Table of Contents

CHAPTER 1

Why Data Structures Matter for Mobile Developers

Picture two developers given the exact same task: build a contacts app that holds ten thousand names. They use the same language, the same screen design, the same icons. Yet when you scroll through the finished apps, one glides like glass and the other stutters every few seconds. The user of the second app does not know what a data structure is, but they can feel that something is wrong, and they quietly switch to a competitor.

The difference is almost never the framework or the design. It is the invisible layer underneath: how the app stores its data, and how it reaches that data when the screen needs it. That invisible layer is what this book is about, and this chapter explains why it matters far more on a phone than almost anywhere else.

A server in a data center has enormous memory, unlimited power from a wall socket, and active cooling. A phone has none of those luxuries. It runs on a battery, it shares a small pool of memory with dozens of other apps, and it has to redraw the screen sixty times every second while sitting in someone's warm pocket. On a phone, a careless decision is not hidden by powerful hardware. The user feels it instantly, as lag, as heat, or as an app that suddenly closes itself. Data structures are the tool that lets you respect those limits.

Memory and battery constraints

The first hard truth of mobile development is that the memory your app uses is not really yours. It is borrowed. The operating system hands you a slice of a shared pool, and it can take that slice back whenever it decides another app needs it more.

On Android, when memory runs low, the system sends your app signals through callbacks like onTrimMemory(), and if pressure keeps rising, the low memory killer simply terminates your process. On iOS, the system sends a memory warning, and if your app keeps growing, a watchdog called Jetsam ends it without ceremony. In both cases the user sees the same thing: they switch back to your app and it has restarted from scratch, losing their place. They blame your app, and they are right to.

This is where data structures enter the picture, because the structure you choose decides how much memory the same information occupies. Imagine holding ten thousand user records. If each record is stored in a heavy, loosely packed structure with lots of pointers and wrappers, the total footprint can be several times larger than the same data held in a lean, compact structure. Same information, very different memory cost. On a server nobody notices. On a phone, the bloated version is the one that gets killed in the background.

Battery is the second constraint, and it follows a simple rule: every instruction the processor runs costs a tiny amount of energy, and those tiny amounts add up. A slow algorithm does not just feel slow, it physically drains the battery and warms the device, because the processor has to stay awake and work harder for longer.

Here is a concrete example. Suppose your app needs to check, for each item in a list, whether that item already exists somewhere else in the app. A beginner often writes one loop inside another loop: for every item, scan the whole list again. With one hundred items that is ten thousand comparisons, which is fine. With ten thousand items that becomes one hundred million comparisons, and the phone gets hot. The very same task, done with the right data structure (a hash set, which you will meet in Chapter 7), needs only about ten thousand steps total. The user sees a fast app with a cool battery, and they never know why. That invisible win is the entire job.

Impact of UI thread vs background thread

Every mobile app has one special thread of execution whose only job is to keep the screen alive. On Android it is called the main thread or UI thread. On iOS it is the main thread, driven by the main run loop. Two different names, exactly the same idea: this is the one thread allowed to touch the screen, and it must never be kept waiting.

To understand why, you need one number. A smooth screen refreshes about sixty times per second, which means the main thread has roughly sixteen milliseconds to prepare each frame. Many modern phones now refresh at one hundred and twenty times per second, cutting that budget to about eight milliseconds. Sixteen milliseconds is not much. It is shorter than a blink. Inside that tiny window the main thread must respond to your touch, update the data, calculate the layout, and hand a finished frame to the display.

CHAPTER 2

Big-O Notation — Understanding It Practically

In Chapter 1 you saw two contacts apps that looked identical but behaved completely differently, and the cause was the data structure underneath. That raises an obvious question. Before you build a feature, before you even run it on a device, how can you tell whether your approach will be fast enough?

You need a way to predict the cost of code just by reading it. That tool is called Big-O notation. Many developers first meet it as a scary interview topic full of symbols, but the real idea is simple and genuinely useful every single day. Before we use Big-O, though, we should understand the activity it belongs to. That activity is called complexity analysis.

What is Complexity Analysis?

Complexity analysis is the practice of estimating how much work a piece of code will need as the amount of data it handles grows, without actually running it.

Two ideas inside that sentence matter.

- The first is "without actually running it." You do not need a device, a stopwatch, or a profiler to do complexity analysis. You read the code and reason about it. This is powerful, because it lets you reject a bad approach while it is still a sketch on a whiteboard, before you have spent a day building it.
- The second is "as the amount of data grows." Complexity analysis is never about one fixed input. It is about the trend. Ten contacts, ten thousand contacts, ten million contacts: analysis describes the shape of that climb.

To make this trend clear, complexity analysis deliberately throws away two kinds of detail.

It throws away hardware differences. A flagship phone and a five year old budget phone run the same code at different speeds, but that difference is roughly a constant multiplier. The growth shape, meaning how the cost climbs as data grows, is the same

on both. By ignoring the constant, analysis gives you a prediction that holds across every device your app will ever run on.

It also throws away constants and lower order terms in the count itself. If you carefully count and find that a function does 3n plus 5 steps, complexity analysis simplifies that to O(n). The reasoning is that when n is large, the 3 and the 5 stop mattering. Both 3n plus 5 and plain n climb as a straight line, and the straight line shape is the insight. The exact slope is noise.

So complexity analysis is closer to weather forecasting than to stopwatch timing. It does not tell you "this run took 23 milliseconds." It tells you "as your users add more data, this code will scale like a straight line" or "like a curve that explodes." On a phone, where you cannot control how much data a user piles up over years, that forecast is exactly what you need.

Three Types of Complexity Notation

When you analyze a piece of code, it often does not have a single fixed cost. Think about searching a list for a name. If the name is the very first item, you finish immediately. If it is the last item, or not in the list at all, you check everything. Same code, very different cost, depending on luck.

To talk about this honestly, computer science uses three notations, each answering a different question.

Big-O, written with the **letter O,** describes the worst case. It is an upper bound, and it promises "the code will never be slower than this." Searching a list of n items is O(n), because in the worst case you inspect all n.

Big-Omega, written with the **symbol Ω**, describes the best case. It is a lower bound, and it says "the code will never be faster than this." That same search is Ω(1), because in the luckiest case the name is first and you finish in a single step.

Big-Theta, written with the **symbol Θ,** describes a tight bound. It is used when the best case and the worst case share the same growth shape, so one notation captures the whole story. It is also the natural way to describe the typical or average case.

Here is the same linear search seen through all three lenses.

CHAPTER 3

Arrays and Dynamic Arrays

Of every data structure in this book, the array is the one you already use the most, very often without giving it a second thought. Every list of messages, every grid of photos, every row of settings sits on top of an array somewhere. Because it is so familiar, it is easy to use it without understanding what it actually costs. This chapter fixes that. We will start from what an array really is at the lowest level, see how the resizable version that you use every day actually grows, understand why it is the natural backbone for the list views on both Android and iOS, and finish with the deeper reason arrays are fast, a reason that Big-O alone does not reveal.

What an array really is

At the lowest level, an array is a single block of memory that holds a fixed number of elements, all of the same type, laid end to end with no gaps. Each element is found by an index, and indexing starts at zero.

That simple layout gives the array its single most important property: reading or writing an element by its index is O(1), constant time. It does not matter whether the array holds ten elements or ten million. The reason is pure arithmetic. Because every element is the same size and they sit directly next to each other, the computer can find element number i with one calculation: take the address where the array starts, and add i multiplied by the size of one element. There is no searching, no walking through earlier elements. One multiplication, one addition, done.

This is worth pausing on, because it is the foundation for everything else. When a scrolling list needs "the item at position 4,217," the array hands it over instantly.

A pure array has one strict limitation. Its size is fixed at the moment it is created. If you make an array for one hundred elements, it holds exactly one hundred, forever. You cannot make it bigger later. In real apps you almost never know the exact size in advance, and that is the problem the dynamic array solves.

ArrayList (Kotlin), Array (Swift)

A dynamic array is an array that can grow. It is the everyday list type you reach for constantly. There is a small but useful difference in how the two languages present this.

In Kotlin, the type called Array<T>, along with primitive versions like IntArray, is the fixed size kind. The growable one is ArrayList, which you usually create through mutableListOf(). In Swift, the type simply called Array, written with square brackets like [String], is already the dynamic, growable one. So a Kotlin developer keeps two ideas separate, while a Swift developer uses one type that is dynamic by default.

Here is the everyday usage in both languages.

KOTLIN · ANDROID

```
// Fixed size: a size is given and it cannot grow
val scores = IntArray(3)              // [0, 0, 0]
scores[0] = 90

// Dynamic: grows as needed, backed by ArrayList
val names = mutableListOf<String>()
names.add("Aman")                     // append, amortized O(1)
val first = names[0]                  // index access, O(1)
names.add(0, "Zoya")                  // insert at the start, O(n)
names.removeAt(0)                     // remove from the start, O(n)
```

SWIFT · iOS

```
// Swift's Array is already a dynamic array
var names: [String] = []
names.append("Aman")                  // append, amortized O(1)
let first = names[0]                  // index access, O(1)
names.insert("Zoya", at: 0)           // insert at the start, O(n)
names.remove(at: 0)                   // remove from the start, O(n)
```

Now the important question. If an array has a fixed size, how can a dynamic array grow?

The trick is that a dynamic array quietly keeps two numbers, not one. The first is the size, meaning how many elements you have actually put in. The second is the capacity, meaning how many slots it has allocated room for. The capacity is usually larger than the size, so there is spare space at the end ready for new elements.

When you append and there is still spare capacity, the element drops into the next free slot. That is a true O(1) operation. But when you append and the size has reached the capacity, there is no room left. At that moment the dynamic array does something expensive. It allocates a brand new, larger block of memory, copies every existing element across into it, and then adds your new element. That copy step is O(n).

CHAPTER 4

Linked Lists

Chapter 3 ended on a clear weakness. A dynamic array is excellent at almost everything, but inserting or removing an element anywhere other than the end costs O(n), because every later element has to shift to keep the memory packed. The linked list is the structure built to attack exactly that weakness. It makes insertion and removal cheap, no matter where in the sequence they happen. As you will see, it pays for that ability with two real sacrifices, and understanding the trade is what makes you able to choose well.

What a linked list is

An array keeps its elements in one solid block of memory. A linked list does the opposite. It scatters its elements, and then connects them with references.

The unit of a linked list is the node. A node is a small object that holds two things: the actual value you want to store, and a reference, often called a pointer, to the next node. The list itself is just a starting reference, called the head, that points to the first node. From the first node you follow its reference to the second, from the second to the third, and so on, until you reach a node whose next reference is empty, which marks the end.

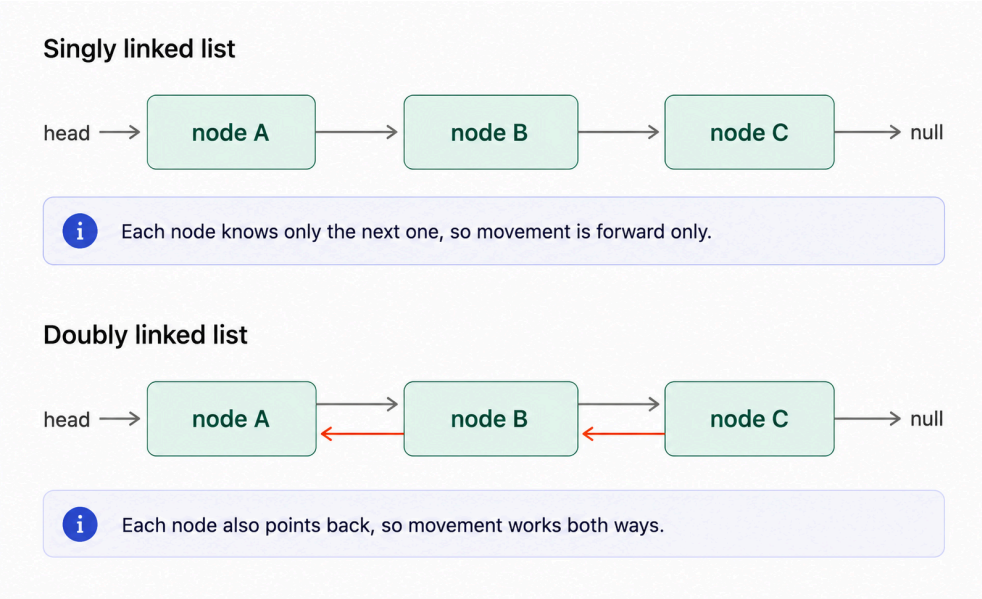
Notice what this means. The nodes do not need to sit next to each other in memory at all. They can be anywhere. The chain is held together purely by the references, not by physical position. That single design choice is the source of every strength and every weakness in this chapter.

Singly vs Doubly linked lists

There are two main forms of linked list, and the difference between them is small to describe but large in consequence.

A singly linked list is the basic form just described. Each node points only to the next node. You can travel through it in one direction, forward, and never backward.

A doubly linked list gives each node a second reference, pointing to the previous node as well as the next. This lets you travel in both directions, and it makes one particular operation, removal, far cheaper, for a reason we will come to shortly.



In code, a node is a tiny class. Here is the singly form and then the doubly form.

KOTLIN · ANDROID

```
// Singly: one pointer, to the next node
class Node<T>(val value: T) {
    var next: Node<T>? = null
}

// Doubly: a second pointer, to the previous node
class DoublyNode<T>(val value: T) {
    var next: DoublyNode<T>? = null
    var prev: DoublyNode<T>? = null
}
```

SWIFT · iOS

```
// Doubly node; drop `prev` for a singly node
final class Node<T> {
    let value: T
    var next: Node<T>?
    var prev: Node<T>?
    init(value: T) { self.value = value }
}
```

A practical note before we go further. Kotlin can use java.util.LinkedList, a ready made doubly linked list from the Java standard library. Swift's standard library has no linked list type at all, so on iOS you build your own when you need one. But here is the honest truth that the rest of this chapter will build toward: you rarely use a bare linked list directly in app code. Its real value is as the engine inside other structures, and the most

CHAPTER 5

Stacks

The structures so far have tried to be flexible. The array lets you reach any position, the linked list lets you splice anywhere. The stack does something that sounds like a step backward. It deliberately forbids most of that freedom. A stack lets you add and remove at one end only, and nowhere else. Yet that single restriction is not a weakness. That is the entire point, because a surprising number of real problems have exactly that shape, and when a problem fits the stack, the stack solves it with almost no effort. Two of those problems, the navigation back button and the undo feature, sit in nearly every app you will ever build.

What a stack is

A stack follows one rule, and the rule has a name: Last In, First Out, usually shortened to LIFO. The last item you put in is the first item you take out. Picture a pile of plates. You add a plate to the top, and when you need one, you take it from the top. You never pull a plate from the middle of the pile. A stack works exactly like that pile.

A stack offers only a tiny set of operations, and that small menu is part of its charm. You push an item, which places it on top. You pop an item, which removes and returns the item from the top. You peek, which looks at the top item without removing it. And you can ask whether the stack is empty. There is no "get the item at position 4" and no "insert into the middle." The stack simply does not allow it.

Because every operation touches only the top, every operation is O(1), constant time. There is no shifting, no walking, no searching. This is the fastest and simplest structure in the book, and its cost table has almost nothing in it.

Operation	Complexity
Push an item onto the top	O(1)
Pop the top item off	O(1)
Peek at the top item	O(1)
Check whether it is empty	O(1)

A stack is not usually a separate type you import. It is a behavior you get from a structure you already know. A dynamic array makes a perfect stack: pushing is appending to the end, popping is removing from the end, and Chapter 3 told you both are O(1). So in practice you use a list as a stack.

KOTLIN · ANDROID

```
val stack = ArrayDeque<String>()
stack.addLast("first")            // push, O(1)
stack.addLast("second")
val top = stack.last()            // peek, O(1)
val removed = stack.removeLast()  // pop, O(1)
val isEmpty = stack.isEmpty()
```

SWIFT · iOS

```
var stack: [String] = []
stack.append("first")             // push, O(1)
stack.append("second")
let top = stack.last              // peek, O(1)
let removed = stack.popLast()     // pop, O(1)
let isEmpty = stack.isEmpty
```

One small note for Kotlin developers. The modern, recommended choice is ArrayDeque. There is also an older java.util.Stack class, but it is a legacy type with extra synchronization overhead, and current guidance is to avoid it. Swift keeps things simple, since its Array already does everything a stack needs.

Navigation back-stack (NavController, UINavigationController)

Think about how a user moves through an app. They start on a home screen, tap into a feed, open someone's profile from the feed, then open settings from the profile. Now they press the back button. Where should they land?

On the profile. Press back again, and they land on the feed. Again, the home screen. The back button always returns to the screen they saw most recently. That is Last In, First Out. The screens form a stack, and it has a name on both platforms: the back stack.

Opening a new screen pushes it onto the back stack and makes it the visible top. Pressing back pops the top screen off, and whatever screen is now on top becomes visible again.

CHAPTER 6

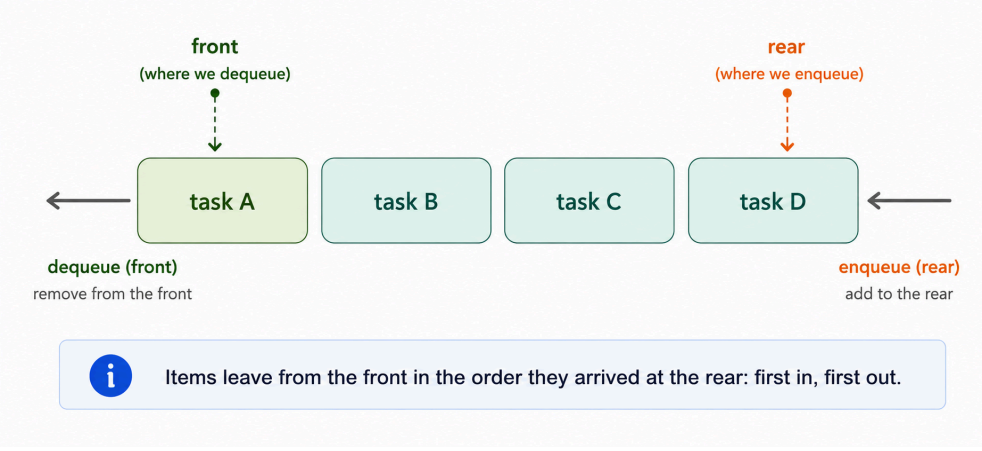
Queues and Deques

The stack from Chapter 5 always reaches for the most recent item. That is perfect for a back button, but it would be a terrible way to run a checkout line, because the person who arrived last would be served first. Many problems need the opposite of a stack. They need fairness, where things are handled in the order they arrived. That structure is the queue, and it turns out to be the quiet engine behind how a mobile app schedules work, how the main thread stays alive, and how an app talks to the network without overwhelming it.

What a queue is

A queue follows one rule, and like the stack's rule, it has a name: First In, First Out, shortened to FIFO. The first item to join is the first item to leave. This is exactly the behavior of a line of people at a counter. You join at the back, you wait your turn, and you are served from the front.

A queue has two ends, and that is the key difference from a stack. You add at one end, called the rear, and you remove from the other end, called the front. Adding an item is called enqueue. Removing the front item is called dequeue. You can also peek at the front item without removing it, and ask whether the queue is empty.



A well built queue does all of its work in O(1). Enqueue, dequeue, and peek should each be constant time.

But here is a trap that catches developers constantly, and it ties straight back to Chapter 3. It is tempting to build a queue from a plain array, enqueueing by appending to the end and dequeueing by removing index zero. Appending is fine, but removing the first element of an array is O(n), because every remaining element must shift one slot to the left to close the gap at the front. A queue built that way is quietly slow, and the slowness grows with the queue's length.

The correct way to build a queue avoids that shifting. A linked list works, with the head as the front and the tail as the rear, giving O(1) at both ends. Better still is a structure designed for exactly this. On Android, Kotlin's ArrayDeque gives true O(1) at both ends, so you use it directly as a queue.

KOTLIN · ANDROID

```
val queue = ArrayDeque<String>()
queue.addLast("first")            // enqueue at the rear, O(1)
queue.addLast("second")
val front = queue.first()         // peek at the front, O(1)
val served = queue.removeFirst()  // dequeue from the front, O(1)
```

Swift's standard library has no queue or deque type, and its Array.removeFirst() is the O(n) trap described above. So on iOS you build a small queue that sidesteps the shifting. A clean trick is to keep a head index that simply moves forward, instead of physically removing the front element each time.

SWIFT · iOS

```
struct Queue<T> {
    private var items: [T] = []
    private var head = 0

    var isEmpty: Bool { head >= items.count }
    var front: T? { isEmpty ? nil : items[head] }

    mutating func enqueue(_ item: T) {        // O(1) amortized
        items.append(item)
    }

    mutating func dequeue() -> T? {           // O(1) amortized
        guard head < items.count else { return nil }
        let item = items[head]
        head += 1
        if head > 64 && head * 2 >= items.count {
            items.removeFirst(head)           // occasional cleanup
            head = 0
        }
        return item
    }
}
```

CHAPTER 7

Hash Tables (HashMap / Dictionary)

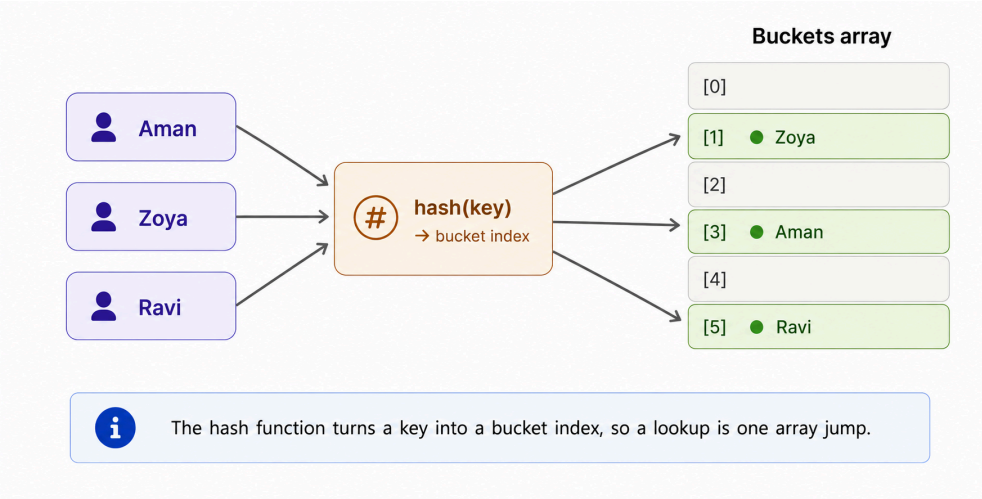
Part 3 of the book leaves the straight line behind. The array, the linked list, the stack, and the queue all arranged their items in a single sequence. Real apps need a different shape of access. They need to ask "give me the value for this key" without caring where in memory it sits. That is what the structures in this part are for, and the first of them, the hash table, has shown up in nearly every chapter so far. The contacts map in Chapter 1, the duplicate check in Chapter 2, the heart of the LRU cache in Chapter 4, the lookup behind every well written list adapter, all of them rely on the same idea. Now we open that idea up and see exactly how it works, and why it costs what it costs.

What a hash table is

A hash table stores pairs of key and value. You hand it a key, and it hands you back the value associated with that key. The famous promise is that it can do so in O(1) time, regardless of how many entries it holds.

That promise sounds impossible at first. If a structure holds a million entries, how can it find the right one without searching? The trick is that it does not search. It computes the address.

Underneath every hash table is an array of slots, called buckets. The clever part is the function that decides which bucket each key belongs in. That function is called a hash function. You hand it a key, it returns an integer, and that integer, after a small mathematical trim to fit the array size, is the bucket index. Putting a value in is "compute the index, store the value at that slot in the array." Reading a value is "compute the index, read the slot." Both rely on a single array index access, and Chapter 3 already proved that array indexing is O(1). The hash table is, at heart, an array with a smart way of choosing positions.



Read that picture as the whole idea in one frame. The keys "Aman", "Zoya", and "Ravi" each pass through the same hash function. The function does not need to know anything special about names. It only needs to scramble each key into an integer that lands in one of the available buckets, ideally in a way that spreads keys evenly. Once the bucket is found, the value is right there. No looping, no searching.

This gives the hash table its famous cost profile, which you have been quietly relying on since Chapter 1.

Operation	Complexity (Average)
Read a value by key	O(1)
Write a value for a key	O(1)
Delete a key	O(1)
Check whether a key exists	O(1)

The word "average" matters and we will come back to it in the final section, because the worst case can be different. For now, hold the picture: a hash table is an array with a smart index calculator on the front.

The most-used structure in mobile apps

It is fair to say the hash table is the single most used data structure in real mobile code, behind even the array in some ways, because every screen seems to need at least one. On Android it is called HashMap, and there is a useful sibling, LinkedHashMap, that

CHAPTER 8

Sets

The previous chapter ended with a question. A hash table answers "what is the value for this key," but very often the question you actually have is simpler: "have I seen this thing before, and what is the collection of distinct things I have seen?" You do not want a value attached. You only want membership and uniqueness. That is precisely what a set provides, and although it is a close cousin of the hash table, the question it answers is so common in mobile development that it earns its own chapter.

What a set is

A set is a collection with two defining properties. It holds no duplicates, and it is built for one fast question above all others: is this item a member, yes or no?

Think of it as the mathematical idea of a set made concrete. Each item is either in or out, and adding an item that is already present changes nothing. There is no concept of position the way an array has indices, and in the most common kind of set there is no guaranteed order either. The set does not care where an item sits or when it arrived. It only knows whether the item belongs.

The relationship to Chapter 7 is direct and worth stating plainly. The most common set is built on exactly the same machinery as a hash table. In fact you can picture a hash set as a hash map where only the keys matter and the values are thrown away. Everything you learned about hashing, buckets, collisions, and load factors carries straight over. That is why a hash set gives you the same headline cost.

Operation	Complexity (average)
Add an item	O(1)
Remove an item	O(1)
Check membership (contains)	O(1)
Get the number of items	O(1)

That third row is the one that makes the set special. Asking "does this set contain x" is O(1). Compare that to asking the same question of an array, where you must scan every element, an O(n) search. Whenever your code checks membership repeatedly, the set is almost always the right structure, and reaching for an array instead is one of the most common quiet performance mistakes in app code.

The basic usage is clean in both languages.

KOTLIN · ANDROID

```
val ids = HashSet<Int>()
ids.add(42)                  // add, O(1)
ids.add(42)                  // no effect, already present
val hasIt = 42 in ids        // contains, O(1)
ids.remove(42)               // remove, O(1)
val count = ids.size
```

SWIFT · iOS

```
var ids: Set<Int> = []
ids.insert(42)               // add, O(1)
ids.insert(42)               // no effect, already present
let hasIt = ids.contains(42) // contains, O(1)
ids.remove(42)               // remove, O(1)
let count = ids.count
```

A set also gives you the classic set operations from mathematics, and these are genuinely useful in real code, not just textbooks. Union combines two sets, intersection keeps only the items in both, and difference keeps the items in one but not the other.

KOTLIN · ANDROID

```
val a = setOf(1, 2, 3)
val b = setOf(2, 3, 4)
val union = a union b            // {1, 2, 3, 4}
val common = a intersect b       // {2, 3}
val onlyInA = a subtract b       // {1}
```

SWIFT · iOS

```
let a: Set = [1, 2, 3]
let b: Set = [2, 3, 4]
let union = a.union(b)               // {1, 2, 3, 4}
let common = a.intersection(b)       // {2, 3}
let onlyInA = a.subtracting(b)       // {1}
```

CHAPTER 9

Trees, Basics

Every structure so far has been flat. The array, the linked list, the stack, the queue, the hash table, and the set all hold a collection of items with no item containing another. But a great deal of the data a mobile app touches is not flat at all. It is nested. A screen is made of layouts that contain other layouts that contain buttons and labels. A folder contains files and other folders. A network response contains objects that contain arrays that contain more objects. Whenever items contain other items, the natural structure is the tree, and this chapter introduces it through three things you already work with every day.

What a tree is

A tree is a structure built from nodes connected by parent and child relationships. It looks, drawn on paper, like a family tree turned upside down. There is one node at the very top called the root, the single ancestor of everything. Each node can have children below it, and every node except the root has exactly one parent above it. A node with no children is called a leaf. The number of levels from the root down to the deepest leaf is the tree's height, or depth.

The defining rule, the one that makes a tree a tree, is that there are no cycles. You can never follow child links and end up back where you started. Every node has exactly one path from the root to reach it. This is what separates a tree from the graph you will meet in Chapter 11, where connections can loop back freely.

A few terms will keep recurring, so it is worth fixing them now. The root is the top node. A parent is a node directly above another. A child is a node directly below another. Siblings share the same parent. A leaf is a childless node at the bottom. A subtree is any node together with everything beneath it, which is a useful idea because it means every node is itself the root of its own little tree. That last point is why trees and recursion fit together so naturally, as you will see.

In its most general form, a tree node simply holds a value and a list of children.

KOTLIN · ANDROID

```
class TreeNode<T>(val value: T) {
```

```
    val children = mutableListOf<TreeNode<T>>()
}
```

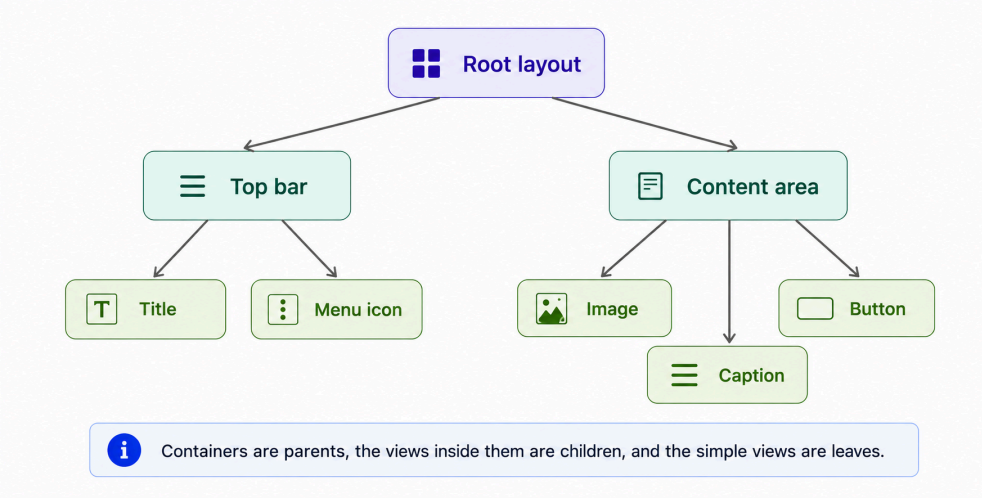
SWIFT · iOS

```
final class TreeNode<T> {
    let value: T
    var children: [TreeNode<T>] = []
    init(value: T) { self.value = value }
}
```

The view hierarchy is a tree

The most important tree for a mobile developer is the one you build every time you lay out a screen, often without thinking of it as a tree at all. The user interface is a tree of views.

A screen has a root container at the top. Inside it sit other containers, and inside those sit the actual buttons, text, and images the user sees. A container is a parent, the views inside it are its children, and a button with nothing inside it is a leaf. This is true on both platforms by design. On Android the nodes are View and ViewGroup objects, or composables nested inside other composables in Jetpack Compose. On iOS the nodes are UIView objects, each with a subviews array and a superview reference, or View values nested in SwiftUI.



Seeing the UI as a tree is not just a mental nicety. It explains real performance behavior that affects every screen you build. When the system draws a frame, it walks this tree. It

CHAPTER 10

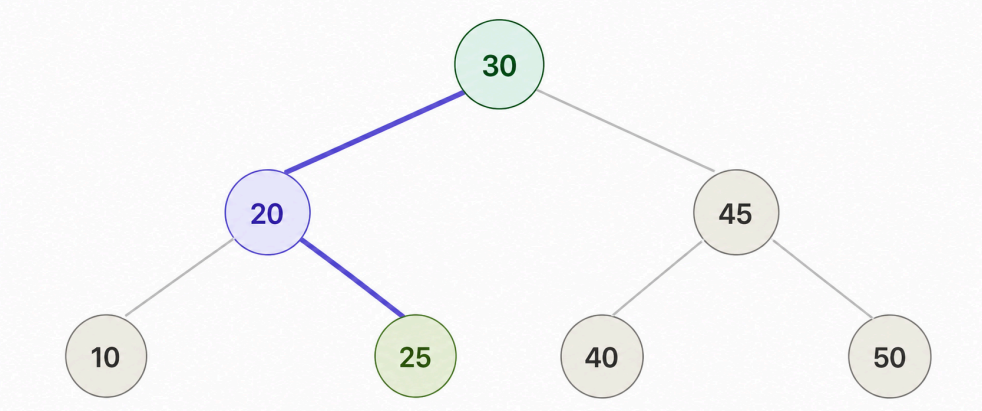
Trees — Practical

Chapter 9 gave you the tree as a shape and the traversals as ways to walk it. This chapter turns that shape into three tools you will actually reach for. The binary search tree keeps data ordered so you can find things quickly. The trie powers the search and autocomplete that users now expect everywhere. And the parsed form of a JSON response is itself a tree, which means the traversal skills you already have are exactly what you need to read network data. Each section connects a tree to a concrete job in a real app.

Binary Search Tree

Chapter 9 introduced the binary tree, where each node has at most a left and a right child. A binary search tree, usually shortened to BST, adds one simple rule on top, and that single rule is what makes it useful. For every node, everything in its left subtree is smaller than the node, and everything in its right subtree is larger.

That ordering rule turns searching into a guided descent. To find a value, you start at the root and compare. If your target is smaller, you go left. If it is larger, you go right. Every comparison throws away an entire half of the remaining tree, exactly like looking up a word in a physical dictionary by repeatedly opening toward the correct half.



Finding 23: at 30 go left, at 20 go right, landing near 25. Each step skips half the tree.

Here is the search and insertion in code, both following the same compare-and-descend logic.

KOTLIN · ANDROID

```
class BST {
    private class Node(val value: Int) {
        var left: Node? = null
        var right: Node? = null
    }
    private var root: Node? = null

    fun contains(target: Int): Boolean {
        var current = root
        while (current != null) {
            current = when {
           target == current.value -> return true
          target < current.value  -> current.left   // go left
          else                    -> current.right  // go right
            }
        }
        return false
    }

    fun insert(value: Int) {
        if (root == null) { root = Node(value); return }
        var current = root!!
        while (true) {
            if (value < current.value) {
                current.left?.let { current = it } ?: run { current.left =
Node(value); return }
            } else {
                current.right?.let { current = it } ?: run { current.right =
Node(value); return }
            }
        }
    }
}
```

SWIFT · iOS

```
final class BST {
    private final class Node {
        let value: Int
        var left: Node?
        var right: Node?
        init(_ value: Int) { self.value = value }
    }
    private var root: Node?

    func contains(_ target: Int) -> Bool {
        var current = root
        while let node = current {
            if target == node.value { return true }
            current = target < node.value ? node.left : node.right
        }
        return false
```

CHAPTER 11

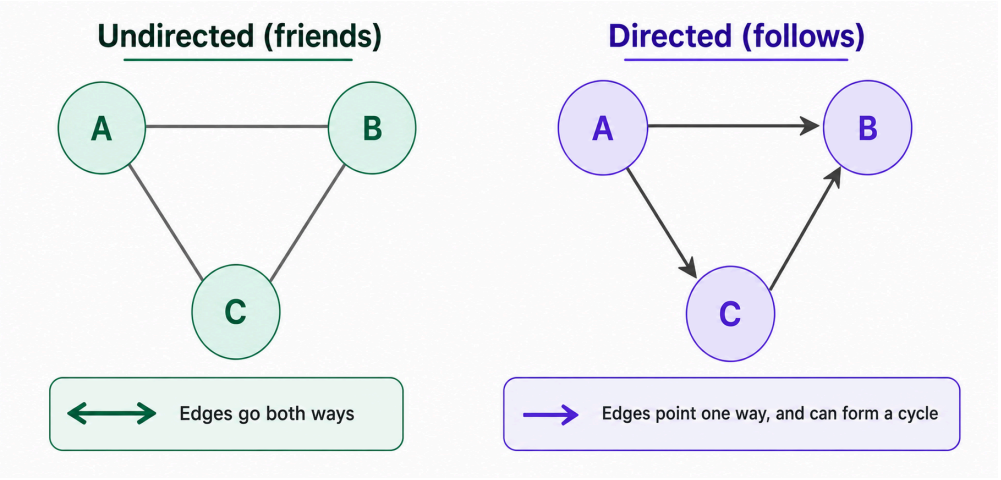
Graphs, a Lightweight Introduction

Chapter 10 ended by promising to relax the one rule that defined a tree. A tree forbids cycles: you can never follow child links and arrive back where you started, and every node has exactly one parent. The graph removes both restrictions. In a graph, anything can connect to anything, connections can loop back, and a node can be reached from many directions. This freedom makes the graph the most general structure in the book, able to model relationships that a tree cannot. The goal of this chapter is not to make you a graph theory expert, because most mobile apps need only the basics. The goal is to help you recognize when your data is secretly a graph, and to give you the two traversal techniques that handle almost every practical case.

What a graph is

A graph is a set of nodes, usually called vertices, connected by links, usually called edges. That is the entire definition, and its looseness is the point. A tree is actually just a special, restricted kind of graph, one with no cycles and a single root. Lift those restrictions and you have a graph.

Two distinctions cover almost everything you will meet. The first is direction. In an undirected graph, an edge between two nodes goes both ways. Friendship on many social networks is undirected: if you are my friend, I am your friend. In a directed graph, an edge points one way. Following on other social networks is directed: you can follow someone who does not follow you back. The second distinction is cycles. Because a graph can contain loops, a path can return to a node it already visited, and this single fact is what forces the one new habit this chapter will teach.



Before traversing a graph you have to store it, and the way you store it shapes everything. There are two standard representations, and for a mobile developer one of them is almost always the right choice.

- The first is the adjacency matrix, a grid where the cell at row i and column j records whether there is an edge from node i to node j. It is simple but it uses memory proportional to the number of nodes squared, written $O(V^2)$, whether or not the edges actually exist. For most real graphs, which are sparse, meaning each node connects to only a handful of others, that is a great deal of wasted memory, and on a phone wasted memory is exactly the cost Chapter 1 told you to avoid.
- The second is the adjacency list, where each node simply stores a list of its direct neighbors. This is the representation you should reach for by default. It uses memory proportional to the nodes plus the edges that genuinely exist, O(V + E), which is far leaner for the sparse graphs that real apps deal with. In practice an adjacency list is most naturally built from a structure you already know well: a hash map from each node to the list of its neighbors.

KOTLIN · ANDROID

```
class Graph {
    // each node maps to the list of nodes it connects to
    private val adjacency = HashMap<Int, MutableList<Int>>()

    fun addEdge(from: Int, to: Int, bidirectional: Boolean = true) {
        adjacency.getOrPut(from) { mutableListOf() }.add(to)
        if (bidirectional) {
            adjacency.getOrPut(to) { mutableListOf() }.add(from)
        }
    }

    fun neighbors(node: Int): List<Int> = adjacency[node] ?: emptyList()
```

CHAPTER 12

Caching Strategies

Part 3 finished the catalogue of structures. Part 4 changes the question from "what is this structure" to "how do real apps combine these structures to stay fast and light." We begin with caching, because it is the single most common place where the data structures of this book come together to solve a problem that every app faces. You already built the core of a cache in Chapter 4, when you combined a hash map and a doubly linked list into an LRU cache. This chapter widens that one idea into the full strategy that production apps use: deciding what to keep, where to keep it, and when to let it go.

Why caching exists

A cache is a store of results you have already computed or fetched, kept close at hand so you do not have to do the expensive work again. The expensive work on a phone is usually one of three things: a network request that costs time, data, and battery; a disk read that is far slower than memory; or a computation like decoding an image that burns processor cycles. Caching trades a resource you have a little of, memory or storage space, for a resource that is precious, time and battery.

The reason caching is not trivial is the constraint from Chapter 1. Memory is borrowed and small. You cannot keep everything, so every cache must answer one hard question: when it is full and something new arrives, what do you throw away? That single question, called the eviction policy, is the heart of caching, and the data structures of this book are what make a good answer fast.

LRU and other eviction policies

Chapter 4 introduced the most popular eviction policy, Least Recently Used. When the cache is full, you discard the item that has gone the longest without being touched, betting that what you have not used in a while is what you are least likely to need next. You built it from a hash map for O(1) lookup and a doubly linked list to track usage order, so that every operation, including finding and evicting the least recently used item, runs in O(1). That combination is worth remembering as the canonical answer, because it is the one you will reach for most.

LRU is not the only policy, and knowing the alternatives helps you choose well. A few are worth recognizing.

The simplest is FIFO, First In First Out, which evicts the oldest inserted item regardless of how often it has been used. It is just the queue from Chapter 6 used as a cache. It is easy but naive, because it will happily evict an item you use constantly simply because it arrived early.

A smarter policy is LFU, Least Frequently Used, which evicts the item that has been accessed the fewest times. This suits data with stable popularity, where some items are perennial favorites, but it needs extra bookkeeping to count accesses and can cling to items that were popular once long ago.

Many real systems use TTL, Time To Live, where each entry carries an expiry time and is considered invalid once that time passes, independent of memory pressure. This is less about space and more about freshness, and it is essential for data that goes stale, like a feed or a price.

For a mobile developer, LRU remains the sensible default for memory caches because it adapts to actual usage cheaply, but TTL frequently rides alongside it to handle staleness. You will often see both: LRU decides what to drop when memory is tight, and TTL decides what is too old to trust regardless.

Memory cache vs disk cache

The most important practical idea in this chapter is that a real cache is not one cache. It is layered. There are two main layers, and they have opposite strengths, which is exactly why apps use both together.

The memory cache, sometimes called the in memory or L1 cache, lives in RAM. It is extremely fast to read, because it is the same memory your running code uses, but it is small and it is volatile, meaning everything in it vanishes the moment your app process is killed, which Chapter 1 told you can happen at any time. The structure behind it is the LRU cache you already built.

The disk cache, sometimes called the L2 cache, lives in the device's storage. It is much slower than memory, because reading from storage is far costlier than reading from RAM, but it is large and it is persistent, surviving app restarts and even reboots. Its keys map to files on disk rather than objects in memory.

CHAPTER 13

Efficient List Rendering

Lists are the backbone of mobile interfaces, and Chapter 3 showed how an array feeds a recycling list view so that even a huge collection scrolls smoothly. But real lists are not static. Items get added, removed, reordered, liked, edited, and marked as read, and every one of those changes has to reach the screen. This chapter is about doing that well. The naive approach, redrawing the whole list on every change, quietly destroys the performance you worked to build. The professional approach uses a diffing algorithm to change only what actually changed, and pagination to load only what the user can actually see. Both are the data structures of this book applied to the most visible surface in your app.

The problem with redrawing everything

Start with the mistake, because understanding it is half the lesson. When the data behind a list changes, the simplest thing a developer can do is tell the list "everything is different, redraw it all." On Android this is the infamous notifyDataSetChanged(). On iOS it is a blanket reloadData().

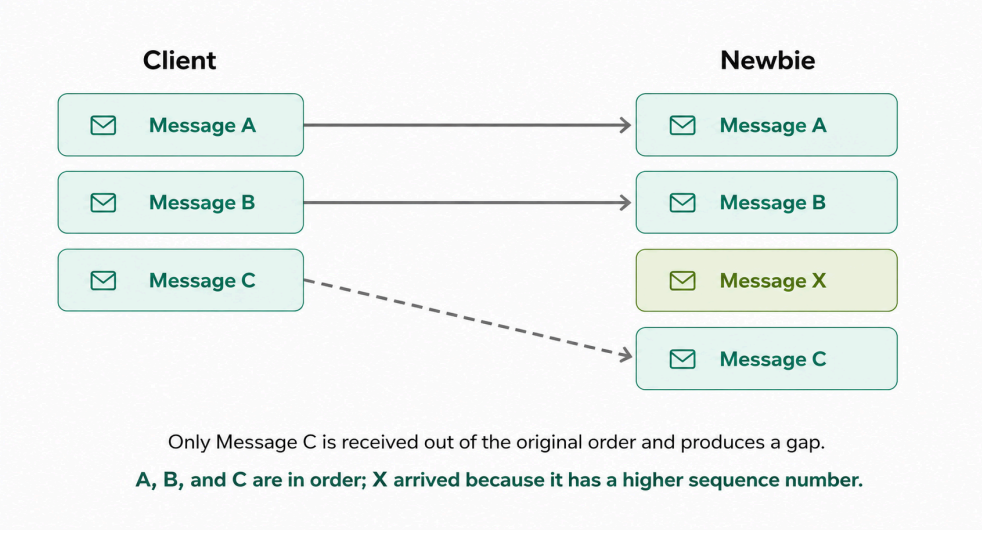
This is expensive for reasons that come straight from earlier chapters. Redrawing everything discards the recycled views and rebinds them, throws away the carefully cached scroll position, and interrupts any in flight image loads from Chapter 12. Worse, it does all of this work inside the sixteen millisecond frame budget from Chapter 2, so a large or frequent redraw drops frames and produces visible jank. And it loses the small, delightful animations, the gentle slide as one new message appears, because the system has no idea that only one item changed. It was told everything changed, so it can animate nothing meaningfully.

The core insight is that a data change is almost never "everything is different." When a new chat message arrives, one row is inserted and the rest are identical. When you like a post, one row's content changes and its position does not. The screen should reflect exactly that: one insertion, or one update, and nothing else. To make the screen do that, the app must first figure out precisely what changed between the old list and the new list. That calculation is called diffing.

DiffUtil and diffing algorithms

Diffing is the process of comparing two versions of a list, the old one and the new one, and computing the minimal set of changes, insertions, removals, and moves, that turns the first into the second. Once you have that minimal set, you can tell the list view exactly what happened, and it updates only the affected rows, preserves scroll position, keeps images loading, and animates each change naturally.

The idea is easiest to see with a small example. Suppose the list held three items and now holds four, with one new item in the middle.



Android provides this as a built in tool called DiffUtil, and the modern ListAdapter wraps it so the diffing runs on a background thread and dispatches only the precise changes to the list. To use it you answer two questions about your items, and this distinction is the single most important thing to understand about diffing.

The first question is "are these the same item?" This asks about identity, and it should compare a stable identifier, such as a message ID or a user ID, never the whole object. The second question is "does this same item have the same contents?" This asks whether an item that is identity-equal has nonetheless changed in a way that needs redrawing, for example a post that kept its ID but gained a new like count. Separating identity from contents is what lets the algorithm tell the difference between "this row moved" and "this row's data changed," so it can animate a move differently from an in-place update.

CHAPTER 14

Choosing the Right Structure — Decision Guide

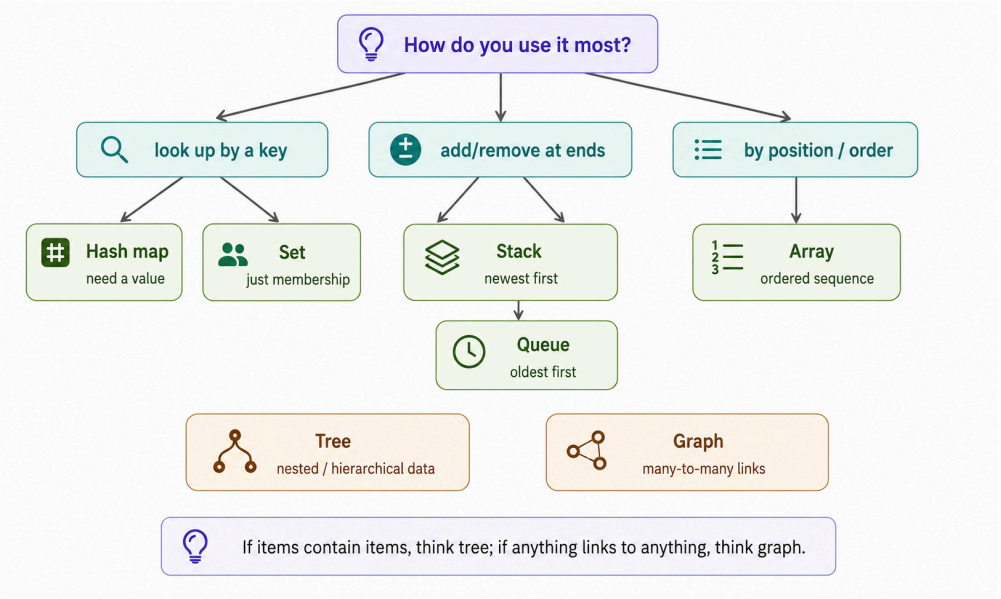
Every chapter so far taught one structure or one application. This chapter does something different. It steps back and turns all of that knowledge into a decision you can make quickly, at the moment you are actually writing code. Choosing a data structure is not about memorizing definitions. It is about asking the right question of your problem and letting the answer point you to the structure that fits. The first half of this chapter gives you that questioning process and a cheat sheet to consult. The second half catalogues the specific mistakes that quietly slow real apps down, so you can recognize them in your own code and in code review. Think of this as the chapter you return to long after the first read.

The one question that drives every choice

Before any cheat sheet, there is a single question that resolves most decisions on its own: what operation will this code do most often? Not what the data is, but what you will do to it, over and over, especially inside anything that runs during a scroll or an animation. A structure is fast at some operations and slow at others, so the operation you repeat most is the one that must be cheap. Everything else is negotiable.

That question splits into a few practical follow ups. Do you look items up by a key, or do you walk through them in order? Does the collection change shape often, with inserts and removals, or is it mostly stable once built? Do you need the items in a particular order, or not at all? And where does this run, on the main thread inside the frame budget, or safely in the background? Answer those, and the structure usually chooses itself.

Here is that reasoning drawn as a flow. Follow it from the top for the collection you are designing.



Read the flow as a conversation with your own code. The top asks how you use the collection most. If you mostly fetch items by some identifier, you are in hash territory, and the only follow up is whether you need a value attached, which means a hash map, or only need to know presence, which means a set. If you mostly add and remove at the ends, you want a stack when the newest item should come out first and a queue when the oldest should. If you mostly reach items by position or need them kept in a sequence, an array is your answer. And underneath those everyday cases sit the two shapes for connected data: a tree when items contain other items, and a graph when anything can link to anything. This single picture captures nine tenths of the choices you will make.

Cheat sheet, situation to structure

The flow gives you the reasoning. This table gives you the fast lookup. Each row is a situation phrased the way it actually appears in real work, followed by the structure that fits and the one line reason, with the chapter where you can revisit it.

When you need to…	Reach for	Because
Find an item by an ID or key, repeatedly	Hash Map (Ch 7)	O(1) lookup instead of scanning
Check "have I seen this / is this selected"	Set (Ch 8)	O(1) membership, no duplicates

CHAPTER 15

Searching and Sorting You Actually Use

Part 4 was about applying structures to real screens. Part 5 shifts to the algorithm techniques that operate on those structures, and it opens with the two most fundamental operations in all of programming: finding a thing, and putting things in order. There is an enormous body of theory about searching and sorting, enough to fill entire books, but a working mobile developer needs only a focused slice of it. This chapter gives you that slice honestly. It shows you the one search technique worth mastering, binary search, and then it makes an unusual argument for a data structures book: that you should almost never write a sorting algorithm yourself. Understanding why is as valuable as any algorithm.

Two ways to search, and when each applies

You have already met the everyday search without naming it as a technique. When you loop through a list checking each item until you find the one you want, that is linear search. It works on any list, in any order, and it is O(n), because in the worst case you inspect every element. For small lists, and for lists you only search occasionally, linear search is perfectly fine and you should not overthink it.

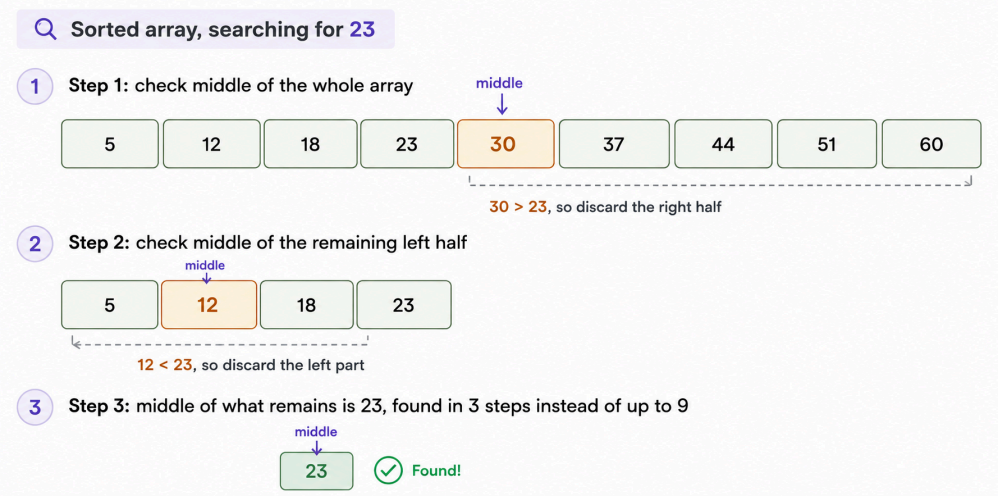
But Chapter 2 taught that O(n) inside a hot path is dangerous, and Chapter 7 gave one escape: if you look items up by a key repeatedly, put them in a hash map and get O(1) lookups. That covers the case where you search by an exact identifier. There is a second, different case that the hash map does not serve, and that is where binary search comes in: when your data is sorted, and you want to find an item, or find where an item would go, faster than scanning.

The distinction is worth stating plainly, because choosing among these three is a real skill. A hash map answers "give me the item with exactly this key" in O(1), but it has no order, so it cannot answer "what is the smallest item larger than this" or "give me everything between these two values." Binary search answers those ordered questions in O(log n), but it requires the data to be sorted first. Linear search answers anything on any list but pays O(n). Match the technique to the question: exact key lookups go to the hash map, ordered or range questions on sorted data go to binary search, and everything else falls back to linear search.

Binary search

Binary search is the technique for finding an item in a sorted collection by repeatedly halving the search space. It is the same idea as looking up a word in a physical dictionary: you do not start at page one and read every word, you open near the middle, see whether your word comes before or after, and throw away half the book in one glance. Then you repeat on the half that remains.

The mechanics are precise. You track a low and a high boundary, initially the whole list. You look at the middle element. If it is your target, you are done. If your target is smaller, the answer must be in the lower half, so you move the high boundary just below the middle. If your target is larger, the answer is in the upper half, so you move the low boundary just above the middle. Each comparison discards half of what remains, and you continue until you find the item or the boundaries cross, meaning it is not present.



The power of this halving is the O(log n) cost, and the numbers are genuinely striking. In a list of one thousand items, linear search may take up to a thousand comparisons, while binary search takes at most about ten. In a list of one million items, linear search may take a million comparisons, while binary search takes about twenty. Each doubling of the data adds only a single extra step. That is what logarithmic growth means, and it is why binary search scales so gracefully.

Here is a correct implementation in both languages. Correctness matters here more than usual, because binary search is famously easy to get subtly wrong at the boundaries.

CHAPTER 16

Recursion and Backtracking

Chapter 15 was honest about which techniques you actually reach for, and this chapter continues in that spirit with two that sit at very different ends of the practical scale. Recursion is a tool you already use constantly, often without labelling it, every time you walk a tree of views or a nested response. Backtracking is a tool you will meet mostly in interviews and puzzle style problems, and only rarely in production app code. Treating them together is deliberate, because backtracking is really just recursion with one extra move, and seeing that connection is what makes the harder of the two feel manageable. This chapter grounds recursion in the daily work you know and then extends it to backtracking honestly, telling you where it genuinely helps and where it does not.

What recursion really is

Recursion is simply a function that solves a problem by calling itself on a smaller piece of the same problem. It has appeared throughout this book, in the tree traversals of Chapter 9, the folder size calculation of Chapter 5, the trie gathering of Chapter 10, and the graph exploration of Chapter 11. Now we look at it directly as a technique.

Every correct recursion has two parts, and getting both right is the whole skill. The first is the base case, the simplest version of the problem that can be answered immediately without recursing further. The second is the recursive case, which breaks the problem into a smaller piece and calls the function on that piece, trusting it to solve the smaller version. The base case is what stops the recursion; without it, or with a wrong one, the function calls itself forever. And "forever" on a phone has a specific, harsh meaning that Chapter 5 introduced: each call consumes a frame on the fixed size call stack, and enough calls overflow it and crash the app.

The reason recursion feels natural for so much mobile data is that the data itself is recursive in shape. A view contains views that contain views. A folder contains folders. A JSON object contains objects. A comment has replies that have replies. Whenever a structure is defined in terms of smaller copies of itself, a function shaped the same way, calling itself on the smaller copies, fits the problem like a glove. The code ends up mirroring the structure of the data, which is why recursive solutions to tree problems are usually shorter and clearer than the loop based equivalents.

Recursion in view tree traversal

The most tangible everyday recursion for a mobile developer is walking the view hierarchy from Chapter 9. The screen is a tree of views, containers holding other containers holding leaf views, and many real tasks require visiting every view in that tree: finding a view with a particular tag, counting how many buttons a screen contains, applying a theme to every text view, or measuring the depth of nesting to diagnose a layout that is too deep.

Each of these is a recursion with the same shape. The base case is a leaf view with no children, which you handle directly. The recursive case is a container, where you handle the container itself and then call the same function on each of its children. Here is a function that counts every view in a hierarchy, written on both platforms.

KOTLIN · ANDROID

```
fun countViews(view: View): Int {
    if (view !is ViewGroup) return 1      // base case: a leaf view is one
view
    var total = 1                         // count this container
    for (i in 0 until view.childCount) {
        total += countViews(view.getChildAt(i))  // recurse into each child
    }
    return total
}
```

SWIFT · iOS

```
func countViews(_ view: UIView) -> Int {
    if view.subviews.isEmpty { return 1 }     // base case: a leaf view is
one view
    var total = 1                             // count this container
    for child in view.subviews {
        total += countViews(child)            // recurse into each child
    }
    return total
}
```

Notice how closely the code mirrors the tree. The function handles one node, then delegates the rest of the work to itself on each child, trusting each call to return the count of its own subtree. This is the mental move that makes recursion click: you do not trace every level in your head, you handle the current node correctly and trust the recursion for the rest. That trust is justified precisely because the base case guarantees the process terminates.

CHAPTER 17

Greedy Thinking

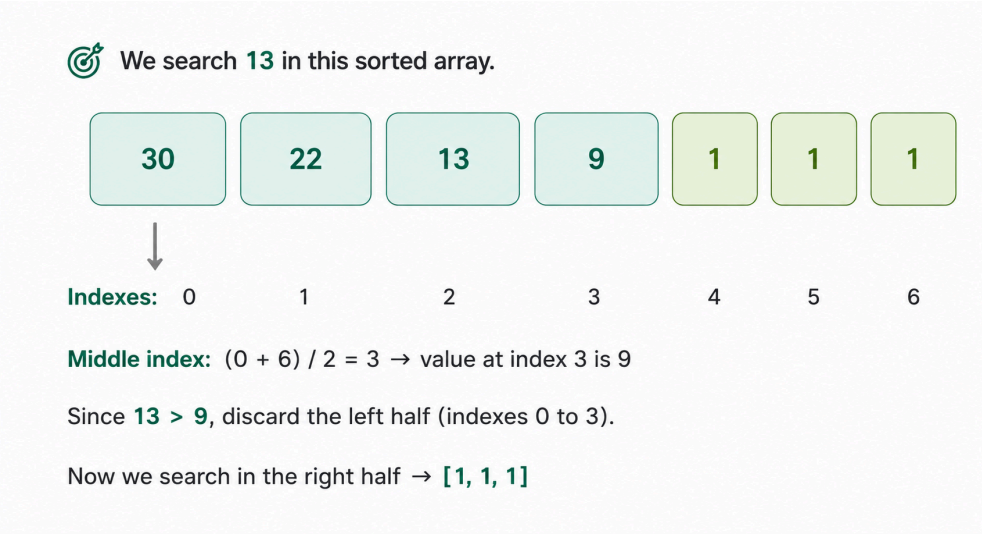
Chapter 16 ended with backtracking, a technique that explores every possibility because it cannot know in advance which choice is right. This chapter is about the opposite instinct: sometimes you can just take the best option in front of you, right now, and never look back. That instinct is called the greedy approach, and it is the simplest and fastest of all the algorithmic techniques when it works. The catch, and the real lesson of this chapter, is knowing when it works, because greedy thinking is powerful when the problem suits it and quietly wrong when it does not. Like the two chapters before it, this one is honest about where greedy thinking earns its place in mobile development, which is more in how you reason about problems than in large amounts of code.

What greedy thinking is

A greedy algorithm builds a solution one step at a time, and at each step it makes the choice that looks best in that moment, without reconsidering earlier choices and without looking ahead to future consequences. It is greedy in the everyday sense: it grabs the most attractive option immediately and commits to it.

Contrast this with the two techniques you just learned. Backtracking tries a choice, explores where it leads, and undoes it if it fails, examining many possibilities. A greedy algorithm never undoes anything and never explores alternatives; it makes each local decision once and moves on. This makes it dramatically faster. Where backtracking can be exponential, a greedy algorithm is usually just a single pass, O(n) or O(n log n) if it needs the data sorted first. It trades the thoroughness of exploration for the speed of decisiveness.

The classic illustration is making change with coins. Suppose you must give ninety three units of change using standard coin values, and you want to use as few coins as possible. The greedy approach is to repeatedly take the largest coin that still fits: take the biggest coin not exceeding what remains, subtract it, and repeat. For standard currency systems, this simple grab the largest strategy actually produces the fewest coins every time.



Here is that greedy coin selection in code, taking the largest coin that fits at each step.

KOTLIN · ANDROID

```
fun makeChange(amount: Int, coins: List<Int>): List<Int> {
    val sorted = coins.sortedDescending()     // largest first
    val result = mutableListOf<Int>()
    var remaining = amount
    for (coin in sorted) {
        while (remaining >= coin) {           // take this coin as long as it
fits
            result.add(coin)
            remaining -= coin
        }
    }
    return result
}
```

SWIFT · iOS

```
func makeChange(amount: Int, coins: [Int]) -> [Int] {
    let sorted = coins.sorted(by: >)          // largest first
    var result: [Int] = []
    var remaining = amount
    for coin in sorted {
        while remaining >= coin {             // take this coin as long as it
fits
            result.append(coin)
            remaining -= coin
        }
    }
    return result
}
```

CHAPTER 18

Heaps & Priority Queues

The structures so far answer questions like "give me this exact item" or "give me the next item in line." The heap answers a different and very common question: **"give me the most important item right now."** Not the newest, not the oldest, but the largest or the smallest by some measure. Interviews at Google, Amazon, and Meta ask heap questions constantly, and real apps quietly need them too: the highest priority task, the top ten trending posts, the closest driver. This chapter builds the heap from zero, in simple steps.

What a heap is

A heap is a special binary tree with two simple rules:

- **Shape rule:** it is a complete tree. Every level is full, except possibly the last, which fills from left to right. No gaps in between.
- **Order rule:** in a **min heap**, every parent is smaller than or equal to its children, so the smallest item is always at the root. In a **max heap**, every parent is larger, so the largest sits at the root.

Notice what the order rule does not say. It does not say the whole tree is sorted. Siblings can be in any order. The heap makes only one promise, and it makes it very well: the root is always the minimum (or maximum). That single promise is the entire product.

The array trick

Here is the beautiful part. Because the shape rule forbids gaps, a heap never needs node objects and pointers. It lives inside a plain array, and simple index math replaces all the links:

- The item at index **i** has its left child at **2i + 1** and its right child at **2i + 2**.
- The parent of index **i** sits at **(i – 1) / 2**, using whole number division.

So a heap is really just an array from Chapter 3 wearing a clever pair of glasses. It gets the array's cache friendliness and compact memory for free, which is exactly what Chapter 3 said a phone loves.

The two moves: bubble up and sink down

Every heap operation is built from two small repair moves:

- **Insert (bubble up):** append the new item at the end of the array, then compare it with its parent and swap while it is smaller (in a min heap). It climbs until the order rule holds again. At most it climbs the height of the tree, so the cost is **O(log n)**.
- **Extract (sink down):** to remove the root, move the last array item into the root's place, then compare it with its children and swap with the smaller child while it is larger. It sinks until the rule holds. Also **O(log n)**.
- **Peek:** just read index 0. **O(1)**.

Operation	Cost	Why
Peek min or max	O(1)	It is always at index 0
Insert	O(log n)	Bubble up at most the tree height
Extract min or max	O(log n)	Sink down at most the tree height
Build a heap from n items	O(n)	Heapify is cheaper than n inserts

Using heaps in Kotlin and Swift

On Android, the JDK gives you a ready made heap called **PriorityQueue**. By default it is a min heap; pass a comparator to flip it or to order custom objects.

KOTLIN · ANDROID

```
import java.util.PriorityQueue

val minHeap = PriorityQueue<Int>()            // min heap by default
minHeap.add(5); minHeap.add(1); minHeap.add(3)
val smallest = minHeap.peek()                  // 1, O(1)
val removed  = minHeap.poll()                  // 1, O(log n)

// Max heap: reverse the comparison
val maxHeap = PriorityQueue<Int>(compareByDescending { it })

// Priority by a field: most urgent task first
data class Task(val name: String, val priority: Int)
val tasks = PriorityQueue<Task>(compareByDescending { it.priority })
```

CHAPTER 19

Dynamic Programming — A Pragmatic Introduction

Chapter 16 was honest that backtracking mostly lives in interviews. Dynamic programming, usually shortened to **DP**, gets the same honest treatment: it appears rarely in day to day app code, but Google and Amazon interviews ask it regularly at the easy to medium level, and a senior candidate who freezes on "climbing stairs" loses the room. The good news is that DP is not a new subject. It is **recursion from Chapter 16, plus one idea: never solve the same subproblem twice.** This chapter builds that idea step by step, in plain language.

The problem DP solves: wasted repetition

Start with the smallest famous example. A staircase has n steps, and you can climb 1 or 2 steps at a time. How many distinct ways can you reach the top? Think recursively: to stand on step n you arrived either from step n−1 or from step n−2, so **ways(n) = ways(n–1) + ways(n–2)**, with ways(1)=1 and ways(2)=2. Clean recursion. Now watch it run for n=6: ways(6) calls ways(5) and ways(4); ways(5) calls ways(4) again; ways(4) gets computed twice, ways(3) three times, and the repetition explodes as n grows. The plain recursion costs **$O(2^n)$**, the worst row of Chapter 2's table, purely from re-answering questions it already answered.

That is the entire disease DP cures. The subproblems **overlap**, and DP simply remembers answers instead of recomputing them.

Memoization: recursion plus a notebook

The first DP technique is called **memoization**, and it is exactly the caching instinct from Chapter 12 applied to a function:

- Keep a map (or array) of answers already computed: the **memo**.
- At the start of the function, if the answer for this input is in the memo, return it instantly.
- Otherwise compute it recursively, store it in the memo, then return it.

Each distinct subproblem is now solved once, so the cost collapses from $O(2^n)$ to **O(n)** time with O(n) space for the memo. Same recursion, plus a notebook.

KOTLIN · ANDROID

```
fun climbStairs(n: Int, memo: HashMap<Int, Long> = HashMap()): Long {
    if (n <= 2) return n.toLong()
    memo[n]?.let { return it }              // already solved: reuse
    val ways = climbStairs(n - 1, memo) + climbStairs(n - 2, memo)
    memo[n] = ways                          // remember before returning
    return ways
}
```

SWIFT · iOS

```
func climbStairs(_ n: Int, _ memo: inout [Int: Int]) -> Int {
    if n <= 2 { return n }
    if let cached = memo[n] { return cached }    // already solved: reuse
    let ways = climbStairs(n - 1, &memo) + climbStairs(n - 2, &memo)
    memo[n] = ways                               // remember before returning
    return ways
}
```

Tabulation: filling the table bottom-up

The second technique, **tabulation**, drops the recursion entirely. Instead of starting from the big problem and working down, you start from the smallest cases and build up, filling an array where each entry depends only on entries already filled. For the stairs: dp[1]=1, dp[2]=2, then loop dp[i] = dp[i−1] + dp[i−2] up to n. Same O(n), no recursion, no call stack risk — which, after Chapter 16's warnings, you know matters on a phone. Most DP solutions can be written either way; tabulation is usually the safer final form, and often you can shrink the table to just the last one or two values, cutting space to **O(1)**.

The four-step DP method

Every DP interview problem yields to the same visible method. Say the steps out loud:

- **Step 1 — define the state:** in one sentence, what does dp[i] mean? ("dp[i] = number of ways to reach step i.") If you cannot say this sentence, stop and find it first.
- **Step 2 — write the transition:** how does dp[i] come from smaller states? ("dp[i] = dp[i−1] + dp[i−2].")
- **Step 3 — set the base cases:** the smallest states you can answer directly.

CHAPTER 20

Graphs — Advanced Patterns

Chapter 11 gave you the graph, BFS, DFS, and the visited set, and honestly said that covers most practical needs. Senior interviews go three steps further, and each step is genuinely useful to understand as a mobile engineer, because each one already runs inside tools you use daily. This chapter adds **topological sort** (the algorithm inside your build system), **Dijkstra** (the algorithm inside every map app), and **Union-Find** (the fastest way to answer "are these two things connected"). All three reuse pieces you already own.

Topological sort: ordering things with dependencies

A **dependency** means "X must happen before Y." Your Gradle modules, your Swift package graph, your app's startup tasks (init logging before analytics, analytics before network) all form a **directed graph of dependencies**, and the question is always the same: give me a valid order to process everything. That order is called a topological sort, and it exists only if the graph has **no cycles** — a cycle like A needs B, B needs A is an unsolvable deadlock, which is exactly why your build tool errors on circular dependencies.

The classic method, **Kahn's algorithm**, is beautifully simple and reuses Chapter 6's queue:

- Count every node's **indegree**: how many arrows point into it (how many things it waits for).
- Put every node with indegree 0 into a queue — these wait for nothing and can go first.
- Repeatedly dequeue a node, add it to the output order, and "complete" it: decrease the indegree of every node it points to. Any node that drops to 0 joins the queue.
- If the output contains all nodes, that is a valid order. If some nodes never reached indegree 0, a **cycle exists** — and you have detected it for free.

KOTLIN · ANDROID

```
fun topologicalOrder(nodes: Int, edges: List<Pair<Int, Int>>): List<Int>? {
    val adjacency = HashMap<Int, MutableList<Int>>()
```

```
    val indegree = IntArray(nodes)
    for ((from, to) in edges) {
        adjacency.getOrPut(from) { mutableListOf() }.add(to)
        indegree[to]++
    }
    val queue = ArrayDeque<Int>()
    for (n in 0 until nodes) if (indegree[n] == 0) queue.addLast(n)

    val order = mutableListOf<Int>()
    while (queue.isNotEmpty()) {
        val n = queue.removeFirst()
        order.add(n)
        for (next in adjacency[n] ?: emptyList()) {
            if (--indegree[next] == 0) queue.addLast(next)
        }
    }
    return if (order.size == nodes) order else null   // null means a cycle
}
```

SWIFT · iOS

```
func topologicalOrder(_ nodes: Int, _ edges: [(Int, Int)]) -> [Int]? {
    var adjacency: [Int: [Int]] = [:]
    var indegree = [Int](repeating: 0, count: nodes)
    for (from, to) in edges {
        adjacency[from, default: []].append(to)
        indegree[to] += 1
    }
    var queue = indegree.indices.filter { indegree[$0] == 0 }
    var head = 0
    var order: [Int] = []
    while head < queue.count {
        let n = queue[head]; head += 1
        order.append(n)
        for next in adjacency[n] ?? [] {
            indegree[next] -= 1
            if indegree[next] == 0 { queue.append(next) }
        }
    }
    return order.count == nodes ? order : nil     // nil means a cycle
}
```

Cost: **O(V + E)**, one pass over nodes and edges. Interview costumes for this pattern: "course schedule" (can you finish courses with prerequisites), "build order," "task scheduling with dependencies," "alien dictionary." The moment a problem says *before/after/prerequisite*, think topological sort.

Dijkstra: shortest path when edges have weights

Chapter 11's BFS finds the shortest path when every hop counts the same. But real edges have **weights**: road segments have minutes, network routes have latency, delivery

CHAPTER 21

Strings — Patterns & Internals

Strings deserve their own chapter for two reasons. First, string questions are a huge share of every interview loop — arrays with characters inside, plus a few twists of their own. Second, strings on mobile have real internals that senior engineers are expected to know: why concatenation in a loop is a performance bug, why Swift strings behave unlike any other language's, and why an emoji can break your character counting. This chapter covers both: the patterns that pass interviews, and the internals that mark you as senior.

The patterns, applied to text

The good news: you already learned the main string patterns in Chapter 18's question bank — they are the array patterns wearing text. A quick map:

- **Hash map counting** → anagrams, first non-repeating character, character frequency.
- **Two pointers** → palindrome checks, reversing in place, comparing from both ends.
- **Sliding window + set** → longest substring without repeating characters, smallest window containing all required characters.
- **Stack** → balanced brackets, undoing backspaces, decoding nested strings.

Two string-specific classics deserve full treatment because they appear constantly.

Group anagrams

Given many words, group the ones that are anagrams of each other ("listen", "silent", "enlist" together). The insight: anagrams become **identical when sorted**, so the sorted form is a perfect group key for a hash map from key to list of words. Cost: O(n · k log k) for n words of length k. In an interview, offer the upgrade: a count-of-each-letter key avoids the sort, giving O(n · k).

KOTLIN · ANDROID

```
fun groupAnagrams(words: List<String>): List<List<String>> {
    val groups = HashMap<String, MutableList<String>>()
    for (w in words) {
```

```
        val key = String(w.toCharArray().sortedArray())   // sorted form as
key
        groups.getOrPut(key) { mutableListOf() }.add(w)
    }
    return groups.values.toList()
}
```

SWIFT · iOS

```
func groupAnagrams(_ words: [String]) -> [[String]] {
    var groups: [String: [String]] = [:]
    for w in words {
        let key = String(w.sorted())                      // sorted form as
key
        groups[key, default: []].append(w)
    }
    return Array(groups.values)
}
```

Longest palindromic substring: expand around center

Find the longest stretch of a string that reads the same both ways. The brute force checks every substring, O(n^3). The elegant answer: every palindrome has a **center** — either a character (odd length) or a gap between two characters (even length). There are only 2n − 1 centers, so try each one, expanding outward with two pointers while the ends match, and remember the longest. Cost: **O(n^2)** time, O(1) space, and it is short enough to write flawlessly under pressure.

KOTLIN · ANDROID

```
fun longestPalindrome(s: String): String {
    if (s.length < 2) return s
    var bestStart = 0
    var bestLen = 1

    fun expand(l: Int, r: Int) {
        var left = l; var right = r
        while (left >= 0 && right < s.length && s[left] == s[right]) {
            left--; right++
        }
        val len = right - left - 1
        if (len > bestLen) { bestLen = len; bestStart = left + 1 }
    }

    for (center in s.indices) {
        expand(center, center)        // odd length, single-char center
        expand(center, center + 1)    // even length, gap center
    }
    return s.substring(bestStart, bestStart + bestLen)
}
```

CHAPTER 22

Bit Manipulation & Matrix Problems

This chapter pairs two topics that finish the interview toolkit. Bit manipulation looks intimidating and is actually a small set of tricks — and as an Android or iOS developer, you have been using it all along without the name, because platform flags are literally bits. Matrix problems are 2D arrays with a handful of movement patterns, and one of them is secretly the image processing your apps already do. Both are asked at every top company, both are learnable in an afternoon once someone explains them plainly.

Bits: the five operations

A number is a row of bits. Five operations manipulate them, and everything else is combinations:

- **AND (&)** — 1 only where both are 1. Used to *test* whether a bit is set: `flags & FLAG != 0`.
- **OR (|)** — 1 where either is 1. Used to *set* a bit: `flags = flags | FLAG`.
- **XOR (^)** — 1 where the two differ. The magic one: a value XOR itself is 0, and XOR with 0 leaves a value unchanged.
- **NOT / complement (~ in Kotlin: inv())** — flips every bit. With AND, it *clears* a bit: `flags and FLAG.inv()`.
- **Shifts (shl/shr in Kotlin, << and >> in Swift)** — slide bits left or right; shifting left by one doubles a number, right by one halves it.

You already use bitmasks daily

Open any Android codebase and the flags are there: combining Intent flags with `or`, view visibility constants, input type masks, `PendingIntent` flags. Each named flag is one bit position, and combining them with OR packs many booleans into a single Int — which is exactly why the pattern exists: **one integer carries 32 answers**, checked in a single AND. iOS does the same with `OptionSet` types like `UIView.AutoresizingMask` — an OptionSet is a Swift-friendly wrapper over a bitmask. When an interviewer asks "how would you store 30 on/off settings compactly," this is the answer, and you can add that your platform has been doing it to you all along.

KOTLIN · ANDROID

```
// Reading and writing flags — the pattern behind Intent flags
const val WIFI      = 1 shl 0    // 0001
const val BLUETOOTH = 1 shl 1    // 0010
const val LOCATION  = 1 shl 2    // 0100

var enabled = 0
enabled = enabled or WIFI or LOCATION       // set two flags
val hasWifi = enabled and WIFI != 0         // test a flag → true
enabled = enabled and LOCATION.inv()        // clear a flag
```

SWIFT · iOS

```
struct Features: OptionSet {
    let rawValue: Int
    static let wifi      = Features(rawValue: 1 << 0)
    static let bluetooth = Features(rawValue: 1 << 1)
    static let location  = Features(rawValue: 1 << 2)
}

var enabled: Features = [.wifi, .location]  // set two flags
let hasWifi = enabled.contains(.wifi)       // test a flag → true
enabled.remove(.location)                   // clear a flag
```

The three interview classics

- **Single number:** every value in a list appears twice except one — find it, with O(1) extra space. XOR everything together: the pairs cancel to 0, and the loner remains. One line, O(n), and it feels like a magic trick the first time.
- **Power of two check:** a power of two has exactly one bit set, so `n > 0 && n & (n – 1) == 0`. The expression `n & (n – 1)` clears the lowest set bit — a trick worth memorizing because it also powers the next classic.
- **Counting set bits:** loop `n = n & (n – 1)`, counting iterations; each pass removes one set bit, so the loop runs only as many times as there are 1s.

KOTLIN · ANDROID

```
fun singleNumber(nums: IntArray): Int {
    var result = 0
    for (n in nums) result = result xor n     // pairs cancel out
    return result
}

fun isPowerOfTwo(n: Int) = n > 0 && (n and (n - 1)) == 0
```

SWIFT · iOS

CHAPTER 23

Concurrency & Thread-Safe Structures

Every structure in this book so far carried a silent assumption: **one thread at a time**. Chapter 6 showed why heavy work leaves the main thread, and Chapter 12's caches and loaders run across threads — which means the assumption is already false in real apps. The moment two threads touch the same structure, a new class of bug appears: rare, unreproducible, and devastating. Senior interviews probe this hard, because it separates engineers who have shipped concurrent code from those who have only read about it. This chapter explains the danger from zero, then gives the tools, in order of preference.

The race condition: why plain structures break

Take the most innocent line imaginable: `counter++`. It looks like one step. It is actually three: **read** the current value, **add** one, **write** it back. Now run it on two threads at once:

- Thread A reads 5. Thread B reads 5.
- Both add one, getting 6.
- Both write 6. Two increments happened; the counter moved by one. **An update was silently lost.**

This is a **race condition**: the result depends on the accidental timing of threads. Collections suffer worse fates. A HashMap being resized (Chapter 7's rehash) while another thread reads it can return garbage or corrupt the bucket links; an ArrayList appended by two threads mid-resize can lose elements or crash. And the cruelest property: races are timing-dependent, so the code passes every test on your machine and fails on one user's device in another country, with a stack trace pointing nowhere useful. This is why thread safety is designed in, never debugged in.

Tool 1: confinement — the best lock is no lock

The safest concurrent structure is one that **only one thread ever touches**. Before reaching for any synchronization, ask whether you can confine the structure:

- Keep UI state on the main thread only — which is exactly what both platforms already push you toward, and why "read on main, mutate on main" makes so much code safely lock-free.
- Route all mutations through a **single serial queue or dispatcher**: many threads may *request* changes, but one lane *performs* them, in order. This is Chapter 6's serial DispatchQueue and a single-threaded coroutine dispatcher used as a safety tool.
- Swift's modern answer builds this in: an **actor** is a type whose state is confined by the compiler — all access is serialized automatically, and forgetting `await` is a compile error rather than a 2 a.m. crash.

SWIFT · iOS

```
actor ScoreBoard {
    private var scores: [String: Int] = [:]   // confined: only the actor
touches it

    func add(_ points: Int, for player: String) {
        scores[player, default: 0] += points  // no lock needed — serialized
by the actor
    }

    func top() -> (String, Int)? {
        scores.max { $0.value < $1.value }.map { ($0.key, $0.value) }
    }
}
// Usage: await board.add(10, for: "aman") — the await is the safety
```

Tool 2: atomics — for a single number

When the shared state is just a counter or a flag, a full lock is overkill. **Atomic types** perform read-modify-write as one indivisible hardware operation, so the lost-update race becomes impossible:

KOTLIN · ANDROID

```
import java.util.concurrent.atomic.AtomicInteger

val downloads = AtomicInteger(0)
downloads.incrementAndGet()        // read+add+write as ONE indivisible step
val current = downloads.get()
```

On iOS, the same role is played by an actor holding the number, or by a tiny lock (`OSAllocatedUnfairLock`) around it. The principle to state in an interview: match the

CHAPTER 24

Interview Questions, Mobile-Dev Focused

Everything in this book was chosen because it matters for building real apps, and there is a happy side effect of that: the same knowledge is exactly what mobile interviews test. But knowledge alone does not crack interviews. Candidates with strong knowledge fail every day because they prepared the wrong way, panicked under pressure, or could not show their thinking out loud. So this chapter comes in two halves. The first half is the strategy: what mobile interviews actually look like, what to focus on, how to practice, and how to behave in the room, the complete journey from deciding to prepare to accepting the offer. The second half is the practice: forty real questions of the kind mobile interviews actually ask, each answered with the same visible approach, so that by the end the method is a habit.

Part One: The Strategy

What a mobile interview actually looks like

Before preparing, know the battlefield. A typical mobile developer interview process, at product companies and startups alike, has a familiar shape, and knowing it removes half the fear.

It usually starts with a screening round, often an online test or a short call, with one or two easy to medium data structure problems. Then come one or two DSA rounds, live coding sessions where an interviewer gives you a problem and watches you solve it while thinking aloud; this is where this book's material is tested directly. Next is a platform round focused on Android or iOS itself: lifecycle, concurrency, memory, architecture, and the internals of lists and rendering, where Chapters 1, 3, 6, 9, 12, and 13 quietly reappear as platform questions. Many companies add a machine coding round, where you build a small feature or mini app in a couple of hours, and here your choice of data structures inside real code is silently graded. Senior roles add a system design round, often mobile flavored, designing a feed, a chat client, an offline sync layer, which is Part 4 of this book expressed as architecture. Finally there is a behavioral round about your projects and how you work.

Notice something important in that list. Data structures are not one round. They are the visible subject of the DSA rounds, the hidden subject of the machine coding round, the vocabulary of the system design round, and the explanation underneath half the platform questions. That is why preparing them properly pays off across the entire process, not just in one hour of it.

What to focus on, honestly

The internet will tell you to solve five hundred problems covering every algorithm ever invented. For mobile roles, that is bad advice, because mobile interviews have a distinct and narrower center of gravity, and focusing there beats spreading thin.

The truthful frequency ranking for mobile interviews looks like this. Arrays, strings, and hashing questions dominate; they open almost every process and appear at every level. Stacks and queues come next, both as puzzles and as platform questions about the main thread and navigation. Linked lists appear regularly, especially the pointer manipulation classics. Trees appear regularly too, mostly traversals and depth questions, rarely anything exotic. Binary search and complexity analysis are constant companions rather than separate topics. Graphs appear occasionally, usually just BFS and DFS. Heavy dynamic programming, advanced graph algorithms, and obscure structures are rare for mobile roles outside the largest companies, and even there they are a minority.

Topic	Weight in Mobile Interviews	Book Chapters
Arrays, Strings, Hashing	Very High	Ch 3, 7, 8
Stacks and Queues	High	Ch 5, 6
Trees and Traversals	High	Ch 9, 10
Linked Lists	Medium–High	Ch 4
Binary Search, Sorting, Big-O	Constant (woven throughout)	Ch 2, 15
Recursion and Backtracking	Medium (Recursion emphasized more)	Ch 16
Graphs (BFS/DFS only)	Low–Medium	Ch 11
System Design (Cache, Feed, Autocomplete)	High (Mid & Senior Levels)	Ch 4, 10, 12, 13

Difficulty matters as much as the topic. The overwhelming majority of mobile interview problems are easy to medium. Interviewers for app roles are not hunting for