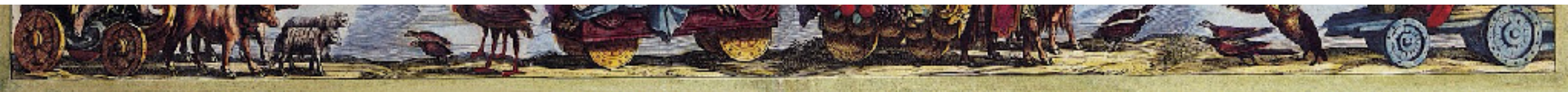
SET IMPLEMENTATIONS



SET UP YOUR ONE-ON-ONE MEETINGS!

Hopefully you've all set up a one-on-one meeting. I presume that I've met with some of y'all (I'm preparing these lecture notes prior to releasing one-on-one scheduling tools)

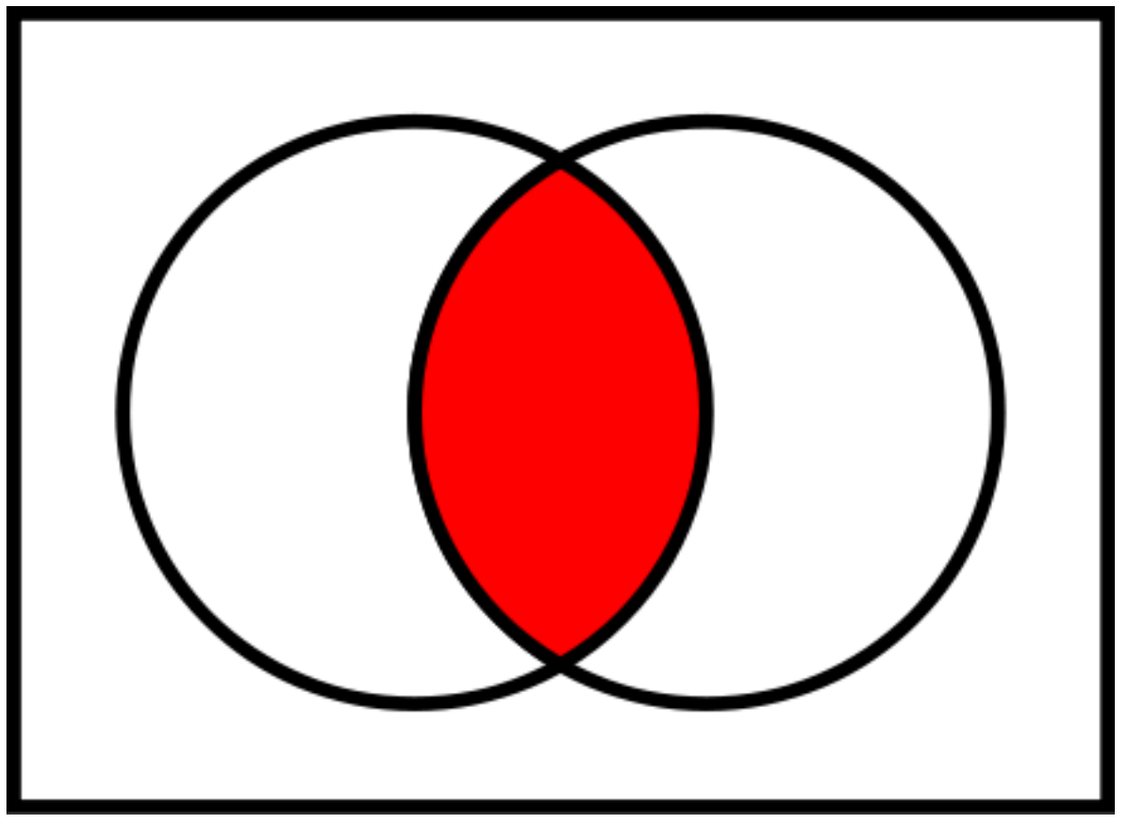
I'm not going to pretend that things are "normal", and neither should you. Wherever you are with things, that's OK.

One-on-one meetings can help add to a sense of normalcy even though it isn't

INTERSECTION -SET

INTERSECTION-SET

- We need to come up with a strategy for implementing intersection-set
- intersection-set returns a set where the elements are part of both input sets



RECURSIVE STRATEGY

- Assume that we've already solved the problem of providing the intersection of *set2* and the *cdr* of *set 1*
  - This is entirely reasonable. Why?
  - If we're trying to say we can provide an intersection of two lists (which are each assumed to be sets), we can say two more things
    - Each list has a *car* and a *cdr*
    - The *cdr* of a list is itself a list (and in this case a set)
- Therefore we can make the assumption for this strategy

RECURSIVE STRATEGY

- Therefore, assuming we can provide an intersection of the *cdr* of *set 1* and *set 2*, then all that remains is to determine if we should add the *car* of *set 1* to the intersection.
  - We solve for the simplest case—the intersection of a single element and a full set
  - Then this simple solution is propagated through the use of recursion
  - *trust the recursion*
    - (I think I've said this before)

INTERSECTION-SET

```
(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2) '())
        ((element-of-set? (car set1) set2)
          (cons (car set1)
                (intersection-set (cdr set1) set2)))
        (else (intersection-set (cdr set1) set)))))


; tip: break this apart by parenthesis
```

ALGORITHM EFFICIENCY

- Theoretically, one should be concerned about the efficiency of one's algorithms
  - This is very difficult to do on modern computer systems
  - (being concerned, not being efficient)
- There's always CPU and memory to spare right?
  - Embedded systems
  - Threads / Distributed computing
  - Tight loops
- So, yes it matters… sometimes

HOW EFFICIENT IS INTERSECTION-SET?

- The efficiency of *intersection-set* is *entirely* dependent upon the efficiency of *element-of-set?*
  - Each iteration through *set 1* requires that we potentially iterate through *all members* of *set 2* using *element-of-set?*
    - The worst case is that the element does not appear in *set 2*
  - Therefore, if the set has *n* elements, *element-of-set?* might take up to *n* steps $\Theta(n)$
  - *adjoin-set*, which also uses *element-of-set?* , also grows in the same way. $\Theta(n)$
  - *intersection-set* grows with the sizes of both sets, or $\Theta(n^2)$
  - So will *union-set*

A PERSONAL CHALLENGE

- Challenge yourself to this: *how can I implement union-set*?
  - Note: this is not *homework*, but should be considered an important part of this class session
  - Work that brain!
  - This will have much in common with *intersection-set*
  - Pause this video and work this out. Resume when you're done.

SET UNION


VRATISSIMA TOTIVS TERRARVM ORBIS TABVL
GROENLAN-
DIA.
MAR DEL
NORT.
AMERICA
OCE-
ANVS
ETHIOPICVS

UNION-SET

```
(define (union-set A B)
   (cond ((null? B) A)
         ((element-of-set? (car B) A)
          (union-set A (cdr B)))
         (else (cons (car B)
                     (union-set A (cdr B))))))
```

SETS AS ORDERED LISTS

FIXING INEFFICIENCY

- The cost of intersection-set and union-set are both pretty high
- Since the cost comes down to the overhead of element-of-set?, improving the efficiency of this routine will improve them both
- This is an *important* lesson of computer engineering; you don't have to have a great solution at first to work towards a great solution overall
- Improvements are incremental

ORDERING THE LIST

- To keep the discussion simple, we'll go back to working with sets of only numbers (so we can use less than and greater than)
- Instead of representing a set with a list in *any* order, we'll instead represent it with a list in numeric order
- With this simple modification, we no longer have to scan the entire set to determine if an element exists within that set

ELEMENT-OF-SET (ORDERED)

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((= x (car set)) true)
        ((< x (car set)) false)
        (else (element-of-set? x (cdr set)))))
```

```
; remember this was
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set)))))
```

HOW MUCH MORE EFFICIENT IS THIS?

- For an element not appearing in the set, or the largest element, there is no improvement
- For elements that are small in value, the improvement is huge
- On the average, we should expect to examine half the list (number of steps would be *n/2)*
- On the average, this is a factor of 2

INTERSECTION-SET IMPROVEMENT

- The improvement really comes into its own in intersection-set
- It requires a complete rewrite of intersection-set to gain this improvement

INTERSECTION-SET (ORDERED)

```
(define (intersection-set set1 set2)
  (if (or (null? set1) (null? set2)) '()
      (let ((x1 (car set1)) (x2 (car set2)))
        (cond ((= x1 x2) (cons x1 (intersection-set
                                   (cdr set1)
                                   (cdr set2))))
              ((< x1 x2) (intersection-set
                          (cdr set1) set2))
              ((< x2 x1) (intersection-set
                          set1 (cdr set2)))))))
```

IMPROVEMENTS

- With this, the number of steps is at most the sum of sizes of set1 and set2, rather than the product of the sizes
- This is $\Theta(n)$ instead of $\Theta(n^2)$!
- This is a considerable speedup, even for small sets

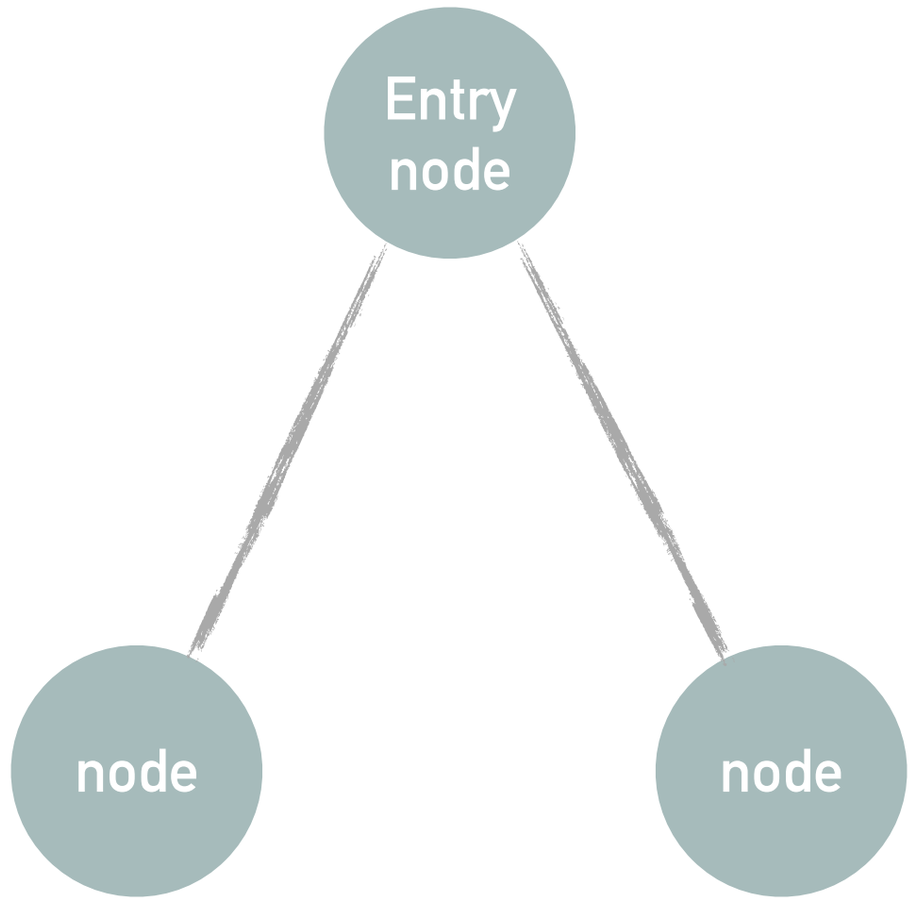
PONDER THIS

- What needs to change to *adjoin-set*?
- What needs to change to *union-set*?

- Pause the video, and resume once you've thought through these problems
- No answers will be provided here for these.

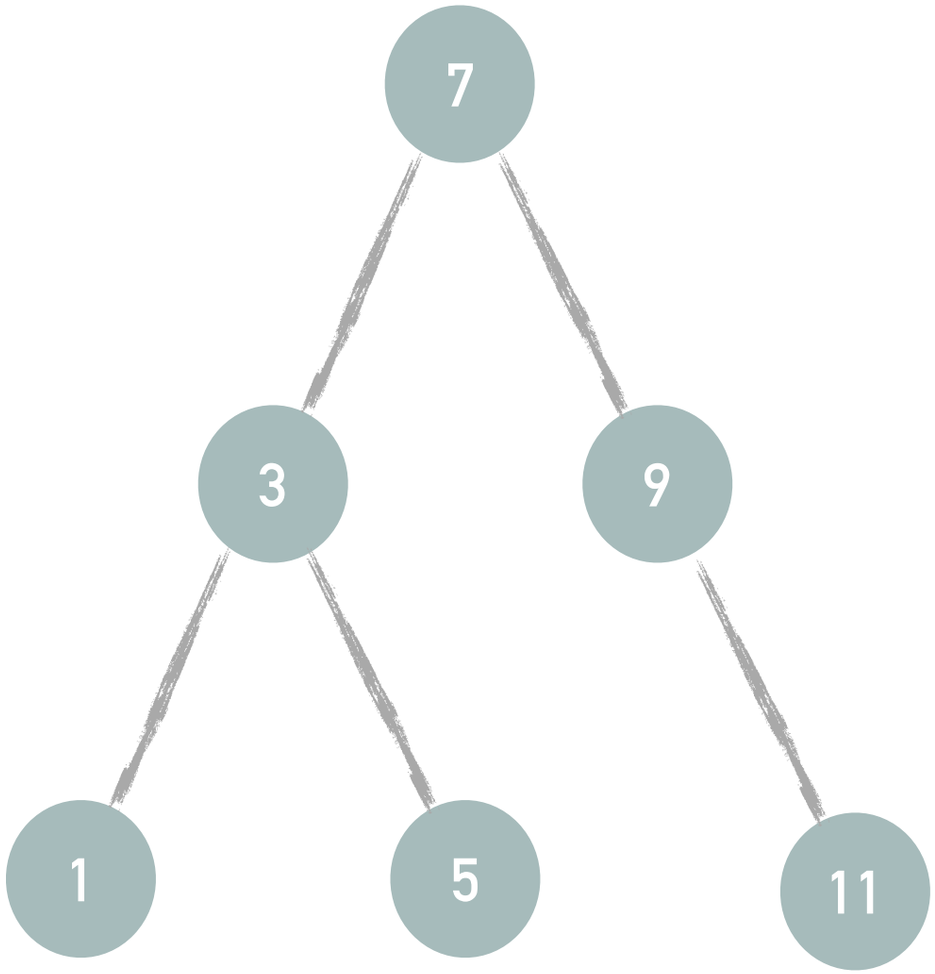
SETS AS BINARY TREES

WHAT IS A BINARY TREE?

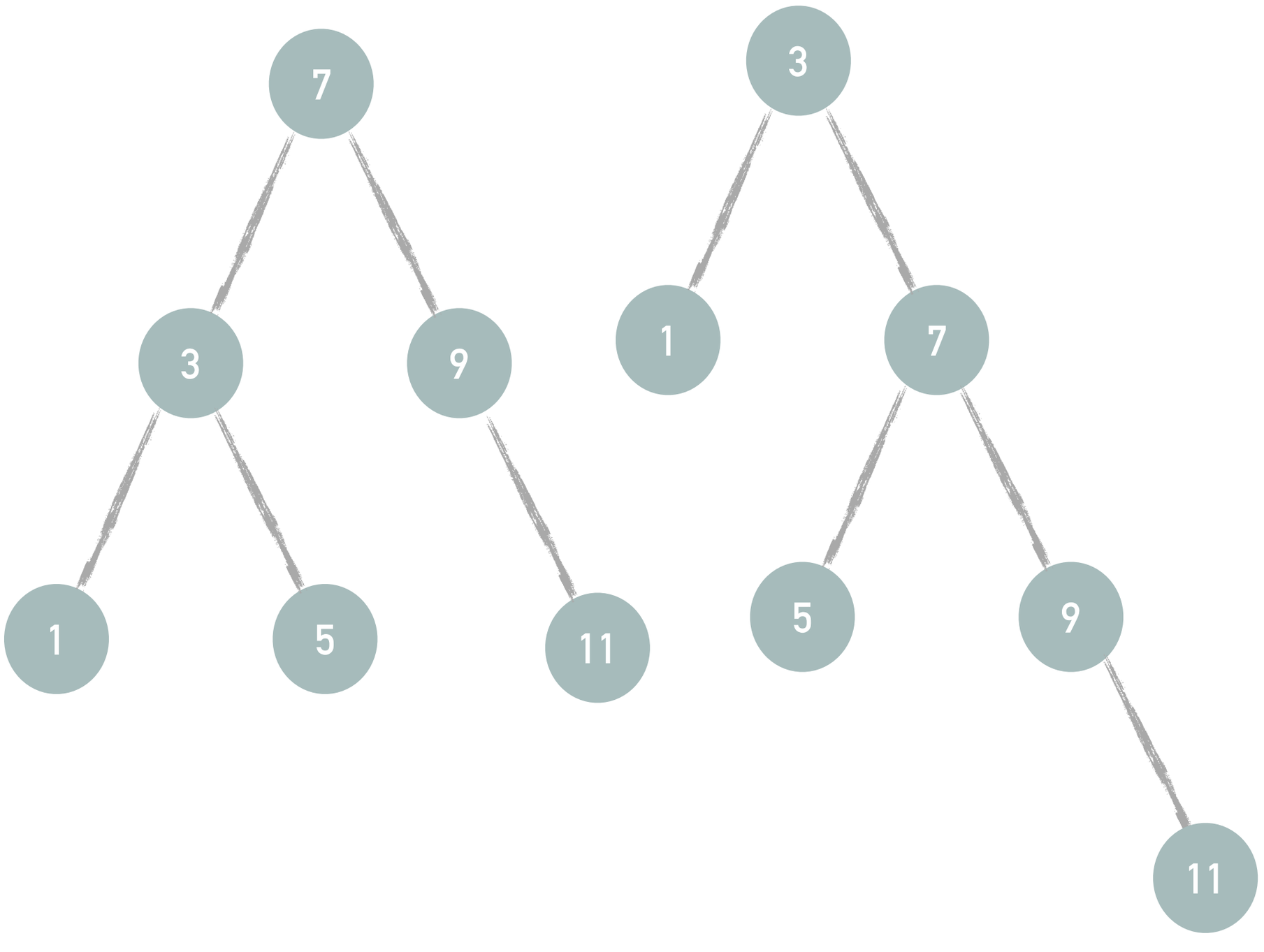


Simple rule: The value at the left must be smaller than its entry node, the value at the right must be greater than its entry node

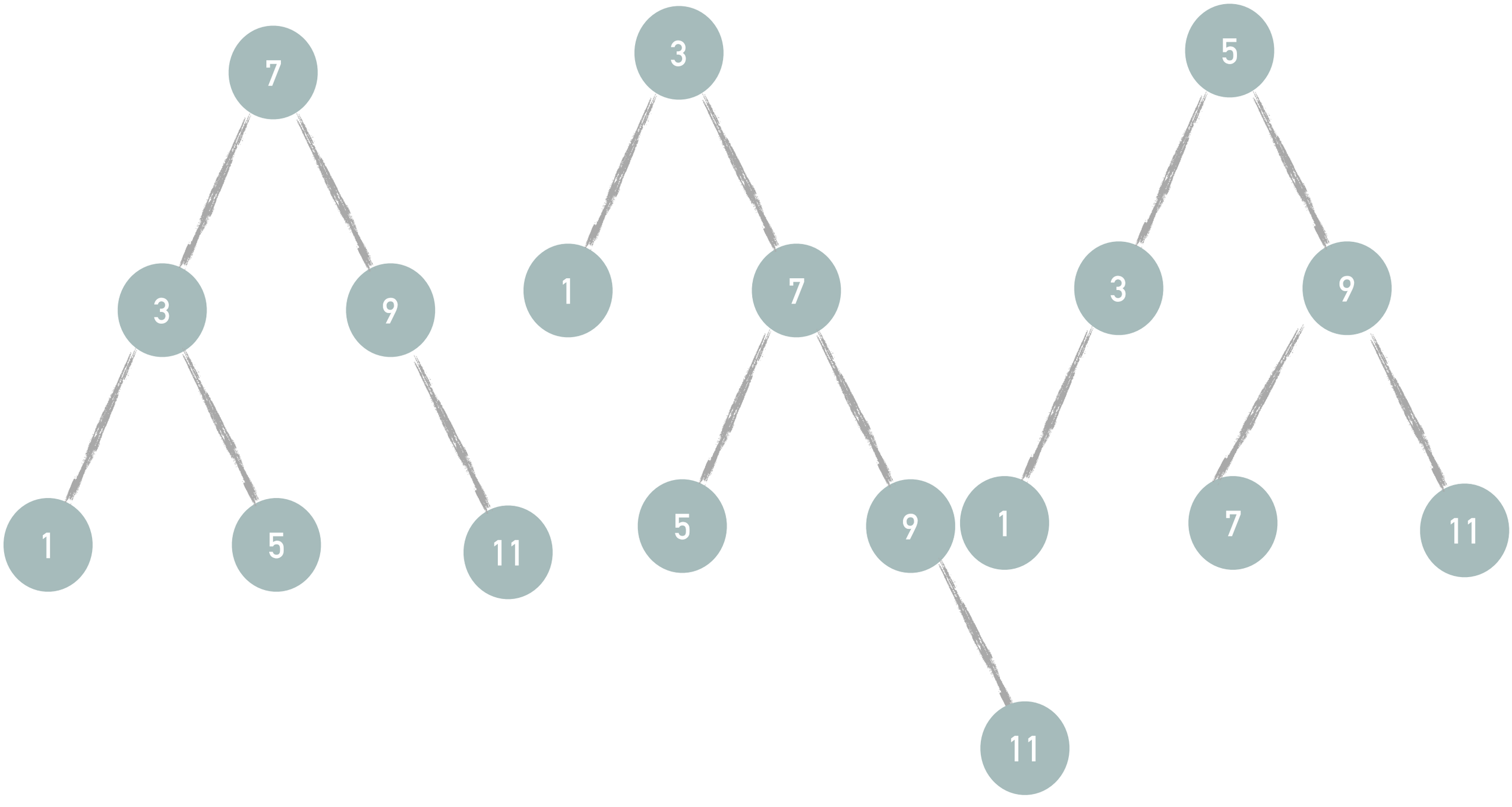
TREES FOR ODD NUMBERS (1, 3, 5, 7, 9, 11)



TREES FOR ODD NUMBERS (1, 3, 5, 7, 9, 11)



TREES FOR ODD NUMBERS (1, 3, 5, 7, 9, 11)



ADVANTAGES OF BINARY TREES

- For large sets, the savings can be tremendous
  - $\Theta(log\ n)$
- We can see why: The number or steps has become very small indeed to move across the elements in a tree
- This is most true when a tree is *balanced*; that is to say when each node has (approximately) the same number of nodes on each side of the tree

DISADVANTAGES OF BINARY TREES

- List retrieval is very quick, as we've seen
- Adding elements to a list is *not* quick
- Rebalancing is the key!

SETS AND INFORMATION RETRIEVAL

CHOOSING A SET

- Obviously the mechanism for implementing a set matters greatly for efficiency
- The techniques involved so far with generating sets plays over and over again when creating applications that do data retrieval

SETS FOR DATABASES

- When implementing a typical database, we use *keys* that are unique per entity
- We then implement a *lookup* for the database that inspects the presence and location of a given *key*
- The choice of *key*s can determine the speed of lookup
- Imagine if we build a database using the *key* of a random ID, but then want to search by birthdate

CHANGING IMPLEMENTATION

- One of the reason we emphasize (or at the very least, that *I* emphasize) software contracts is that we're able to quickly implement a system, and then update portions as it becomes important to improve performance

CONSIDERATIONS

- Do we insert data more than we retrieve it?
  - If so, then a daily (or even less frequent) balancing may be adequate
  - If not, then a balance on insert may be worthwhile
- Instrumenting the application is essential to know how to make it work best

HOMEWORK QUIZ

SIMPLE PASS/FAIL QUIZ (SUBMIT YOUR ANSWERS, YOU PASS)

- By NEXT MONDAY 11PM
- Via EMAIL

- 1) What symbol can we use to create a bit of symbolic data in Scheme?
- 2) How can we compare two elements of symbolic data in Scheme?
- 3) What are the four elements of interfacing with sets (as we've studied them)?
- 4) What advantage does using an ordered list have over an unordered list when representing a set?
- 5) What advantage does using a binary tree have over an unordered list when representing a set?