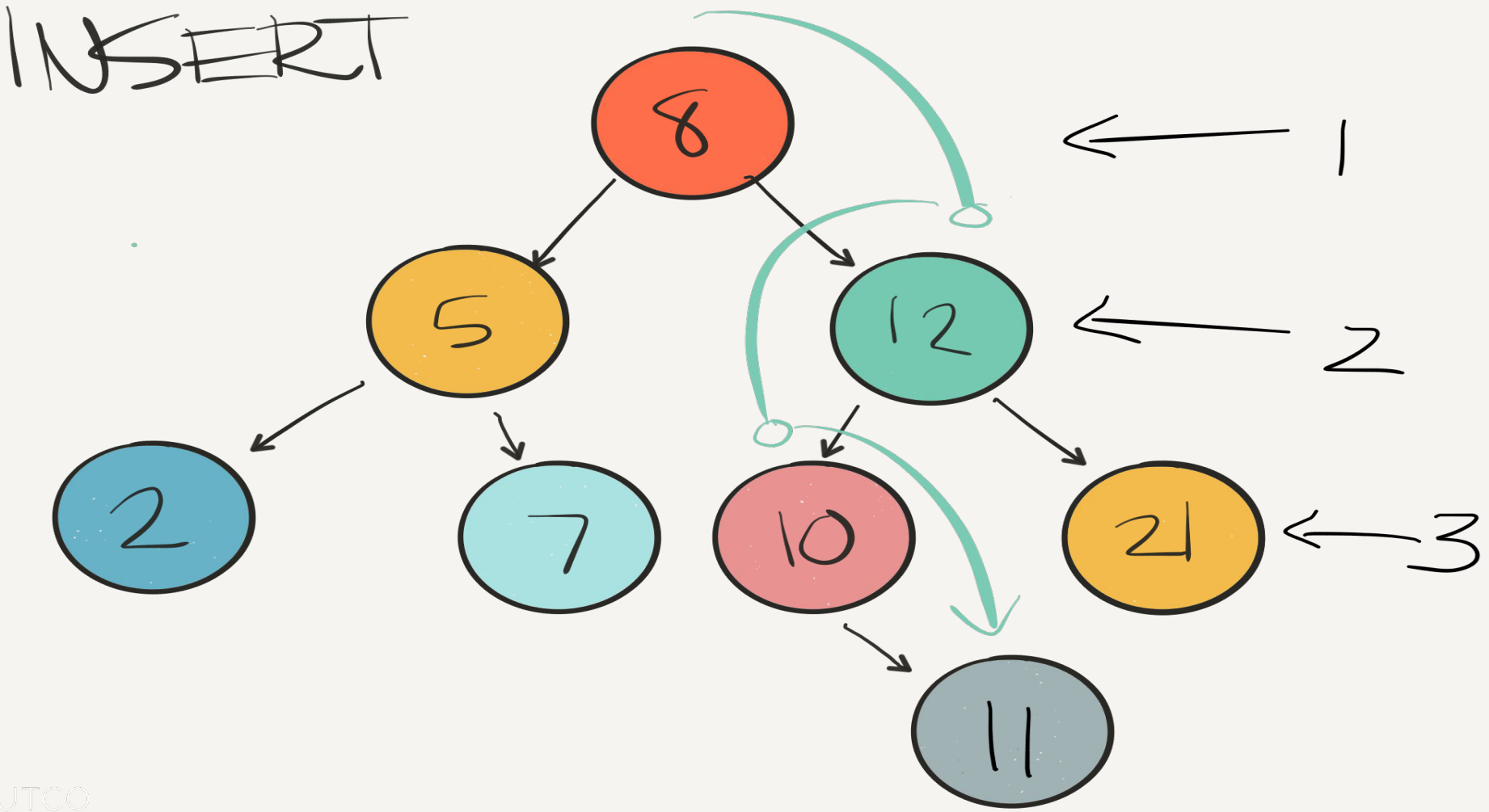

INSERT
11
8
5
12
2
7
10
21


INSERT
11
8
5
12
2
7
10
21

Time Complexity - BST

Average case

- Insert ?


INSERT
8
5
12
2
7
10
21
11
1
2
3

Time Complexity - BST

Average case

- Insert
  - O(logn)
- Search
  - O(logn)
- Delete
  - O(logn)

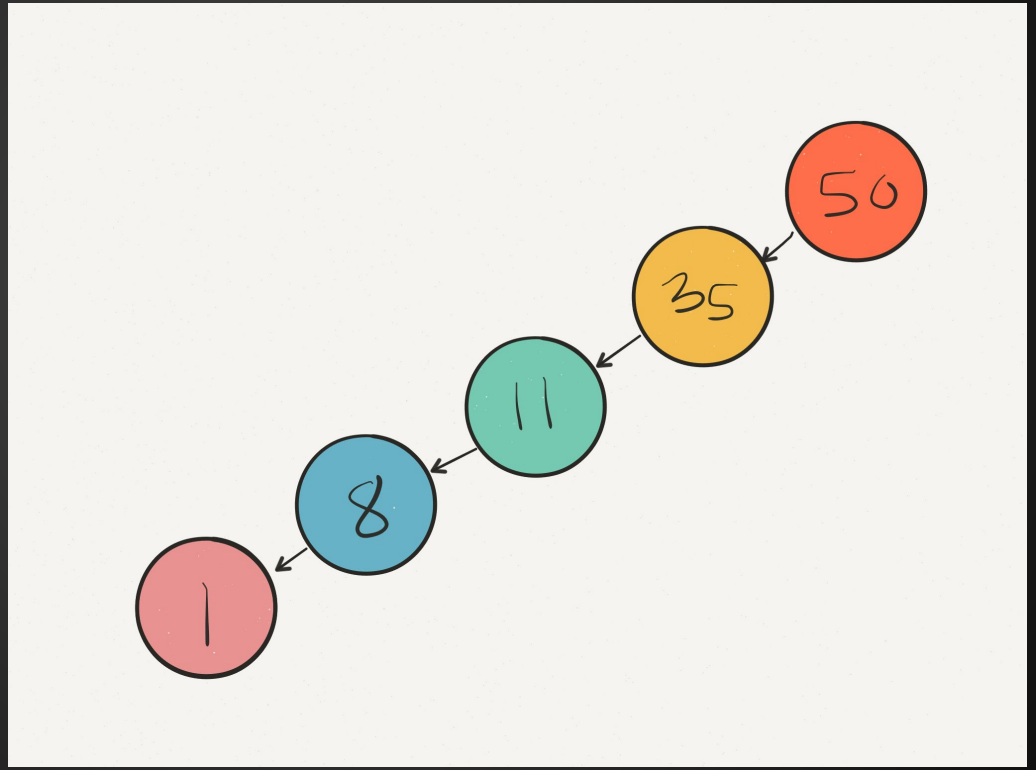
Time Complexity - BST

Worst case

- Insert
  - O(n)
- Search
  - O(n)
- Delete
  - O(n)

Time Complexity - BST

Worst case:
Imbalanced
Tree



How do we fix that?

- Balanced search trees
- Has logic to reorder nodes in order to keep elements balanced
- Examples
  - B-trees
  - AVL Trees
  - Red-black Trees, etc.

Advantages of BST

- Fast search average O(logn)
- Typically less time than arrays to keep values sorted
- Expand and contract memory as needed

Disadvantages of BST

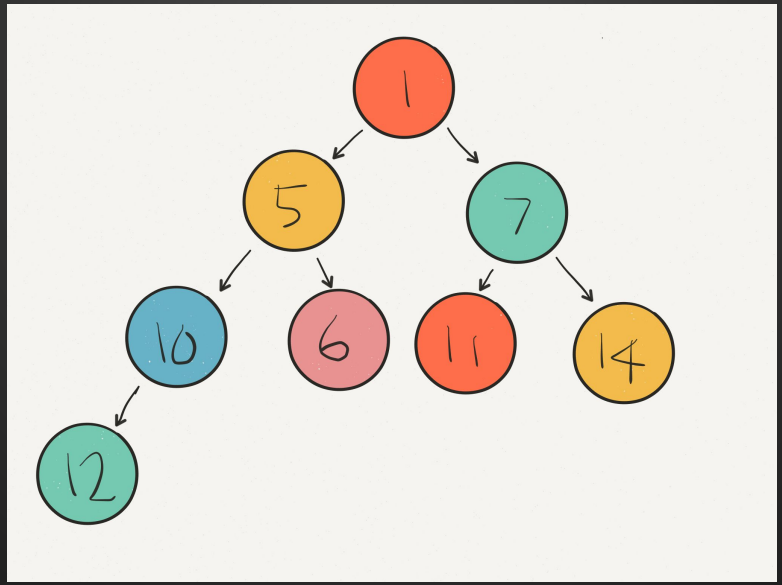
- Time complexity depends on order of insertion of values
- Insertion and search requires many comparisons

Example Uses of BSTs

- Search applications
  - Lots of insertions and deletions
- Lookup tables
- Interview questions are common

Binary Heap

Return the max value or min value



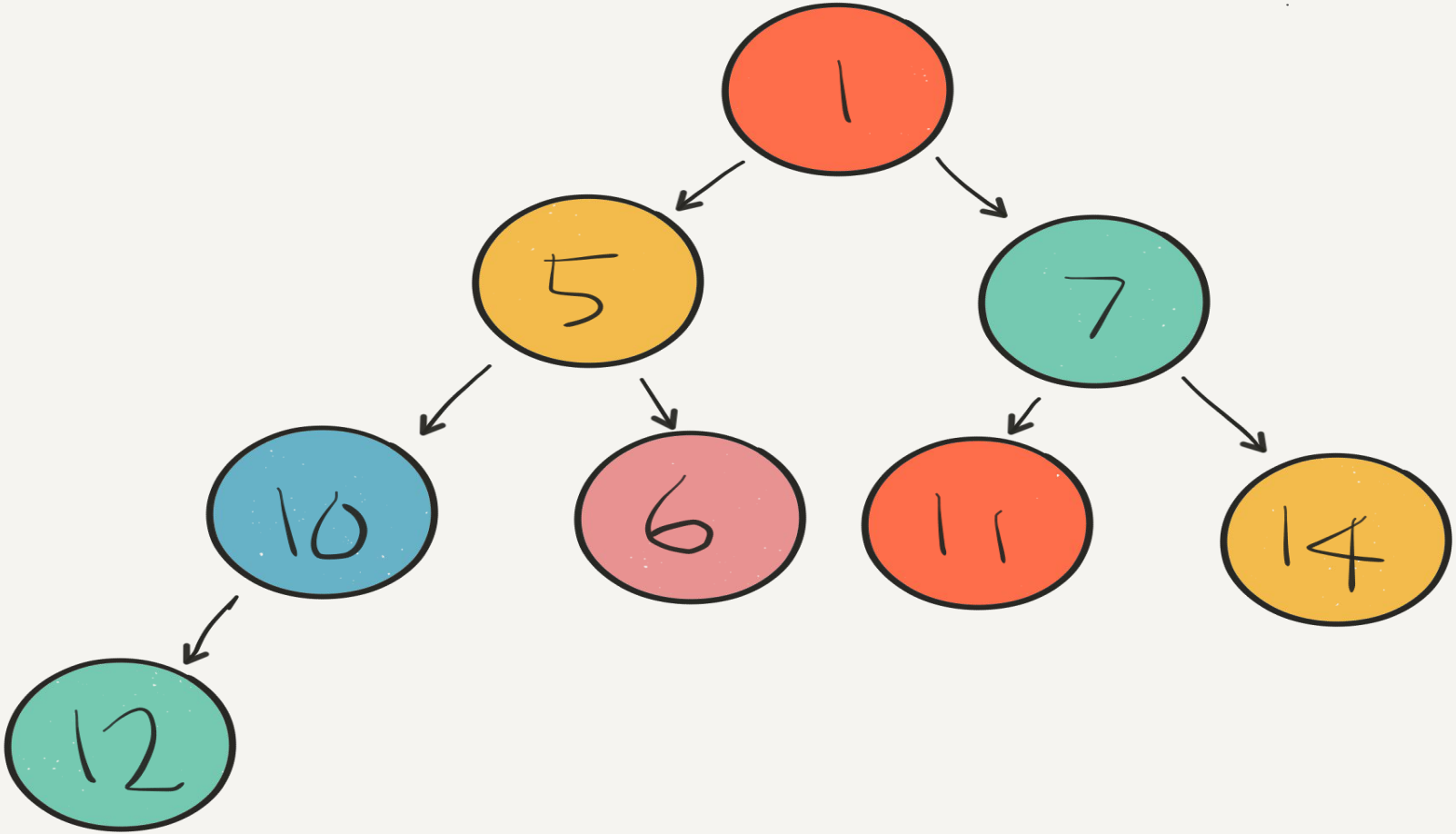
Binary Heap

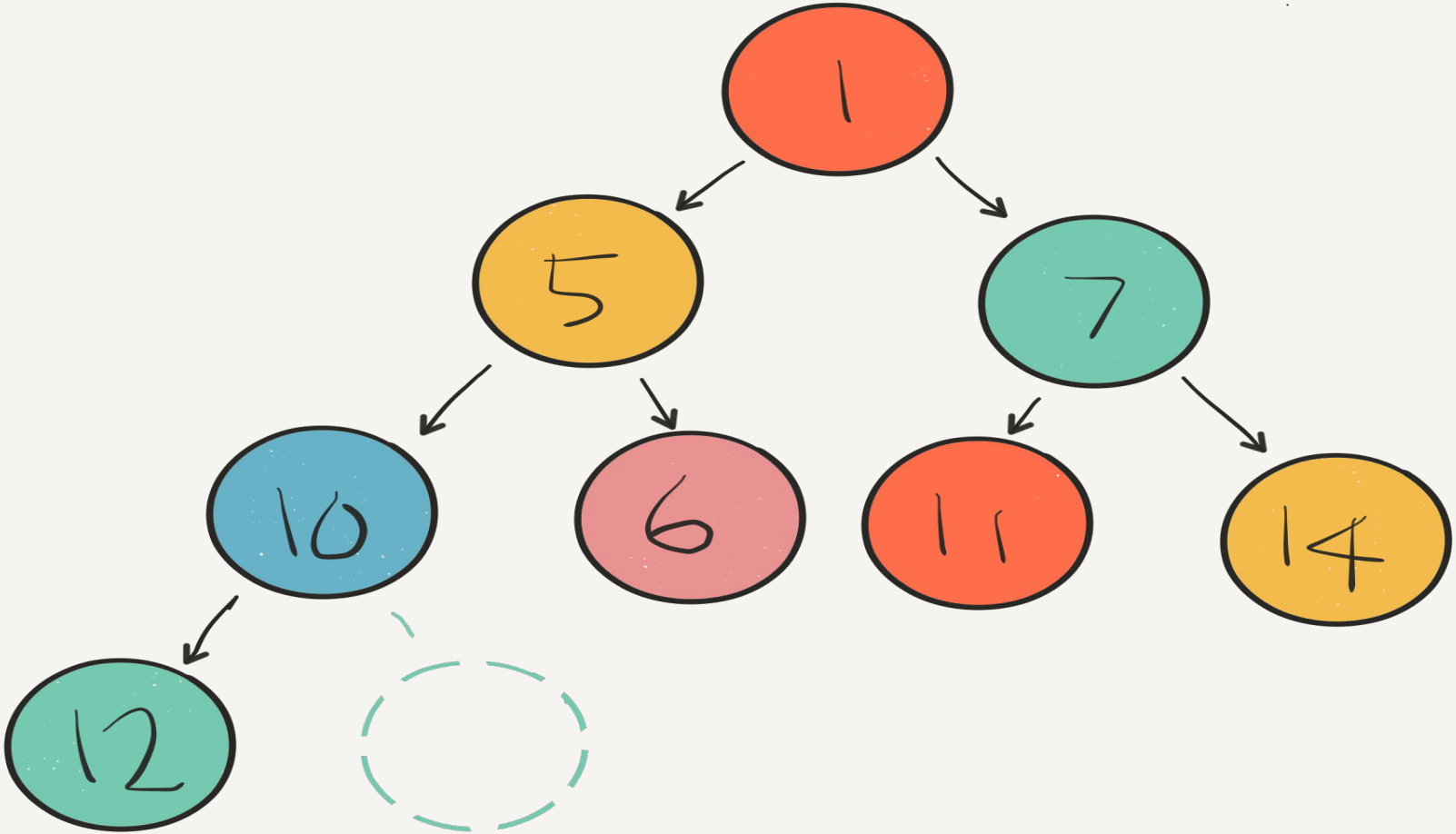
Special binary trees with these properties

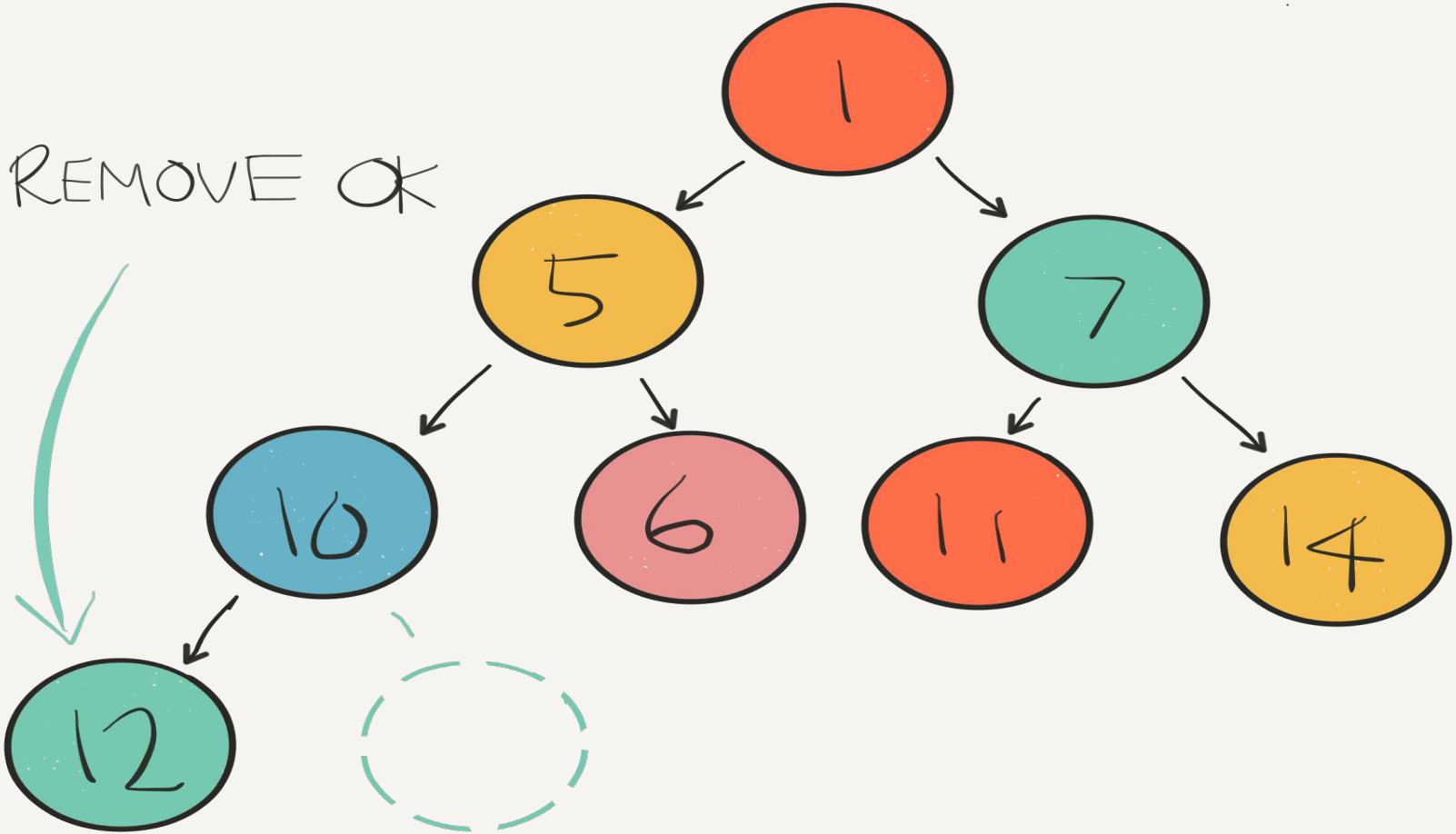
- Must be complete trees

Complete Trees

- Nodes must have both children before next node can fill children
- Can only add children to oldest incomplete node
- Can only remove youngest node


1
5
7
10
6
11
14
12

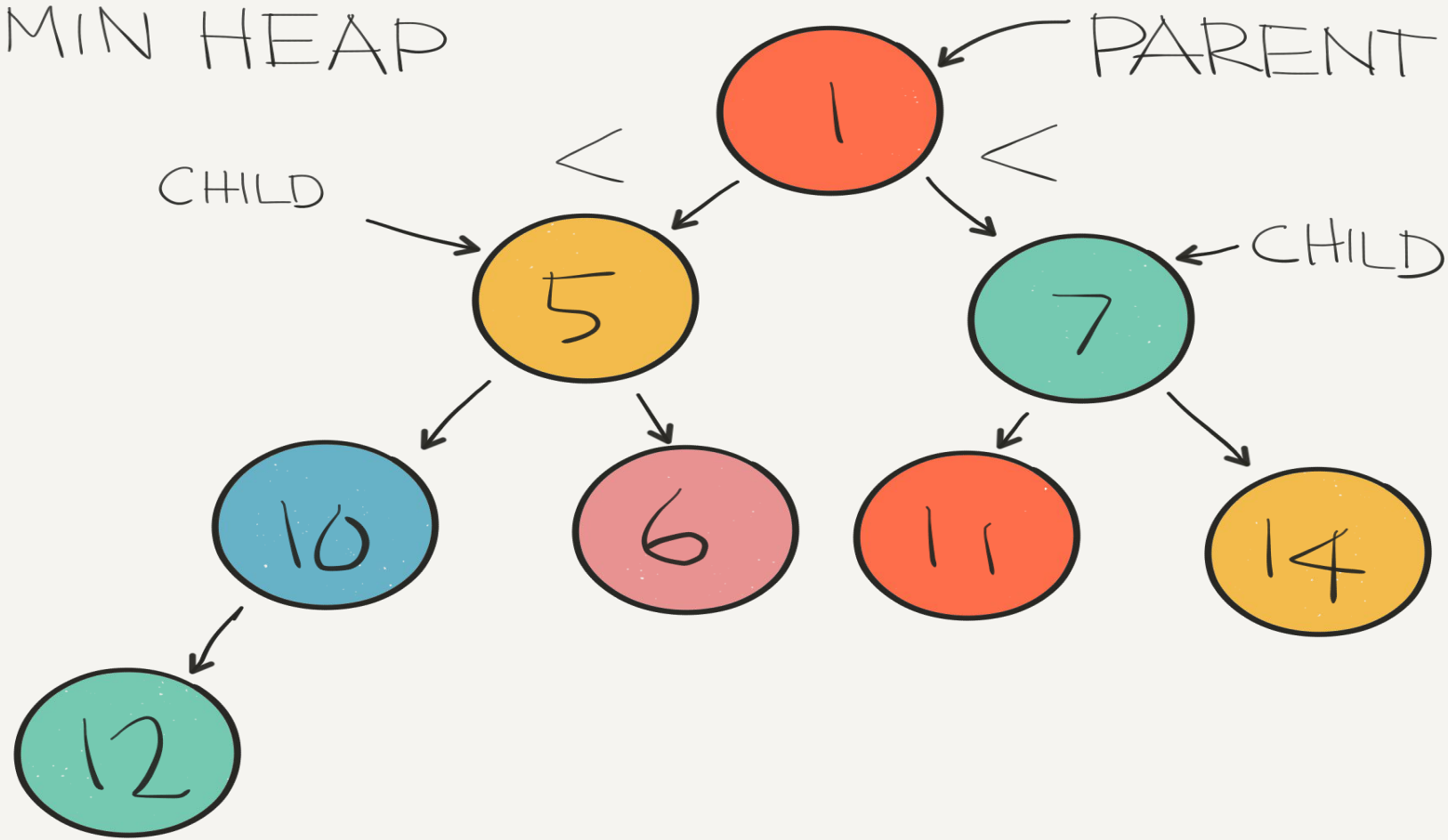

1
5
7
10
6
11
14
12


REMOVE ok
1
5
7
10
6
11
14
12

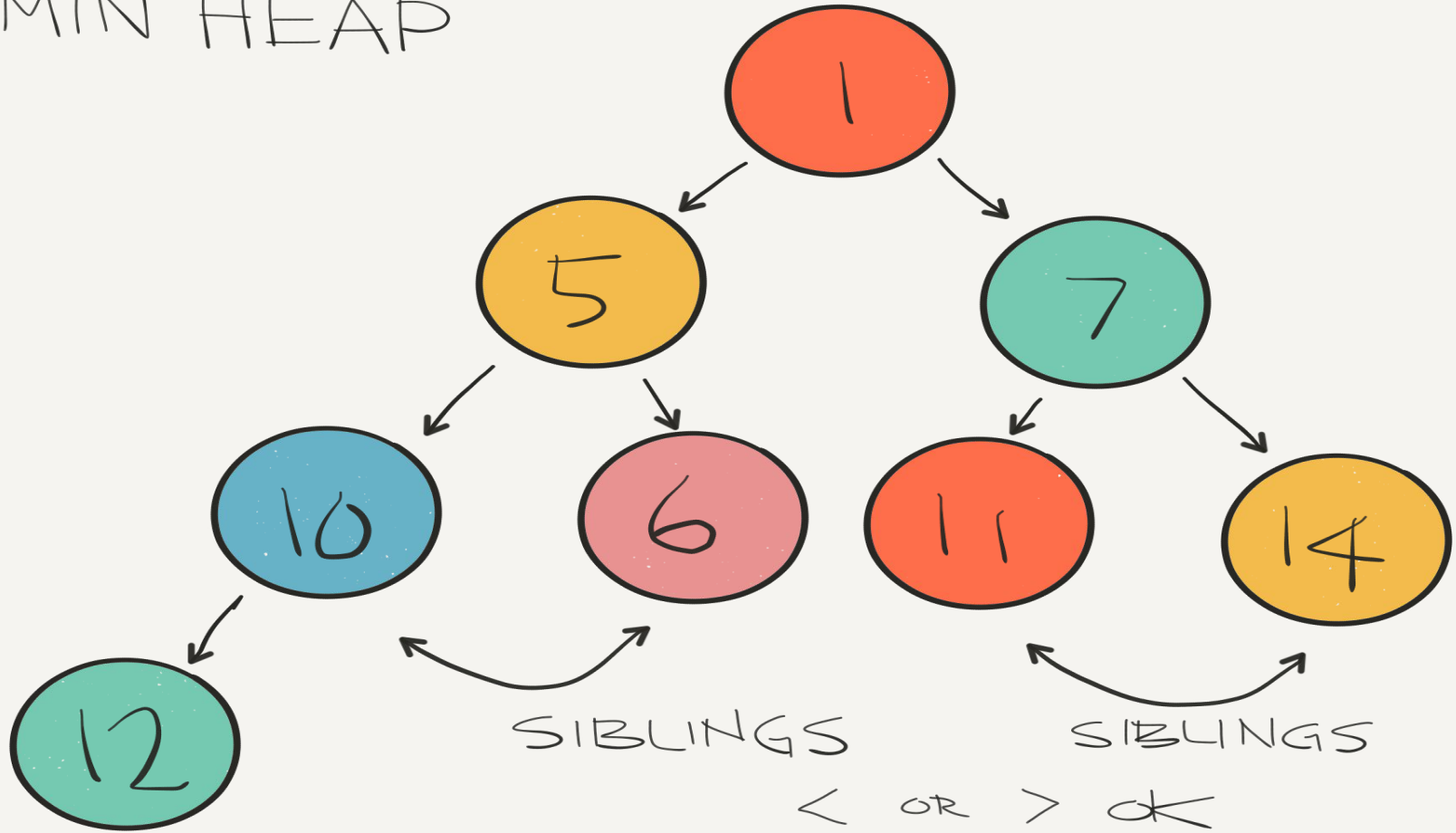
Binary Heaps

Special binary trees with these properties:

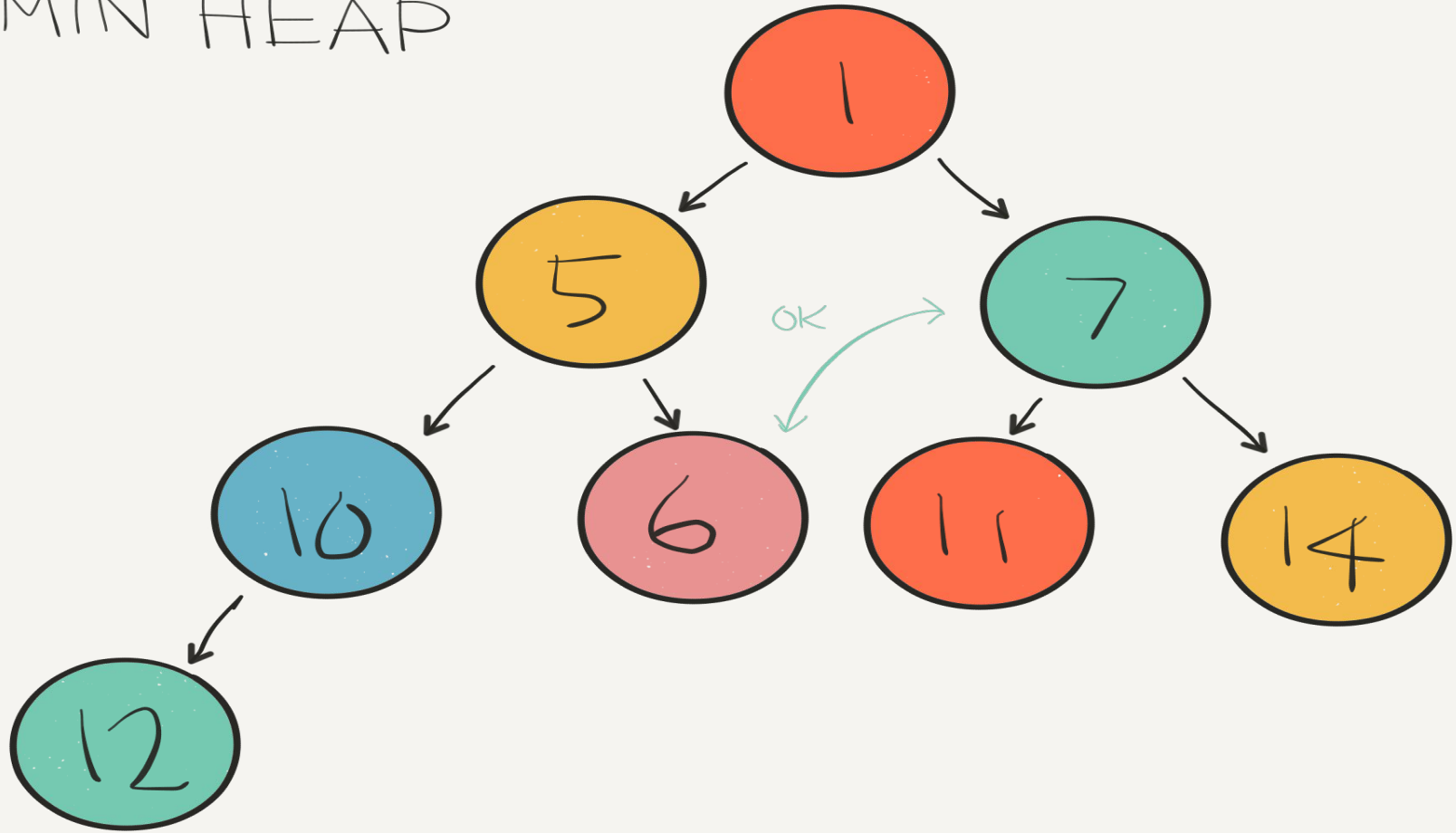
- Must be complete trees
- Parent-child must meet heap condition
  - Max Heap
    - Parent is larger than its 2 children
  - Min Heap
    - Parent is smaller than its 2 children


MIN HEAP
PARENT
1
<
<
CHILD
5
7
CHILD
10
6
11
14
12

MIN HEAP



MIN HEAP

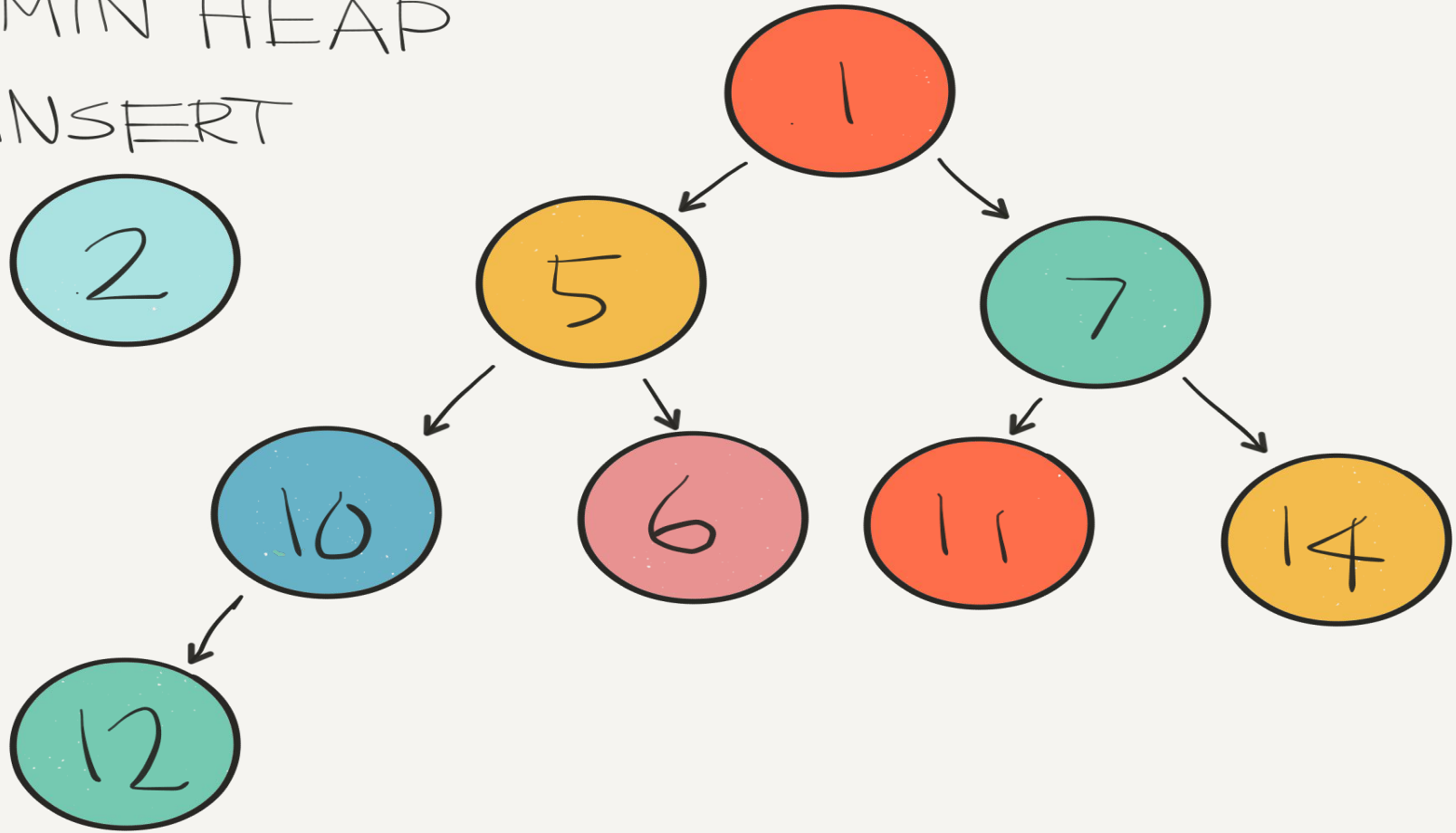


Binary Heaps

- Storage
  - Array, Linked List, Binary Tree
- Methods
  - Insert
  - Extract
- Helper methods
  - Bubble up
  - Bubble down
  - Swap

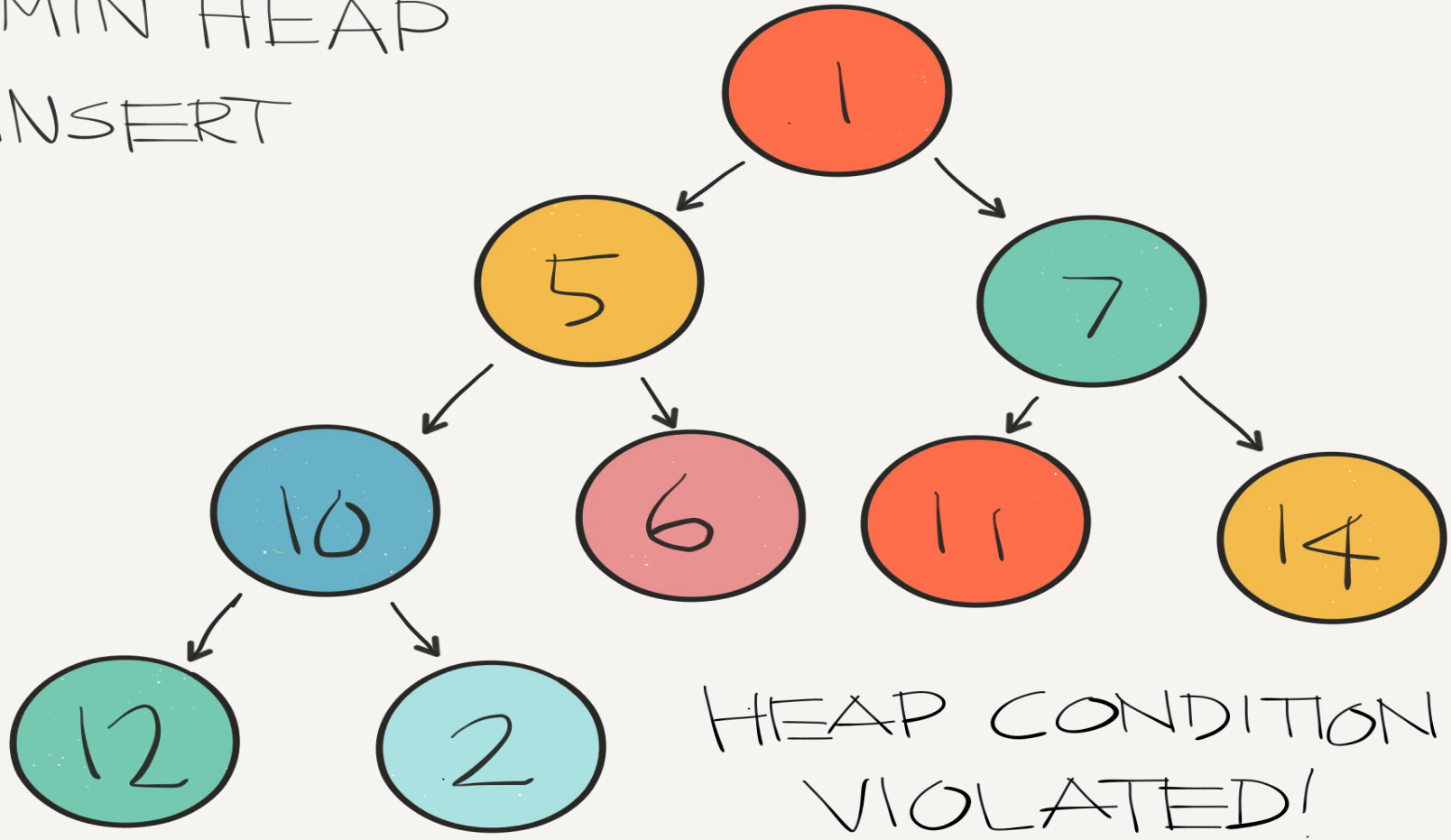
MIN HEAP

INSERT



MIN HEAP

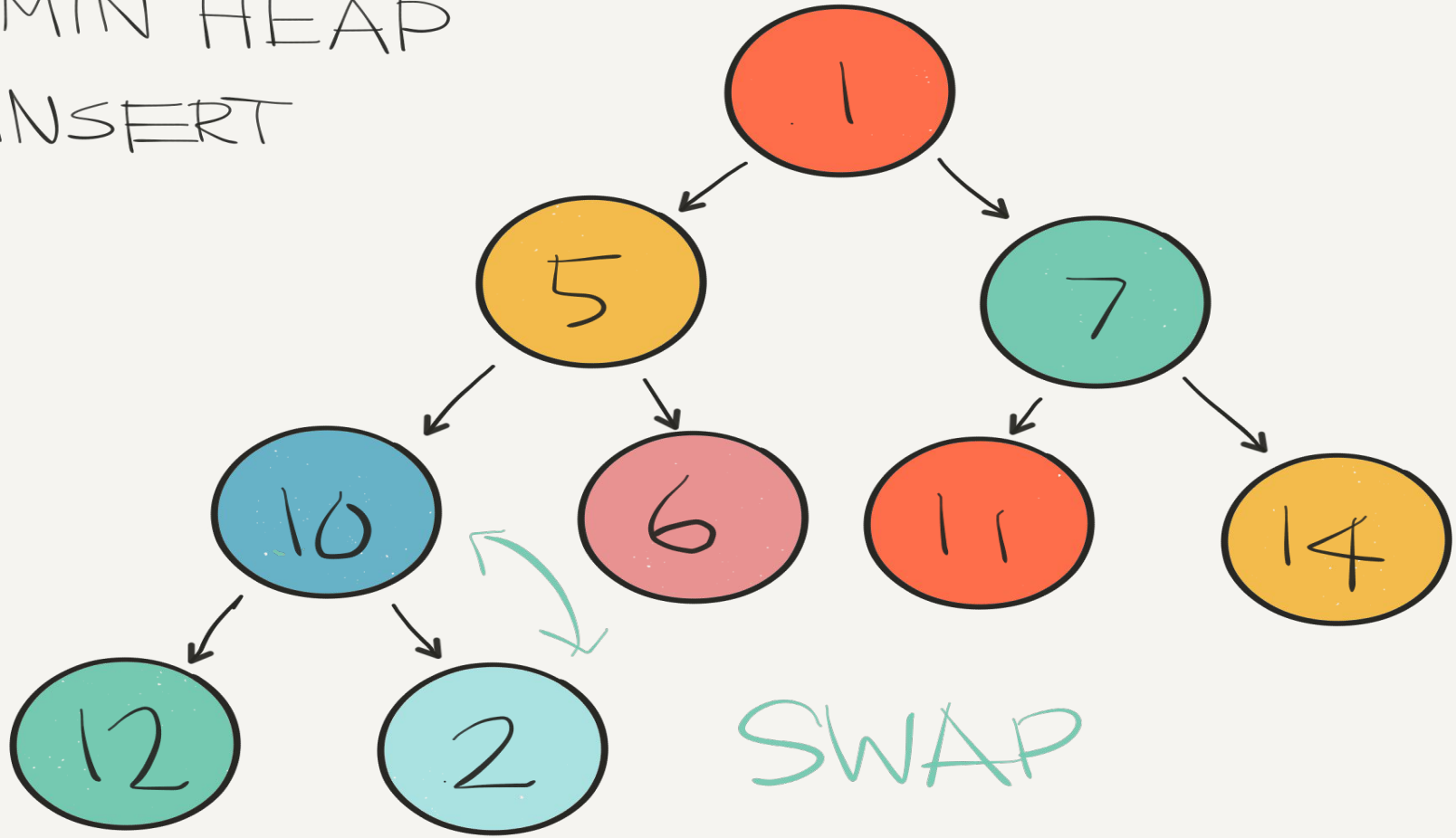
INSERT



HEAP CONDITION VIOLATED!

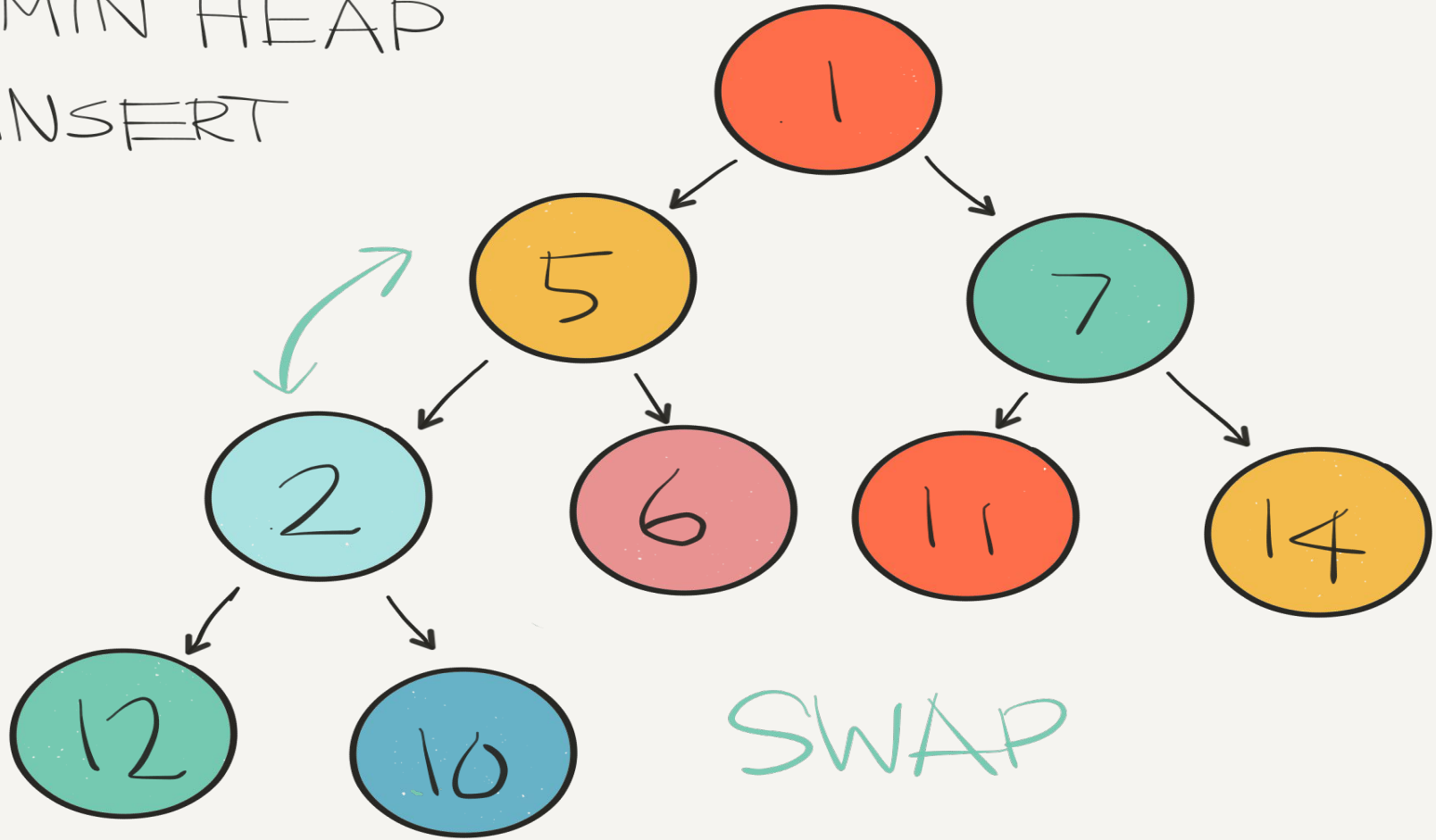
MIN HEAP

INSERT

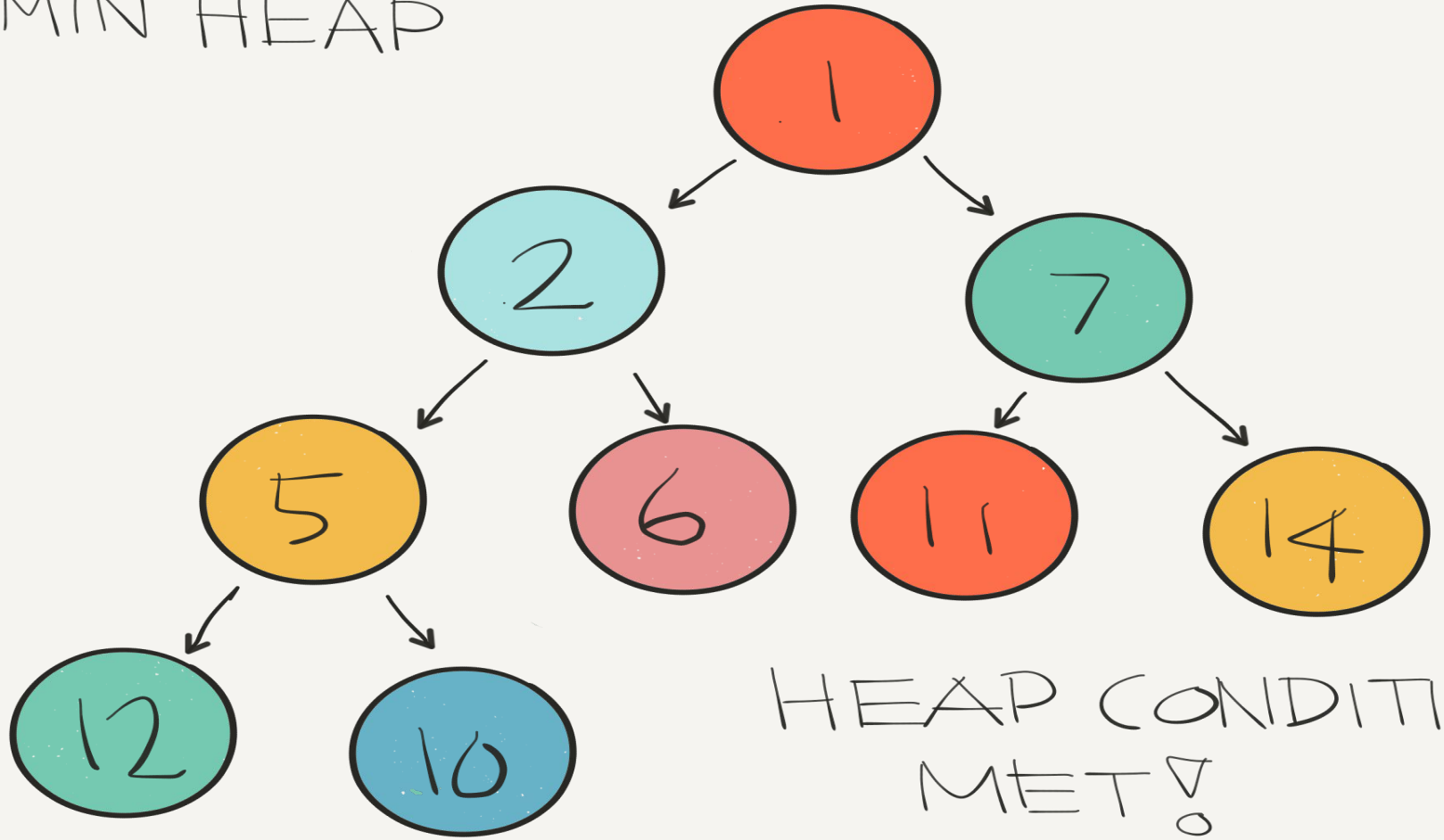


MIN HEAP

INSERT



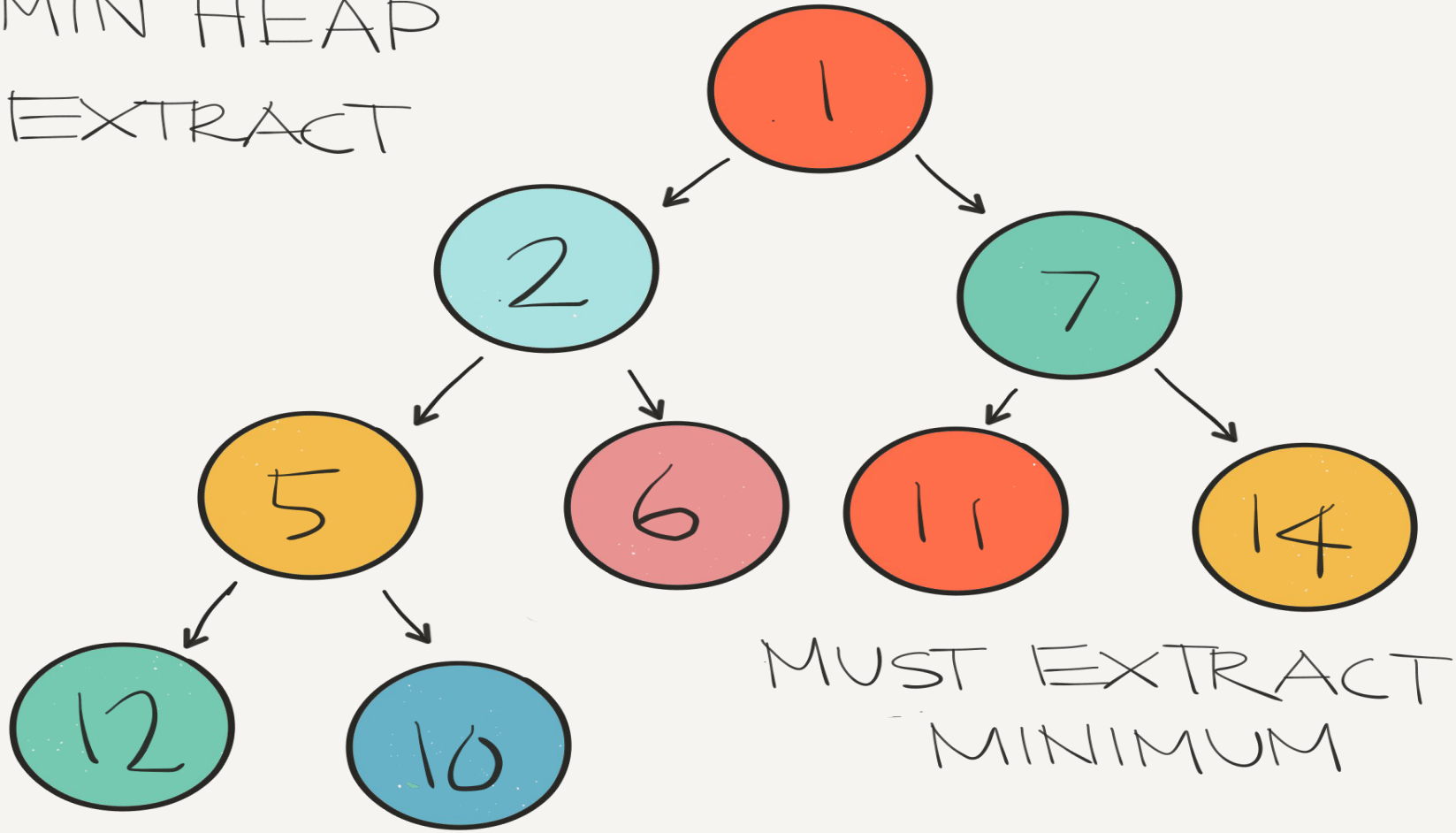
MIN HEAP



HEAP CONDITION MET!

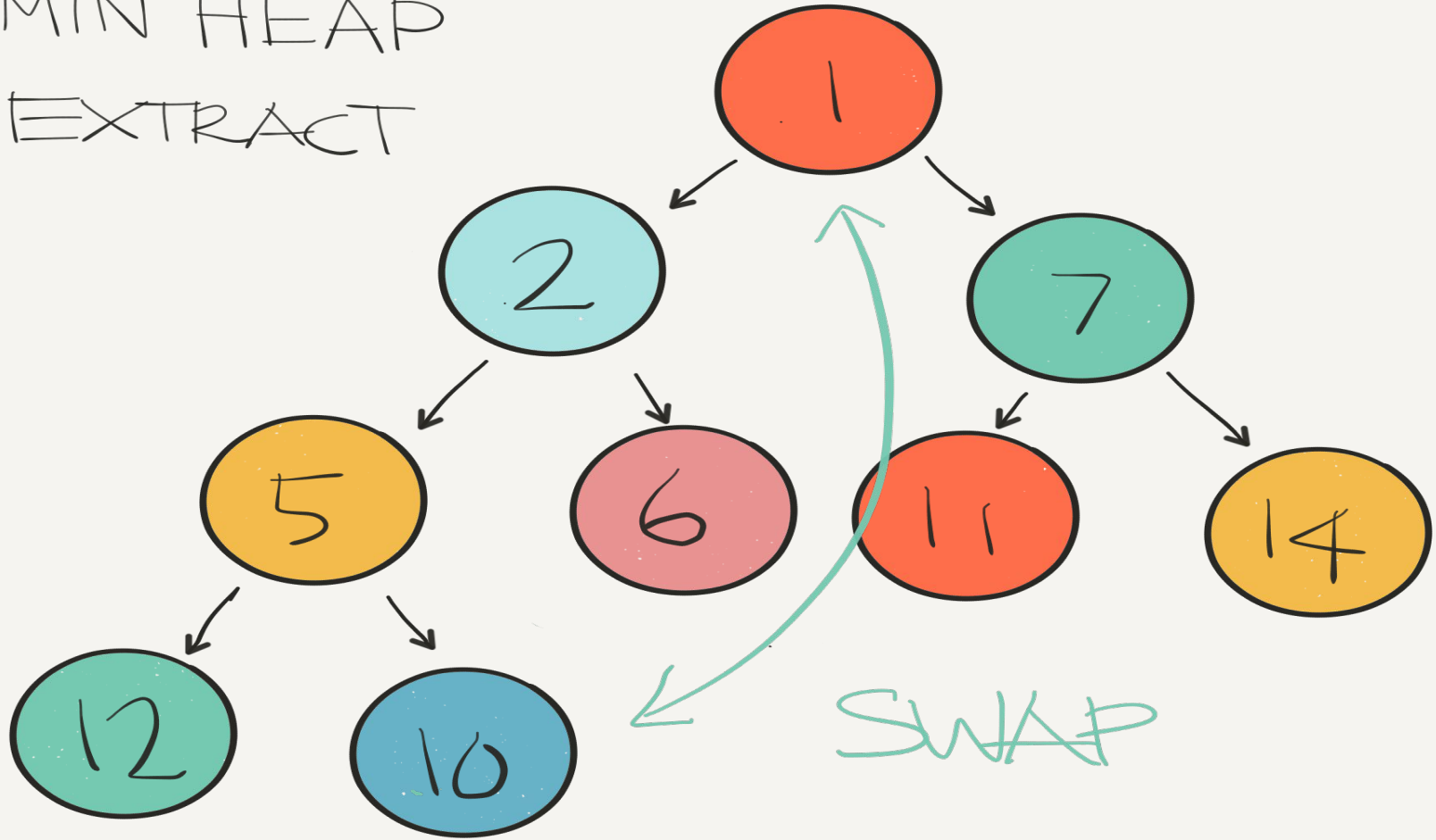
MIN HEAP

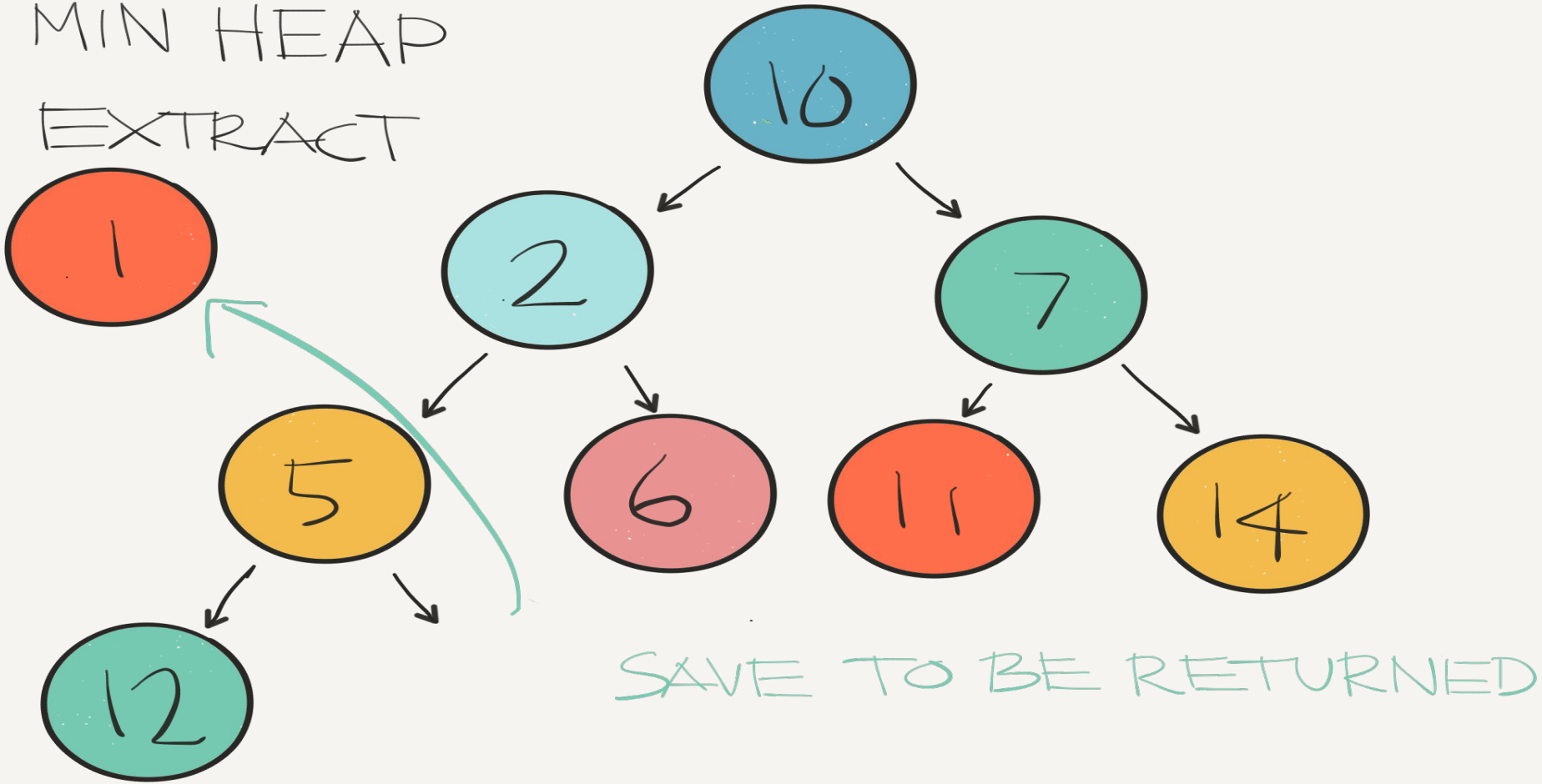
EXTRACT

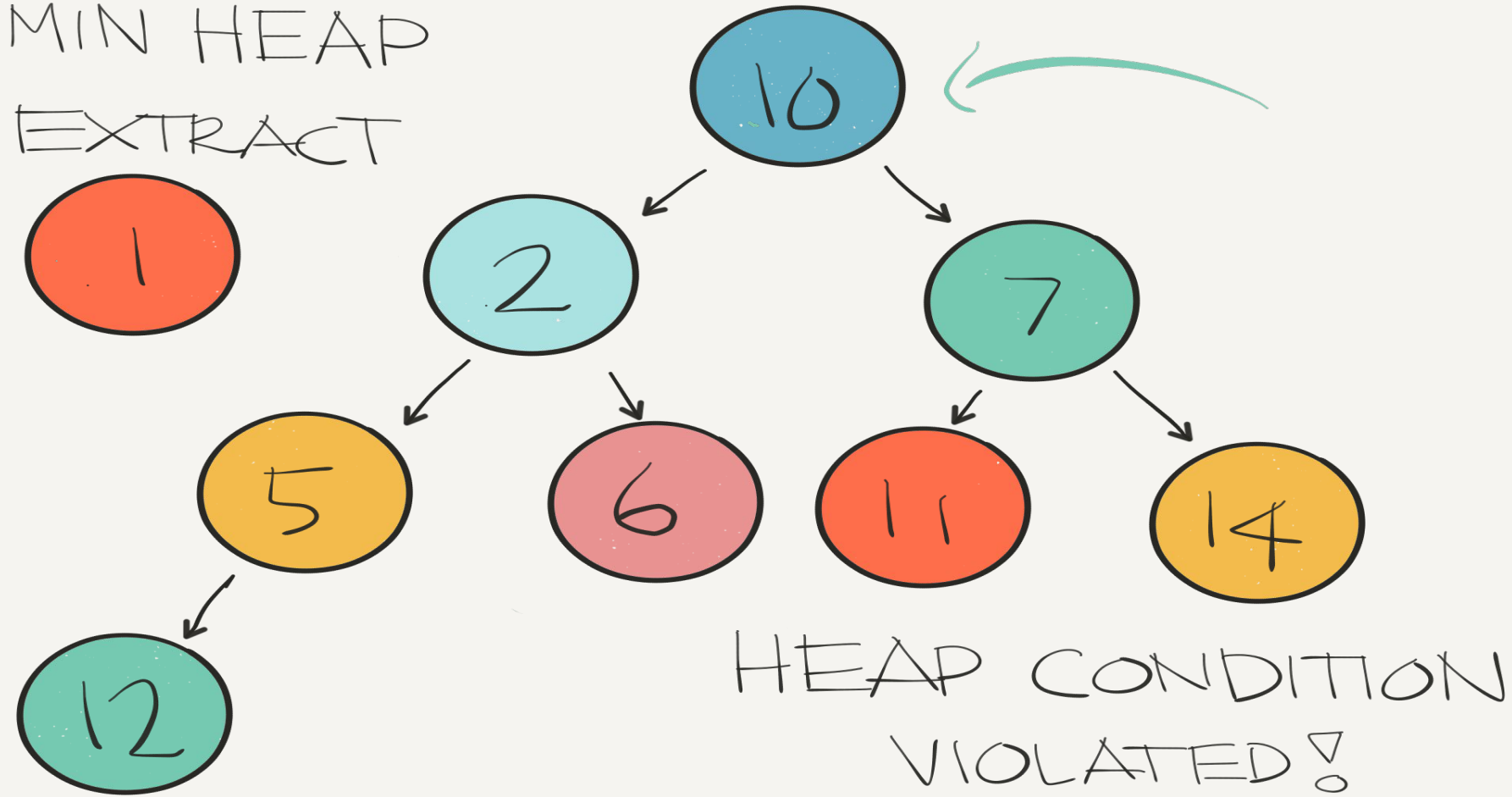


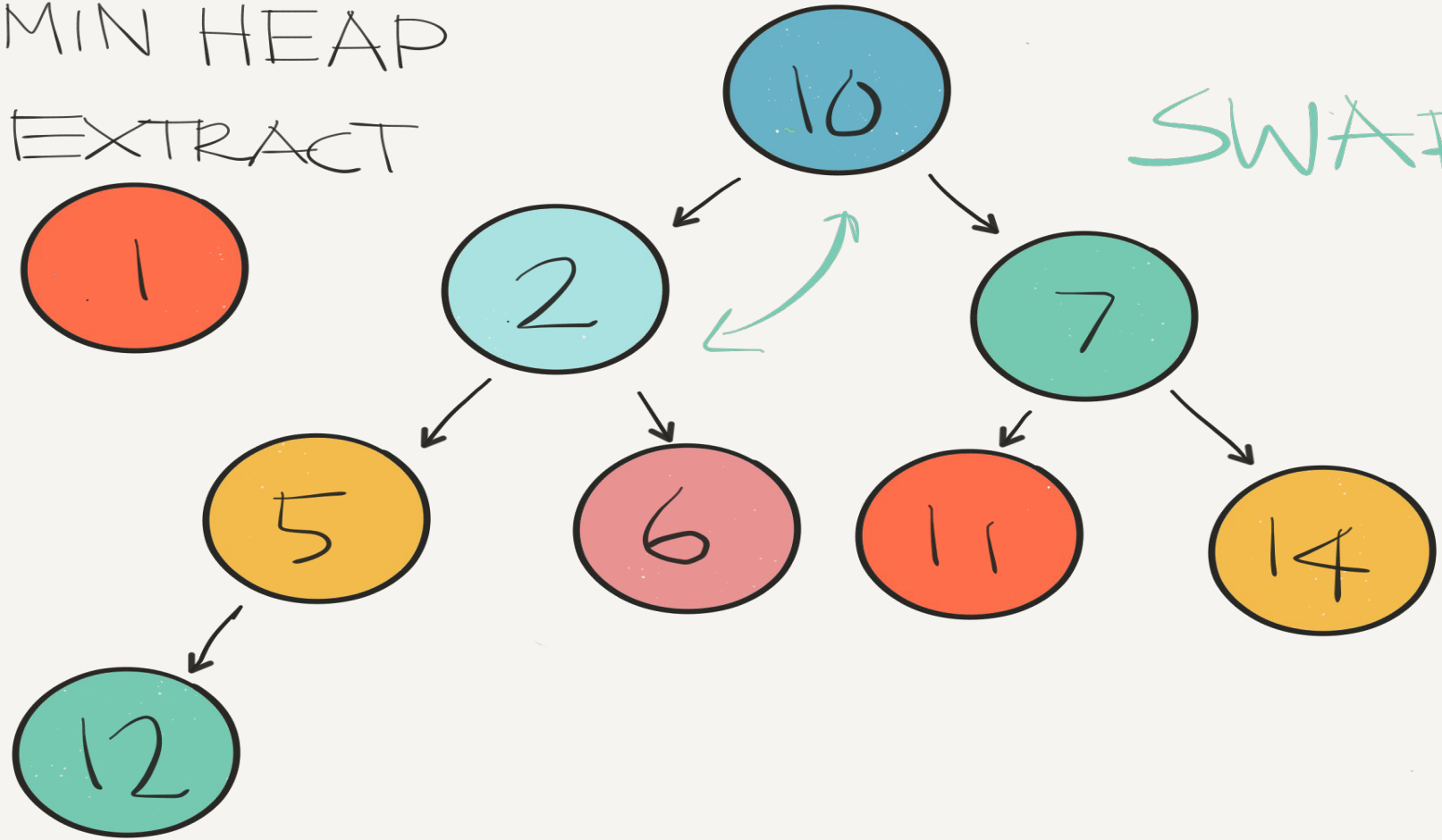
MUST EXTRACT MINIMUM

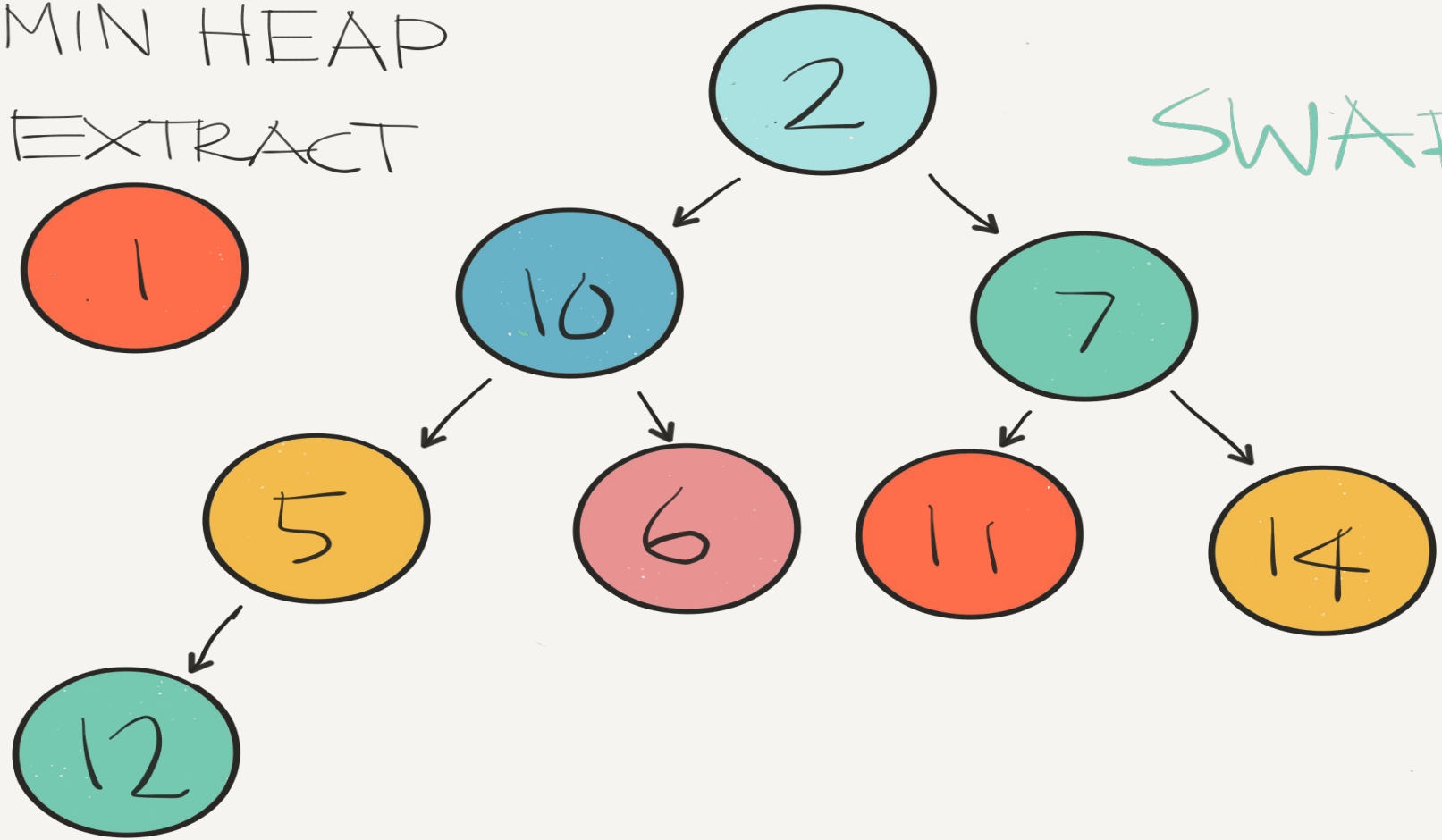
MIN HEAP EXTRACT

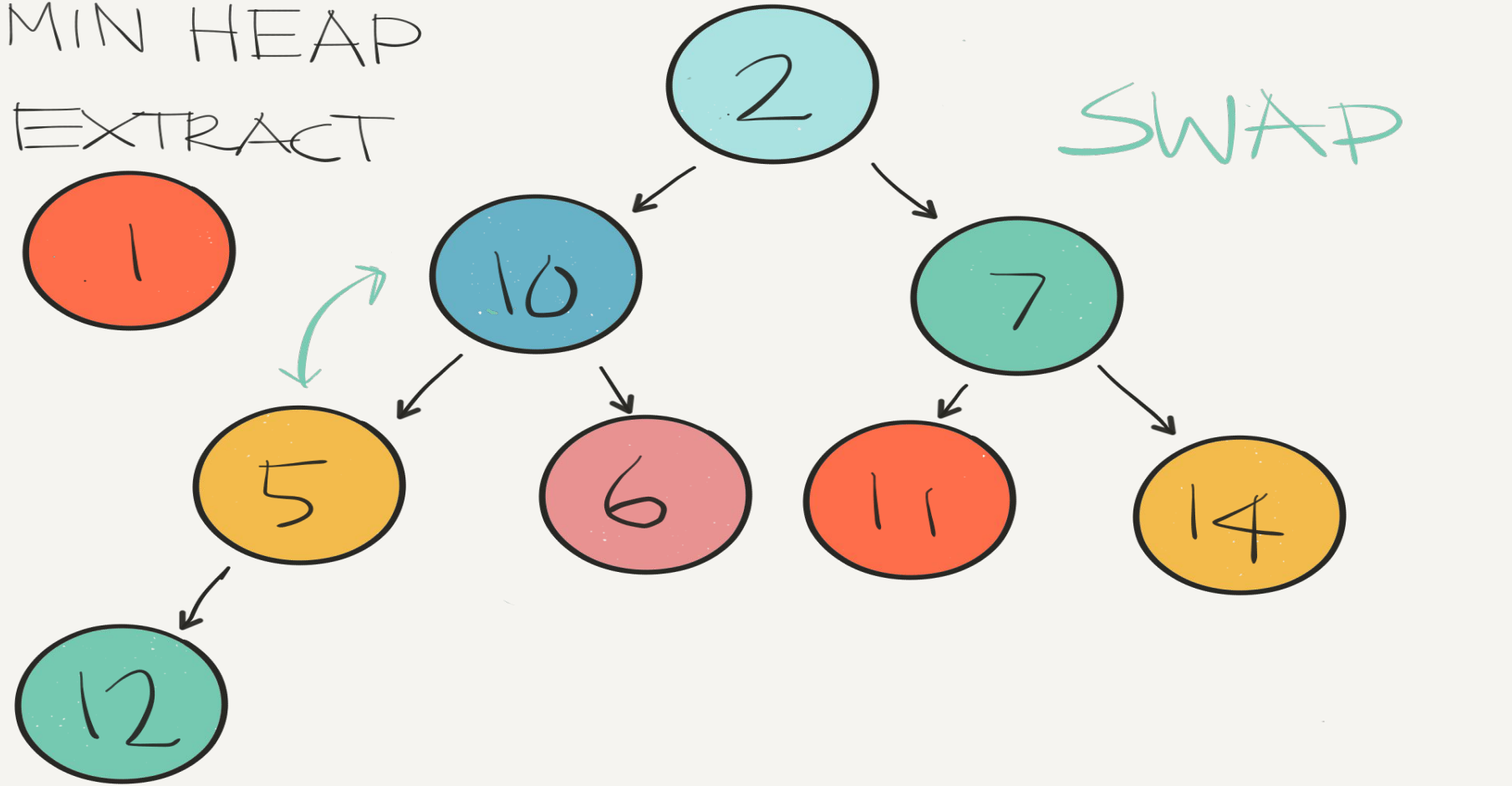


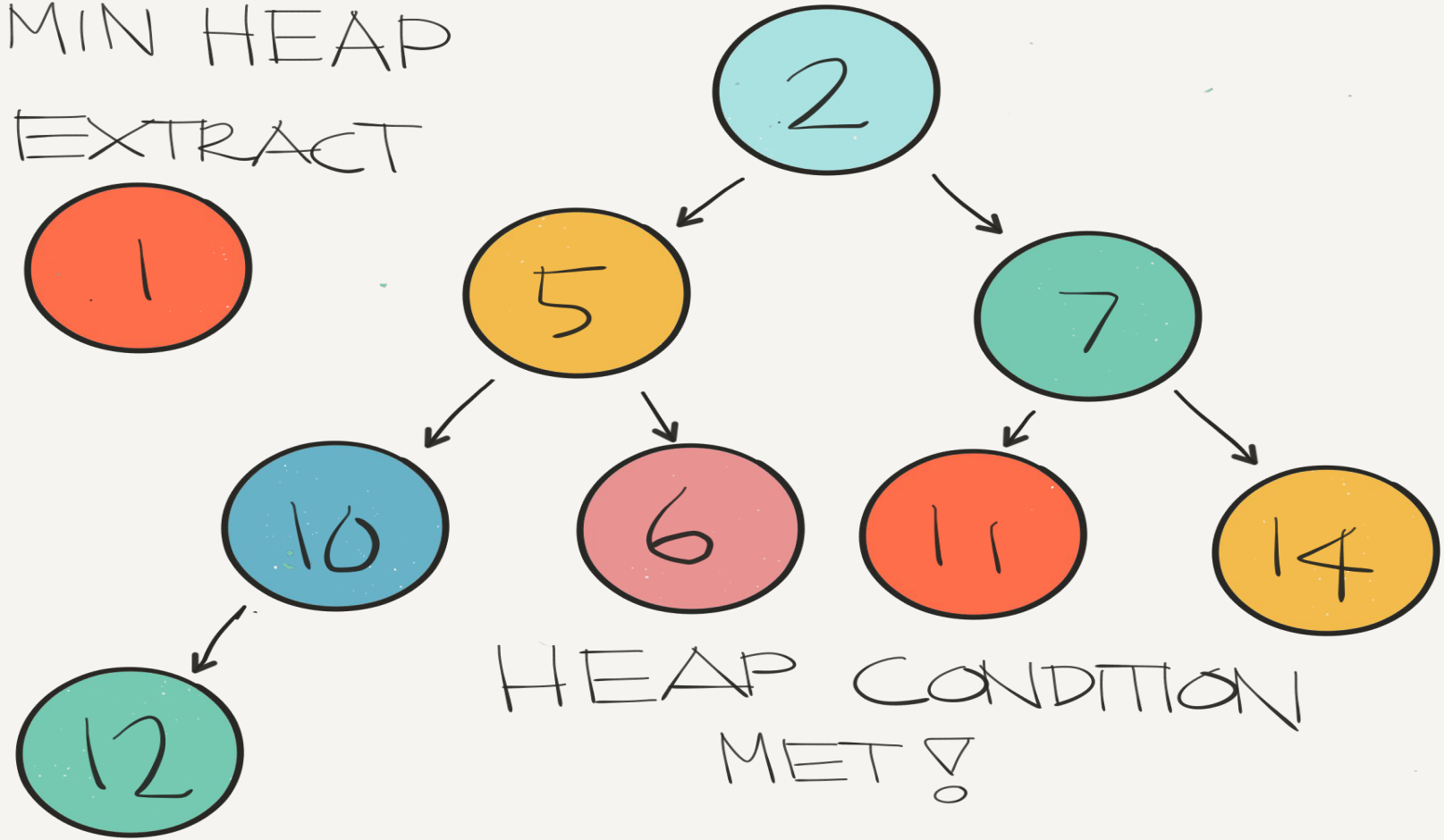

MIN HEAP
EXTRACT
10
1
2
7
5
6
11
14
12
SAVE TO BE RETURNED


MIN HEAP
EXTRACT
10
1
2
7
5
6
11
14
12
HEAP CONDITION
VIOLATED!


MIN HEAP
EXTRACT
10
SWAP
1
2
7
5
6
11
14
12

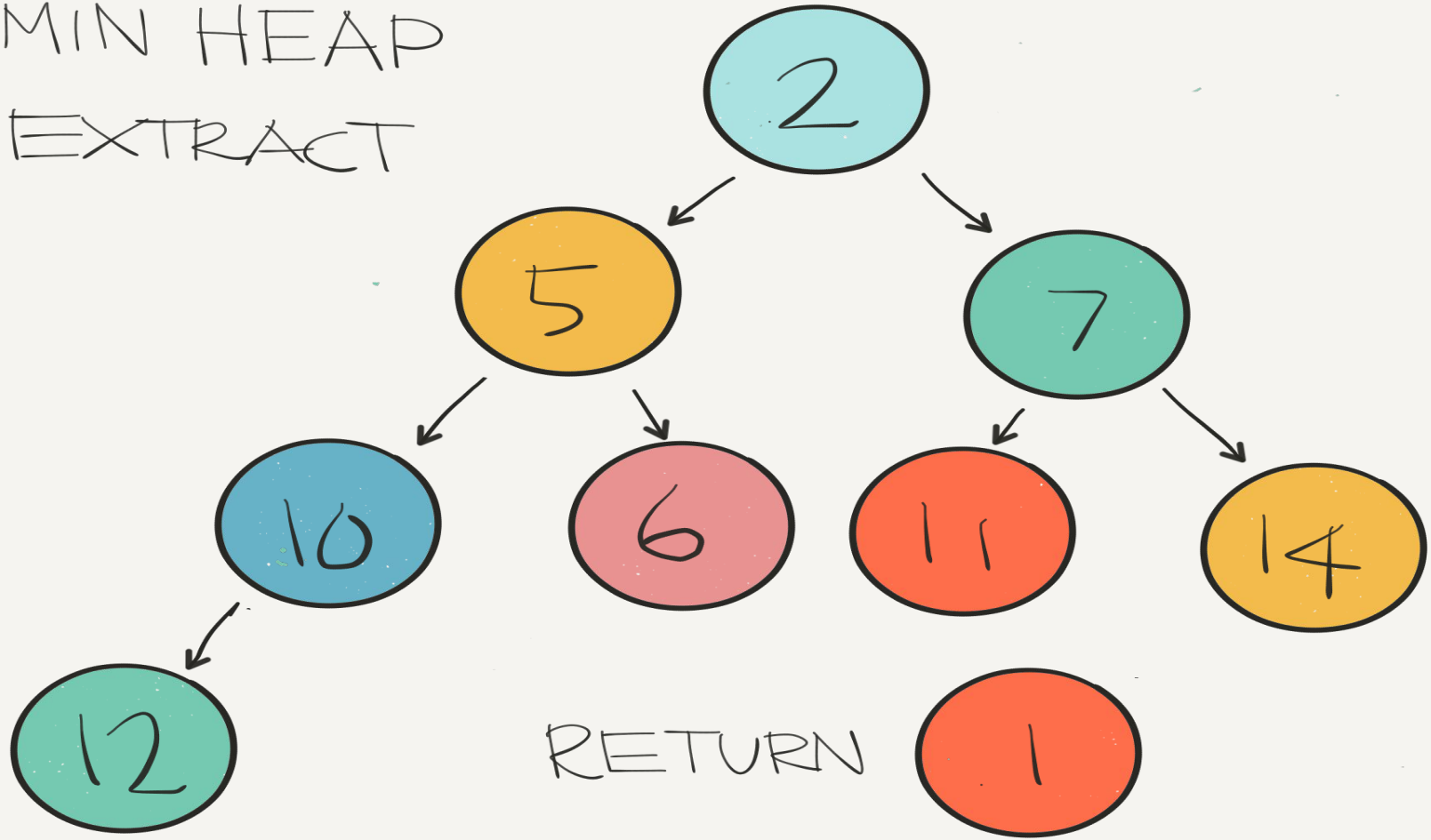

MIN HEAP
EXTRACT
SWAP
2
1
10
7
5
6
11
14
12


MIN HEAP
EXTRACT
SWAP
2
1
10
7
5
6
11
14
12


MIN HEAP
EXTRACT
1
2
5
7
10
6
11
14
12
HEAP CONDITION
MET!

MIN HEAP

EXTRACT



RETURN

Time Complexity

- Insert - O(logn)
- Extract - O(logn)
- Get max or min - O(1)

Example uses of binary heaps

- Priority queue system
- Implementing heapsort
- Optimized path finding
  - Djikstra’s method