Review and Additional Data Structure Tradeoffs

Quote of the Week: “You know people for a reason, a season, or a lifetime.”
via Samantha Eng, source unknown

Regex puzzle hunt winning teams

- Will be announce on piazza tonight, after I confer with the TAs
- Come to lab tomorrow to pick up your prize!!

There’s a final this Friday

- 3 - 6 pm, 1 Pimentel
- Cheat sheet: 3 sides of 8.5” x 11” paper

So many surveys — sorry

- This course has always conducted a final survey — see link on piazza
  - Is worth 1 point of your final
- HKN conducts a survey during this lecture, at the end
  - Is the lecture quiz for this lecture

What we want

- Computer scientists are problem solvers first, coders second
- Ideally, we solve problems purely in terms of algorithms, taking advantage of high-level structures called ADTs

What we want: example

- **Problem**: Given a list of Strings, how would you group them by those that are anagrams of each other?

- **Solution**: Maintain a map, where the key is the sorted representation of each String, and the values is the list of all Strings that sort to it. Fill up this map by checking each String one-by-one

What we want: example

- **The point:** We speak purely in terms of conceptual ideas like *maps* and *lists* and *sorting*
- You could implement this algorithm in Java, or Python, or Ruby…

What we want

List

Set

Map

Priority Queue

Graph

What we have (to begin with)

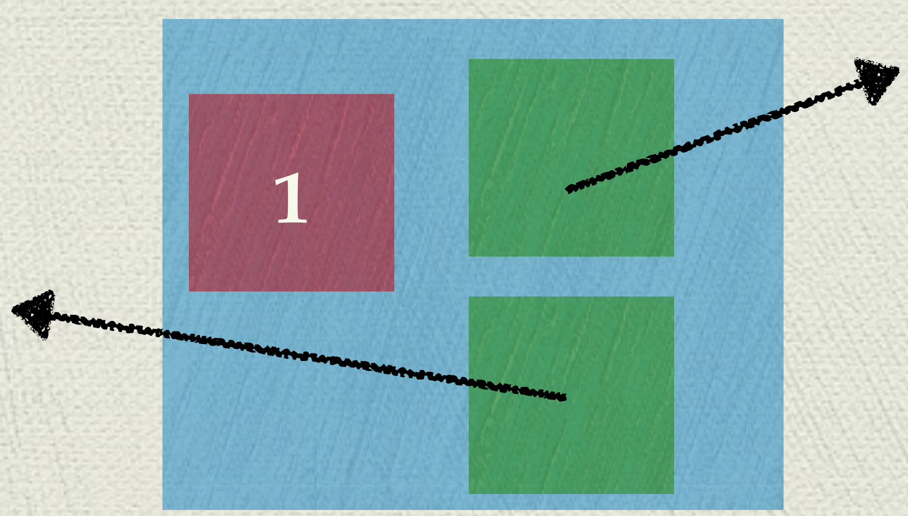
node

array

What we have (to begin with)

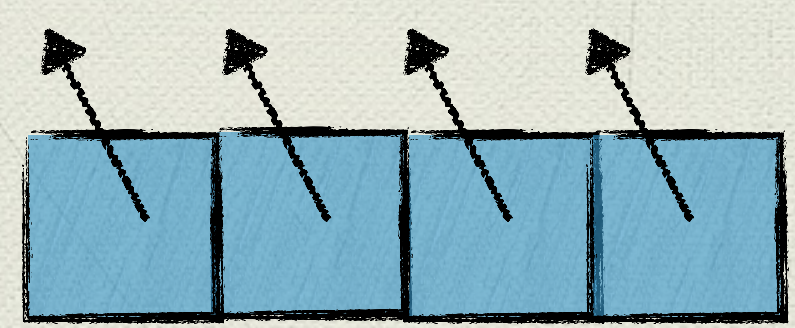
node

an object with data, and references to other nodes



array

a fixed-length region of memory that stores objects in a row, with constant time access



How can we build the top structures out of the bottom primitives?

List

Set

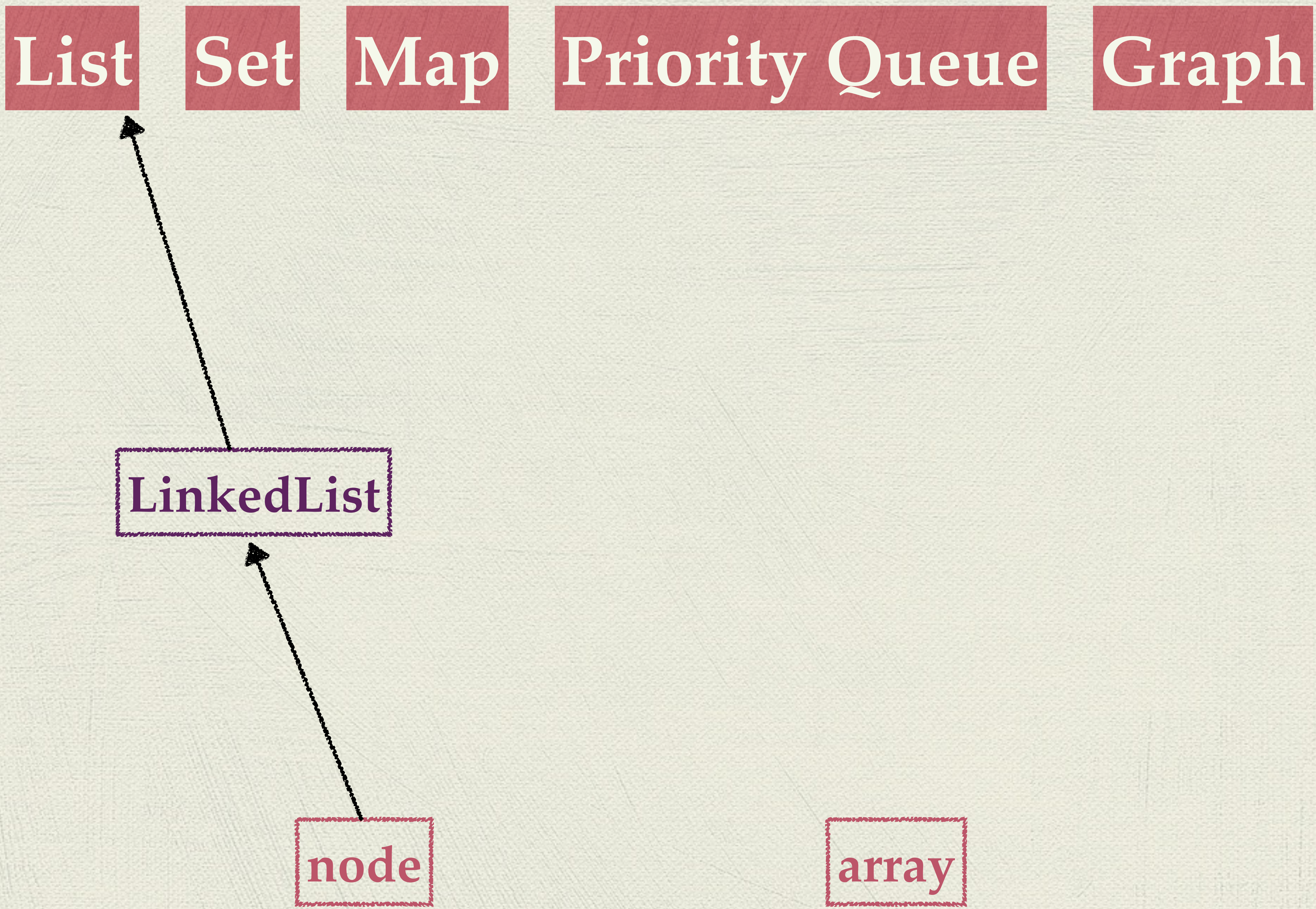
Map

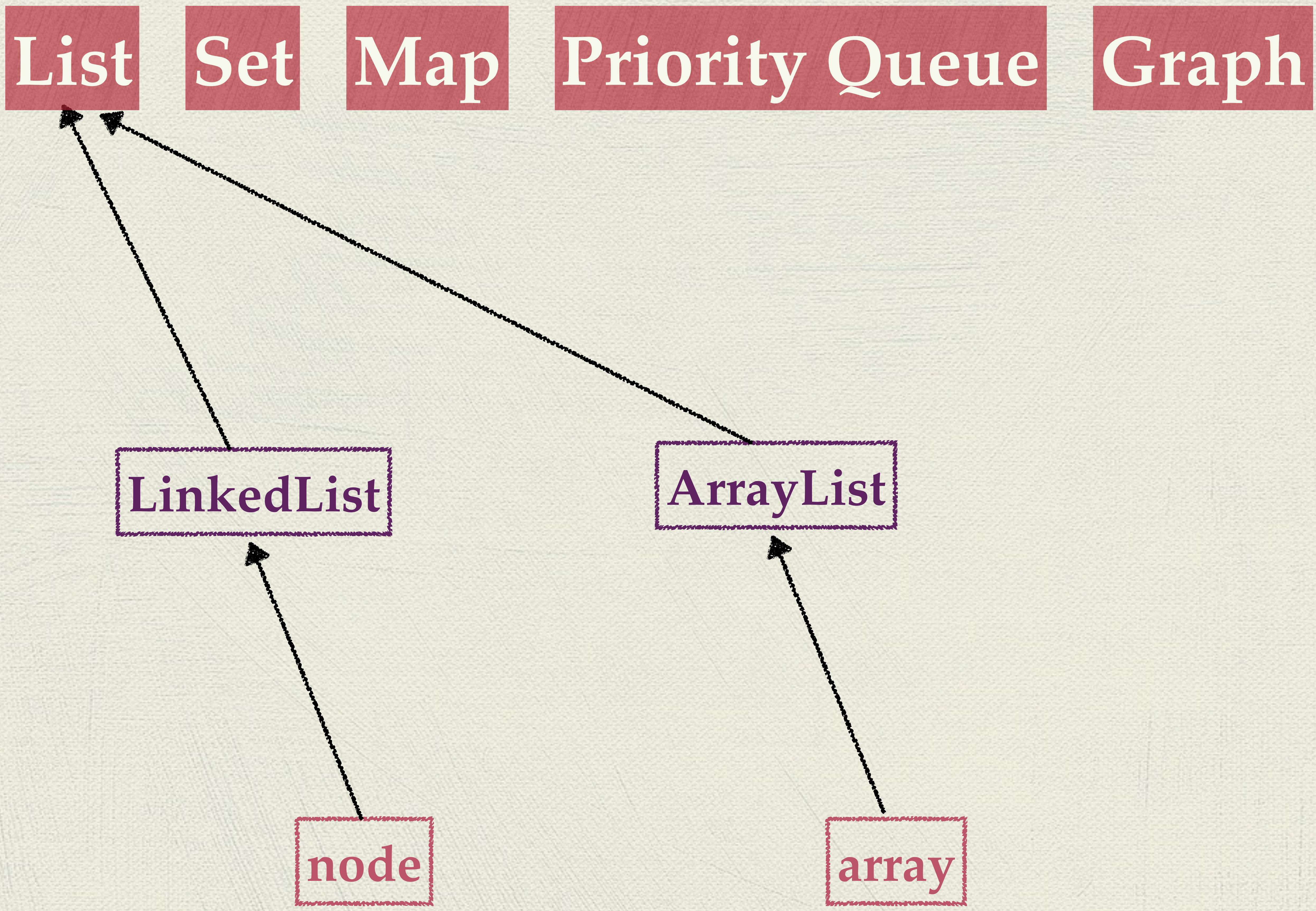
Priority Queue

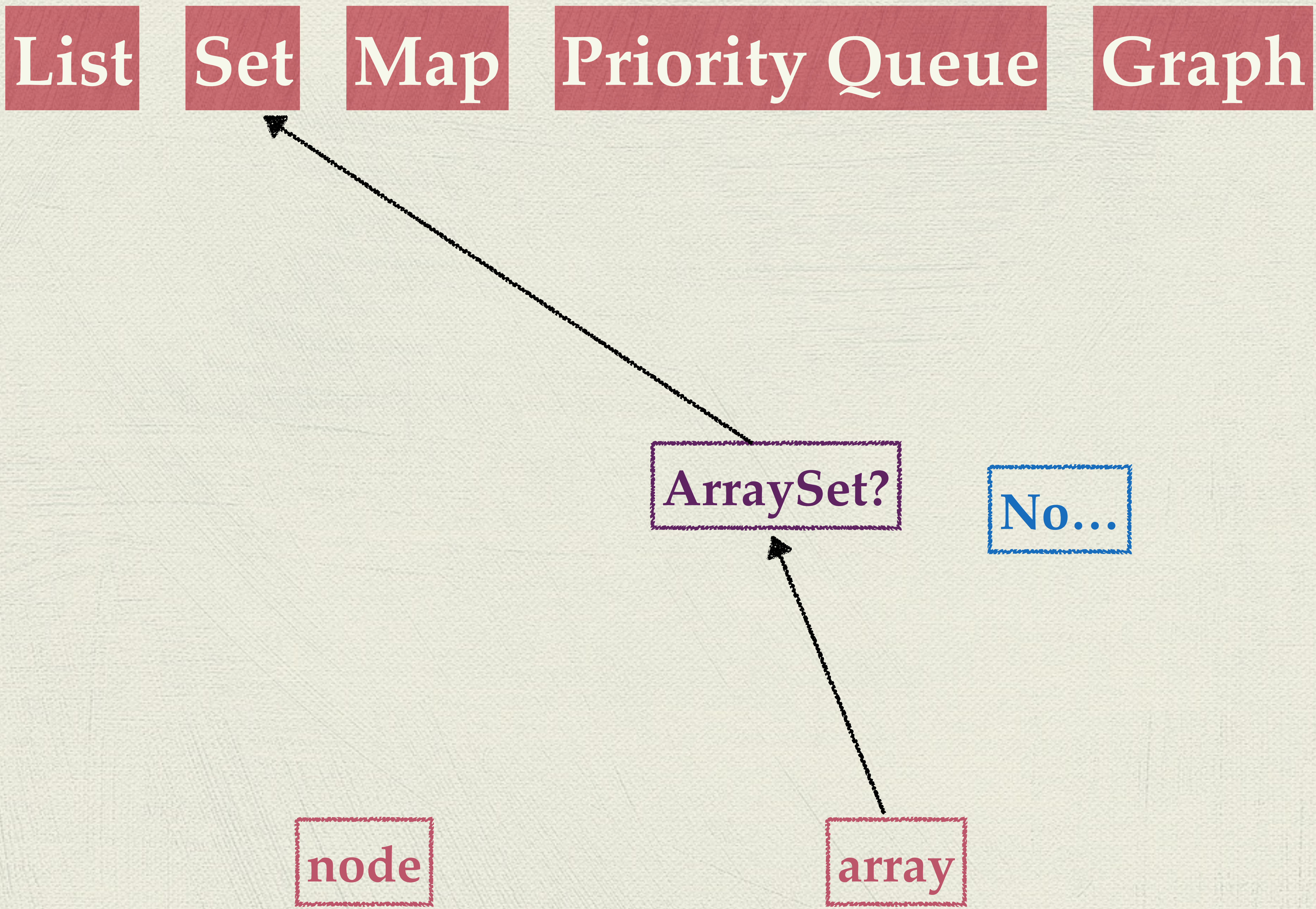
Graph

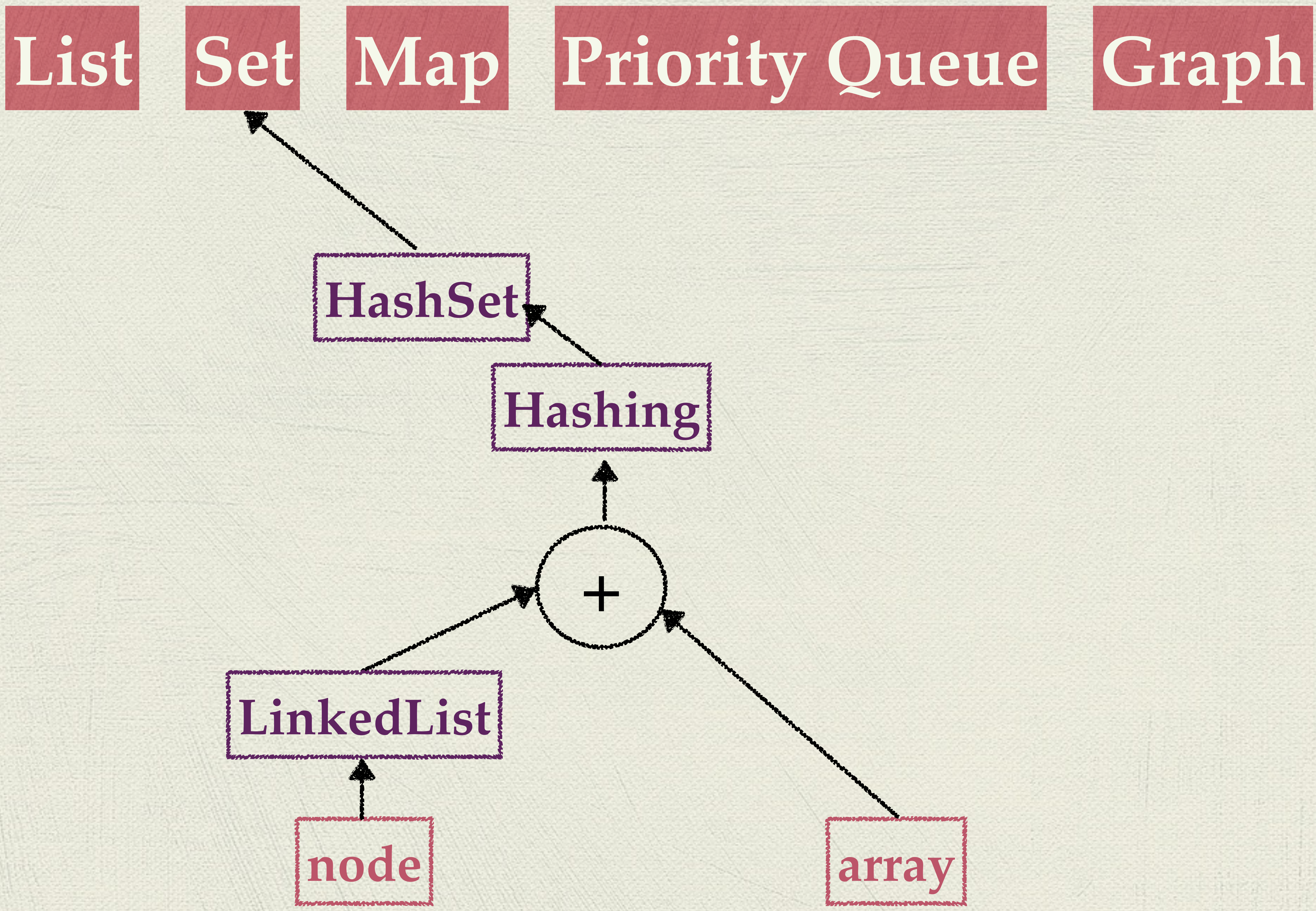
node

array


List
Set
Map
Priority Queue
Graph
LinkedList
node
array

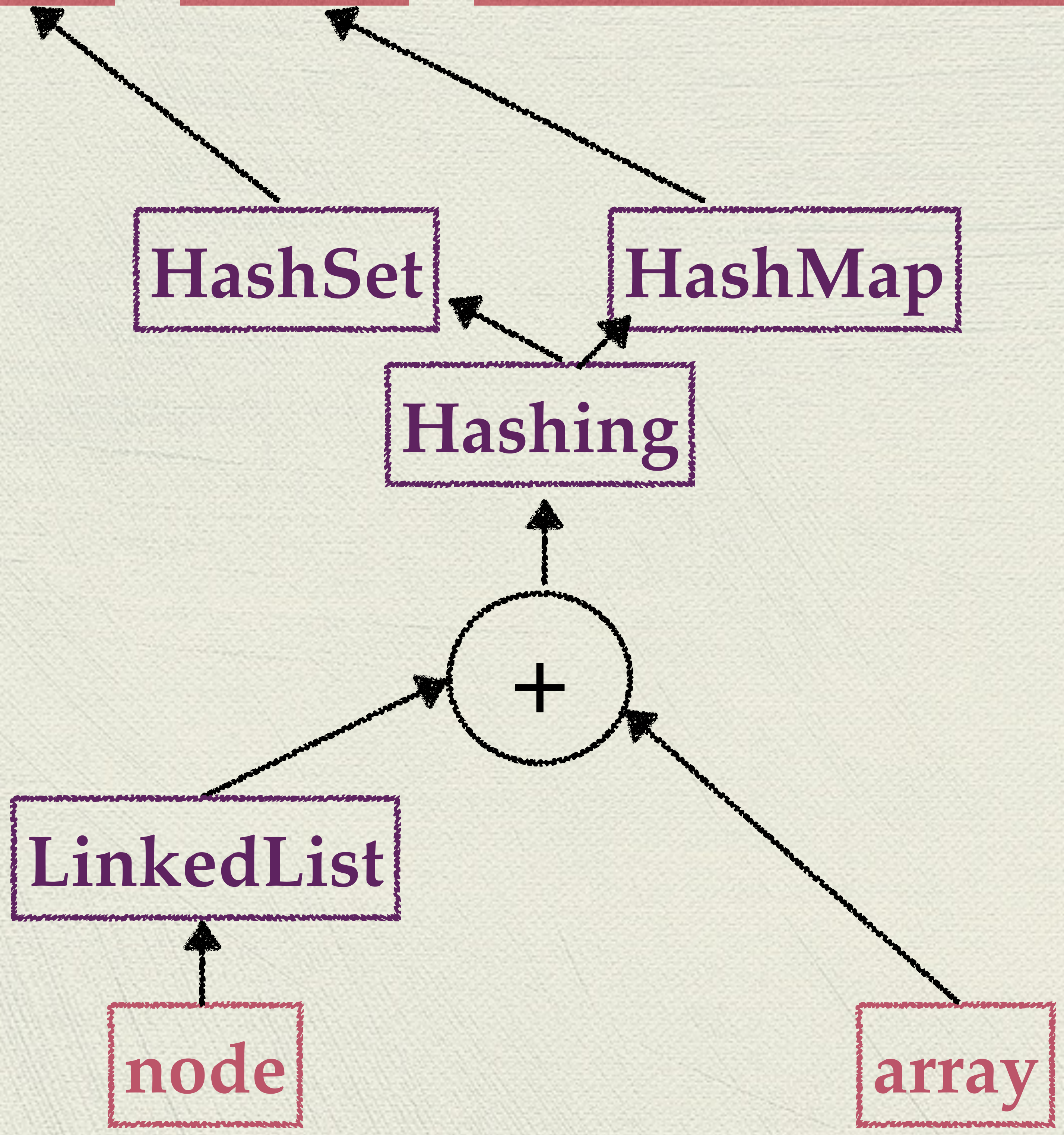

List
Set
Map
Priority Queue
Graph
LinkedList
ArrayList
node
array

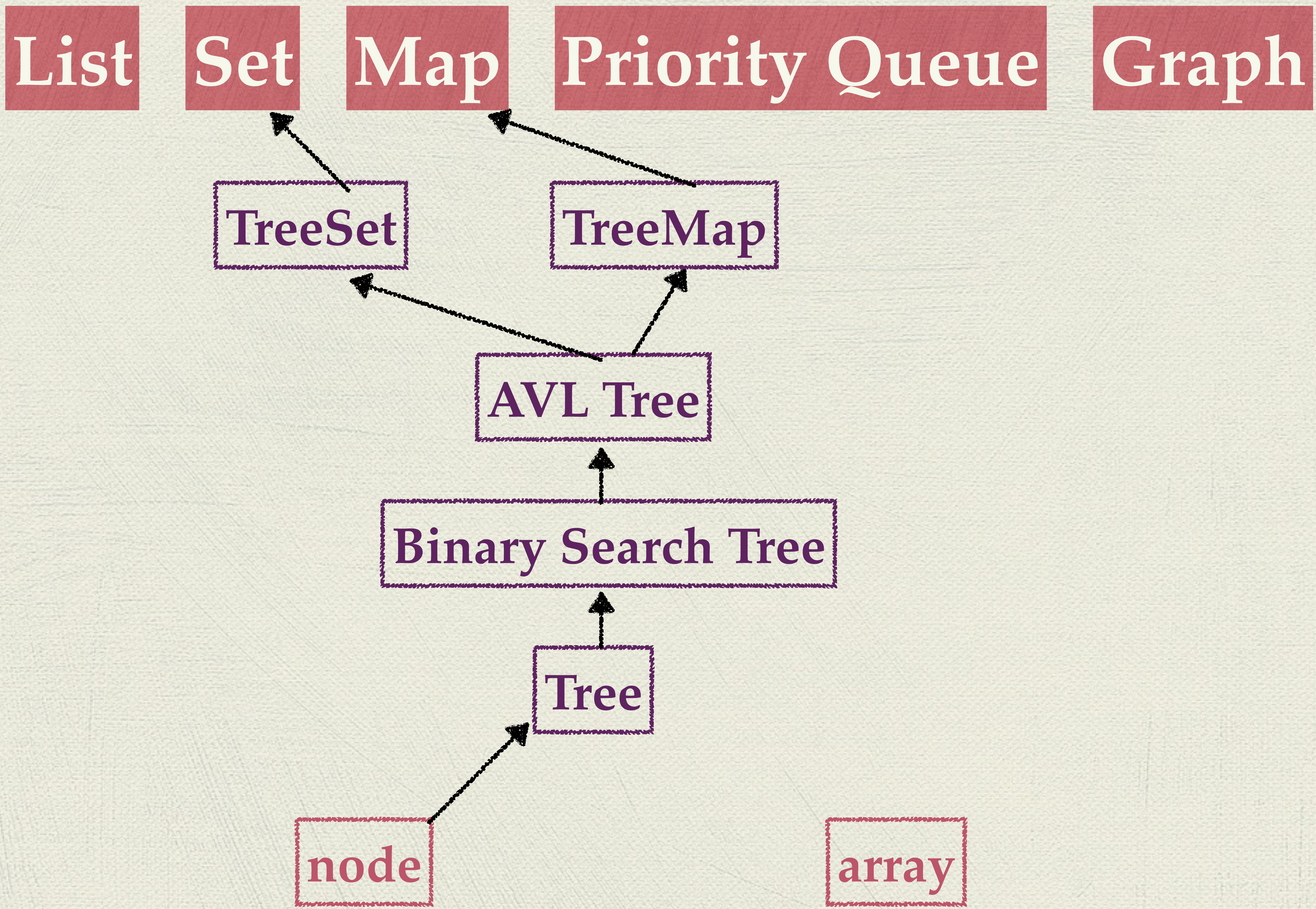

List
Set
Map
Priority Queue
Graph
ArraySet?
No...
node
array

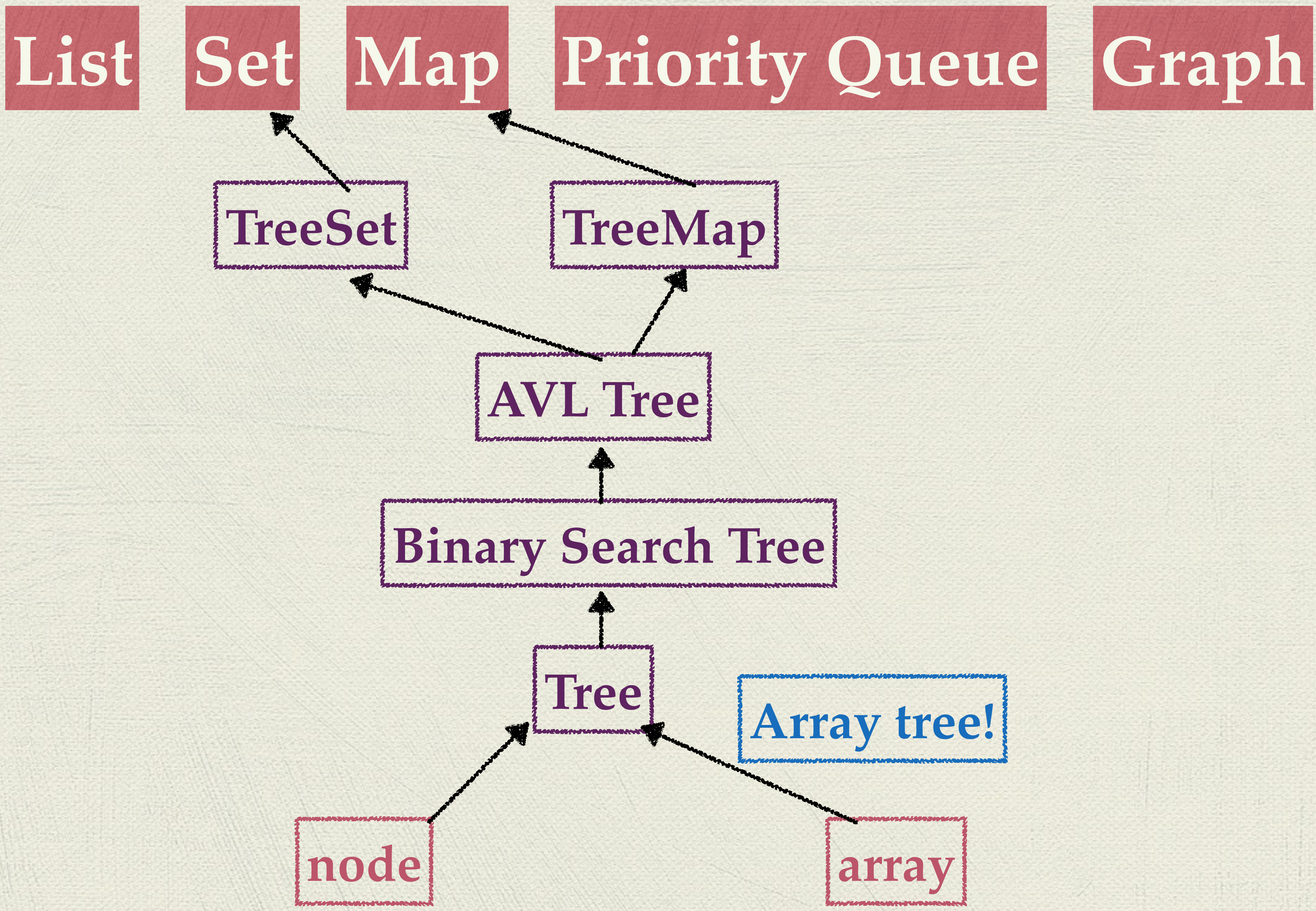


List
Set
Map
Priority Queue
Graph
HashSet
Hashing
+
LinkedList
node
array

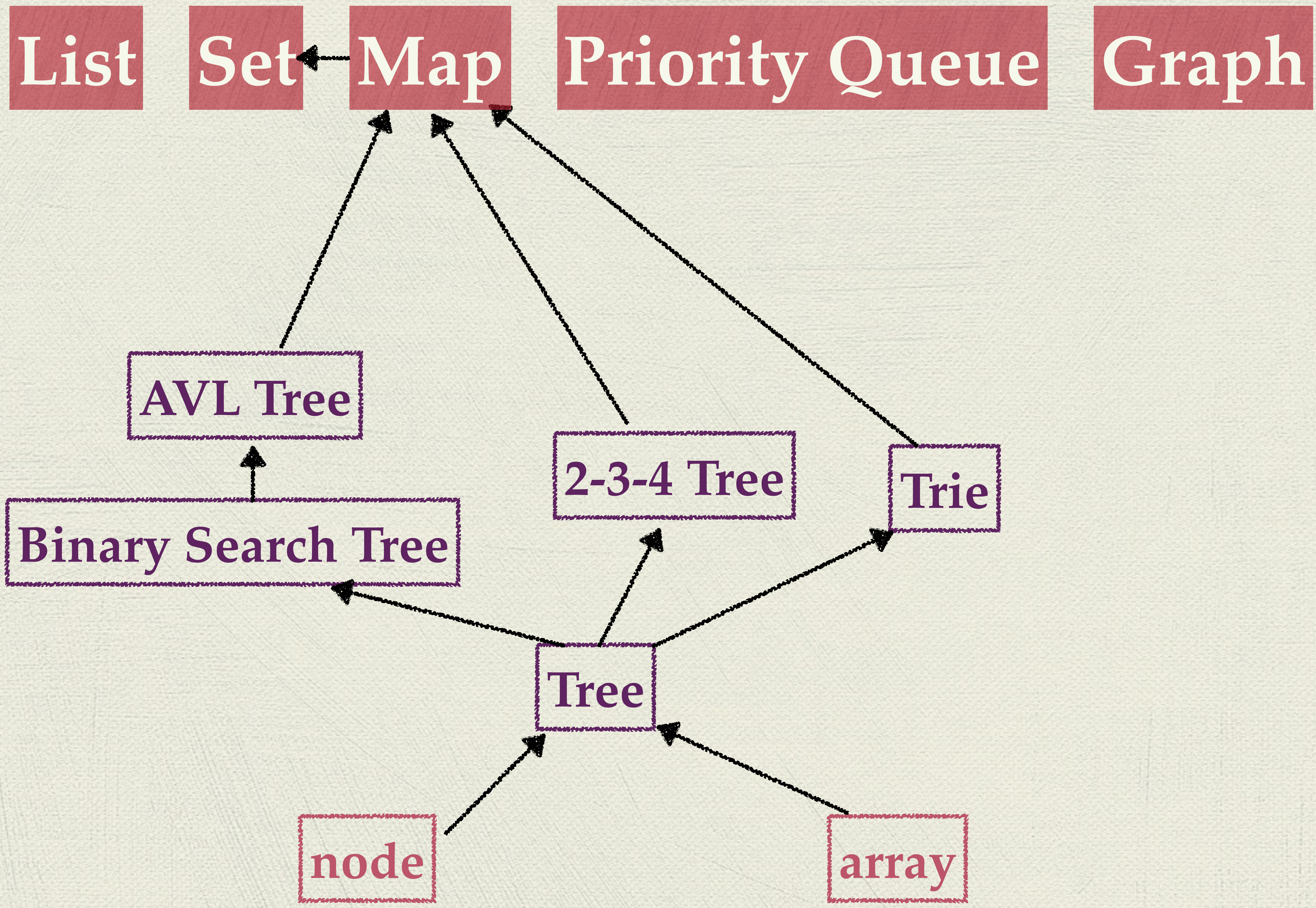
List Set Map Priority Queue Graph

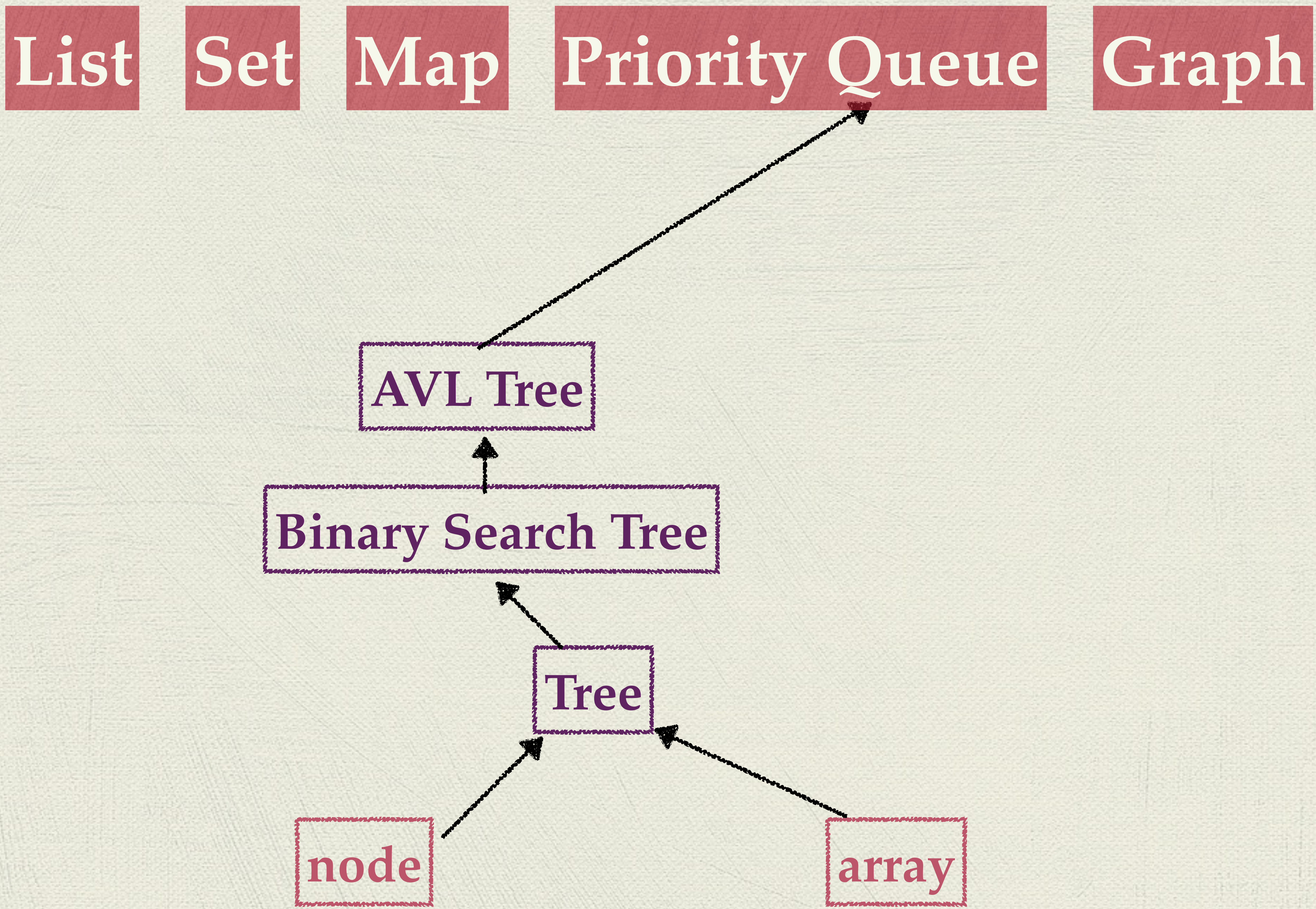


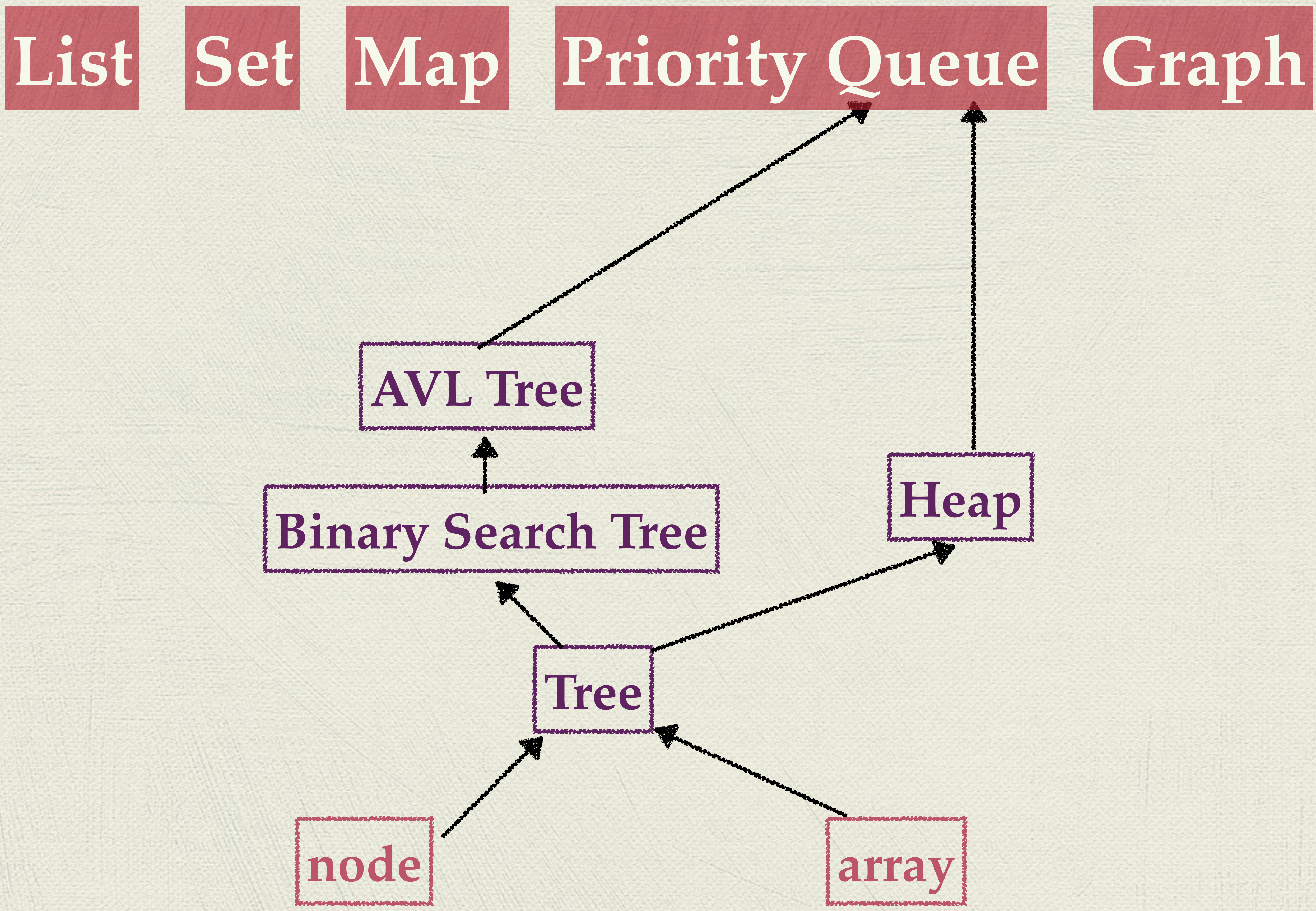

List
Set
Map
Priority Queue
Graph
TreeSet
TreeMap
AVL Tree
Binary Search Tree
Tree
node
array

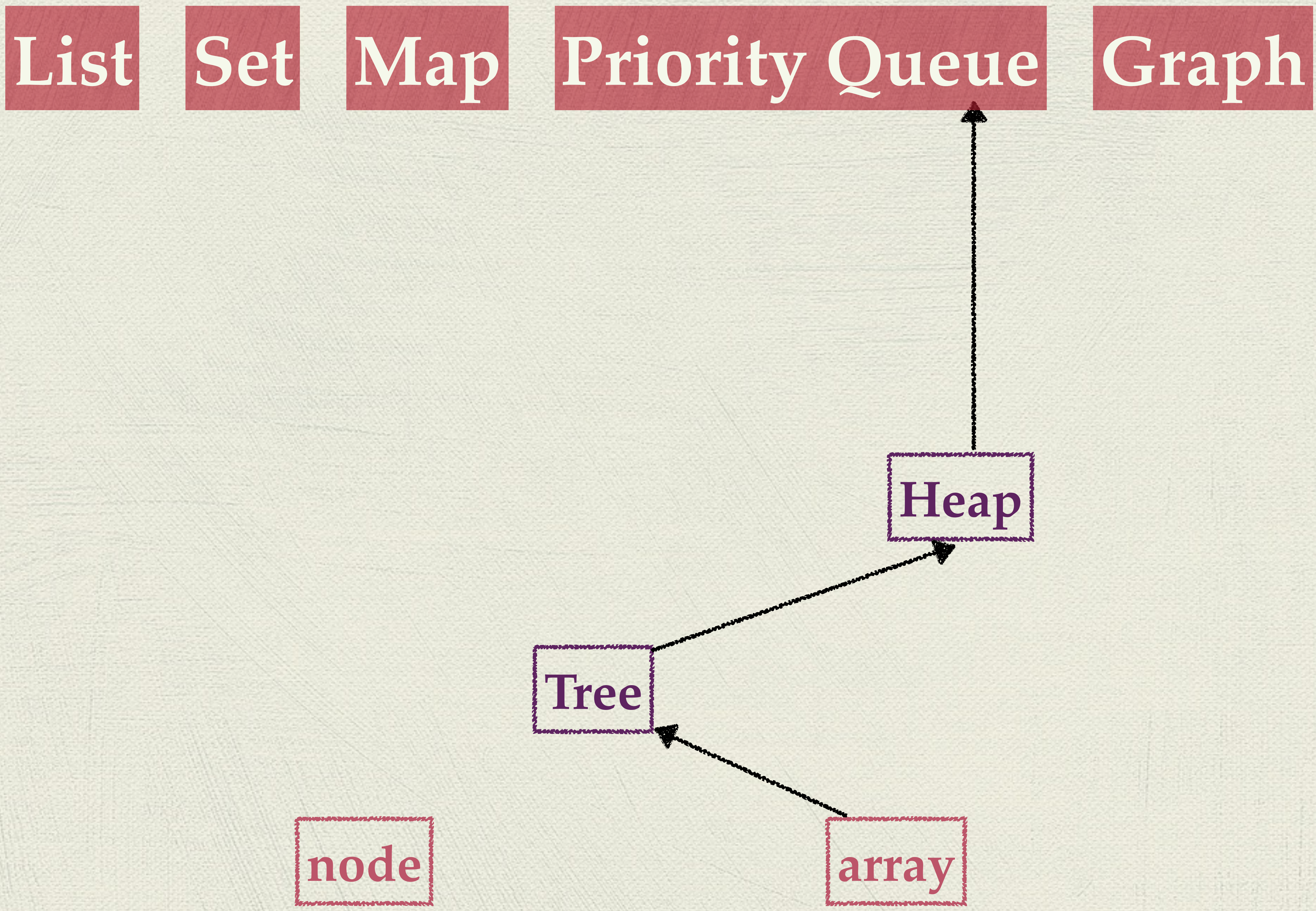


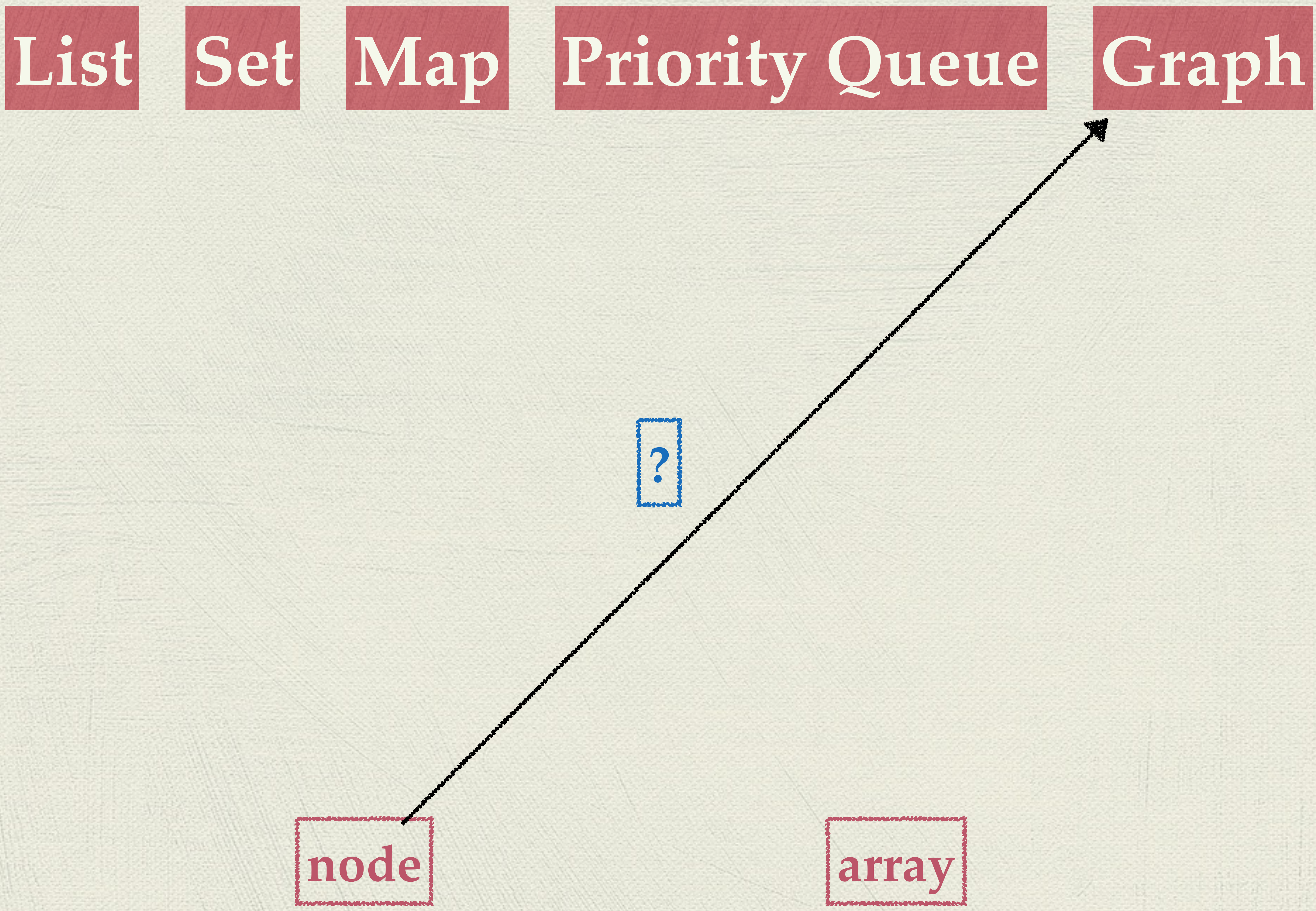
List
Set
Map
Priority Queue
Graph
TreeSet
TreeMap
AVL Tree
Binary Search Tree
Tree
Array tree!
node
array


List
Set
Map
Priority Queue
Graph
AVL Tree
2-3-4 Tree
Trie
Binary Search Tree
Tree
node
array

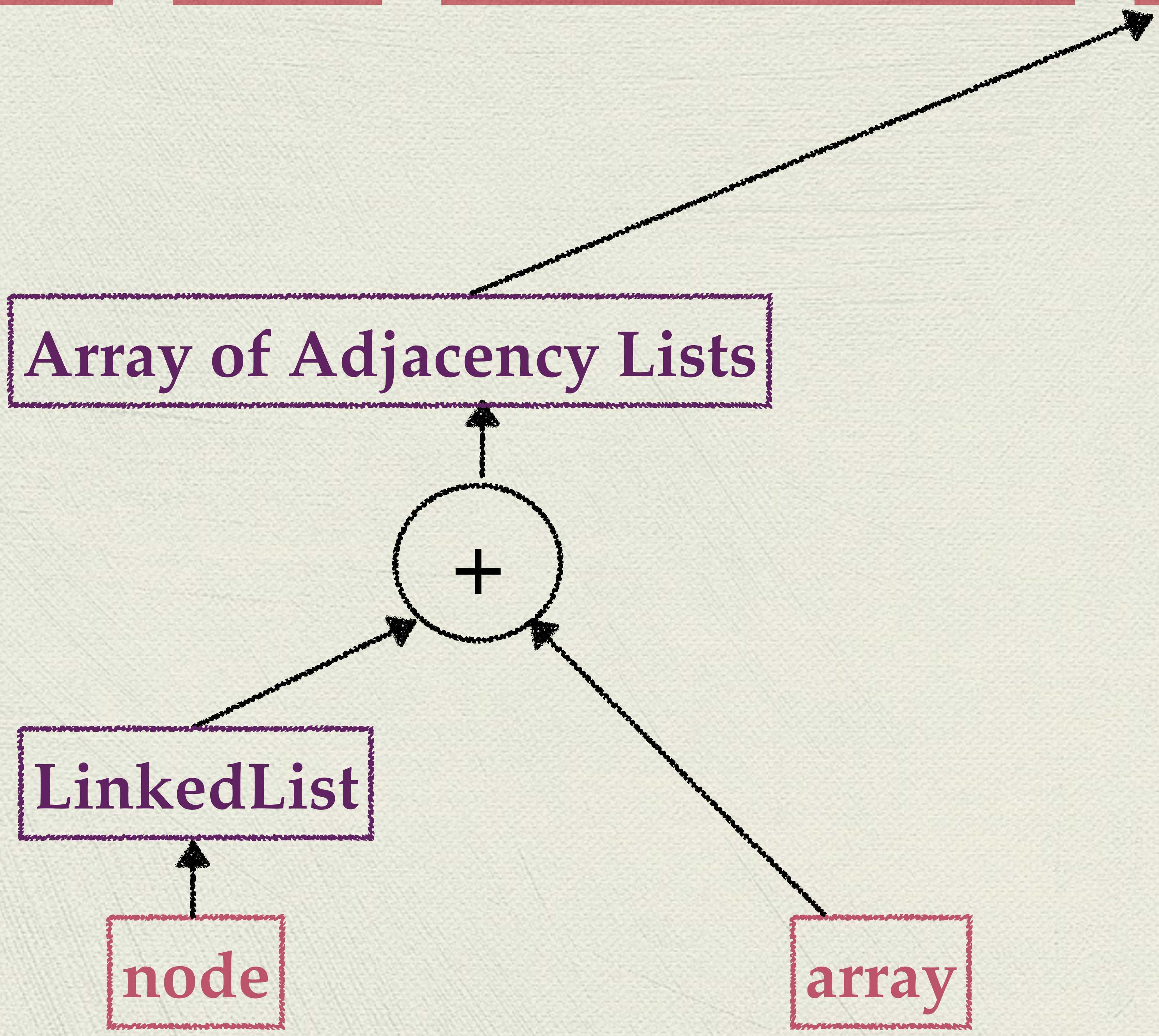

List
Set
Map
Priority Queue
Graph
AVL Tree
Binary Search Tree
Tree
node
array

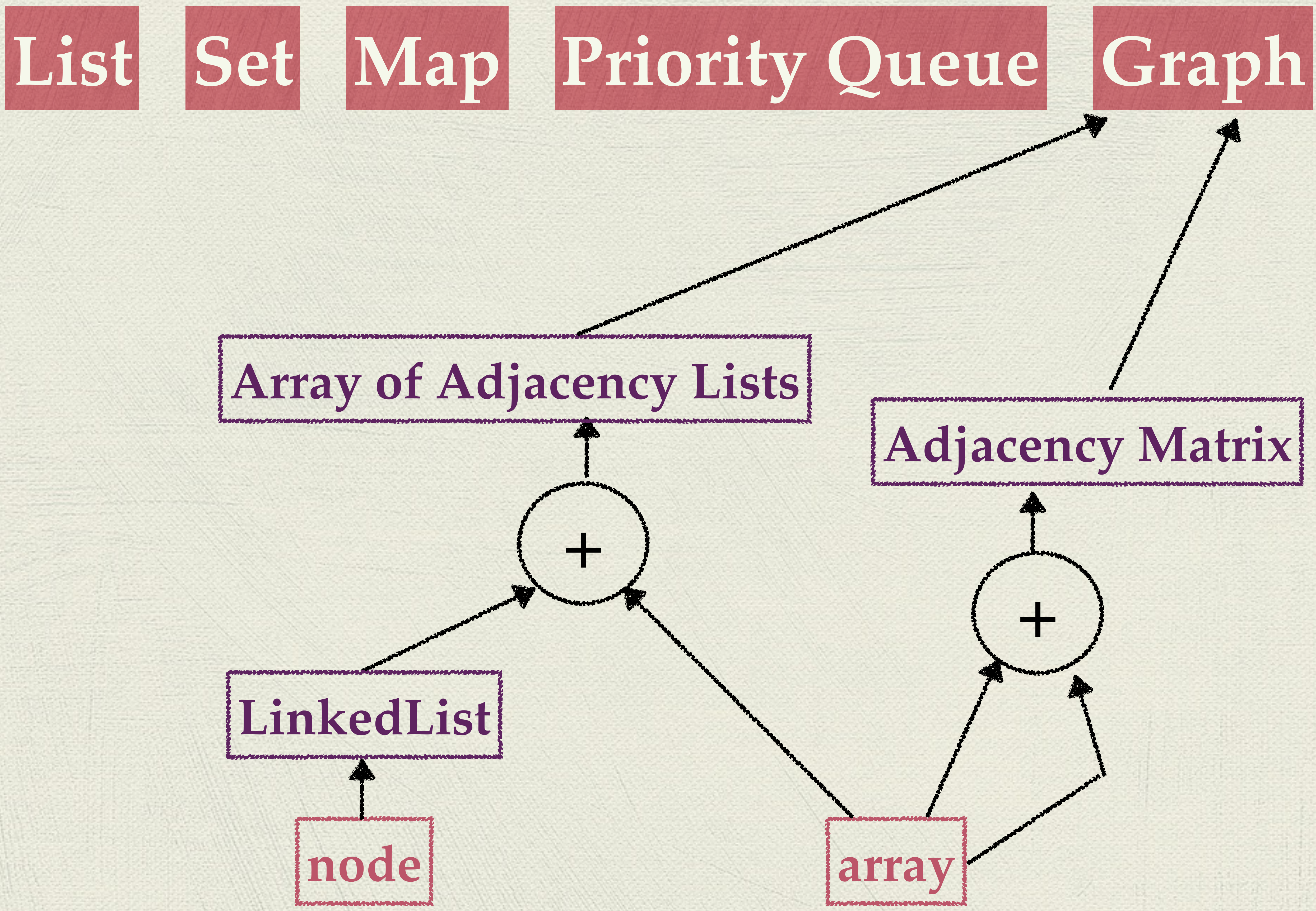

List
Set
Map
Priority Queue
Graph
AVL Tree
Binary Search Tree
Heap
Tree
node
array


List
Set
Map
Priority Queue
Graph
Heap
Tree
node
array


List
Set
Map
Priority Queue
Graph
?
node
array

List Set Map Priority Queue Graph




List
Set
Map
Priority Queue
Graph
Array of Adjacency Lists
Adjacency Matrix
+
+
LinkedList
node
array

List Set Map Priority Queue Graph

Is this the end of the story?

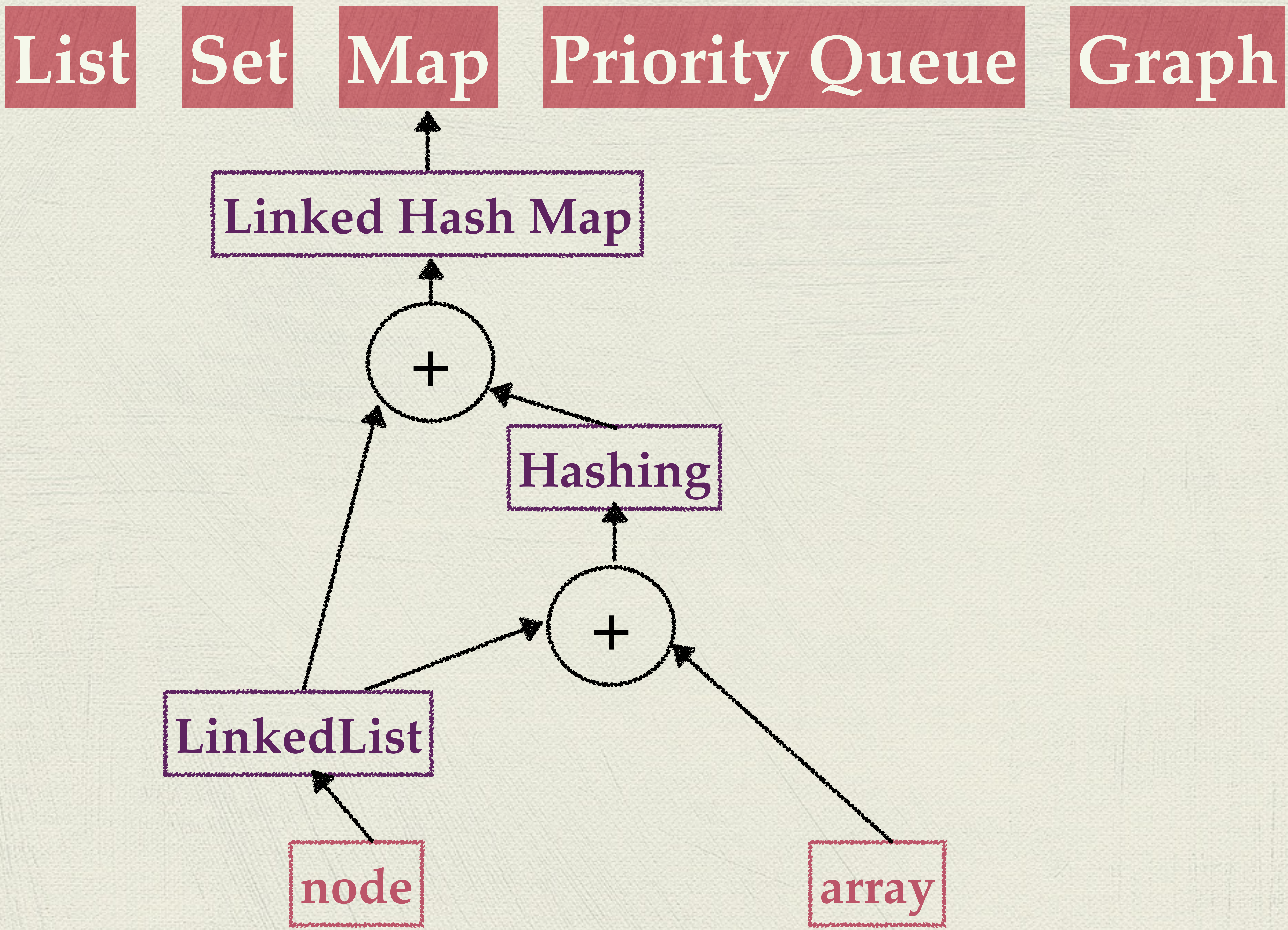
node

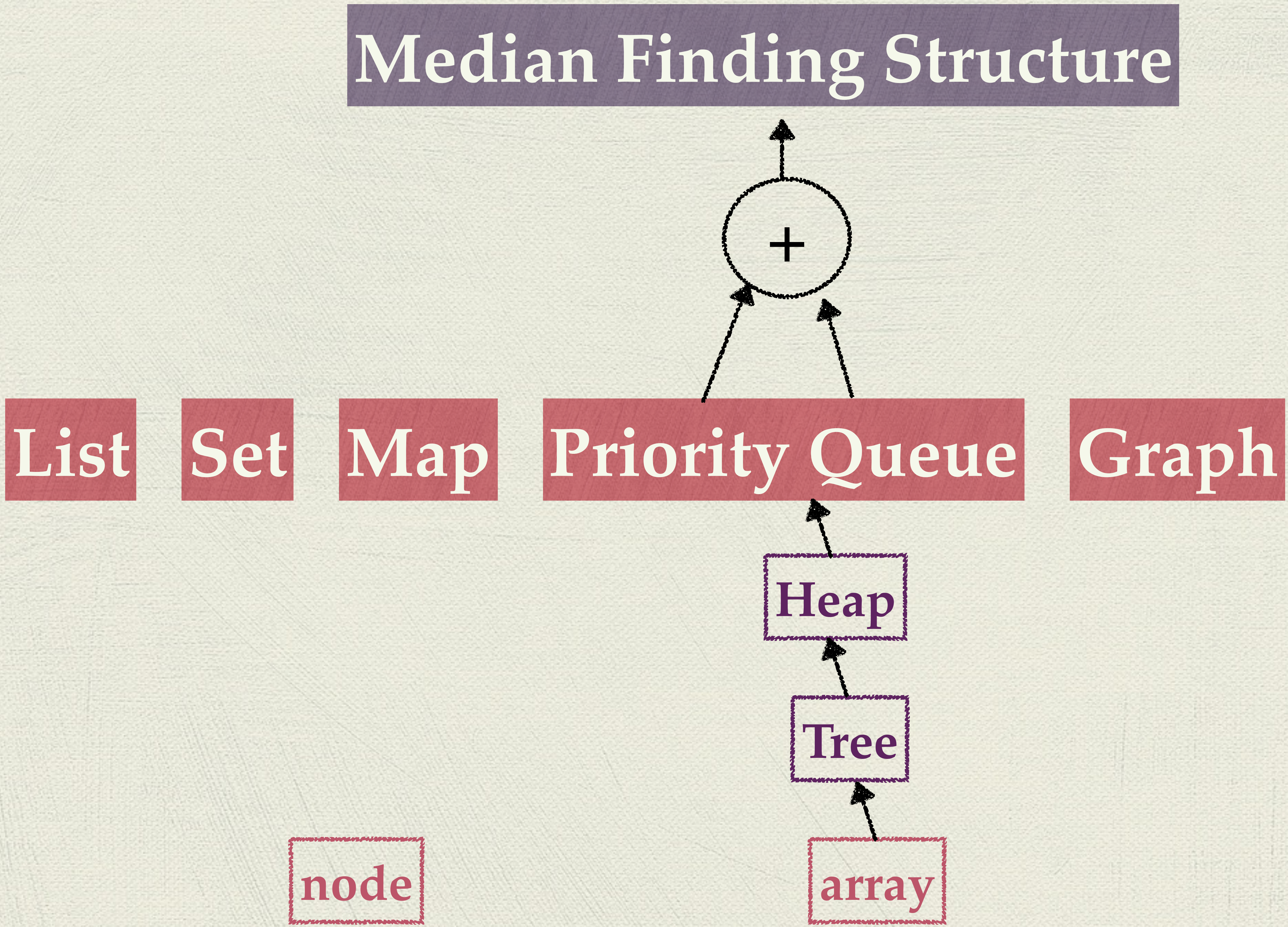
array

List Set Map Priority Queue Graph

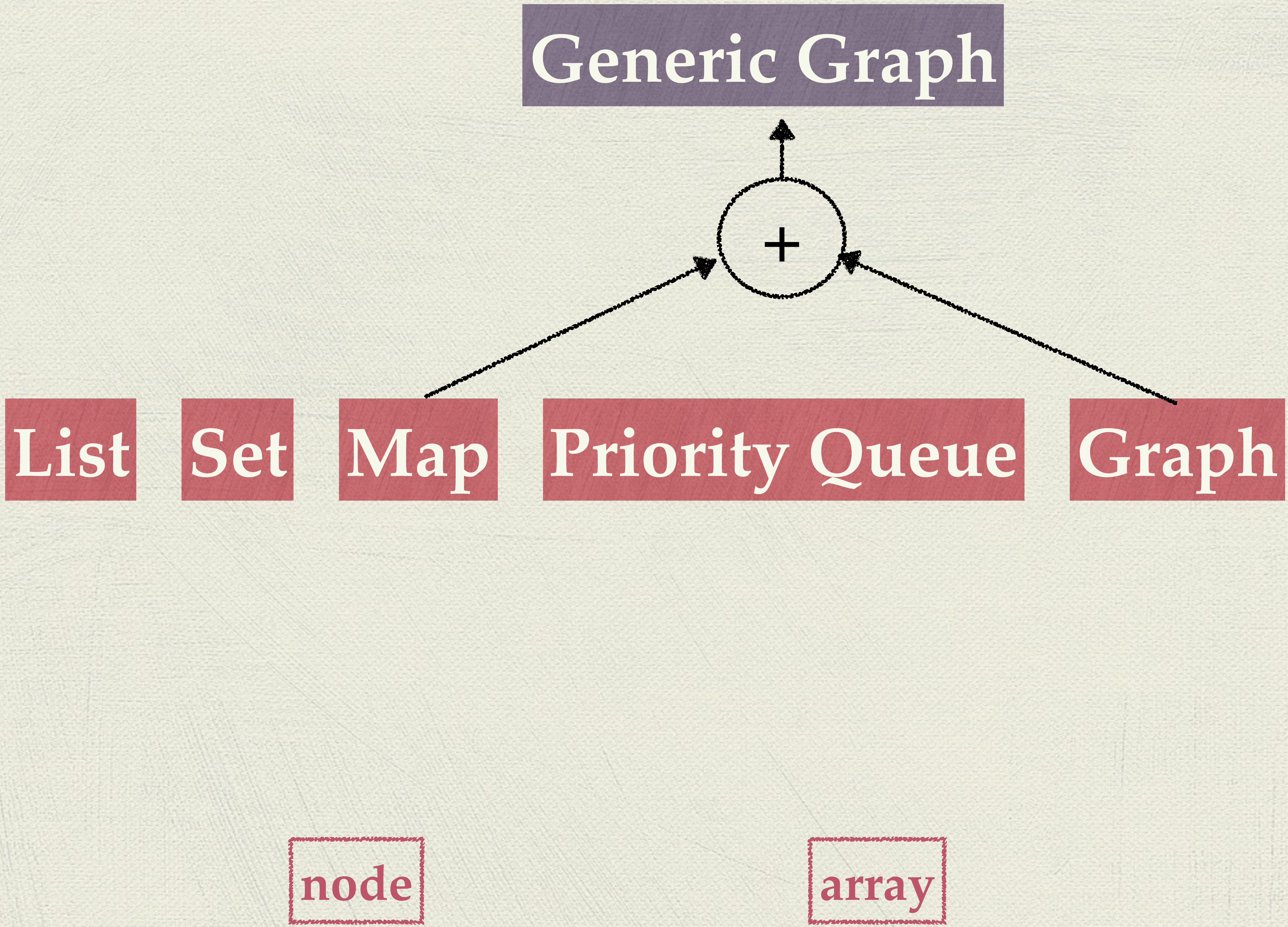
Nope!

node array


List
Set
Map
Priority Queue
Graph
Linked Hash Map
+
Hashing
+
LinkedList
node
array



Median Finding Structure
+
List
Set
Map
Priority Queue
Graph
Heap
Tree
node
array


Generic Graph
+
List
Set
Map
Priority Queue
Graph
node
array

The sky's the limit, really…

List | Set | Map | Priority Queue | Graph

node

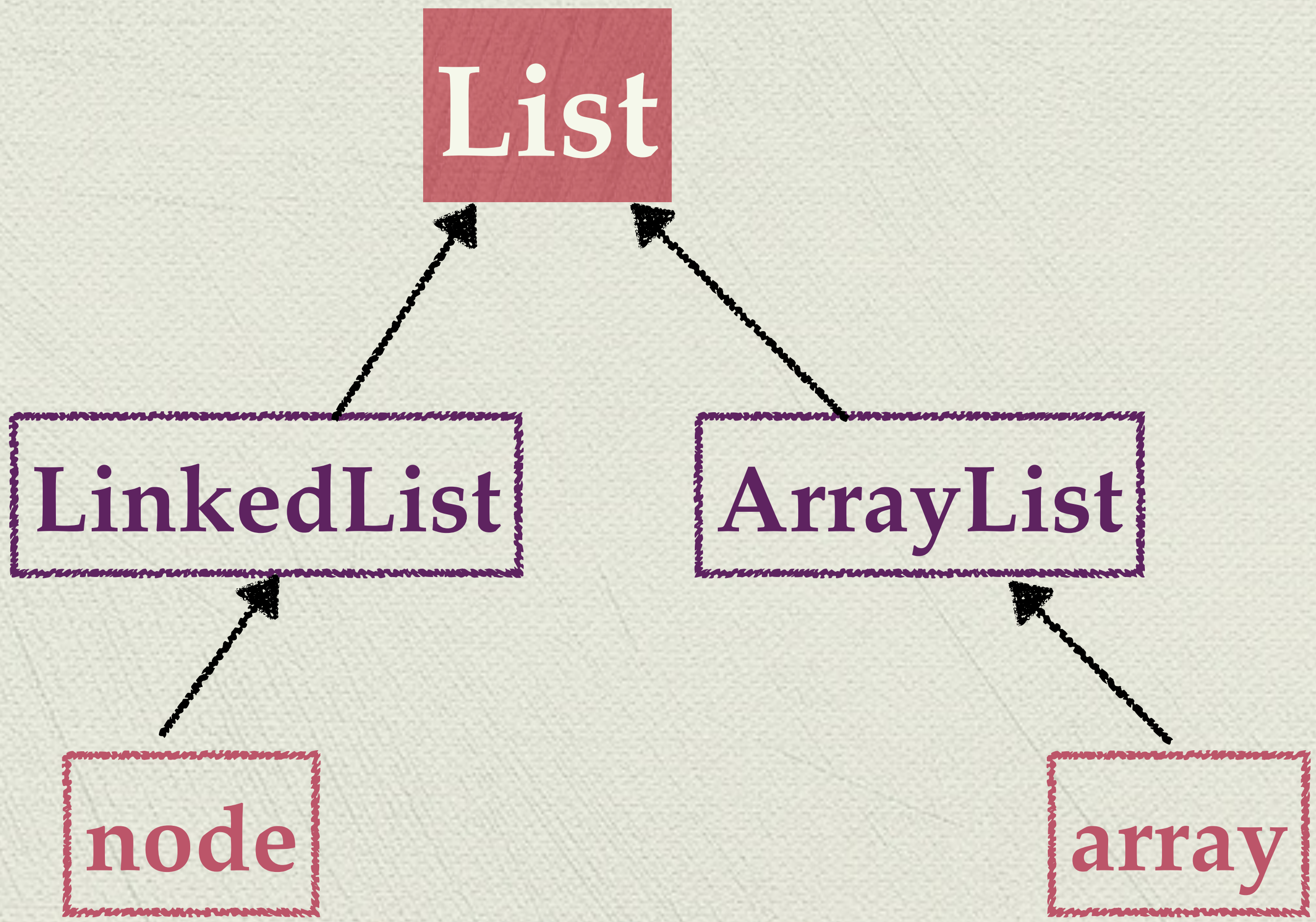
array

From nothing, something

- From our primitives, the **node** and the **array**, we can build anything
- Data structures can be endlessly recombined with each other to create more fanciful and complicated ones

The interface / implementation distinction

- There is a difference between what something *acts like* **(the interface)**, and what it *is underlyingly* **(the implementation)**



The interface/implementation distinction

- In terms of **functionality**, all that matters is what the data structure acts like, but…
- What a data structure is underlyingly may affect its **runtime**

The interface/implementation distinction

- Ex: **LinkedList** and **ArrayList** pretty much have the same methods, but they run at different speeds

The interface/implementation distinction

- Java likes to maintain the interface/implementation distinction using **polymorphism**
  - `List l = new LinkedList();`
  - `Map m = new HashMap();`
  - `Set s = new TreeSet();`
- The static type defines what behavior is allowed, but the dynamic type determines what actually happens

Data structures are flexible

- We taught you the standard/basic implementations of some common data structures
- But these are not the only options, in principle

Example Problem: Mapping out your priorities

- (rejected final problem, because it was too similar to last year)
- Can you implement a priority queue using a hash map?

Example Problem: Mapping out your priorities

- How do the runtimes of the **HashPriorityQueue** differ from the **HeapPriorityQueue**?
  - For adding an item with a priority?
  - For taking out the min value?
  - For changing priority?
- Can you think of an application where the **HashPriorityQueue** might be preferred?

Example Problem: Mapping out your priorities

- Where there are N items in the queue…
- For adding an item with a priority?: **O(1) hashing**, O(logN) heap
- For taking out the min value?: O(N) hashing, **O(logN) heap**
- For changing priority?: **O(1) hashing**, O(N) heap

The Final Break

Take a moment, close your eyes, and relax
Compose yourself, and return with a refreshed mind

A criminally ignored topic: **memory efficiency** and data structures

Two types of efficiency

- When we design data structures, we can optimize for two types of efficiencies
  - **Runtime**, roughly corresponding to the number of operations the structures has to do
  - **Memory**, roughly corresponding to the number of objects in the structure

The time/memory tradeoff

- There is often a **tradeoff** between these two efficiencies
    - Using more memory can yield faster runtimes

The time/memory tradeoff

- Example: The **BiMap** (a two-way map)
  - `public V get(K key)`
  - `public K get(V value)`
- How to implement this structure?

The BiMap

- **Option 1**: use one `HashMap<K, V>`
  - `public V get(K key)`: Simply get from the hash map
  - `public K get(V value)`: Iterate through all the keys until you find the one with the given value

The BiMap

- **Option 1**: use one `HashMap<K, V>`
  - `public V get(K key)`: Simply get from the hash map: **O(1)**
  - `public K get(V value)`: Iterate through all the keys until you find the one with the given value: **O(N)**, if N keys

The BiMap

- **Option 2**: use two maps, one `HashMap<K, V>` and one `HashMap<V, K>`
    - `public V get(K key)`: Simply get from one hash map: **O(1)**
    - `public K get(V value)`: Simply get from the other hash map: **O(1)**

The BiMap

- **Moral of the story:** We can store twice as much memory to get a speedup on one of our methods
- This is a common tradeoff throughout data structures

The time/memory tradeoff

- So, how do you pick between optimizing time or memory?
- Optimize time. Always. Time is what matters (unless you straight-up run out of memory). Except…

The time/memory tradeoff

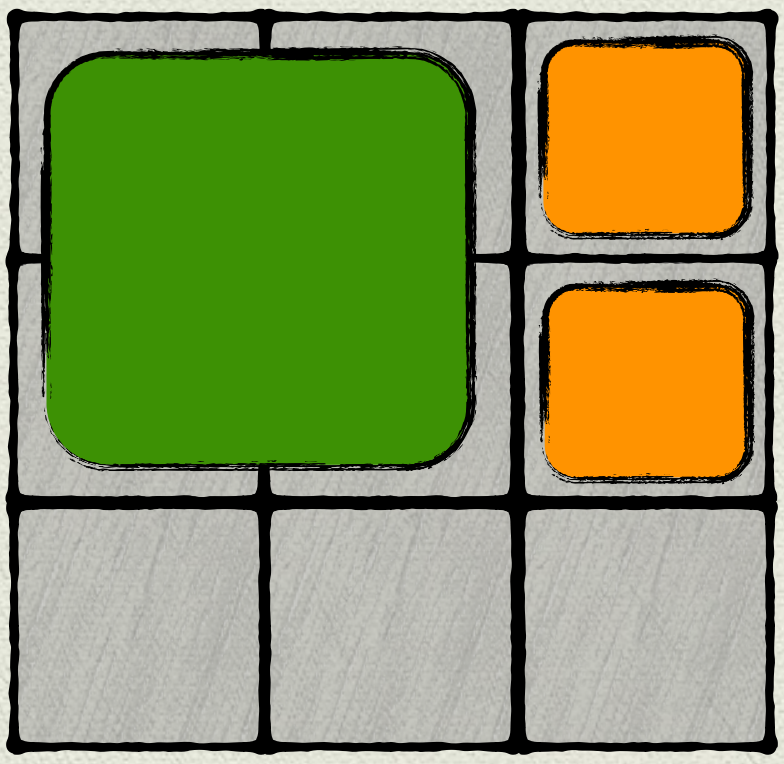
- …Being too wasteful with memory will actually slow your time down
- (wait for 61C for the reason)

Analyzing memory efficiency

- Can be done with big O, in a way very similar to analyzing time
    - A single reference or primitive takes **O(1)** space
    - An array of N references takes **O(N)** space
    - An object takes 1 + as much space as the primitives and references inside it

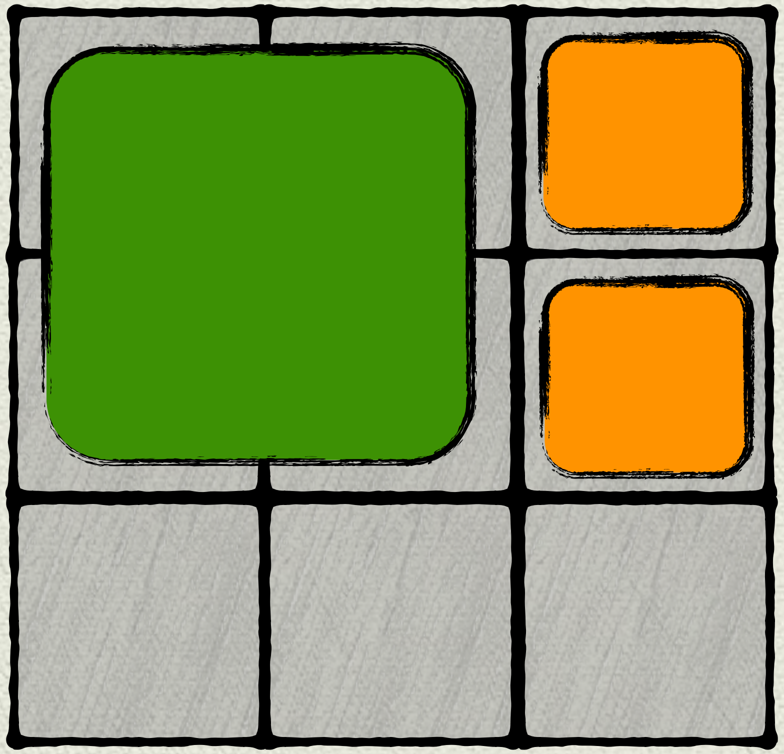
Memory efficiency example: don't repeat yourself

- Here's a sliding block puzzle, like from project 3

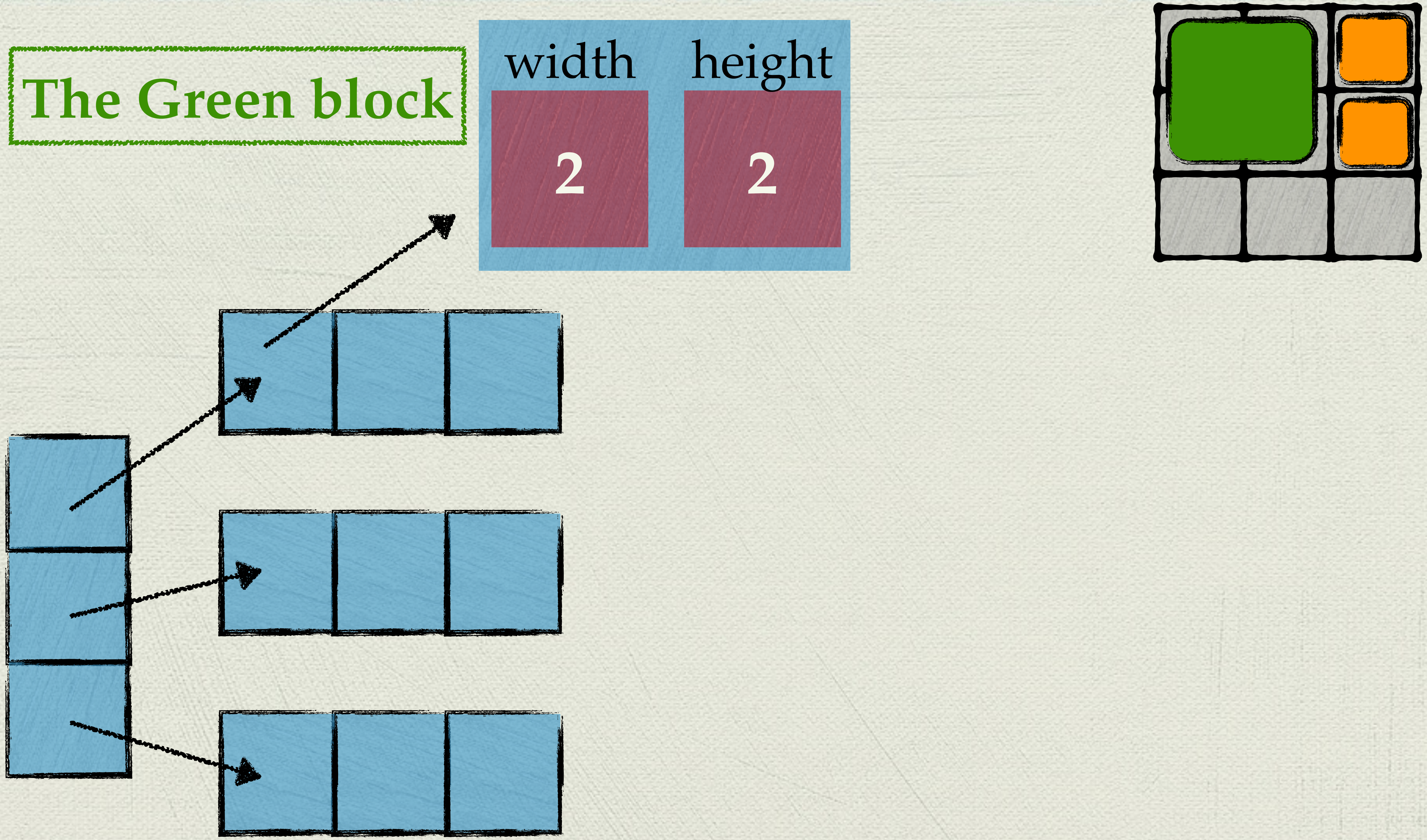


Memory efficiency example: don't repeat yourself

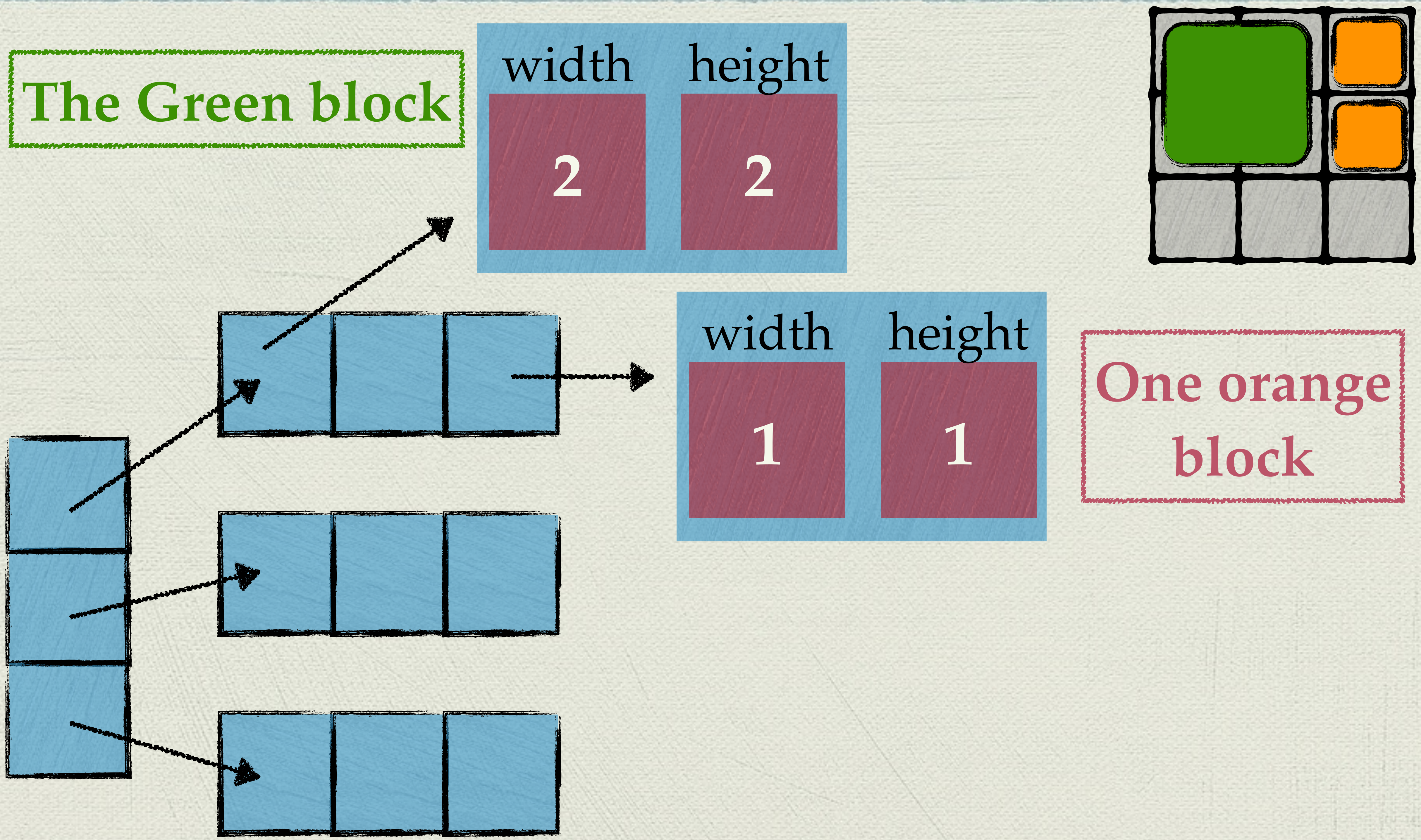
- How would you represent this as an object?



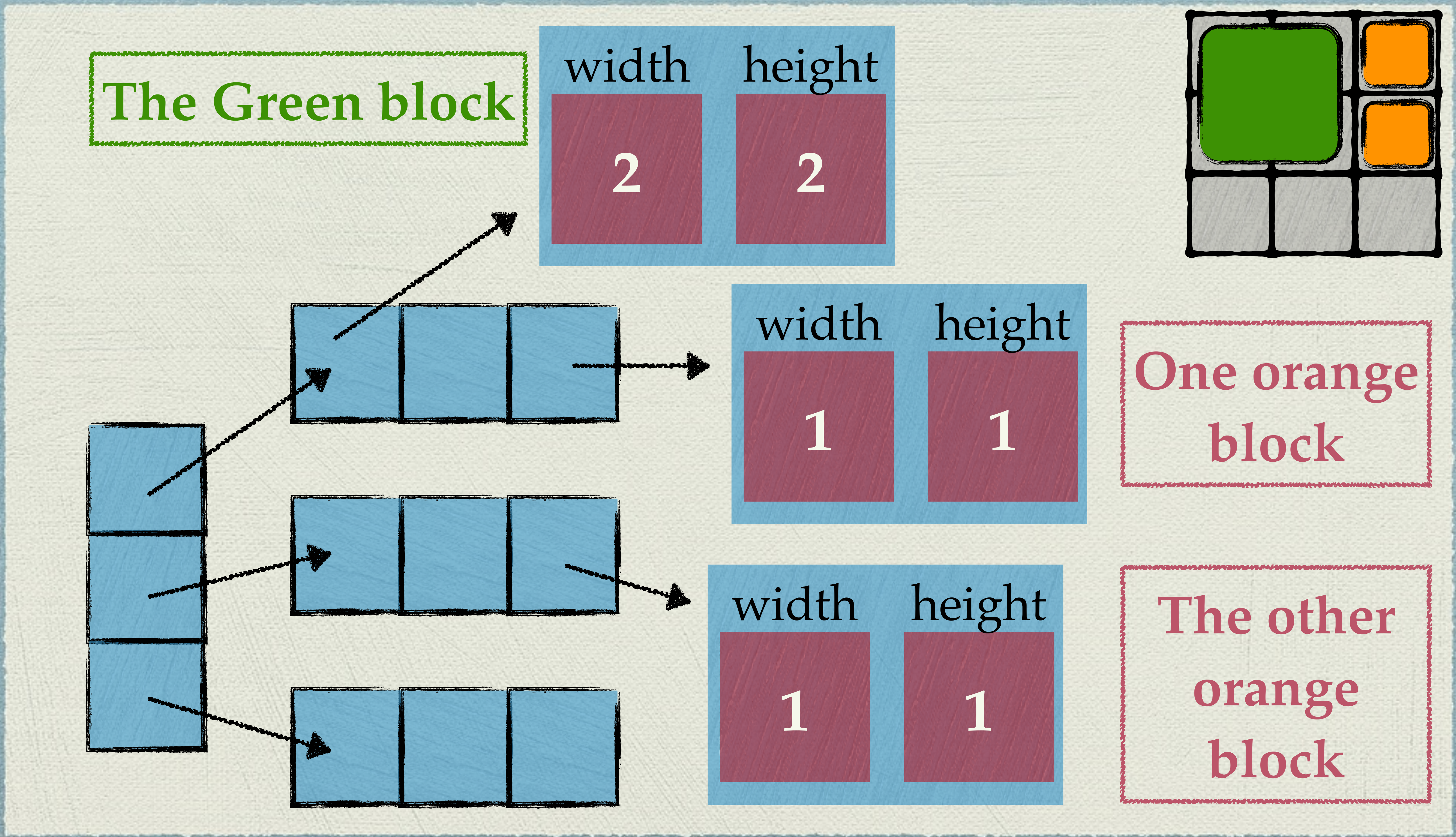
One approach: 2D array of Block objects



One approach: 2D array of Block objects



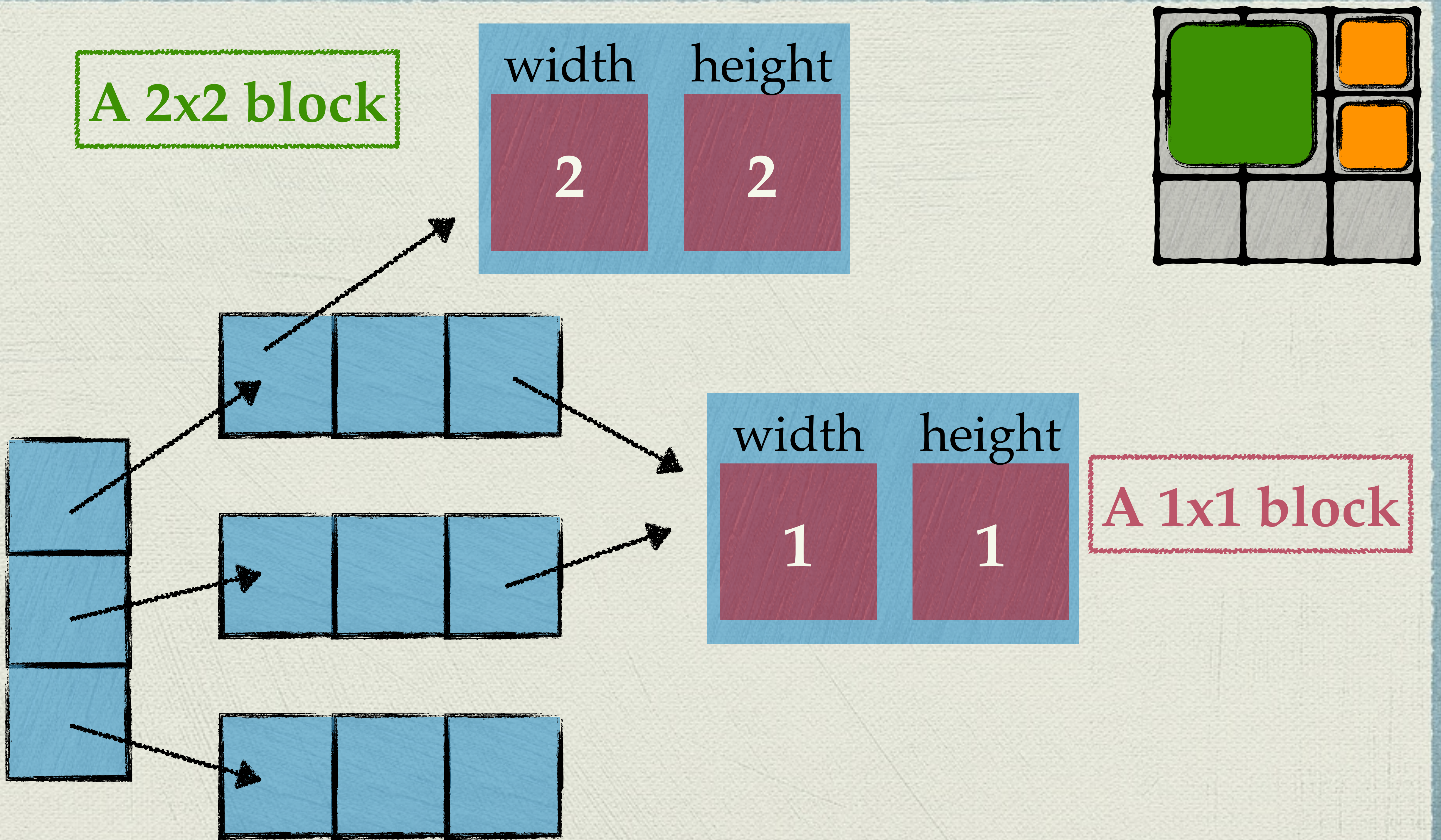
One approach: 2D array of Block objects



One approach: 2D array of Block objects

- But do we really need two objects for the orange blocks? They contain exactly the same information…

One approach: 2D array of Block objects



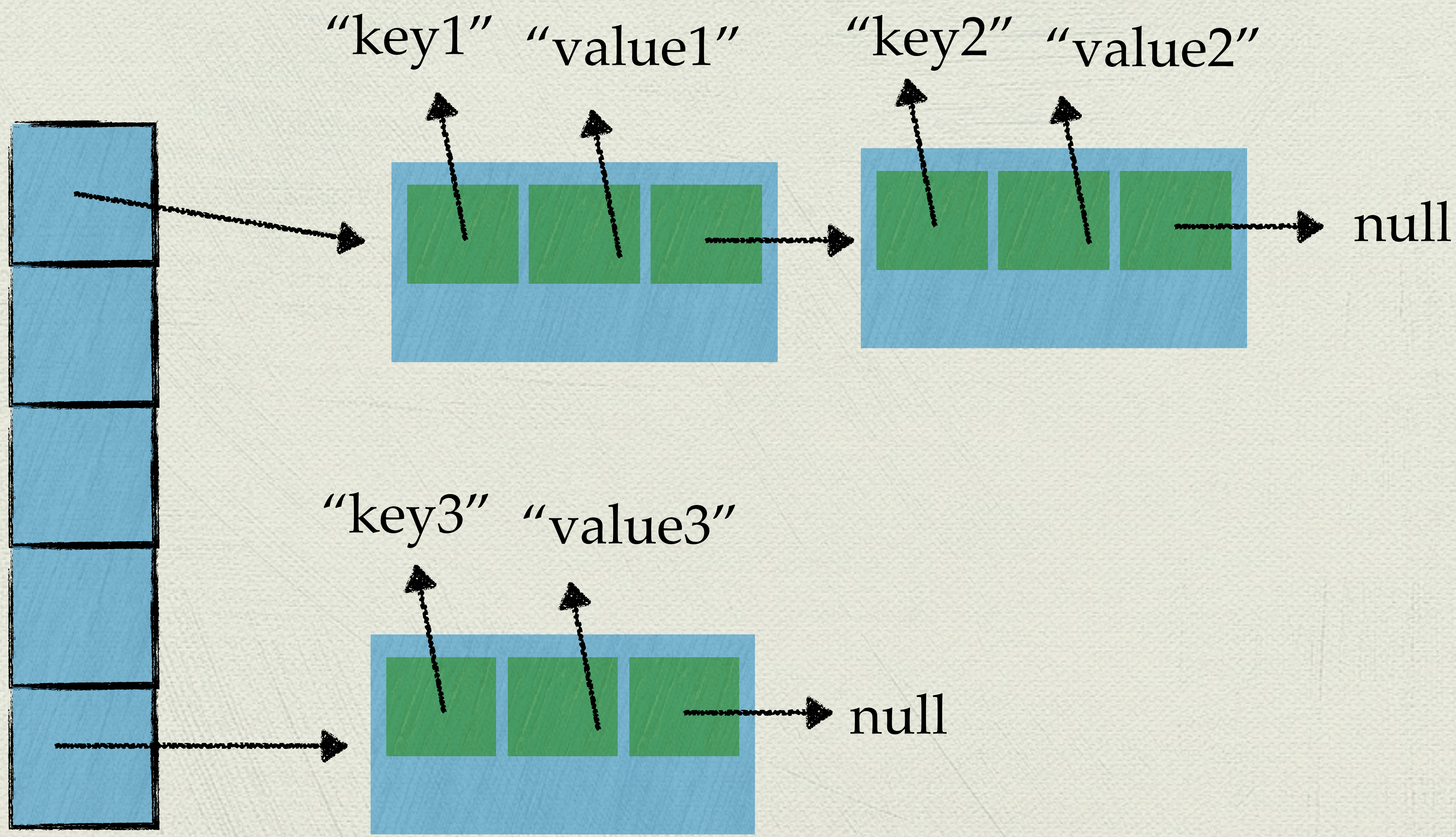
One approach: 2D array of Block objects

- **The point**: think about whether you really need two *objects*, or if you can get away simply with two *references* to one object
- We really would need two objects if blocks could be modified in some way. But in this case, a block never changes, so

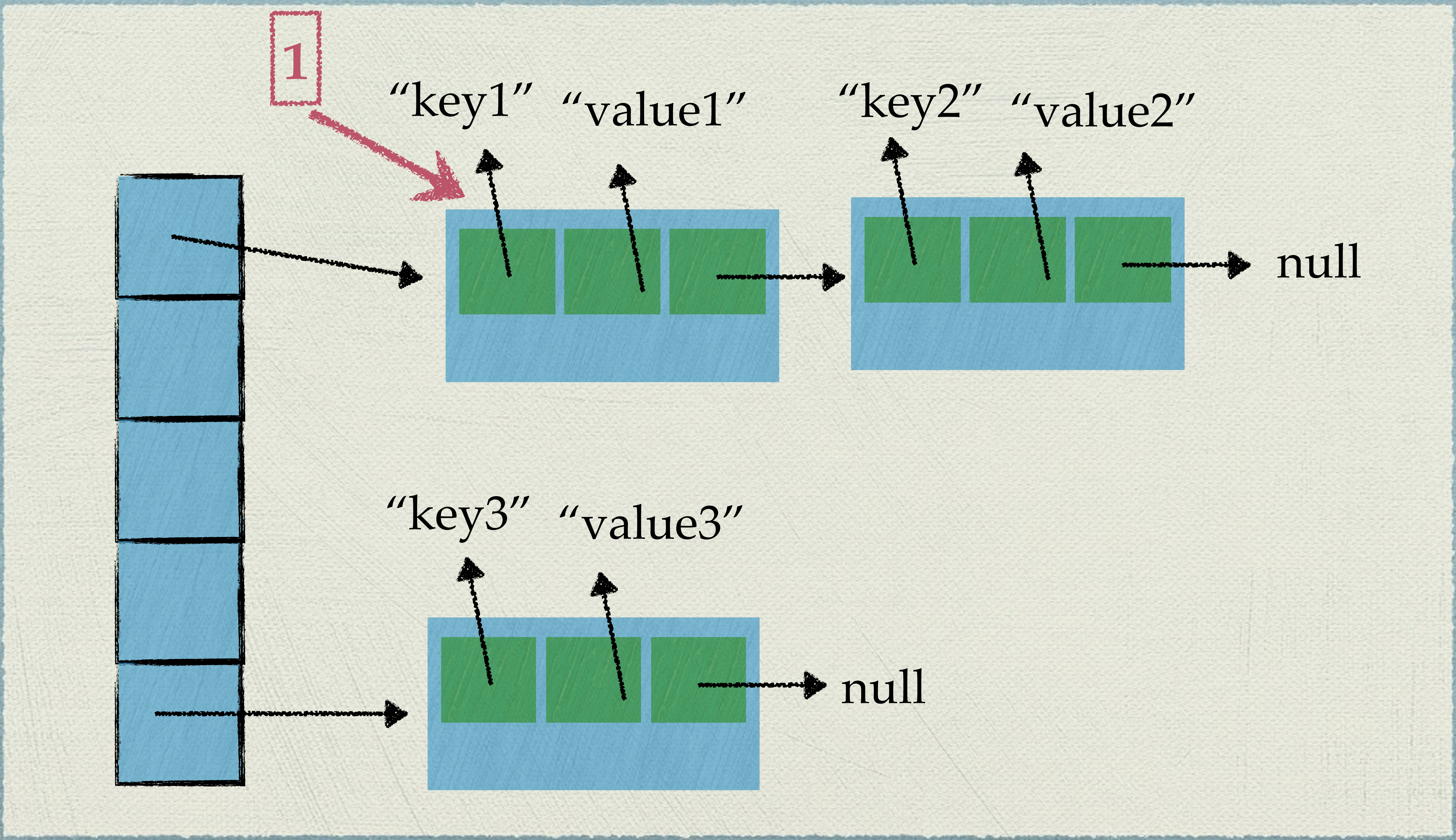
Example: memory efficiency of chaining hash map

- How much memory does it take to store a Key and Value object in a HashMap?
- Let's count the references

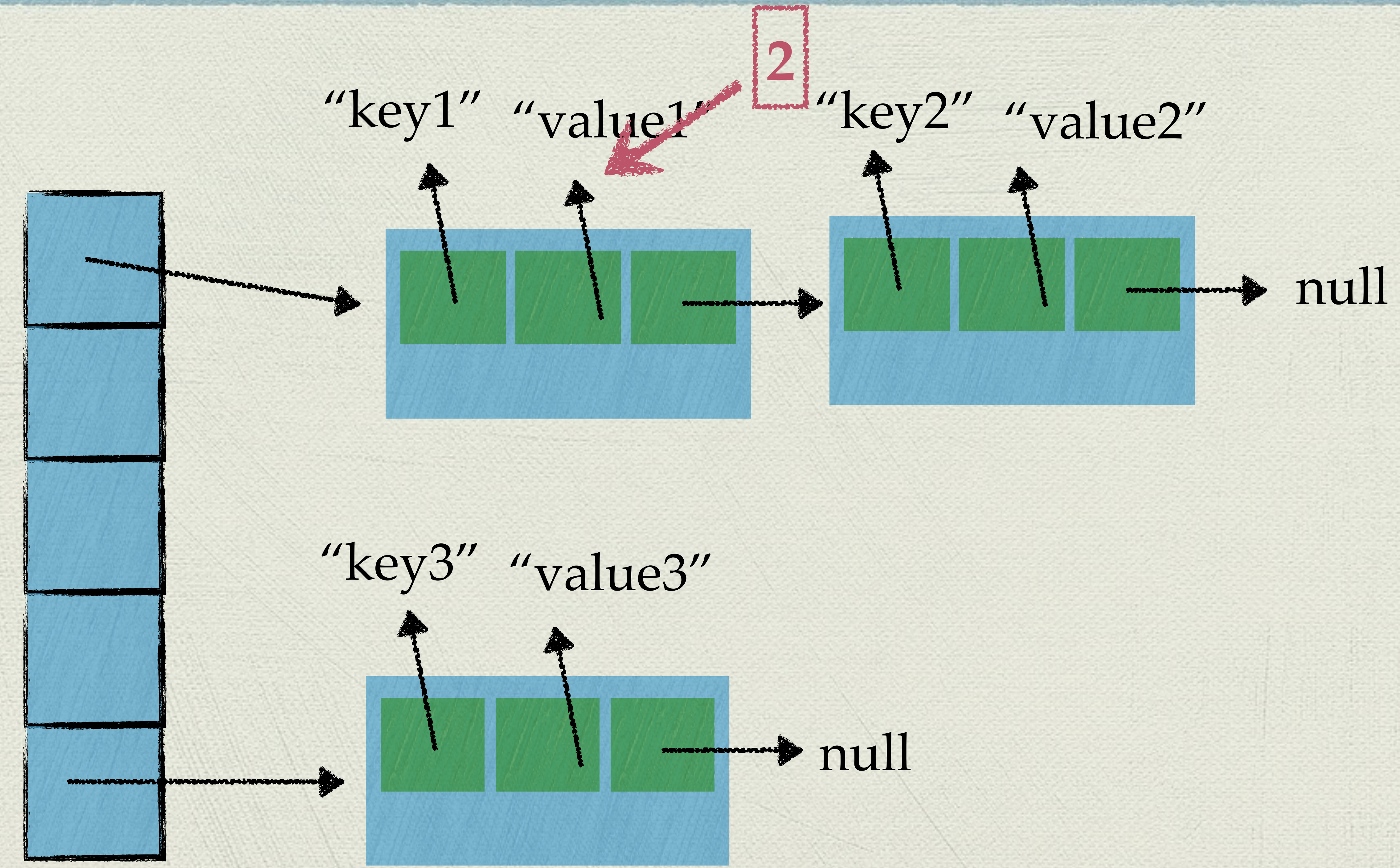
Example: memory efficiency of chaining hash map



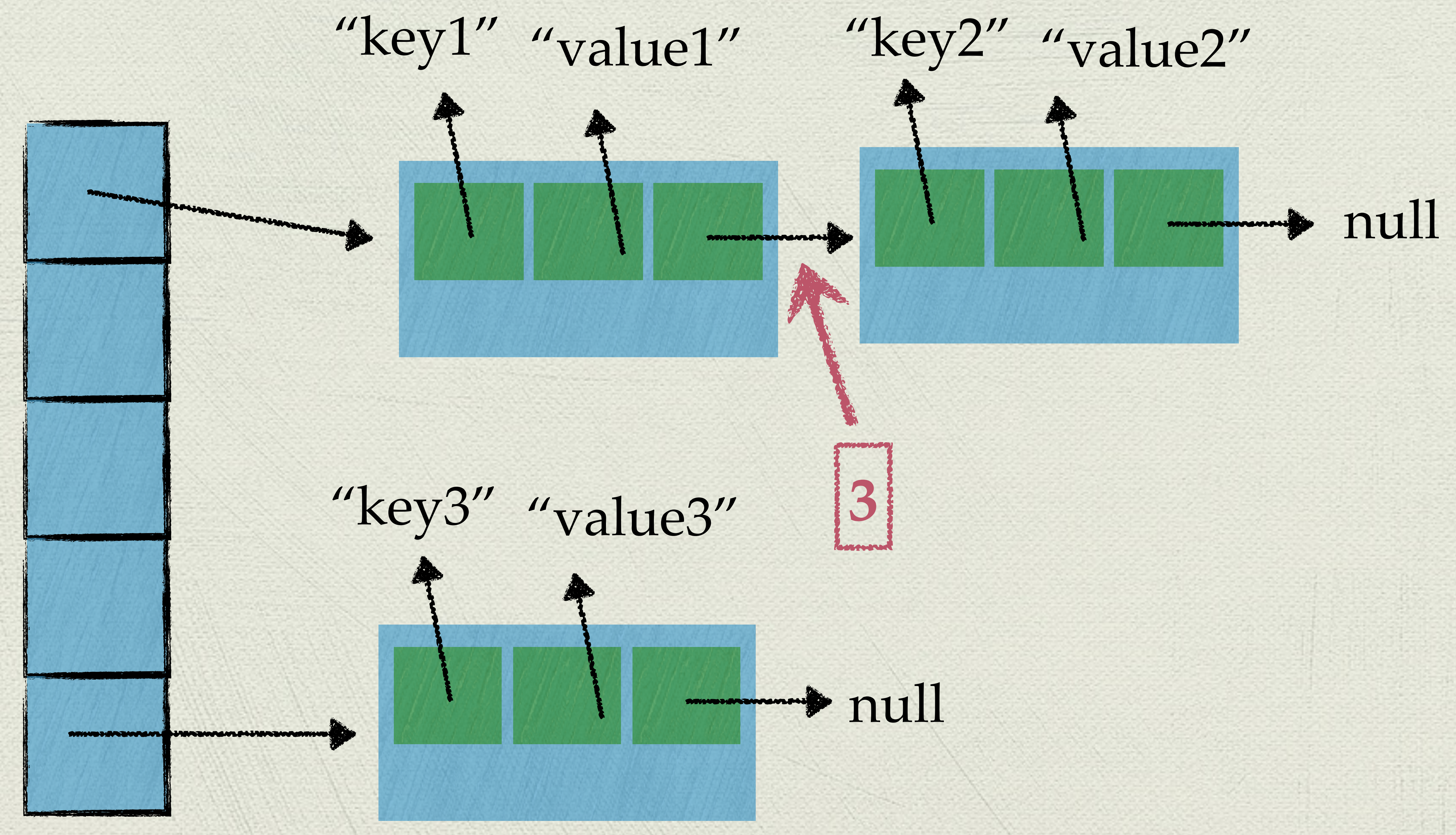
Example: memory efficiency of chaining hash map



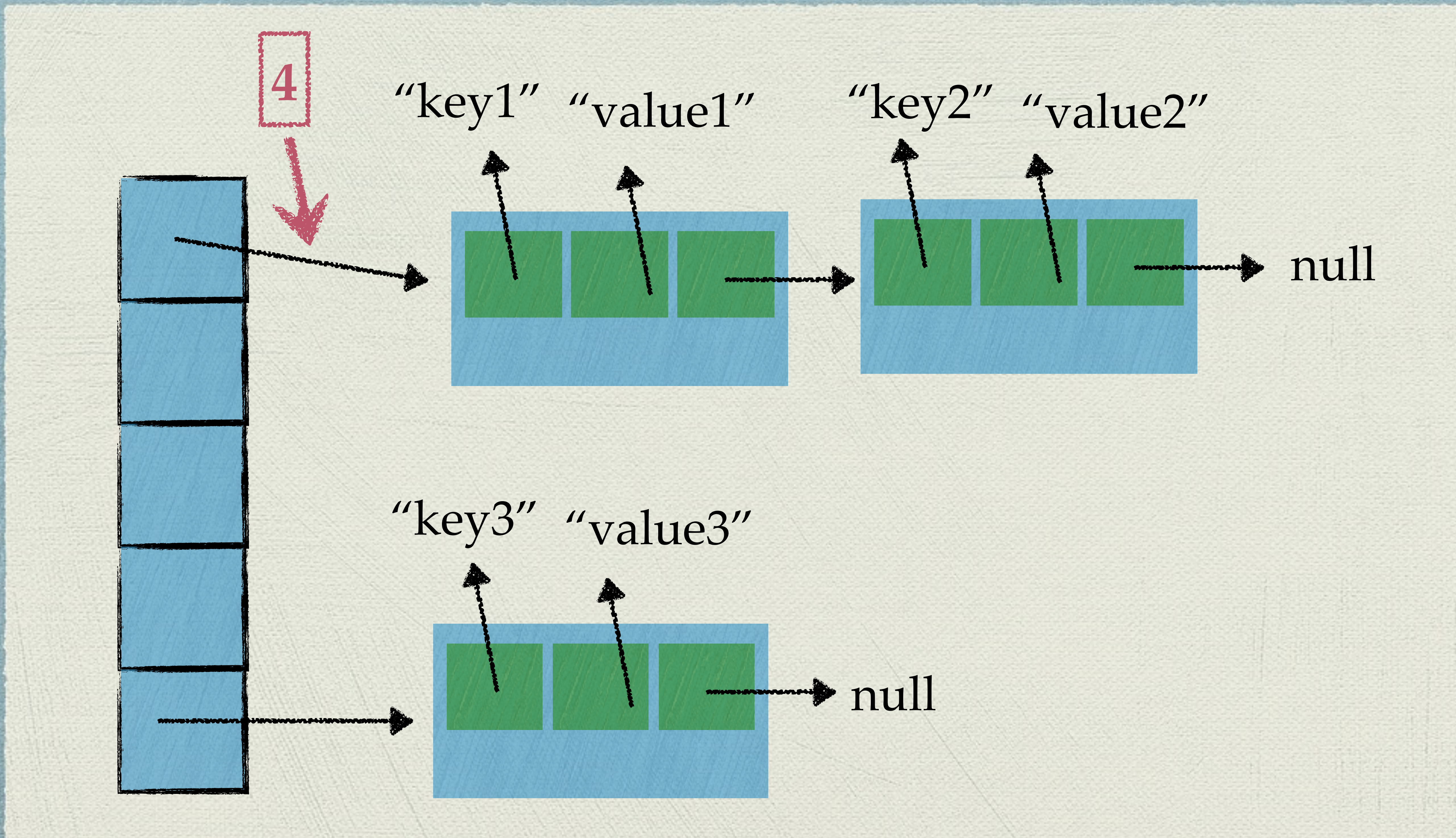
Example: memory efficiency of chaining hash map



Example: memory efficiency of chaining hash map



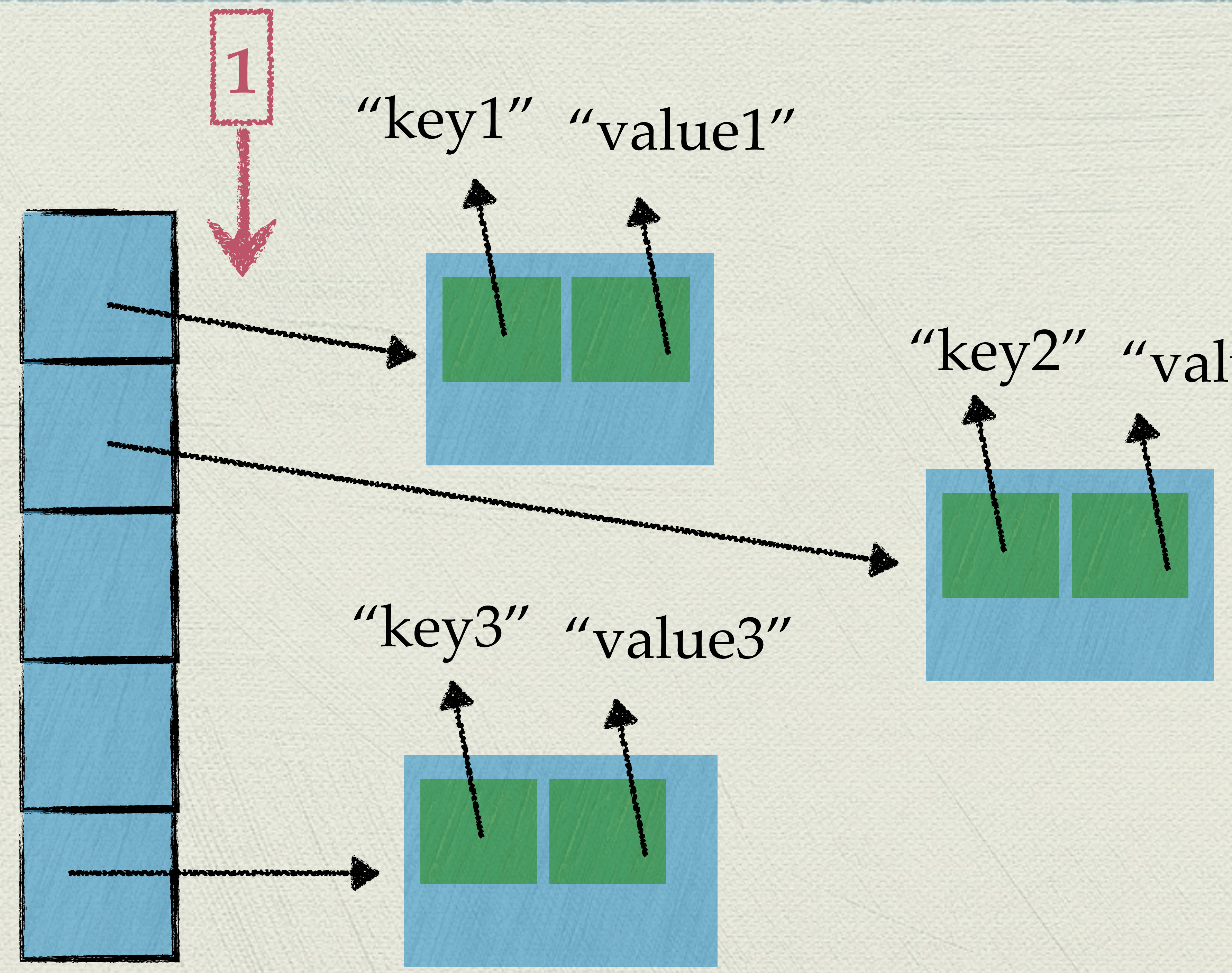
Example: memory efficiency of chaining hash map



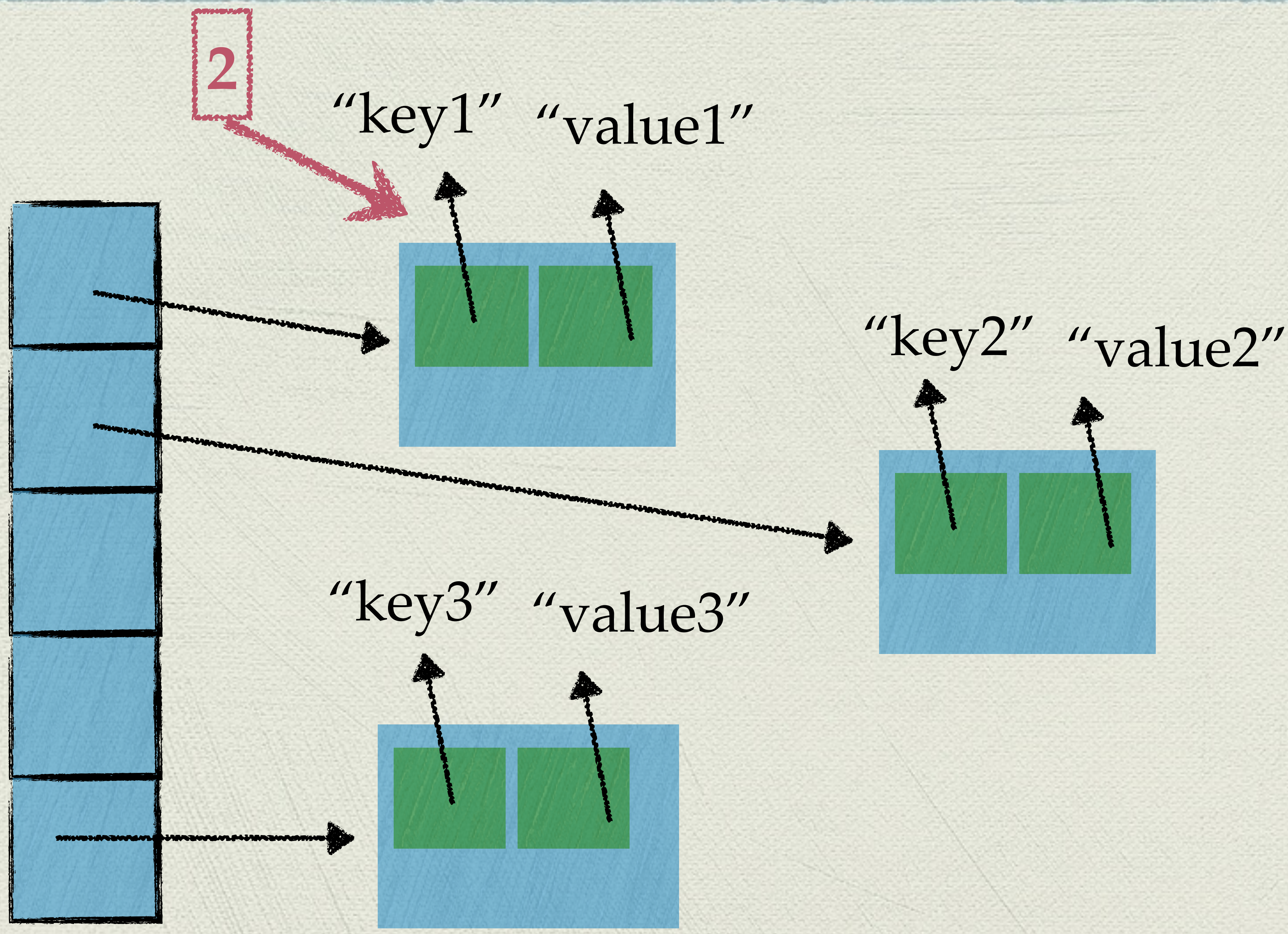
Example: memory efficiency of chaining hash map

- From this picture, we count about **3-4** references per key/value pair
- What if we tried something else? Say, a **linear probing** hash map

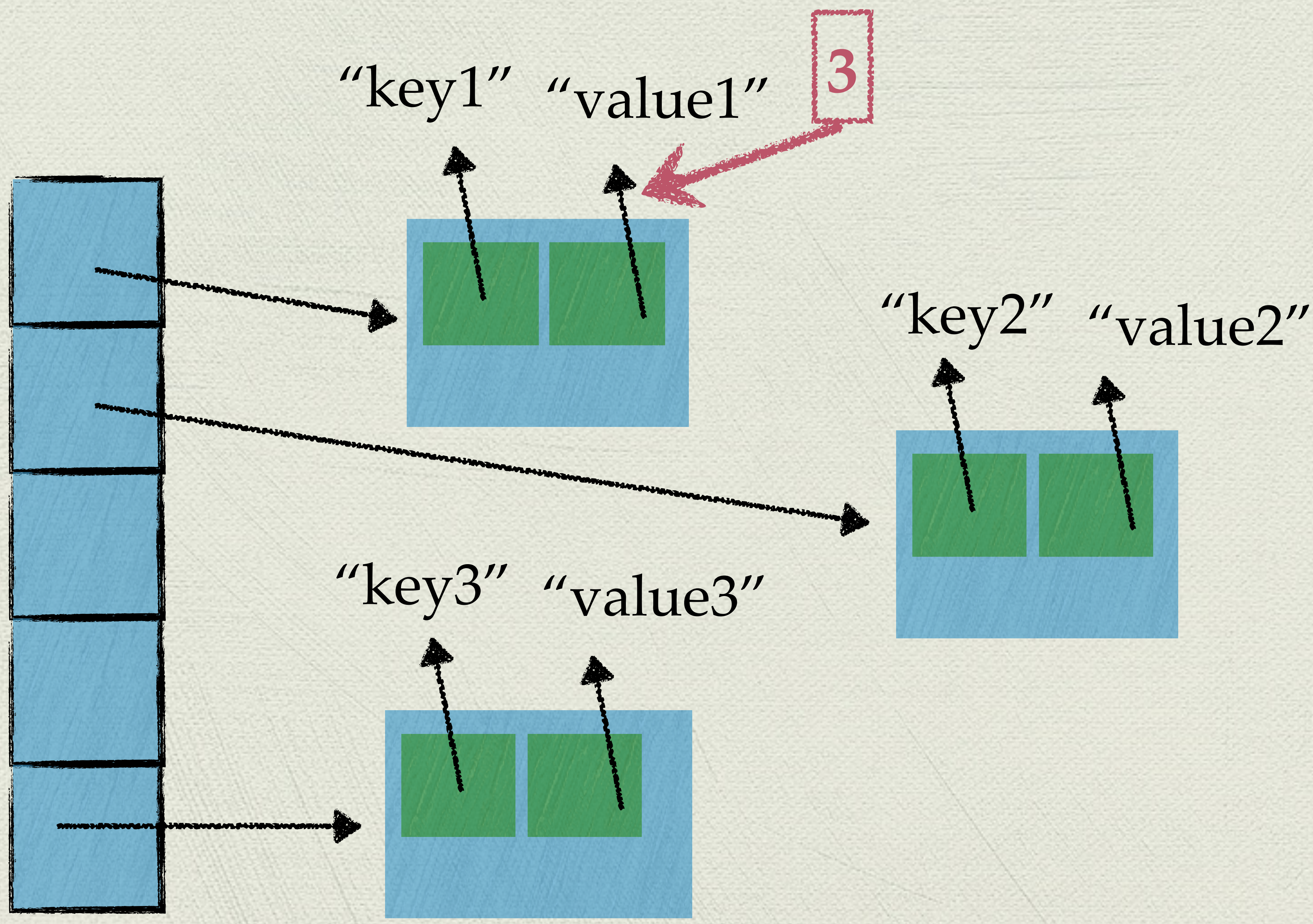
Example: memory efficiency of linear probing hash map



Example: memory efficiency of linear probing hash map



Example: memory efficiency of linear probing hash map



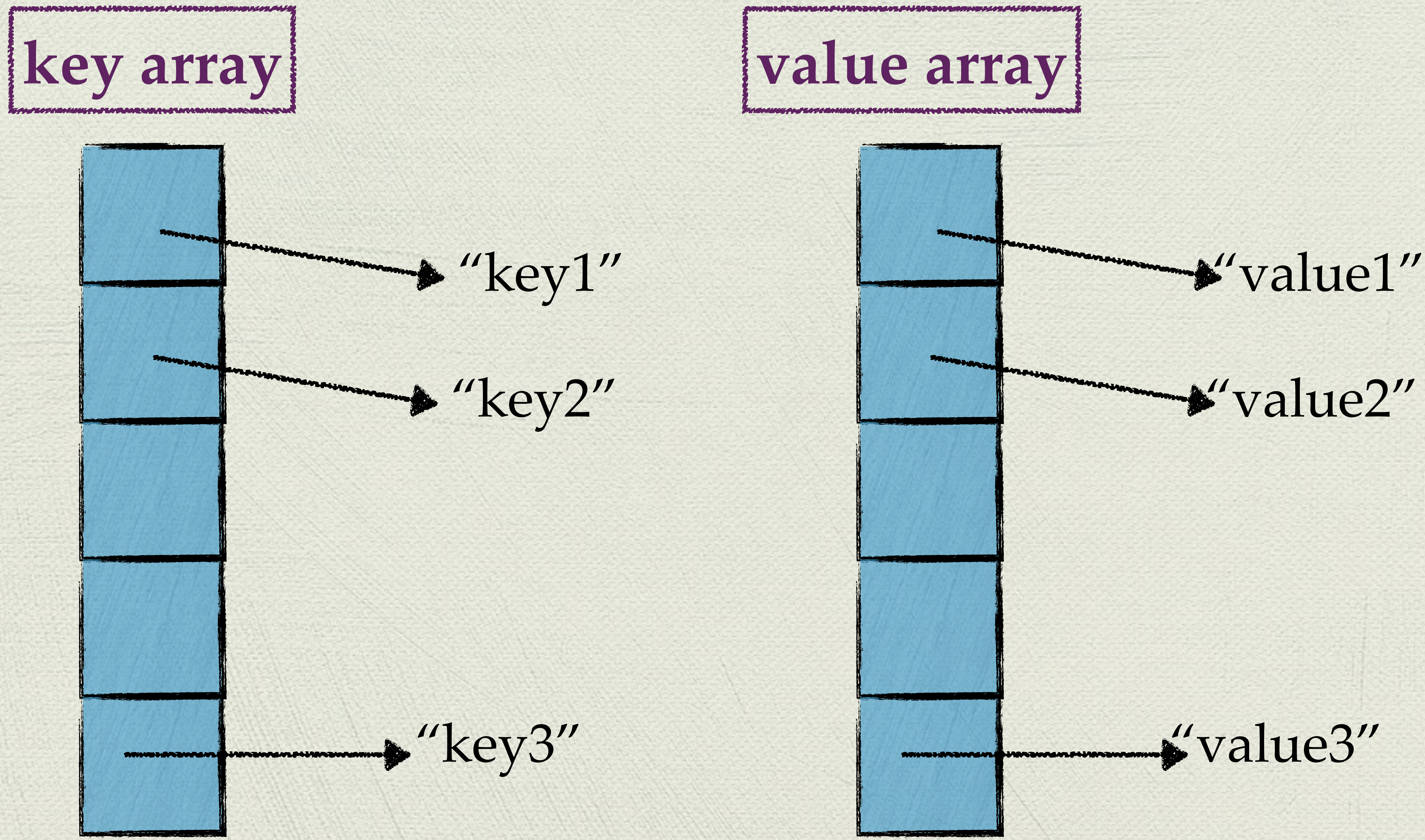
Example: memory efficiency of linear probing hash map

- From this picture, we count about **3** references per key/value pair
- We tried to save memory by not having to store linked list **next** pointers, but there wasn't much effect

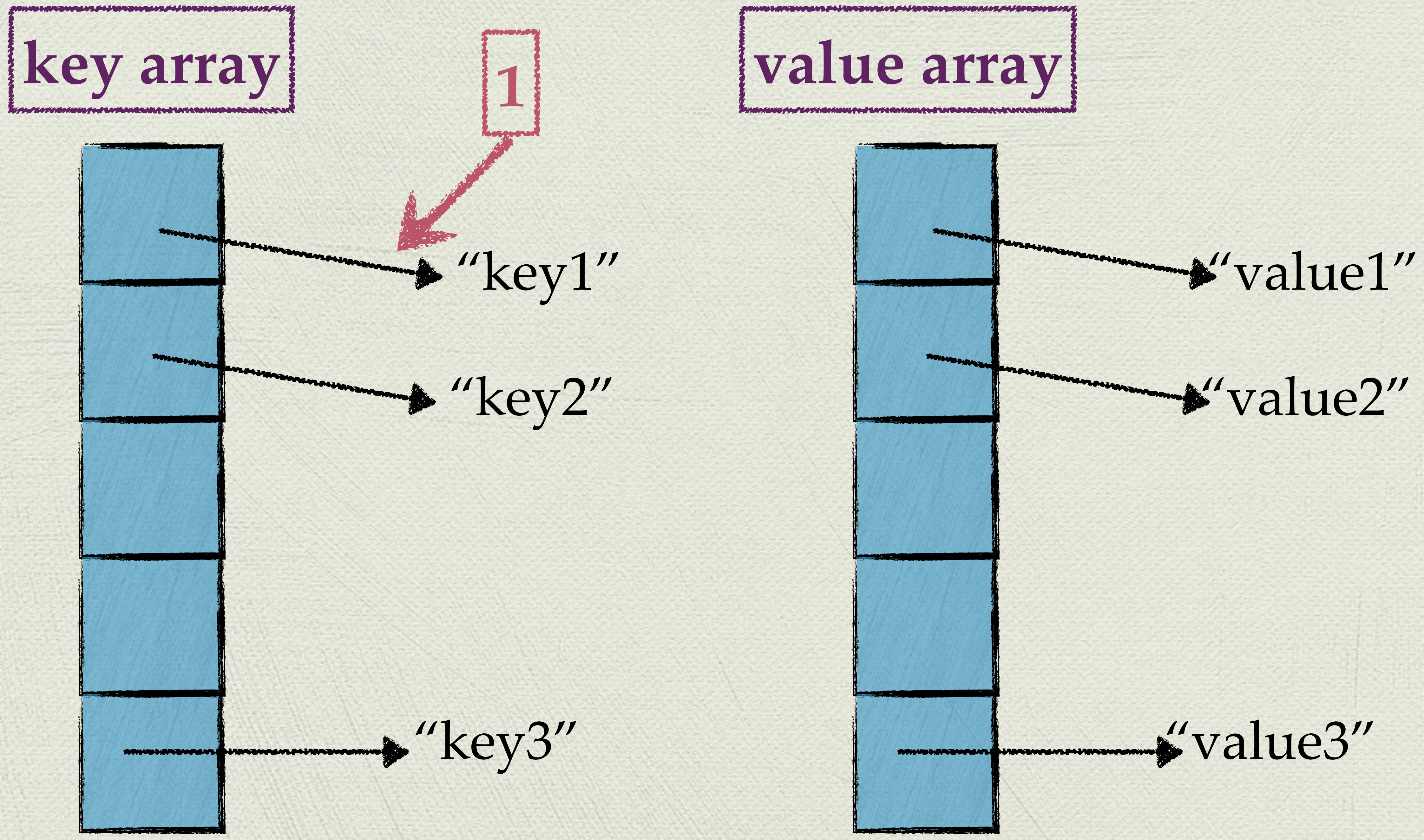
Example: memory efficiency of linear probing hash map

- But this idea isn’t done yet

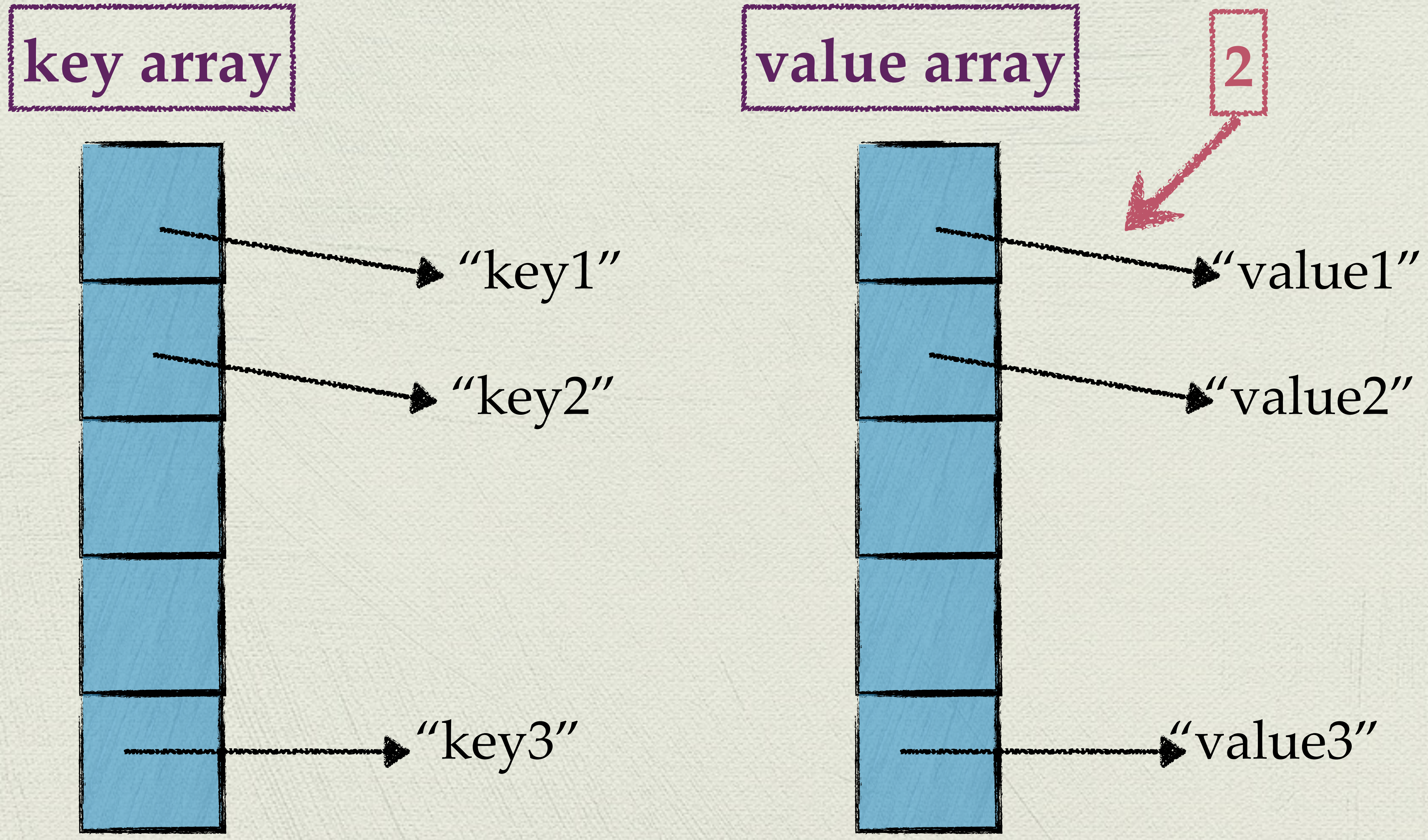
Example: memory efficiency of linear probing hash map



Example: memory efficiency of linear probing hash map



Example: memory efficiency of linear probing hash map



Example: memory efficiency of linear probing hash map

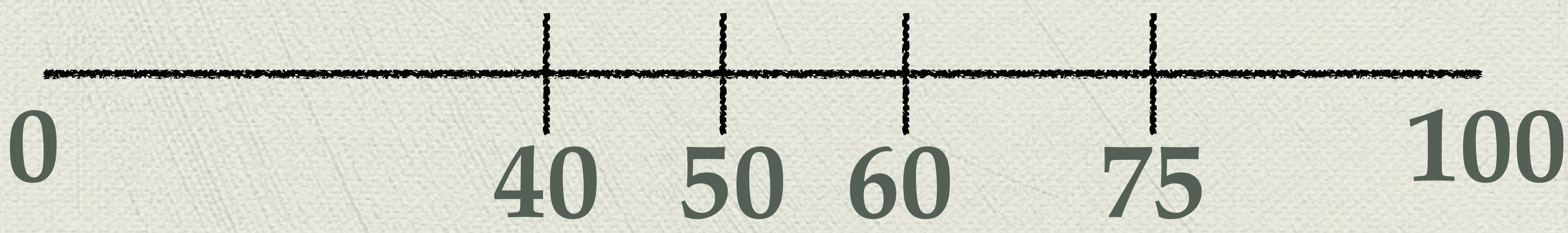
- From this picture, we count about **2** references per key/value pair
- We save memory by not having to storing a KVPair object, but instead storing keys and values in separate arrays

Example: memory efficiency of hash map

- **The point**: Java's chaining HashMap is good enough for most purposes
- But if for some reason you need to be *really* memory efficient, there are lighter-weight options available

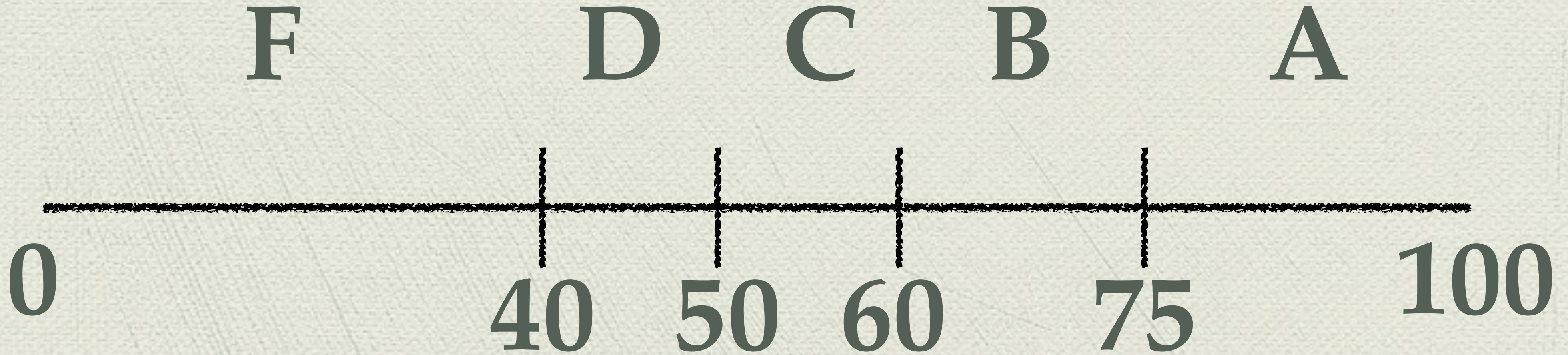
Example problem: ranges data structure

- (also a rejected final problem)
- Consider a number line, from 0 to 100
- You're given a list of numbers that represent marks on this line, for example:



Example problem: ranges data structure

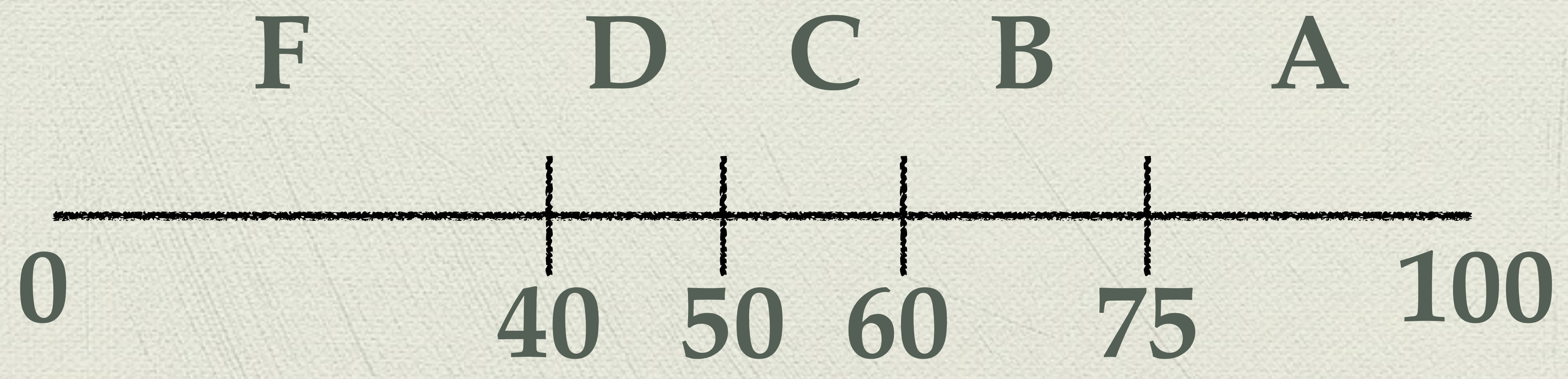
- You're also given **labels** for each region of the number line



Example problem: ranges data structure

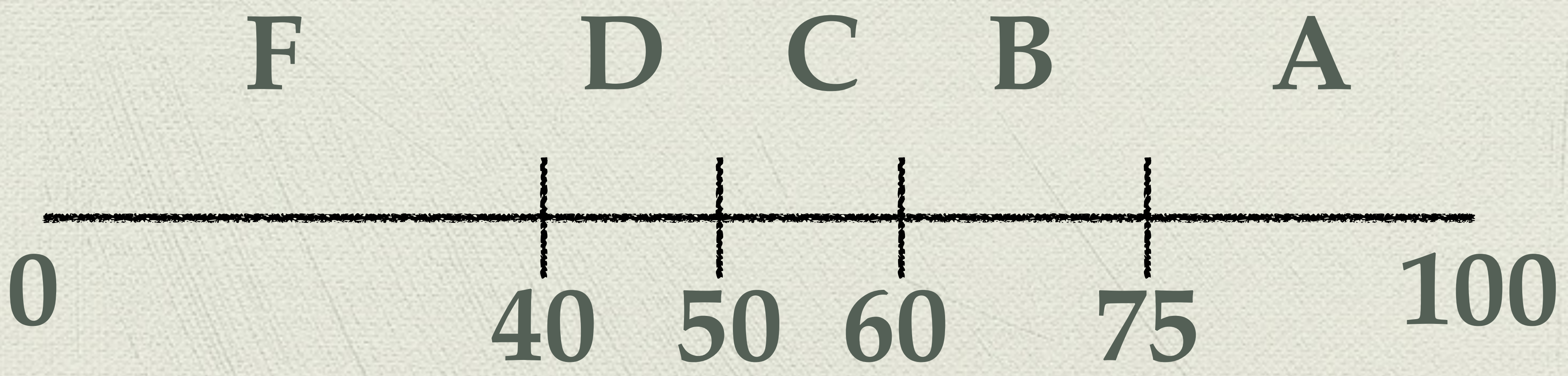
- Design a data structure that can take in a number (say, 88.3), and decide what the label for that number is

- Memory efficiency matters a little bit



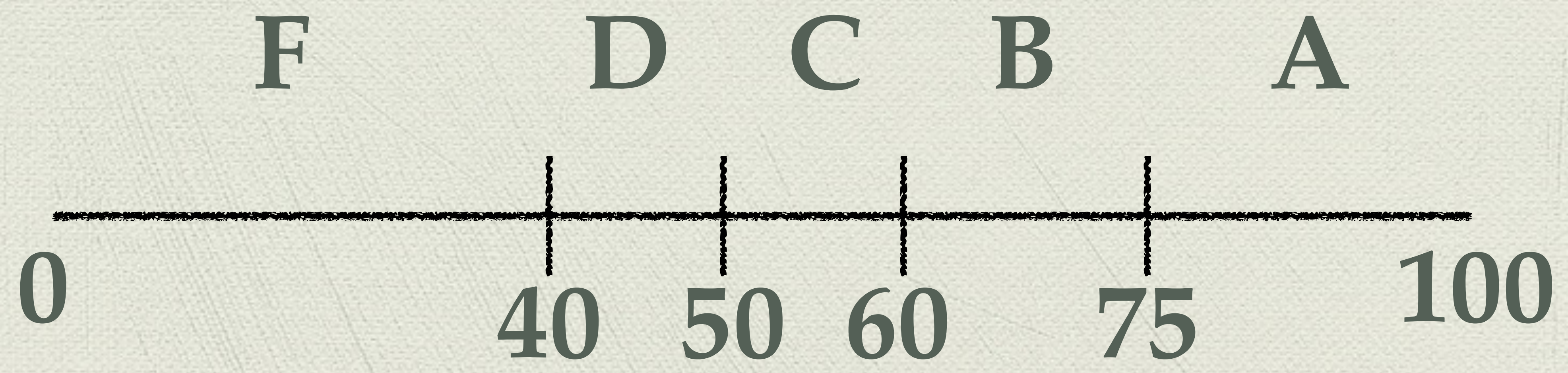
Crazy solution

- Create a `HashMap<Double, String>` from *every possible double* between 0 and 100, to the label
- Gets **O(1)** lookup time!
- But the amount of memory required for this is prohibitively bad

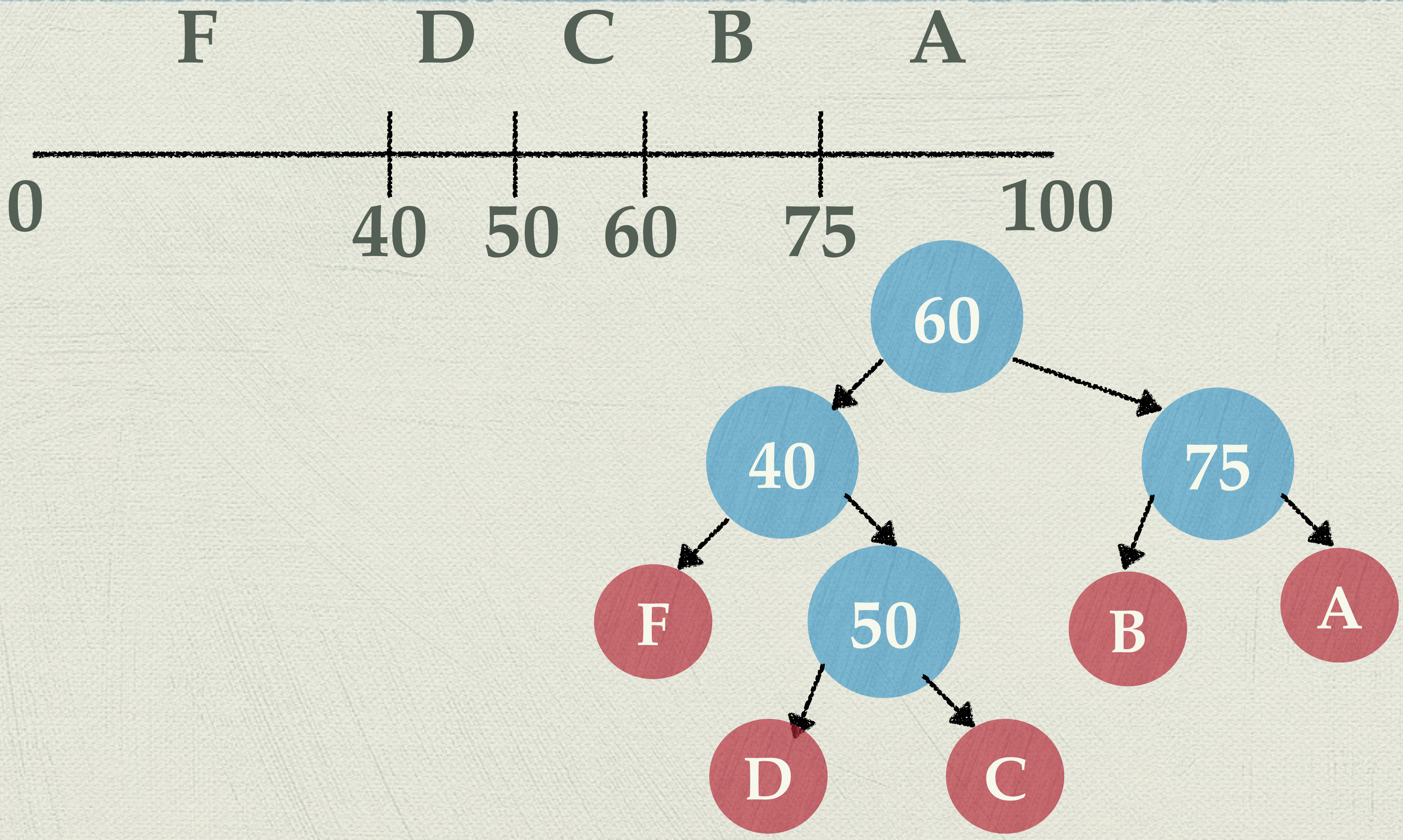


Better solution

- Recognize this is exactly a binary search tree



Better solution



Other tradeoffs

The complexity / simplicity tradeoff

- In this class, we've seen how using more complicated data structures and algorithms can provide asymptotic runtime improvements over simple ones
- For example, the sorting algorithm **merge sort** is faster than the comparatively simple **insertion sort**

The complexity / simplicity tradeoff

- However, these were only **asymptotic benefits**
- What about in the real world?
- Remember that asymptotic benefits only apply when the amount of data is very large

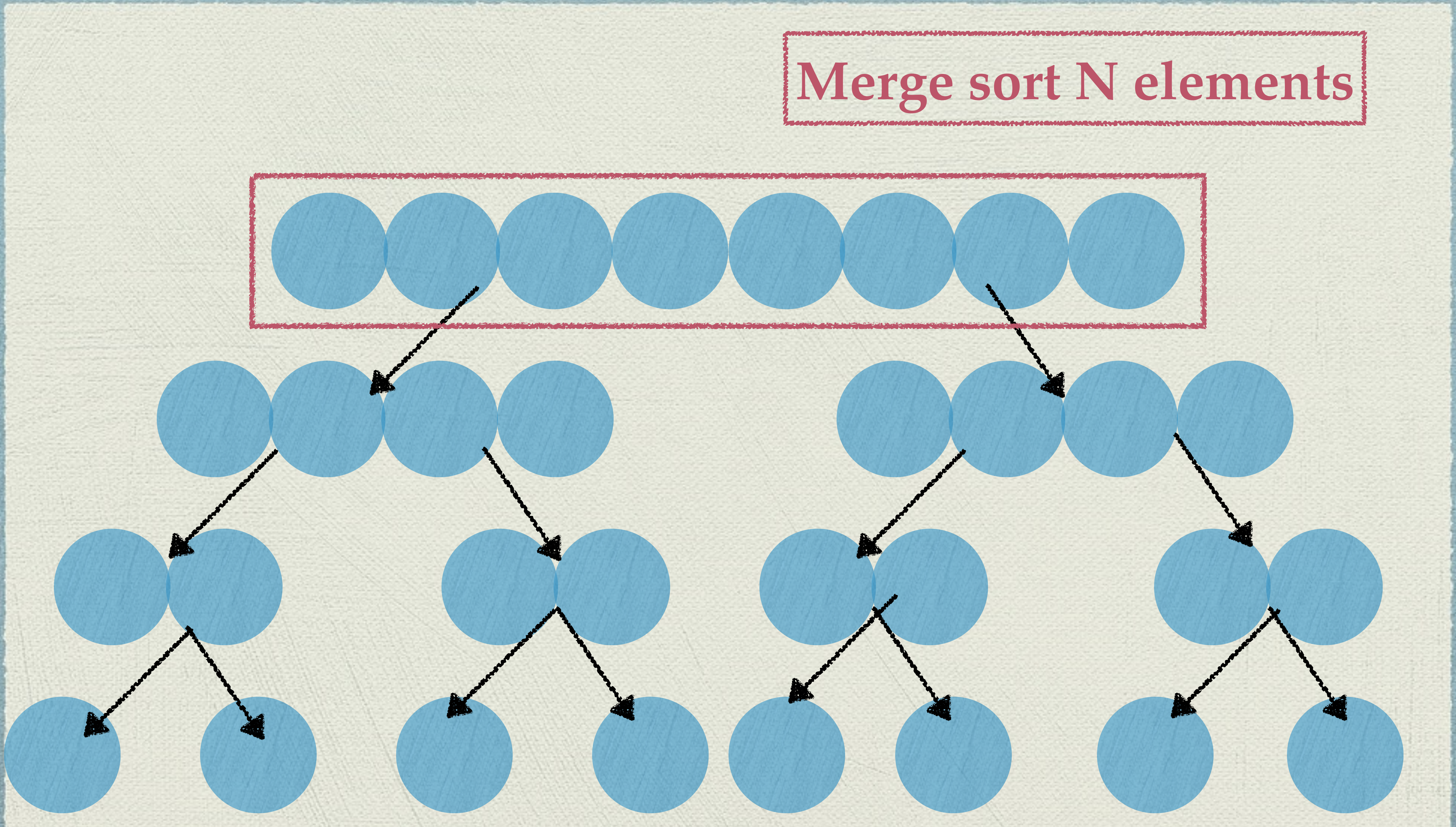
The complexity / simplicity tradeoff

- It turns out, simpler solutions work better on small amounts of data
  - Less overhead getting started, etc.
- *Does this really matter? I thought we only cared about large data*

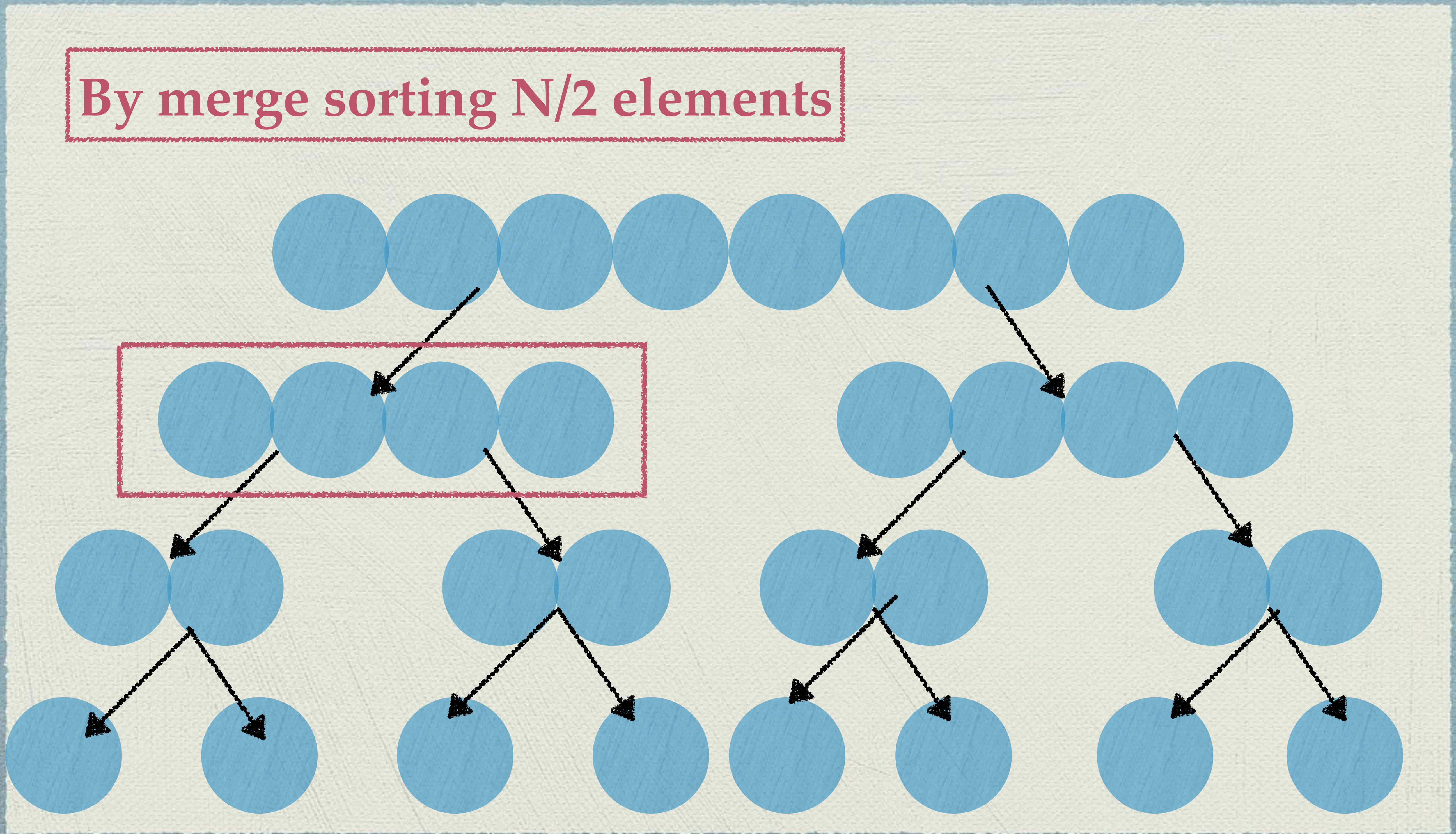
The complexity / simplicity tradeoff

- Consider merge sort

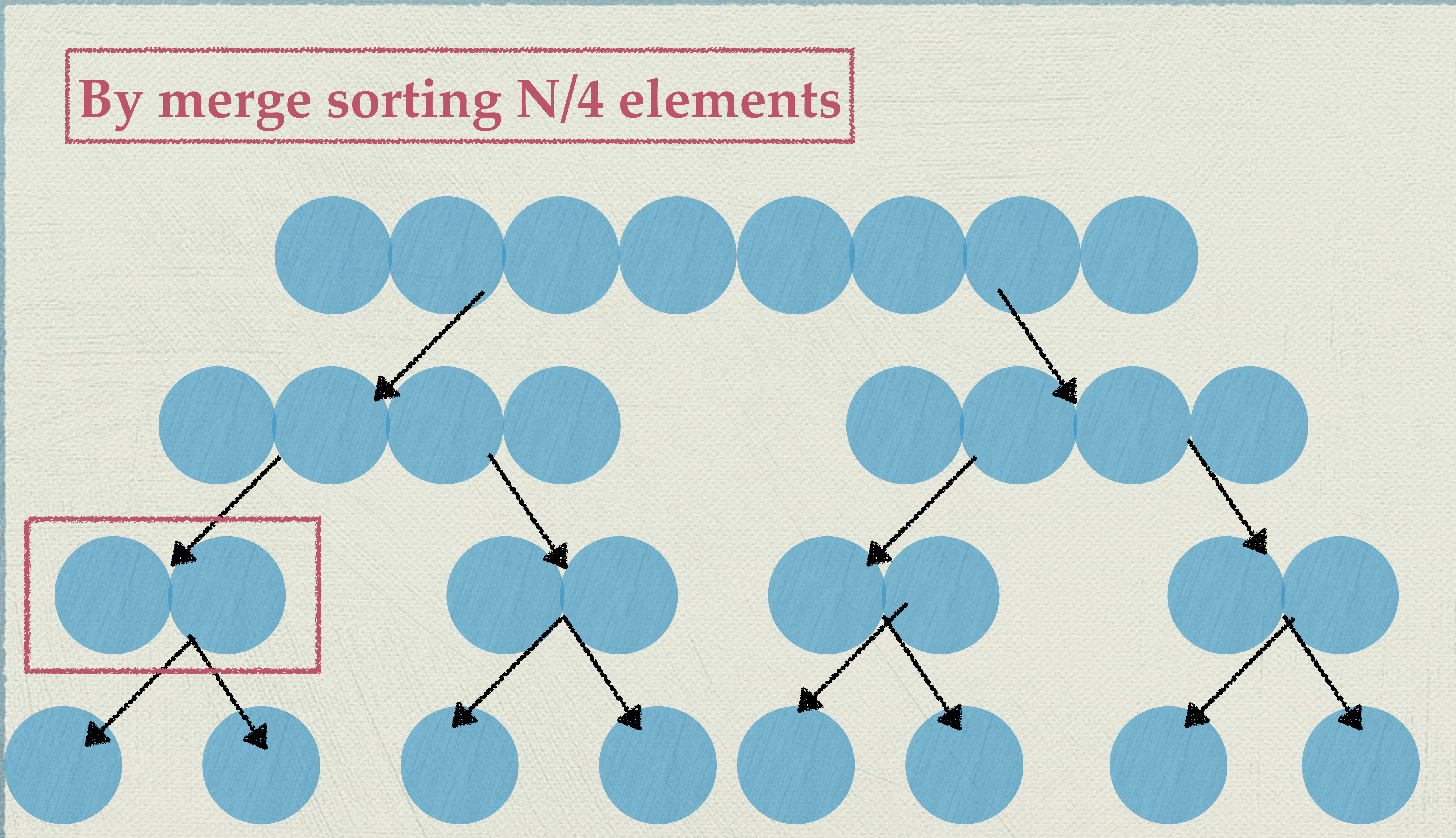
Merge sort



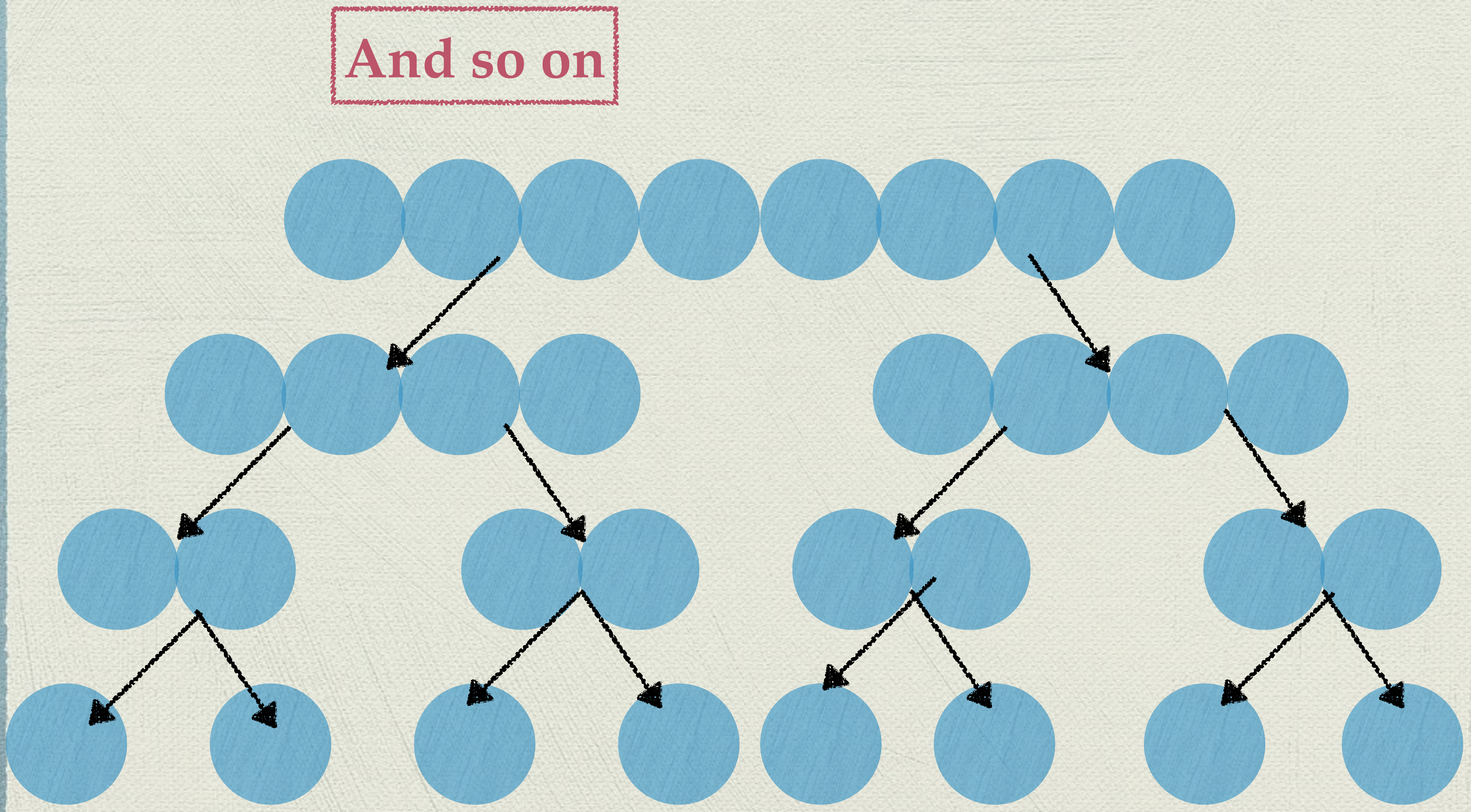
Merge sort



Merge sort



Merge sort



Merge sort complexity

- Even when we merge sort a very large list, we eventually end up merge sorting very small lists
- But is the complexity of merge sort really necessary to sort small lists?
  - If you're sorting a list of length 8, do you *really* have to split in in half 3 more times?

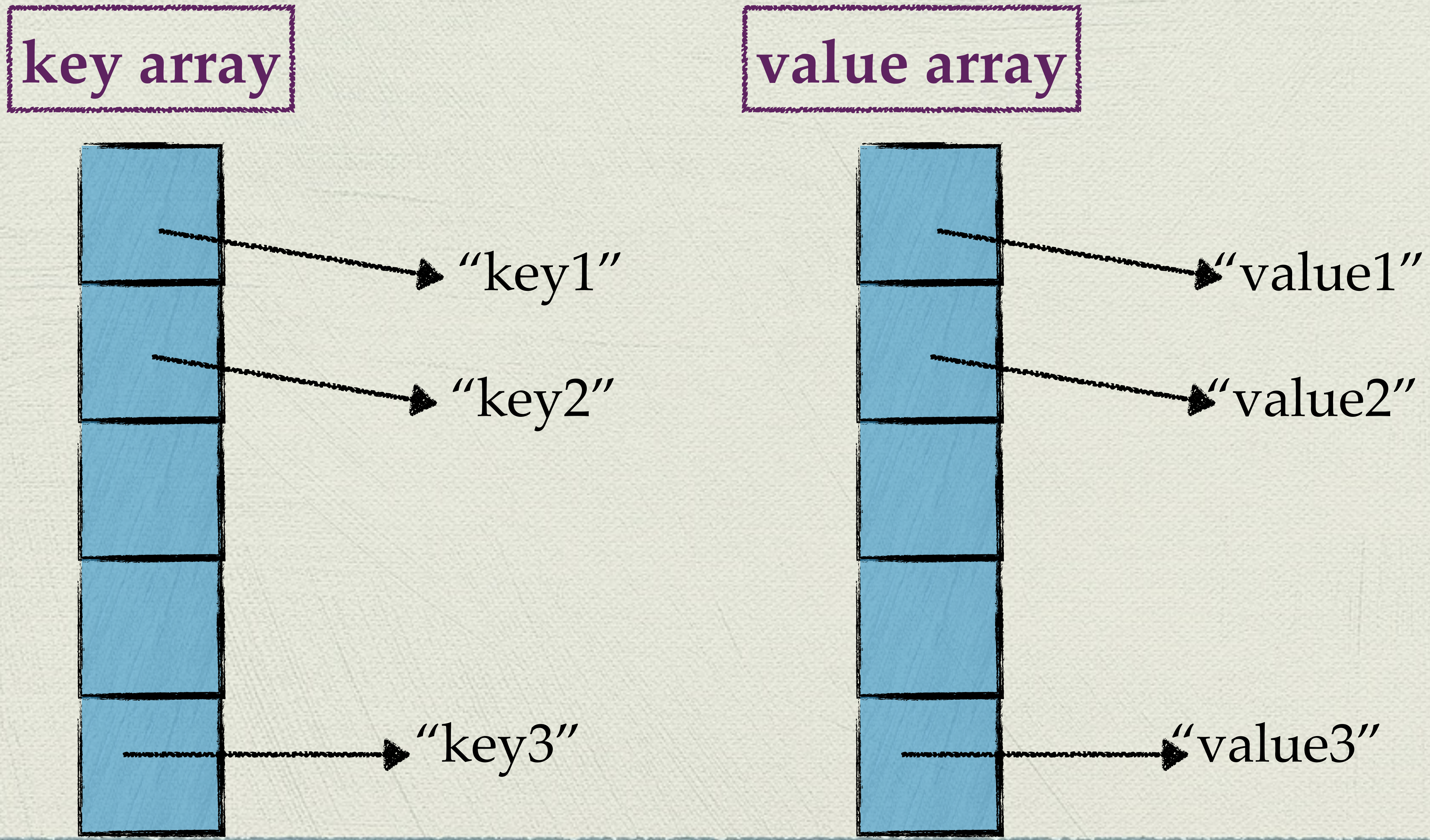
Merge sort complexity

- **Idea**: Merge sort on the big lists, but when the list gets broken down to smaller sizes, insertion sort them
- This is closer to what people do in practice. *Adapt* the sorting algorithm based on conditions

And now, introducing…

My actual favorite data structure

- This hash map (from earlier)



My actual favorite data structure

- This hash map (from earlier)
- With a special hashing scheme

My actual favorite data structure

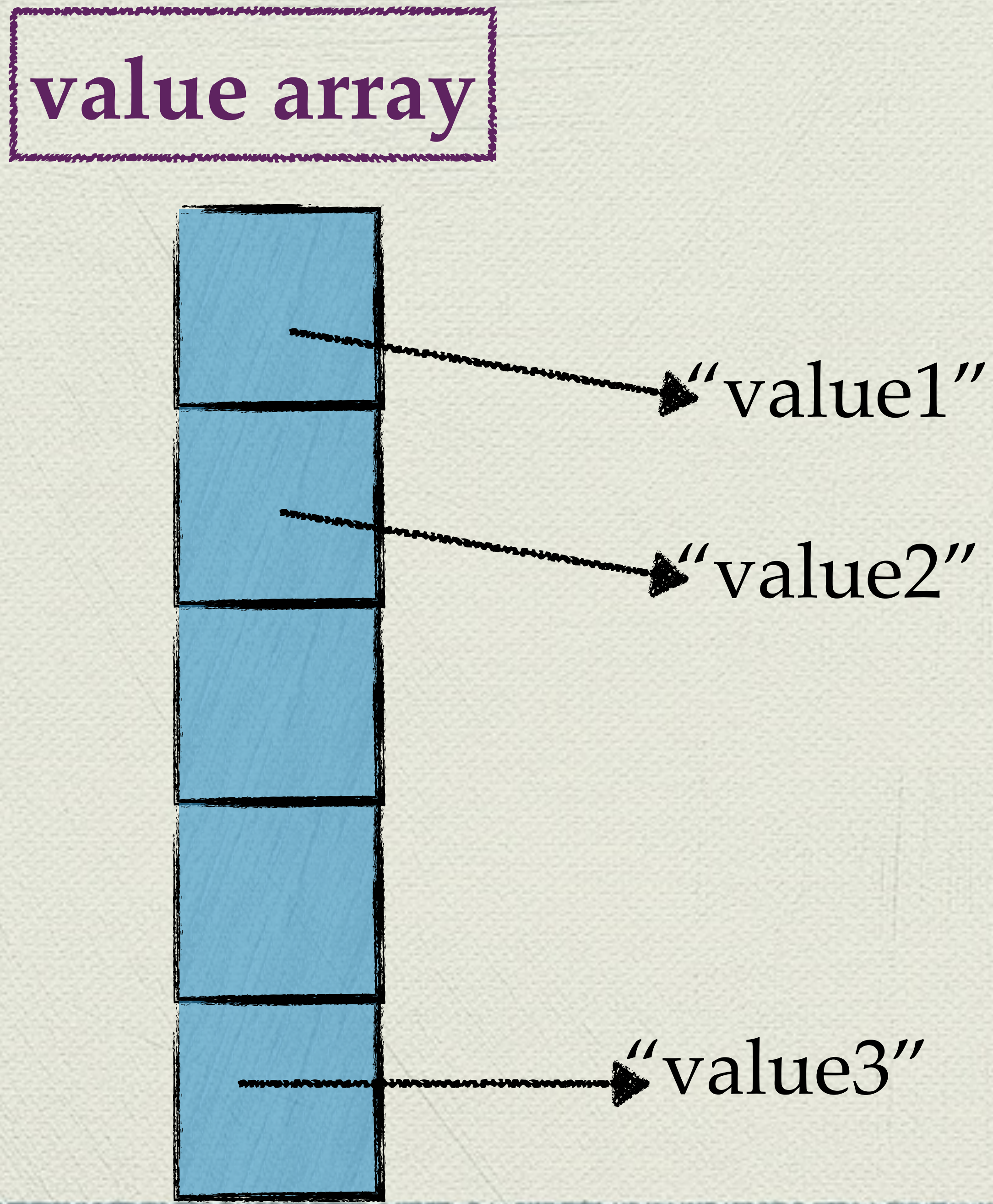
- This hash map (from earlier)
- With a special hashing scheme
- …called “hash and hope”

My actual favorite data structure

- **The idea**: When you call `put(key, value)`, if you happen to get a collision, just ignore it, and overwrite the value
- When you call `get(key)`, don't check for key equality, just return the value that's there regardless

My actual favorite data structure

- So actually, we didn't need the key array



My actual favorite data structure

- What's great about this data structure
  - **By far the fastest map** (no need to handle collisions: always O(1), no worst case)
  - **By far the most memory efficient map** (just a single array)
  - **By far the easiest map to code**

My actual favorite data structure

- Any cons?
  - (it might sometimes return the wrong answer)

My actual favorite data structure

- But for certain applications, “hash and hope” might not be so crazy

My actual favorite data structure

- But for certain applications, “hash and hope” might not be so crazy
- **Example**: Maintain a map from words to their probability of occurrence in a language
  - This is useful in translation

My actual favorite data structure

- What happens with collisions?

My actual favorite data structure

- If two words with similar probabilities collide, then there’s not much of a problem, because the values were similar anyway

My actual favorite data structure

- Say a very common word collides with a very uncommon word
- Since the common word is more common, you'll be putting it into the map more often. This means, it's more likely that the value in the map is the value for the common word, not the uncommon word

My actual favorite data structure

- The common word will usually have the right value
- The uncommon word will usually have the wrong value

My actual favorite data structure

- But it isn’t so bad for the uncommon word to have the wrong value
  - Because it’s so uncommon, returning the wrong answer will be a rare occurrence
  - So it probably won’t affect our analysis much, anyway

My actual favorite data structure

- **The point:** "Hash and hope" is far from your standard map…
    - …but, sometimes you can get a little crazy with highly specialized applications

Data structure tradeoffs

- **Runtime for one operation** vs. **runtime for another**
- **Speed** vs. **memory**
- **Simplicity** vs. **complexity**
- **Efficiency** vs. ***getting the right answer***

If you remember nothing else from this class

- Checking if a list contains something is **not fast**

Oh by the way, if you were curious