

# Recess Week Remedial

**CS1101S AY20/21 SEM 1**  
**Studio 03A**

**Chen Xihao**  
**Year 2 Computer Science**

**[chenxihao@u.nus.edu](mailto:chenxihao@u.nus.edu)**  
**@BooleanValue**

# Recess Week Remedial :’)

## Agenda

- Searching
- Sorting
- Binary Search Trees
- Data Structures
- Review:
  - Studio 5 in-class sheet
  - Studio 6 in-class sheet

**FREE LABOR**

**FREE LABOR EVERY WHERE**

**Searching**

# Searching

## Linear Search

- Intuitively:
  - If we want to search for the element  $x$  in a list
  - Go through every element in the list
  - Check if it matches
- But what if the item we want is the last one?
  - Then it's a waste of resources to go thru every element before that



# Searching

## Binary Search

- Remove redundant stuff
- Analogy:
  - I have a 500 page book, I want to find something around page 300
  - I can discard the first half of the book (pages 1 - 250)
  - I can then discard pages 376 - 500 (leaving me with 251 - 375)
  - ... continue discarding until we reach around page 300

# Searching

## Binary Search

- Notice that page numbers are in ascending order
- Criteria for binary search:
  - Items in the list must be ordered (ascending, descending)

# Searching

## Put on the Thinking Cap

- In a list of distinct numbers in ascending order, find the smallest index such that the item at that index is equal to the index itself.
  - `list_ref(xs, index) == index`
  - How do we go about finding this?
  - Answer
  - Google interview question



# Sorting

# Sorting

## Overview

- Integral part of modern algorithms
- Sometimes we choose to sort the list once so that operations later can run faster, improving overall runtime
  - “Pre-processing”: just some insights into CS2040S
- Sorts:
  - Insertion sort, selection sort, quick sort, merge sort

# Sorting

## Insertion Sort

- Select the first element in the list ( $x$ )
- Recursively sort the tail of the list ( $ys$ )
- Insert  $x$  into the sorted  $ys$
- Wishful thinking!

# Sorting

## Insertion Sort

- Complexity:
  - time:  $O(n^2)$
  - space:  $O(n)$

# Sorting

## Selection Sort

- Select the smallest element (or largest)
- Place it at the front of the list (or end) (its correct location)
- Recursively sort the rest of the list
- More wishful thinking...

# Sorting

## Selection Sort

- Complexity:
  - Time:  $O(n^2)$
  - Space:  $O(n)$

# Sorting

## Quick Sort

- Take a pivot
- Partition into two sublists
- Recursively sort the two sublists
- Join (using append): left sublist + pivot + right sublist
- And even more wishful thinking!
  - Divide and conquer algorithm



# Sorting

## Quick Sort

- Complexity:
  - Time: depends!
    - If we select good “enough” pivots:  $O(n \log n)$
    - If we select bad pivots:  $O(n^2)$
  - Space:  $O(n)$

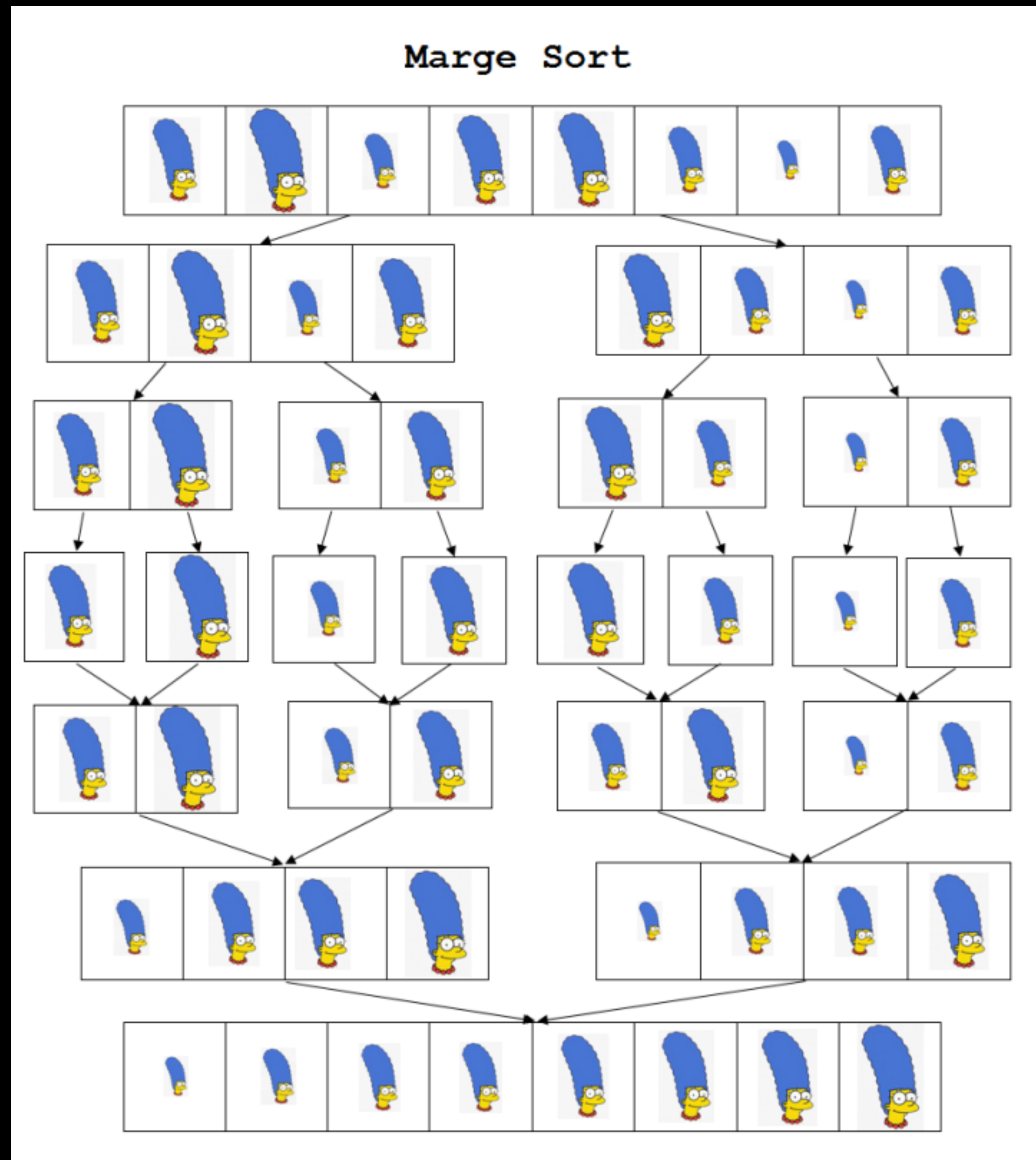
# Sorting

## Quick Sort

- Exercise: what are bad pivots?
  - Good pivots: roughly equal number of elements in each partition
  - Bad pivots: every element belongs to one partition
    - One partition of  $n$  elements, the other has no elements
    - Then this is just insertion sort!

# Sorting

## Merge Sort



# Sorting

## Merge Sort

- Split the list into two
- Sort the two lists separately
- Merge the two lists into one
- Lastly, more wishful thinking! :D

# Sorting

## Sorting Complexities

- Best we can do for the above mentioned:  $O(n \log n)$
- Can we do better?





mathew ✓

@mathew@mastodon.social

I came up with a single pass  $O(n)$  sort algorithm I call StalinSort. You iterate down the list of elements checking if they're in order. Any element which is out of order is eliminated. At the end you have a sorted list.

2018/10/26 04:20:16

++ [hades0299](#) 3.8k points · 11 months ago

-- An idea for optimization to  $O(1)$ :

Delete the whole list, as an empty list is sorted.

[Give Award](#) [Share](#) [Report](#) [Save](#)



# Sorting

## Sorting Complexities

- Best we can do:  $O(n \log n)$
- Can we do better?
  - All comparison based sorting algorithms have a lower bound of  $\Omega(n \log n)$ 
    - aka at least  $\log(n)$  comparisons are made
  - There are sorting algorithms that can do better
    - Counting sort: trades space for faster runtimes

# Sorting

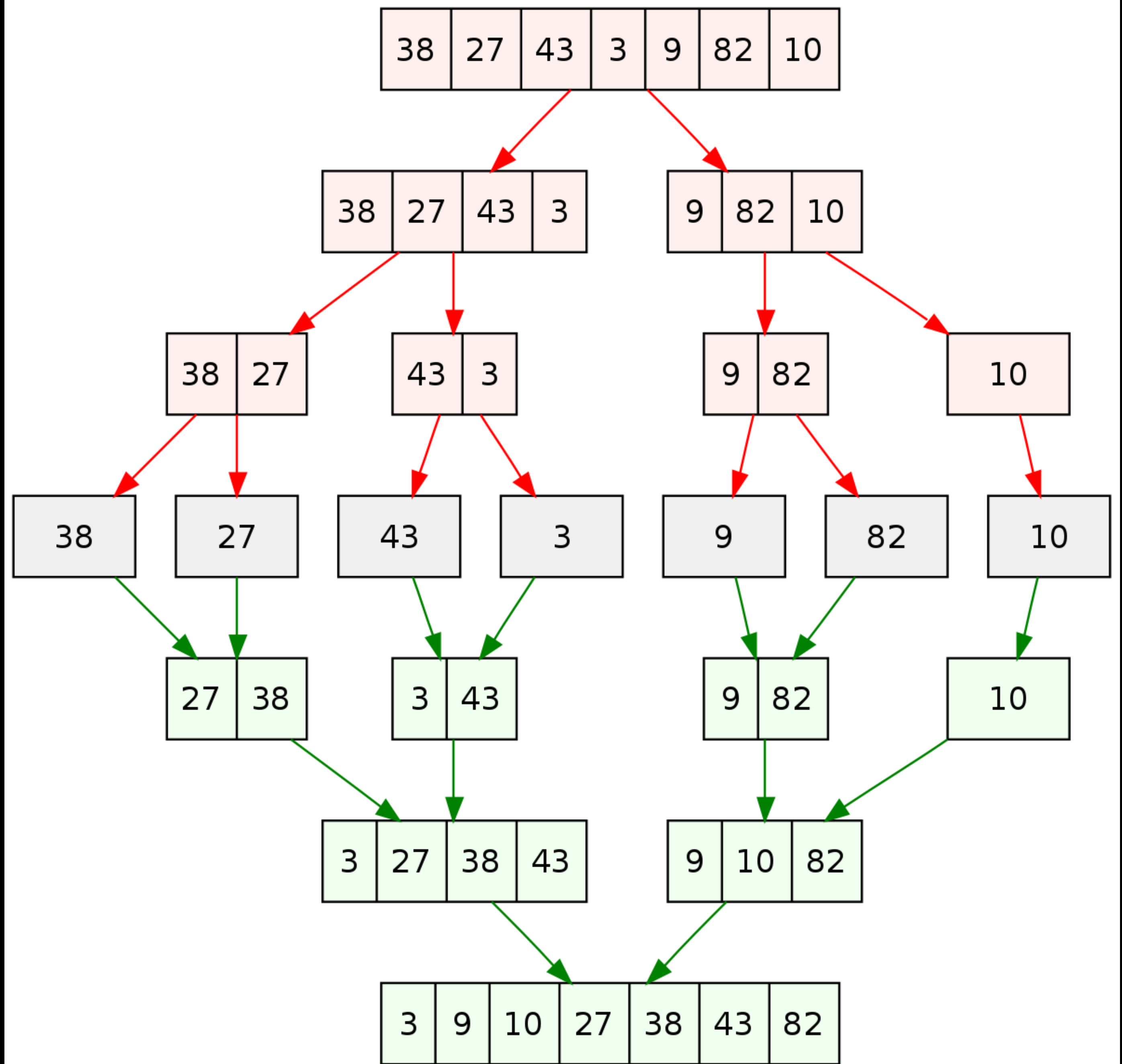
## Common Mistake

- During divide and conquer (merge sort, quick sort)
  - Visually we think of both partitions / lists being sorted simultaneously
  - Actually:
    - Computers are sequential machines
    - One partition is fully processed first before going on to the other

# Sorting

## Common Mistake

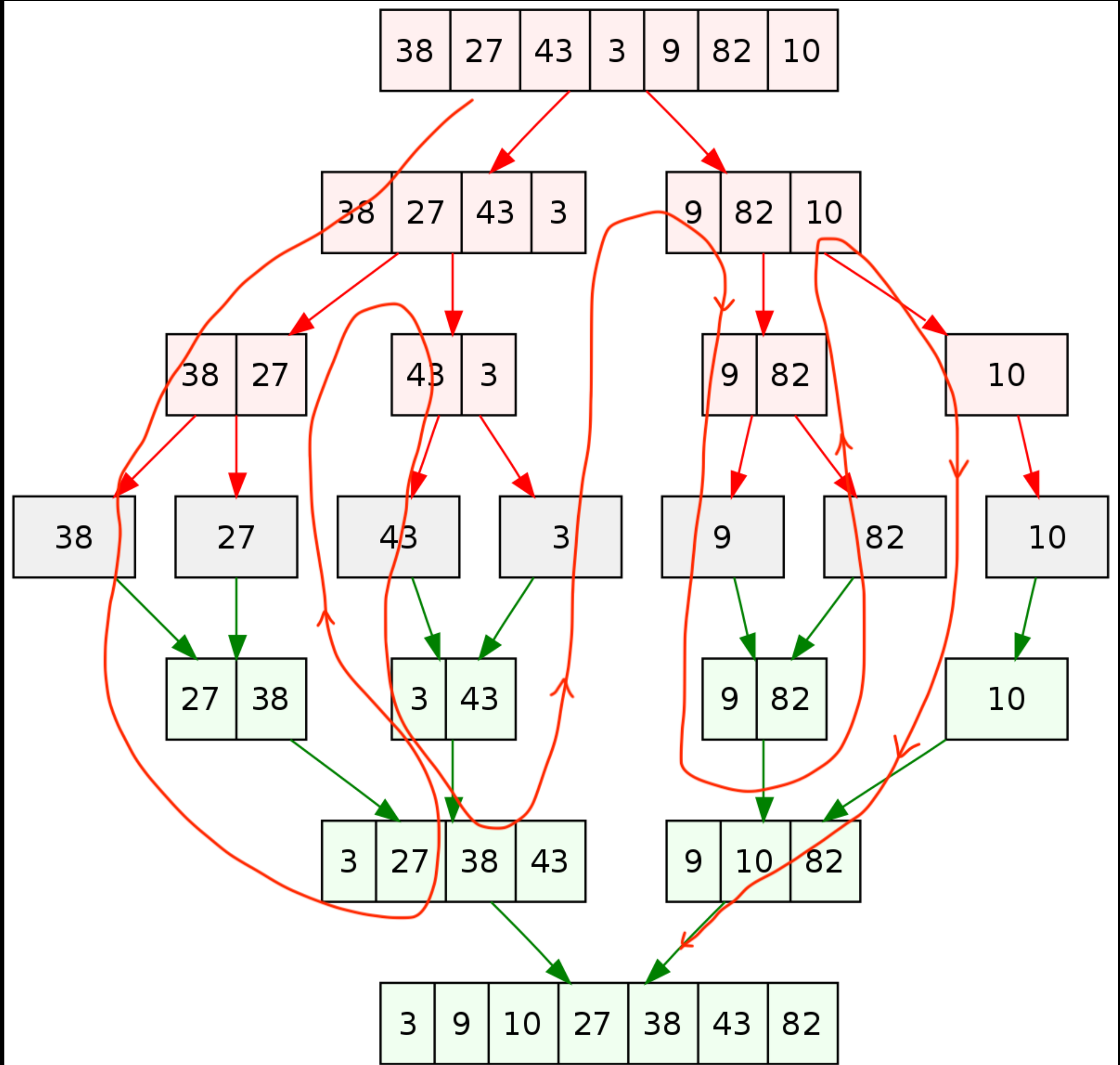
- “Simultaneously”



# Sorting

## Common Mistake

- In reality:
  - follow red arrow



# Binary Search Trees

# Binary Search Tree

## Definition

- A binary search tree is an abstraction of binary search
  - Can see the similarity?
  - Removes redundant stuff
- A binary search tree is the empty tree, or it has an entry, a left branch and a right branch (both also binary search trees).
- All entries in the left branch are smaller than the entry, and all entries in the right branch are larger than the entry.
  - By definition: no duplicates!

# Binary Search Tree

## Definition

- Common mistake:
  - Some only compare the values with the immediate children
- Correct:
  - Need to compare with the ENTIRE left and right sub-tree
  - Root should be larger than all the elements in the left sub-tree
  - Root should be smaller than all the elements in the right sub-tree

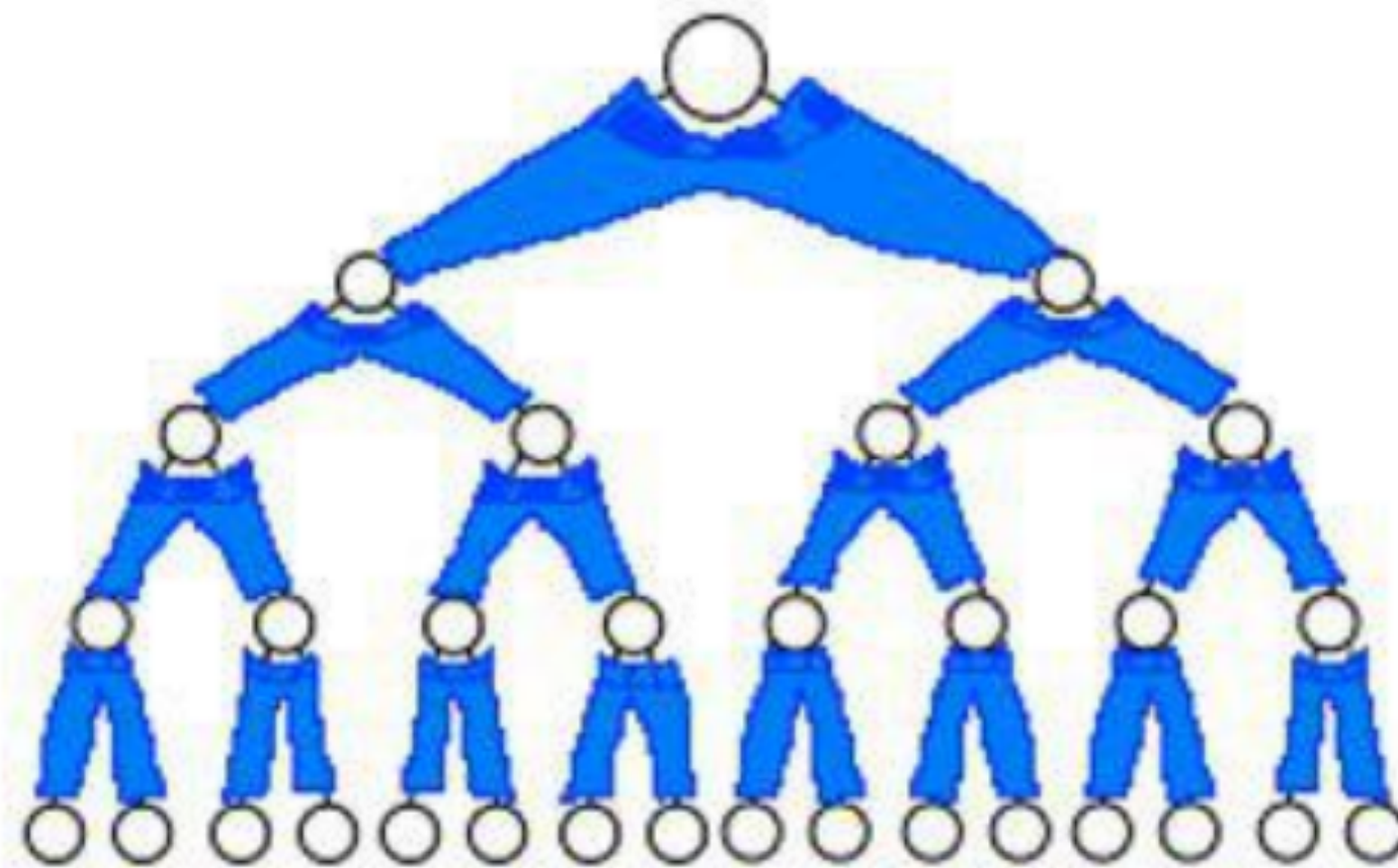


# Binary Search Tree

## Definition

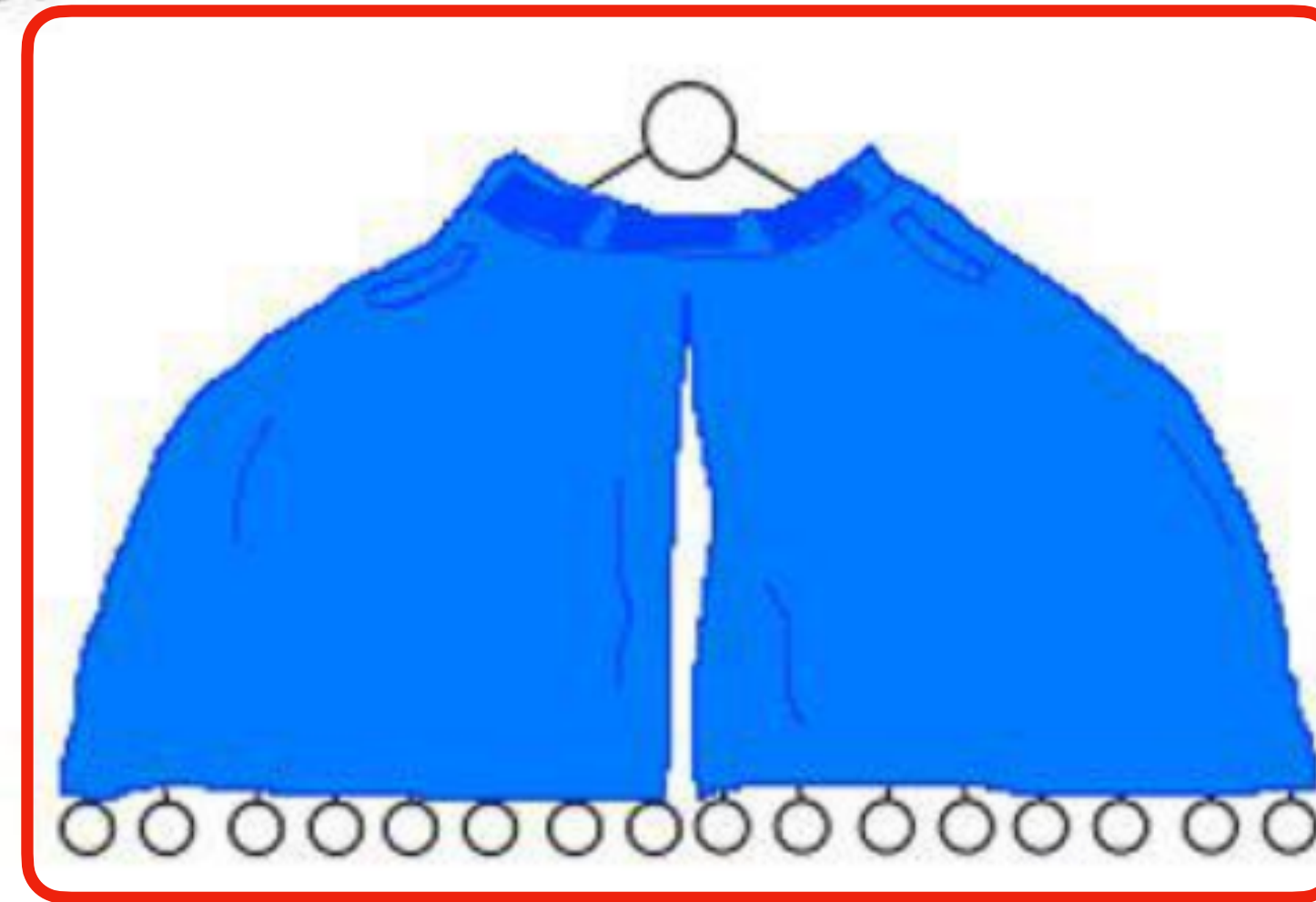
If a binary tree wore pants would he wear them

like this



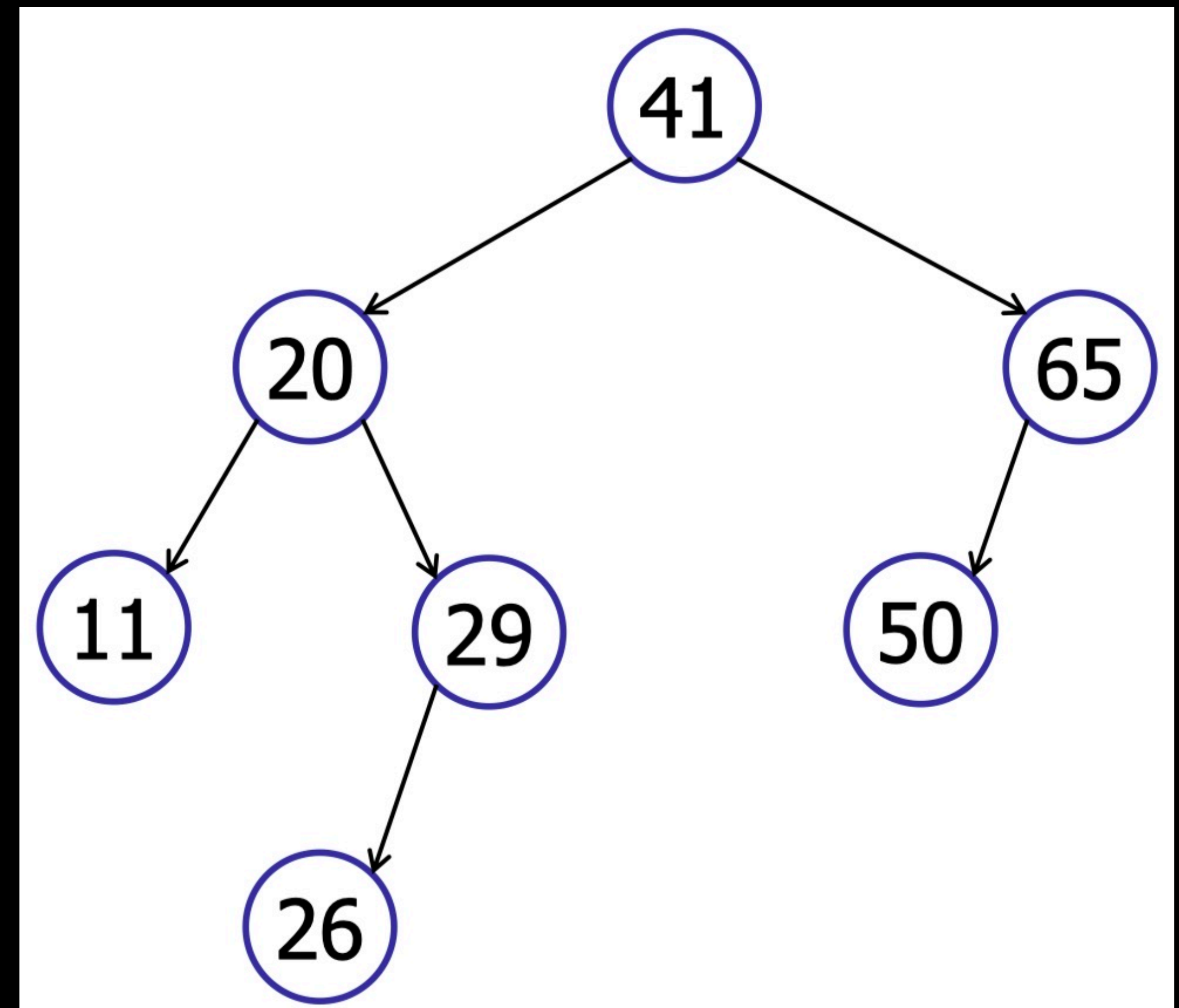
or

like this?



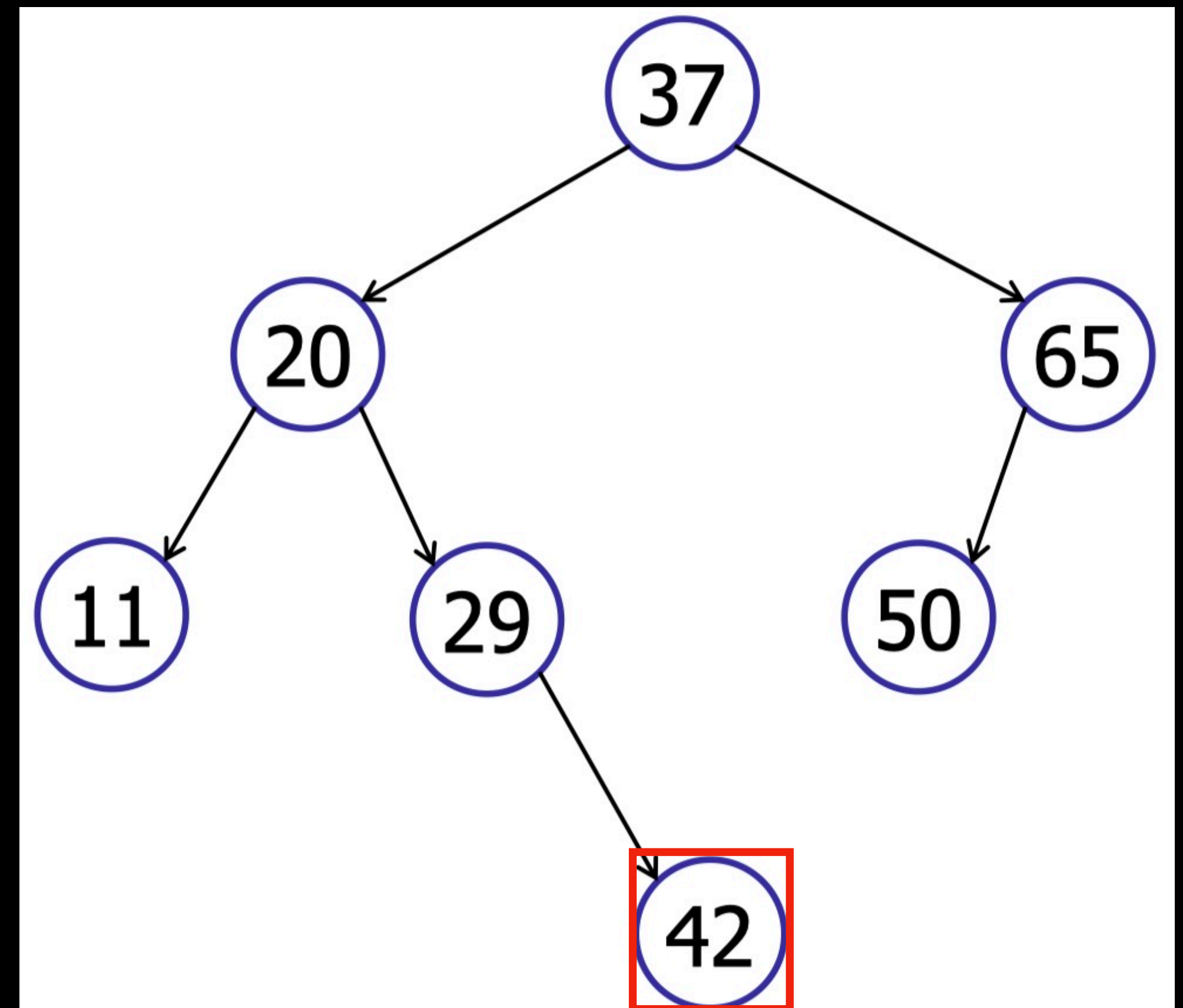
# Binary Search Tree Quiz

- Is this a BST?
  - Yes



# Binary Search Tree Quiz

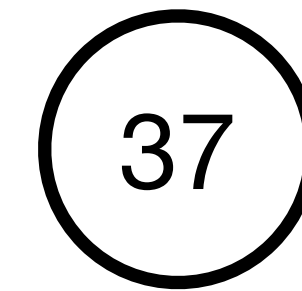
- Is this a BST?
  - No



# Binary Search Tree

## Quiz

- Is this a BST?
  - Yes



# Binary Search Tree

## Definition

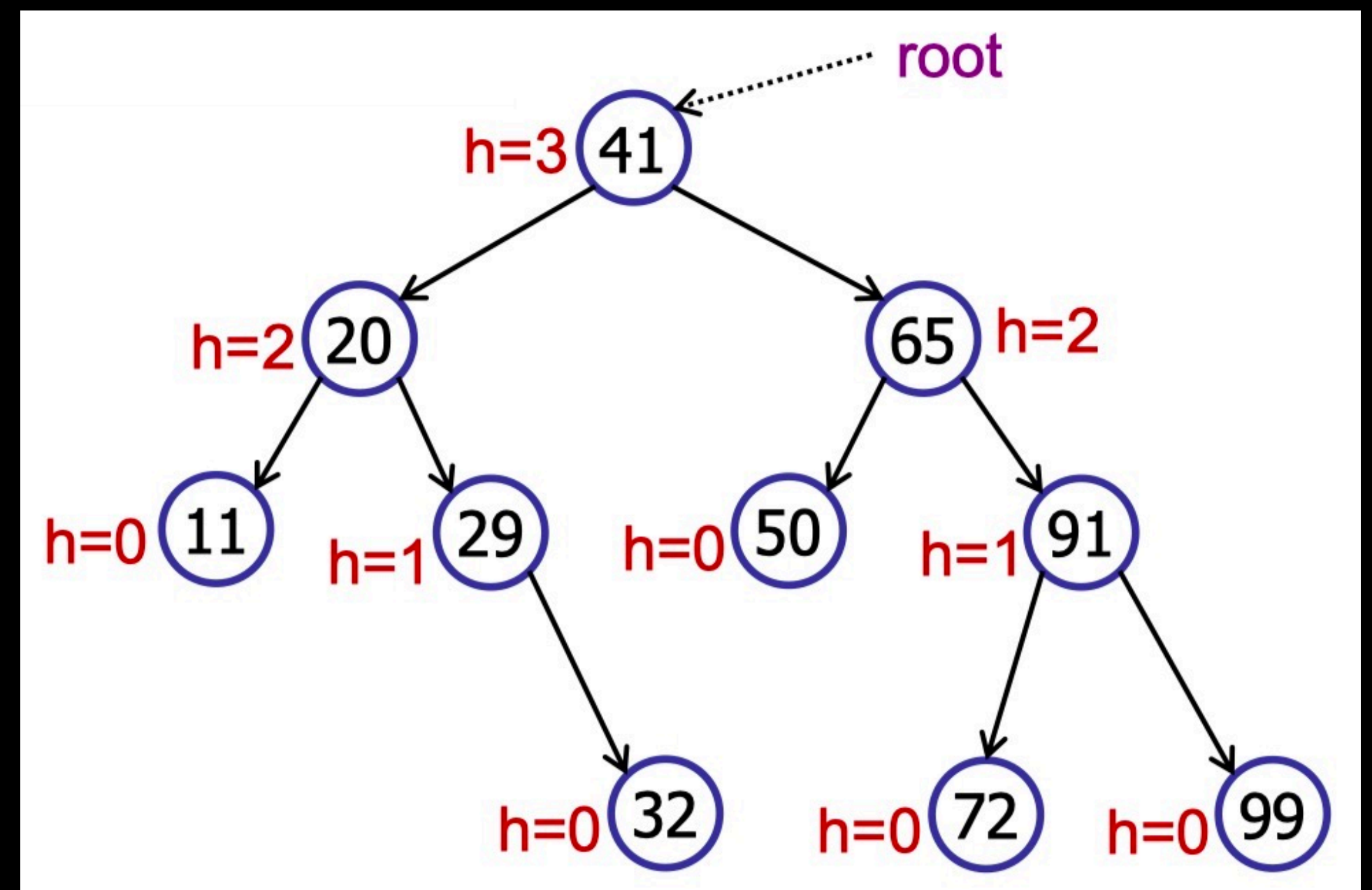
- Hack:
  - If you “flatten” a tree to get a sequence of numbers
  - If the sequence is in ascending order, then it is a BST
  - Else, it's not



# Binary Search Tree

## Height

- Height of a tree
  - Number of edges (pointers) on the longest path from root to leaf



# Binary Search Tree

## Purpose

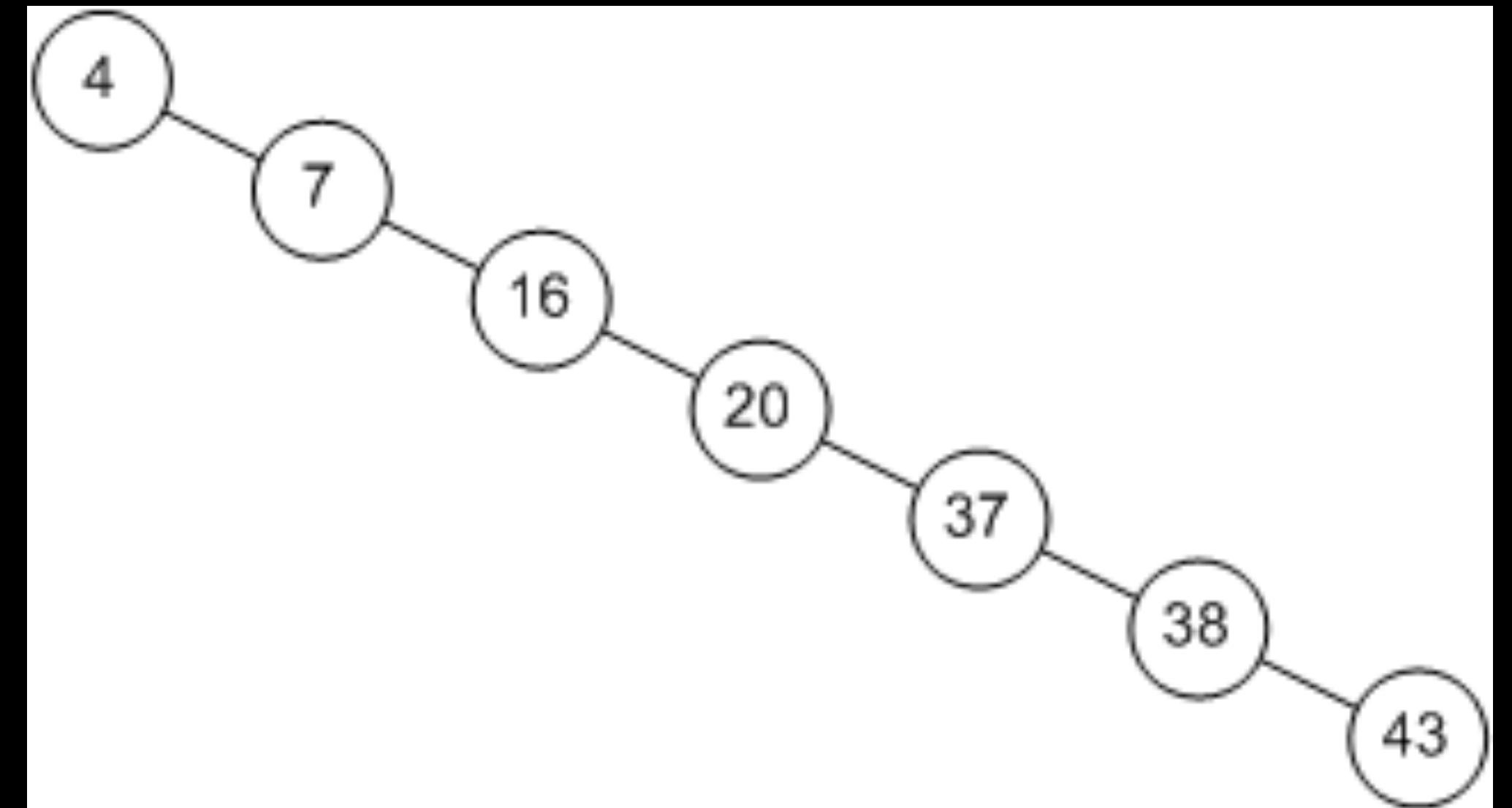
- Linear search runs in  $O(n)$  time
- If we have a binary tree:
  - We can search for the item in  $O(\log n)$  time
  - Why? Recall binary search and relationship to BSTs
- Best case:  $O(\log n)$
- Worse case:  $O(n)$ 
  - How?



# Binary Search Tree

## Purpose

- Worst case  $O(n)$ :
  - Imagine if we only add to the right sub-tree
  - It's just a list...



# Binary Search Tree

## Traversal

- Pre-order
- In-order
- Post-order

# Binary Search Tree

## Traversal

- Pre-order:
  - Visit root
  - Visit left sub-tree recursively
  - Visit right sub-tree recursively

# Binary Search Tree

## Traversal

- In-order:
  - Visit left sub-tree recursively
  - Visit root
  - Visit right sub-tree recursively

# Binary Search Tree

## Traversal

- In-order:
  - Cool fact about in-order traversal:
    - Visits items in sorted order
    - Refer to Mission Search and Rescue

# Binary Search Tree

## Traversal

- Post-order:
  - Visit left sub-tree recursively
  - Visit right sub-tree recursively
  - Visit root

# Data Structures

# Data Structures

## A Recap

- `const a = list(1, 2, 3);`
- `a === insertion_sort(a) // ?`

- Answer: false

```
function insert(x, xs) {  
    return is_null(xs) ? list(x) : x <= head(xs)  
        ? pair(x,xs)  
        : pair(head(xs), insert(x, tail(xs)));  
}  
  
function insertion_sort(xs) {  
    return is_null(xs)  
        ? xs  
        : insert(head(xs), insertion_sort(tail(xs)));  
}
```



# Data Structures

## A Recap

- Why?
  - When we call `insert`, we created new pairs
  - New list created
- Not sorted “in-place”

```
function insert(x, xs) {  
    return is_null(xs) ? list(x) : x <= head(xs)  
        ? pair(x,xs)  
        : pair(head(xs), insert(x, tail(xs)));  
}
```

**Any questions?**