ACADEMIND

Getting Started with Data Structures

What & Why?

What are “Data Structures”?

Data Structures allow you to manage data

Examples: Arrays, Objects, Maps, Sets

- [1, 2, 3]
- { name: ‘Max’, age: 31 }
- new Map()
  map.set(‘a’, ‘b’)
- new Set()
  set.add(1)

Different Tasks Require Different Data Structures

Ordered List of Data, Duplicates Allowed	Unordered List of Data, No Duplicates Wanted	Key-value Pairs of Unordered Data	Key-value Pairs of Ordered, Iterable Data
Array	Set	Object	Map
`[1, 2, 5, 3]`	`new Set()` `set.add('pizza')`	`{ name: 'Max', age: 31 }`	`new Map()` `map.set('loc', 'Germany')`

Arrays – A Closer Look

`[1, 3, 6, 2]`

- Insertion order is kept
- Element access via index
- Iterable (= you can use the for-of loop)
- Size (length) adjusts dynamically
- Duplicate values are allowed
- Deletion and finding elements can require “extra work”

Sets – A Closer Look

```
new Set()
set.add('pizza')
set.add('burger')
set.add('pizza') // not added
```

Insertion order is **not** stored/ memorized

Element access and extraction via method

Iterable (= you can use the for-of loop)

Size (length) adjusts dynamically

Duplicate values are **not** allowed (i.e. unique values only)

Deletion and finding elements is trivial and fast

Arrays vs Sets

Arrays	Sets
You can always use arrays	Only usable if order does not matter and you only need unique values
Must-use if order matters and/ or duplicates are wanted	Can simplify data access (e.g. finding, deletion) compared to arrays

Objects – A Closer Look

```
{
name: 'Max', age: 31,
greet() { console.log('Hi, I am ' + this.name); }
}
```

Unordered key-value pairs of data

Element access via key (property name)

Not iterable (only with for-in)

Keys are unique, values are not

Keys have to be strings, numbers or symbols

Can store data & "functionality" (methods)

Maps – A Closer Look

```
new Map()
map.set('name', 'Max')
map.set(true, true) // Boolean key
```

Ordered key-value pairs of data

Element access via key

Iterable (= you can use the for-of loop)

Keys are unique, values are not

Keys can be anything (incl. reference values like arrays)

Pure data storage, optimized for data access

Objects vs Maps

Objects	Maps
Very versatile construct and data storage in JavaScript	Focused on data storage and access
Must-use if you want to add extra functionality	Can simplify and improve data access compared to objects

WeakSet & WeakMap

Variations of Set and Map

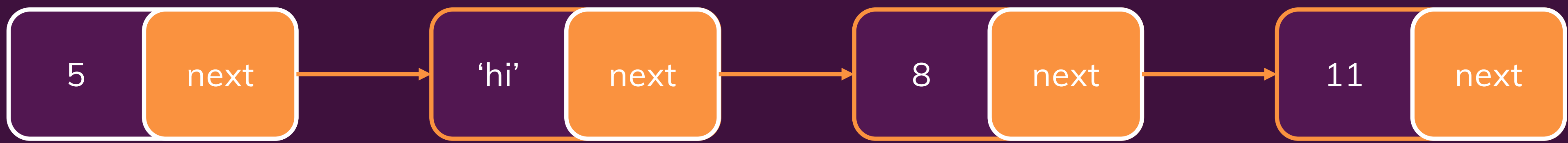
↓

Values and keys are only “weakly referenced”

↓

Garbage collection can delete values and keys if not used anywhere else in the app

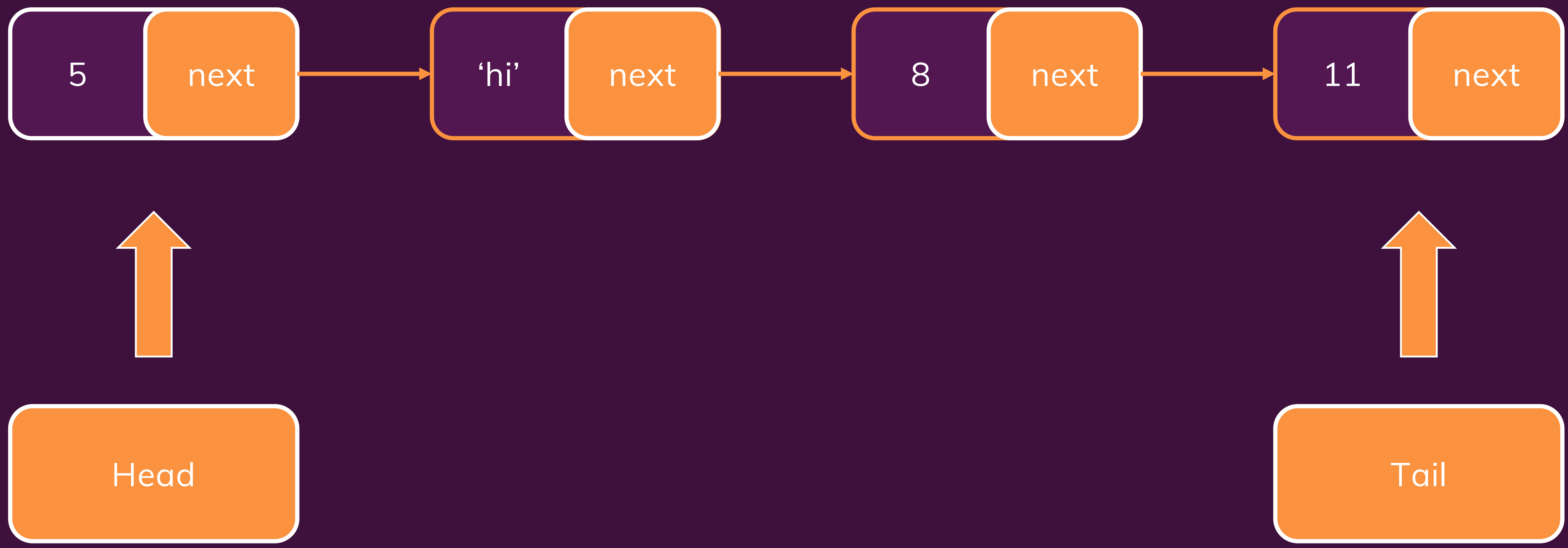
A Custom Data Structure: “Linked List”



Every element knows about the next element

This allows for efficient re-sizing and insertion at start and end of the list

A Custom Data Structure: “Linked List”



Why would you want a “Linked List”?

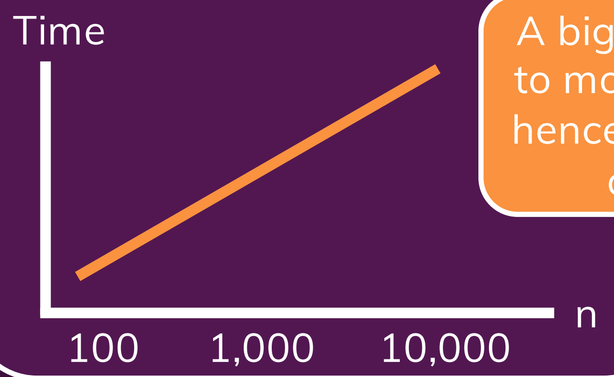
Historically (in other programming languages), the main reason was **memory management**: You didn’t have to specify (occupy) the size in advance

Nowadays, JavaScript has **dynamic arrays** (dynamic re-sizing built in) and memory **isn’t really the primary issue** in JavaScript apps

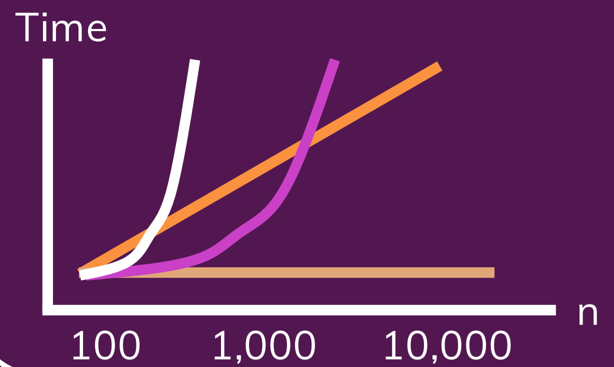
Linked Lists can be useful if you do a lot of **insertions at the beginning of lists** – linked lists are faster than arrays at this

Time Complexity & Big O Notation

```
function sumUp(n) {
  let result = 0;
  for (let i = 1; i <= n; i++) {
    result += i;
  }
  return result;
}
```



A bigger number leads to more loop iterations, hence time increases in a **linear way**.



Linear Time	$O(n)$
Constant Time	$O(1)$
Quadratic Time	$O(n^2)$
Cubic Time	$O(n^3)$

We care about the **trend/ kind of function**.

Big O Notation

Linked List Time Complexity & Arrays

	Linked List	Arrays
Element Access	O(n)	O(1)
Insertion at End	With tail: O(1) Without tail: O(n)	O(1)
Insertion at Beginning	O(1)	O(n)
Insertion in Middle	Search time + O(1)	O(n)
Search Elements	O(n)	O(n)

Data Structures & Algorithms

Algorithms	Data Structures
Solve problems	Help with solving problems
Functions that produce result X for input Y	Data storages that can be used as part of algorithms
You write algorithms all the time!	You work with data structures (e.g. arrays) all the time but you don't need custom ones all the time

List & Table Structures

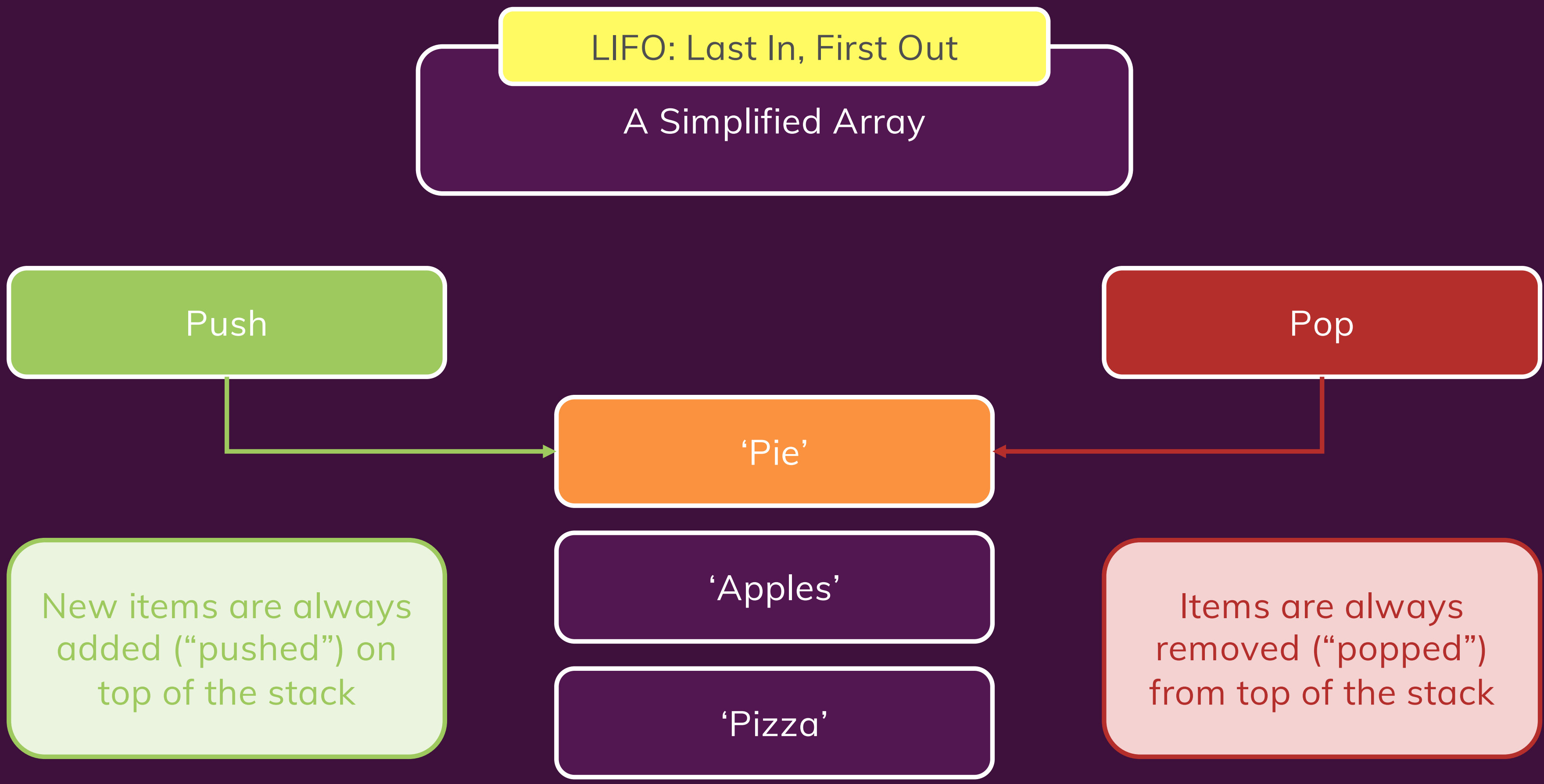
Arrays & Objects “on Steroids”

ACADEMIND

What are “List & Table Structures”?

Lists	Tables
Collections of Values	Collections of Key-Value Pairs
e.g. Arrays, Sets, LinkedLists	e.g. Objects, Maps
Great for storing values that are retrieved by position (via index or search) Also great for loops	Great for storing values that are retrieved by key Not primarily focused on loops

Stack


LIFO: Last In, First Out
A Simplified Array
Push
Pop
'Pie'
'Apples'
'Pizza'
New items are always added ("pushed") on top of the stack
Items are always removed ("popped") from top of the stack

Stack Time Complexity & Arrays

	Stacks	Arrays
Element Access	O(1) But limited to “top element”	O(1)
Insertion at End	O(1)	O(1)
Insertion at Beginning	O(n) With “Data Loss”	O(n)
Insertion in Middle	O(n) With “Data Loss”	O(n)
Search Elements	O(n) With “Data Loss”	O(n)

Queue

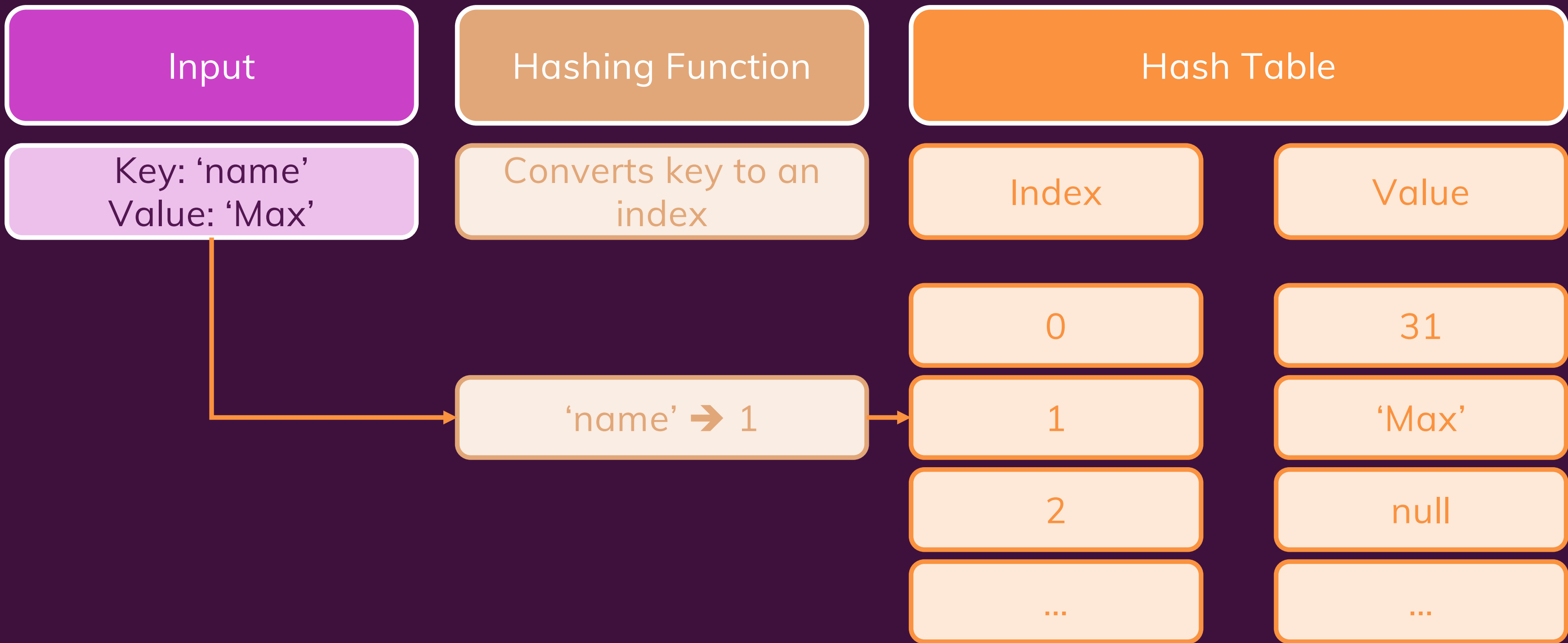
FIFO: First in, First Out
A Simplified Array
Dequeue
1.19
5.89
-3.19
59.83
0.55
Enqueue

Queue Time Complexity & Arrays

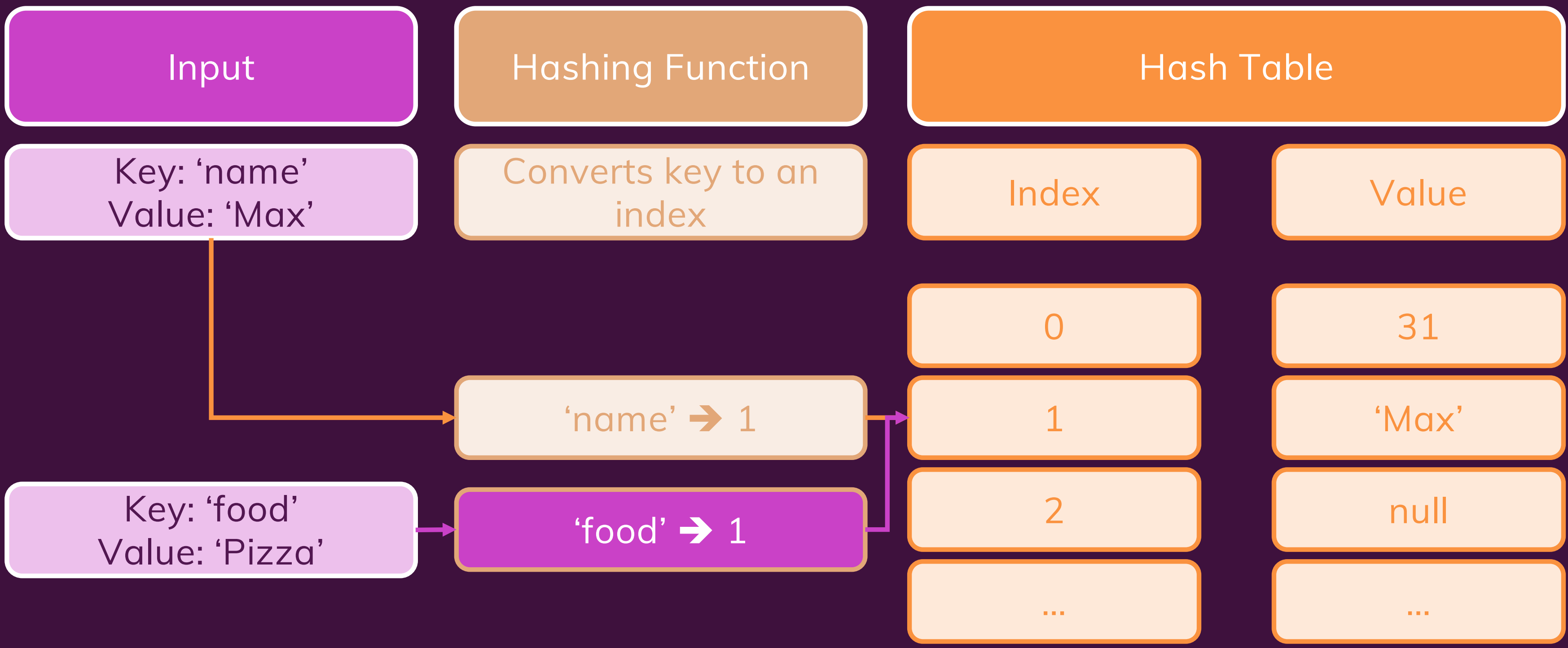
	Queues	Arrays
Element Access	O(1) But limited to “first element”	O(1)
Insertion at End	O(n) With “Data Loss”	O(1)
Insertion at Beginning	O(1)	O(n)
Insertion in Middle	O(n) With “Data Loss”	O(n)
Search Elements	O(n) With “Data Loss”	O(n)

Hash Table

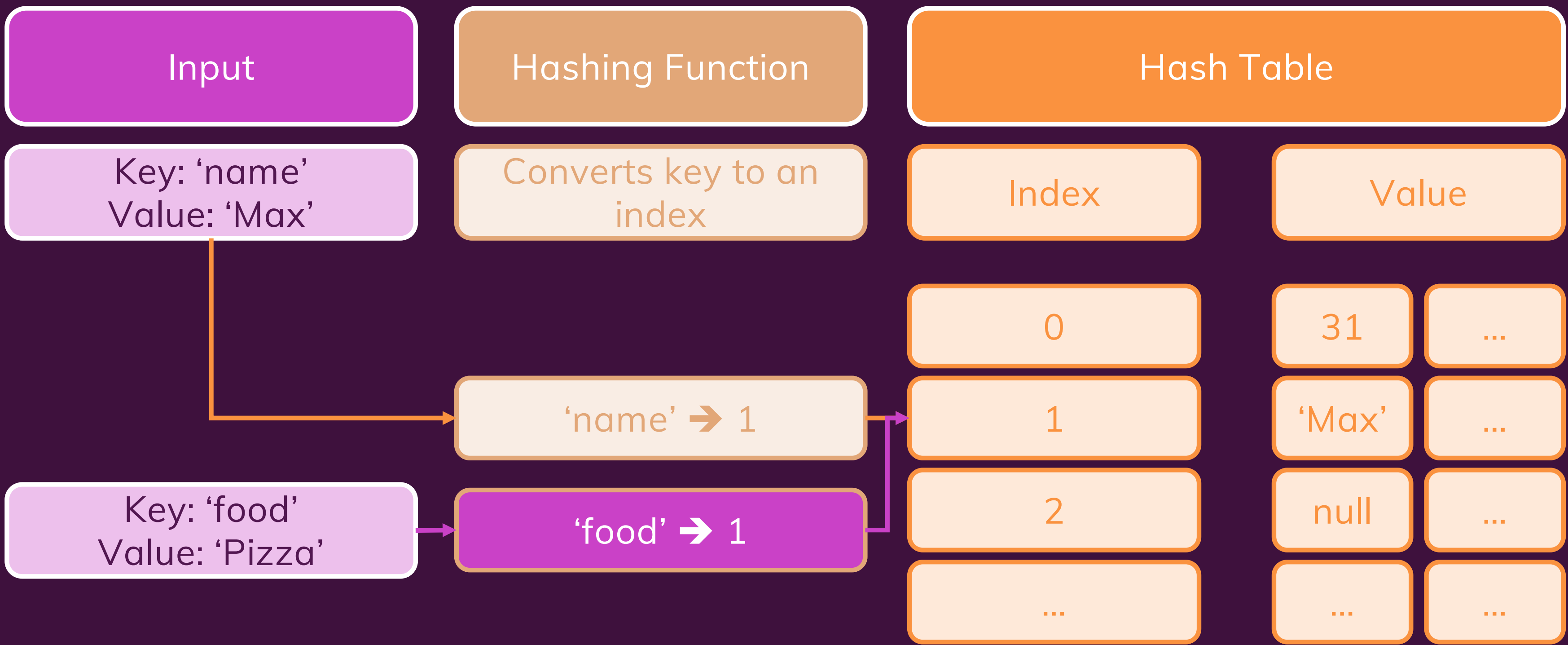
The existing JavaScript “object” is implemented as a Hash Table!



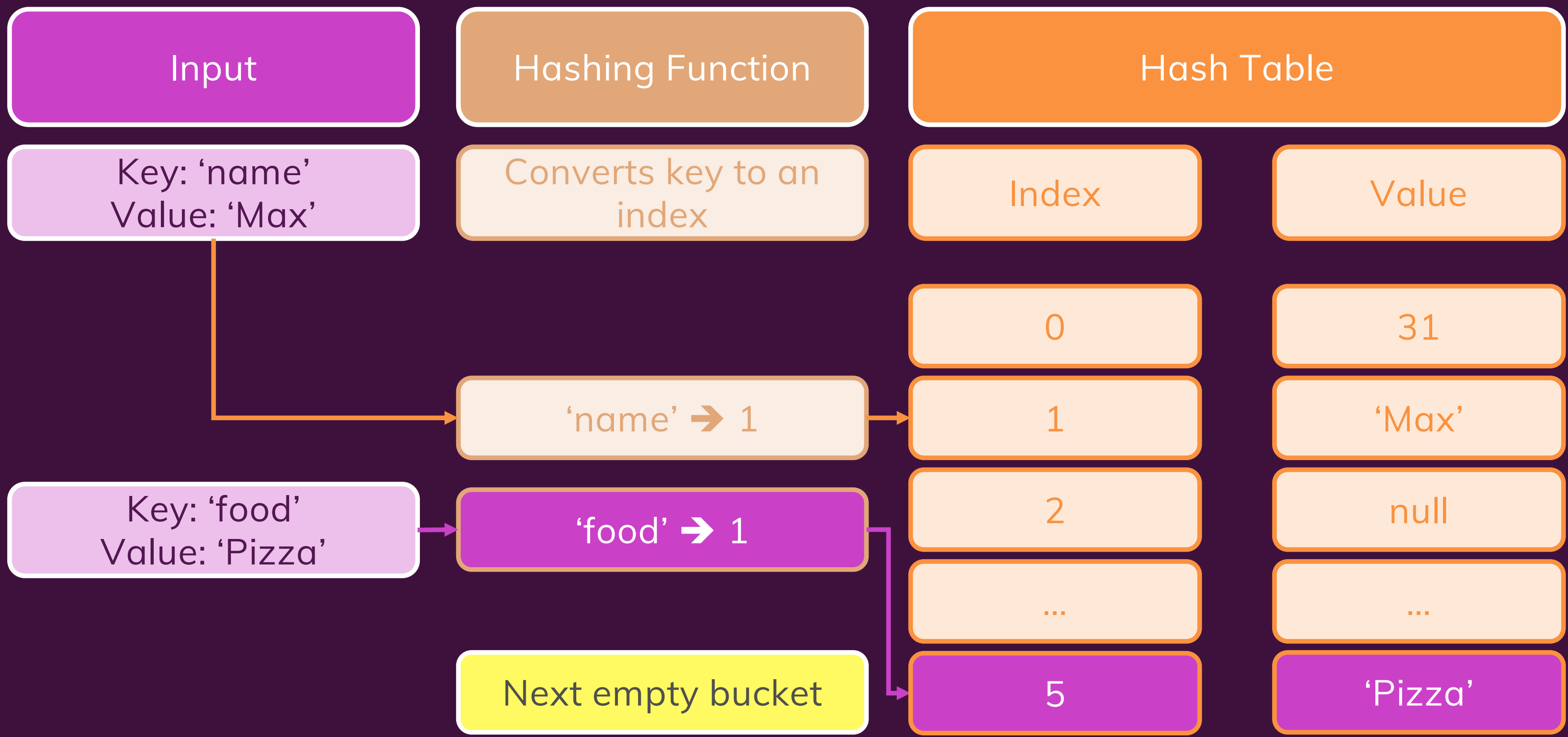
Collisions


Input
Hashing Function
Hash Table
Key: 'name'
Value: 'Max'
Converts key to an index
Index
Value
0
31
'name' → 1
1
'Max'
Key: 'food'
Value: 'Pizza'
'food' → 1
2
null
...
...

Resolving Collisions with “Chaining”



Resolving Collisions with “Open Addressing”



Hash Table Time Complexity & Arrays & Objects

	Hash Tables	Arrays	Objects
Element Access	O(1) in theory O(n) with lots of hash collision	O(1)	O(1)
Insertion at End	O(1) O(n) with lots of hash collision	O(1)	O(1)
Insertion at Beginning	O(1) O(n) with lots of hash collision	O(n)	O(1)
Insertion in Middle	O(1) O(n) with lots of hash collision	O(n)	O(1)
Search Elements	O(1) O(n) with lots of hash collision	O(n)	O(1)

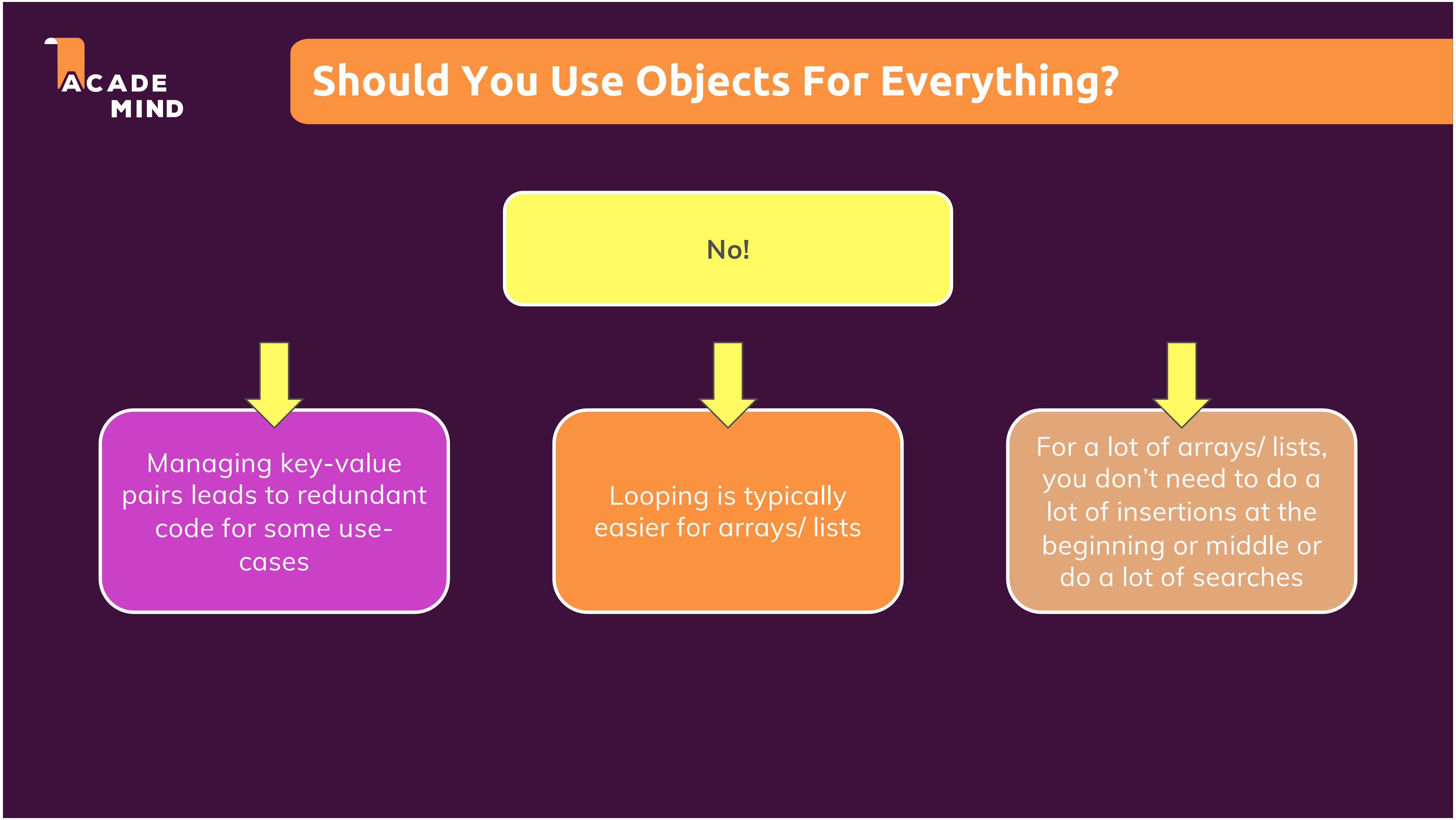
Hash Tables vs Objects

The existing JavaScript “object” is implemented as a Hash Table!

You don’t really need to build your own Hash Tables in JavaScript!

In other programming languages, you might not have the built-in “object” data structure

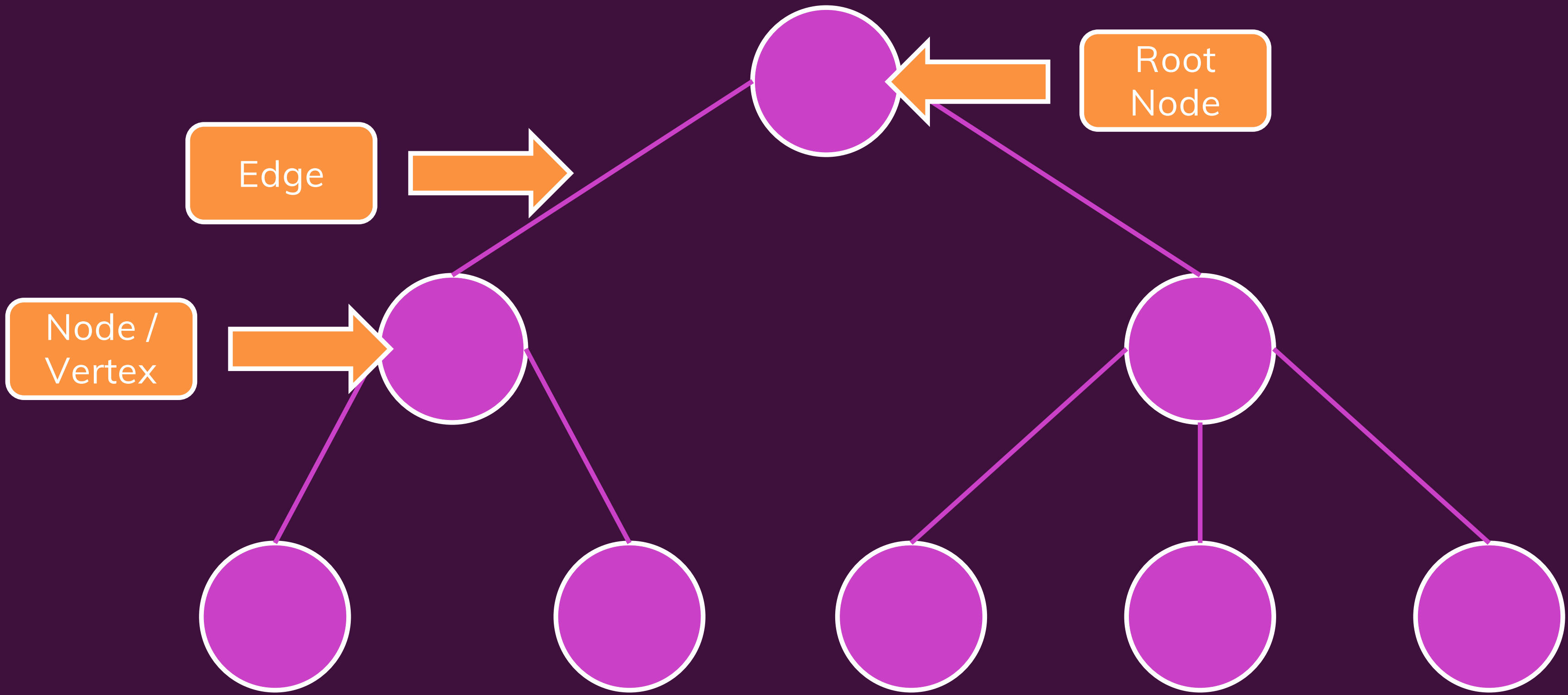
It always helps to understand how the language works internally


Should You Use Objects For Everything?
No!
Managing key-value pairs leads to redundant code for some use-cases
Looping is typically easier for arrays/ lists
For a lot of arrays/ lists, you don't need to do a lot of insertions at the beginning or middle or do a lot of searches

Tree Structures

Adding Depth

What are "Tree Structures"?



A **unidirectional**, **non-linear** data structure with **edges** that connect **vertices** (nodes). There is a **root node** and there **are no cycles** (loops).

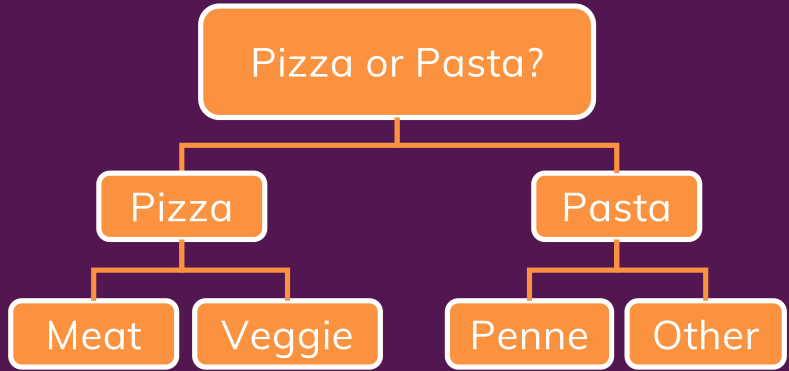
Examples for Trees

The Browser DOM

```
<html>
  <head>…</head>
  <body>
    <h2>Welcome!</h2>
    <p>This is a tree!</p>
  </body>
</html>
```

One root element (<html>) and then any amount of nested child elements (e.g. <p>)

A Decision Tree

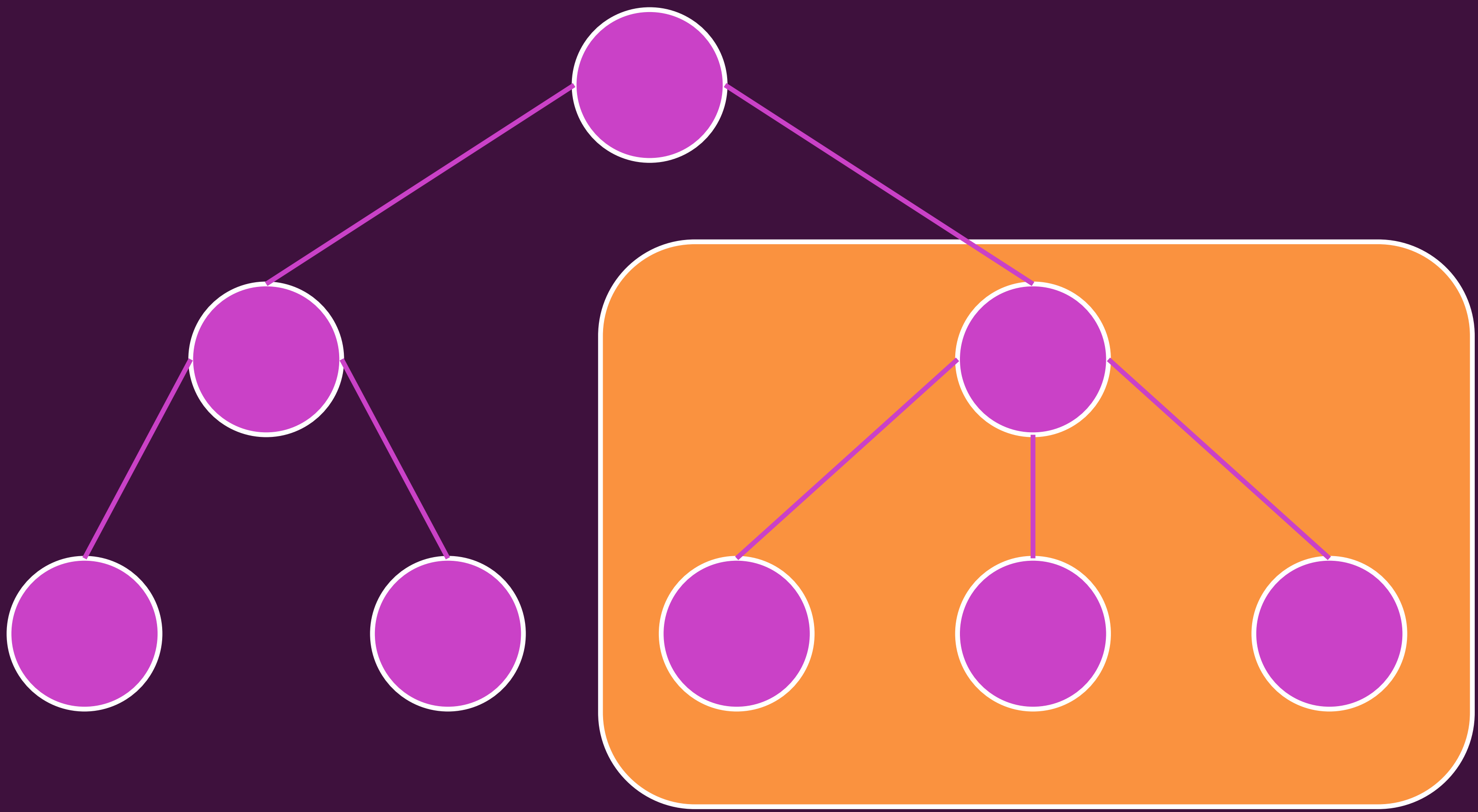


A couple of decisions where every choice leads to new possible decisions (until an outcome is reached)

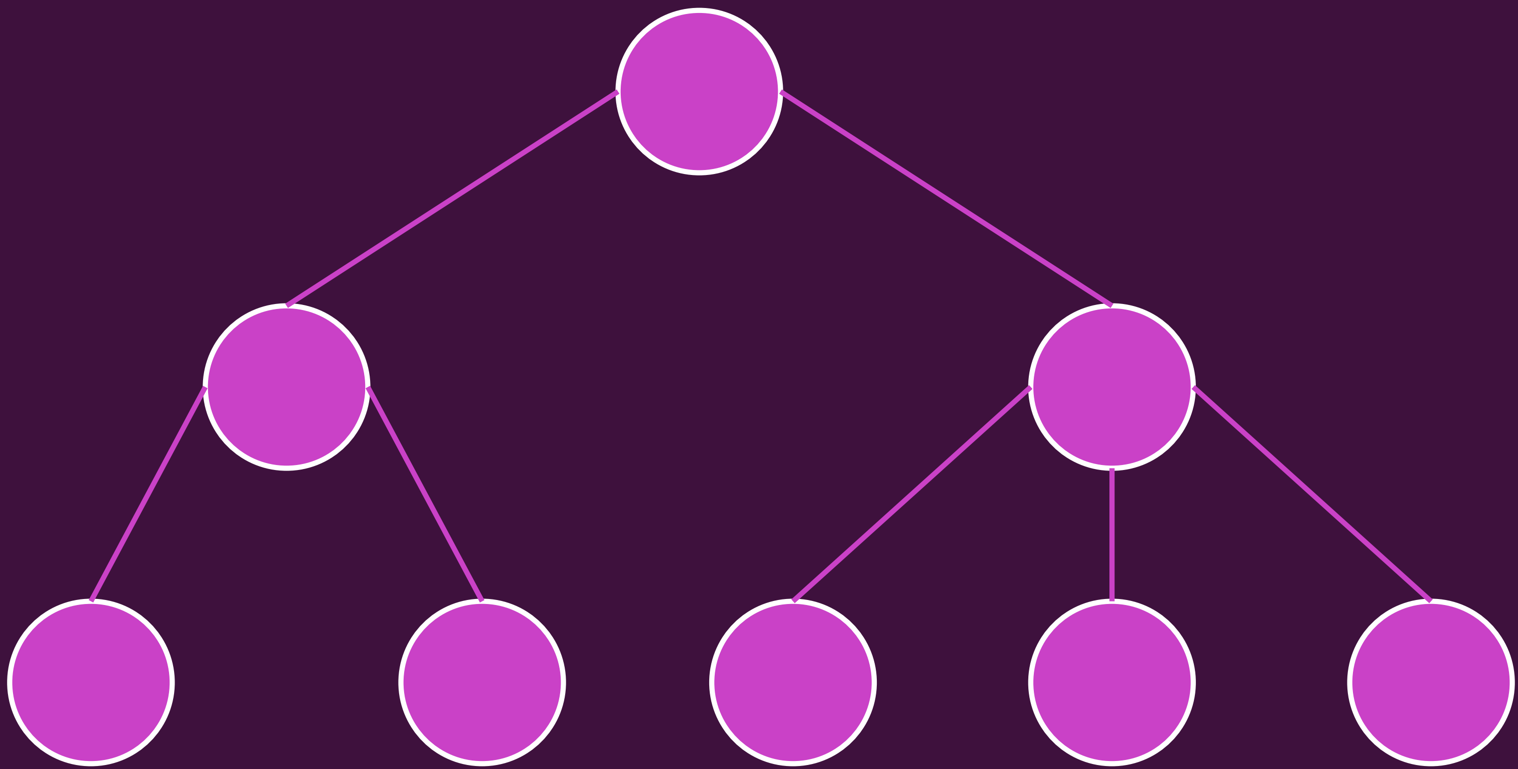
Important Terminology (1/2)

Node / Vertex	A structure that contains a value	Path	A sequence of nodes and edges that connects two nodes
Edge	A connection between two nodes	Distance	The number of edges between two nodes
Root Node	The top-most node in the tree	Parent / Child	Two directly connected nodes, parent node is “above” child node
Sub Tree	A nested tree (i.e. sub tree root node is NOT main tree root node)	Ancestor / Descendant	Two nodes that are connected by multiple parent-child paths
Leaf	A node without any child nodes (i.e. without a sub tree)	Sibling	Two adjacent nodes with the same parent

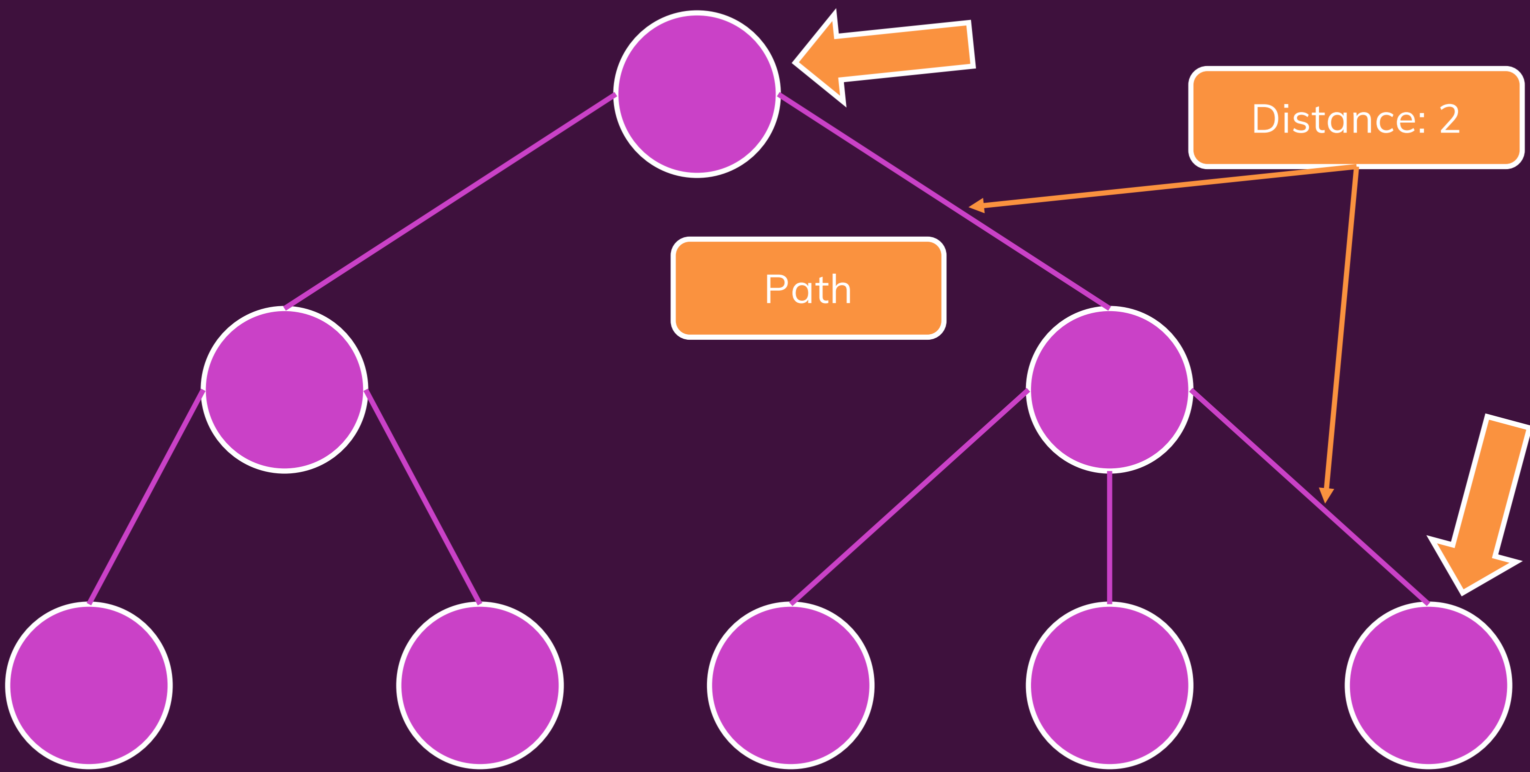
Sub Tree



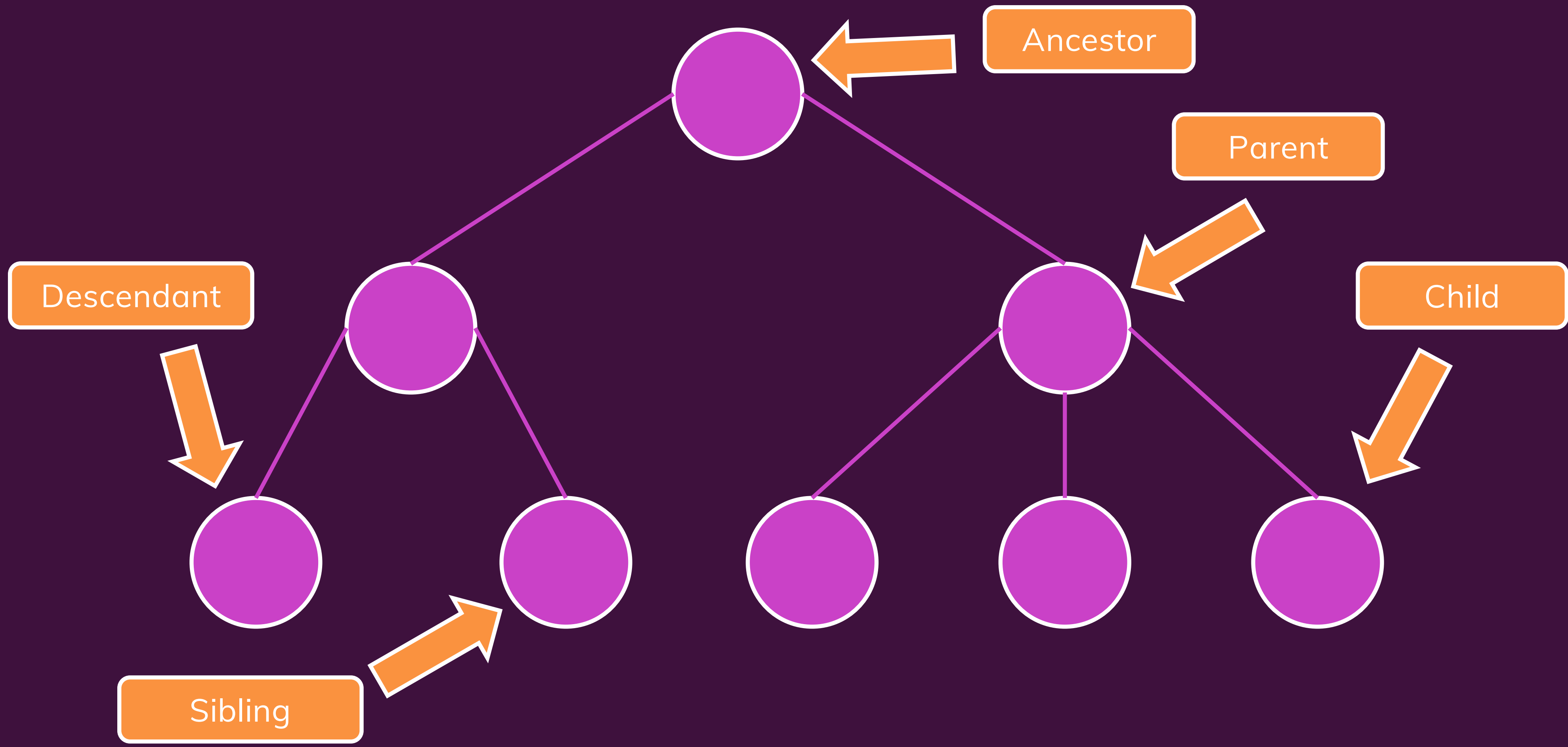
Leaf



Path & Distance



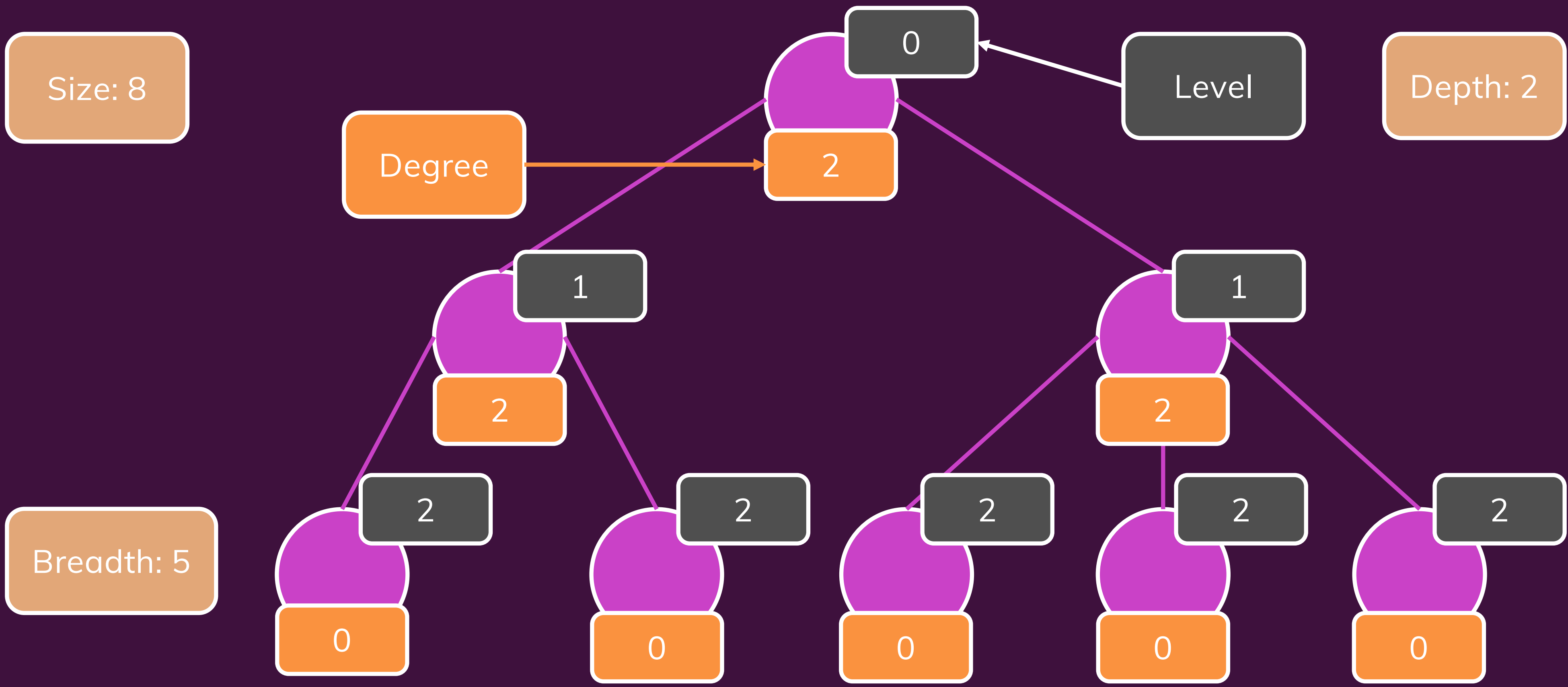
Parent, Child, Ancestor, Descendant & Sibling



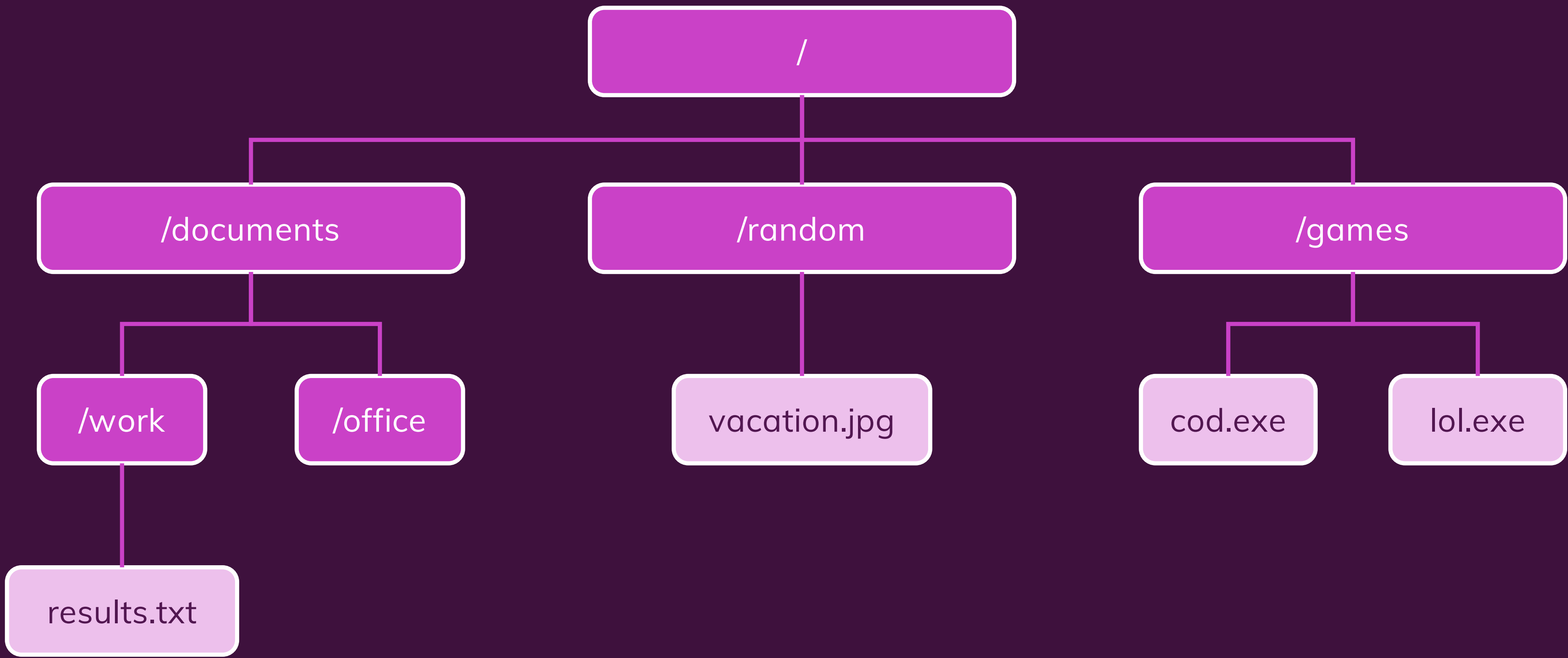
Important Terminology (2/2)

Degree	The number of child nodes of a given node
Level	The distance between a node and the root node
Depth	The maximum level in a tree
Breadth	The number of leaves in a tree
Size	The total number of nodes in a tree

Degree, Level, Depth, Breadth & Size



Example: A Filesystem



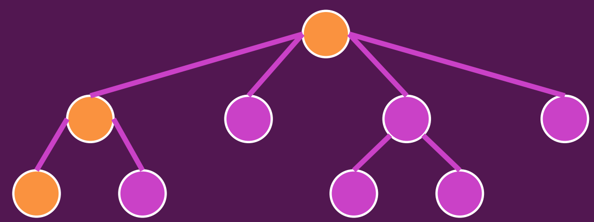
Tree Time Complexity

	Tree	Array
Access / Search	Worst Case: O(n)	O(1) (with Index) O(n) (Search)
Insertion	Worst Case: O(n)	O(1) (at end) O(n) (at beginning)
Removal	Worst Case: O(n)	O(1) (at end) O(n) (at beginning)

Traversing a Tree

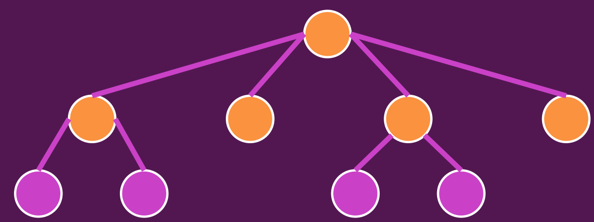
Depth-First

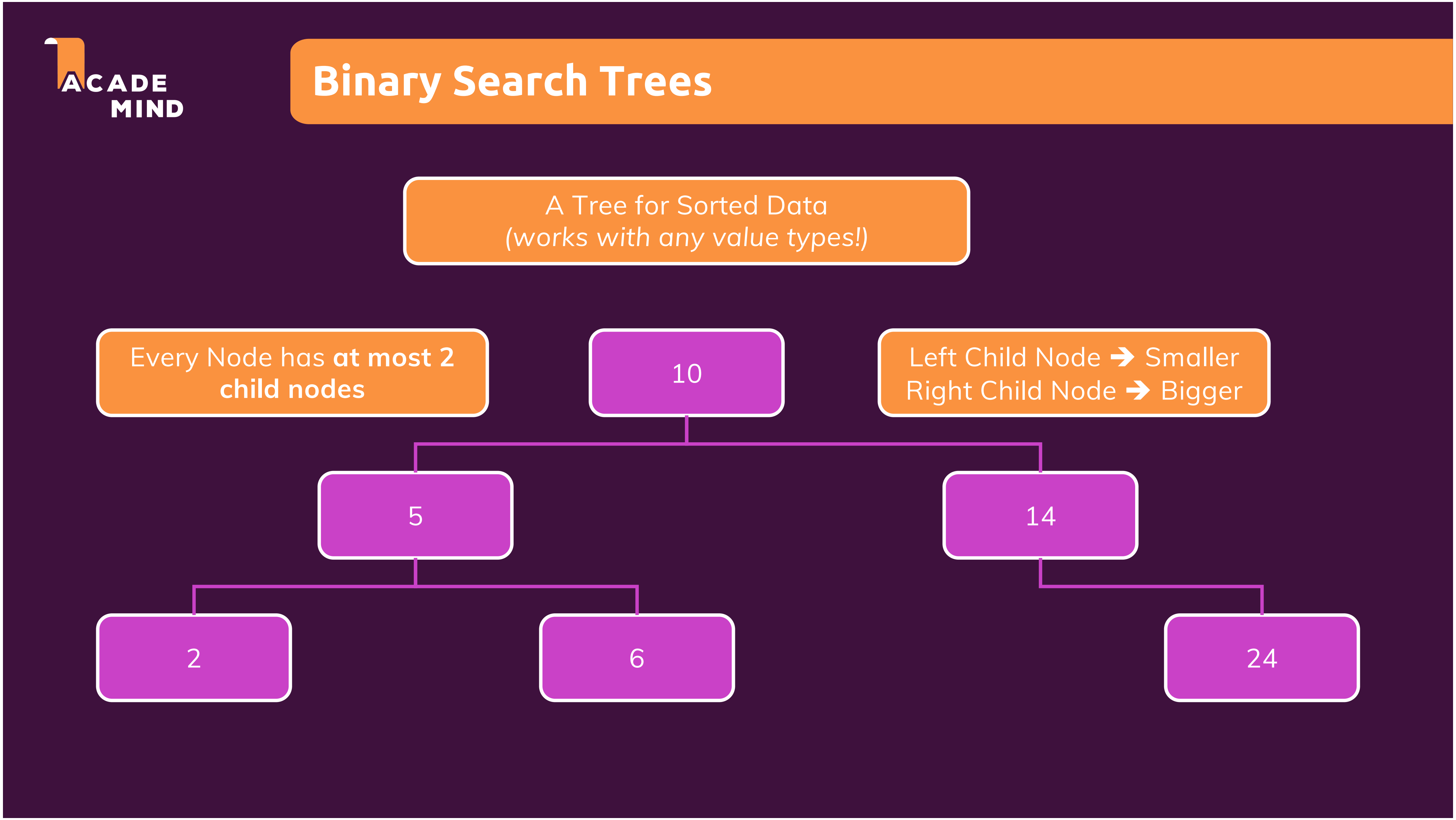
Dig into the tree first and explore sibling trees step by step



Breadth-First

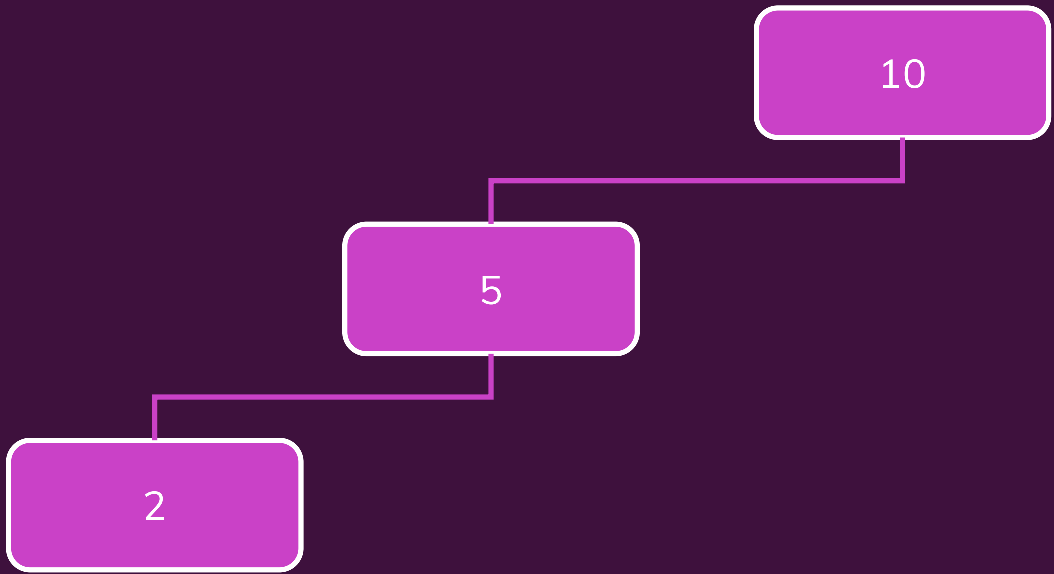
Evaluate all sibling values first before you dig into the tree in depth




ACADE MIND
Binary Search Trees
A Tree for Sorted Data
(works with any value types!)
Every Node has at most 2 child nodes
10
Left Child Node → Smaller
Right Child Node → Bigger
5
14
2
6
24

Binary Search Trees – Worst Case

Worst Case: Only one “line of Nodes” & Looking for “2”

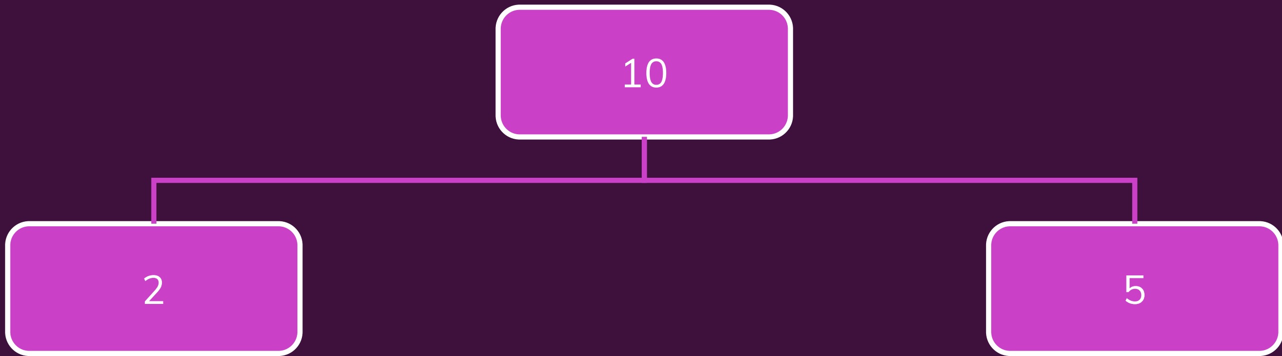


Binary Search Tree Time Complexity

	BST	Array
Access / Search	Worst Case: O(n) Average Case: O(log n)	O(1) (with Index) O(n) (Search)
Insertion	Worst Case: O(n) Average Case: O(log n)	O(1) (at end) O(n) (at beginning)
Removal	Worst Case: O(n) Average Case: O(log n)	O(1) (at end) O(n) (at beginning)

Fixing the BST Worst Case via Balancing

Subtrees should have a **depth** that is **equal or differs by at most 1**



This is called "AVL Tree"

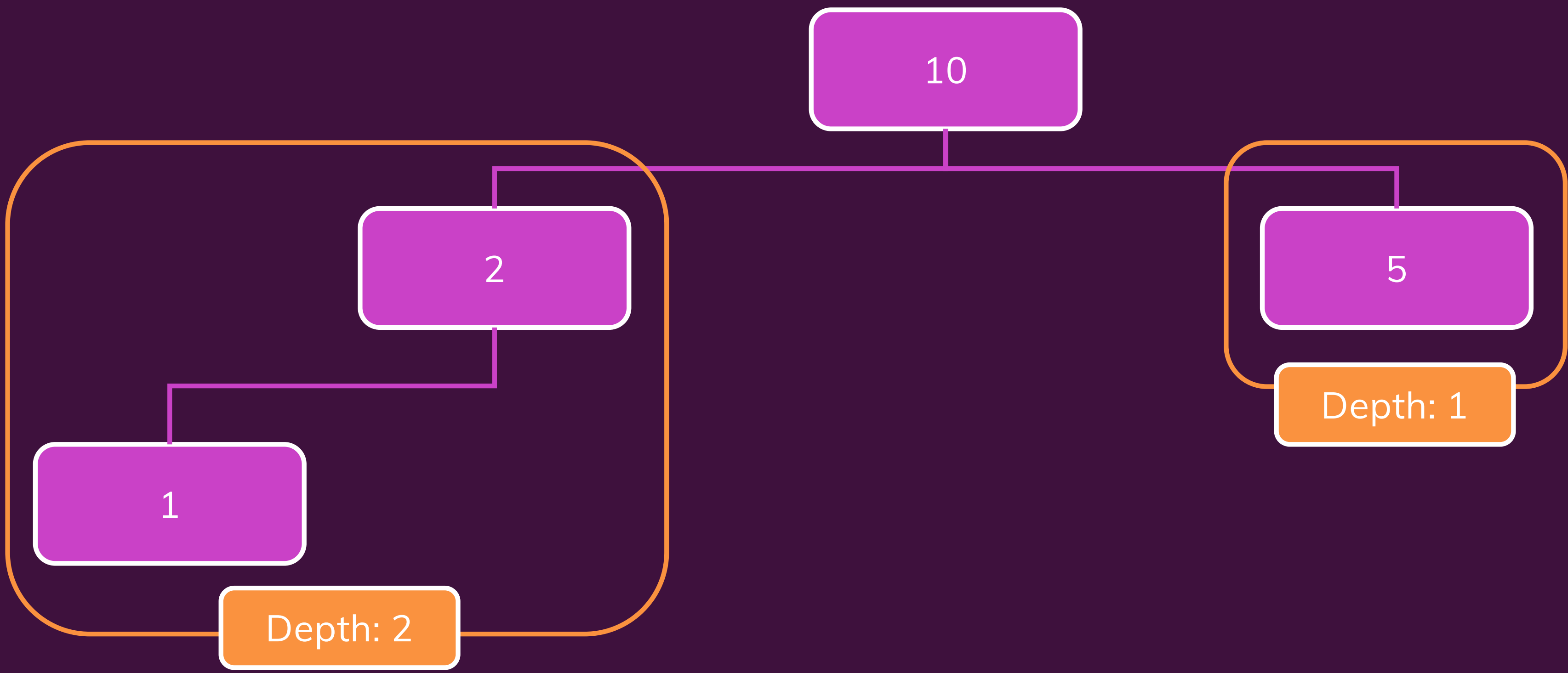
AVL?

Georgy **A**delson-**V**elsky

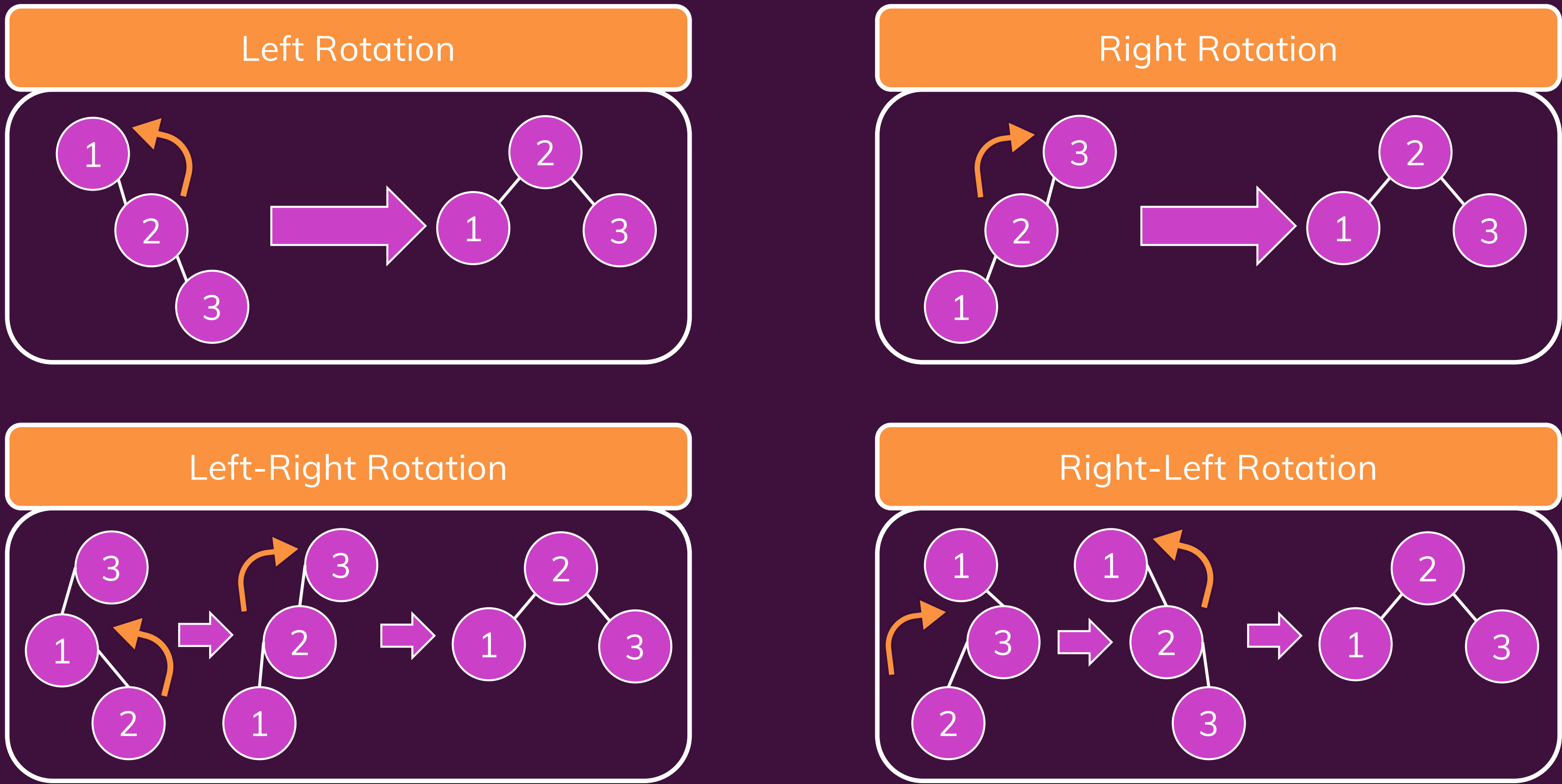
Evgenii **L**andis

A Valid AVL Tree

This is an AVL tree because subtree depth only differs by 1

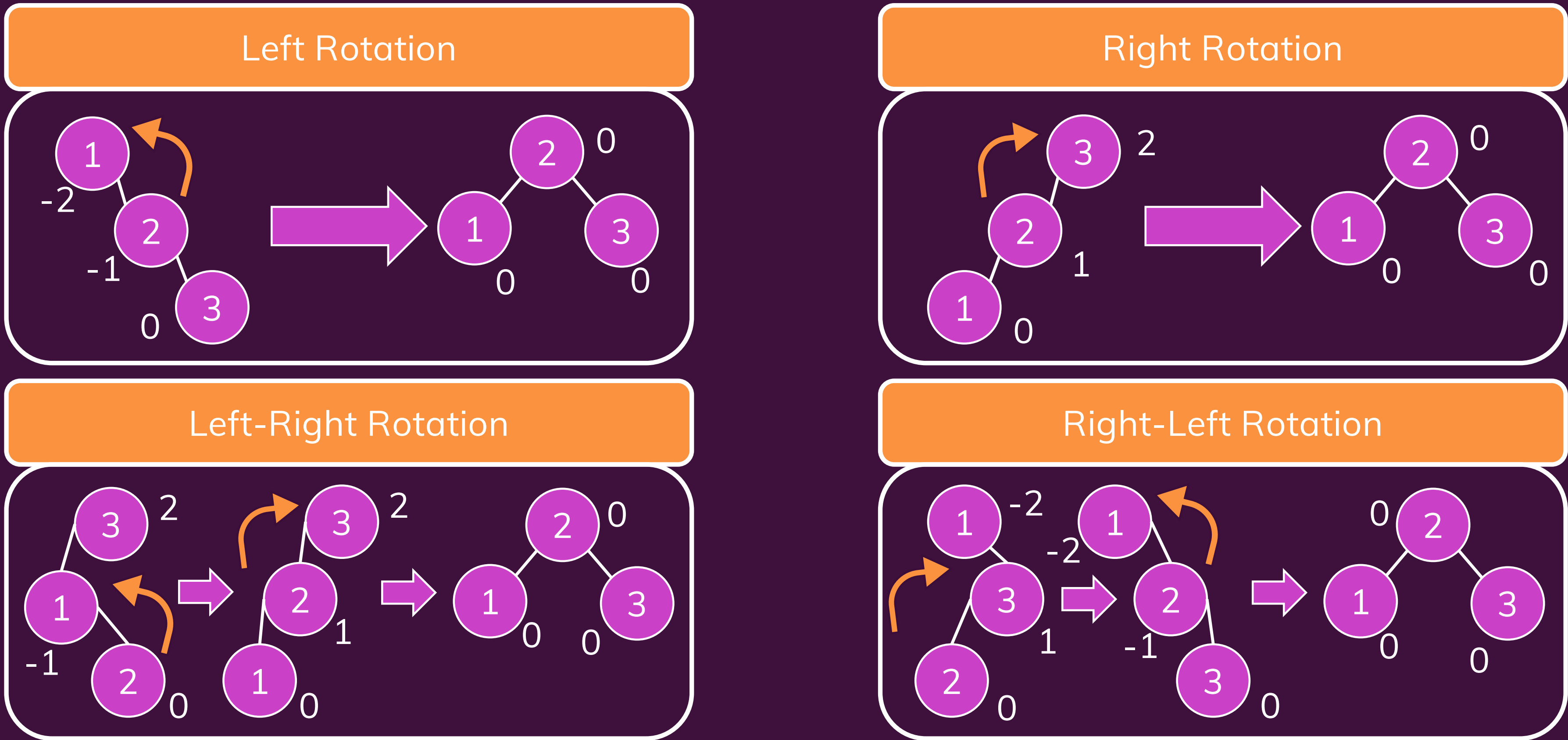


Balancing AVL Trees

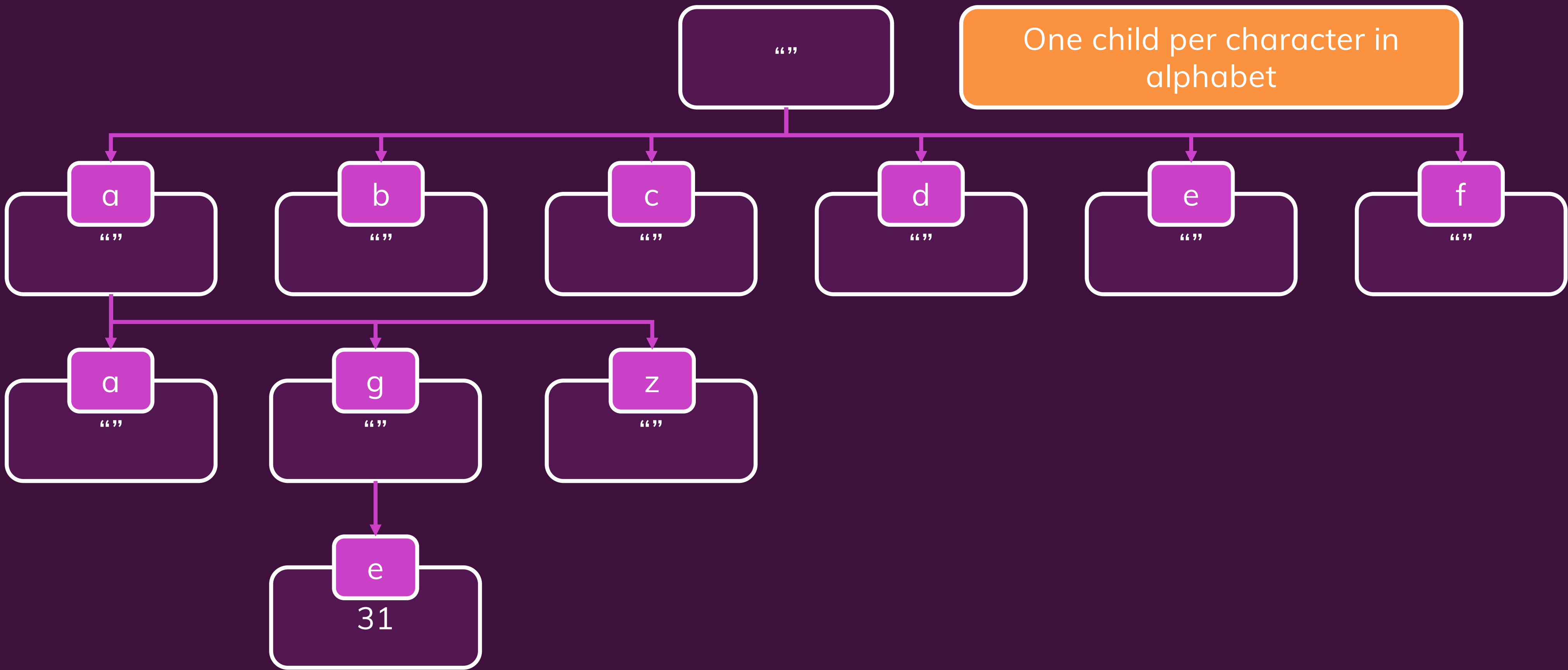


Balance Factors

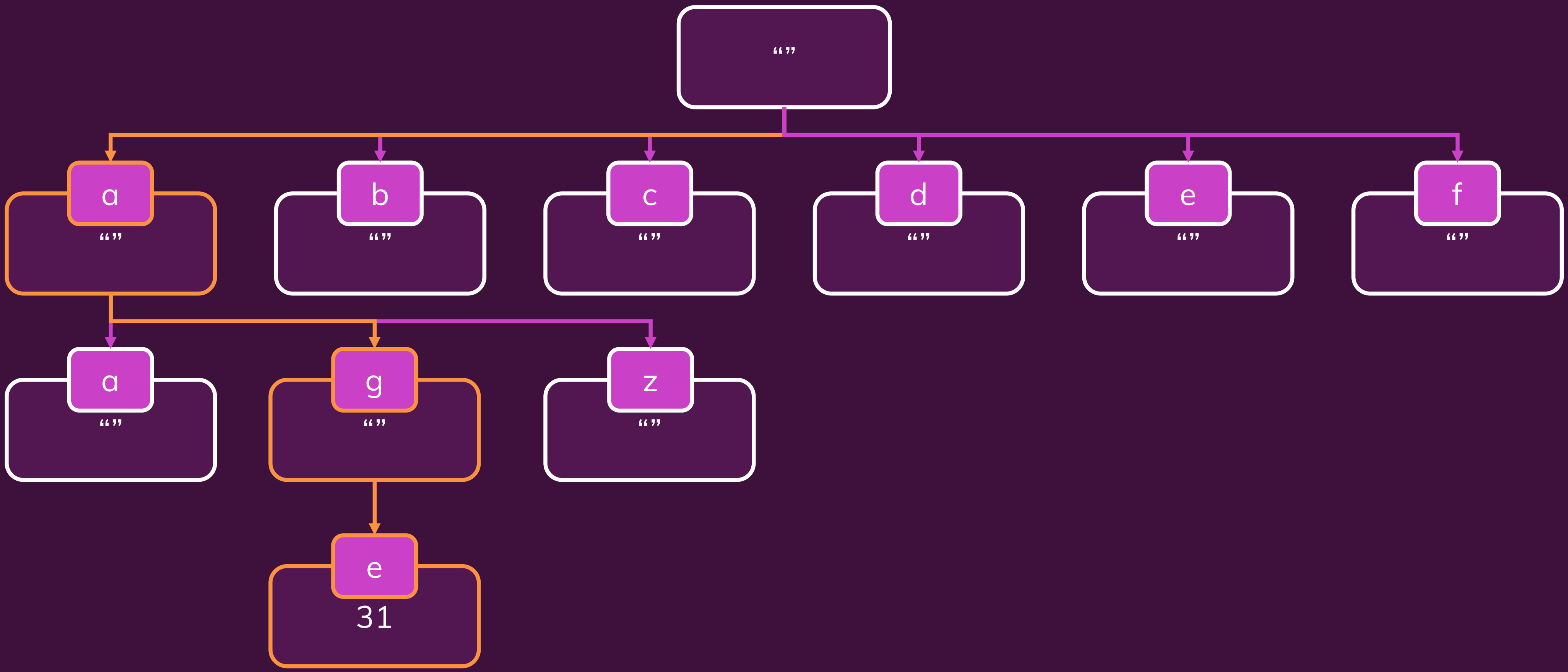
Balance Factor: Difference between subtree depths (left – right)



Tries



Tries



Tries – Time Complexity & Hash Table Comparison

Operation	Tries	Hash Tables
Insert	O(n)	O(n) (with hash collisions)
Find	O(n)	O(n) (with hash collisions)
Delete	O(n)	O(n) (with hash collisions)
Space Complexity	O(n*k)	O(n)

Heaps & Priority Queues

Trees with a Twist

Refresher: Queues

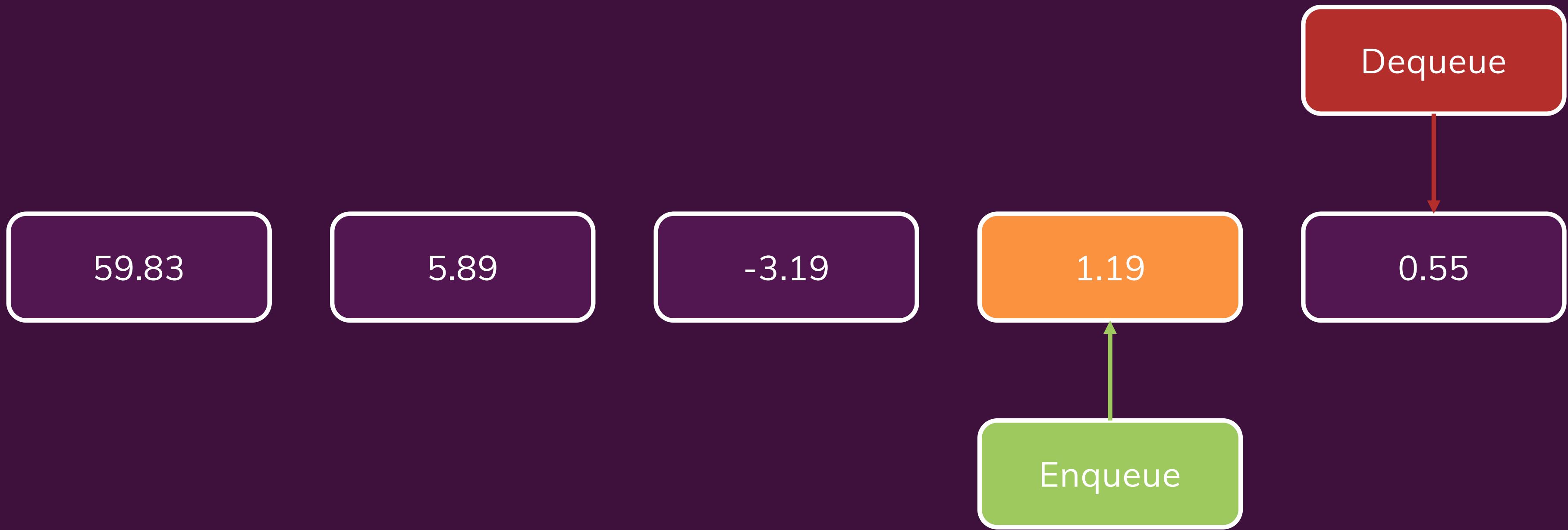
FIFO: First in, first out

Priority Queues

Prioritize items in a queue, **instead** of processing them one after another (with equal priority)

Priority Queues

Prioritize items in a queue, **instead** of processing them one after another (with equal priority)

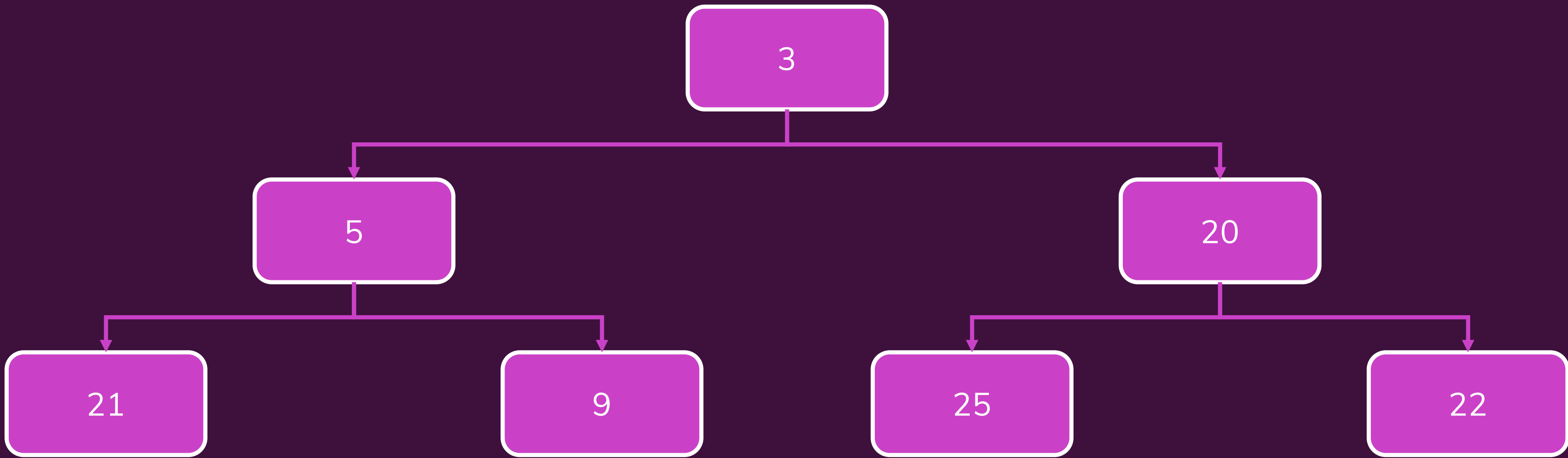


LinkedList Priority Queue – Time Complexity

Insert	O(n) (O(1) if first item / high priority)
Process	O(1)

Heaps

(Min) Heaps are **Trees** where the **parent node values are smaller** or equal than the child node values (for a "**max heap**", it's the **other way around**).



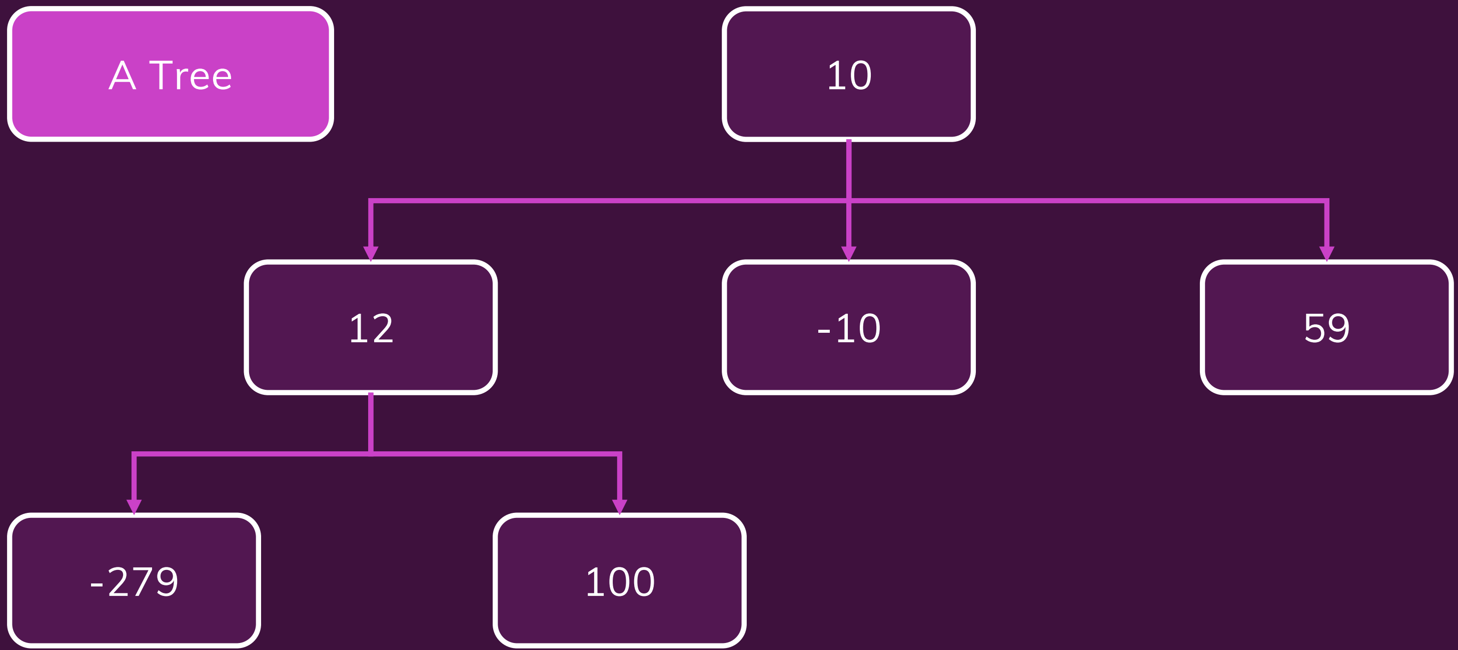
Heap Priority Queue – Time Complexity

Insert	O(log n)
Process	O(log n)

Graph Structures

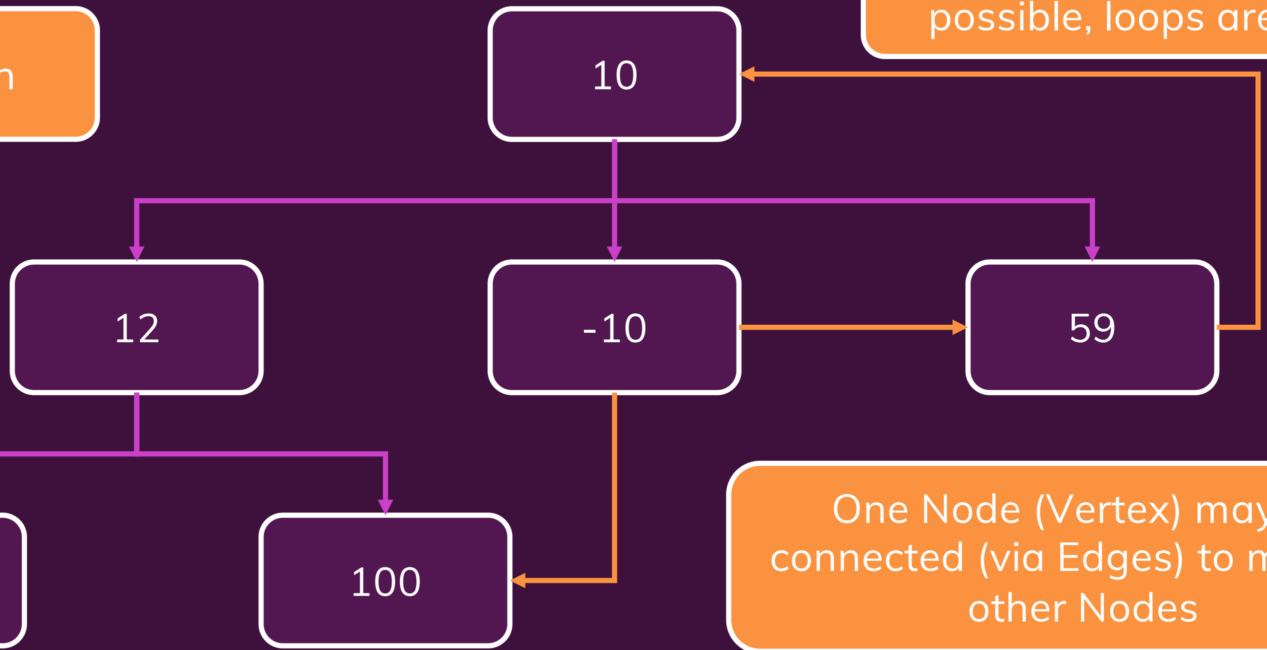
"Complex Trees"

What are "Graph Structures"?



What are “Graph Structures”?

A Graph



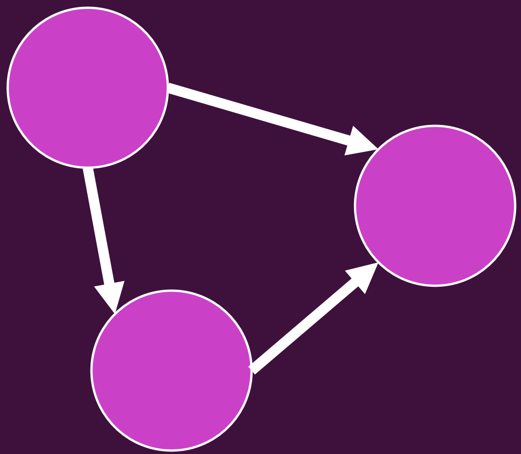
Bi-directional connections are possible, loops are possible

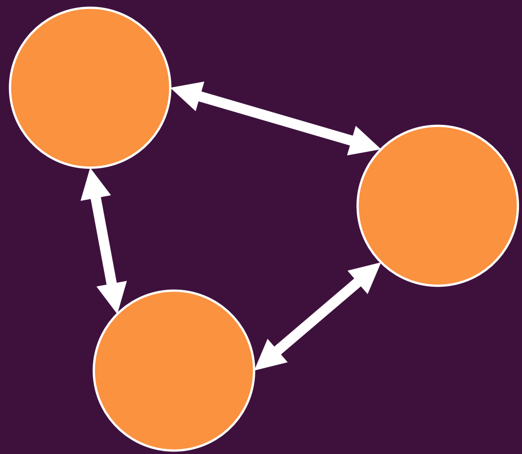
One Node (Vertex) may be connected (via Edges) to multiple other Nodes

The concepts of “levels”, “nesting”, “child/ parent” don’t exist

Directed vs Undirected Graphs

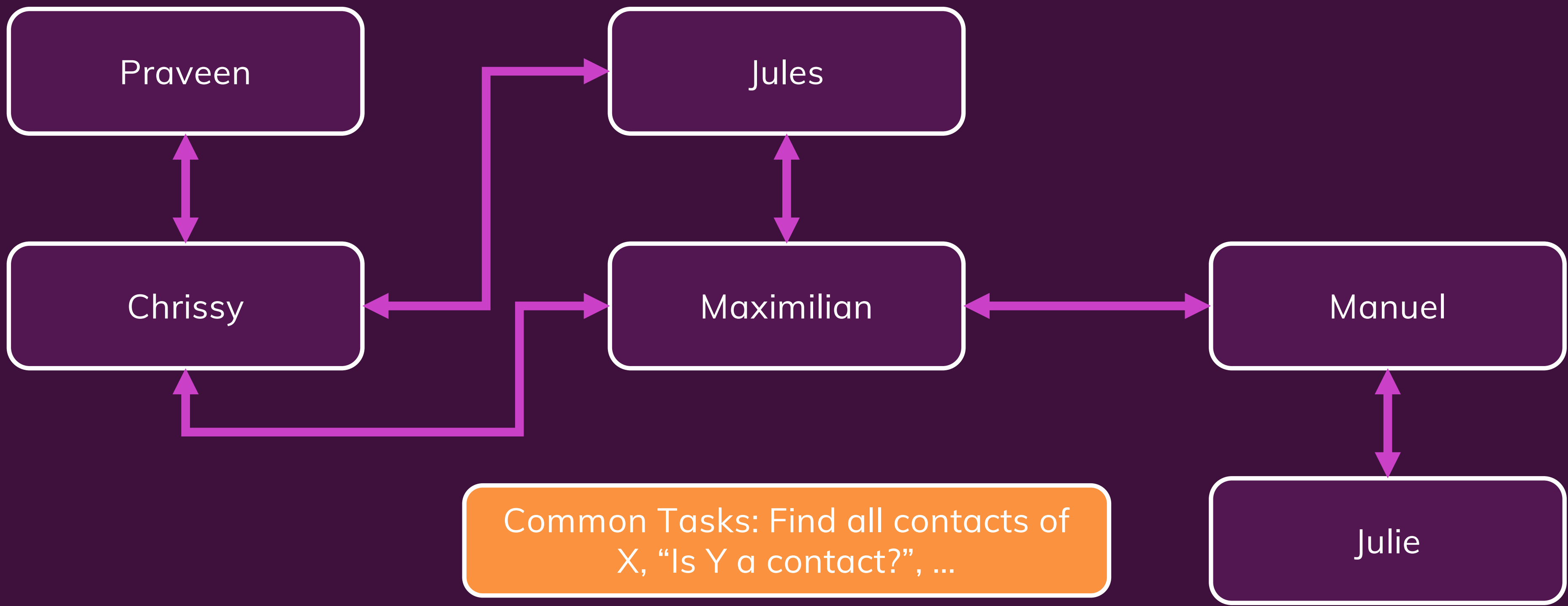
Directed Graphs	Undirected Graphs
Edges between Nodes are unidirectional	Edges between Nodes are bidirectional





Graphs in Real Life / Real Applications

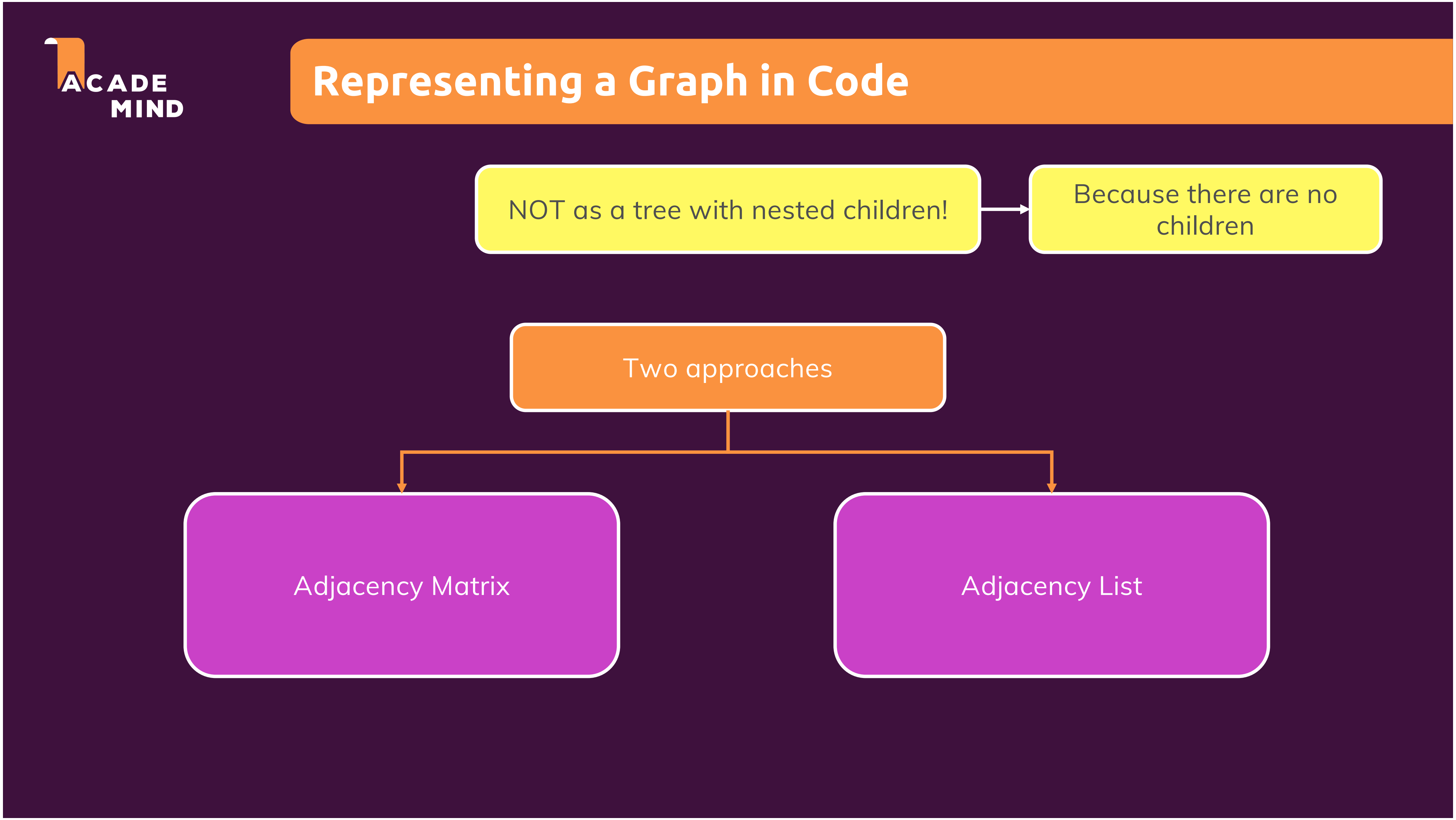
Social Network: Contacts



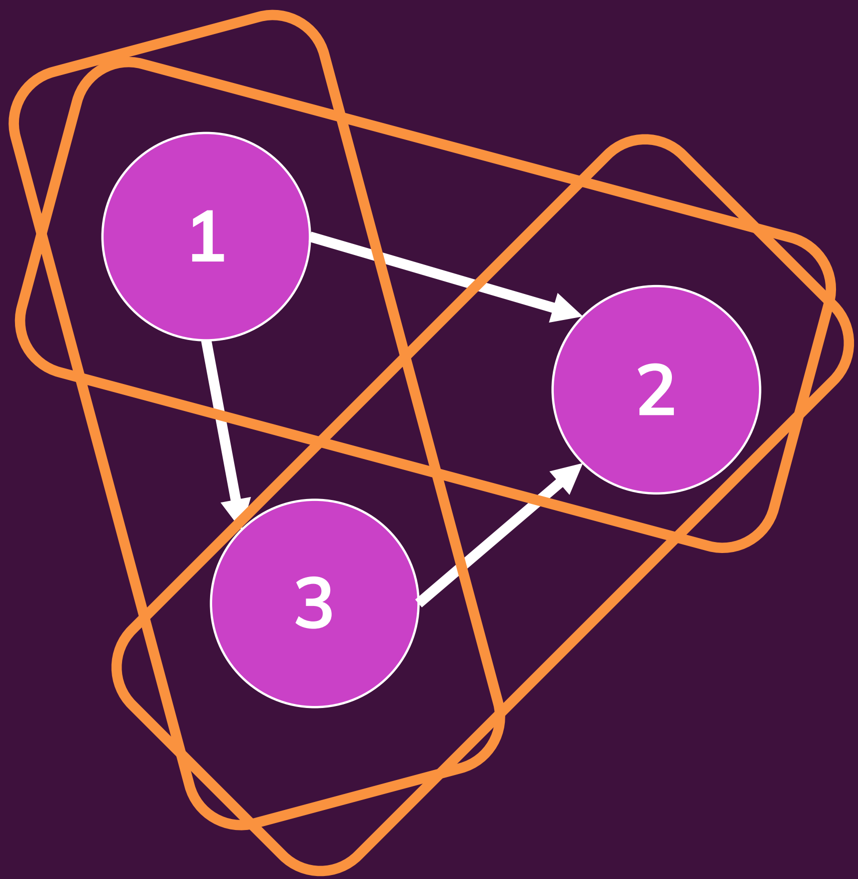
Common Tasks: Find all contacts of X, “Is Y a contact?”, ...

More Real Life Examples

- Maps / Directions
- Knowledge Graph
- Disease Spread
- Recommendation Engines

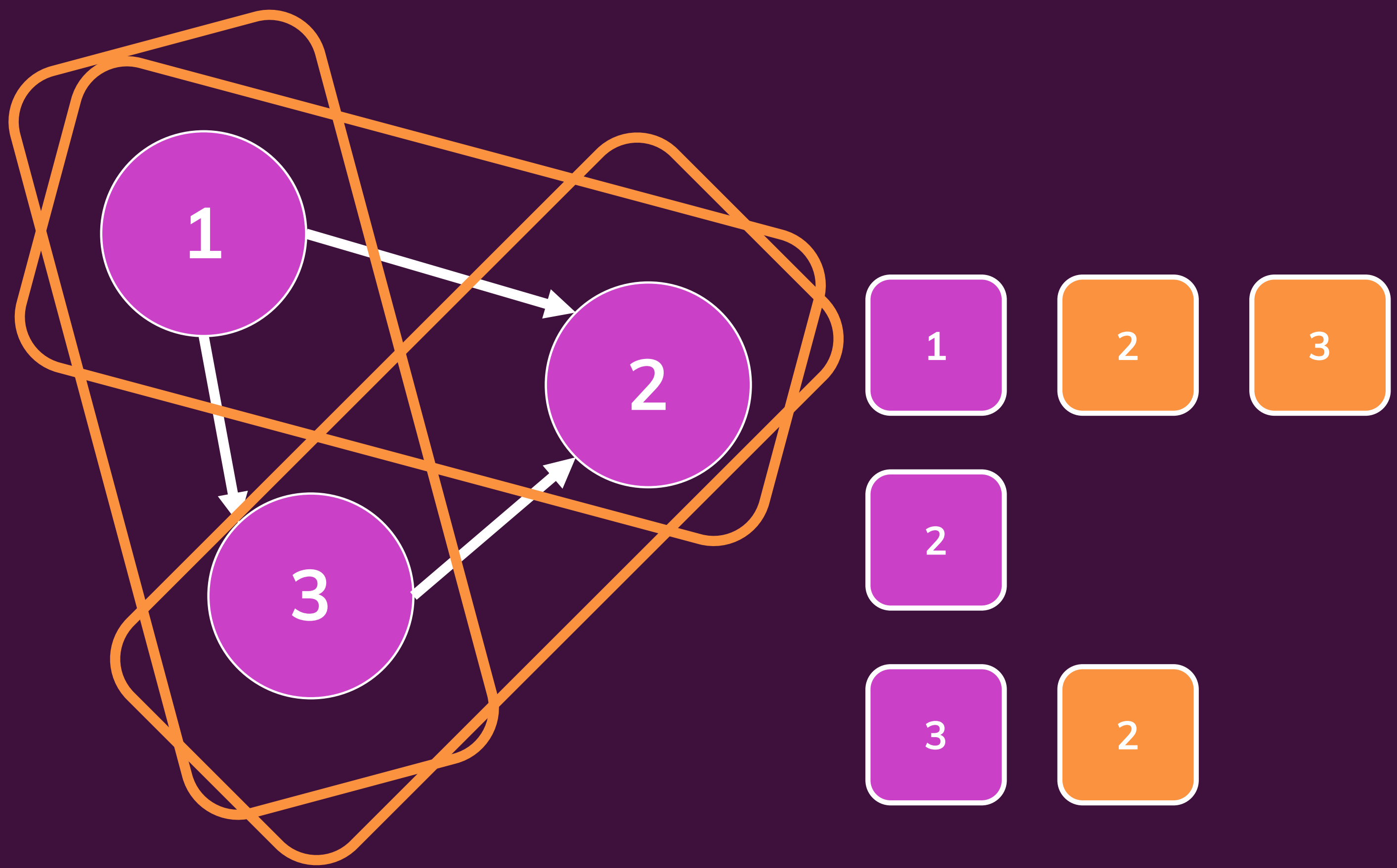

Representing a Graph in Code
NOT as a tree with nested children!
Because there are no children
Two approaches
Adjacency Matrix
Adjacency List

Adjacency Matrix



	1	2	3
1	0	1	1
2	0	0	0
3	0	1	0

Adjacency List

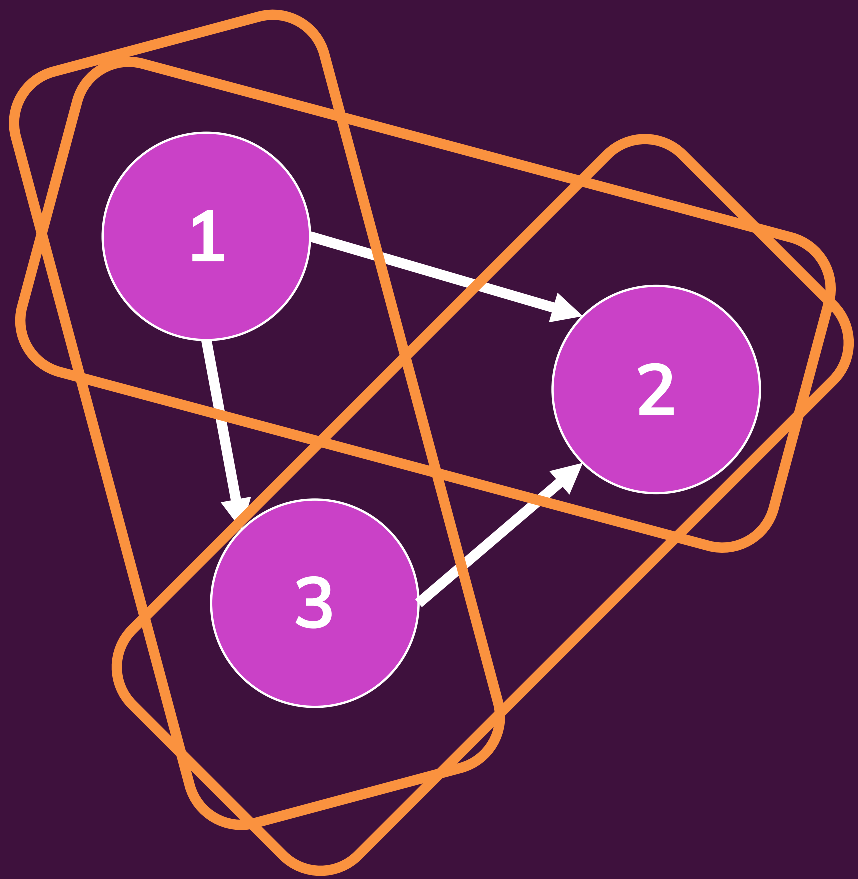


Adjacency Matrix vs List

We'll use this!

	Adjacency Matrix	Adjacency List
Time Complexity		
Insert	**O(n)**	**O(1)**
Find Edge between Nodes	**O(1)**	**O(n)** or **O(1)** (depends on implementation)
Find all adjacent Nodes	**O(n)**	**O(1)**
Space Complexity	O(n^2)	O(n+e)

Adjacency Matrix



	1	2	3	4
1	0	1	1	1
2	0	0	0	0
3	0	1	0	0
4	0	1	0	0