ECS32B

Introduction to Data Structures

Sorting etc.

Lecture 21

Announcements

- Chapters 6.1-6.5, 6.10-6.13 are the background reading for Homework 5.
- The Makeup Homework will be posted Wednesday May 22nd tentatively due Thursday June 6th at 11:59pm
- Homework 6 will be posted Tue May 28th due Wednesday June 5th at 11:59pm
- The next SLAC will cover HW5, that will be followed by a combined SLAC for HW6 and the Makeup HW

Quicksort

A simple algorithm for quicksort is:

```
quicksort(unsorted_list)

    Choose one element to be the pivot

    Partition (i.e. reorder) the list so that all
      elements < the pivot are to the left of the pivot,
      and all elements >= the pivot are to the right of
      the pivot

    Apply quicksort recursively to the sublist to the
      left and the sublist to the right
```

Quicksort

Here's a version of **quicksort** in Python. You may find this version simpler to understand than the one in the textbook, but it uses more memory.

```
def quicksort(alist):
    if alist == []:
        return []
    else:
        pivot = alist[0]
        rest = alist[1:]
        return (quicksort(ltfilter(rest, pivot)) +
                [pivot] + quicksort(gtfilter(rest, pivot)))
```

Quicksort

```
def ltfilter(flist, pivot):
    result = []
    for value in flist:
        if value < pivot:
            result.append(value)
    return result


def gtfilter(flist, pivot):
    result = []
    for value in flist:
        if value >= pivot:
            result.append(value)
    return result
```

Quicksort

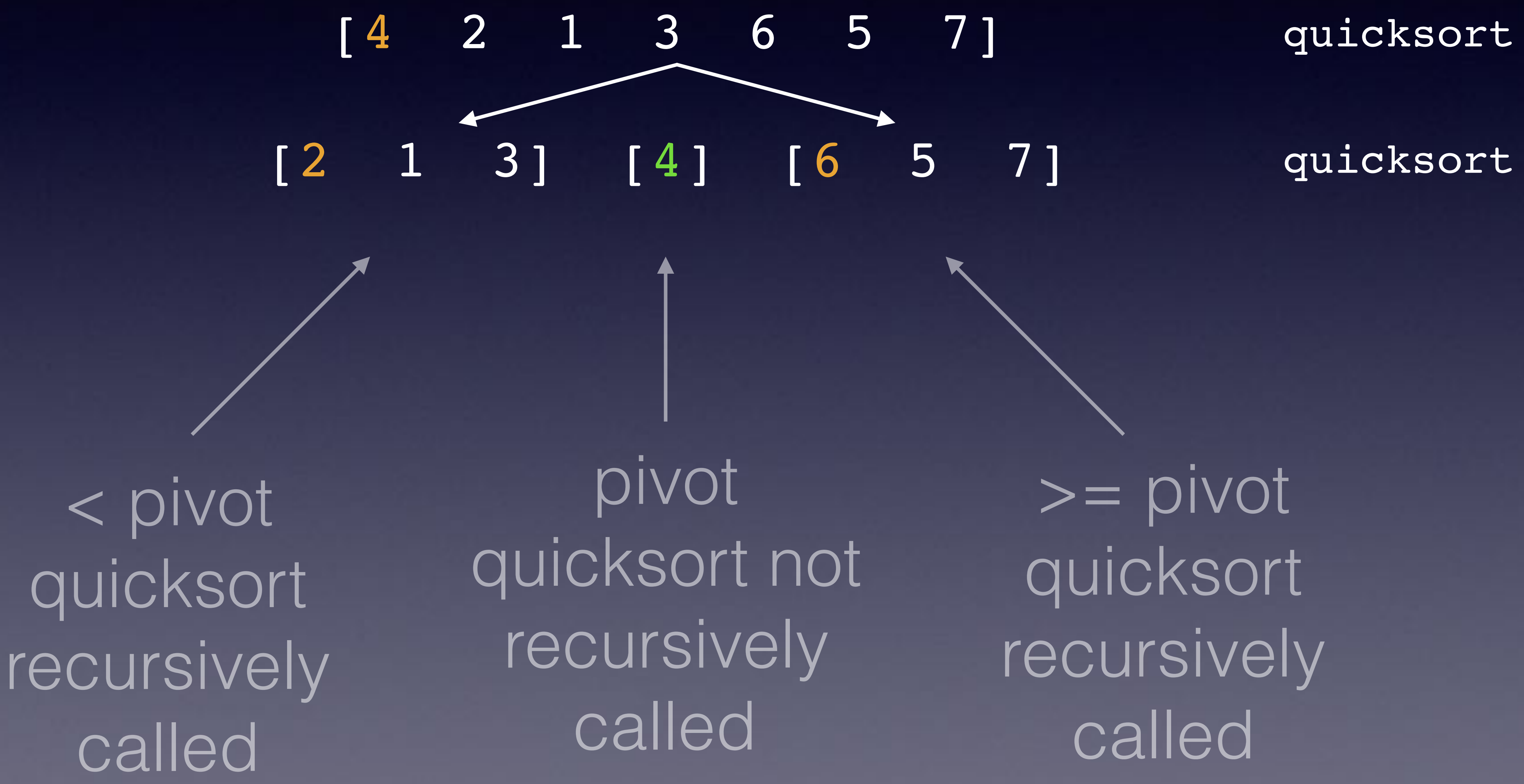
Here's how this algorithm would sort a list of integers:

```
[4  2  1  3  6  5  7]          quicksort
```

Let's call recursive quicksort on the list

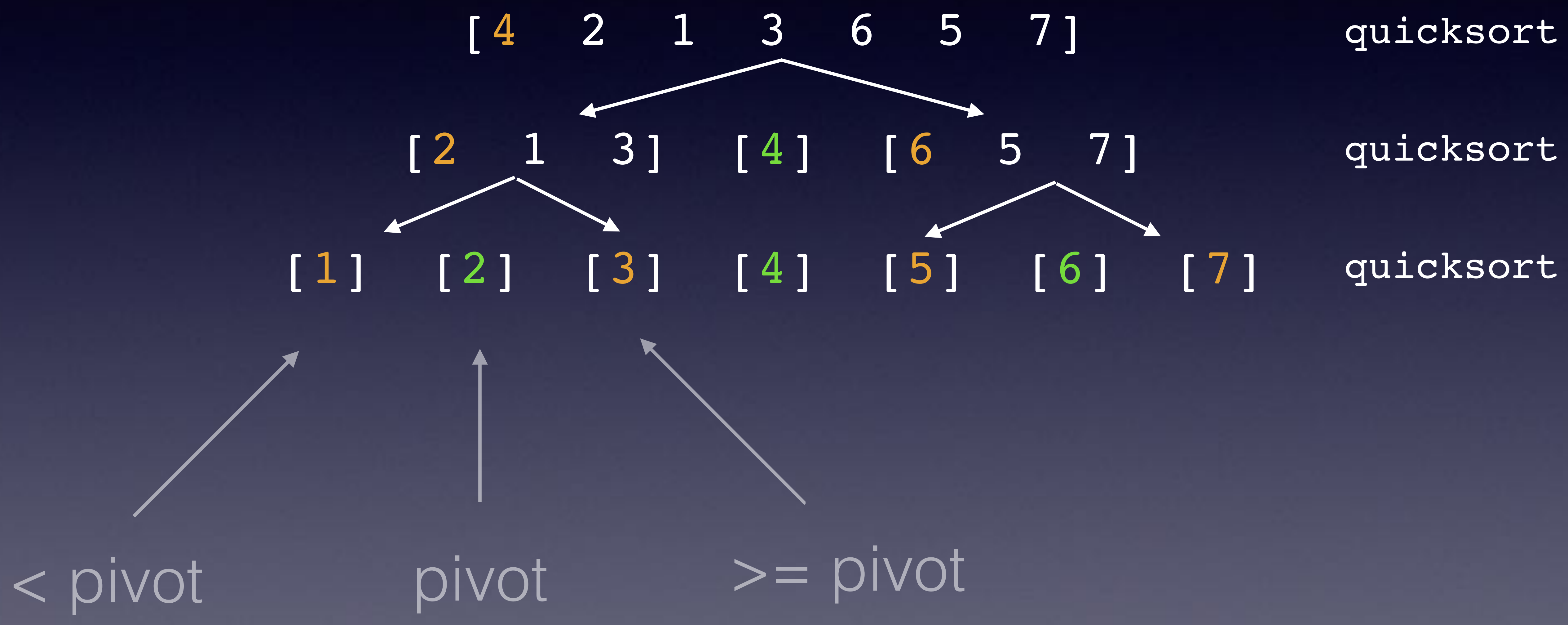
Quicksort

Here's how this algorithm would sort a list of integers:


[4 2 1 3 6 5 7]
quicksort
[2 1 3]
[4]
[6 5 7]
quicksort
< pivot quicksort recursively called
pivot quicksort not recursively called
>= pivot quicksort recursively called

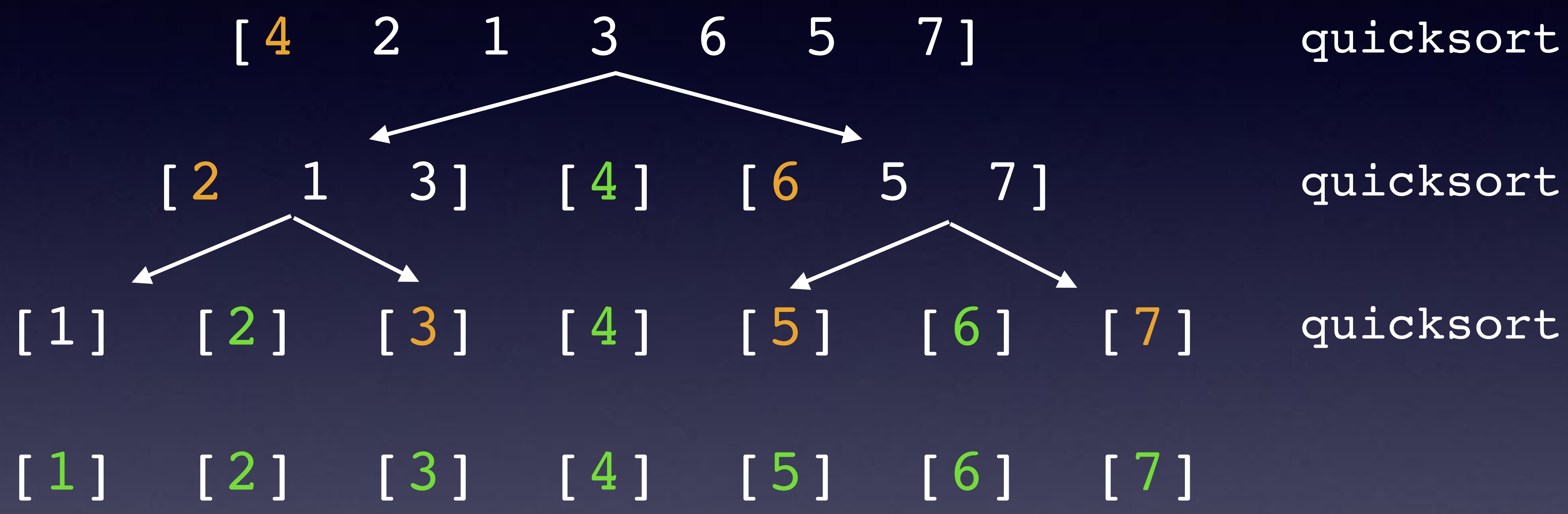
Quicksort

Here's how this algorithm would sort a list of integers:



Quicksort

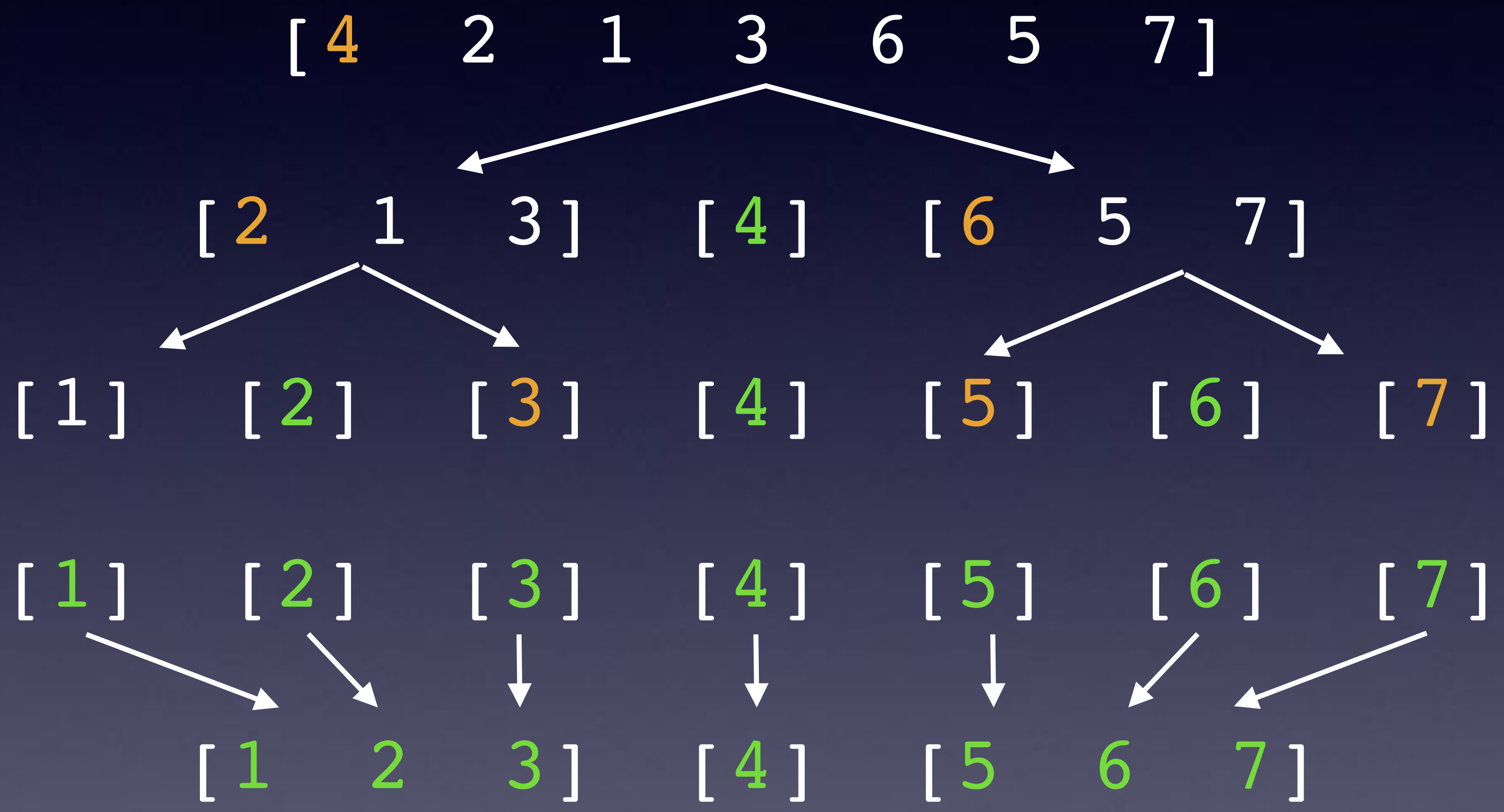
Here's how this algorithm would sort a list of integers:



At this recursion depth all values are now pivots.

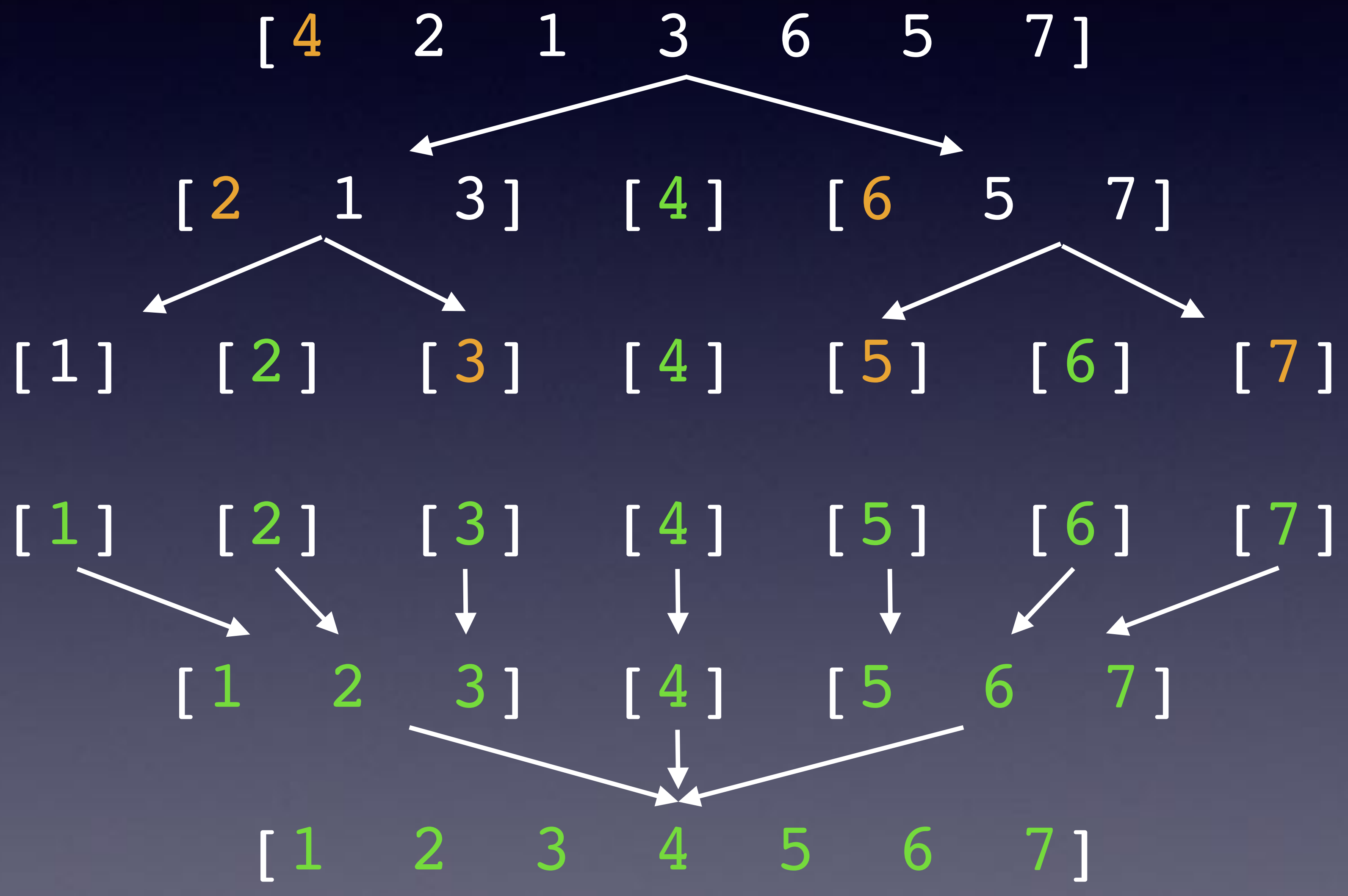
Quicksort

Here's how this algorithm would sort a list of integers:



Quicksort

Here's how this algorithm would sort a list of integers:



What's the time complexity of quicksort?

Quicksort

Here's how this algorithm would sort another list of integers:

```
[7   2   8   5   1   3   6   4]
```

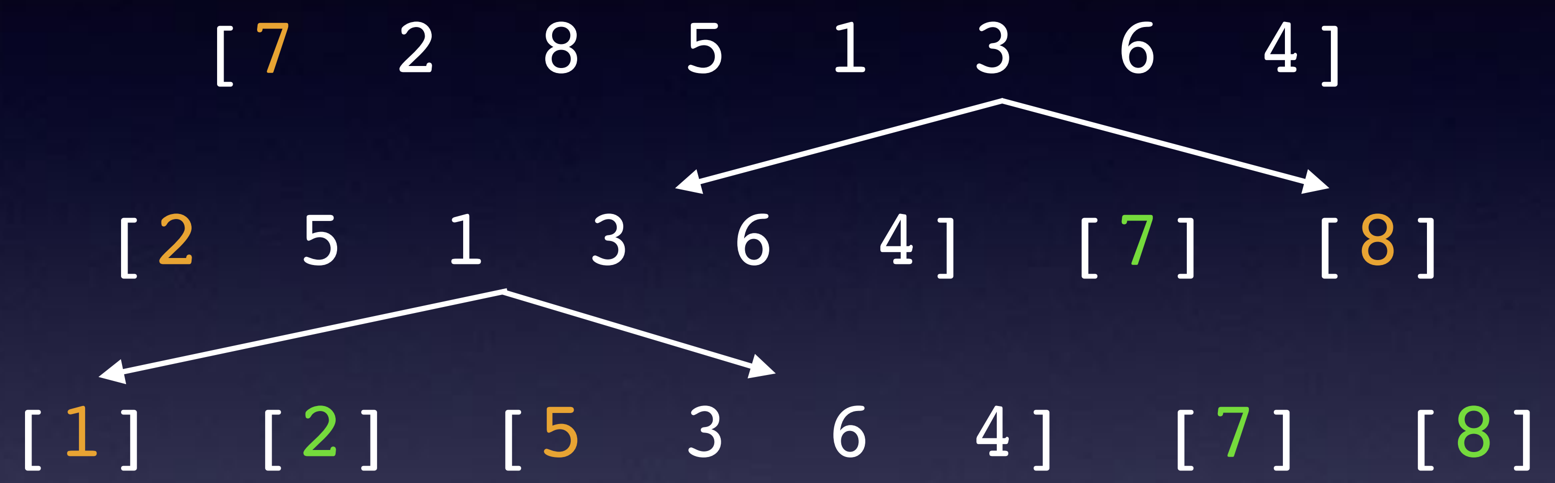
Quicksort

Here’s how this algorithm would sort another list of integers:

$$\begin{tikzcd}
 & \texttt{[7 2 8 5 1 3 6 4]} \arrow[dl] \arrow[dr] & \\
\texttt{[2 5 1 3 6 4]} & \texttt{[7]} & \texttt{[8]}
\end{tikzcd}$$

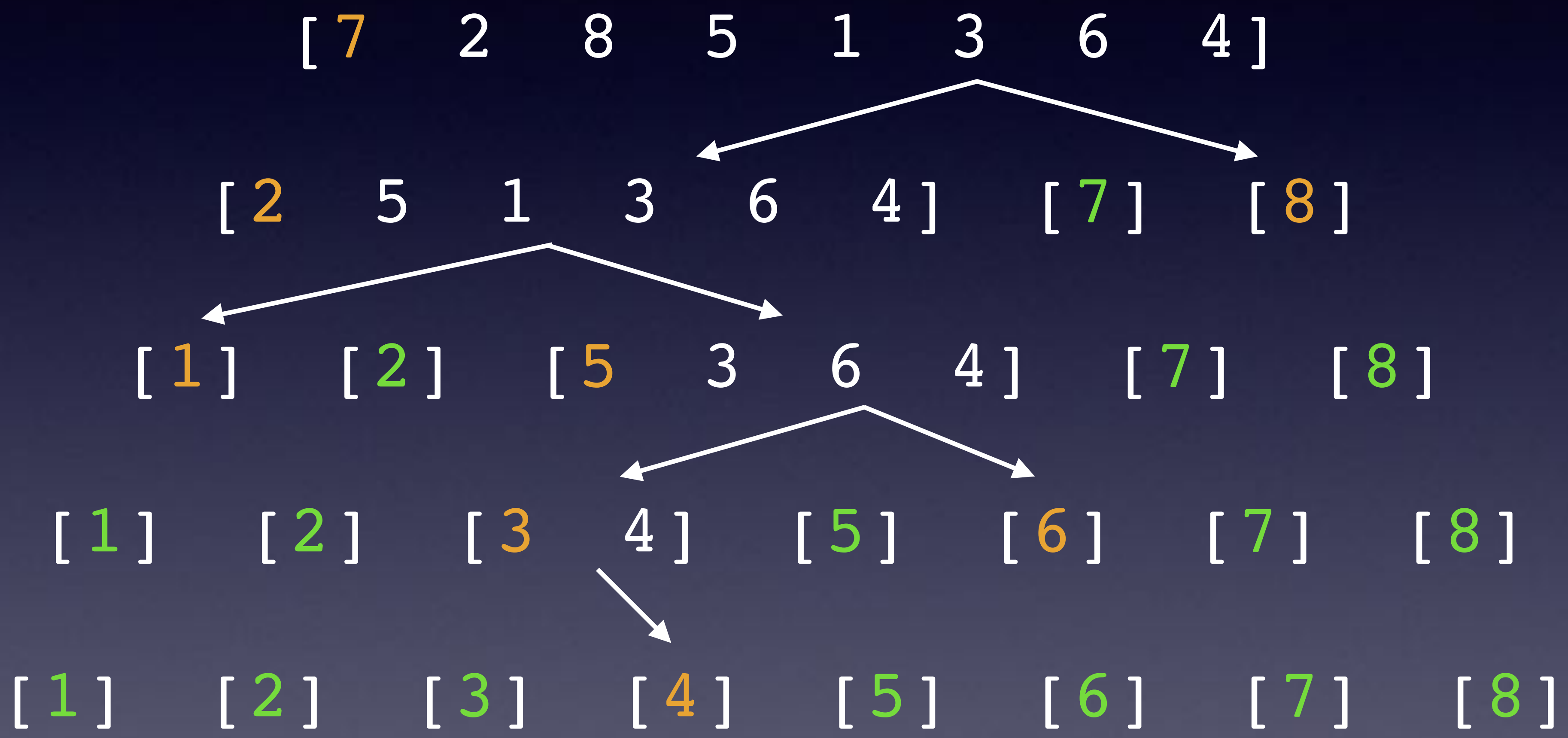
Quicksort

Here's how this algorithm would sort another list of integers:



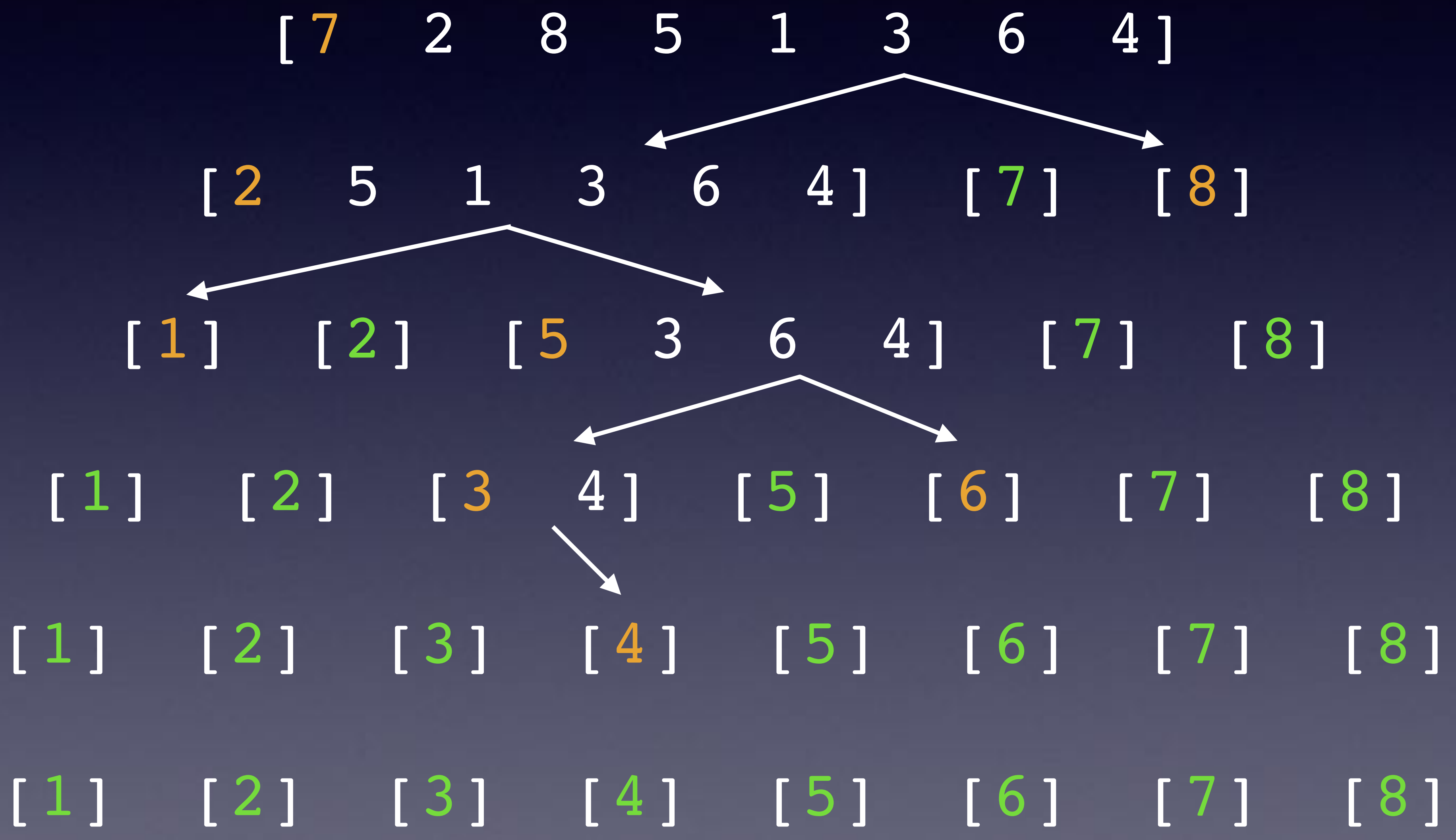
Quicksort

Here’s how this algorithm would sort another list of integers:



Quicksort

Here's how this algorithm would sort another list of integers:



(we have run out of slide, so you can do the appends on your own)

Quicksort

What input will give quicksort its **worst** performance?

Quicksort

What input will give quicksort its worst performance?

A list that's already sorted.

$1 \ \boxed{2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8}$ n – 1 comparisons

Quicksort

What input will give quicksort its worst performance?

A list that's already sorted.

$1\ \boxed{2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8}$ n – 1 comparisons

$1\ \ 2\ \boxed{3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8}$ n – 2 comparisons

Quicksort

What input will give quicksort its worst performance?

A list that's already sorted.

$1\ \boxed{2\ 3\ 4\ 5\ 6\ 7\ 8}$ n – 1 comparisons

$1\ 2\ \boxed{3\ 4\ 5\ 6\ 7\ 8}$ n – 2 comparisons

$1\ 2\ 3\ \boxed{4\ 5\ 6\ 7\ 8}$ n – 3 comparisons and so on...

Quicksort

What input will give quicksort its worst performance?

A list that's already sorted.

$1\ \boxed{2\ \ 3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8}$ n – 1 comparisons

$1\ \ 2\ \boxed{3\ \ 4\ \ 5\ \ 6\ \ 7\ \ 8}$ n – 2 comparisons

$1\ \ 2\ \ 3\ \boxed{4\ \ 5\ \ 6\ \ 7\ \ 8}$ n – 3 comparisons and so on...

$(n - 1) + (n - 2) + (n - 3) + \ldots + 2 + 1 = n(n - 1)/2$

or $\mathbf{O(n^2)}$

Quicksort

Performance is **best** when the pivot splits the unsorted portion in half.

It's **worst** when the pivot doesn't split it at all.

Performance depends on choosing good pivots.

With good pivot choices, **average** case behavior for quicksort is **O(n lg n)**.

An advantage of quicksort is that it can be done in-place, i.e. without a large temporary space. The book’s algorithm does this.

Sorting Algorithms to Know

We have explored the following sorting algorithms in this class, leaving the last one as a homework exercise.

selection sort

insertion sort

mergesort

quicksort

bubble sort

You can visualize the sorting algorithms here

Is that all there is?

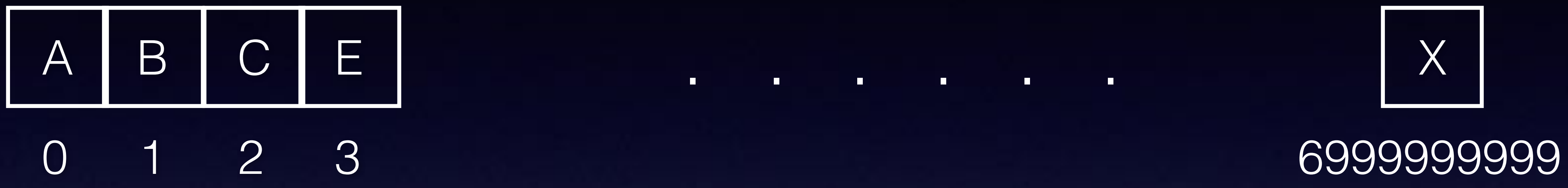
All these sorting algorithms will turn this …..

S	B	Q	A	E	P	G	L	C	N	U	K	R	X	W
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Into this …..

A	B	C	E	G	K	L	N	O	P	R	S	U	W	X
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Issue 1

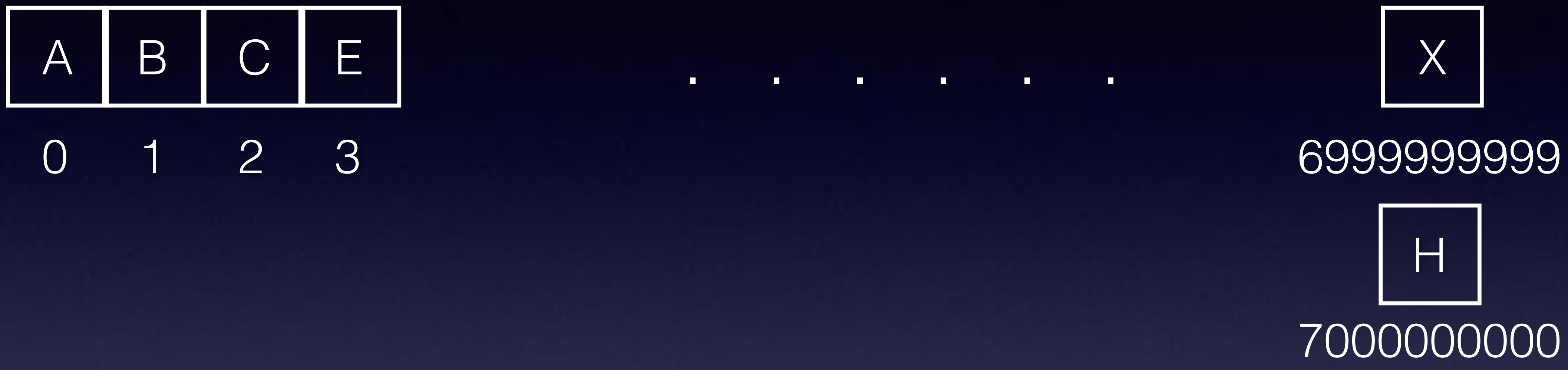


Where are we gonna put that 7 billion element "world phone book" list in memory?

Finding that much contiguous available memory might be problematic. A data structure that was more flexible, more dynamic, and much less dependent on contiguous memory could be helpful. What kind of structure might that be?

Some sort of **linked** structure seems like a possibility.

Issue 2

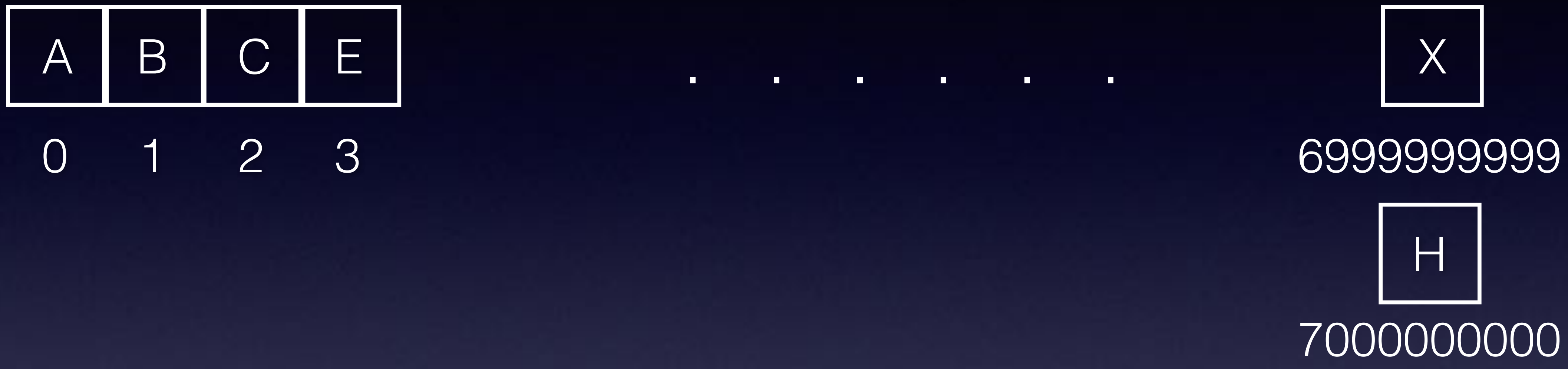


Now that we've sorted our 7 billion element list in O(n lg n) time, what do we do if we want to add one more element in the right place?

How about if we add that element to the end of the existing list and then apply quicksort to the whole thing again?

What input may cause quicksort to perform with its absolute worst time complexity $O(n^2)$?

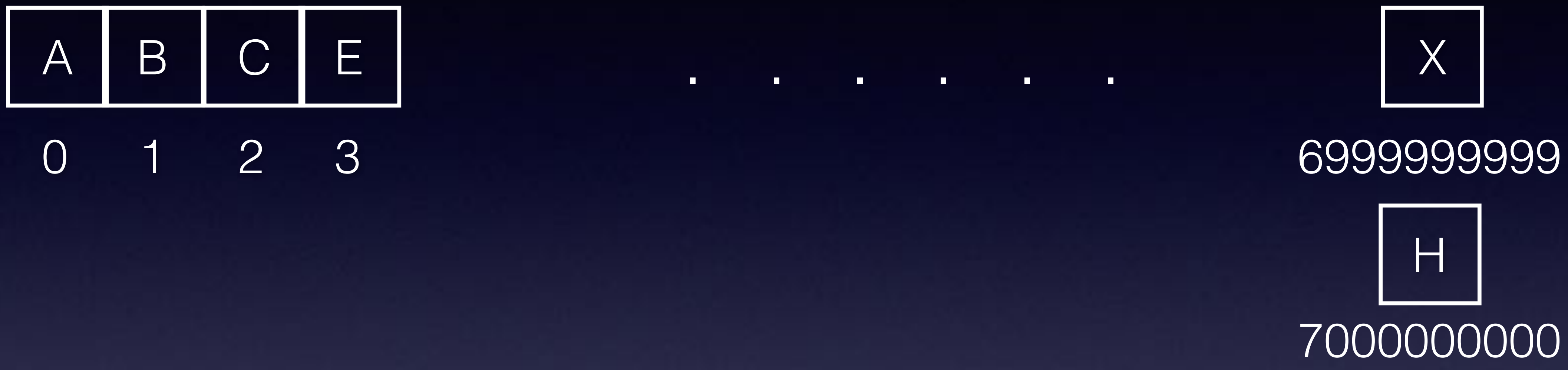
Issue 2



Isn't it nice that you know something about algorithm analysis to avoid that problem?

Let's try another approach...

Issue 2

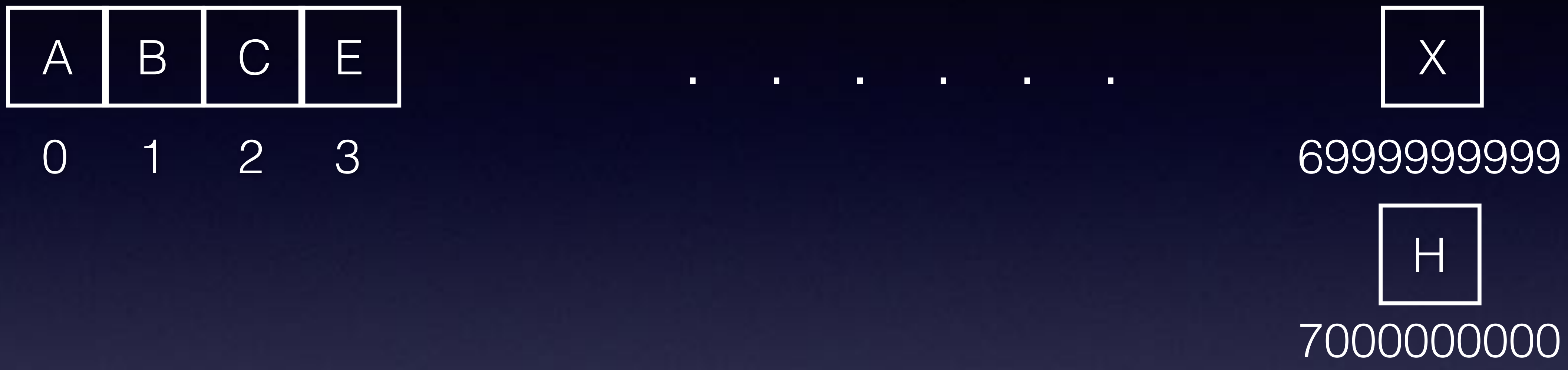


How about we just insert the element in its appropriate place by searching the sorted array until we find the right place.

Then make a space by moving all the elements past that place one space toward the back, like one pass of insertion sort.

What’s the time complexity of insertion now?

Issue 2

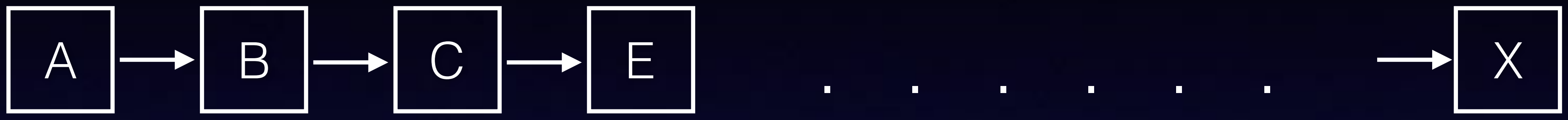


What's the time complexity of insertion now?

Using binary search, we can find the insertion point in O(lg n) comparisons, but moving elements will take O(n) movements.

So O(n), at least it beats O(n^2), but...

Issue 2



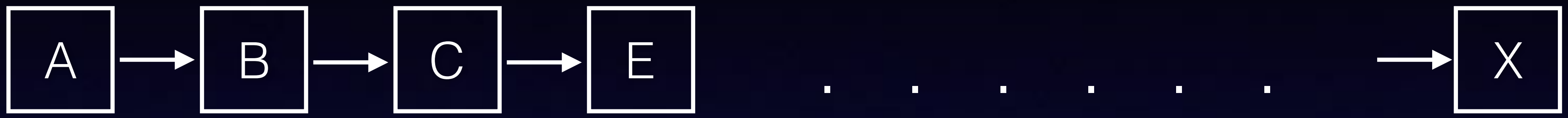
Maybe we could speed up the insertion time even more.

Before we hinted at a linked list structure for flexibility.

Can we do binary search on a singly-linked list to get that O(log n) time complexity?

No, we have to be able to jump in both directions in the sequence to make binary search work.

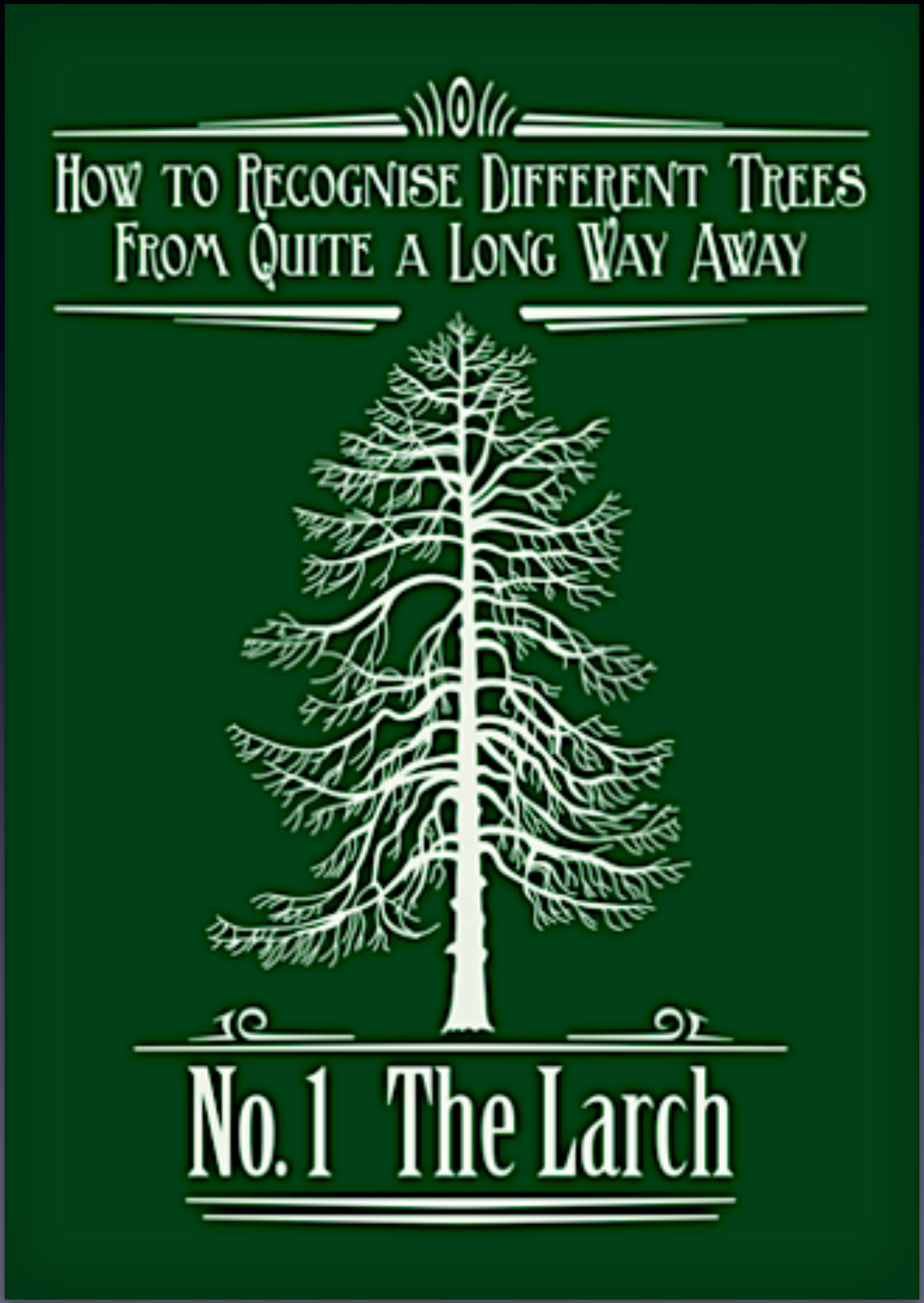
Issue 2



A doubly-linked list lets us move in both directions. So binary search should be easier, but we can't move the pointers (left, right, mid) with simple index arithmetic. We would have to do link traversals, and as long as we're traversing the links in the list, we might as well look at the values at the nodes as we pass by, and this just turns into linear search.

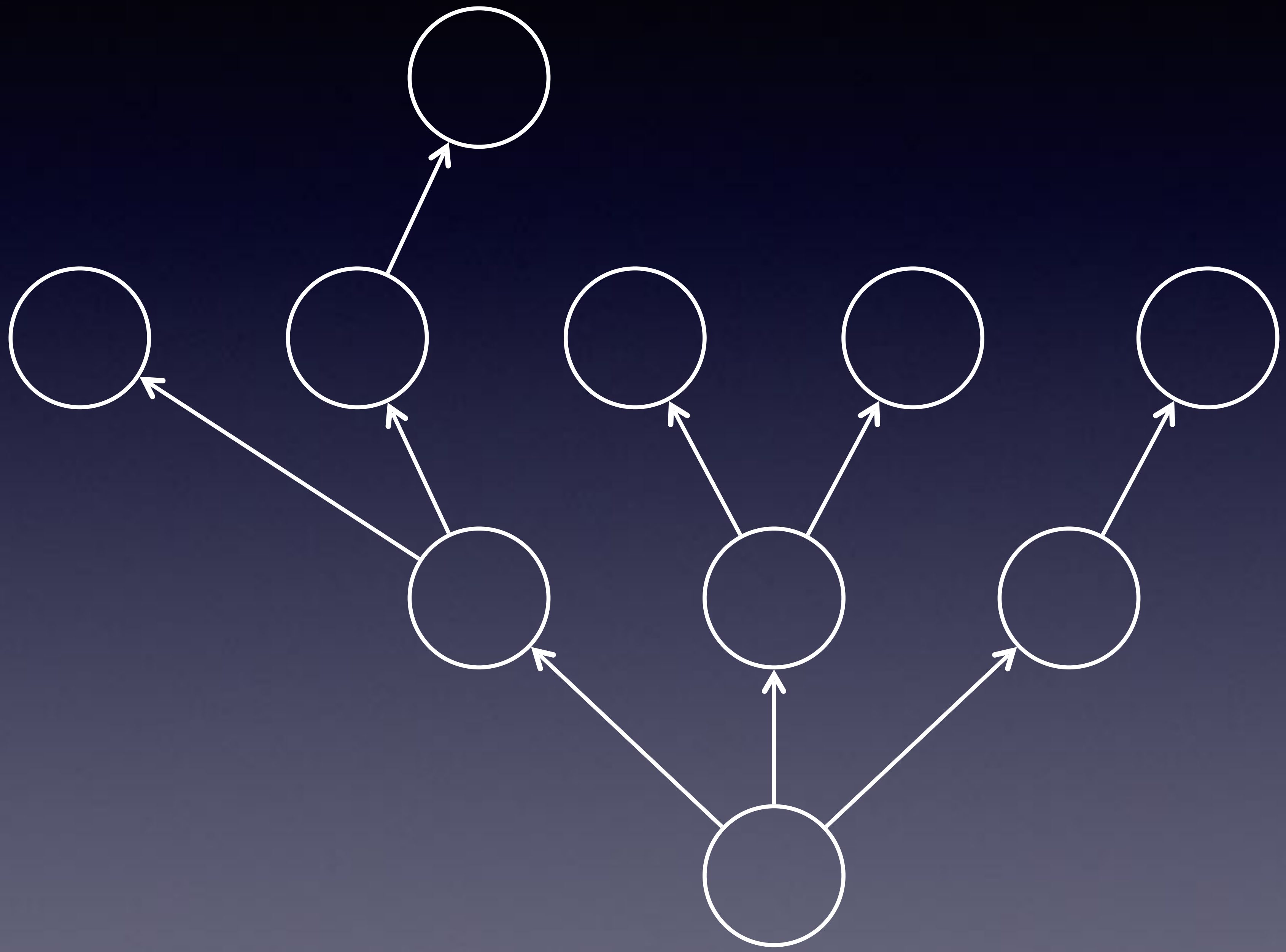
What is the linked structure that makes it all come together?

The Tree

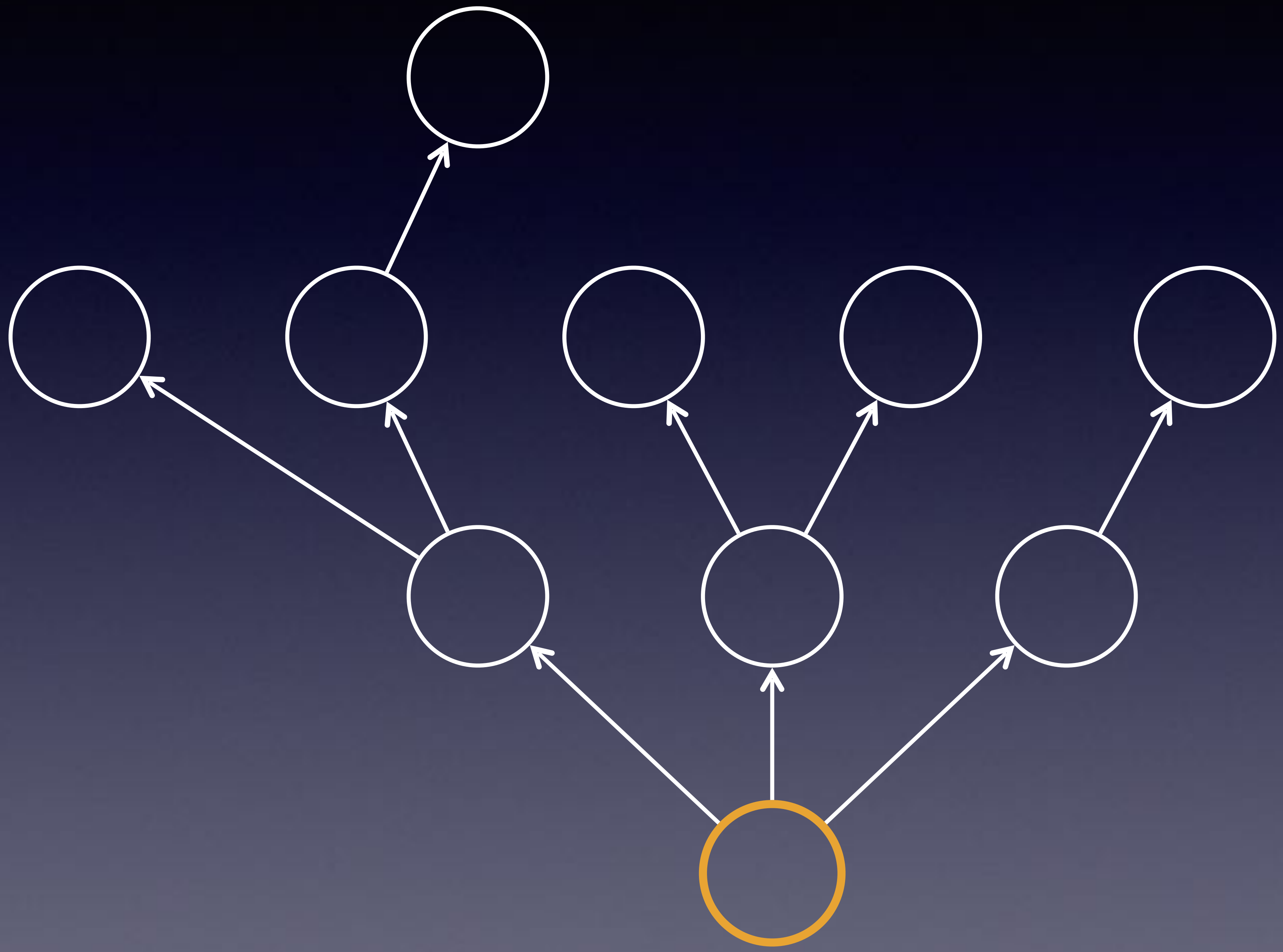


No, not that kind…

The Tree

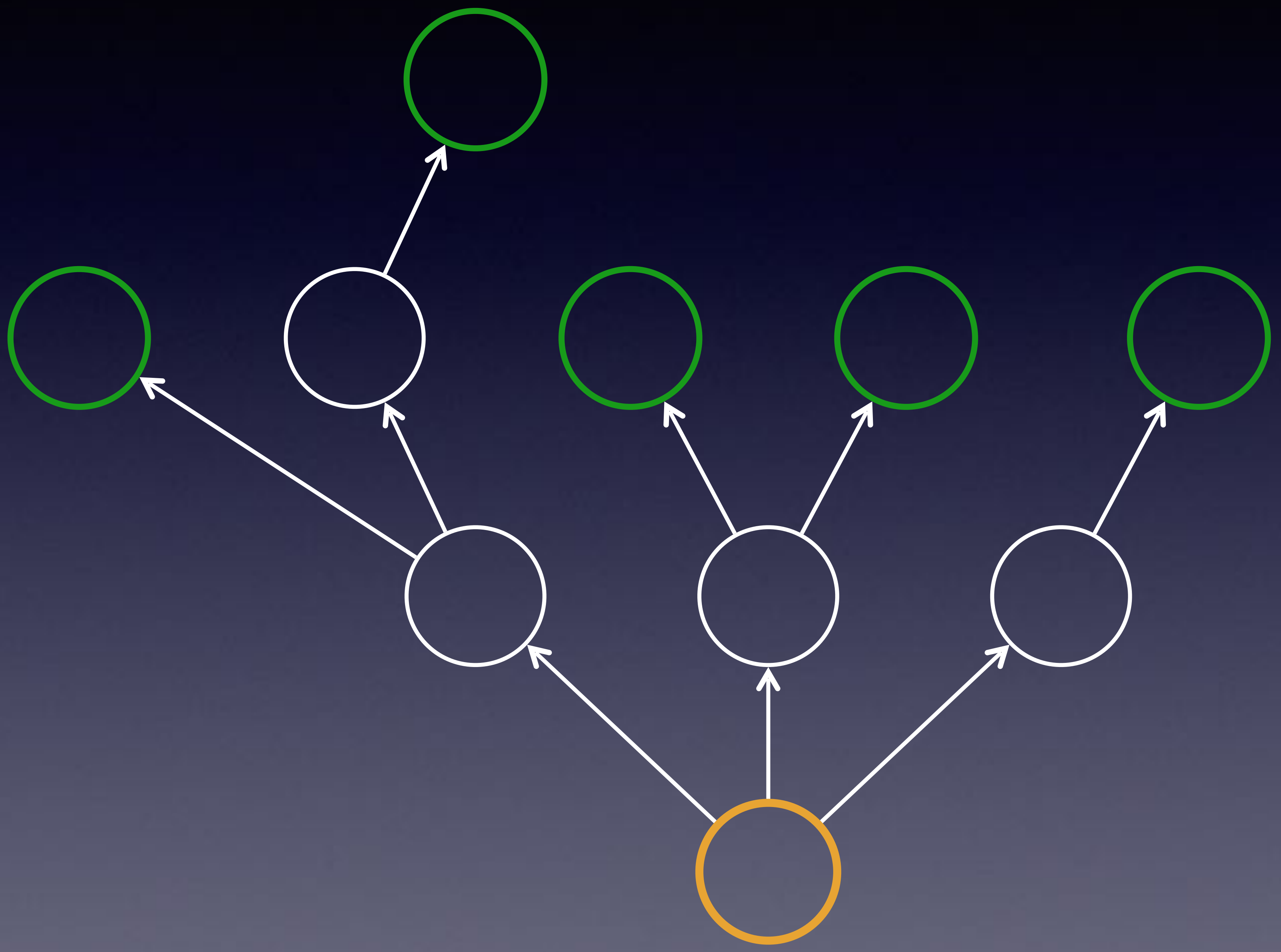


The Tree



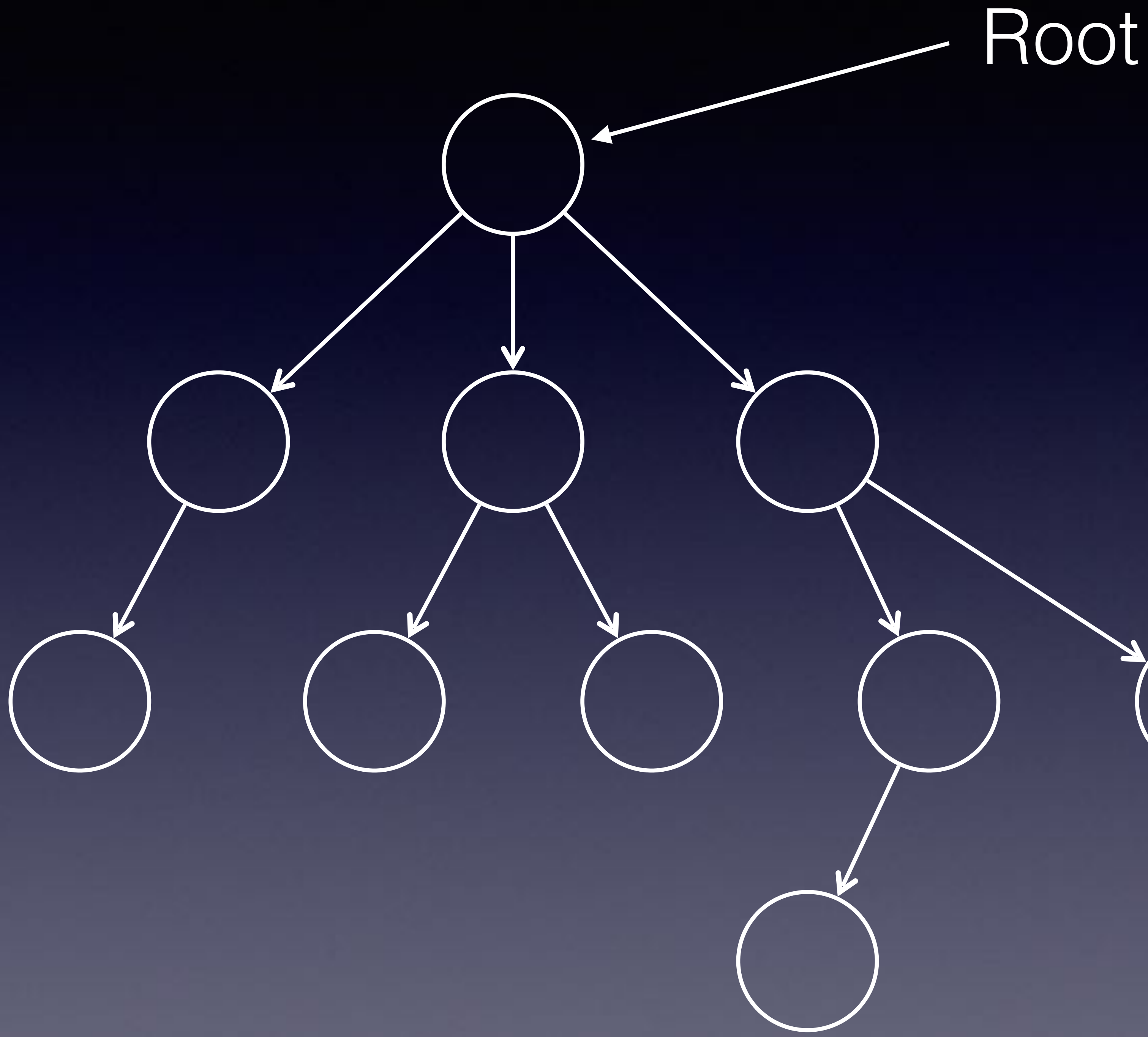
In computing, a tree has a **root** (or root node)

The Tree



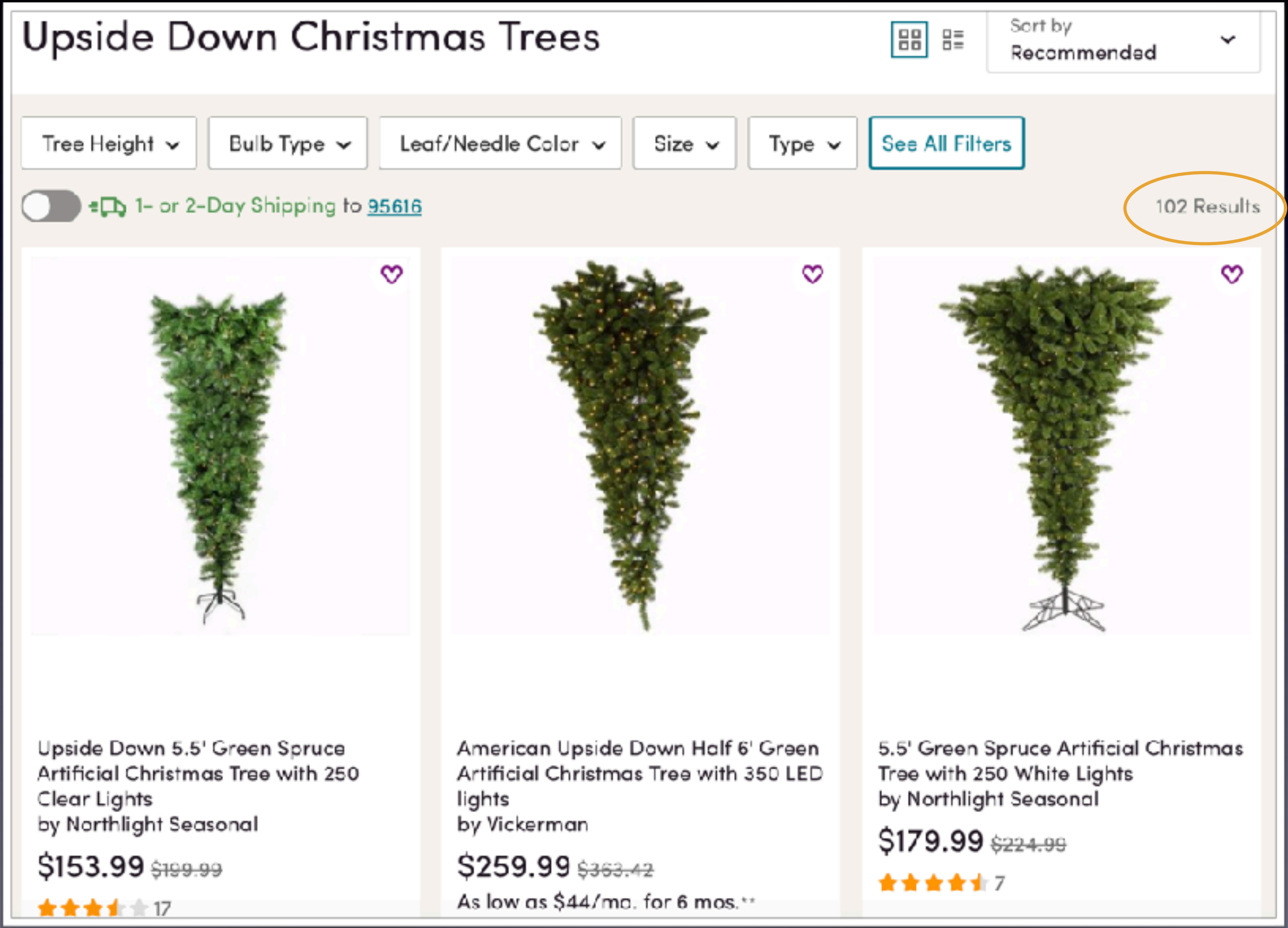
In computing, a tree has a **root** (or root node) and **leaves** (or leaf nodes) just like a botanical tree. But the resemblance between our trees and nature's best sort of ends there.

The Tree



For example, we like our trees upside down. Non-computing people tend to think this is weird. Really, we're just trendsetters...

Upside down trees



Upside Down Christmas Trees

1- or 2-Day Shipping to 95616

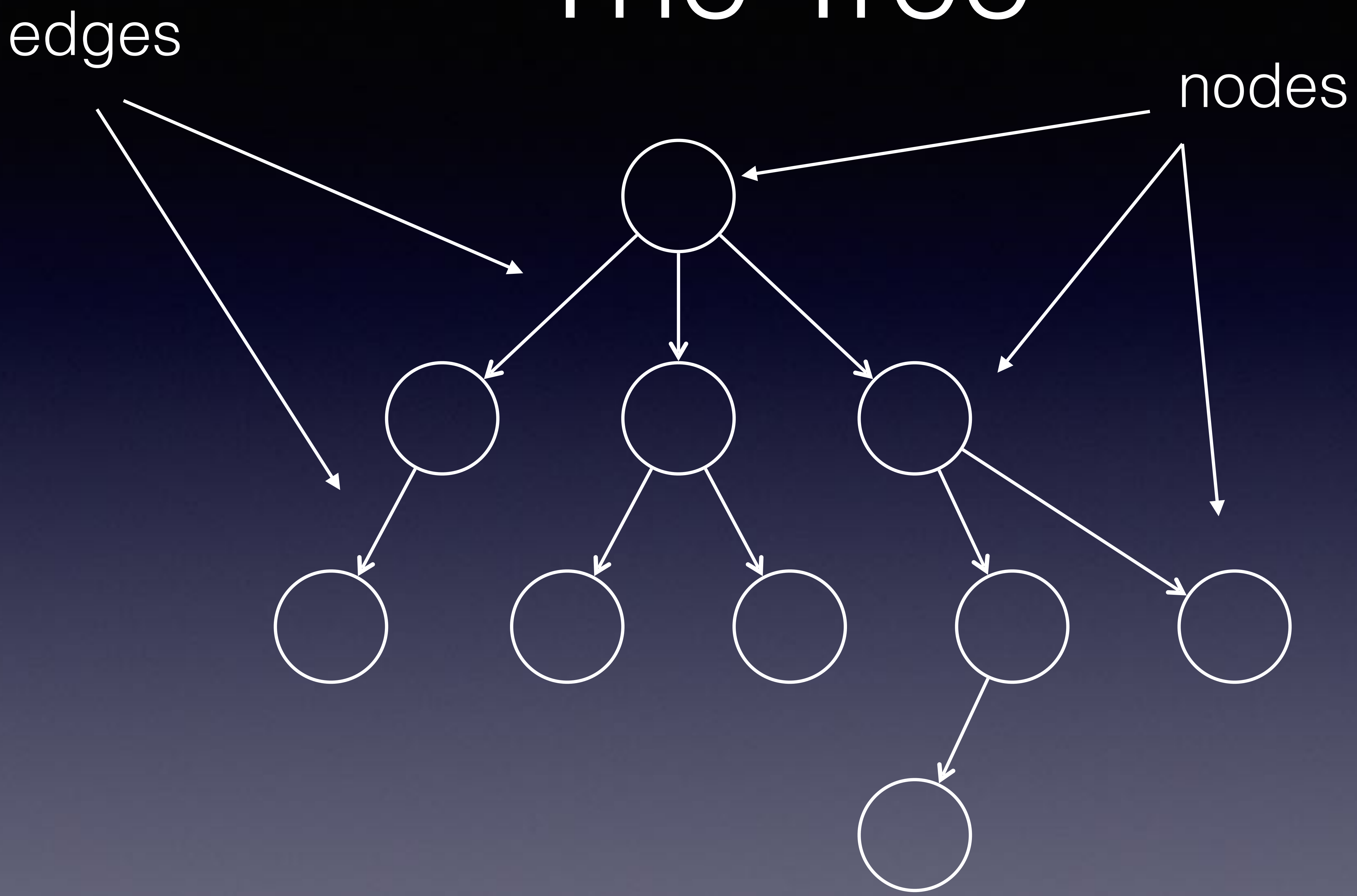
102 Results

Upside Down 5.5' Green Spruce Artificial Christmas Tree with 250 Clear Lights
by Northlight Seasonal
$153.99 ~~$199.99~~
17

American Upside Down Half 6' Green Artificial Christmas Tree with 350 LED lights
by Vickerman
$259.99 ~~$363.42~~
As low as $44/mo. for 6 mos.**

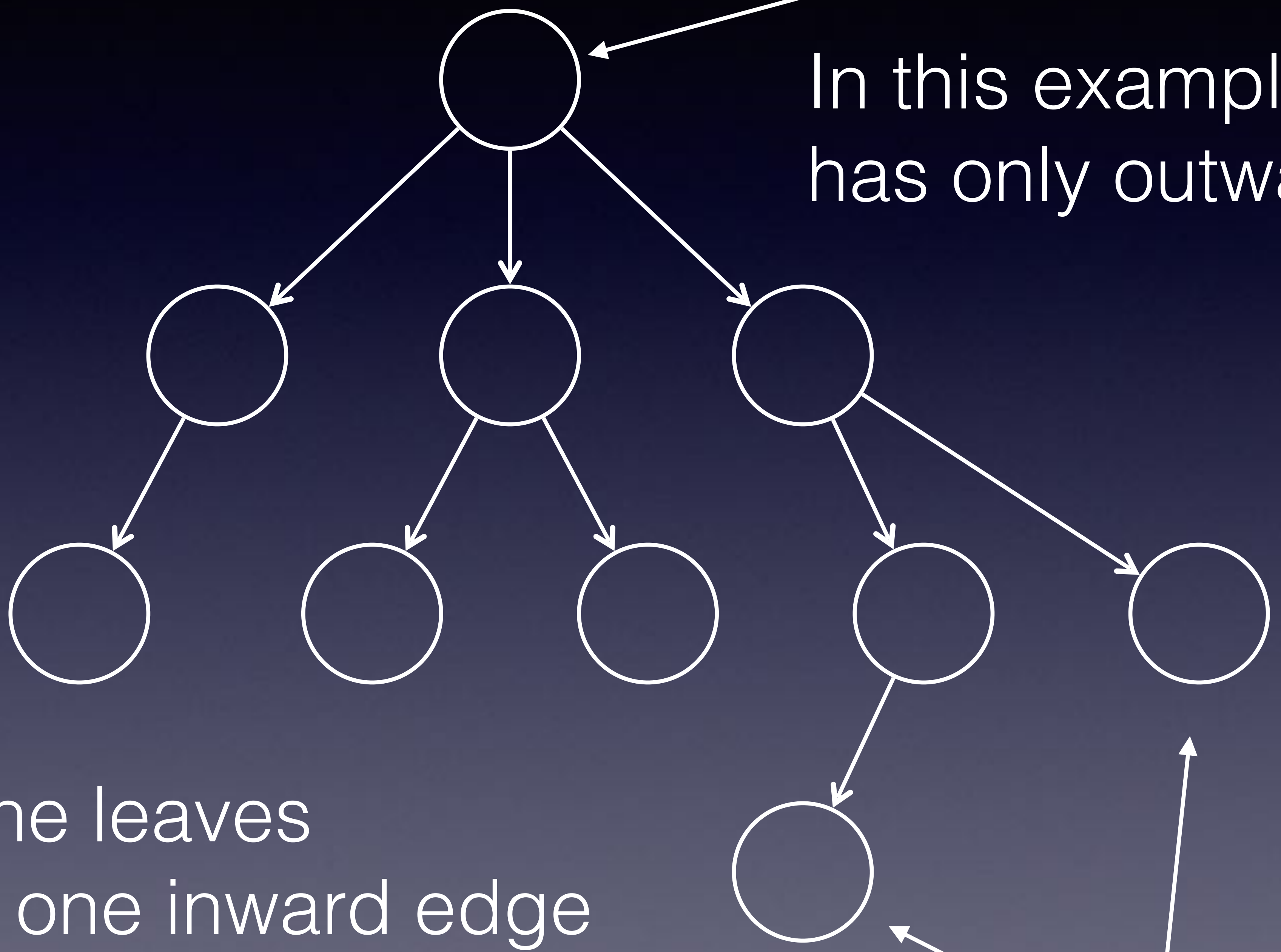
5.5' Green Spruce Artificial Christmas Tree with 250 White Lights
by Northlight Seasonal
$179.99 ~~$224.99~~
7

The Tree



A tree has **nodes** and **edges**

The Tree



Root node

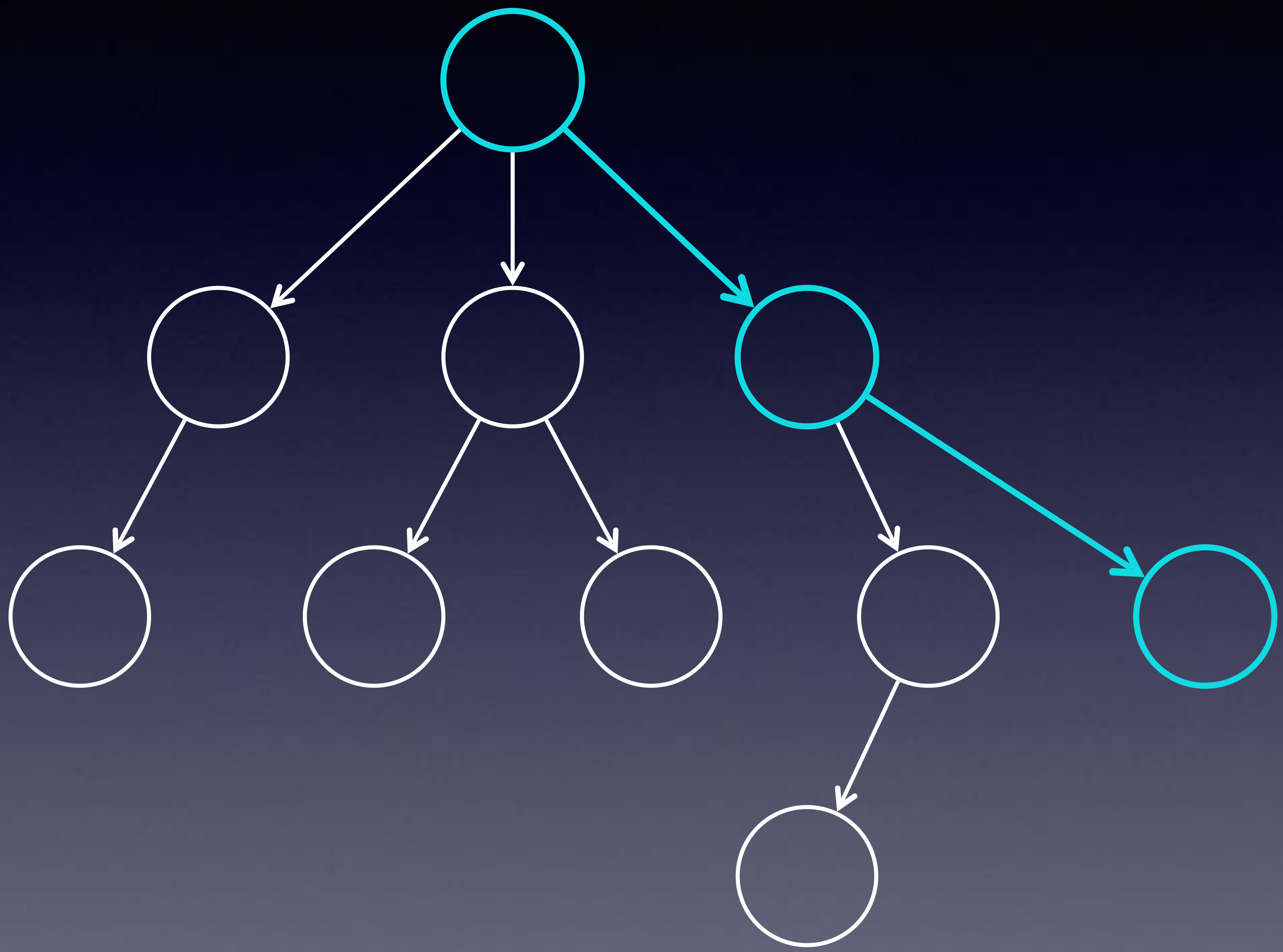
In this example, the root has only outward edges

The leaves each only one inward edge

Leaf nodes

Edges point away from the root

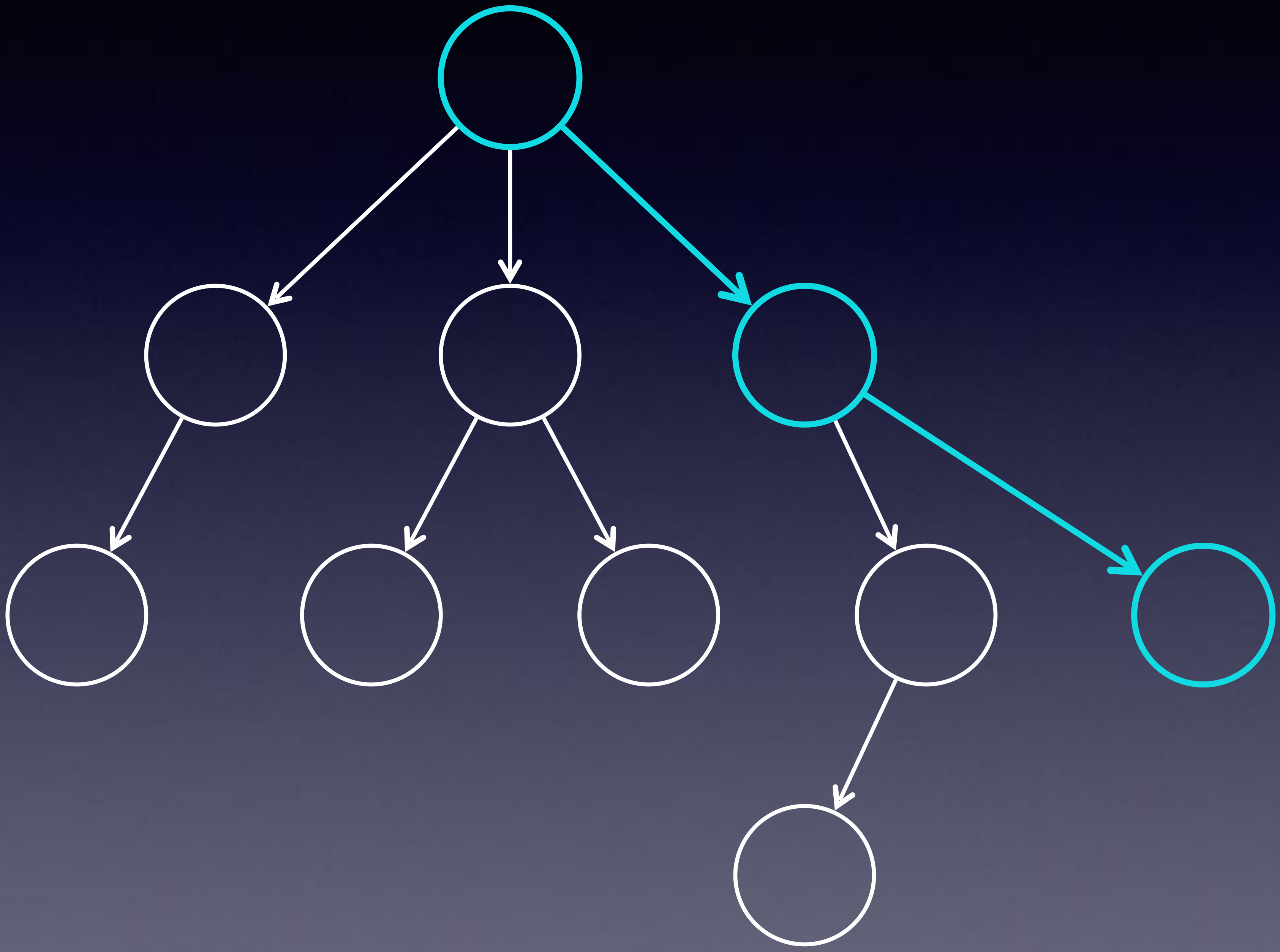
The Tree



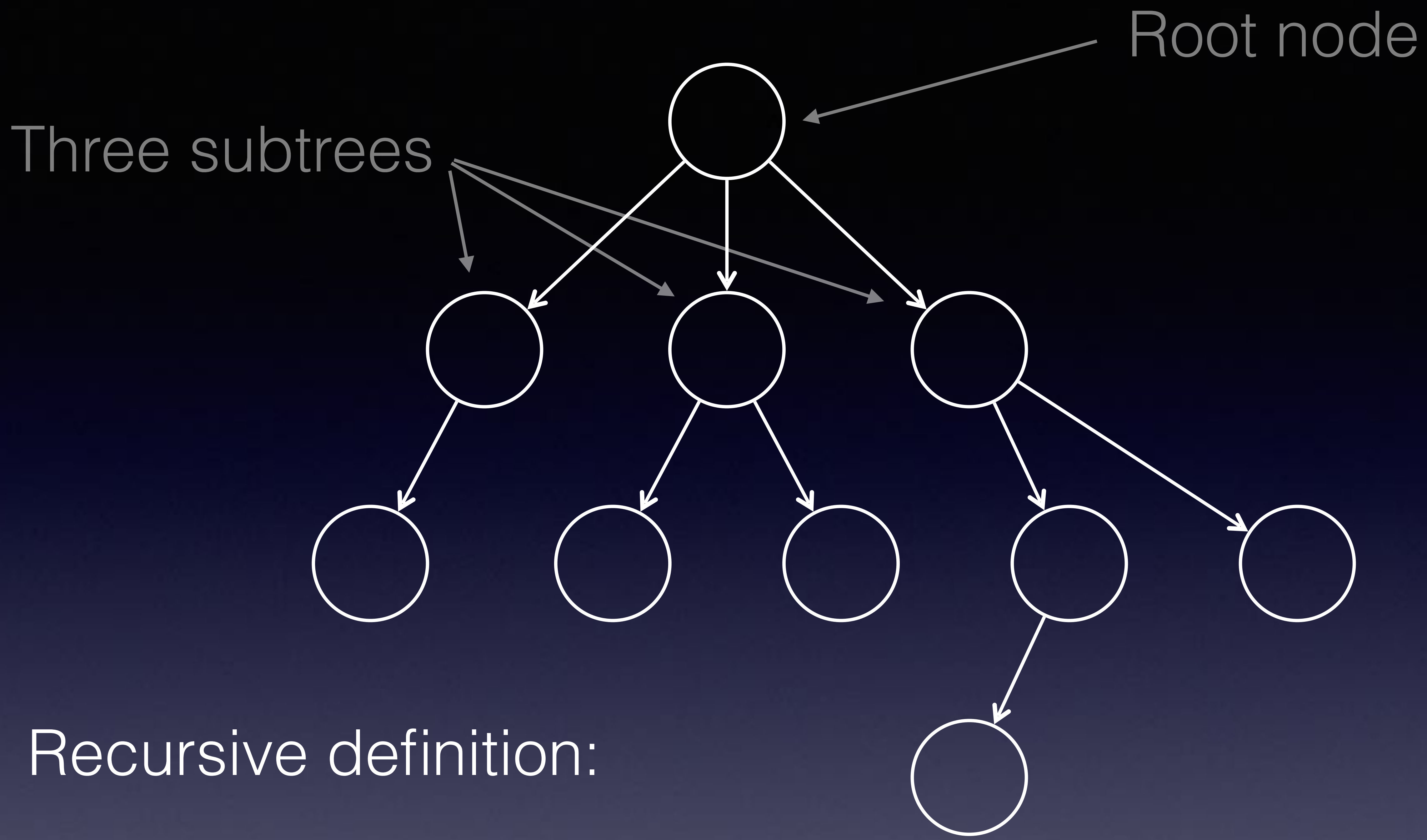
A **path** uses edges to traverse a sequence of nodes.

No path will include a node twice, that’s a tree property.

The Tree



A unique path traverses from the root to each node.



Recursive definition:

A **tree** is a data structure made up of **nodes** (or sometimes called vertices) and **edges.** The tree with no nodes is called the empty tree. A tree that is not empty consists of a root node and and zero or more **subtrees**, each of which is also a tree.

Binary Trees

For reasons which will become obvious, we're primarily interested in binary trees (and variations on the binary-tree theme).

A **binary tree** is a tree that is either

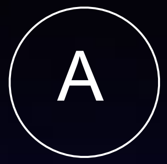
- empty, or
- a node called the root node and two binary trees called the left subtree and a right subtree

(Remember that there can't be multiple paths from the root to any leaf node in a tree.)

Binary Trees

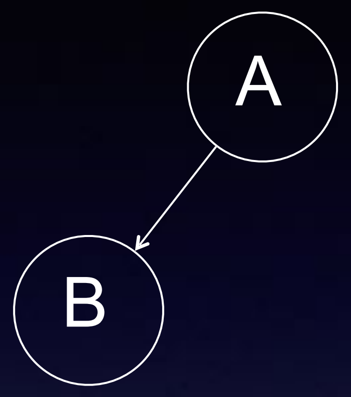
So this is a binary tree. It’s the empty tree.

Binary Trees



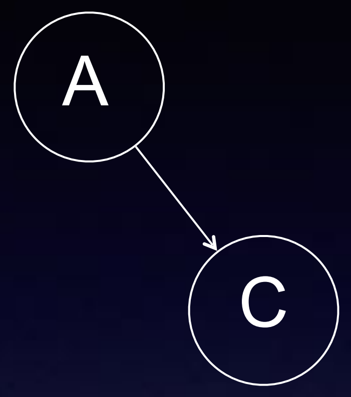
This is a binary tree too.
It consists of a root with empty left and right subtrees.

Binary Trees


$$\begin{tikzcd}
 & A \arrow[dl] \\
B &
\end{tikzcd}$$

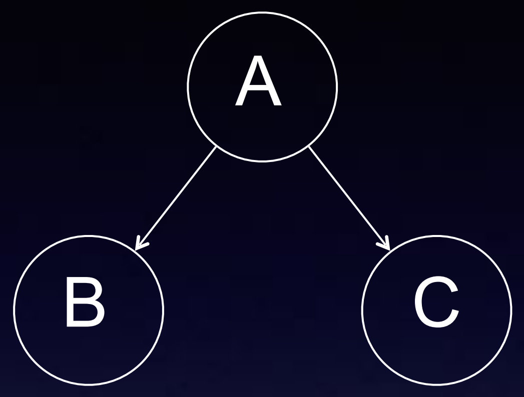
And this is a binary tree.

Binary Trees



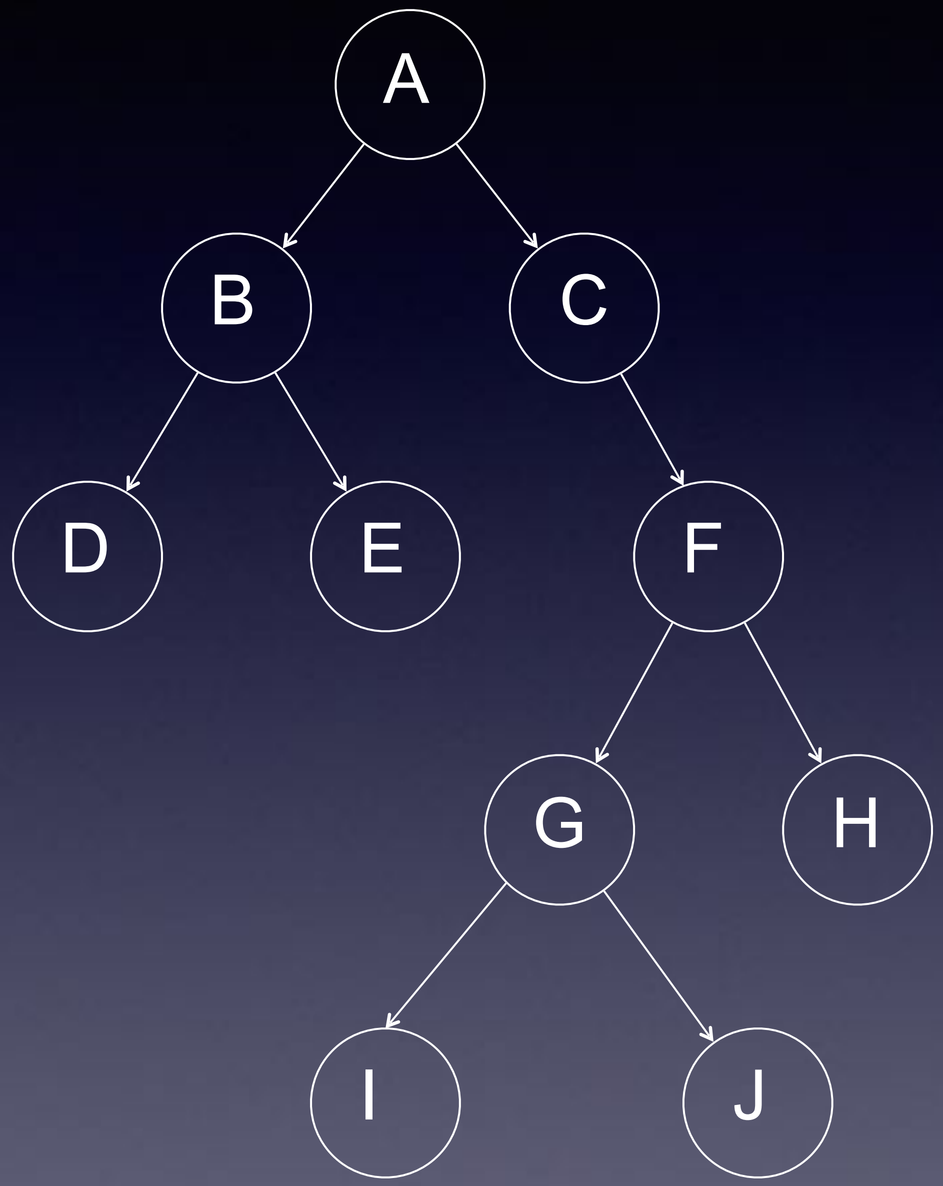
Here’s another one.

Binary Trees



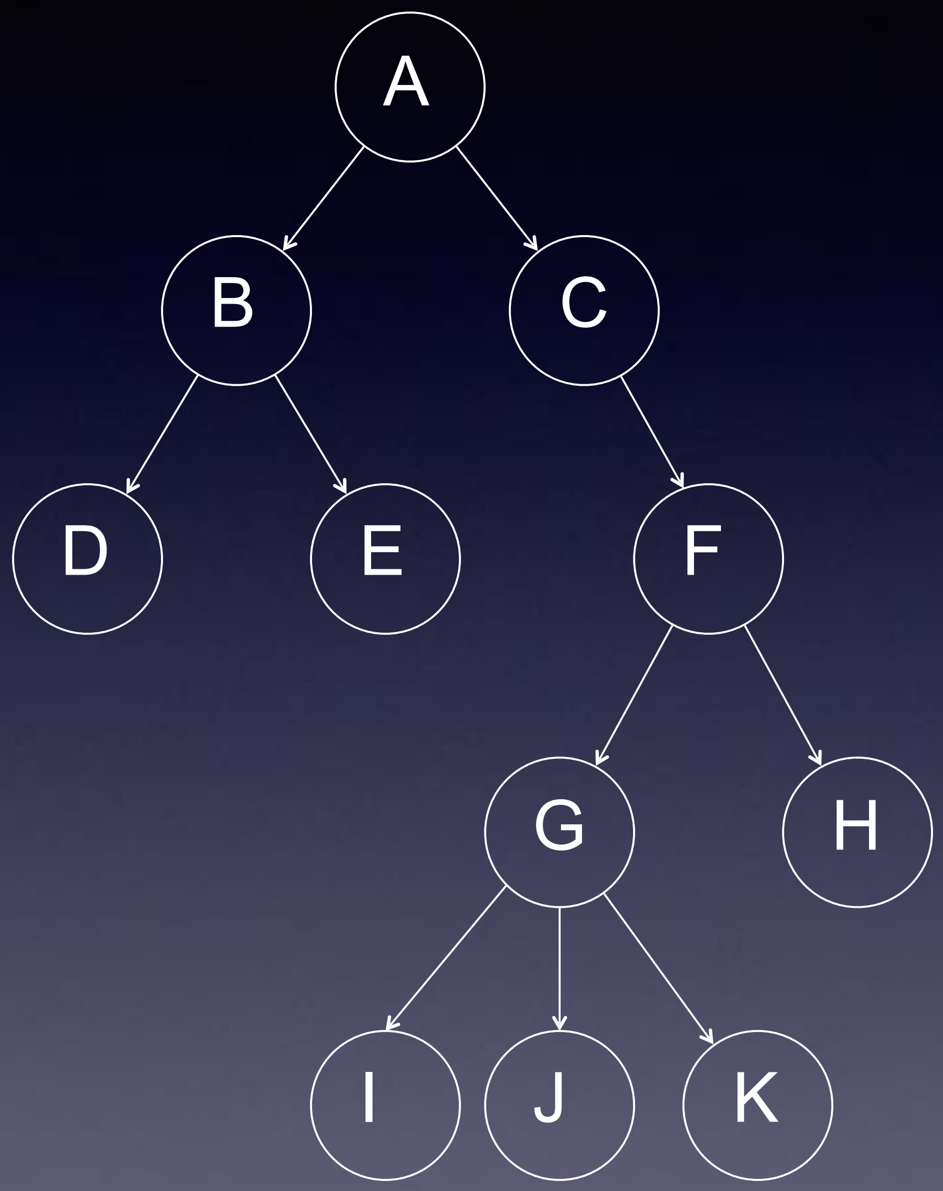
Of course this is a binary tree.

Binary Trees



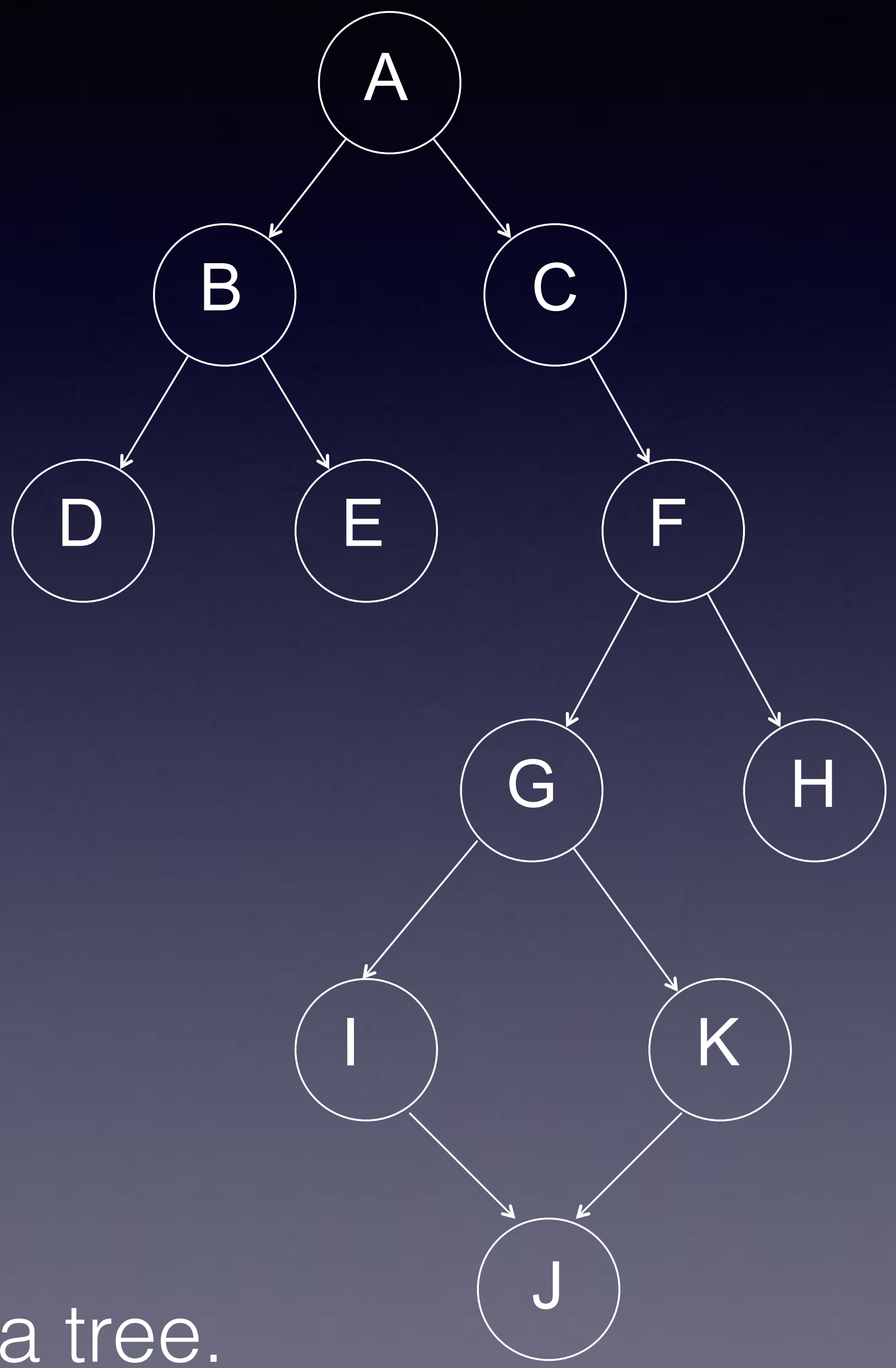
Here is yet another.

Binary Trees



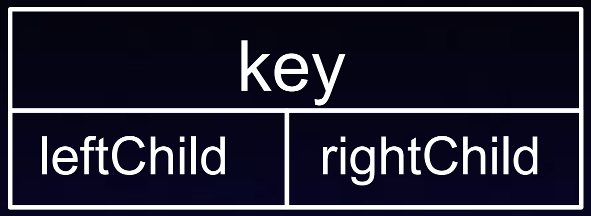
This is a tree, but it’s not binary.

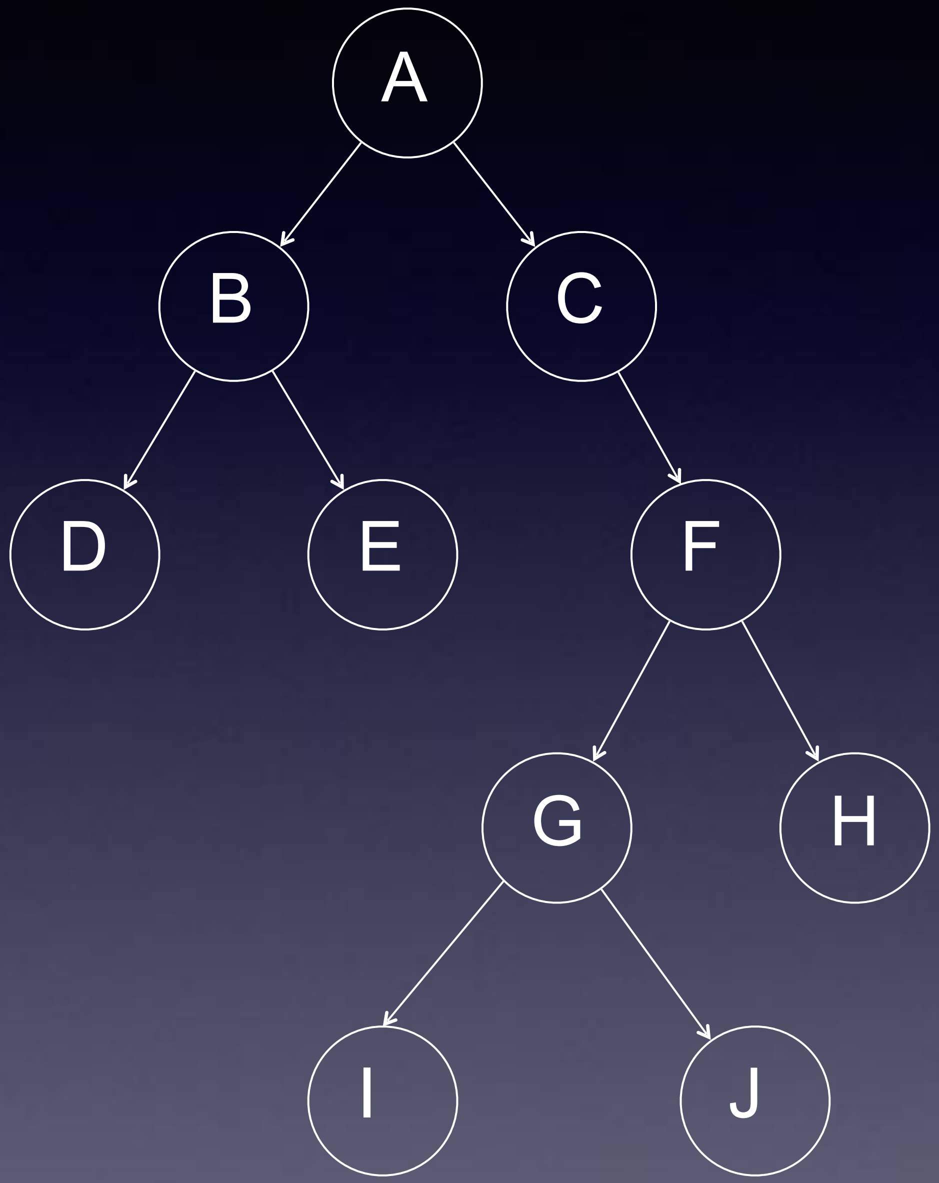
Binary Trees



This might be "binary", but it’s not a tree.

Binary Trees

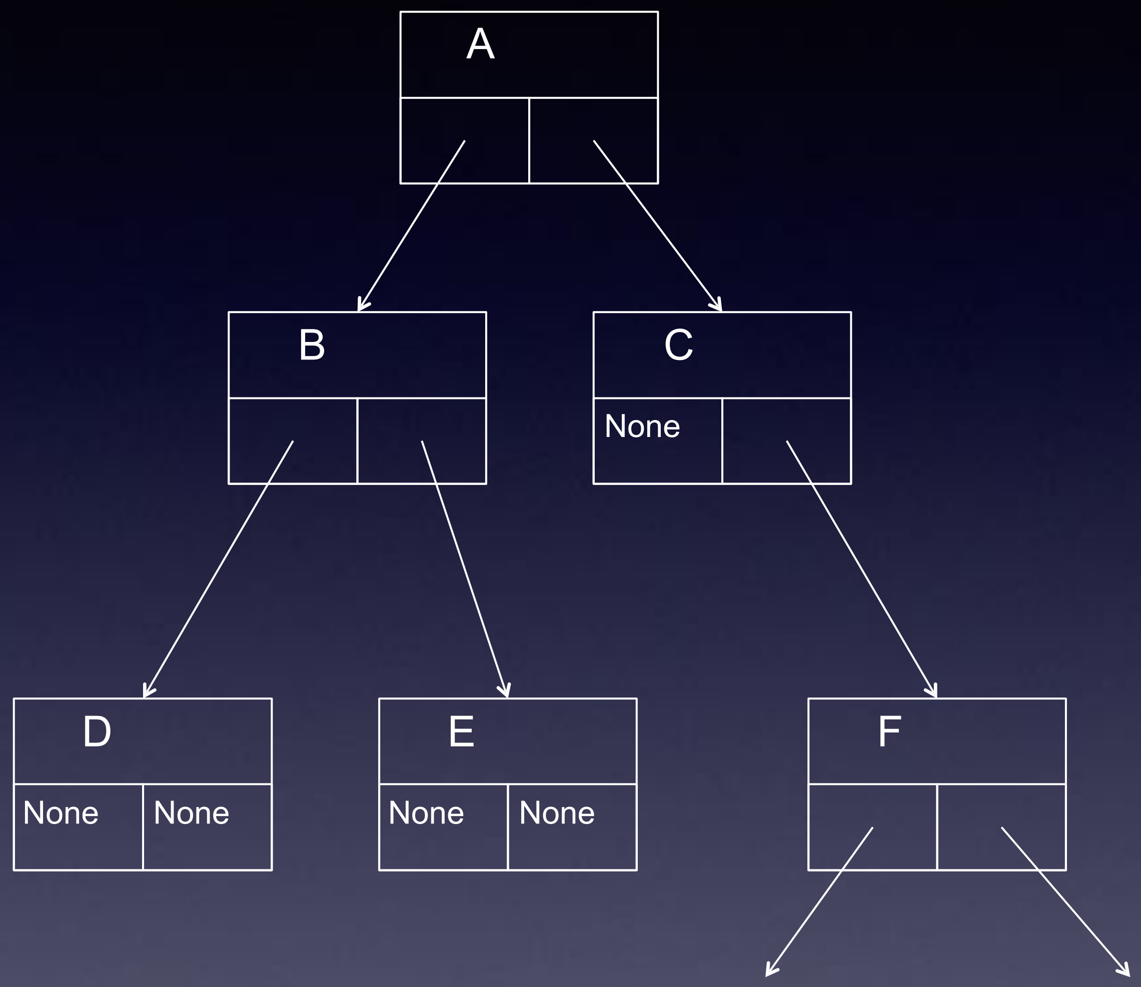


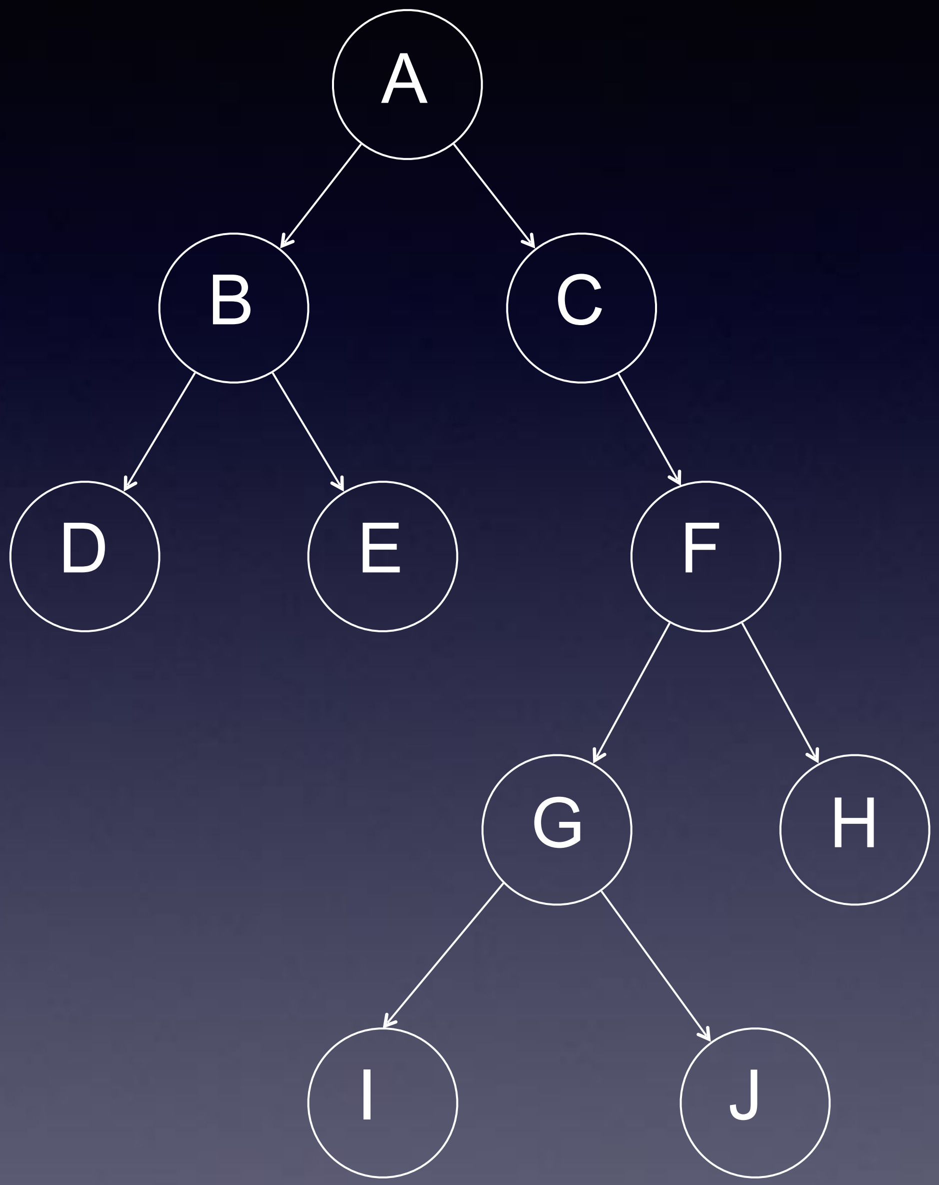


```
class BinaryTree:
  def __init__(self, rootObj):
    self.key = rootObj
    self.leftChild = None
    self.rightChild = None
```

Each binary tree node holds some data, a pointer to its left subtree and a pointer to its right subtree.

Binary Trees

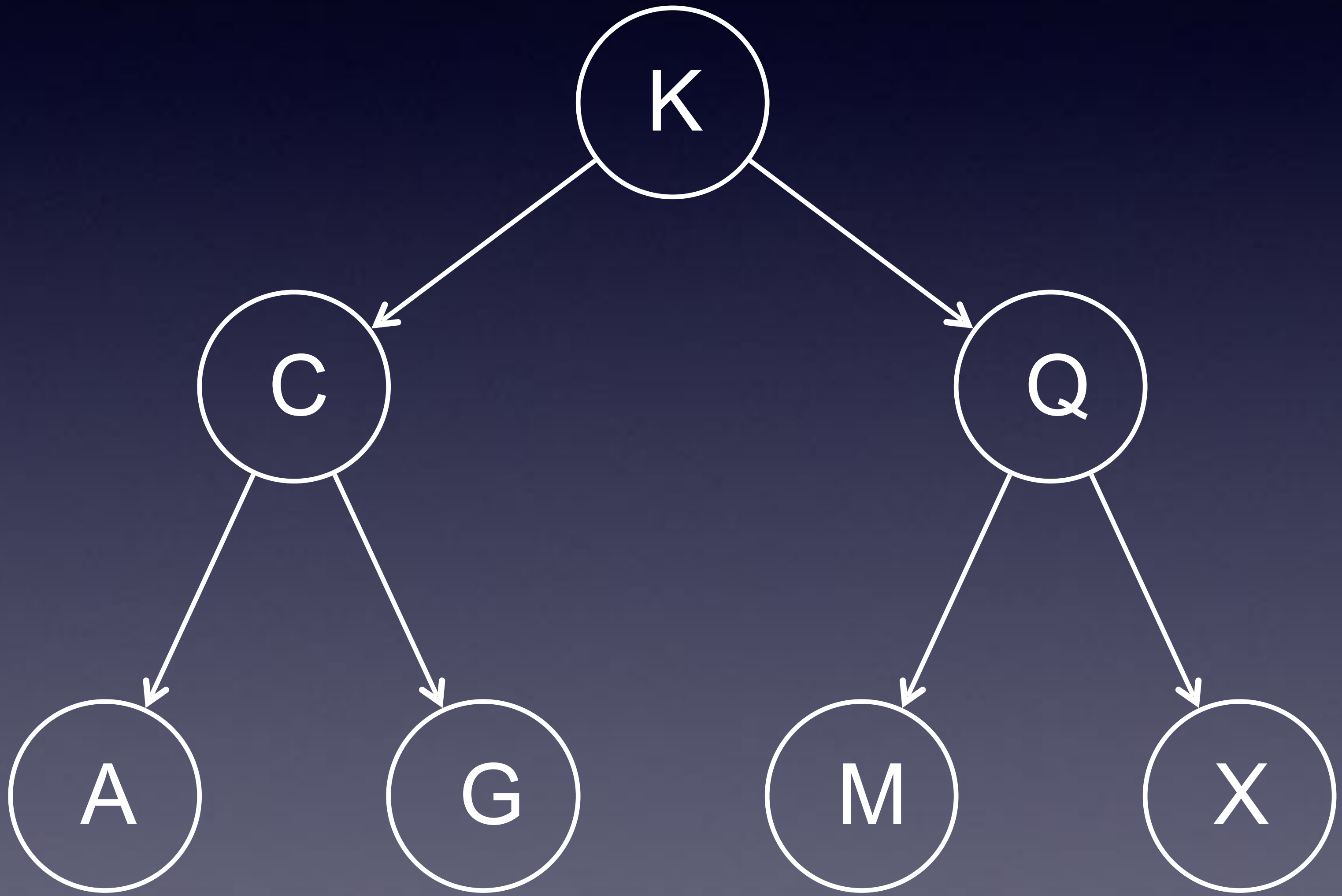




Each binary tree node holds some data, a pointer to its left subtree and a pointer to its right subtree.

Binary search trees

This is a binary search tree:



Binary search trees*

Binary trees are cool, but for searching, we are really interested in is a class of binary trees called binary search trees.

A **binary search tree** is a binary tree in which every node is

- empty or
- the root of a binary tree in which all the values in the left subtree are less than the value at the root, and all the values in the right subtree are greater than the value at the root.

* We are deviating from the order in which things are presented in chapter 6.

Binary search trees

This is also a binary search tree: