



Never send a human  
to do a machine's job.

Performance,  
Algorithms,  
and  
Data Structures

# Performance

# Linear Search

- Brute force, linear search
- Start at the first element
- compare subsequent elements until the desired value if found, or there are no more.

# Linear Search

```
def search(ary, n)
  ary.each do |e|
    return true if e == n
  end
  false
end
```

# Linear Search

```
def search(ary, n)
  ary.each do |e|
    return true if e == n
  end
  false
end
```

'Fancy algorithms are slow when  
n is small, and n is usually small.'

-- Rob Pike

# $O()$ Notation

- We need a way to measure and compare the efficiency of algorithms
- Big O notation is a order of magnitude measure of the amount of work required in the worst case for a given algorithm (or class of algorithms)
- This measure is typically in terms of the size of the input data set.

$O(1)$

```
def a(ary)
  ary[0].nil?
end
```

- The size of `ary` has no impact of how much work this code does.
- So  $O(1)$

# $O(n)$

```
def search(ary, n)
  ary.each do |e|
    return true if e == n
  end
  false
end
```

- As the size of ary increases, the amount of work in the worst case (when n is not in ary) increases linearly.
- So  $O(n)$

# $O(n^2)$

```
def contains_duplicates(ary)
  (0..ary.count).each do |i|
    (0..ary.count).each do |j|
      if i != j
        return true if ary[i] == ary[j]
      end
    end
  end
  false
end
```

- For each element, ary is scanned looking for duplicates that match it.
- If there are no duplicates, ary is fully walked for each element, other than where  $(i == j)$ .
- Hence the comparison is done  $(n^2 - n)$  times, so  $O(n^2)$

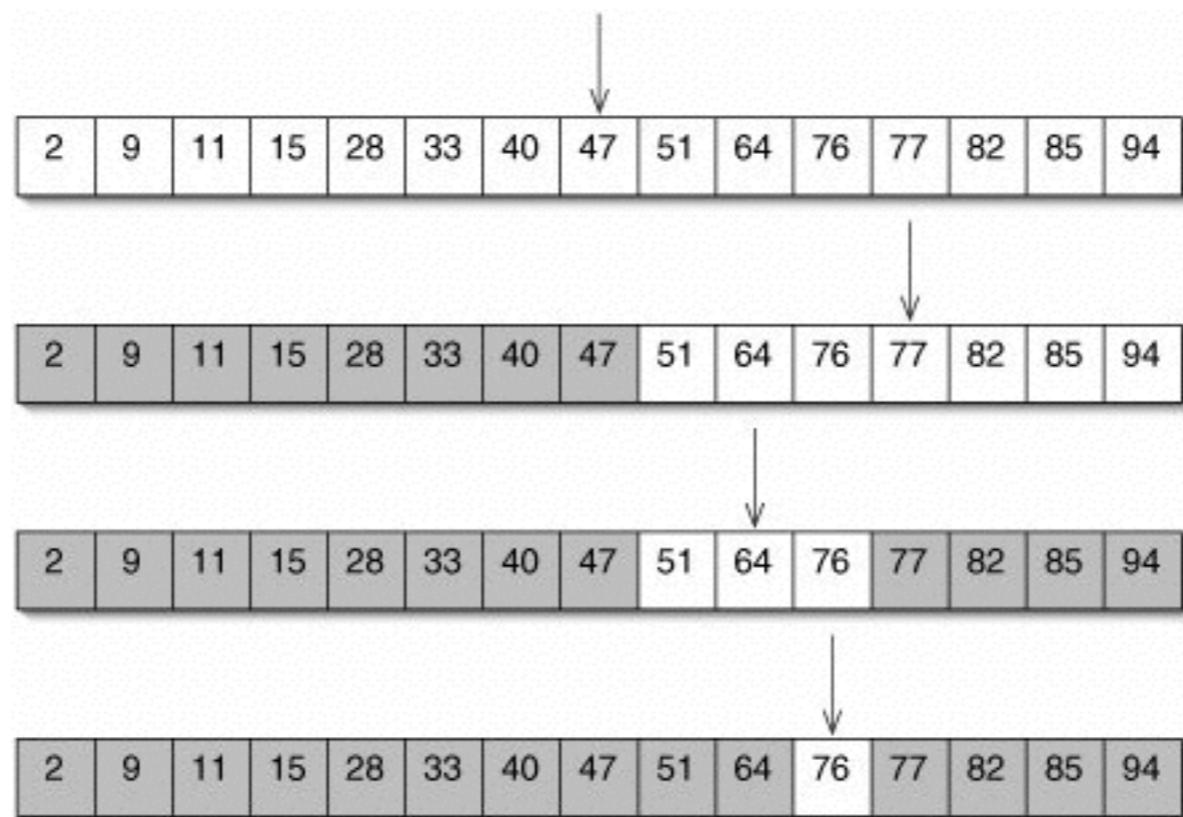
# Linear Search

- Our linear search algorithm has to compare the value being searched for with every element before knowing it isn't there.
- So it is  $O(n)$
- How can we do better?
- What about if we sort the array?
- How can we take advantage of that?

# Binary search

- If it's sorted, then we could look at the middle element.
- if it's what we want, we're done
- if what we're looking for is less, we can ignore the second half of the array
- if it's greater, we can ignore the first half.
- Do this repeatedly until found or you're down to a single element without being found

# Binary search



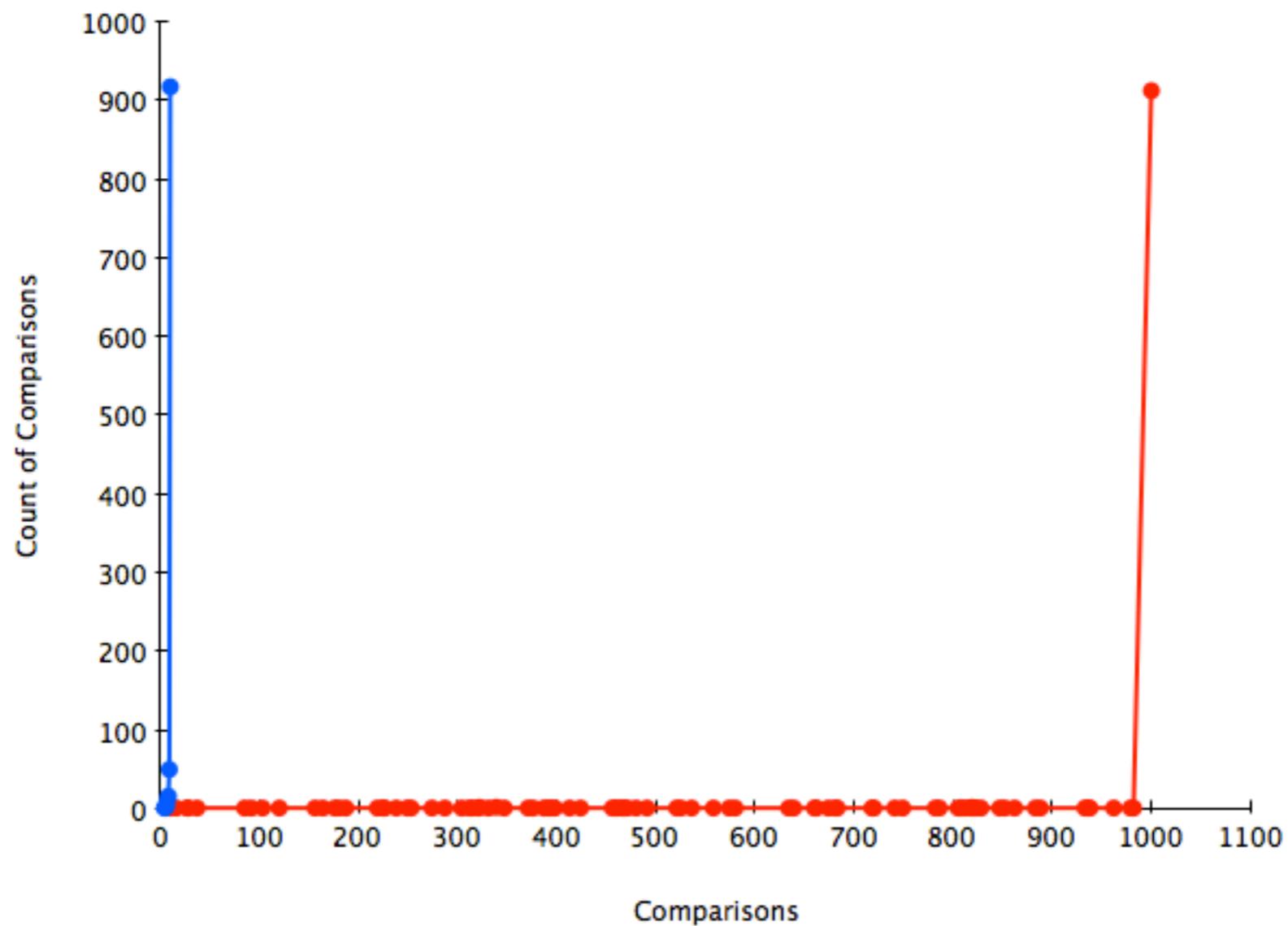
# Binary search code

```
def binary_search(ary, n)
    l = 0
    h = ary.count
    while l <= h
        m = ((l + h) / 2).floor
        if n < ary[m]
            h = m - 1
        elsif n > ary[m]
            l = m + 1
        else
            return true
        end
    end
    false
end
```

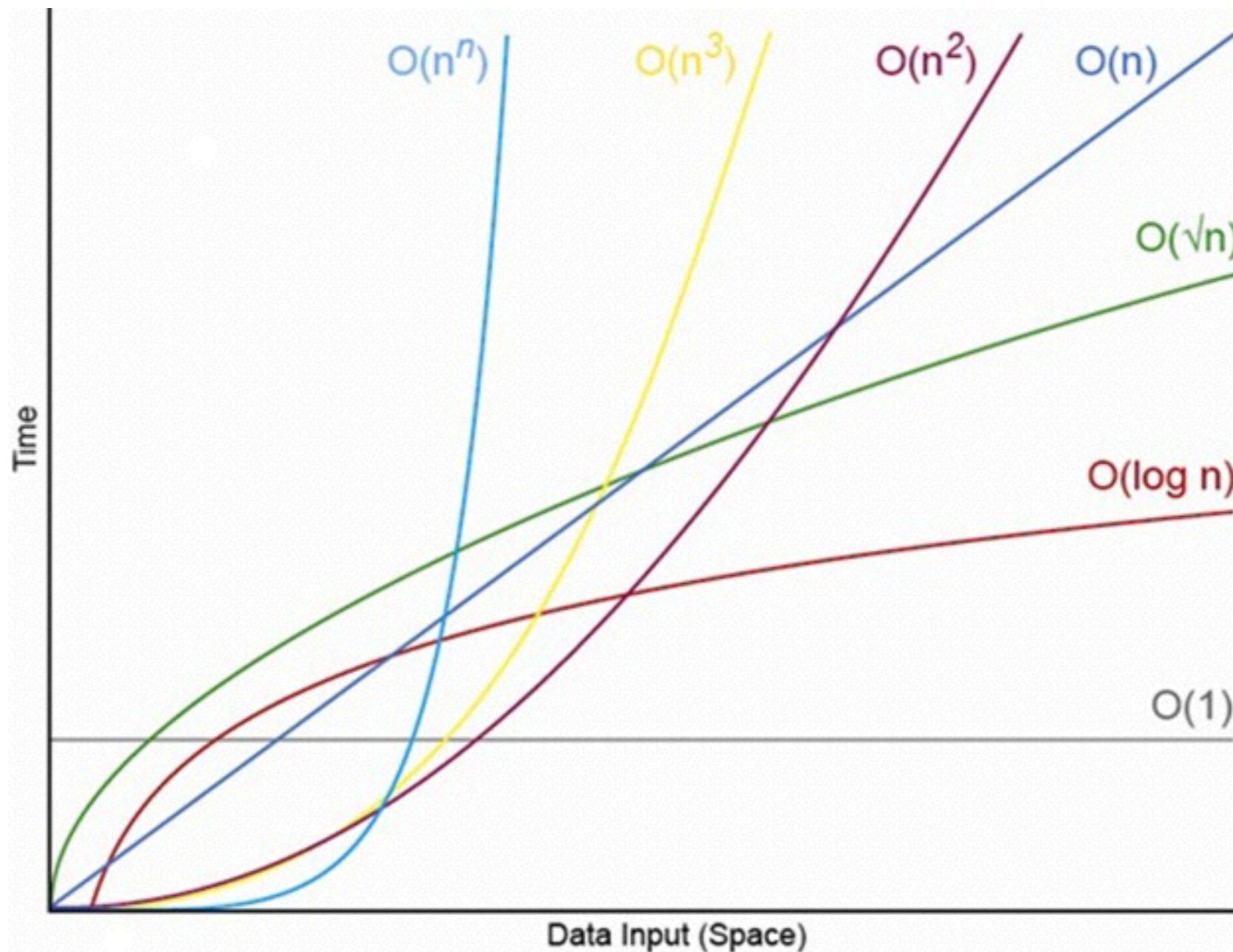
# $O(\log(n))$

- Actually,  $O(\log_2(n))$
- This is pretty ideal.
- Depends on treating the problem as a balanced binary search tree (which is easy with a sorted array)

# Comparison



# Big O comparison

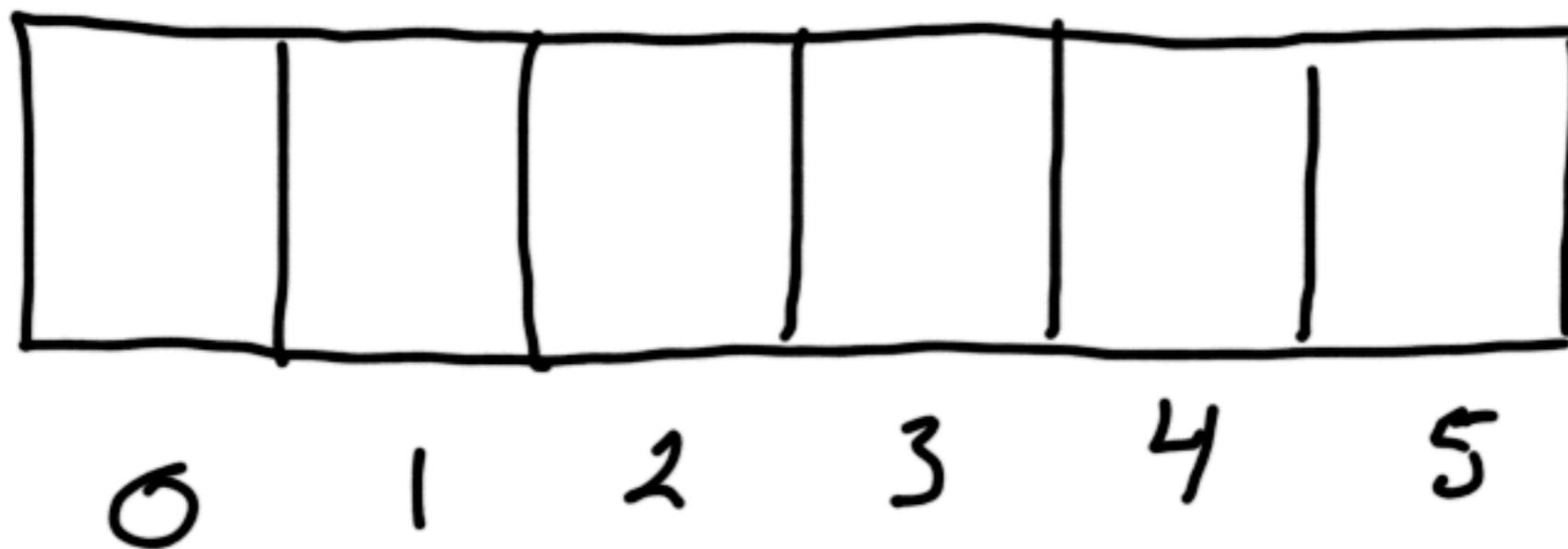


# Array

# Structure

- A collection of values, indexed by integer.
- Indices are typically contiguous, but sparse arrays are at times useful. Unless otherwise stated, you can assume indices starting at 0 (sometimes 1) and increase contiguously.
- Random access

# Structure



# Operations

- empty?
- size
- append
- access
- update
- insert
- remove

# Exercise

- Implement DbcArray
  - <https://github.com/dastelsdbc-deep-dives.git>
  - Algo directory
  - delete the contents of the lib dir, and put dbc\_array.rb there
  - You can build on top of Ruby's array
  - Use rspec spec/array\_spec.rb to guide you

# Applications

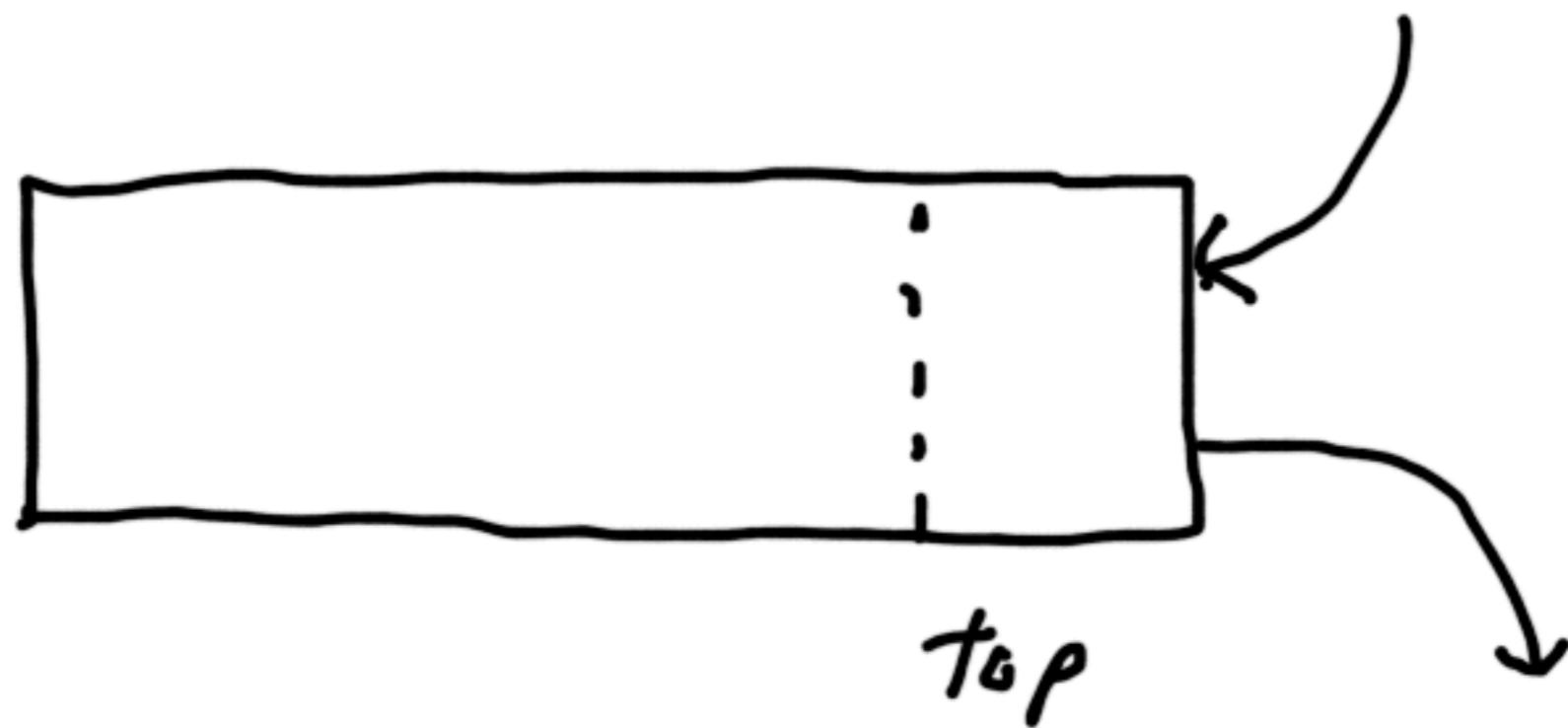
- When you need a collection of things and need random access

# Stack

# Structure

- A linear collection of values, which can be added and removed.
- Access only by peeking or popping
- Access is last-in, first-out

# Structure



# Operations

- empty?
- push
- pop
- peek
- full? Might be useful if we limit the size of the stack
- do we need to know the size?

# Exercise

- Implement DbcStack
  - <https://github.com/dastelsdbc-deep-dives.git>
  - Algo directory
  - delete the contents of the lib dir, and put dbc\_stack.rb there
  - You can build on top of Ruby's array
  - Use rspec spec/stack\_spec.rb to guide you

# Applications

- When you need to keep track of things for later, and revisit them as you backtrack.
- Example: check for balanced parens & brackets:
  - the empty string is balanced
  - if S is balanced, then (S) is
  - if S is balanced then [S] is
  - if S and T are balanced, then ST is

# Queue

# Structure

- A linear collection of values, which can be added and removed.
- Add to one end (rear), remove from the other (front)
- Access only by dequeuing
- Access is first-in, first-out

# Structure



# Operations

- empty?
- enqueue
- dequeue
- full? Might be useful if we limit the size of the queue
- do we need to know the size?

# Exercise

- Implement DbcQueue
  - <https://github.com/dastelsdbc-deep-dives.git>
  - Algo directory
  - delete the contents of the lib dir, and put dbc\_queue.rb there
  - You can build on top of Ruby's array
  - Use rspec spec/queue\_spec.rb to guide you

# Applications

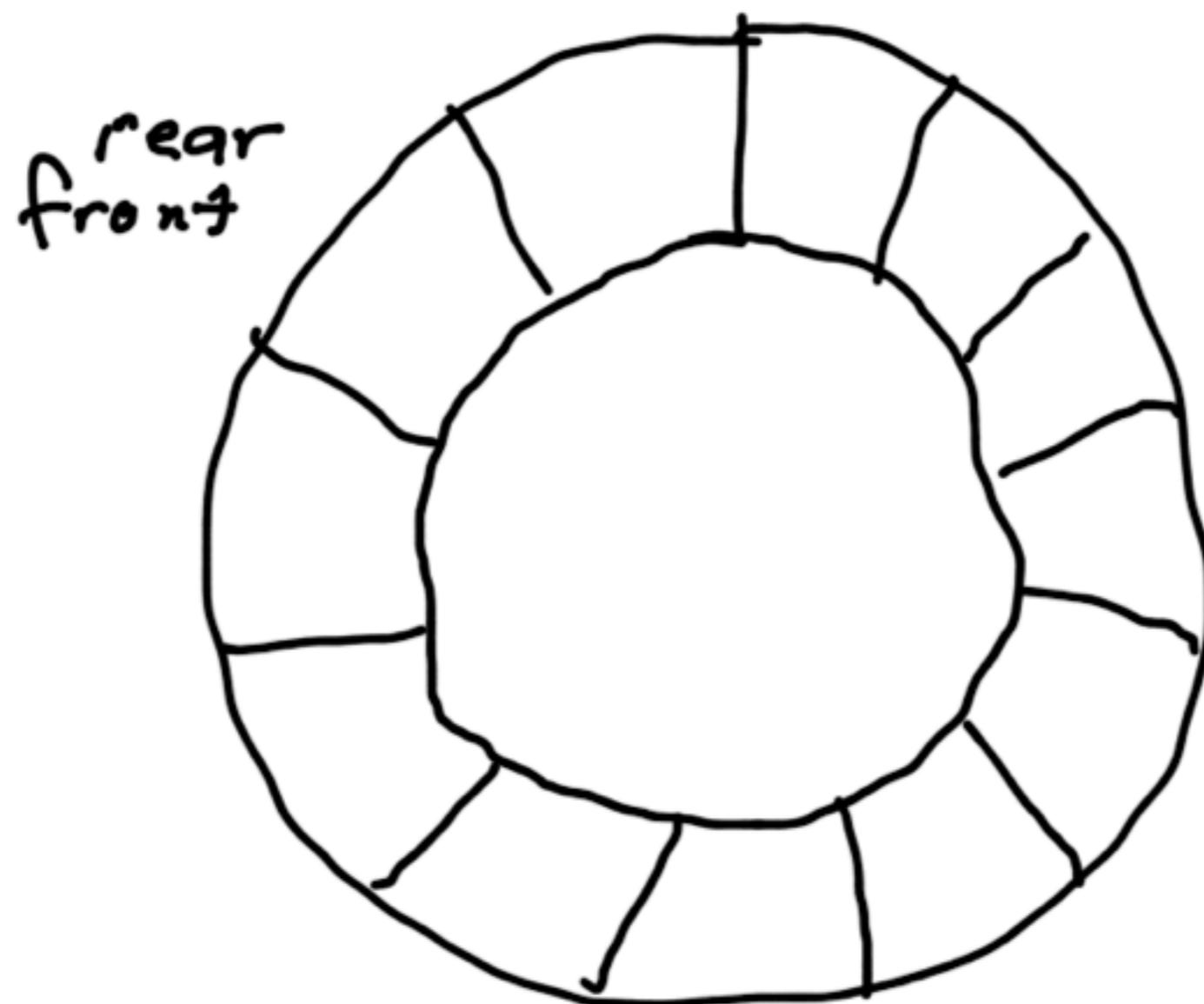
- Job queues

# Circular Queue

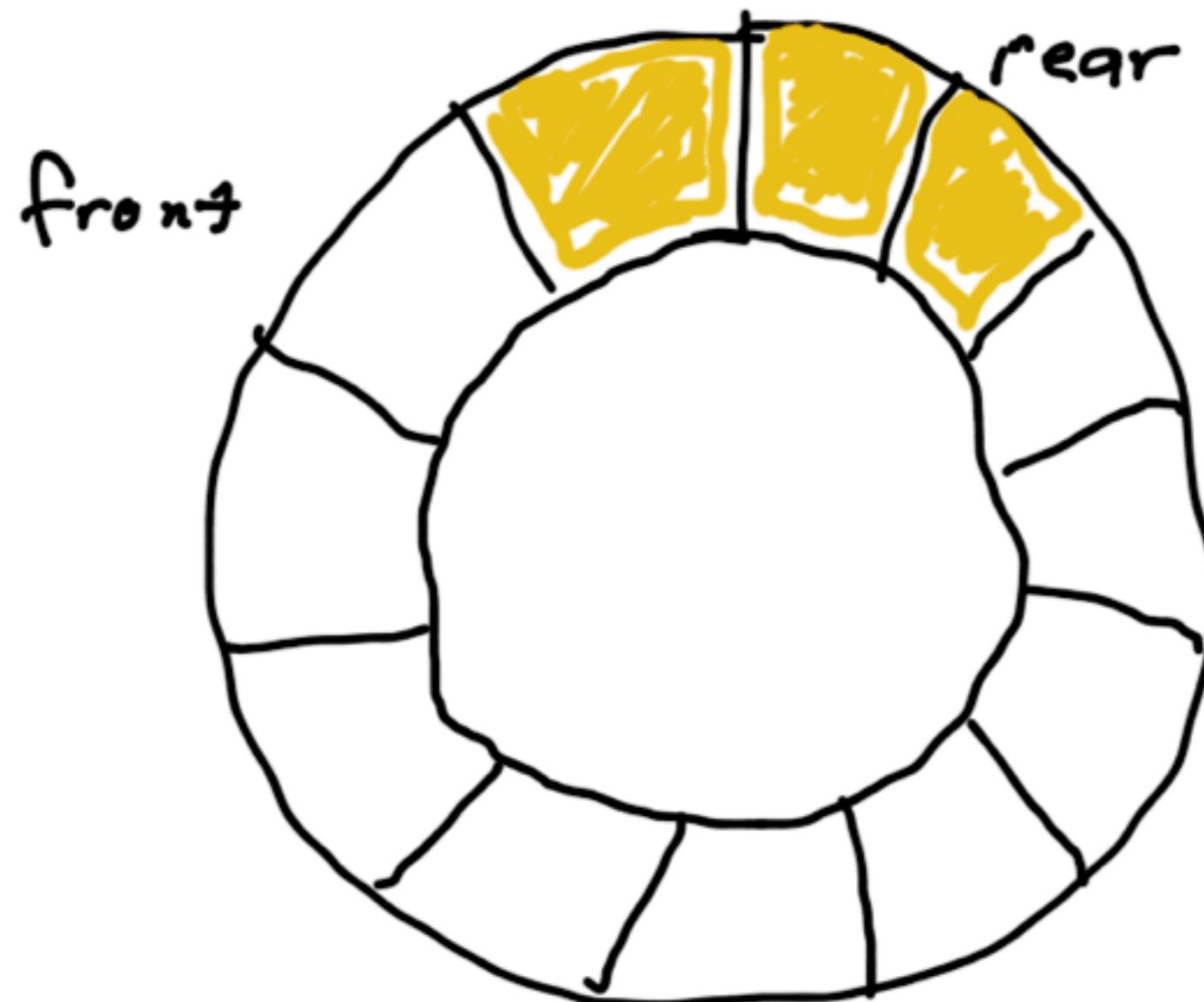
# Structure

- Same as a queue, except:
- Size is limited whereas a plain queue can grow arbitrarily
- Typically implemented with a ring buffer built on an array

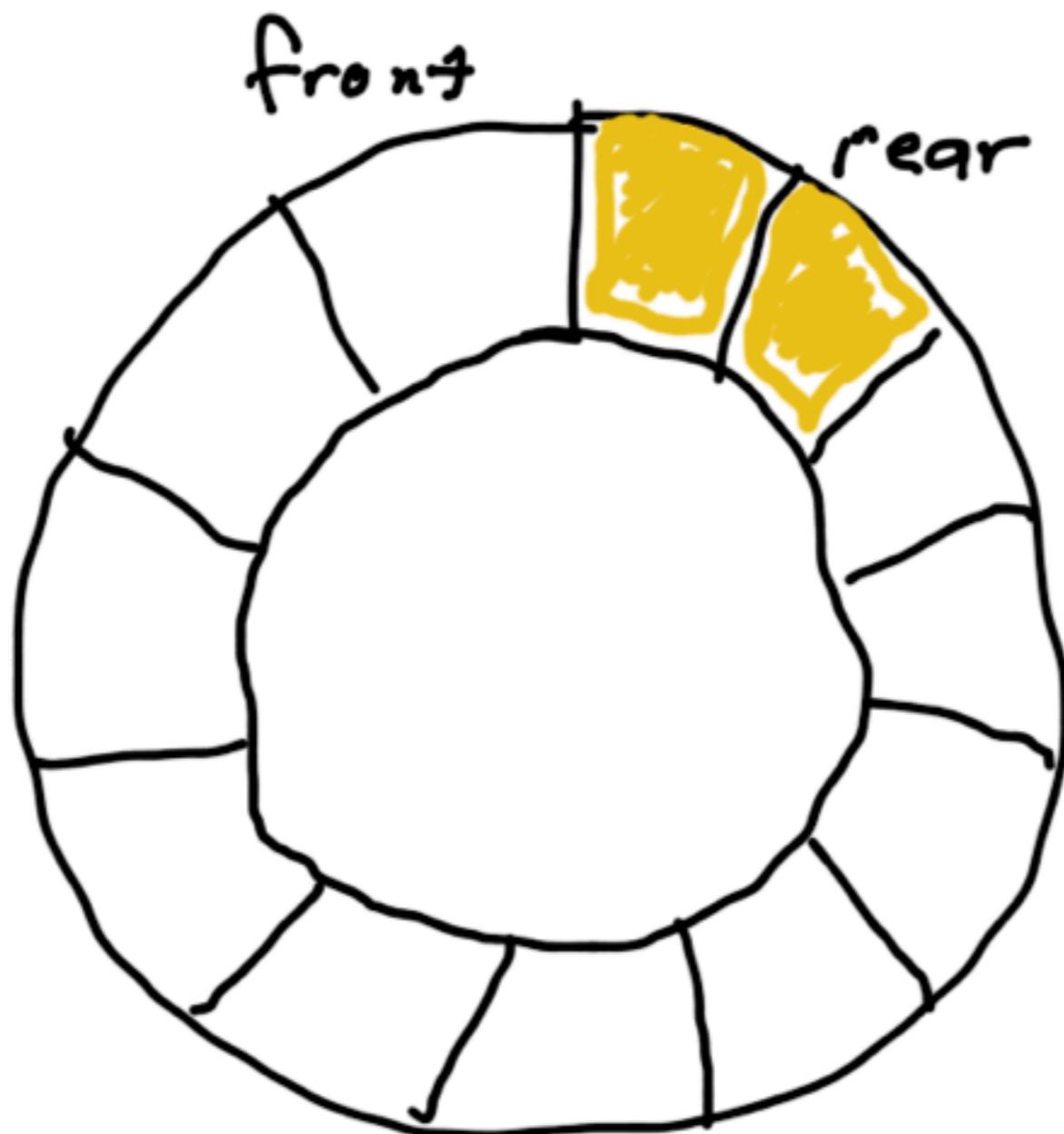
# Empty



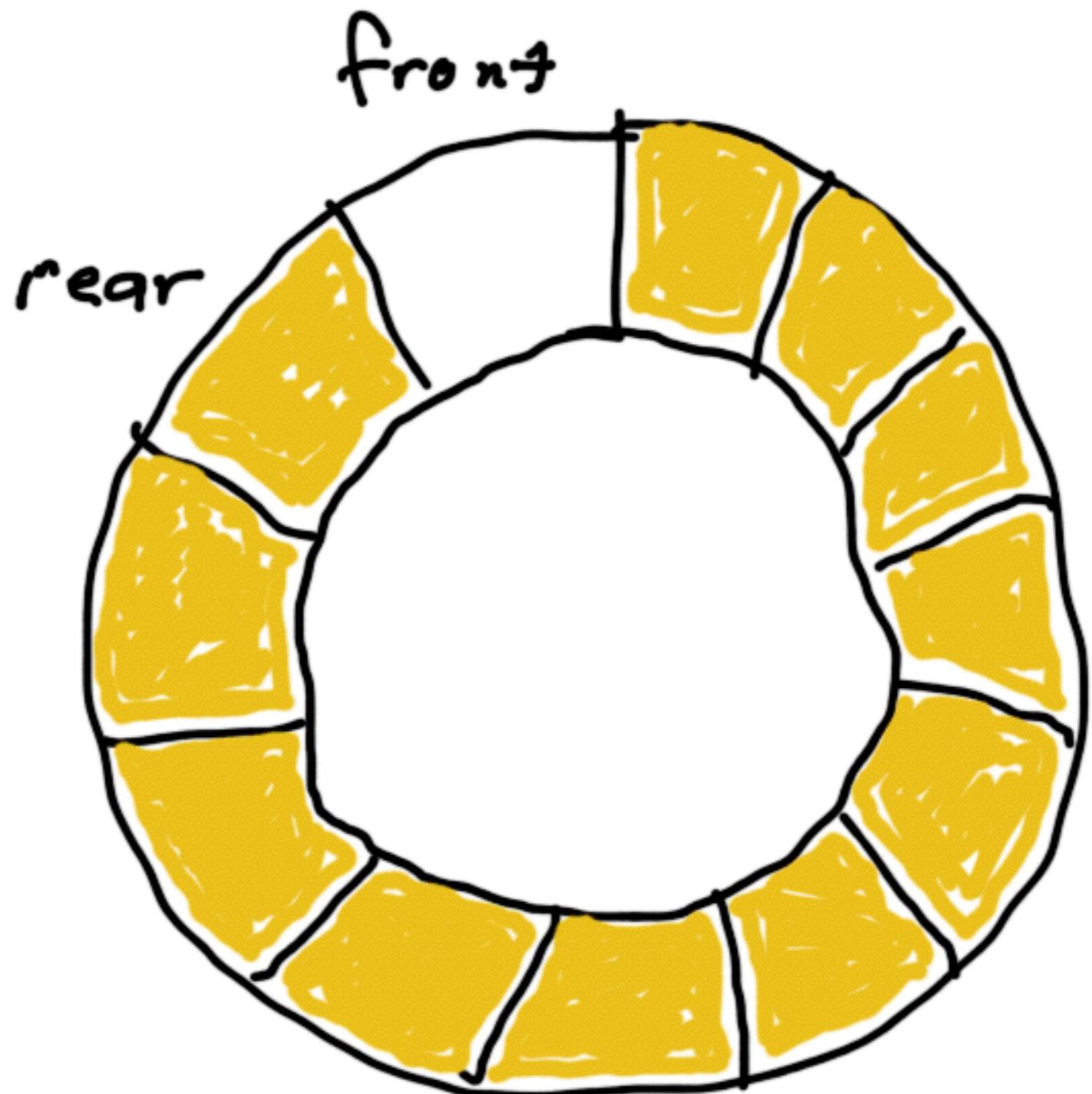
# Add a few



# Remove one



Full



# Operations

- empty?
- full?
- enqueue
- dequeue
- do we need to know the size?

# Exercise

- Implement DbcCircularQueue
  - <https://github.com/dastels/dbc-deep-dives.git>
  - Algo directory
  - delete the contents of the lib dir, and put dbc\_circular\_queue.rb there
  - You can build on top of Ruby's array
  - Use rspec spec/circular\_queue\_spec.rb to guide you

# Applications

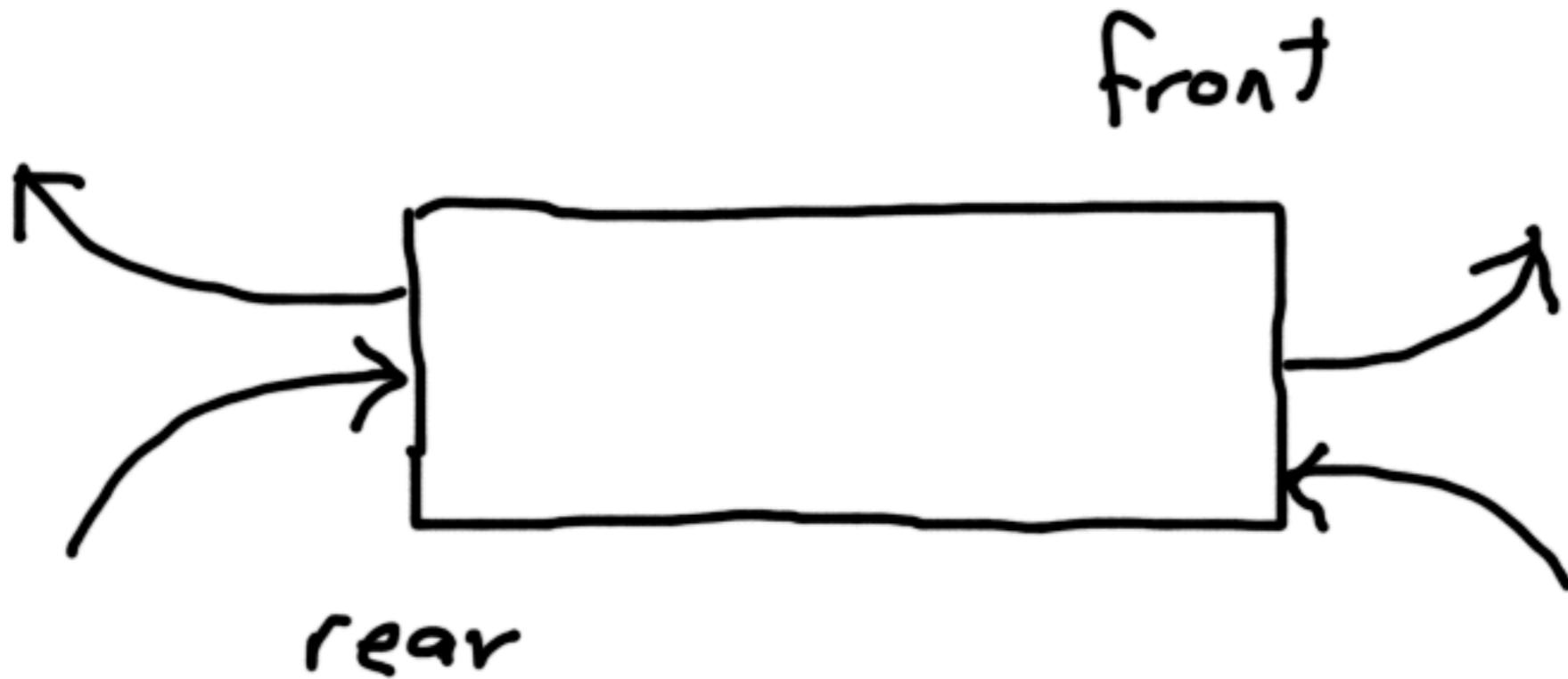
- Anywhere you would use a queue, but want to constrain it's size, with fixed memory allocation.

# Double Ended Queue

# Structure

- Like a queue, but you enqueue and dequeue on both ends.

# Structure



# Operations

- empty?
- full?
- enqueue\_front
- enqueue\_rear
- dequeue\_front
- dequeue\_rear
- do we need to know the size?

# Exercise

- Implement DbcDoubleEndedQueue
  - <https://github.com/dastels/dbc-deep-dives.git>
  - Algo directory
  - delete the contents of the lib dir, and put dbc\_double-ended\_queue.rb there
  - You can build on top of Ruby's array
  - Use rspec spec/double-ended\_queue\_spec.rb to guide you

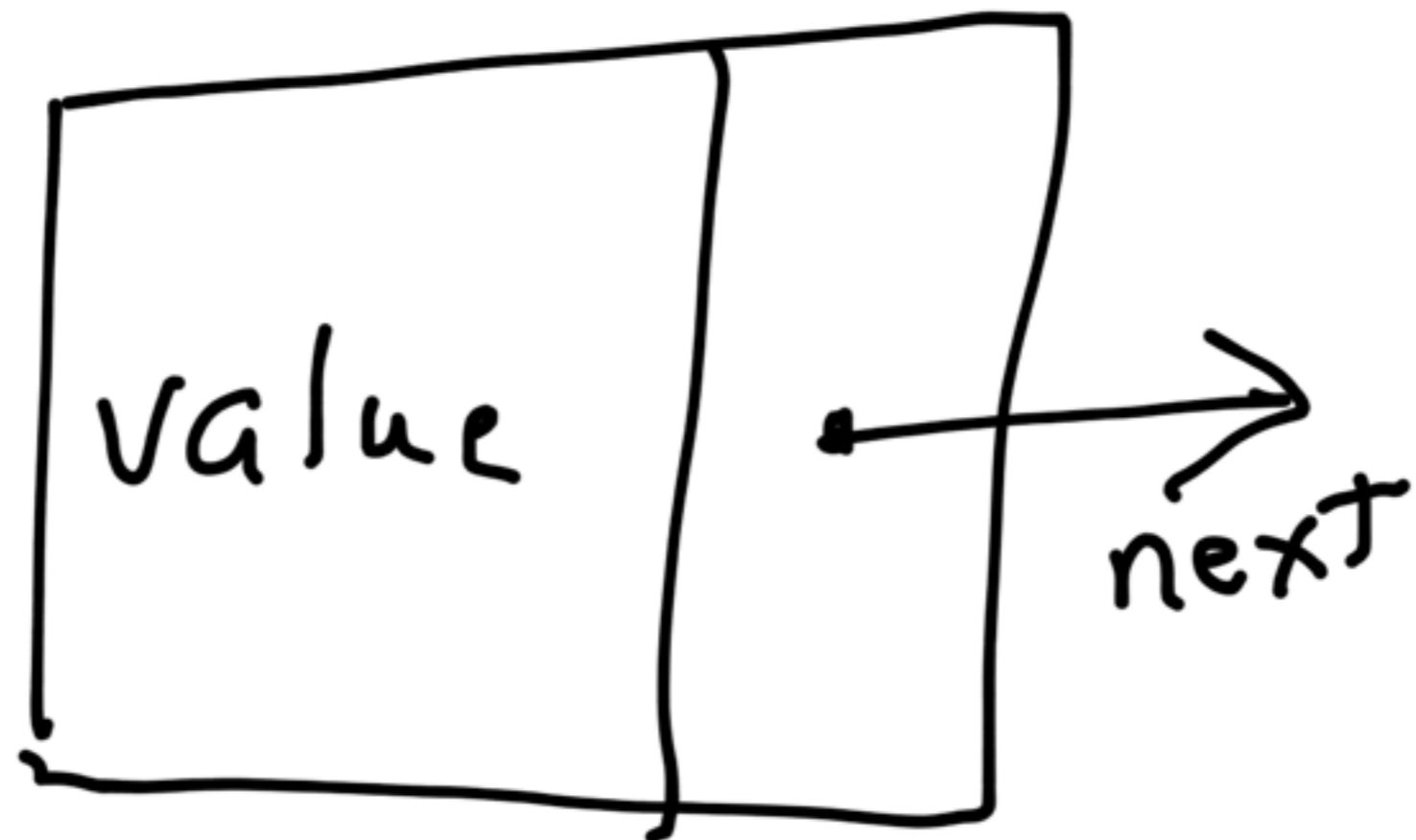
# Applications

# Singly Linked List

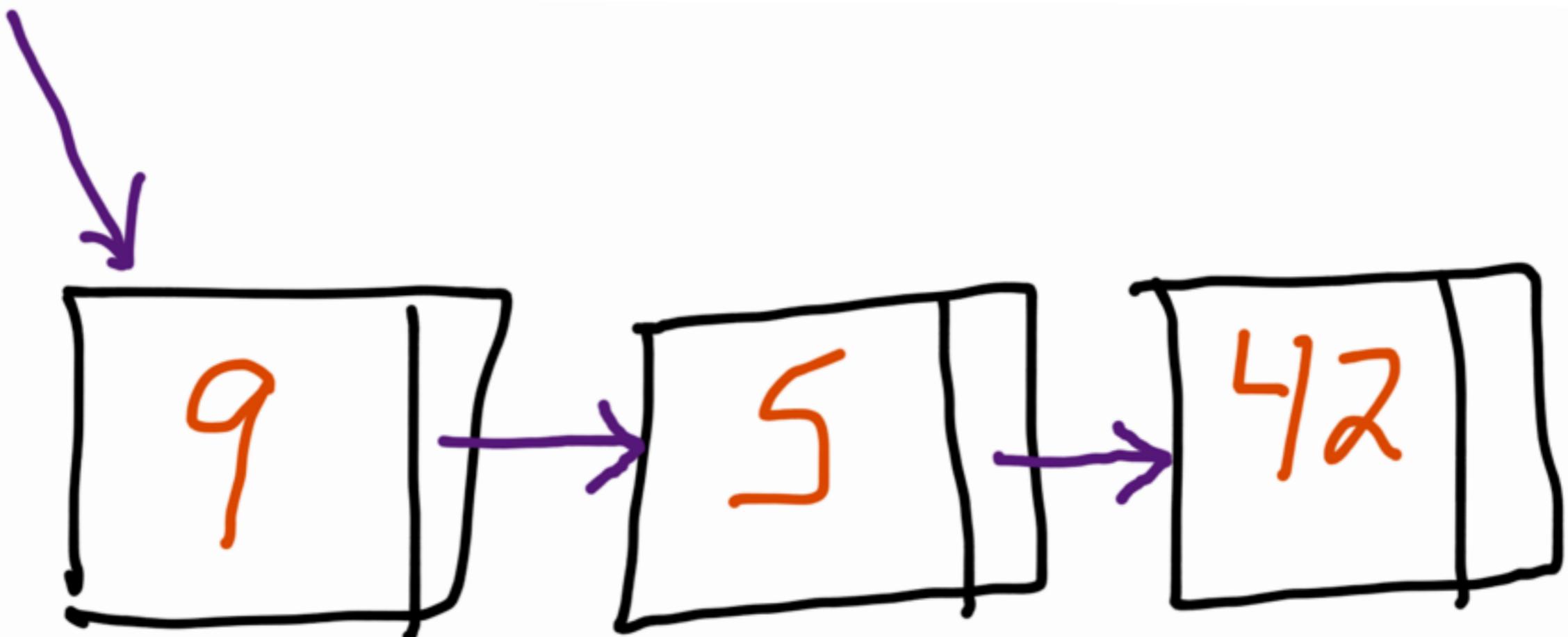
# Structure

- A list of nodes, each with a value and a reference to the next node in the list.

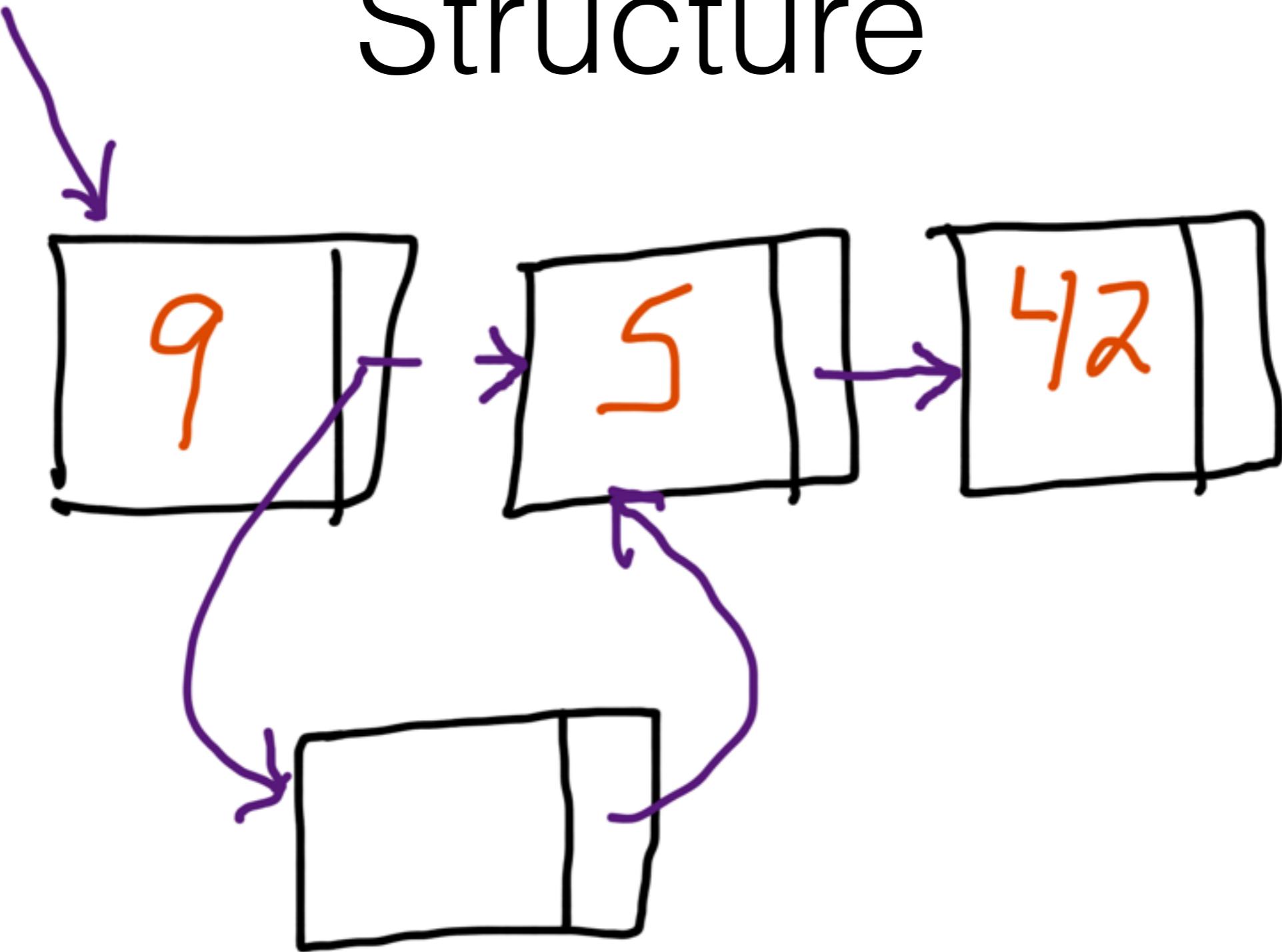
# Structure



# Structure



# Structure



# Operations

- empty?
- size
- insert
- delete
- append
- prepend
- find
- map
- reduce

# Exercise

- Implement DbcLinkedList
  - <https://github.com/dastels/dbc-deep-dives.git>
  - DataStructures directory
  - delete the contents of the lib dir, and put dbc\_linked\_list.rb there
  - Use rspec spec/linked\_list\_spec.rb to guide you

# Exercise

- Try to make recursive solutions.
- Let's walk through implementing size together.

# Applications

# Doubly Linked List

Tree

# Structure

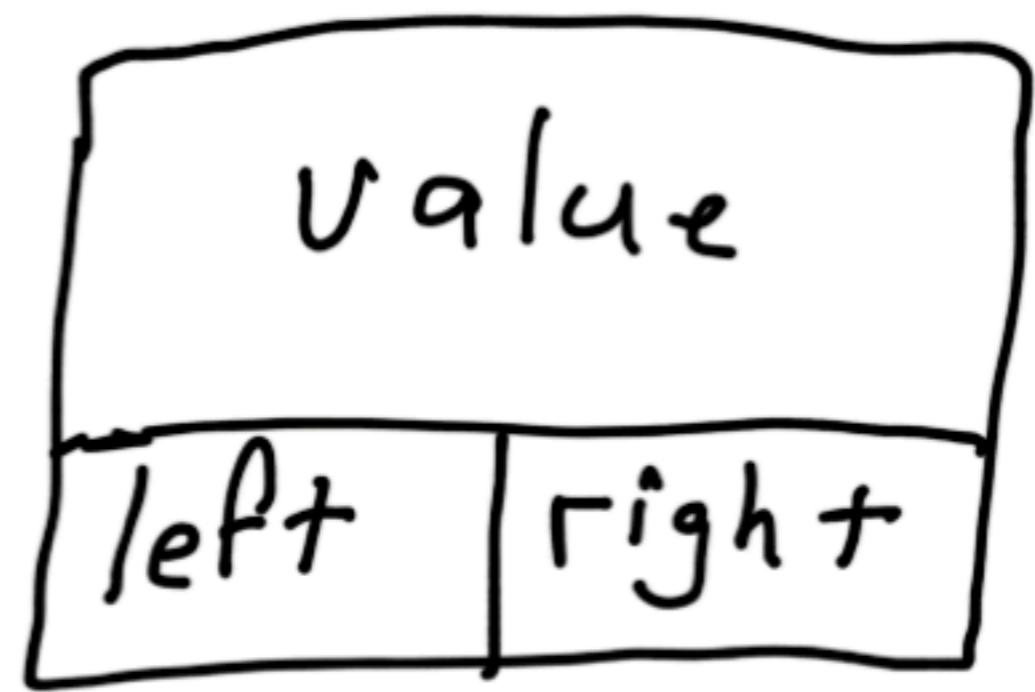
- An arrangement of nodes, each of which can reference some number of other nodes, and can only be referenced by a single node.
- A linked list is like a degenerate tree where each node refers to at most one other node.

# Binary Search Tree

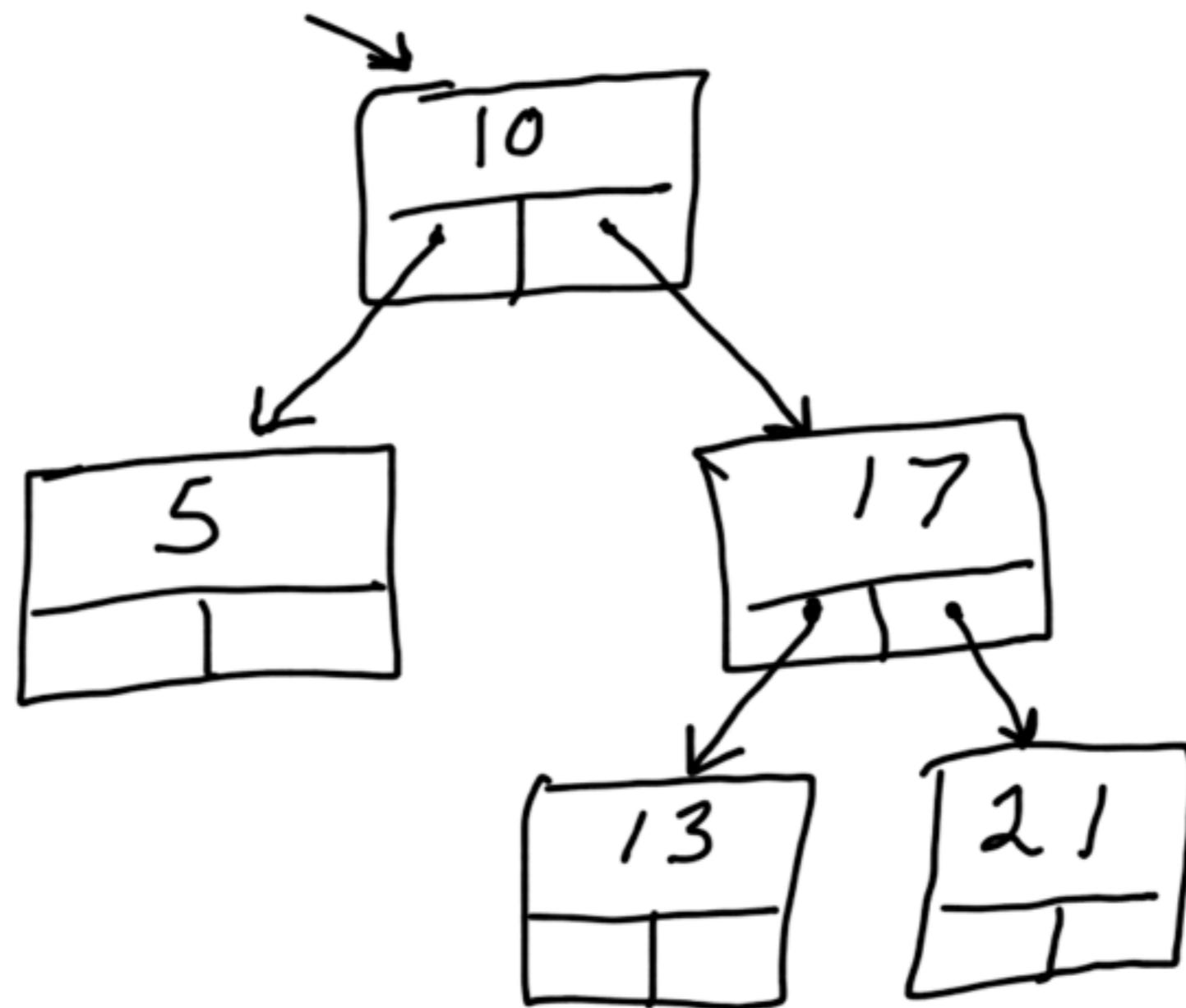
# Structure

- An arrangement of nodes, each of which can reference at most **two** other nodes, and can only be referenced by a single node.
- Acyclic
- Directed

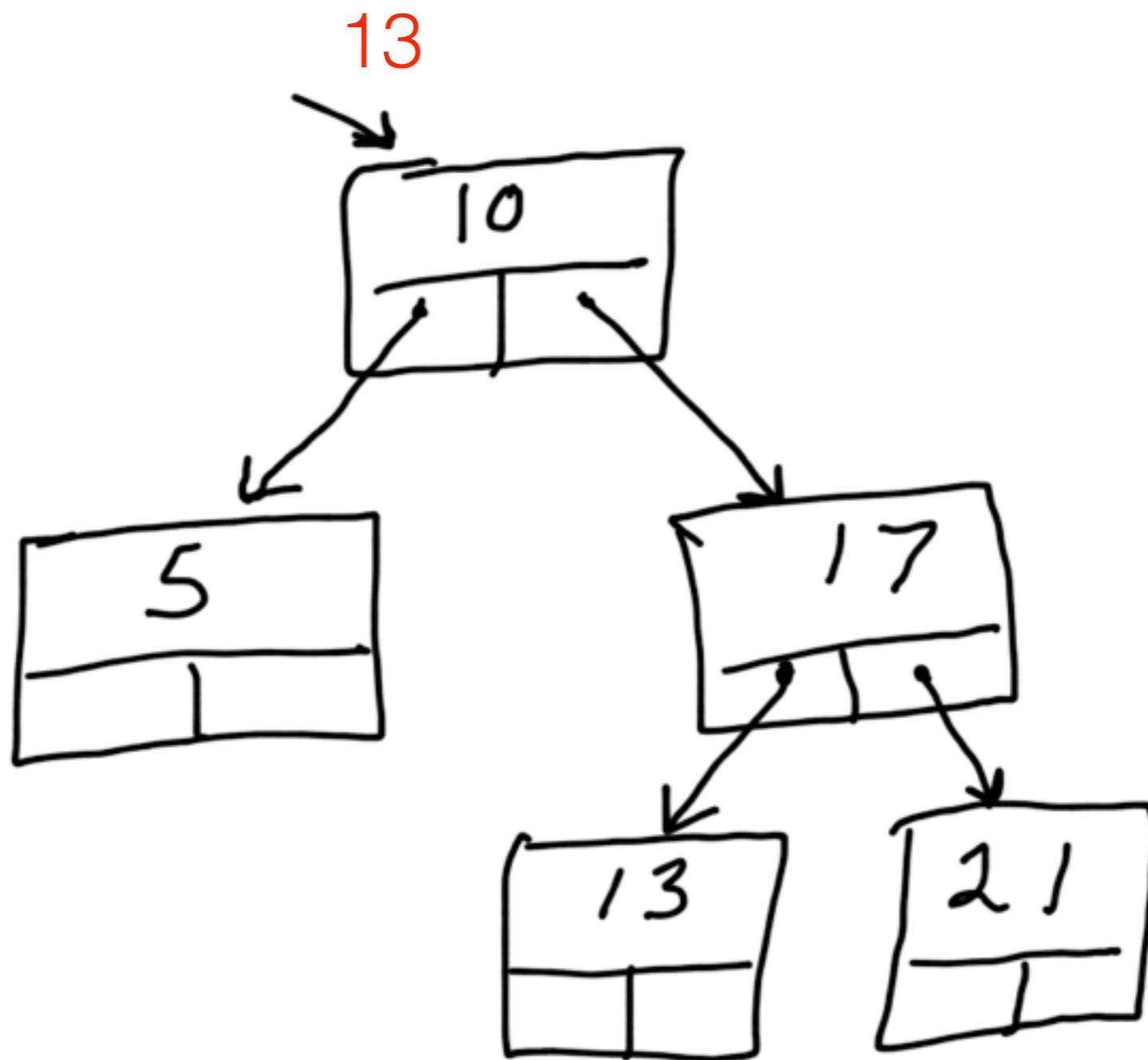
# Structure



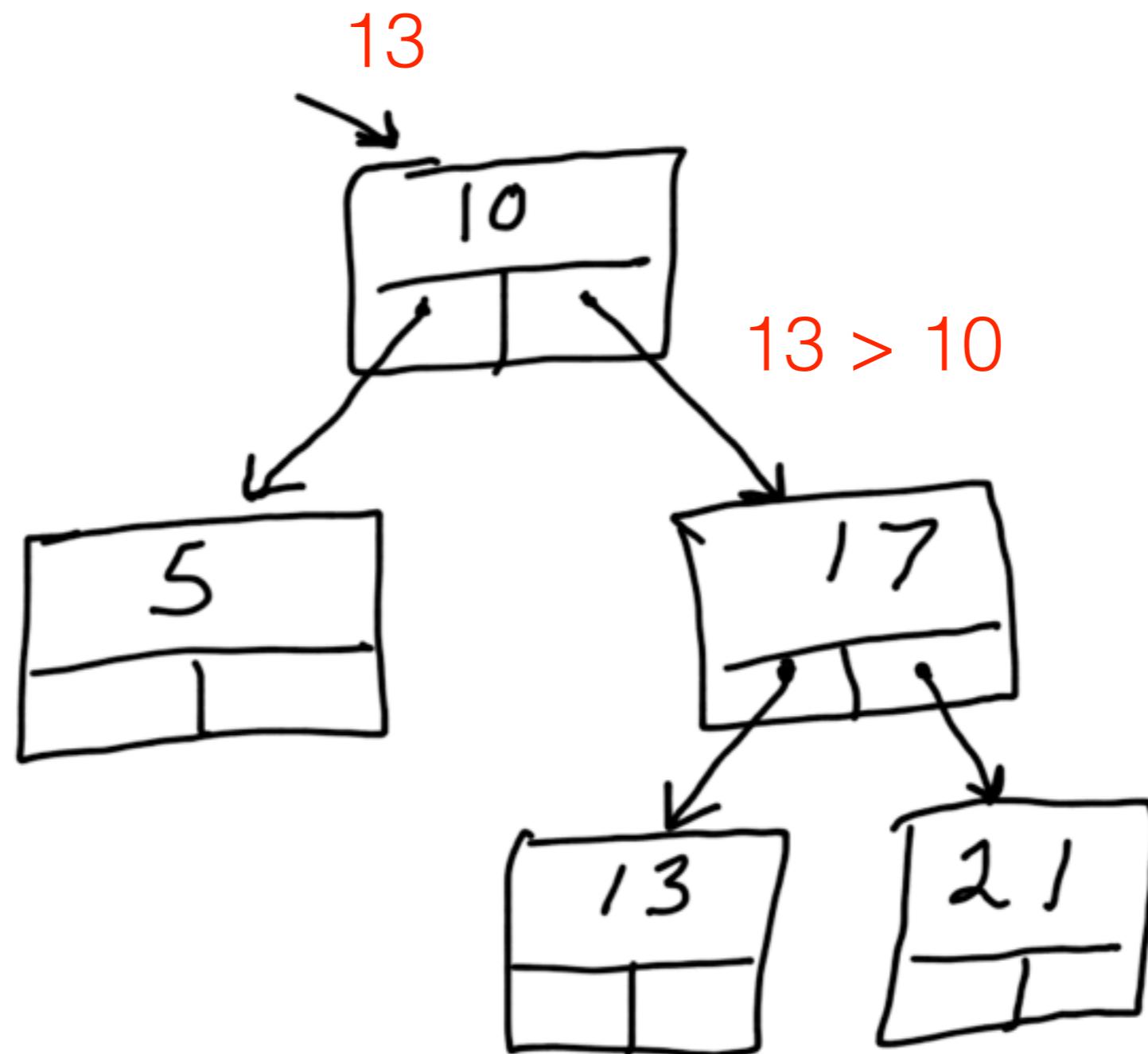
# Searching



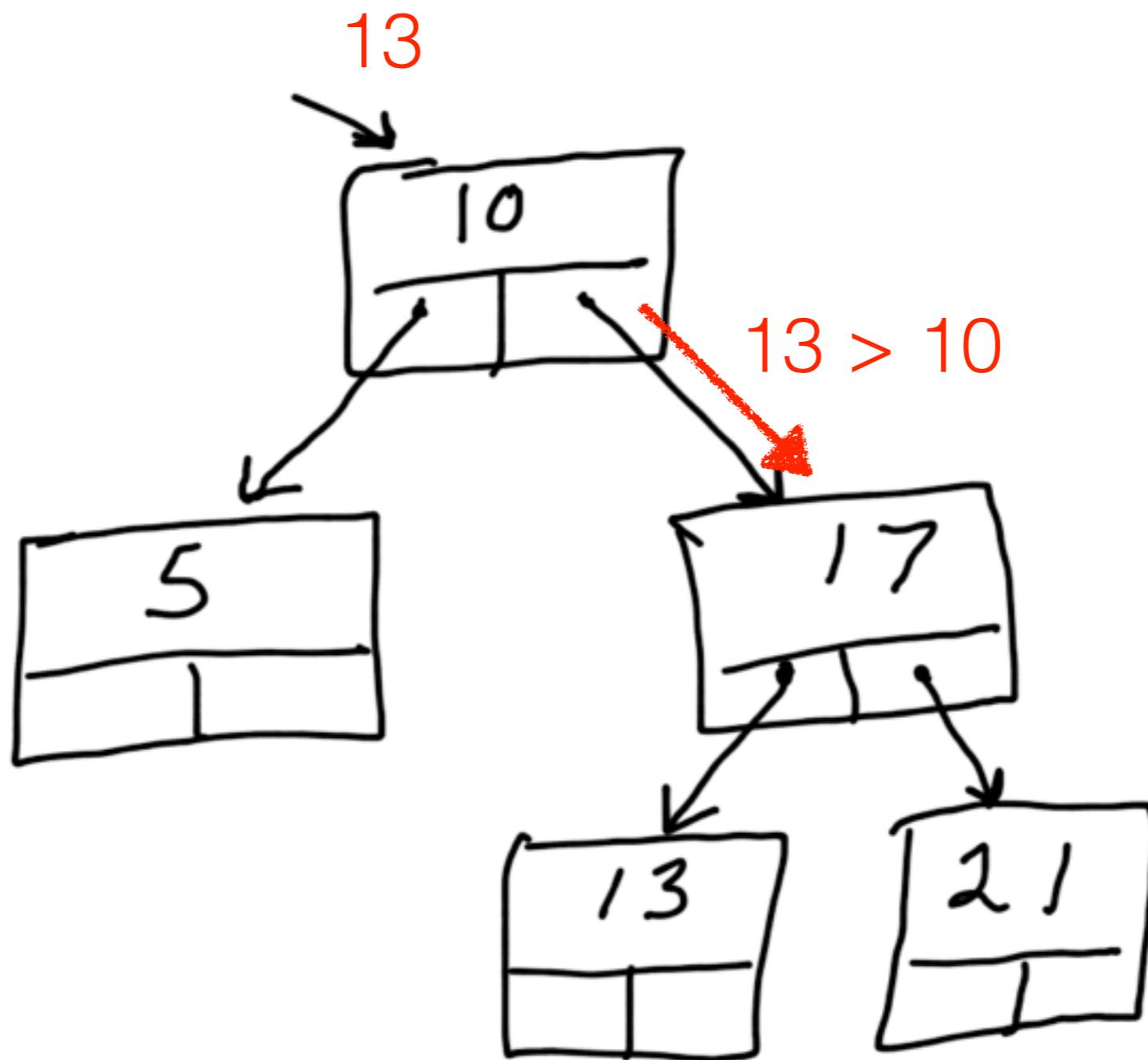
# Searching



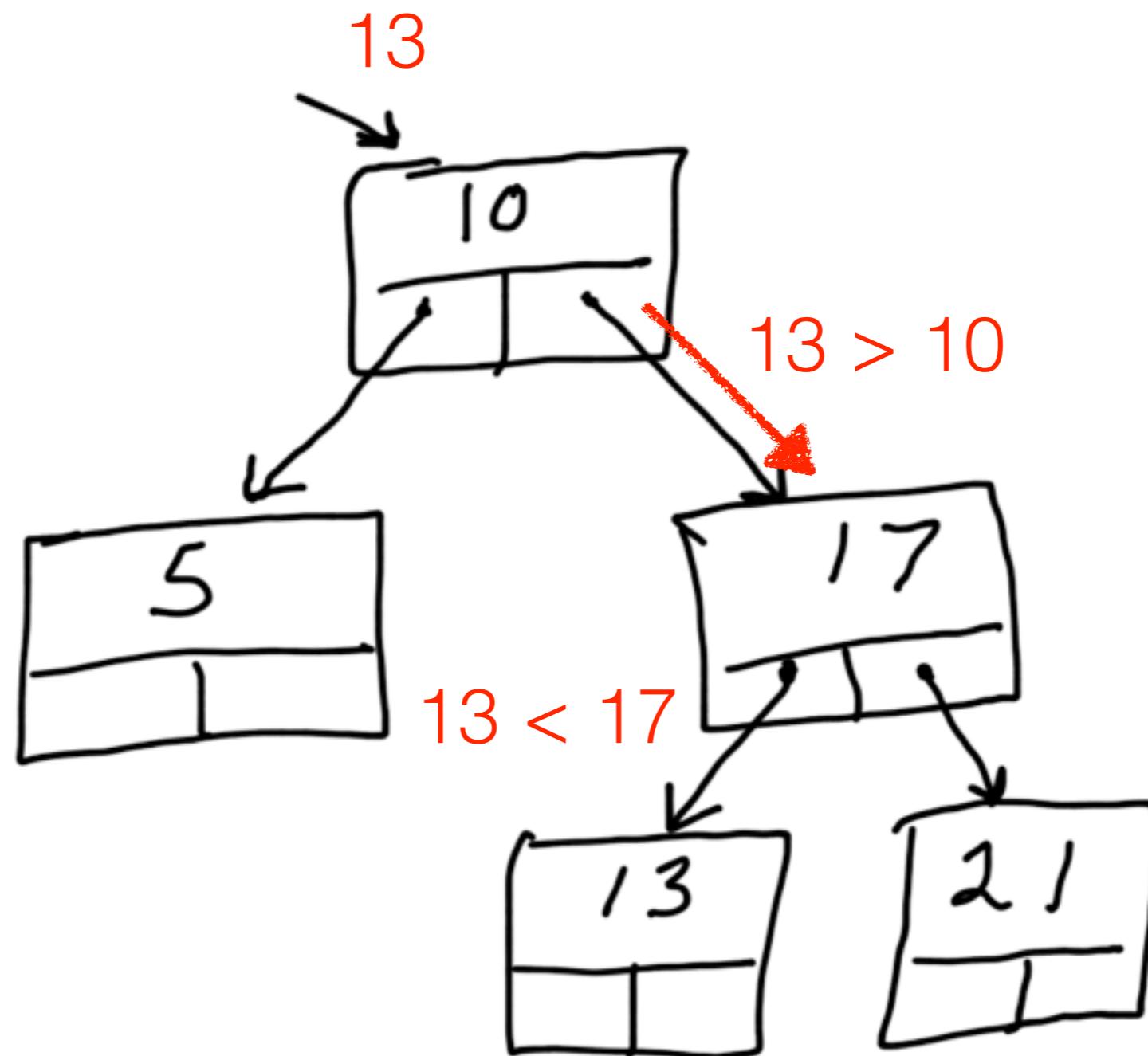
# Searching



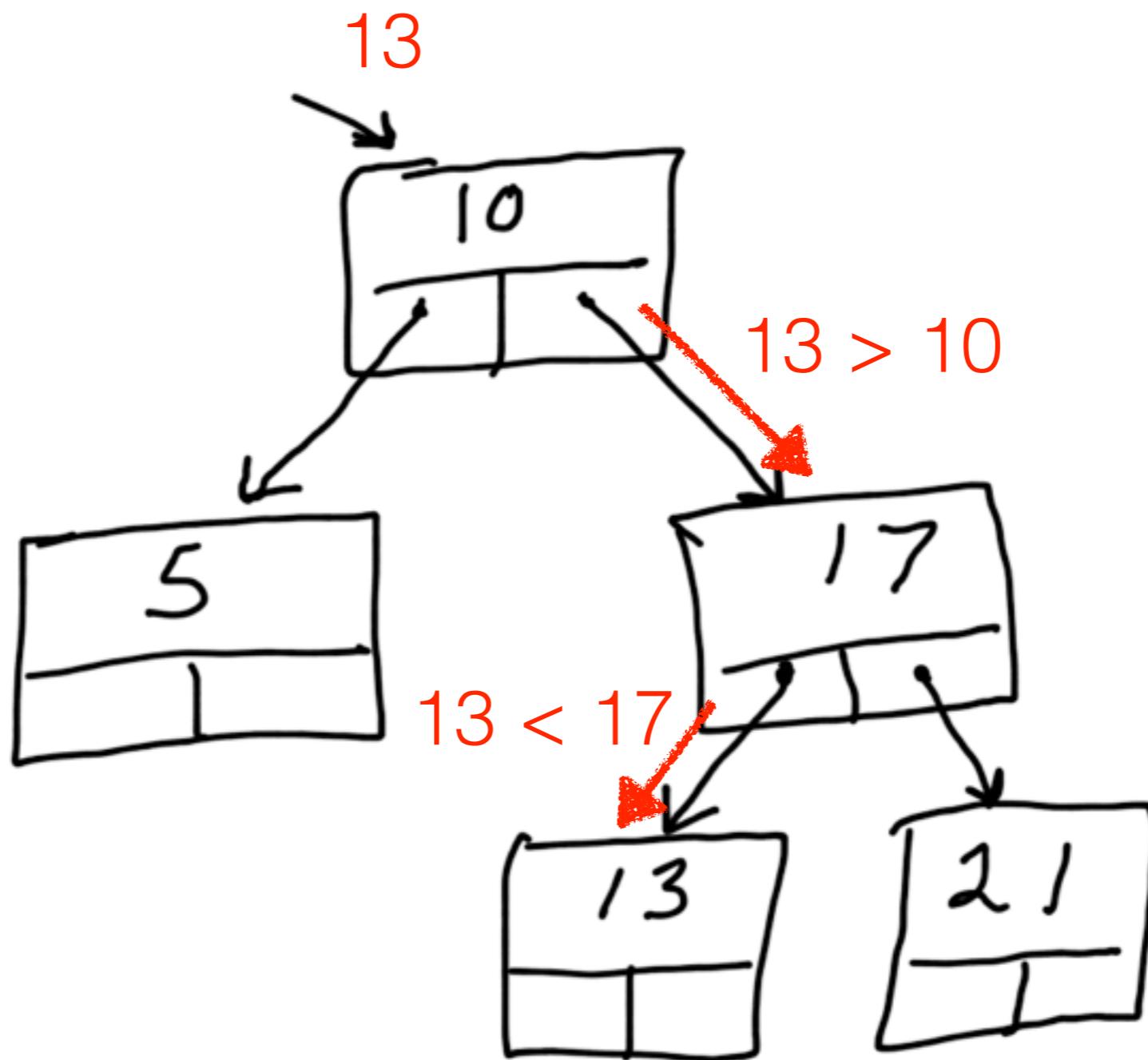
# Searching



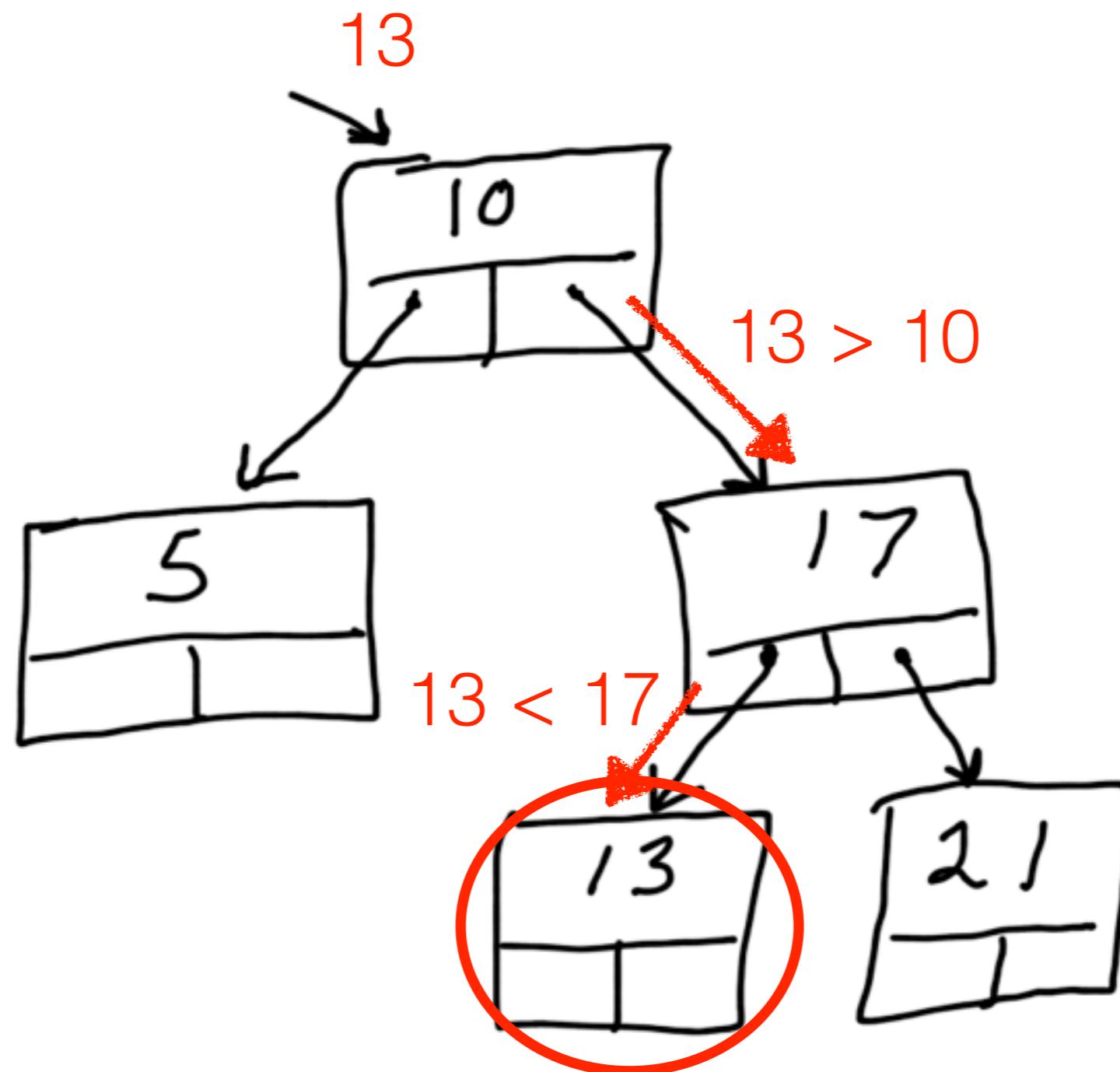
# Searching



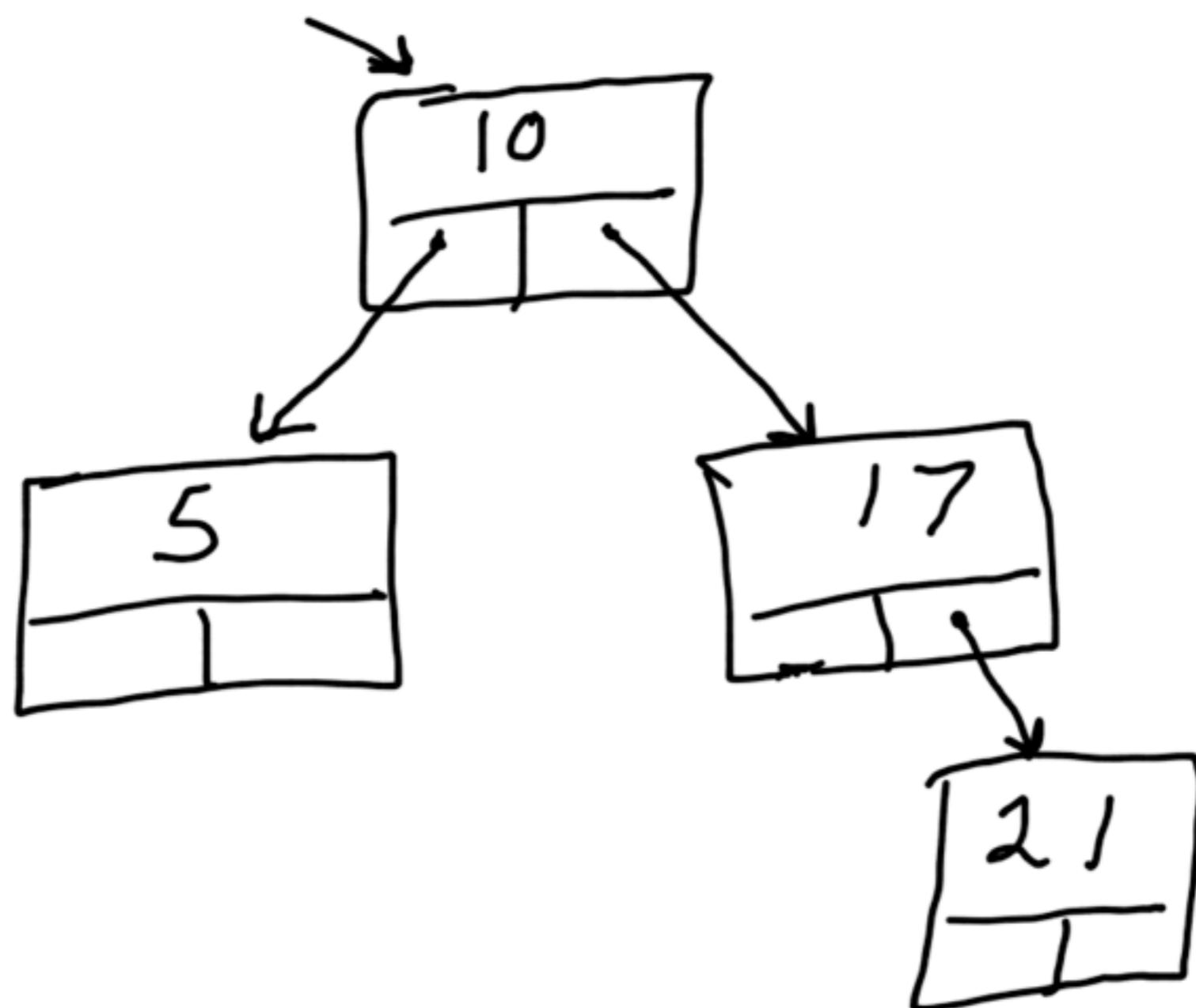
# Searching



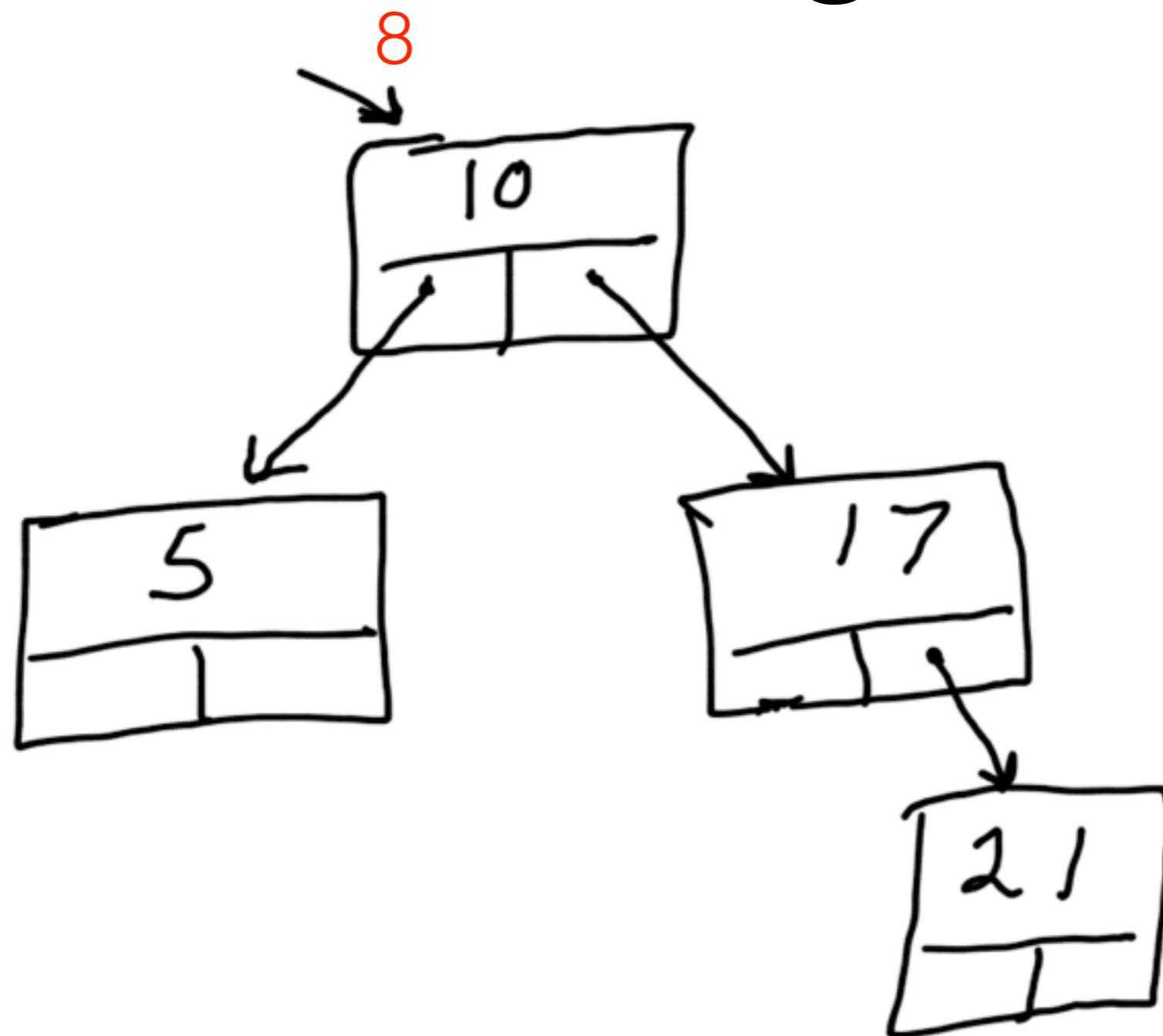
# Searching



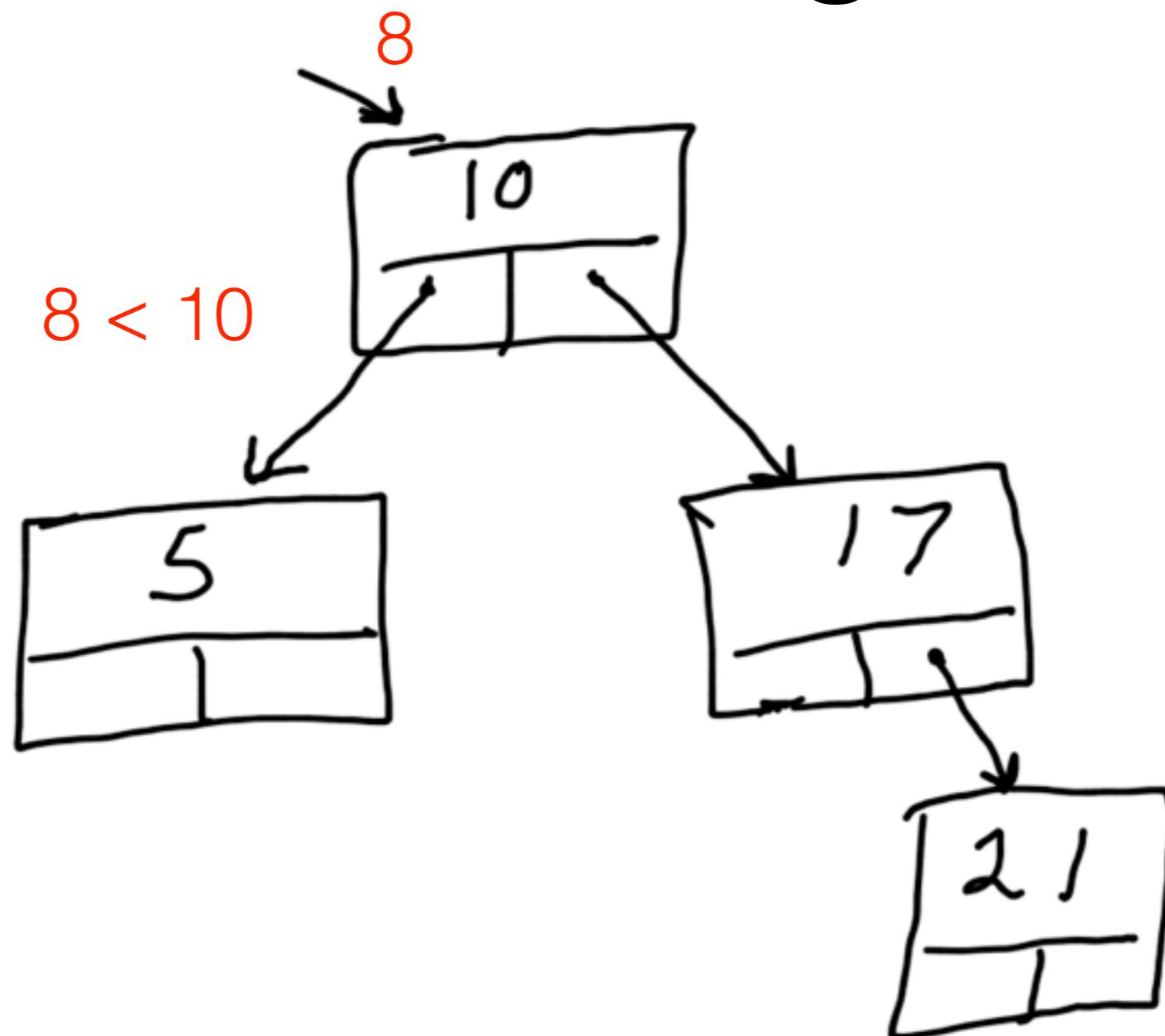
# Inserting



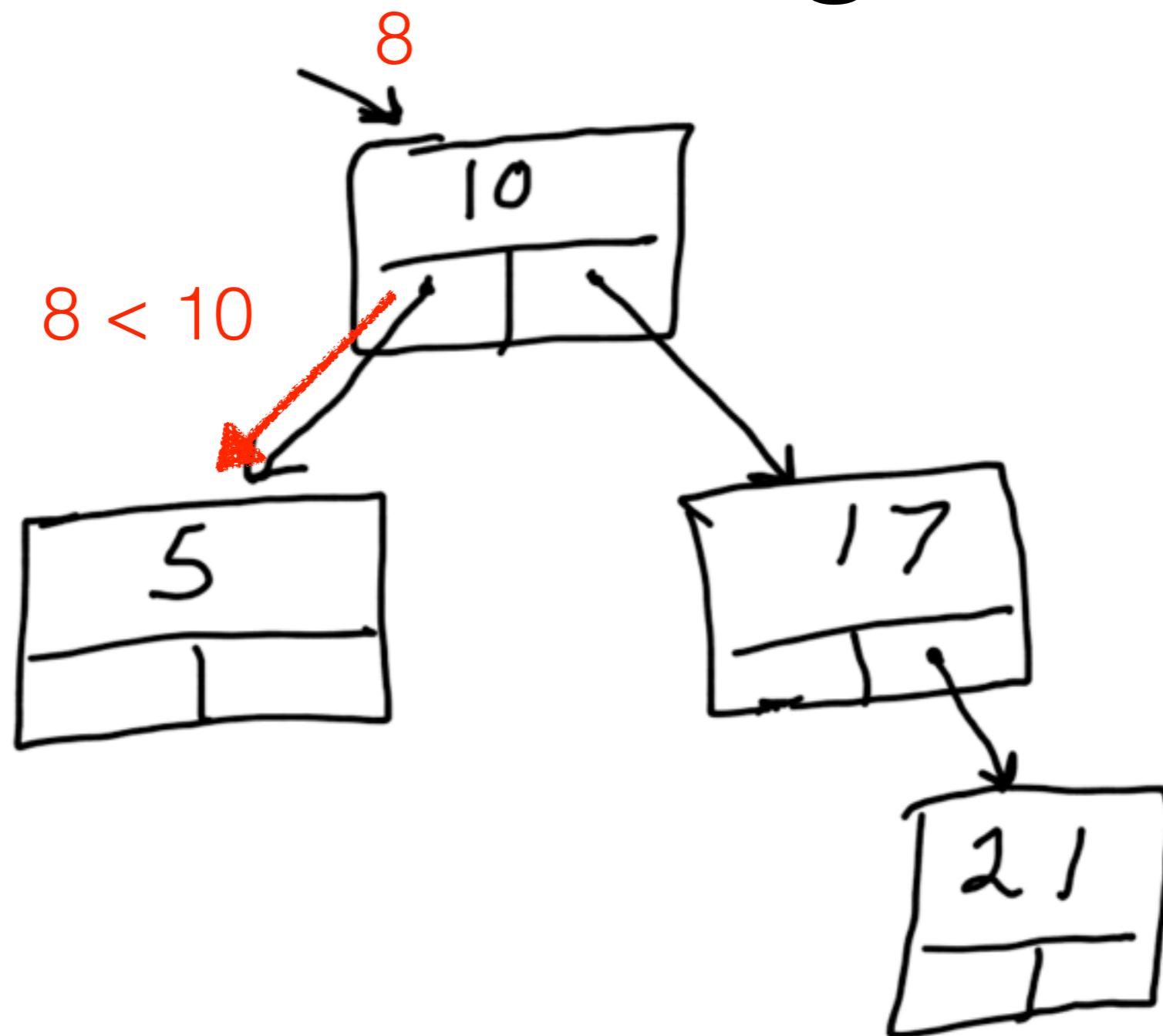
# Inserting



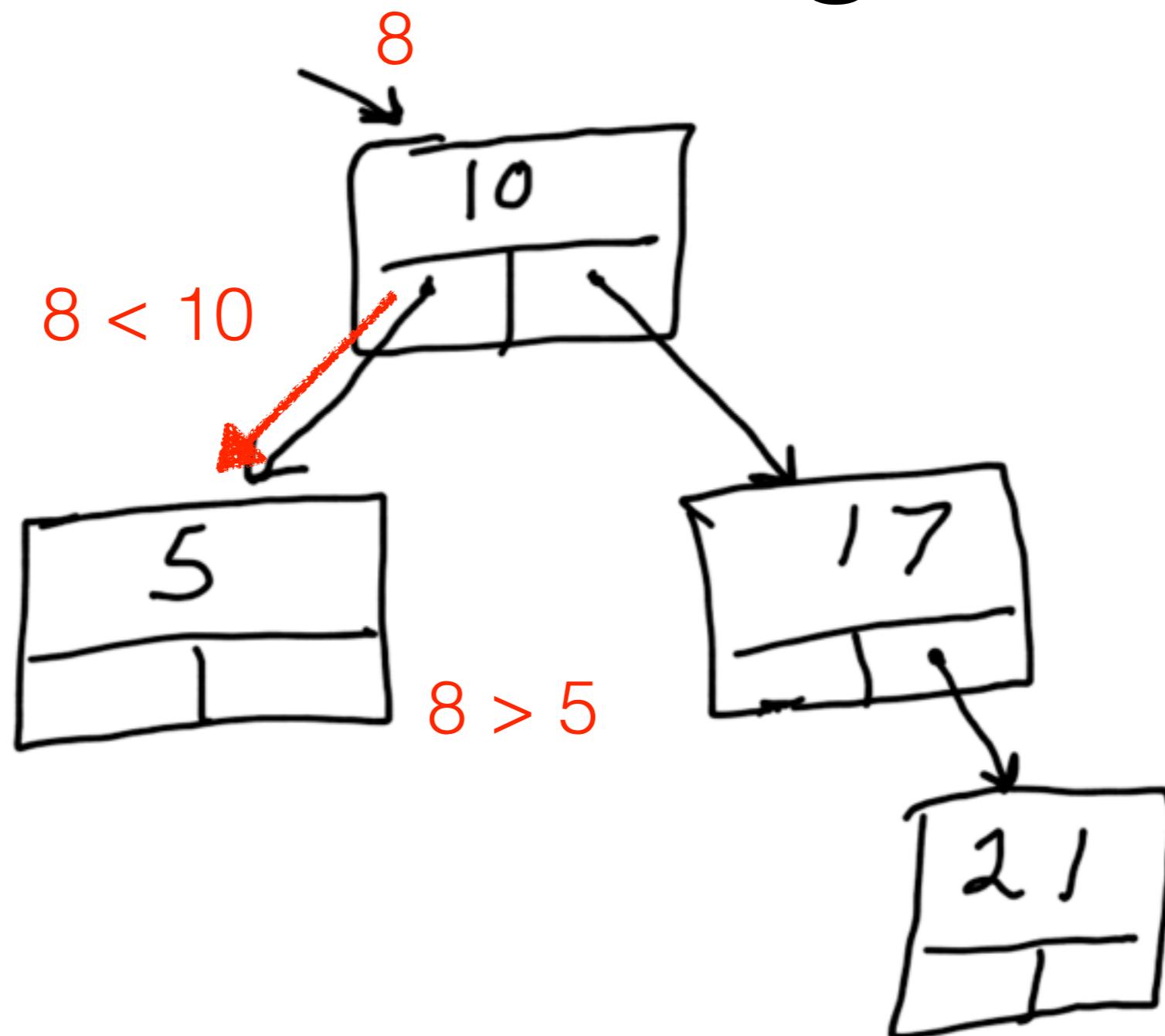
# Inserting



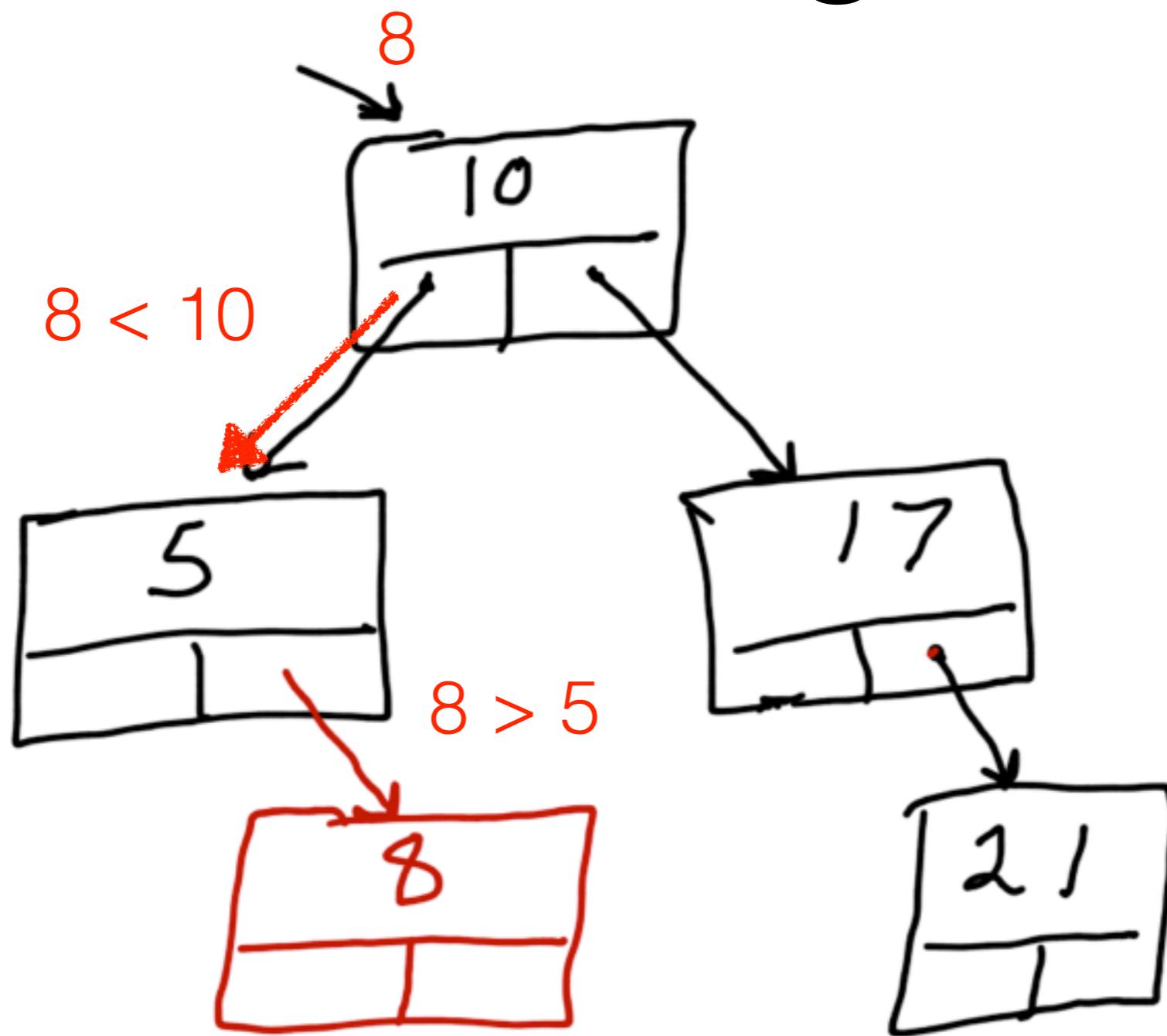
# Inserting



# Inserting



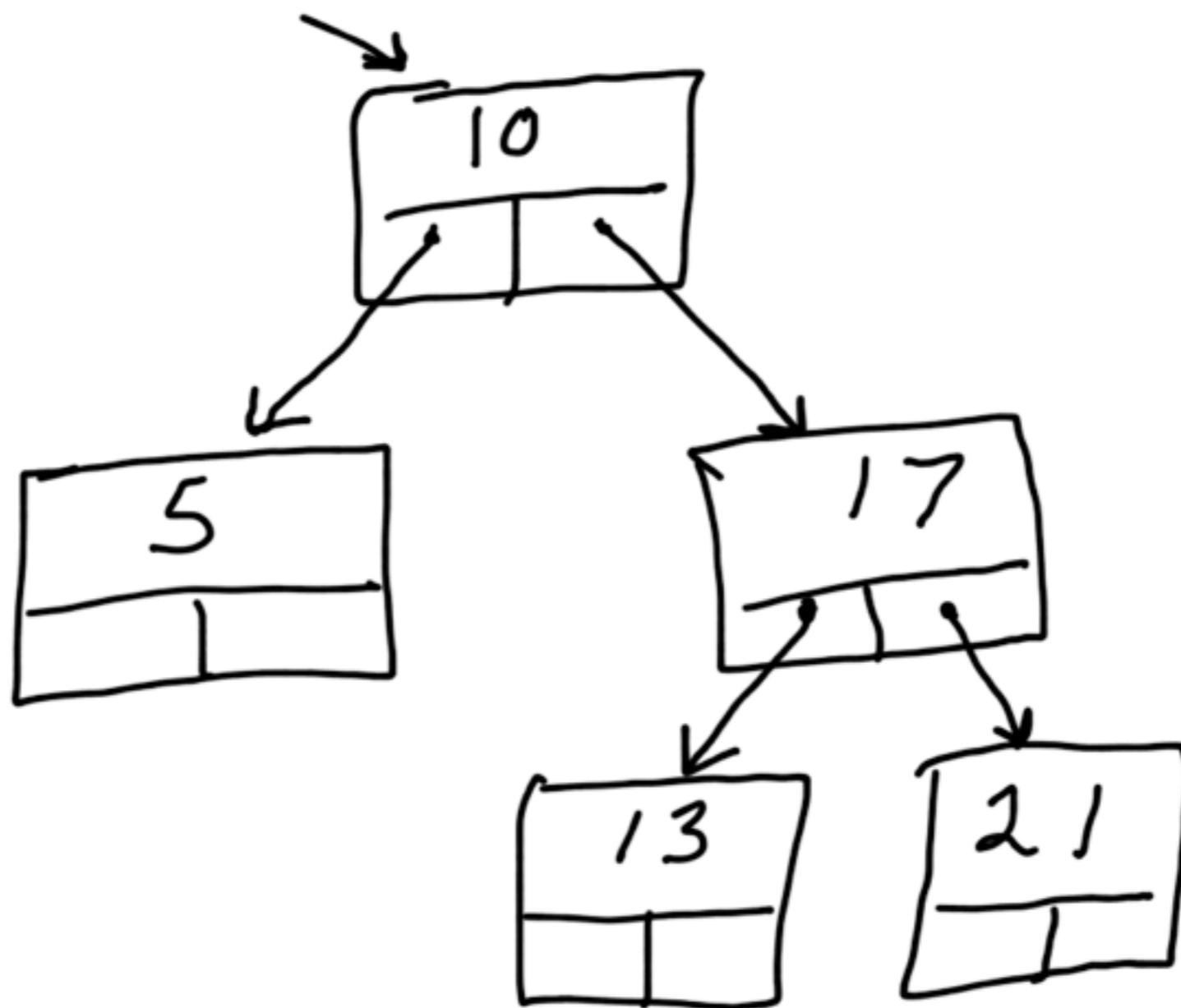
# Inserting



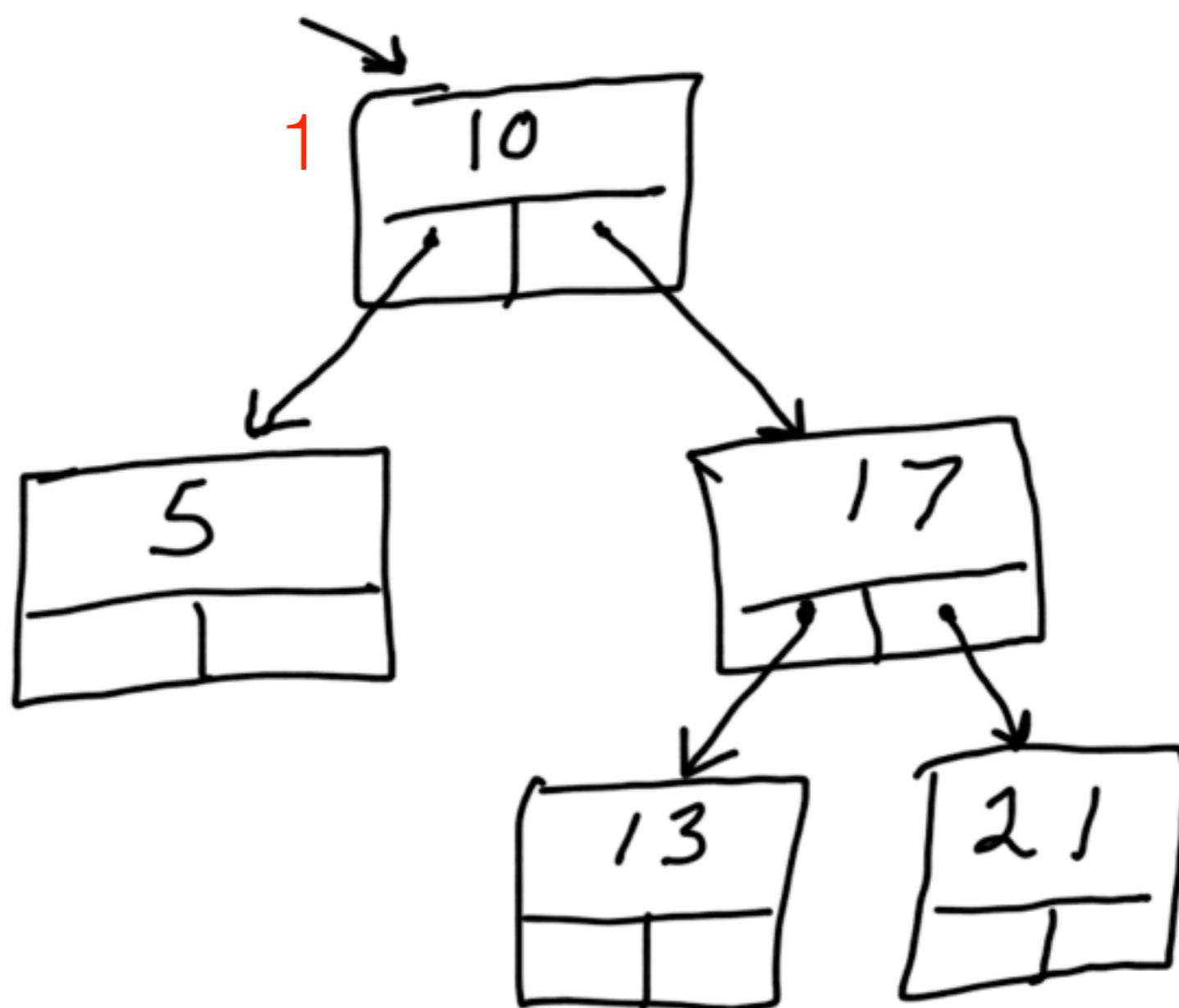
# Operations

- size
- depth
- insert
- find
- preorder traversal
- inorder traversal
- postorder traversal
- Bonus: map
- Bonus\*: delete

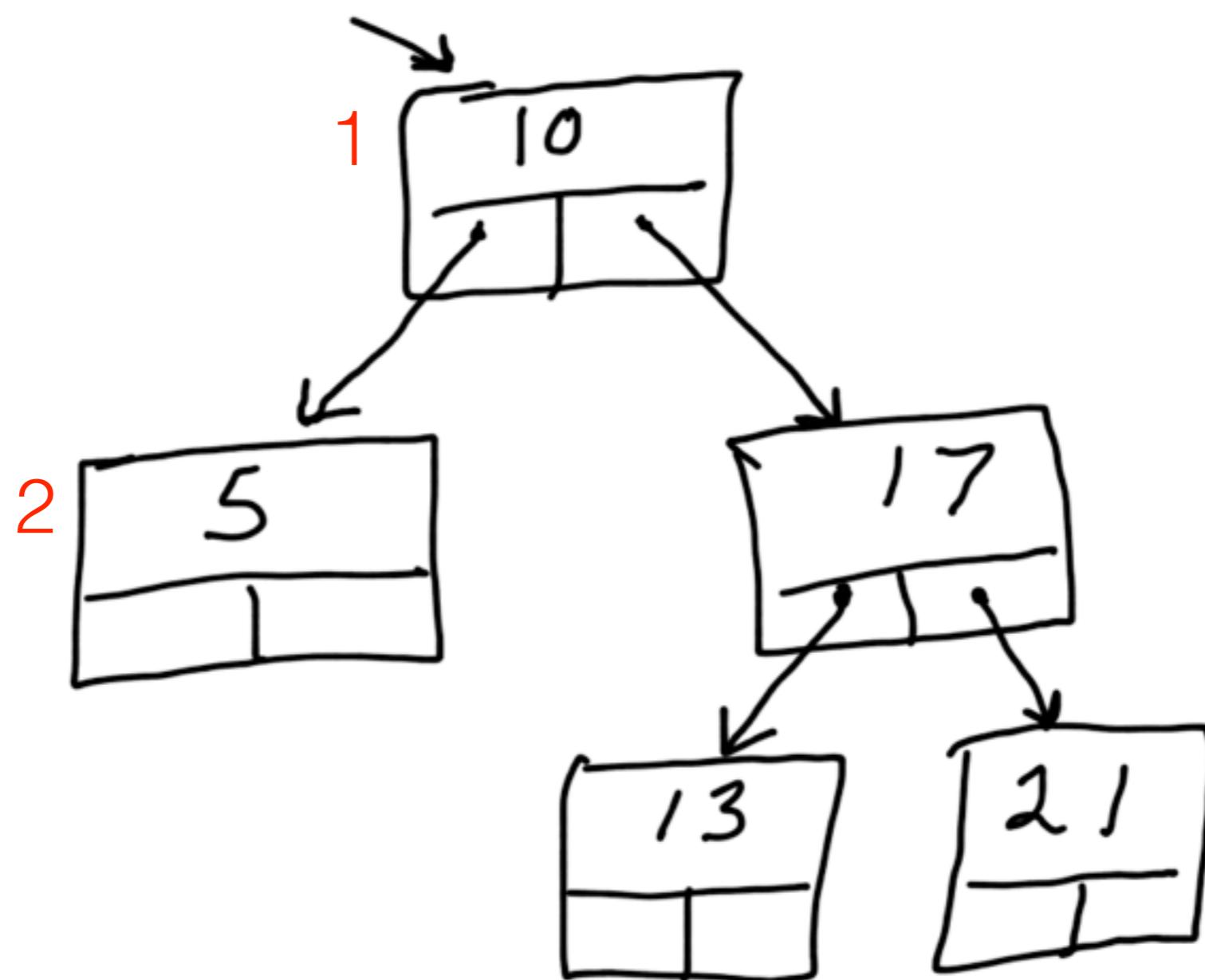
# Preorder



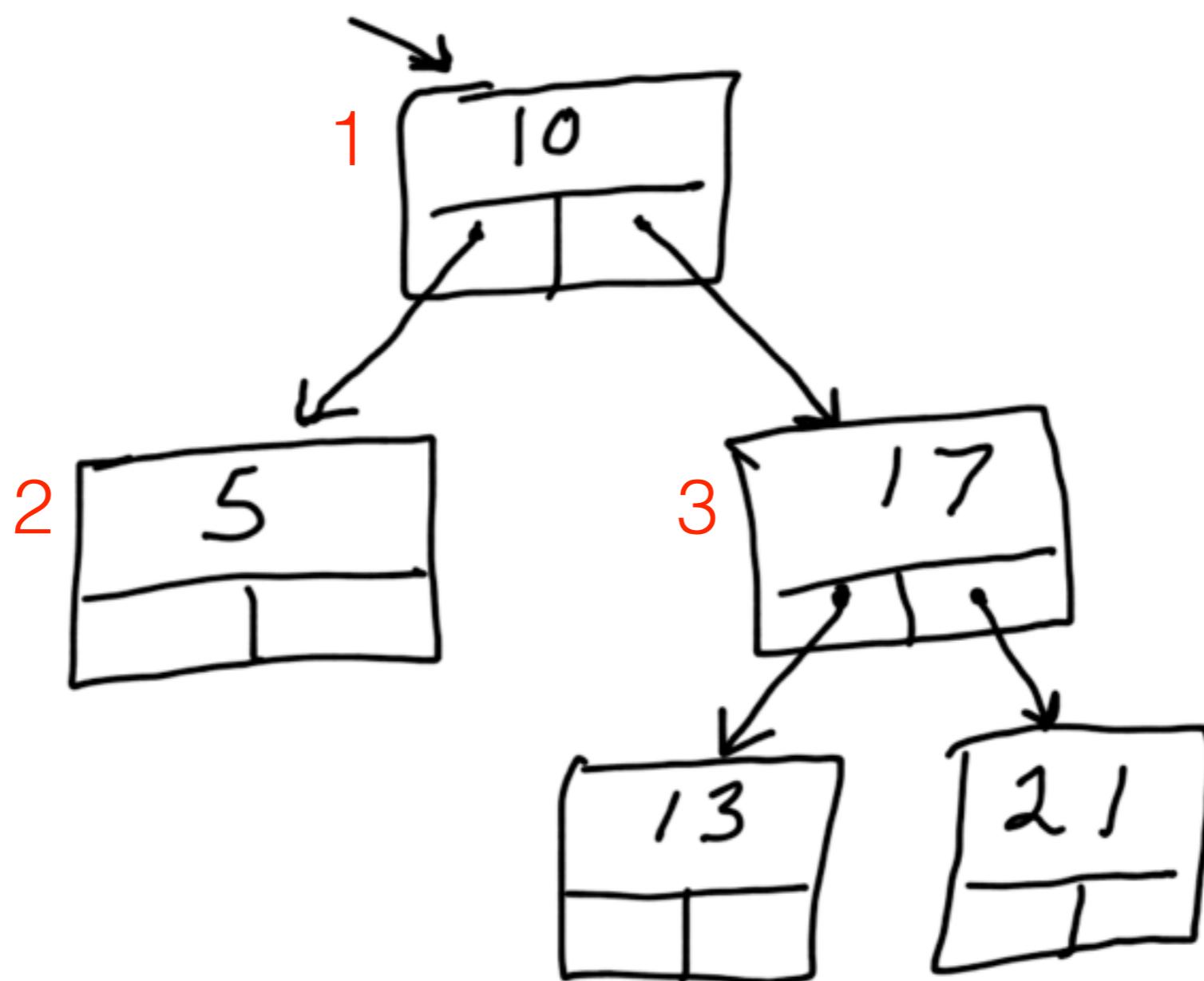
# Preorder



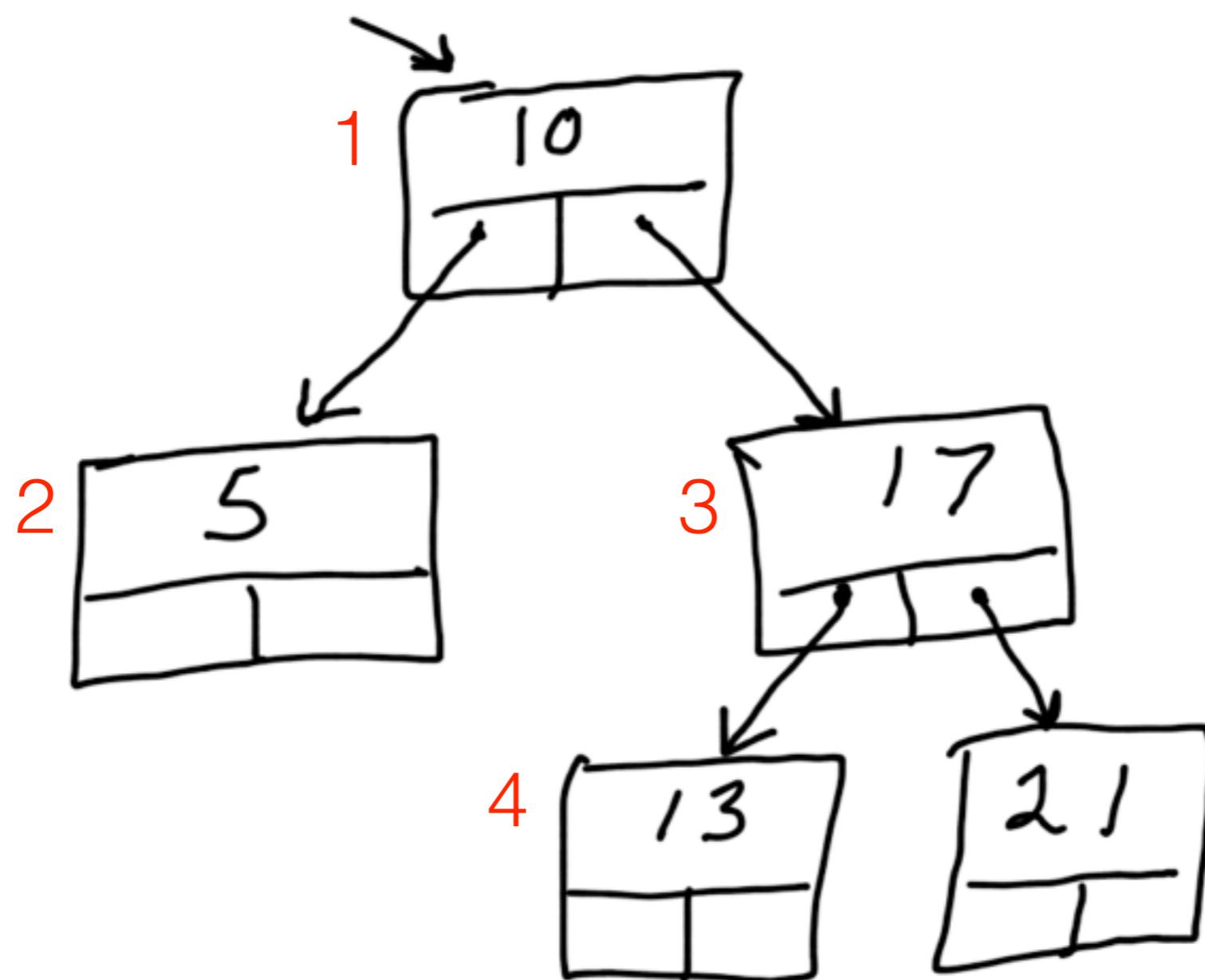
# Preorder



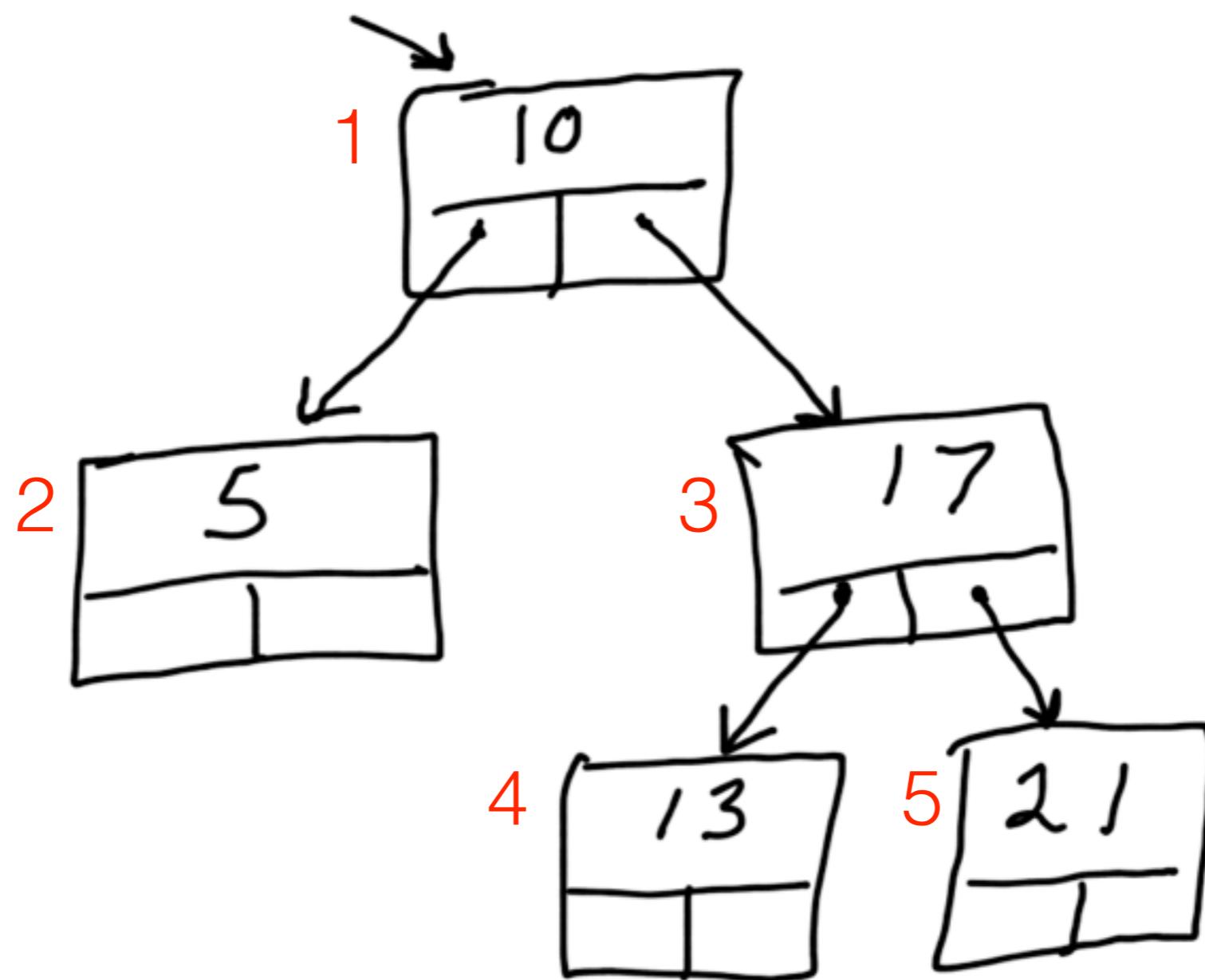
# Preorder



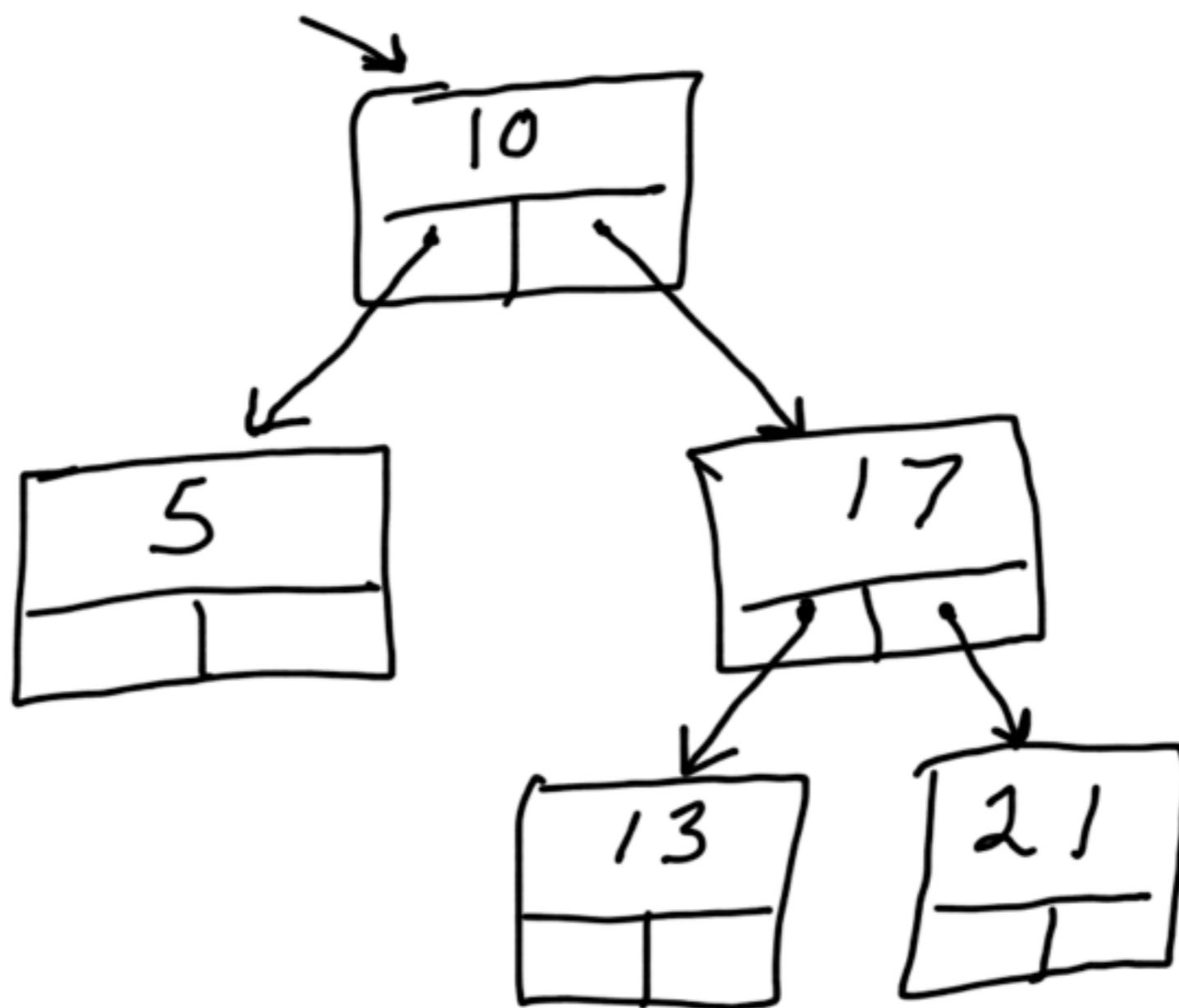
# Preorder



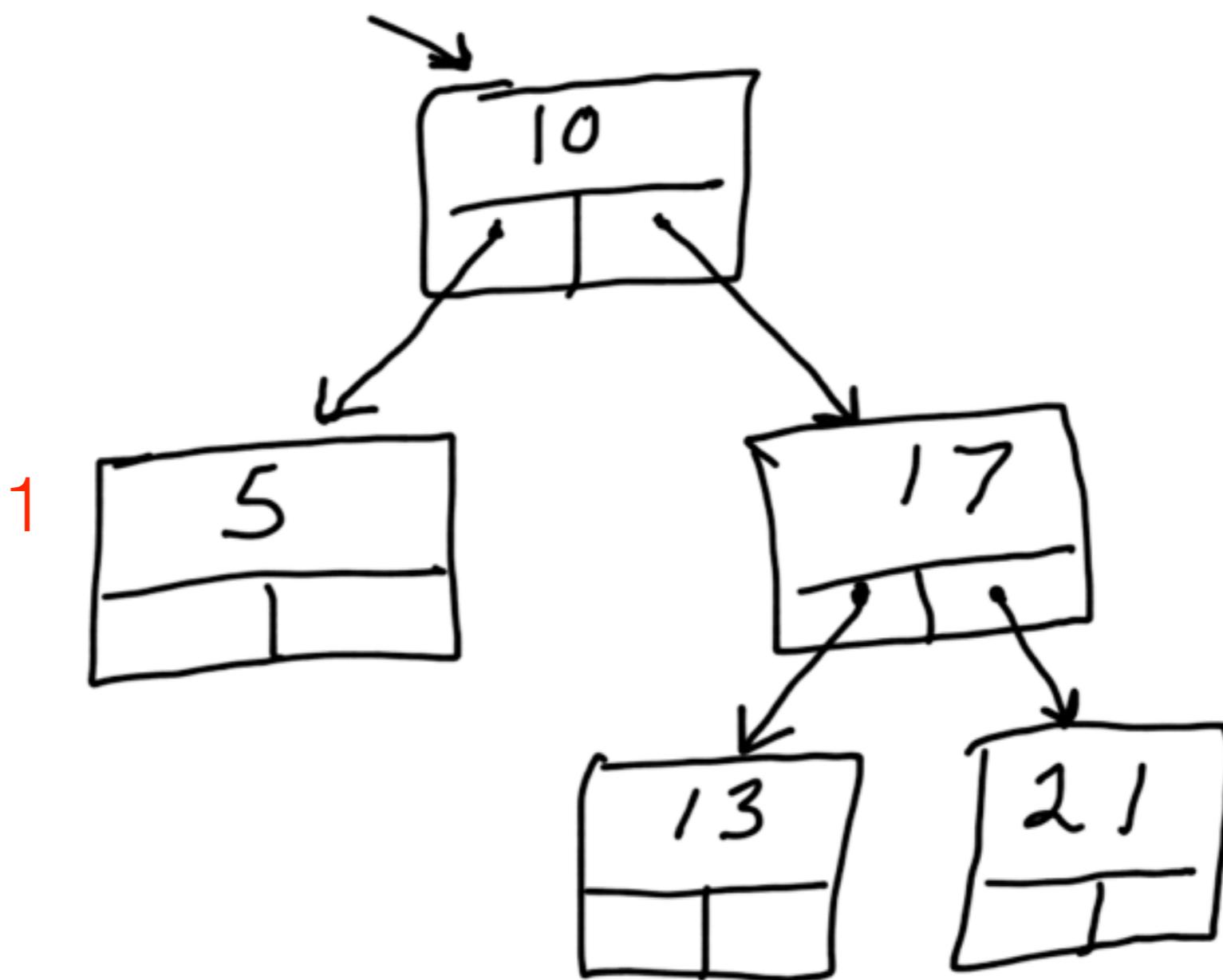
# Preorder



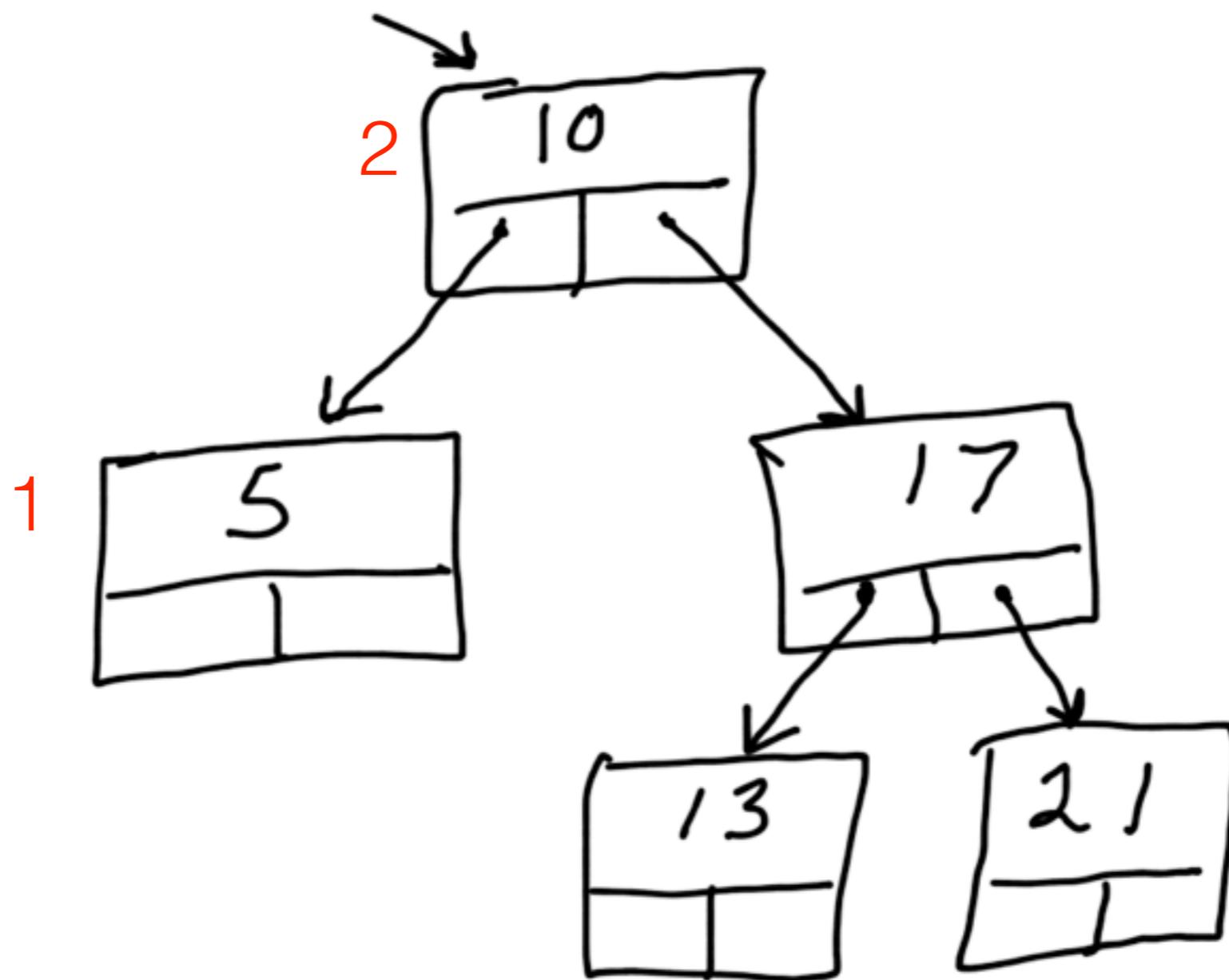
# Inorder



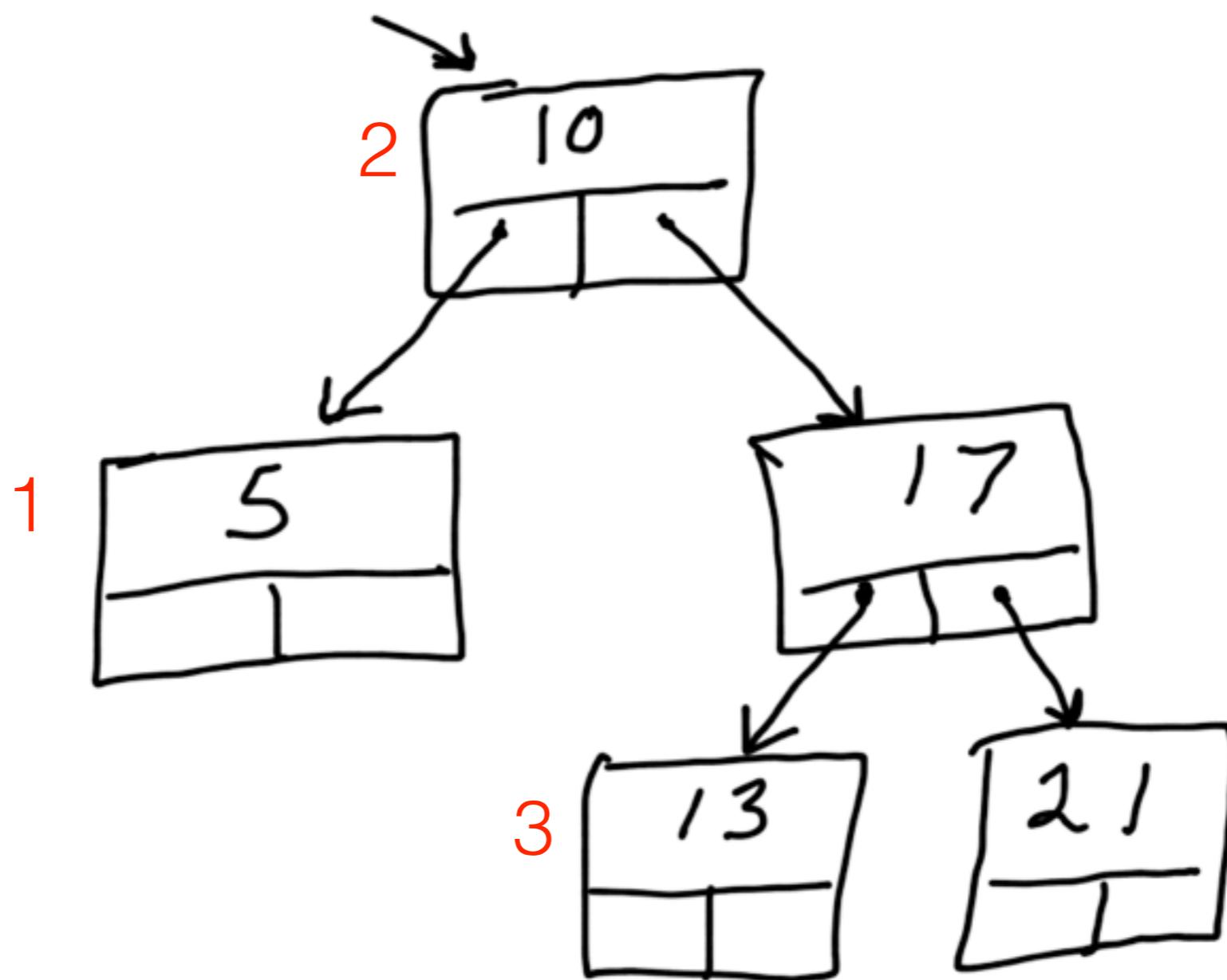
# Inorder



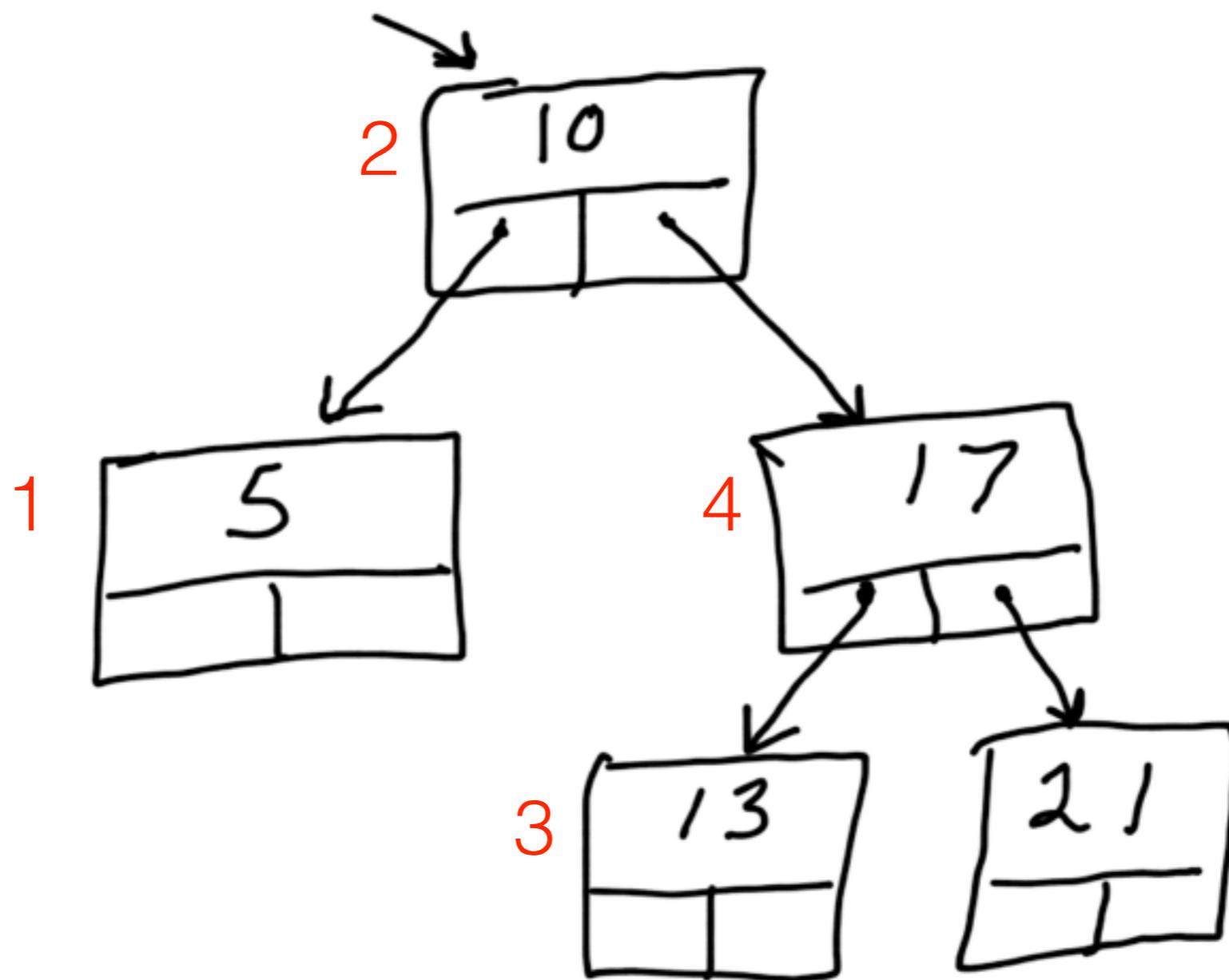
# Inorder



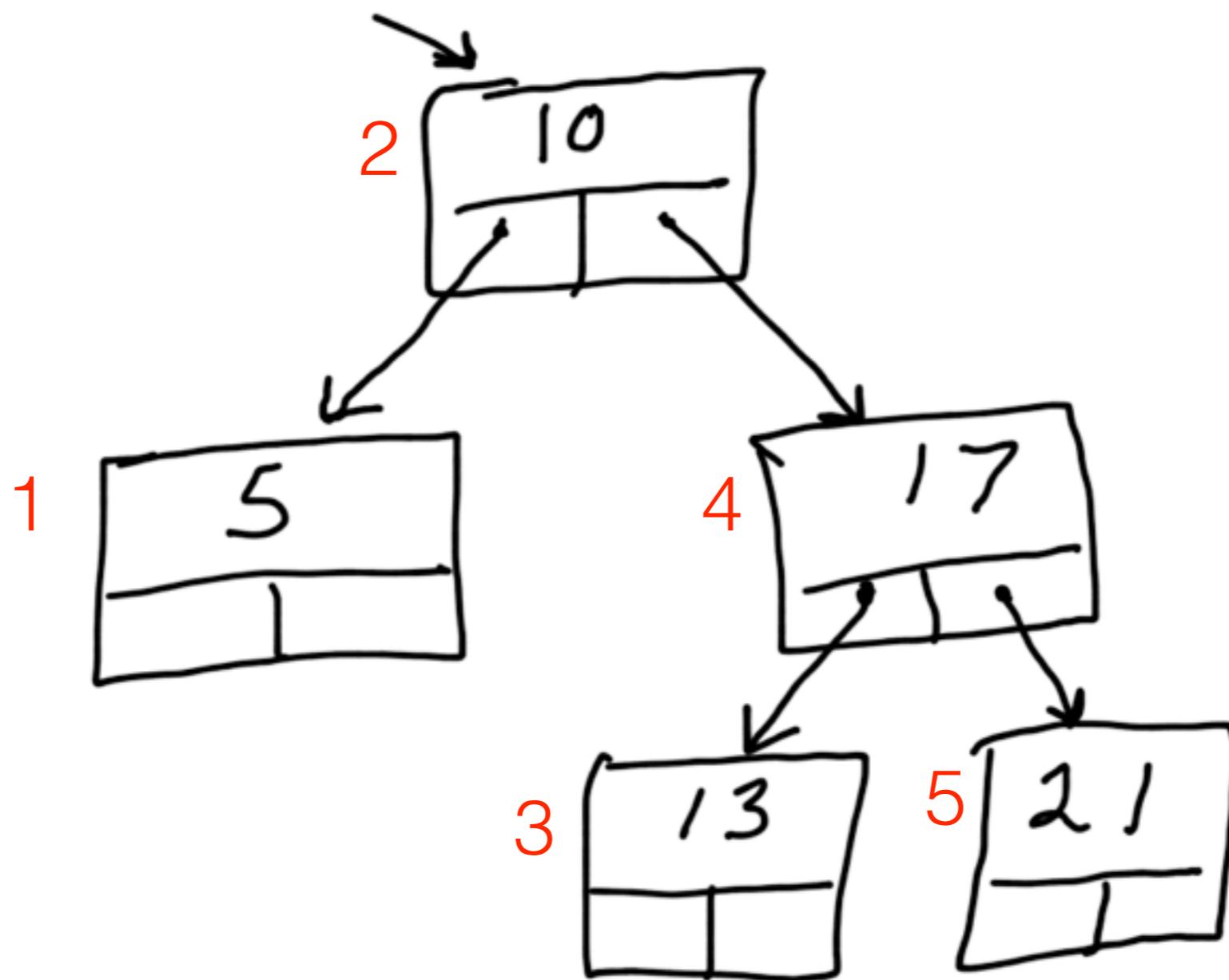
# Inorder



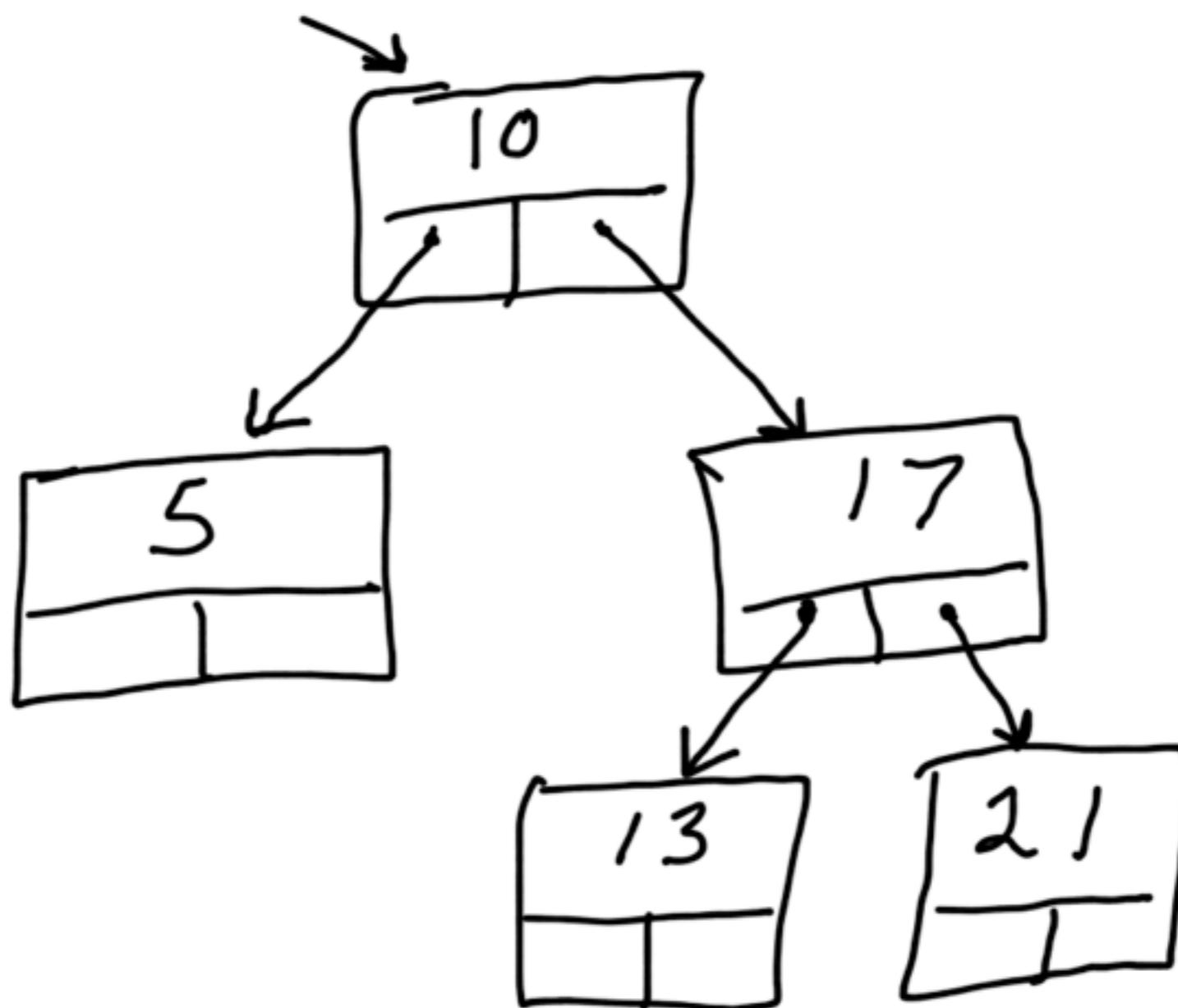
# Inorder



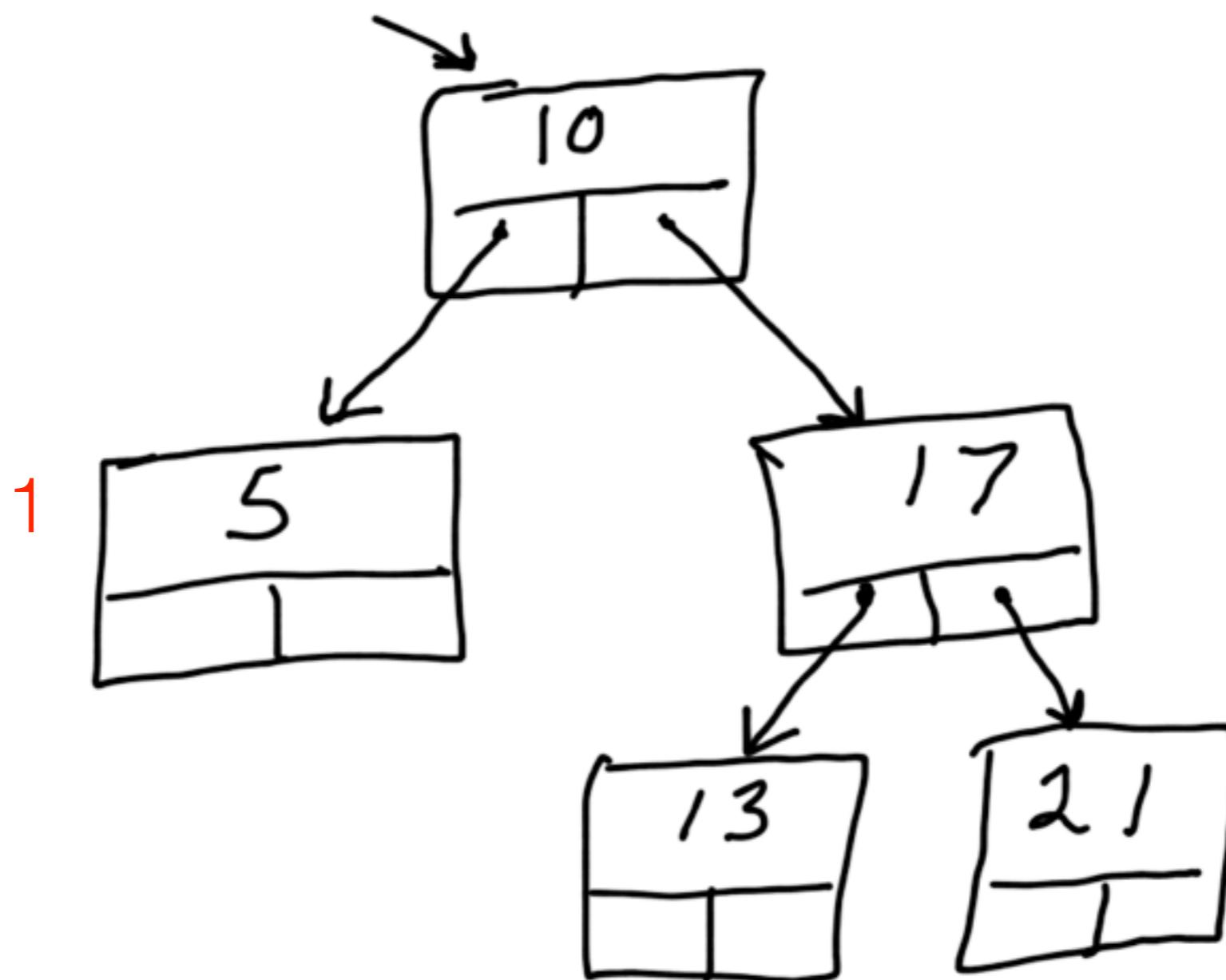
# Inorder



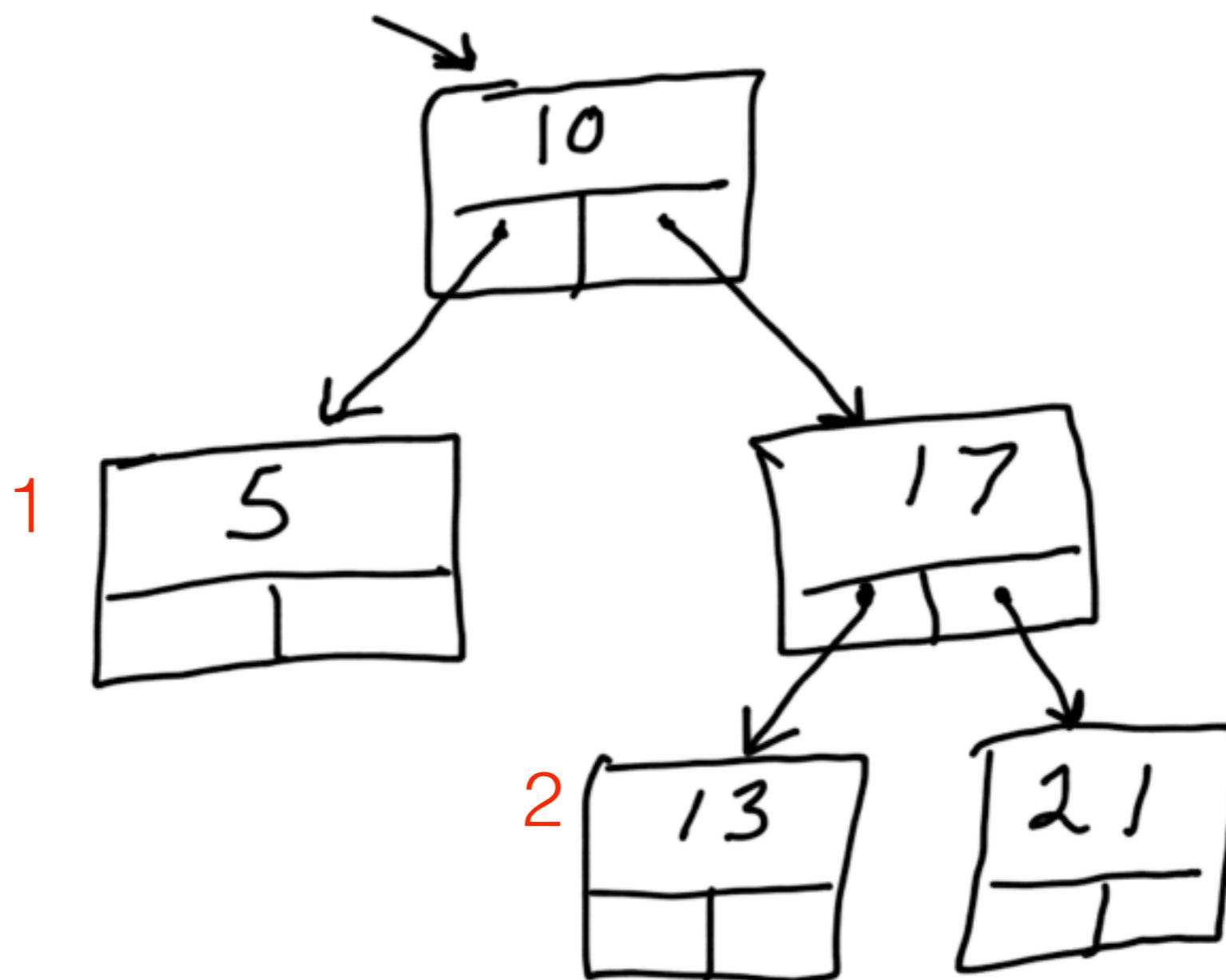
# Postorder



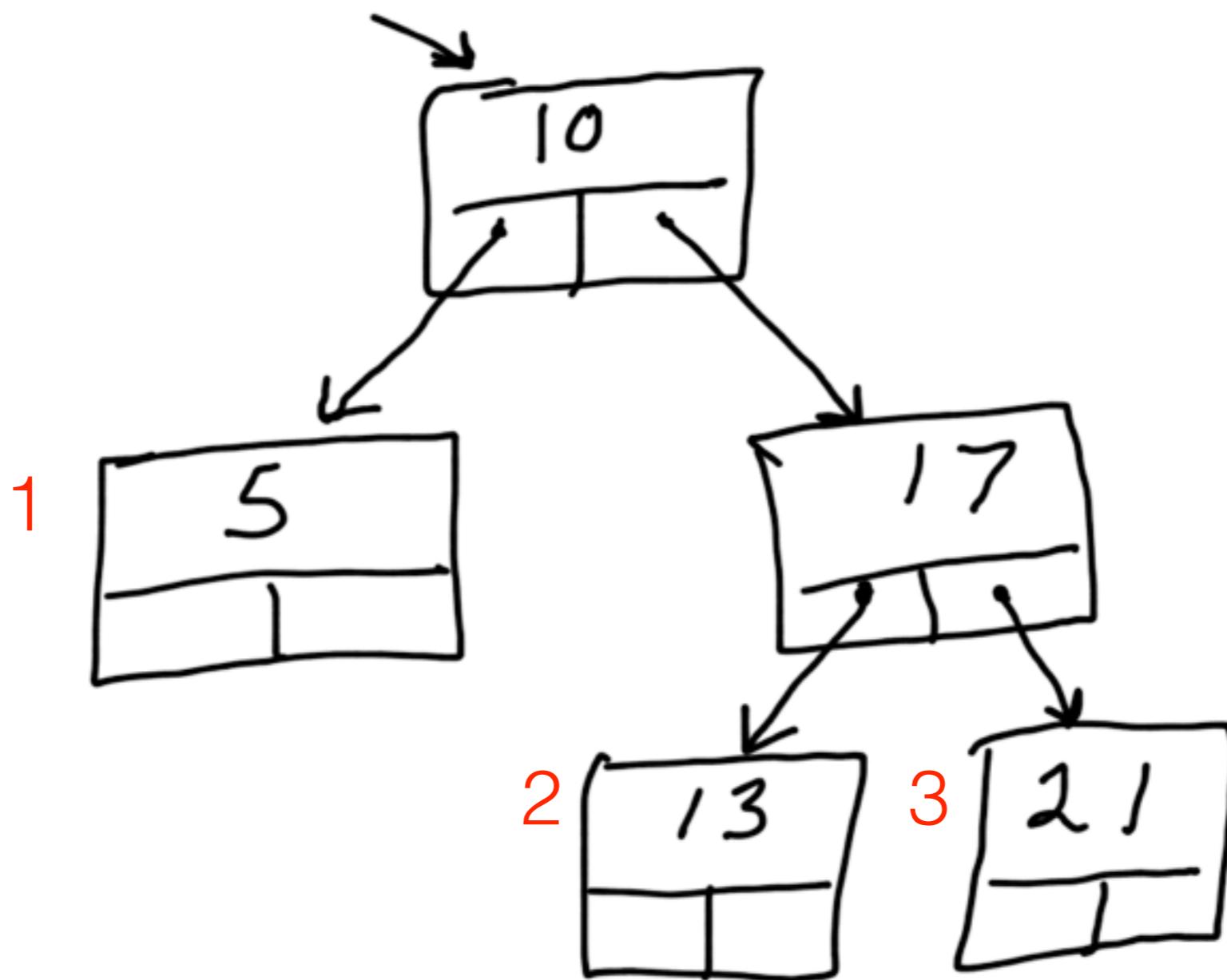
# Postorder



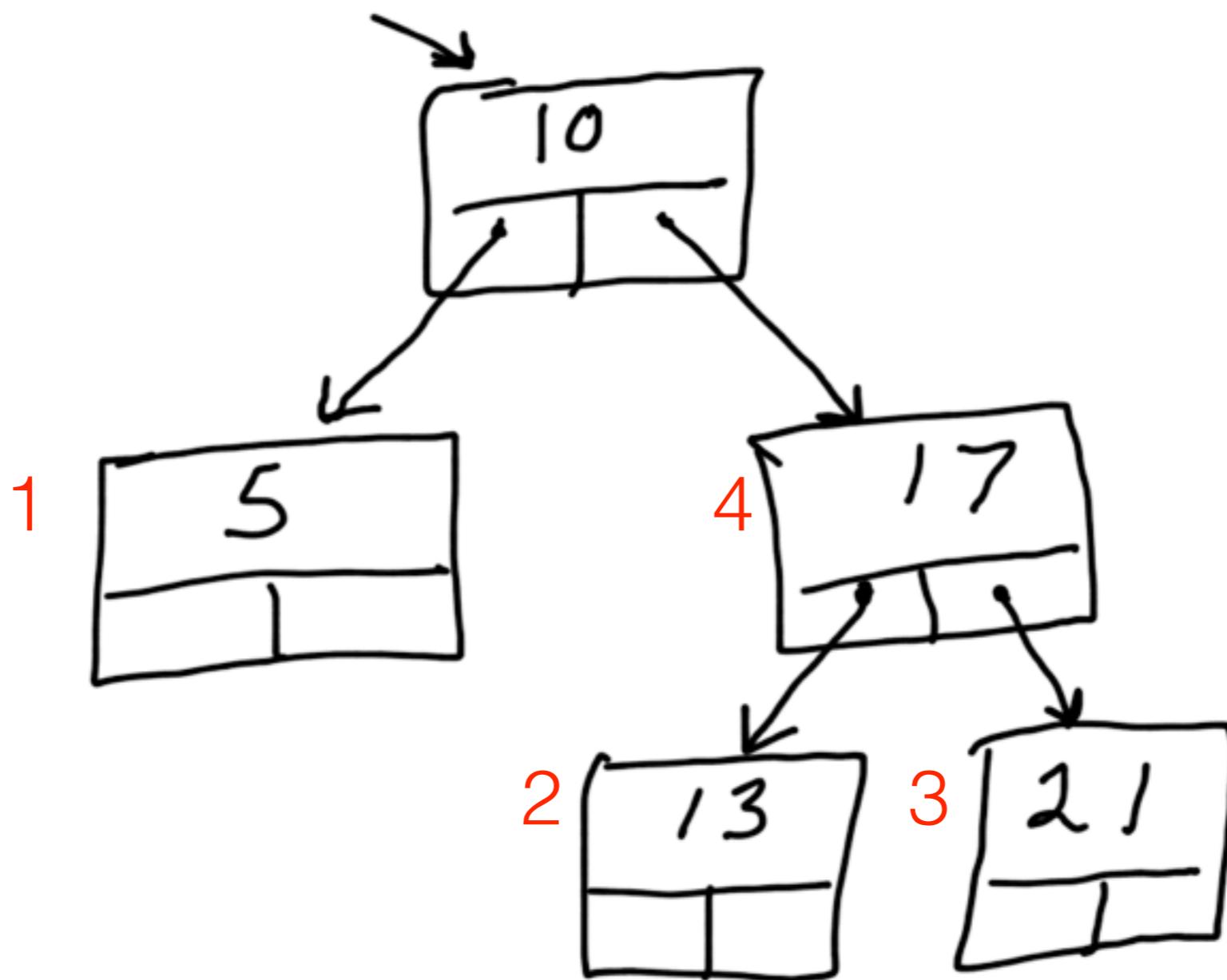
# Postorder



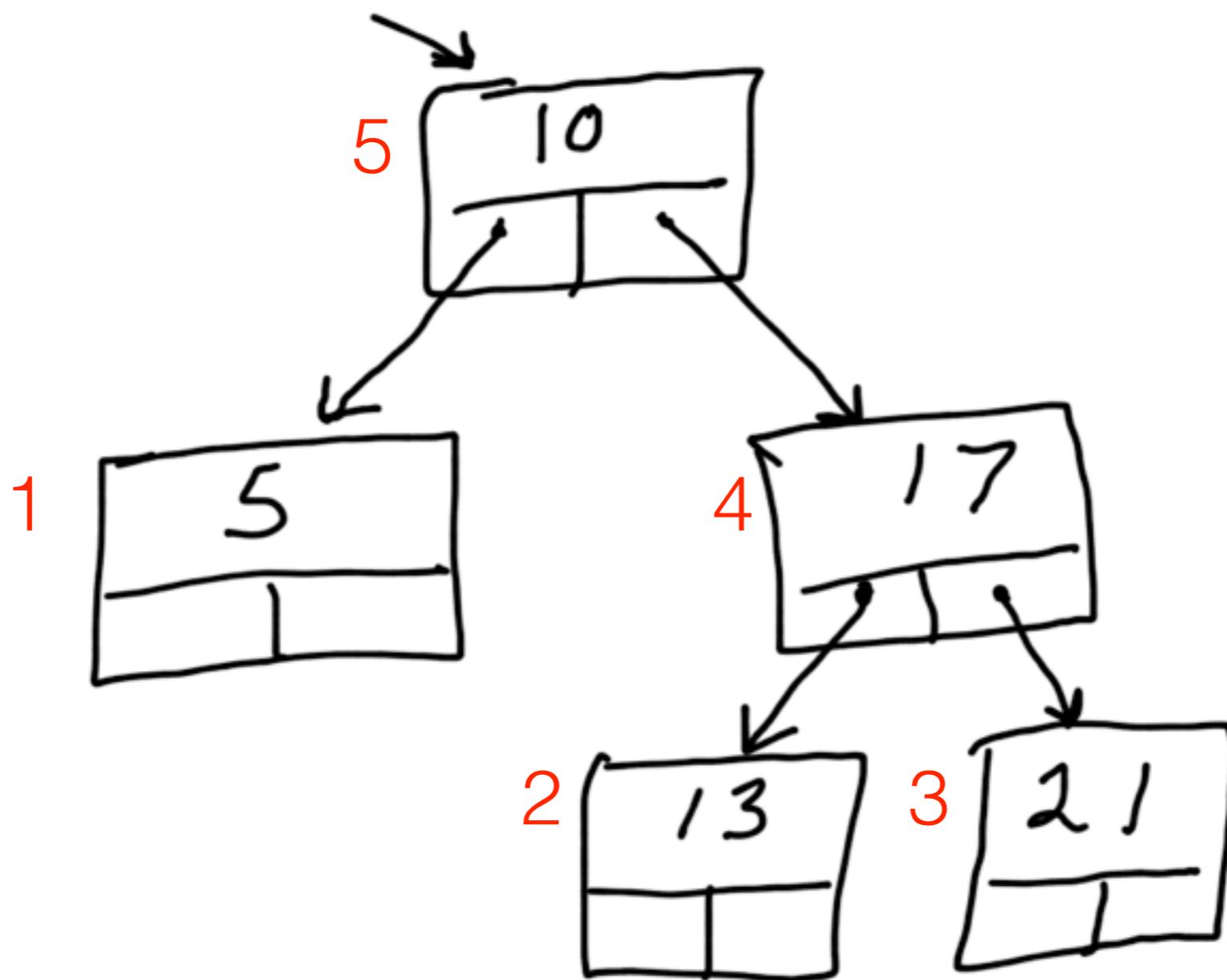
# Postorder



# Postorder



# Postorder



# Exercise

- Implement DbcBinaryTree
  - <https://github.com/dastels/dbc-deep-dives.git>
  - DataStructures directory
  - delete the contents of the lib dir, and put dbc\_binary\_tree.rb there
  - Use rspec spec/binary\_tree\_spec.rb to guide you

# Exercise

- Make recursive solutions.
- Let's walk through implementing size together.

# AVL Tree

# But First

- Lots of nil checks are messy
- Use a nil node object instead

# What

- A binary search tree that is maintained in a balanced state
- The depths of the left & right subtrees at any node differ by at most 1

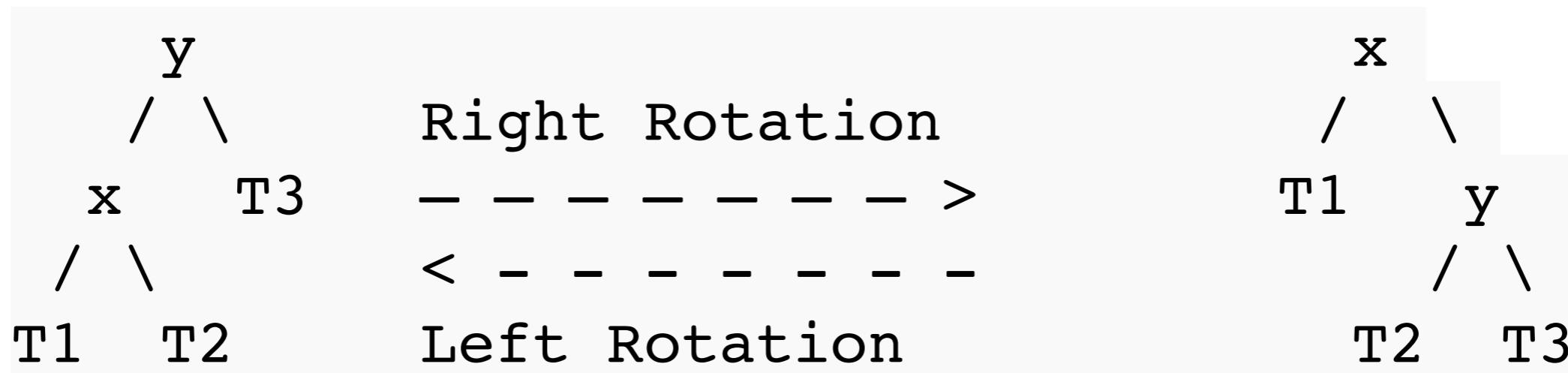
# Why

- Keeping a binary search tree balanced maintains  $O(\log n)$  performance

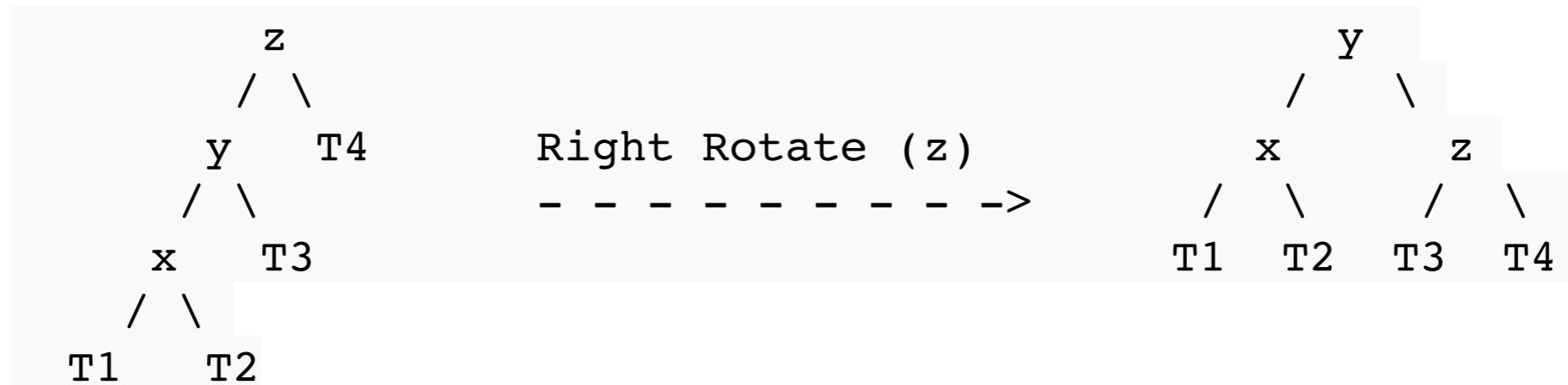
# How

- After each insertion into a node, the balance of that node is checked. If the node is unbalanced, it needs to be rebalanced via rotation.

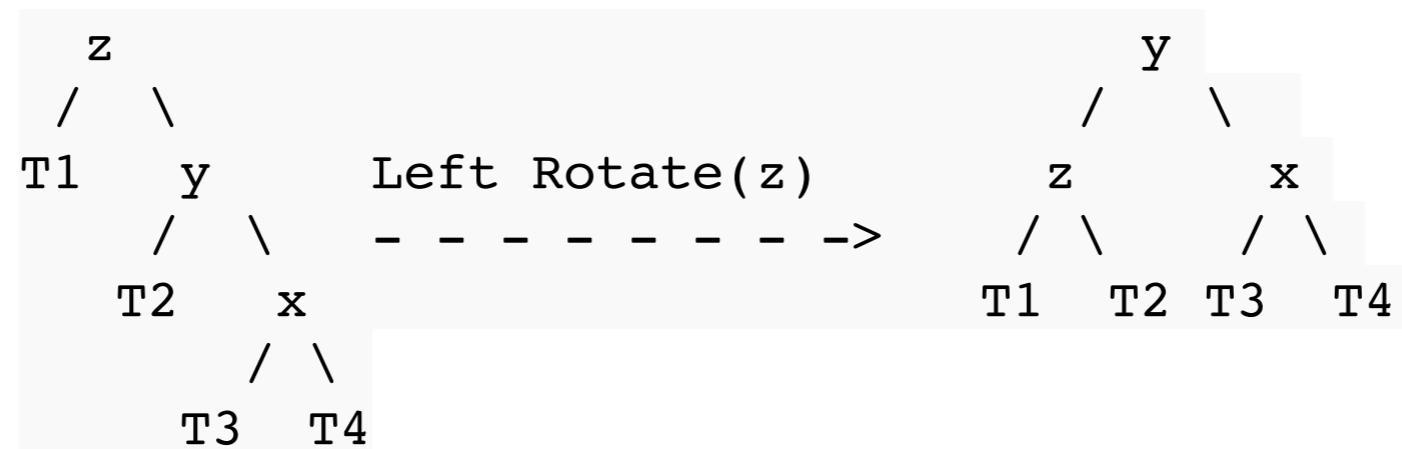
# Left & Right Rotation



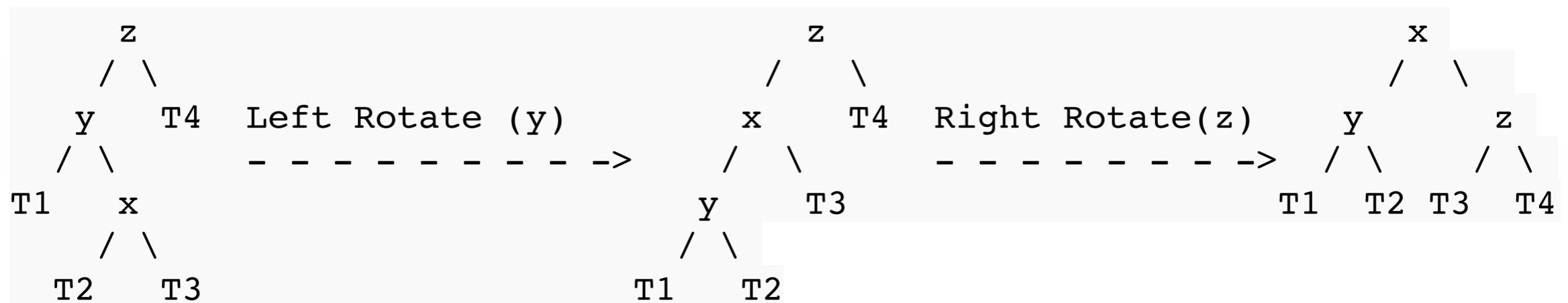
# Left Left Case



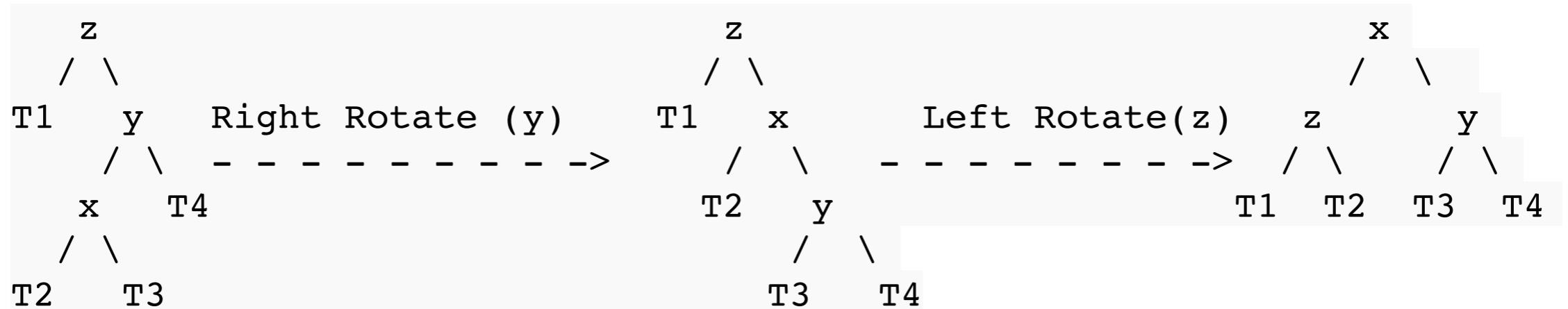
# Right Right Case



# Left Right Case



# Right Left Case



# Rotating Left

```
def left_rotate
  AVLTreeNode.new(@right.value,
                  AVLTreeNode.new(@value,
                                  @left,
                                  @right.left),
                  @right.right)
end
```

# Rotating Left

```
def right_rotate
  AVLTreeNode.new(@left.value,
                  @left.left,
                  AVLTreeNode.new(@value,
                                  @left.right,
                                  @right))
end
```

# References

- [http://en.m.wikipedia.org/wiki/AVL\\_tree](http://en.m.wikipedia.org/wiki/AVL_tree)
- <https://www.cs.auckland.ac.nz/software/AlgAnim/AVL.html>
- <http://pages.cs.wisc.edu/~ealexand/cs367/NOTES/AVL-Trees/index.html>

# B-Tree

# Graph

# Hashtable

# Set

Bag  
MultiSet

# Sorting

# Bubble Sort

# Insertion Sort

# Quicksort

# Contact

[dastels@icloud.com](mailto:dastels@icloud.com)  
@dastels

# Bubble Sort

# Insertion Sort

# Quicksort