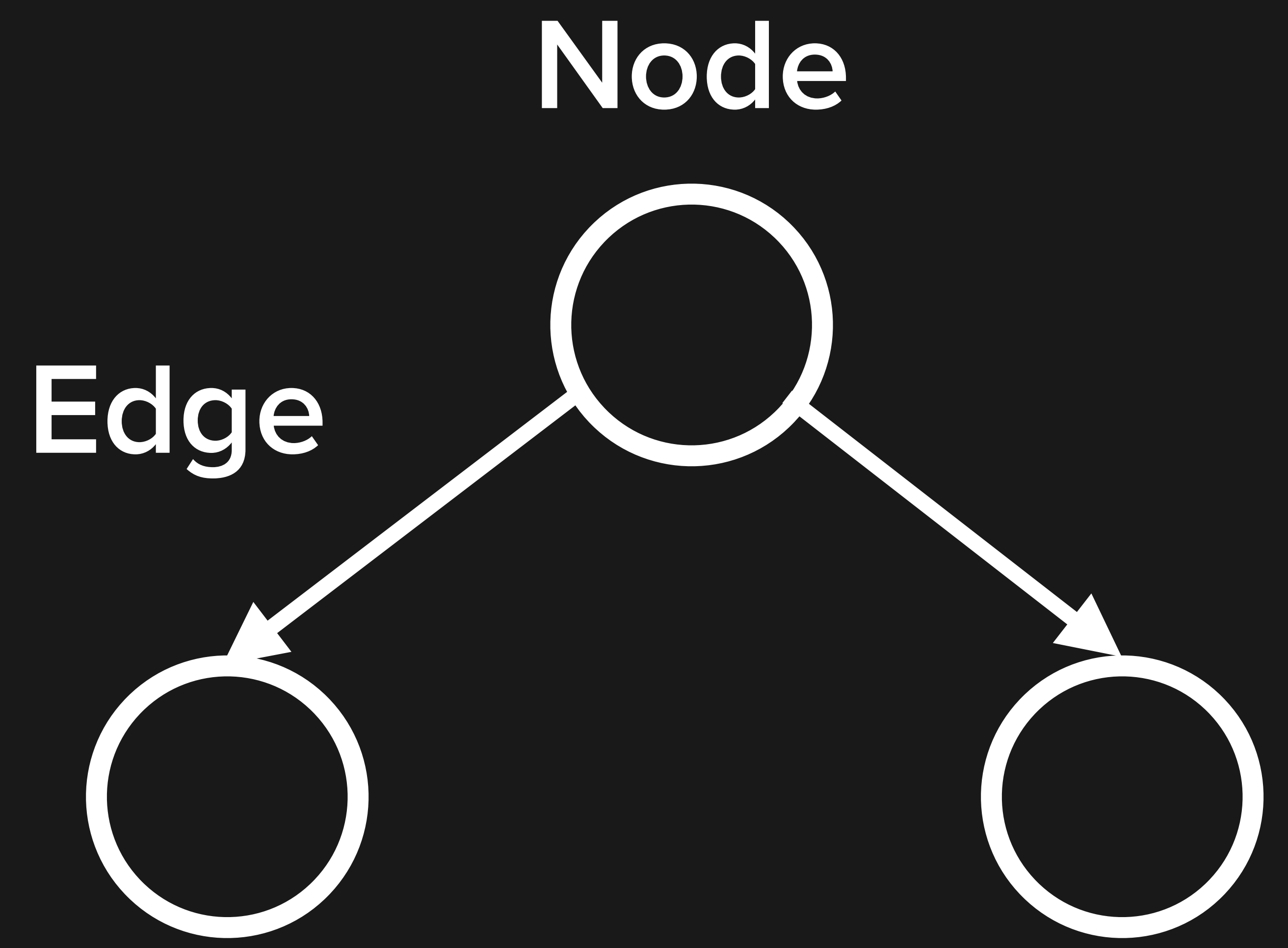
TREES

So log-arithmic. (Get it!?)

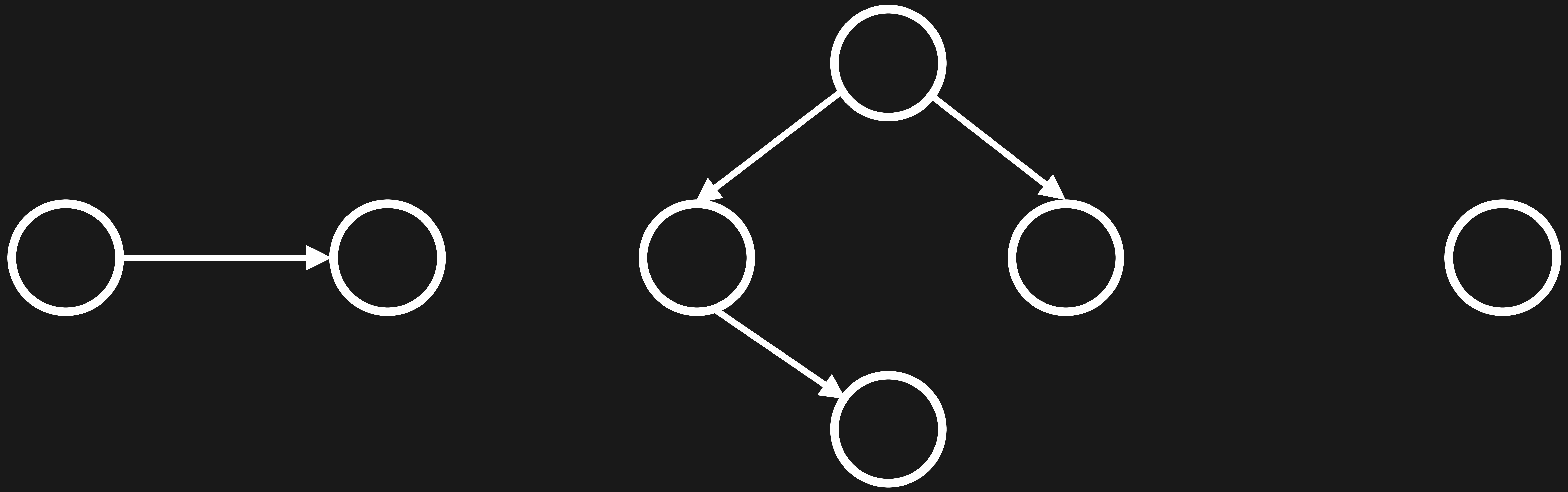
TREE

Set of nodes and edges (references to child nodes) without any cycles

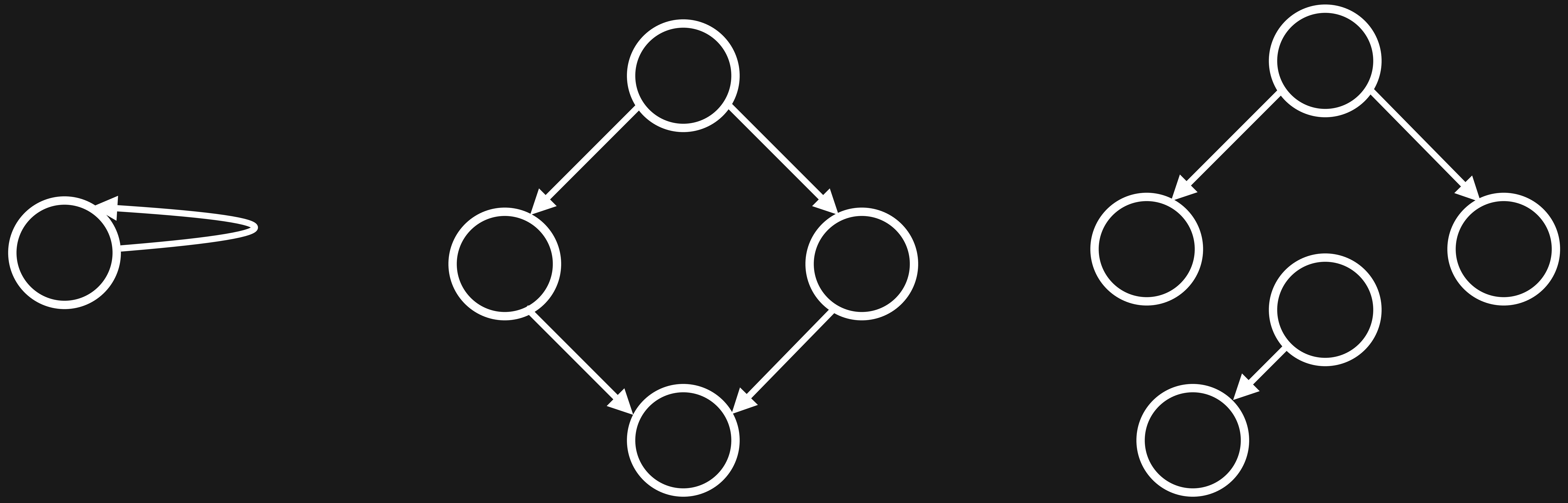


http://en.wikipedia.org/wiki/Tree_(data_structure)

TREES

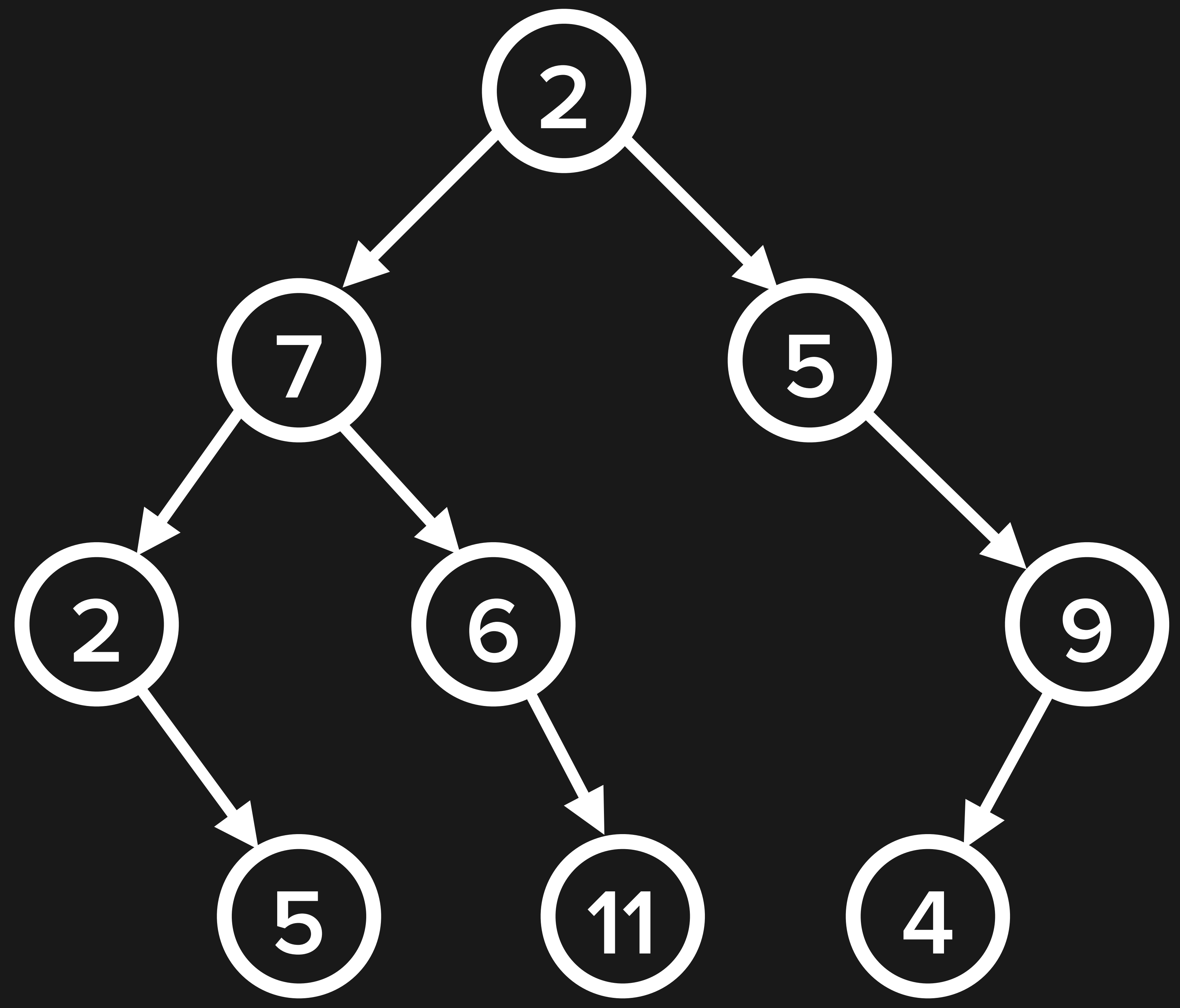


NOT TREES



BINARY TREE

Tree in which each node has at most two children



Size 9   Height 3

http://en.wikipedia.org/wiki/Binary_tree

root
unique topmost node of tree that can reach all other nodes

parent ➡ child
node above ➡ node below

descendant
node reachable from parent to child, grandchild, etc.

ancestor
node reachable from child to parent, grandparent, etc.

leaf / external node
node with no children

branch / internal node
node with at least one child

size (tree)
number of nodes in the tree

height (tree)
number of edges on *longest* downward path from ***root*** to leaf

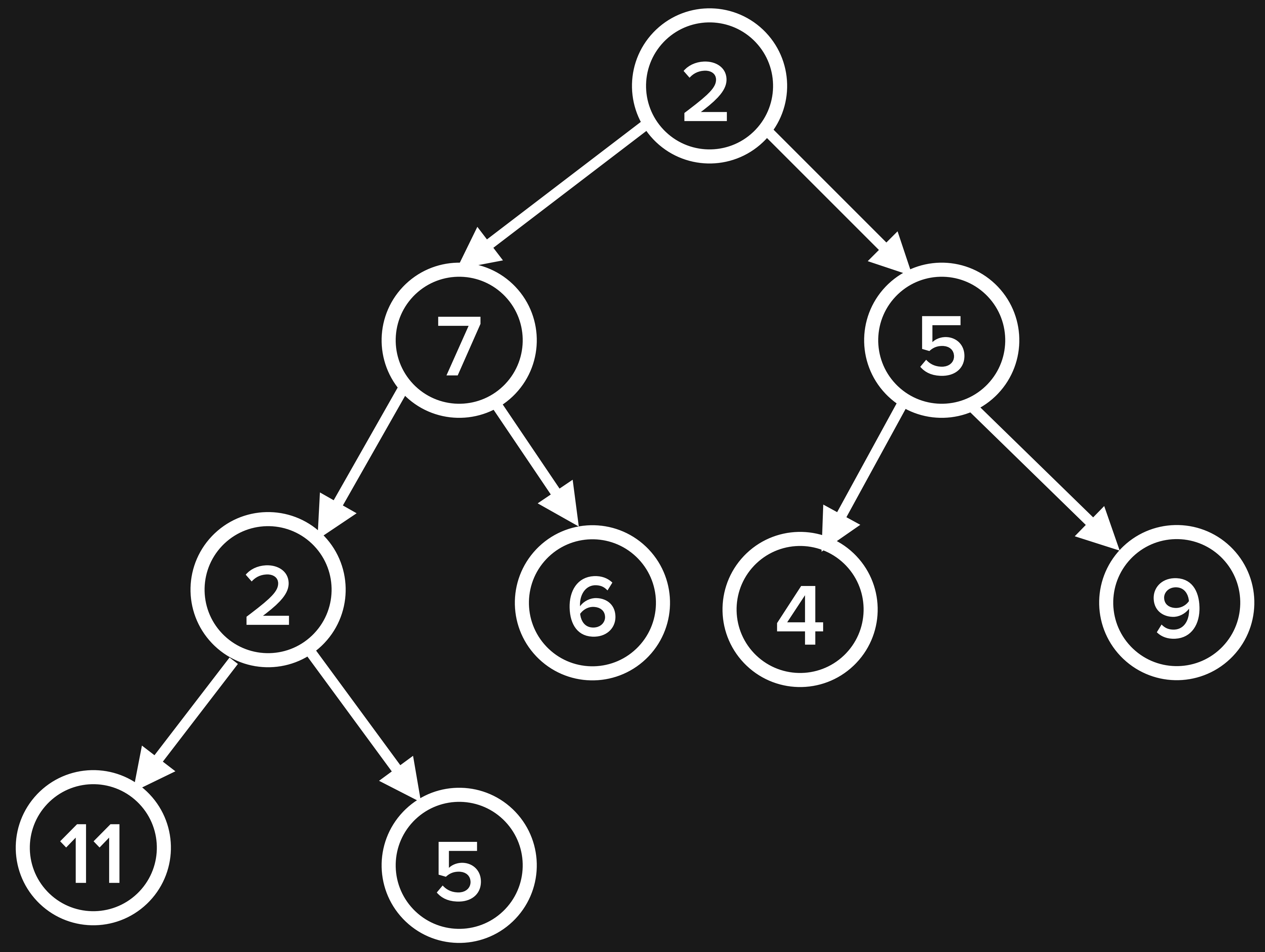
height (node)
number of edges on *longest* downward path from ***node*** to leaf

depth (node)
number of edges between the ***node*** and the root

level (node)
1 + number of edges between the ***node*** and the root

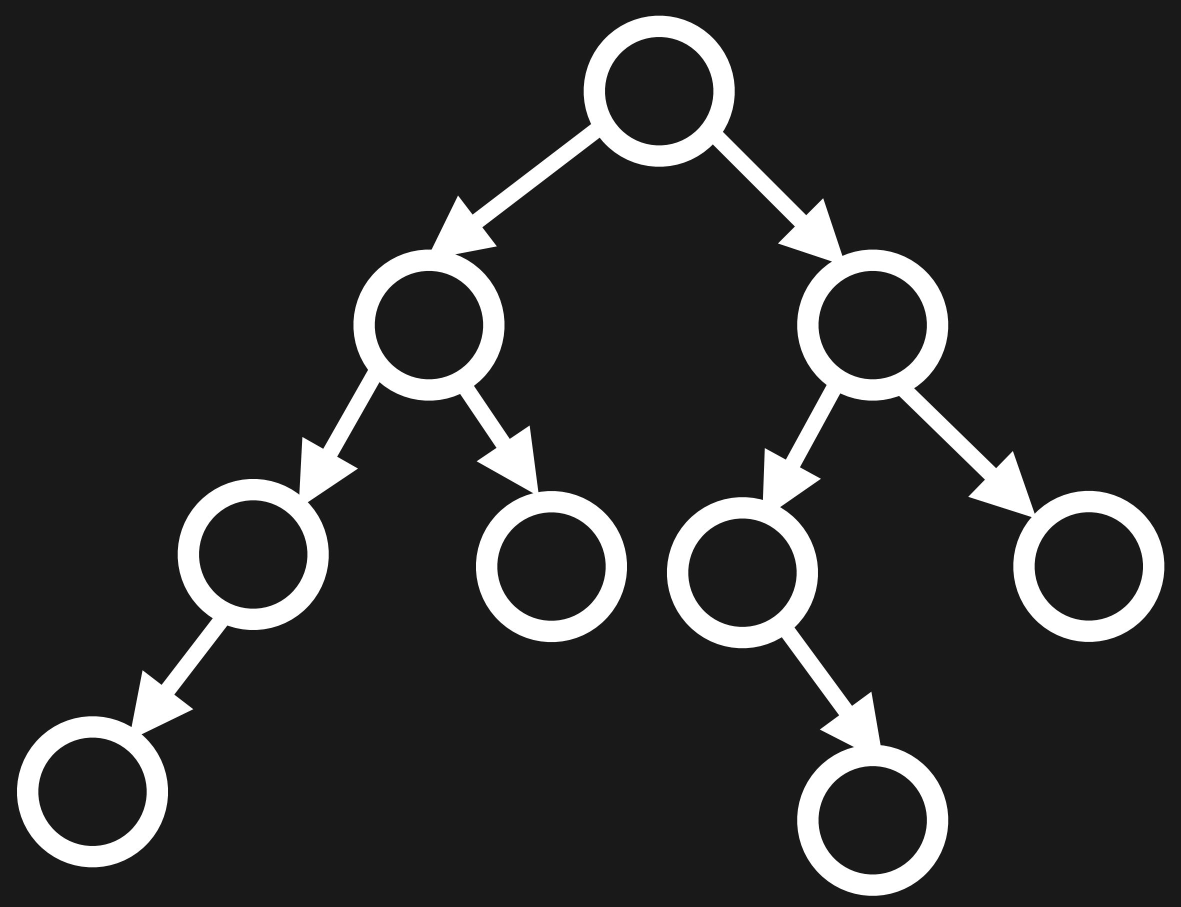
COMPLETE TREE

Every level except possibly last is completely filled and nodes are as far left as possible

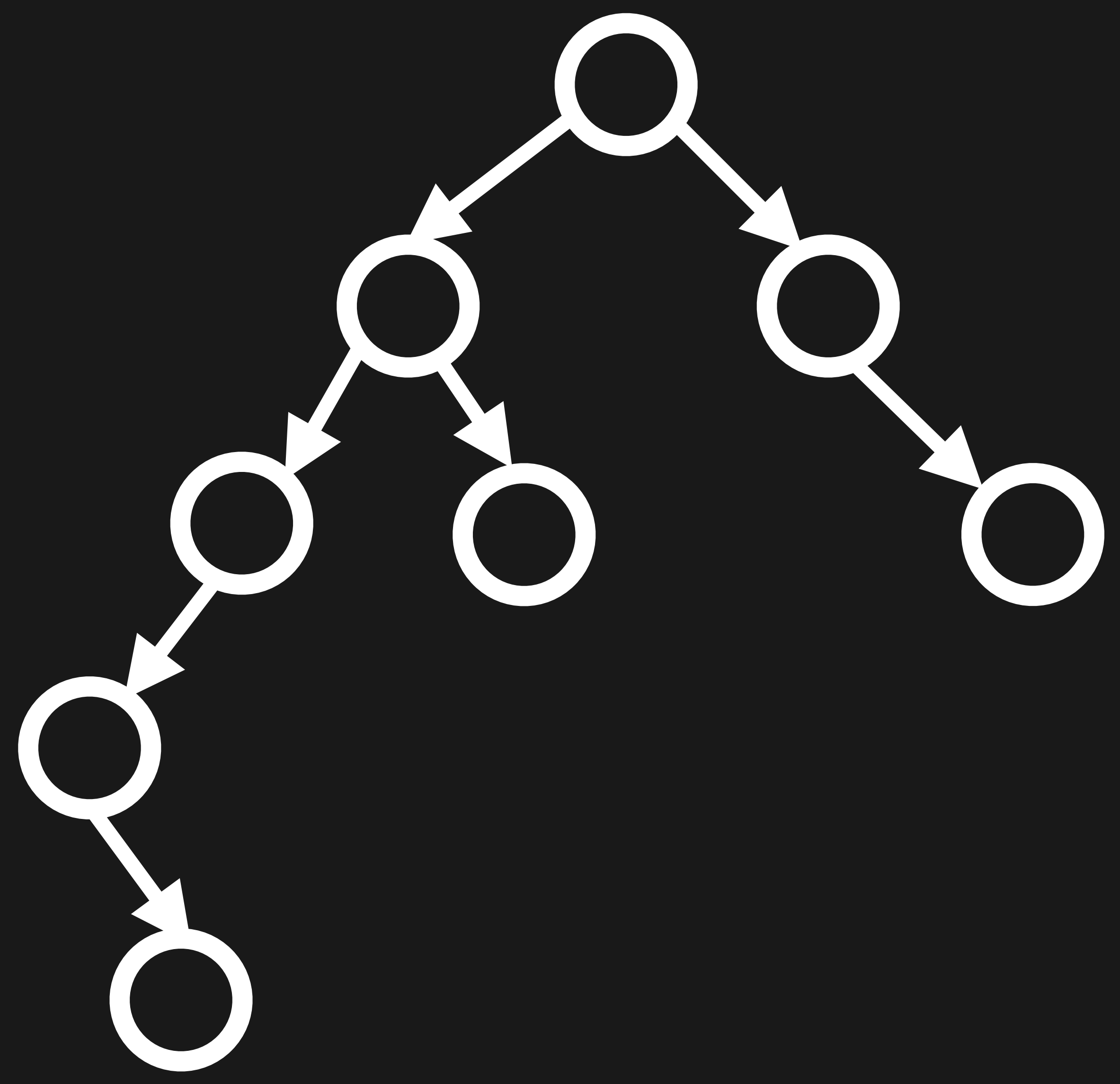


BALANCED TREE

All leaves are at minimum possible depth



Balanced



Unbalanced

BINARY SEARCH TREE

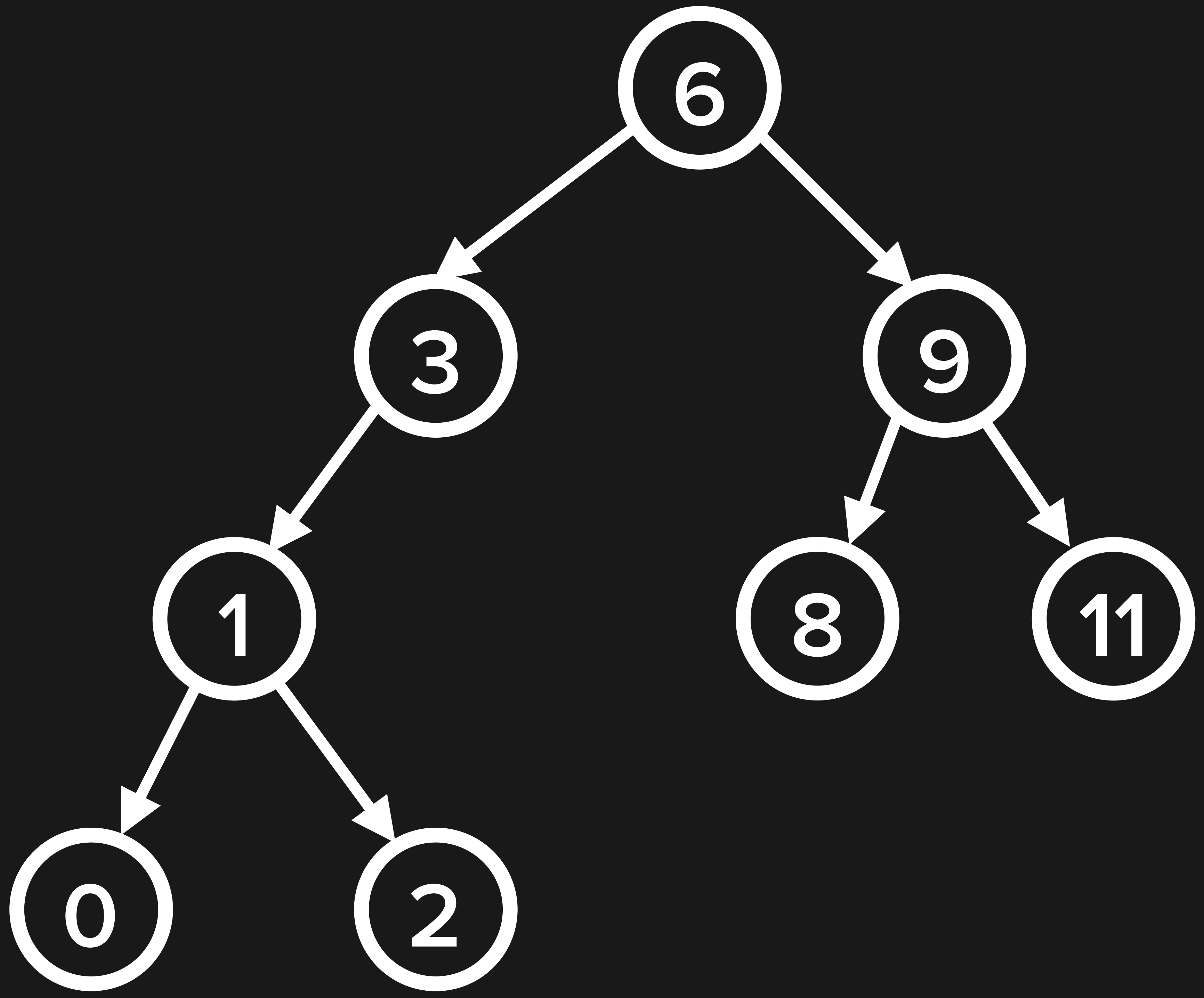
Always sorted

For each node

Left children are smaller

Right children are larger

No duplicate keys (usually)



WHY USE A BST?

Fast search, insertion, deletion - especially when balanced

Sort as you go instead of all at once

Fairly simple implementation for good performance

WHY USE A BST?

Only allocates memory as it's needed

Doesn't have to reallocate memory to grow (like a hash table)

A NOTE ON LOG N

When discussing time and space complexity of algorithms, log n usually means $\log_2 n$

BINARY LOGARITHM

$\log_2 n = x \leftrightarrow 2^x = n$

the power by which 2 must be raised by to obtain n

$\log_2 16 = 4$ $\log_2 32 = 5$ $\log_2 143 = 7.15987$

http://en.wikipedia.org/wiki/Binary_logarithm

WHY IS LOG N GOOD?

Imagine a binary search tree with 2^{32} nodes

2^{32} is 4,294,967,296 (that's over 4 billion!)

When searching, we only have to visit a maximum of 32 nodes to find the node containing the data we're looking for (assuming the tree is perfectly balanced)

SEARCH

```
# Call initially with node == root node
def find_recursive(key, node):
    if node is None or node.key == key:
        return node
    elif key < node.key:
        return find_recursive(key, node.left)
    else:
        return find_recursive(key, node.right)
```

http://en.wikipedia.org/wiki/Binary_search_tree#Searching

INSERTION

Same as search except once you find a node without a child on the next side you're traversing, add it there.

http://en.wikipedia.org/wiki/Binary_search_tree#Insertion

DELETION

Three cases

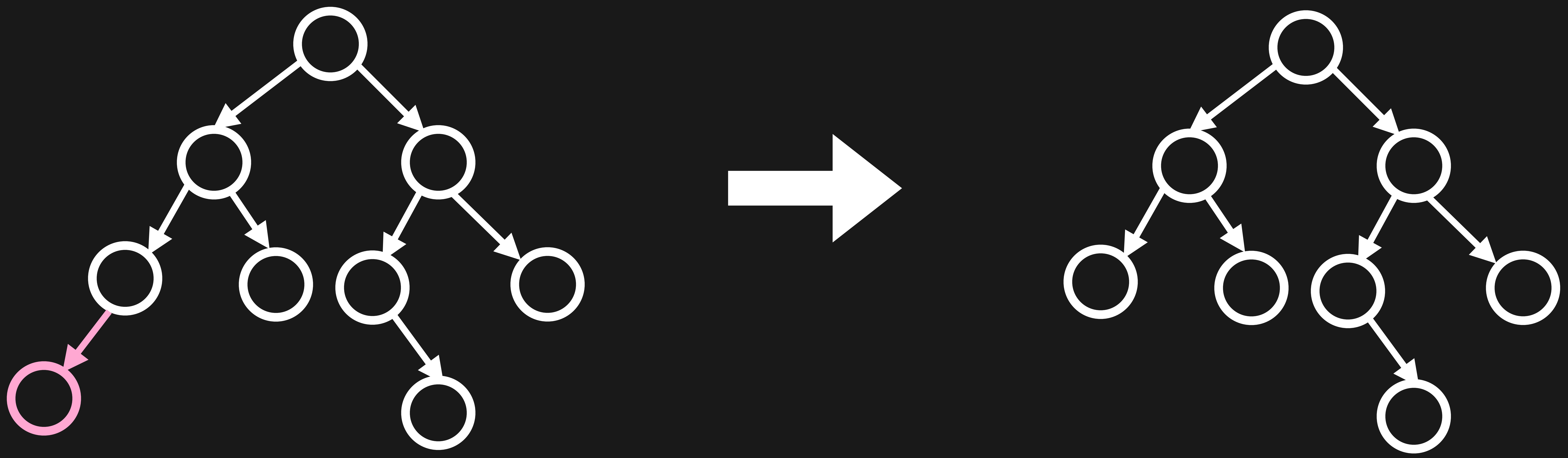
No children

One child

Two children

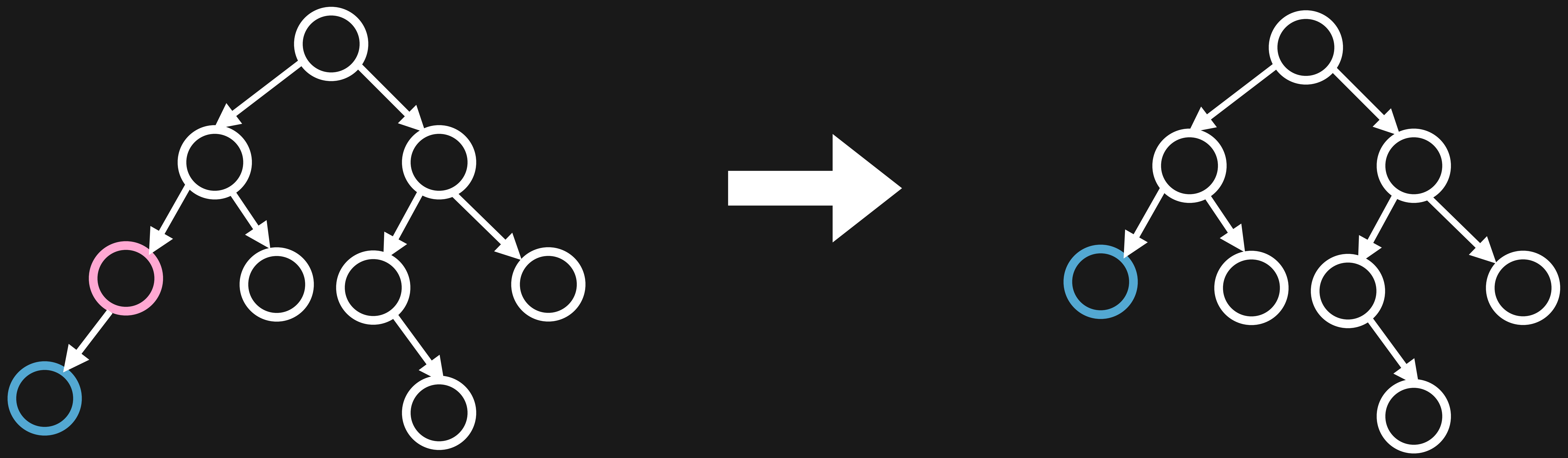
http://en.wikipedia.org/wiki/Binary_search_tree#Deletion

NO CHILDREN



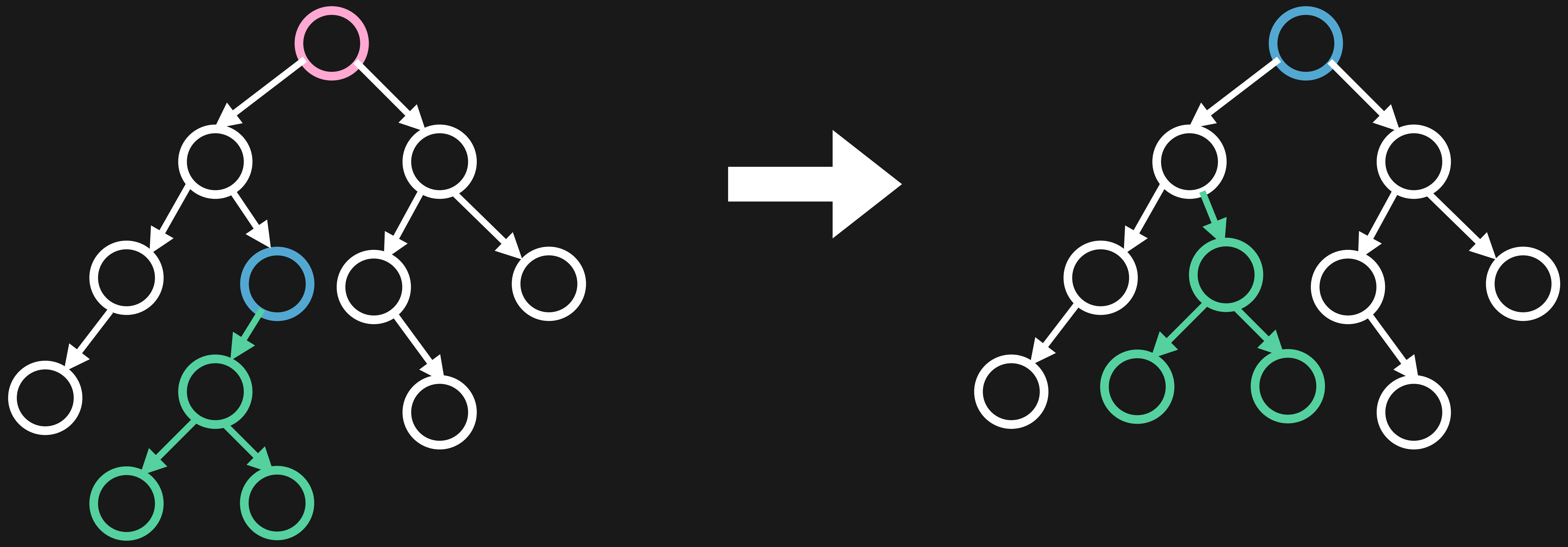
http://en.wikipedia.org/wiki/Binary_search_tree#Deletion

ONE CHILD



http://en.wikipedia.org/wiki/Binary_search_tree#Deletion

TWO CHILDREN



http://en.wikipedia.org/wiki/Binary_search_tree#Deletion

COMPLEXITY

	Average Case	Worst Case
Space	O(n)	O(n)
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)