

COURSE

Data Structures in Practice

10 Chapters

Philip Osztromok

Generated with Claude

DATA STRUCTURES IN PRACTICE

Contents

Chapter 1 Choosing a Structure 2

Chapter 2 Inside the List 7

Chapter 3 Tuples, Sets, Dicts 13

Chapter 4 Stacks, Queues 21

Chapter 5 Linked Lists 29

Chapter 6 Heaps 37

Chapter 7 Trees 46

Chapter 8 Graphs 54

Chapter 9 Hash Tables 62

Chapter 10 Capstone 73

Data Structures in Practice

Chapter 1 · Choosing a Structure

Two programs can give exactly the same answers and still differ by a factor of ten thousand in speed. The usual cause is not clever code but the **data structure** underneath: the way the values are arranged in memory and the operations that arrangement makes cheap or expensive. This course is practical. Each chapter takes one family of structures, shows what it costs, and gives you a rule for when to reach for it. All examples are Python 3.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> The comparison counts in this chapter come from running the code shown and are exact. Timings depend on the machine and Python version, so only the pattern (which is slower, and how the gap grows) is claimed, not the exact seconds.

Cost Is a Property of the Operation

A structure is never simply "fast" or "slow". It is fast at some operations and slow at others. The honest question is always: *what will my code do most often?* Append at the end? Look up by key? Check "is this already here?" Remove from the front? The answer picks the structure.

OPERATION	LIST	SET	DICT	DEQUE
x in c	scans every item	one hash lookup	one hash lookup (keys)	scans every item
add at the end	cheap (amortised)	cheap	cheap	cheap
add at the front	moves every item	no front	no front	cheap
item by position	direct	no positions	no positions	slow in the middle
keeps order	yes	no	yes (insertion order)	yes

Counting Work: Linear Versus Binary Search

Instead of timing, count. The function below walks a list from the start and counts how many times it compares an item with the target. That count is the work, and it does not depend on the

machine.

```
def linear_count(items, target):
    comparisons = 0
    for x in items:
        comparisons += 1
        if x == target:
            return True, comparisons
    return False, comparisons
```

If the list is **sorted**, you can do far better. Look at the middle item. If it is too small, the target can only be in the upper half; if too large, in the lower half. Each comparison throws away half of what is left.

```
def binary_count(items, target):
    lo, hi, comparisons = 0, len(items) - 1, 0
    while lo <= hi:
        mid = (lo + hi) // 2
        comparisons += 1
        if items[mid] == target:
            return True, comparisons
        if items[mid] < target:
            lo = mid + 1
        else:
            hi = mid - 1
    return False, comparisons

for n in (10, 1000, 1000000):
    data = list(range(n))
    print(n, linear_count(data, -1), binary_count(data, -1))
```

Output, for a target that is not present (the worst case):

```
10 (False, 10) (False, 3)
1000 (False, 1000) (False, 9)
```

```
1000000 (False, 1000000) (False, 19)
```

A thousand times more data costs linear search a thousand times more work, but binary search only a few more steps. That growth pattern is what "Big-O" describes: linear search is **O(n)**, binary search is **O(log n)**. You do not need the maths yet, only the habit of asking how the cost grows when the data does.

> **THE PRICE OF BINARY SEARCH**
>
> Binary search only works on sorted data. Sorting costs work too, so it pays off when you search many times, not once.

The Hash Shortcut: set and dict

A `set` and a `dict` use a **hash table**: the value is turned into a number that points almost straight at where it is stored, so the lookup does not scan. No sorting is required either. The rule of thumb that follows is one of the most useful in everyday Python: **if you ask "is x in here?" more than a few times, use a set or dict, not a list.**

```
import timeit

def measure(n):
    items = list(range(n))
    as_set = set(items)
    return (timeit.timeit(lambda: -1 in items, number=100),
            timeit.timeit(lambda: -1 in as_set, number=100))

for n in (1_000, 100_000):
    t_list, t_set = measure(n)
    print(n, "list is", round(t_list / t_set), "times slower")
```

On the author's machine the gap was roughly 100 times at a thousand items and roughly ten thousand times at a hundred thousand items. Your numbers will differ; what will not differ is that the gap widens as the data grows.

The Front of a Queue: list Versus deque

Removing from the front of a `list` forces every other item to shuffle along one place, so it costs work proportional to the length. `collections.deque` (pronounced "deck", a double-ended queue) is built for both ends. For first-in-first-out queues, use it.

```
from collections import deque

q = deque()
for i in range(3):
    q.append(i)          # join at the back
print(q.popleft(), list(q))   # leave from the front: 0 [1, 2]
```

A DECISION HABIT

Before you write a loop that searches or removes, say which operation is most frequent and look it up in the table above. Choosing the structure first usually removes the need to optimise later.

Challenges

EXERCISE 1

Run `linear_count` and `binary_count` on `list(range(n))` for n = 100 and n = 100000, searching for a target that is present at the very end (n - 1). Predict the counts first, then compare them with the results.

View solution

EXERCISE 2

Write `dedupe(items)` that removes duplicates but keeps the first occurrence of each value in its original order. Make it fast for long inputs, and say which two structures you used and why.

View solution

EXERCISE 3

A program keeps a queue of waiting jobs held in a `list` and removes the oldest with `pop(0)`. Rewrite it with `deque`, then time both for 100,000 jobs. State what you expect before you run it.

View solution

QUICK REFERENCE

- **Count work** rather than relying on raw timings
- **Linear search:** O(n); **binary search:** O(log n), needs sorted data
- **Membership test:** use set or dict, not list
- **Queue:** use deque, not list.pop(0)
- **Ask first:** which operation will my code do most often?

COMING NEXT

Data Structures in Practice 2 looks inside the list: how Python stores it, why appending is cheap on average, and what slicing and copying really cost.

Data Structures in Practice

Chapter 2 · Inside the List

The `list` is the structure you use most, so it pays to know what it is. A Python list is a **dynamic array**: one block of memory holding a row of references, one per item, side by side. Being a single block explains almost every cost in the table from Data Structures in Practice 1.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Everything below was run. Exact sizes in bytes and the exact points where a list grows can differ between Python versions, so the chapter claims only the pattern. Timing ratios depend on the machine; only the direction and the widening gap are claimed.

Why Position Lookup Is Instant

Because the references sit in a row, item number `i` is found by arithmetic: start of the block plus `i` steps. No scanning, whatever the length. That is why `items[500000]` costs the same as `items[0]`. The same layout explains the weak spot: to insert at the front, every reference must shift one place to make room, so the work grows with the length of the list.

OPERATION	COST	WHY
items[i]	constant	arithmetic on the block's start
append(x)	constant, amortised	usually there is spare room
pop()	constant	just shrink the length
insert(0, x)	grows with length	every item shifts right
pop(0)	grows with length	every item shifts left
x in items	grows with length	scans from the start

Over-Allocation: Spare Room at the End

If the block were exactly the right size, every `append` would need a bigger block and a copy of everything. Python avoids that by **over-allocating**: when the block is full it grabs noticeably more space than needed. We can watch this with `sys.getsizeof`, which reports the list's memory size.

```
import sys

a = []
last = None
changes = []
for i in range(40):
    size = sys.getsizeof(a)
    if size != last:
        changes.append((len(a), size))
        last = size
    a.append(i)
print(changes)
```

One run printed:

```
[(0, 56), (1, 88), (5, 120), (9, 184), (17, 248), (25, 312), (33, 376)]
```

Read it as: the size changed only seven times while forty items went in. Between those points, appends found spare room and cost almost nothing. (Your byte counts may differ on another Python version, but you will see the same stepped shape.) In one test, 1000 appends caused only 29 size changes.

Amortised Cost

A growth step is expensive: allocate, copy every reference, free the old block. But because the new block is larger by an amount proportional to the current size, such steps become rarer as the list grows. Spread the occasional big cost over the many cheap appends and the average is a small constant. This is called **amortised O(1)**.

AMORTISED IS NOT "EVERY TIME"

A single append can occasionally be slow, because it triggers a growth step. If you need every operation to be fast and predictable, amortised analysis is not a promise about any one call, only about the average over many.

The Cost of Making Room at the Front

Compare two ways to build the same list in reverse order: insert each new item at the front, or append and reverse once at the end.

```
def insert_front(n):
    a = []
    for i in range(n):
        a.insert(0, i)
    return a

def append_rev(n):
    a = []
    for i in range(n):
        a.append(i)
    a.reverse()
    return a

print(insert_front(5), append_rev(5))
print(insert_front(2000) == append_rev(2000))
```

```
[4, 3, 2, 1, 0] [4, 3, 2, 1, 0]
True
```

Same result. On the author's machine, `insert_front` was about 50 times slower at 5,000 items and about 200 times slower at 20,000. The gap grew with the size, the signature of work that depends on the length.

Slicing Copies

A slice such as `items[1:]` builds a **new list** and copies the references in it. It is not a view. So a slice that keeps most of a long list costs work proportional to its length, while a slice of a handful of items is cheap whatever the original size.

```
import timeit

for n in (1000, 100000):
    d = list(range(n))
    big = timeit.timeit(lambda: d[1:], number=200)
    small = timeit.timeit(lambda: d[-5:], number=200)
    print(n, "big slice is", round(big / small), "times slower")
```

The author saw roughly 16 times at 1,000 items and over 5,000 times at 100,000. A recursive function that does `f(items[1:])` on every call therefore copies the list again and again; pass an index instead.

Copies Are Shallow

Copying a list copies the references, not the objects they point to. For a list of numbers that is harmless. For a list of lists it hides a trap: the copy and the original share the inner lists.

```
import copy

x = [1, 2, 3]
y = x            # same list, two names
z = x[:]         # a new list
y.append(4)
print(x, z, y is x, z is x)

m = [[0, 0], [0, 0]]
c = m[:]                 # shallow copy
c[0][0] = 9
print(m, c)
```

```
d = copy.deepcopy(m)      # copies the inner lists too
d[1][1] = 7
print(m, d)
```

```
[1, 2, 3, 4] [1, 2, 3] True False
[[9, 0], [0, 0]] [[9, 0], [0, 0]]
[[9, 0], [0, 0]] [[9, 0], [0, 7]]
```

THE GRID TRAP

`[[0] * 2] * 2` does not make two rows; it makes one row referenced twice. Setting `r[0][0] = 1` gives `[[1, 0], [1, 0]]`. Build grids with a comprehension, `[[0] * 2 for _ in range(2)]`, which creates a fresh row each time.

A DECISION HABIT

Appending and popping at the *end* of a list is what lists are built for. The moment you work at the front or in the middle, or test membership, ask whether the table points to another structure.

Challenges

EXERCISE 1

Write `growth_points(n)` that appends `n` items to an empty list and returns the list lengths at which `sys.getsizeof` changed. Run it for n = 100 and say whether the gaps between growth points get wider or narrower.

View solution

EXERCISE 2

This function sums a list recursively: `def total(items): return 0 if not items else items[0] + total(items[1:])`. Say what it costs in copying, then rewrite it with an index parameter so no slices are made. Check both give the same answer on `list(range(500))`.

View solution

EXERCISE 3

Make a 3 by 3 grid of zeros two ways: with `[[0] * 3] * 3` and with a comprehension. Set the middle cell of each to 5 and print both. Explain the difference, then make an independent copy of the good grid and show that changing the copy leaves the original alone.

View solution

QUICK REFERENCE

- **A list is a dynamic array:** one block of references, so indexing is instant
- **Over-allocation** gives amortised O(1) append and pop at the end
- **Front and middle:** insert and pop shift items, so cost grows with length
- **Slices copy;** avoid big slices in loops and recursion
- **Copies are shallow:** use `copy.deepcopy` for nested data
- **Grids:** use a comprehension, never `*` on a list of lists

COMING NEXT

Data Structures in Practice 3 covers tuples, sets and dicts: what immutability buys you, the rules for hashing, set algebra, and common dict patterns.

Data Structures in Practice

Chapter 3 · Tuples, Sets, Dicts

Chapter 2 looked inside the list. This chapter covers three neighbours that solve problems a list handles badly: the **tuple** (a list that cannot change), the **set** (membership and uniqueness) and the **dict** (lookup by key). The last two rest on one idea, hashing, so the rules for what may be hashed come first.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every snippet below was run and the output shown is what it printed. No timings are claimed in this chapter. Where the order of a set is involved, the text says so: set order is not something to rely on.

Tuples: Fixed Shape

A tuple is an ordered row of references like a list, but it cannot be changed after creation: no append, no item assignment. That gives you a promise to the reader (this group of values is a single fixed record) and a practical benefit: a tuple of hashable items can itself be hashed.

```
p = (3, 4)
try:
    p[0] = 9
except TypeError as e:
    print("TypeError:", e)

t = ([1, 2], 5)
t[0].append(3)
print(t)
```

```
TypeError: 'tuple' object does not support item assignment
([1, 2, 3], 5)
```

SHALLOW IMMUTABILITY

A tuple fixes which objects it holds, not what those objects contain. A list inside a tuple can still change, exactly like the shallow copies of Data Structures in Practice 2. A tuple holding a list is also not hashable, as the next section shows.

The Hashing Rule

Sets and dict keys are stored by **hash**: a whole number computed from the value, used to jump straight to the right spot instead of scanning (the idea behind the constant-cost lookups in Data Structures in Practice 1). For that to work, a value's hash must never change while it is stored, so Python only allows **hashable** values, which in practice means immutable ones. Equal values must also have equal hashes.

```
def key_ok(k):
    try:
        hash(k)
        return True
    except TypeError:
        return False

print(key_ok((1, 2)), key_ok([1, 2]), key_ok((1, [2])))
print(key_ok(frozenset({1})), key_ok({1}))

grid = {(0, 0): "start", (2, 3): "goal"}
print(grid[(2, 3)])
```

```
True False False
True False
goal
```

So: numbers, strings, `None`, tuples of hashables and `frozenset` (an immutable set) may be keys; lists, sets and dicts may not. The tuple-as-key trick is the everyday payoff: coordinates, pairs and composite identifiers make natural keys.

Equal Values Collide

Because equal values must hash alike, `1`, `1.0` and `True` are the same key:

```
print(hash(1) == hash(1.0), 1 == 1.0)
print({1: "a", 1.0: "b", True: "c"})
```

```
True True
{1: 'c'}
```

One entry survives, keeping the first key but the last value. For your own classes, defining `__eq__` without `__hash__` makes instances unhashable (Python refuses to guess); define both, based on the same fields, to use them in sets:

```
class Q:
    def __init__(self, x):
        self.x = x
    def __eq__(self, other):
        return isinstance(other, Q) and self.x == other.x
    def __hash__(self):
        return hash(self.x)

print(len({Q(1), Q(1)}))
```

```
1
```

Sets: Membership and Algebra

A set stores each value once and answers `x in s` by hashing, not scanning. It also supports the operations from school maths, which replace many hand-written loops:

QUESTION	OPERATOR	METHOD
In either?	**a | b**	union
In both?	**a & b**	intersection
In a but not b?	**a - b**	difference
In exactly one?	**a ^ b**	symmetric_difference
Is a inside b?	**a <= b**	issubset
Nothing shared?	(none)	isdisjoint

```
a = {1, 2, 3, 4}
b = {3, 4, 5}
print(a | b, a & b, a - b, a ^ b)
print({1, 2} <= a, a.isdisjoint({9}))
```

```
{1, 2, 3, 4, 5} {3, 4} {1, 2} {1, 2, 5}
True True
```

DO NOT RELY ON SET ORDER

Small whole numbers happen to print in order here, but a set makes no ordering promise. Use `sorted(s)` whenever order matters. Also, `{}` is an empty *dict*; an empty set is `set()`.

Removing Duplicates

`set(items)` removes duplicates but loses the original order. To keep first-seen order, use `dict.fromkeys`, since a dict remembers insertion order (next section):

```
words = ["to", "be", "or", "not", "to", "be"]
print(len(set(words)))
print(list(dict.fromkeys(words)))
```

```
4
['to', 'be', 'or', 'not']
```

Dicts: Patterns You Will Reuse

A dict maps hashable keys to any values. Since Python 3.7 it **remembers insertion order** (a language guarantee, unlike sets). Re-assigning an existing key keeps its position; deleting and re-adding moves it to the end. Two dicts are equal when they hold the same pairs, regardless of order, which is different from lists.

```
d2 = {"b": 1, "a": 2}
d2["c"] = 3
del d2["b"]
d2["b"] = 4
print(d2, list(d2))
print({"x": 1, "y": 2} == {"y": 2, "x": 1}, [1, 2] == [2, 1])
```

```
{'a': 2, 'c': 3, 'b': 4} ['a', 'c', 'b']
True False
```

Counting and Grouping

The two commonest jobs are counting things and collecting things by key. Plain dicts do both with `get` and `setdefault`; the standard library adds `Counter` and `defaultdict` to say it

more directly.

```
from collections import defaultdict, Counter

d = {}
for w in words:
    d[w] = d.get(w, 0) + 1
print(d)

g = defaultdict(list)
for w in words:
    g[len(w)].append(w)
print(dict(g))

print(Counter(words).most_common(2))
```

```
{'to': 2, 'be': 2, 'or': 1, 'not': 1}
{2: ['to', 'be', 'or', 'to', 'be'], 3: ['not']}
[('to', 2), ('be', 2)]
```

Safe Lookups and Merging

```
print(d2.setdefault("z", 0), d2.pop("q", "none"))
print({**{"a": 1, "b": 2}, **{"b": 3}}, {"a": 1} | {"a": 2})
```

```
0 none
{'a': 1, 'b': 3} {'a': 2}
```

In both merges the right-hand dict wins on a shared key. `d[k]` raises `KeyError` for a missing key; `d.get(k, default)` and `d.pop(k, default)` do not.

A DECISION HABIT

Ask what the question is. "Is it in there?" or "what is unique?" points to a set. "What goes with this key?" points to a dict. "Which position?" or "keep it in sequence" points to a list. "A fixed record, or a key made of several parts" points to a tuple.

Challenges

EXERCISE 1

Write `can_be_key(value)` that returns True if the value could be used as a dict key (use `hash` inside `try` / `except`). Test it on 5, "hi", (1, 2), [1, 2], (1, [2]), {1}, frozenset({1}) and None, and say which results come from the tuple rule.

View solution

EXERCISE 2

Write `unique_in_order(items)` two ways: with `dict.fromkeys`, and with a `seen` set plus a result list. Check both give `[3, 1, 2, 4]` for `[3, 1, 3, 2, 1, 4]`, and explain why plain `list(set(items))` is not good enough.

View solution

EXERCISE 3

Write `group_by_first_letter(words)` returning a dict from first letter to the list of words starting with it (use `setdefault`). Then, given the sets `{"ann", "bob", "cy"}` and `{"bob", "dee"}`, print the sorted union, intersection, difference and symmetric difference.

View solution

QUICK REFERENCE

- **Tuple:** fixed record; hashable if everything inside is hashable
- **Hashable** means immutable in practice; equal values must hash equal

- **Set:** unique values, fast membership, algebra with `| & - ^`; never rely on order
- **Dict:** key to value, insertion order guaranteed, equality ignores order
- **Counting and grouping:** `Counter`, `defaultdict`, `setdefault`
- **Order-preserving unique:** `list(dict.fromkeys(items))`

COMING NEXT

Data Structures in Practice 4 covers stacks and queues: last-in-first-out and first-in-first-out, `deque`, the `queue` module, and uses such as undo history and task queues.

Data Structures in Practice

Chapter 4 · Stacks, Queues

Stacks and queues are not new containers so much as **rules about which end you use**. A stack takes and gives from the same end. A queue takes at one end and gives from the other. Choosing the right underlying structure (Data Structures in Practice 1 and 2) is what makes those rules cheap.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every snippet below was run and the output shown is what it printed. No timings are claimed; costs are described as patterns, as in earlier chapters.

Stack: Last In, First Out

The most recently added item is the first one out, like a pile of plates. A Python `list` is a fine stack: `append` pushes and `pop()` pops, both at the end, which Data Structures in Practice 2 showed to be cheap (amortised constant).

Queue: First In, First Out

The oldest item leaves first, like a line at a counter. A list is a poor queue because `pop(0)` shifts every remaining item. `collections.deque` is built for both ends: `append` and `popleft` are both constant cost.

```
from collections import deque

stack = []
stack.append("a"); stack.append("b"); stack.append("c")
print(stack.pop(), stack.pop(), stack)

q = deque()
q.append("a"); q.append("b"); q.append("c")
print(q.popleft(), q.popleft(), q)
```

```
try:
    [].pop()
except IndexError as e:
    print("IndexError:", e)
try:
    deque().popleft()
except IndexError as e:
    print("IndexError:", e)
```

```
c b ['a']
a b deque(['c'])
IndexError: pop from empty list
IndexError: pop from an empty deque
```

Same three items in, opposite order out. Popping an empty stack or queue raises `IndexError`, so test for emptiness first (`if stack:`) or catch the error.

NEED	GOOD CHOICE	AVOID
Stack	**list** (`append`, `pop`) or **deque**	Inserting or removing at the front of a list
Queue	**deque** (`append`, `popleft`)	`list.pop(0)`: shifts everything
Queue shared between threads	**queue.Queue**	A bare list or deque with hand-made locking
Keep only the last N items	**deque(maxlen=N)**	Slicing a list after every append

Stack in Action: Matching Brackets

Anything that must be closed in the reverse order it was opened is a stack problem. Push each opener; on a closer, the top of the stack must be its partner.

```
def balanced(text):
    pairs = {")": "(", "]": "[", "}": "{"}
```

```
    st = []
    for ch in text:
        if ch in "([{":
            st.append(ch)
        elif ch in pairs:
            if not st or st.pop() != pairs[ch]:
                return False
    return not st

for s in ["(a[b]{c})", "(]", "((", "))", ""]:
    print(repr(s), balanced(s))
```

```
'(a[b]{c})' True
'(]' False
'((' False
'))' False
''  True
```

Three failure modes are covered: a wrong partner (`"(]"`), a closer with nothing open (`"))"`, caught by `not st`), and openers left over at the end (`"(("`, caught by the final `not st`). The same pattern underlies the call stack, expression evaluation and back-tracking.

Two Stacks: Undo and Redo

Undo history is a stack of earlier states. Redo needs a second stack that catches whatever undo removes. A new edit invalidates the redo history, so it is cleared.

```
class Editor:
    def __init__(self):
        self.text = ""
        self.undo_stack = []
        self.redo_stack = []

    def type(self, s):
        self.undo_stack.append(self.text)
```

```
            self.redo_stack.clear()
            self.text += s

        def undo(self):
            if self.undo_stack:
                self.redo_stack.append(self.text)
                self.text = self.undo_stack.pop()

        def redo(self):
            if self.redo_stack:
                self.undo_stack.append(self.text)
                self.text = self.redo_stack.pop()

    e = Editor()
    e.type("Hi"); e.type(" there")
    print(repr(e.text))
    e.undo(); print(repr(e.text))
    e.redo(); print(repr(e.text))
    e.undo(); e.type("!"); e.redo(); print(repr(e.text))
```

```
    'Hi there'
    'Hi'
    'Hi there'
    'Hi!'
```

In the last line the redo did nothing: typing `"!"` after an undo cleared the redo stack, which is how editors behave. (This stores whole snapshots for clarity; a real editor would store small changes.)

Deque: Both Ends, and a Size Limit

With `maxlen`, a deque silently drops from the opposite end when full. That is a ready-made "most recent N" buffer. `rotate` moves items around the ring, which suits round-robin turns.

```
    recent = deque(maxlen=3)
```

```
for i in range(1, 6):
    recent.append(i)
print(recent)

rr = deque(["A", "B", "C"])
order = []
for _ in range(5):
    order.append(rr[0])
    rr.rotate(-1)
print(order, rr)
```

```
deque([3, 4, 5], maxlen=3)
['A', 'B', 'C', 'A', 'B'] deque(['C', 'A', 'B'])
```

> **INDEXING THE MIDDLE OF A DEQUE**
>
> A deque is quick at both ends, but reaching an item in the middle by index is not its strength (unlike a list). If you find yourself indexing a deque in the middle, you probably want a different structure.

The queue Module: Task Queues

`queue.Queue` wraps the same idea with locking, so several threads can safely put and get. It offers `put`, `get`, `qsize`, `empty`, an optional `maxsize` (a full queue makes `put` wait, or raise `queue.Full` with `put_nowait`), and `queue.LifoQueue` for a thread-safe stack.

```
import queue

tq = queue.Queue()
for t in ["t1", "t2", "t3"]:
    tq.put(t)
print(tq.qsize(), tq.get(), tq.get(), tq.empty())

try:
    queue.Queue().get_nowait()
except queue.Empty:
```

```
    print("queue.Empty")

lq = queue.LifoQueue()
for t in [1, 2, 3]:
    lq.put(t)
print(lq.get(), lq.get())

bq = queue.Queue(maxsize=1)
bq.put("x")
try:
    bq.put_nowait("y")
except queue.Full:
    print("queue.Full")
```

```
3 t1 t2 False
queue.Empty
3 2
queue.Full
```

The classic use is producer and worker: one side puts tasks, a worker thread gets and processes them, and a `None` acts as a "stop" signal.

```
import threading

results = []
wq = queue.Queue()

def worker():
    while True:
        item = wq.get()
        if item is None:
            break
        results.append(item * item)

th = threading.Thread(target=worker)
th.start()
for n in [1, 2, 3, 4]:
    wq.put(n)
```

```
wq.put(None)
th.join()
print(results)
```

```
[1, 4, 9, 16]
```

Because there is one worker and a first-in-first-out queue, results arrive in submission order. With several workers the order of results would not be guaranteed. For a single-threaded program, a plain `deque` is simpler and avoids the locking overhead.

A DECISION HABIT

Ask which item you need next. "The newest" is a stack. "The oldest" is a queue. "The most urgent" is neither, and is the subject of Data Structures in Practice 6.

Challenges

EXERCISE 1

Write `rpn(expr)` that evaluates a postfix expression such as `"3 4 + 2 *"` using a stack: push numbers, and on `+`, `-` or `*` pop two values (the second pop is the left operand) and push the result. Check that `"3 4 + 2 *"` gives 14 and `"5 1 2 + 4 * + 3 -"` gives 14.

View solution

EXERCISE 2

Write `last_n(lines, n)` that returns the last `n` items of any iterable using `deque(maxlen=n)`, so it never holds more than `n` at once. Test it on `range(10)` with n=3, on a list shorter than n, and with n=1. Then say what happens with `list.pop(0)` as a queue on a very long list and why `deque` avoids it.

View solution

EXERCISE 3

Write `hot_potato(names, k)` using a deque: repeatedly rotate so the k-th person is at the front, remove them, and record the order of removal until one remains. Return the list of removed names and the survivor. For `["Ann", "Bob", "Cy", "Dee", "Eve"]` with k=3, work out the answer by hand first, then check it.

View solution

QUICK REFERENCE

- **Stack (LIFO):** `list.append` and `list.pop()`
- **Queue (FIFO):** `deque.append` and `deque.popleft()`; avoid `list.pop(0)`
- **Empty pop** raises `IndexError`; check first
- **deque(maxlen=N):** keeps the latest N, drops from the other end
- **rotate(k):** shifts the ring, handy for round-robin
- **queue.Queue / LifoQueue:** thread-safe, with `maxsize`, `queue.Empty` and `queue.Full`
- **Undo and redo:** two stacks; a new edit clears redo

COMING NEXT

Data Structures in Practice 5 covers linked lists: nodes and pointers, how they work, and why in Python they rarely beat the list.

Data Structures in Practice

Chapter 5 · Linked Lists

A **linked list** stores items in separate small objects, each holding a value and a reference to the next one. Nothing is stored side by side, which is the opposite of the array inside a Python `list` (Data Structures in Practice 2). It is a classic structure that every programmer should be able to build, and understanding it also explains why it is rarely the right tool in Python.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every snippet below was run and the output shown is what it printed. No timings or memory sizes are quoted, because they differ between machines; costs are described as patterns.

Nodes and Pointers

Each **node** holds a `value` and a `next` reference. The last node's `next` is `None`. The list itself is just a reference to the first node, called the **head**. To visit every item you start at the head and follow `next` until you run out.

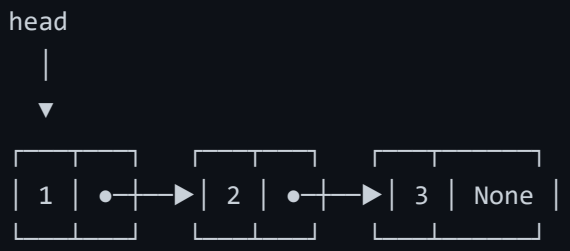
```
class Node:
    def __init__(self, value, next=None):
        self.value = value
        self.next = next

a = Node(1)
b = Node(2)
c = Node(3)
a.next = b
b.next = c

n = a
while n:
    print(n.value, end=" ")
    n = n.next
print()
```

```
1 2 3
```

The picture to hold in your head:



A Small LinkedList Class

Wrapping the head in a class gives a tidy interface. This version keeps a `size` counter so that `len()` does not have to walk the chain, and an `__iter__` so that `list(ll)` and `for` loops work.

```
class LinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def push_front(self, value):
        self.head = Node(value, self.head)
        self.size += 1

    def append(self, value):
        node = Node(value)
        if self.head is None:
            self.head = node
        else:
            cur = self.head
            while cur.next:
                cur = cur.next
```

```
            cur.next = node
        self.size += 1

    def __iter__(self):
        cur = self.head
        while cur:
            yield cur.value
            cur = cur.next

    def __len__(self):
        return self.size

    def find(self, value):
        cur = self.head
        while cur:
            if cur.value == value:
                return cur
            cur = cur.next
        return None

    def remove(self, value):
        prev, cur = None, self.head
        while cur:
            if cur.value == value:
                if prev:
                    prev.next = cur.next
                else:
                    self.head = cur.next
                self.size -= 1
                return True
            prev, cur = cur, cur.next
        return False

    def insert_after(self, node, value):
        node.next = Node(value, node.next)
        self.size += 1
```

```
ll = LinkedList()
for v in [2, 3, 4]:
    ll.append(v)
ll.push_front(1)
print(list(ll), len(ll))
```

```
ll.insert_after(ll.find(2), 99)
print(list(ll), len(ll))
print(ll.remove(99), ll.remove(1), ll.remove(42))
print(list(ll), len(ll))
```

```
[1, 2, 3, 4] 4
[1, 2, 99, 3, 4] 5
True True False
[2, 3, 4] 3
```

Note the two cases in `remove`: deleting the head means moving `self.head`, while deleting anywhere else means the previous node must skip over it. Tracking `prev` as you walk is what makes that possible, because a node cannot see backwards.

Reversing in Place

A favourite exercise, and a good test of whether you can hold several references at once. Walk the chain and flip each `next` to point backwards. The original `next` must be saved before it is overwritten.

```
def reverse(head):
    prev = None
    cur = head
    while cur:
        nxt = cur.next      # save it before overwriting
        cur.next = prev
        prev = cur
        cur = nxt
    return prev

ll.head = reverse(ll.head)
print(list(ll))
```

```
[4, 3, 2]
```

No new nodes were made: the same objects were relinked. The function returns the new head, so the caller must store it.

Where Linked Lists Win, on Paper

In theory the trade is simple. Once you hold a reference to a node, inserting after it or unlinking its neighbour changes one or two references, with no shifting of anything else. A list has to move every item after the position.

OPERATION	PYTHON LIST	LINKED LIST
Index `x[i]`	Constant	Walk *i* nodes
Append at the end	Amortised constant	Constant with a tail reference, otherwise a full walk
Insert or remove at the front	Shifts every item	Constant
Insert or remove next to a node you already hold	Shifts items after it	Constant
Find a value	Walk (linear)	Walk (linear)
Extra memory per item	One reference	A whole object plus one or two references
Memory layout	Contiguous references	Scattered objects

Why Python Rarely Needs One

- **The "constant insertion" advantage needs a node in hand.** Finding that node is a linear walk, so inserting at a position you only know by index or value is linear for both structures.
- **Every node is a full Python object.** It carries object overhead on top of the value and the reference, so a linked list uses noticeably more memory than a list of the same items.

- **Scattered objects are slower to walk.** A list's references sit next to each other; a chain of nodes can be spread across memory, which tends to make traversal slower in practice even when the step count is equal.
- **Python's own list is written in C.** Shifting items inside it is a tight low-level loop, while your `while cur:` loop runs through the interpreter one step at a time. For all but very large sizes, the "slow" list is the faster choice.
- **The common cases already have a tool.** Fast work at both ends is a `deque` (Data Structures in Practice 4), which is itself built from linked blocks. Fast membership is a set or dict (Data Structures in Practice 3).

> **DO NOT ASSUME "INSERT IN THE MIDDLE IS FASTER"**
>
> That sentence is true of the textbook model and often false in measured Python code. If it matters for your data, measure both on realistic sizes (Data Structures in Practice 1) before building a custom structure.

Where You Still Meet Linked Structures

The idea is everywhere even when you do not build the list yourself: `deque` uses linked blocks internally, trees and graphs (Data Structures in Practice 7 and 8) are nodes with several references, and hash table chains, undo histories in some systems and operating-system queues are all linked. Lower-level languages use plain linked lists far more often, and technical interviews ask about them for exactly that reason.

Two Pointer Habits

Most linked-list problems rely on a few patterns.

- **Track `prev` and `cur`.** You need the previous node to remove one (as in `remove`) or to flip direction (as in `reverse`).
- **Slow and fast pointers.** Move one pointer a step at a time and another two steps. When the fast one finishes, the slow one is at the middle; if there is a loop, the fast one catches the slow one. Exercises 1 and 2 use this.
- **A dummy node.** Start with a placeholder in front of the real head so the "list is empty" case needs no special code. Exercise 3 uses this.

SINGLY AND DOUBLY LINKED

This chapter's nodes only point forwards (singly linked). A **doubly linked** node also stores a `prev` reference, so you can walk backwards and unlink a node without hunting for its predecessor, at the price of one more reference per node and more links to keep consistent.

Challenges

EXERCISE 1

Using `Node` and a helper `build(values)` that creates a chain from a Python list, write `middle(head)` that returns the value of the middle node in one pass, using slow and fast pointers and no length count. Test it on 5 nodes, 4 nodes (say which middle you return), 1 node and an empty list (return `None`).

View solution

EXERCISE 2

Write `has_cycle(head)` that returns `True` if following `next` would loop forever. Build a four-node chain, check it gives `False`, then make the last node point back to the second and check it gives `True`. Explain why you compare nodes with `is` rather than `==`.

View solution

EXERCISE 3

Write `merge_sorted(a, b)` that merges two already-sorted linked lists into one sorted chain by relinking the existing nodes (no new nodes except one dummy). Test `[1, 4, 6]` with `[2, 3, 7, 9]`, an empty list with `[5]`, and `[1, 1]` with `[1]`. Add a short `to_list(head)` helper to print results.

View solution

QUICK REFERENCE

- **Node:** a value plus `next`; the list is its `head`
- **Strengths:** constant-cost insert or remove at the front or next to a node you hold

- **Weaknesses:** no indexing, linear search, more memory, scattered layout
- **In Python:** use `list` by default and `deque` for both ends; build a linked list to learn or when relinking nodes is the point
- **Patterns:** `prev` / `cur`, slow/fast, dummy head
- **Reverse:** save `next` first, flip, advance; return the new head
- **Measure** before choosing a custom structure for speed

COMING NEXT

Data Structures in Practice 6 covers heaps and priority queues: `heapq`, finding the top k items and simple scheduling, the answer to "the most urgent".

Data Structures in Practice

Chapter 6 · Heaps

A **priority queue** hands back the item that matters most, not the one that arrived first or last. A **heap** is the structure that does this cheaply: it keeps just enough order to find the smallest item instantly, and does not waste effort sorting the rest. Python provides it in the `heapq` module.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every snippet below was run and the output shown is what it printed. No timings are quoted, because they differ between machines; costs are described as patterns.

What a Heap Gives You

`heapq` works on an ordinary Python `list`. The rule it maintains is that the item at index 0 is always the smallest, and every item is no larger than its two "children" at positions `2i+1` and `2i+2`. Nothing else is promised: the list as a whole is *not* sorted.

```
import heapq

h = []
for n in [5, 1, 8, 3, 2, 9]:
    heapq.heappush(h, n)
print(h)
print(h[0])

out = []
while h:
    out.append(heapq.heappop(h))
print(out)
```

```
[1, 2, 8, 5, 3, 9]
1
```

```
[1, 2, 3, 5, 8, 9]
```

The printed list is a valid heap but is not in order (8 comes before 5). Only `h[0]` is guaranteed. Popping repeatedly produces everything in sorted order, because each pop re-establishes the rule for what is left. The picture of that list as a tree:

```
index:   0  1  2  3  4  5
value: [ 1, 2, 8, 5, 3, 9 ]

           1
         /   \
        2     8
       / \   /
      5   3 9
```

Building and Reading a Heap

If you already have all the items, do not push them one by one. `heapify` rearranges a list in place in linear time.

```
data = [5, 1, 8, 3, 2, 9]
heapq.heapify(data)
print(data)
print(heapq.heappop(data), data)
```

```
[1, 2, 8, 3, 5, 9]
1 [2, 3, 8, 9, 5]
```

OPERATION	CALL	COST
Look at the smallest	`h[0]`	Constant
Add an item	`heappush(h, x)`	Logarithmic
Remove the smallest	`heappop(h)`	Logarithmic
Turn a list into a heap	`heapify(h)`	Linear
Find an arbitrary value	`x in h`	Linear (a heap is not searchable)
Fully sorted order	pop everything	n pops, so n log n

NEVER REORDER A HEAP BY HAND

Do not call `h.sort()` on something you will use as a heap (a sorted list happens to be a valid heap, but you will soon be tempted to `insert` or `append` directly, which silently breaks the rule). Change a heap only through `heappush`, `heappop`, `heapreplace` and `heapify`.

Max-Heaps by Negation

`heapq` only offers a min-heap. For "largest first" with numbers, store the negatives and flip the sign on the way out.

```
nums = [5, 1, 8, 3]
mh = [-n for n in nums]
heapq.heapify(mh)
print(-heapq.heappop(mh))
```

```
8
```

Priorities, Ties and Tuples

To attach a payload to a priority, push a tuple. Python compares tuples element by element, so the priority decides first and the payload only matters on a tie.

```
h = []
heapq.heappush(h, (2, "write"))
heapq.heappush(h, (1, "deploy"))
heapq.heappush(h, (2, "test"))
print(heapq.heappop(h), heapq.heappop(h), heapq.heappop(h))
```

```
(1, 'deploy') (2, 'test') (2, 'write')
```

Two things went wrong quietly here. On a tie the payload strings were compared, so "test" beat "write" alphabetically rather than by arrival order. Worse, if the payloads cannot be compared at all, a tie raises an error:

```
h = []
try:
    heapq.heappush(h, (1, {"a": 1}))
    heapq.heappush(h, (1, {"b": 2}))
except TypeError as e:
    print("TypeError", e)
```

```
TypeError '<' not supported between instances of 'dict' and 'dict'
```

The standard fix is a **tie-breaker counter** in the middle of the tuple: `(priority, count, item)`. The count is unique and increasing, so comparison never reaches the item, and equal priorities come out in first-in, first-out order.

```
import itertools

counter = itertools.count()
h = []
for pr, name in [(2, "write"), (1, "deploy"), (2, "test"), (1, "backup")]:
    heapq.heappush(h, (pr, next(counter), name))
while h:
    pr, _, name = heapq.heappop(h)
    print(pr, name)
```

```
1 deploy
1 backup
2 write
2 test
```

Top k Without Sorting Everything

"The three highest scores" does not need a full sort. `nlargest` and `nsmallest` take a `key` function like `sorted` does.

```
scores = [41, 7, 93, 15, 88, 62, 3, 77]
print(heapq.nlargest(3, scores), heapq.nsmallest(3, scores))
words = ["pear", "fig", "banana", "kiwi", "apple"]
print(heapq.nlargest(2, words, key=len))
print(heapq.nsmallest(2, words, key=len))
```

```
[93, 88, 77] [3, 7, 15]
['banana', 'apple']
['fig', 'pear']
```

The idea behind them is worth knowing, because it works on streams too. Keep a *min*-heap holding the k best items seen so far. Its root is the weakest of the group, so a new item only matters if it beats the root, and then `heapreplace` swaps them in one step. Memory stays at k items however long the input is.

```
def top_k(stream, k):
    h = []
    for x in stream:
        if len(h) < k:
            heapq.heappush(h, x)
        elif x > h[0]:
            heapq.heapreplace(h, x)
    return sorted(h, reverse=True)

print(top_k(scores, 3))
```

```
[93, 88, 77]
```

NEED	BEST TOOL
Smallest or largest single item	`min()` / `max()`
A few best items (k small)	`heapq.nlargest` / `nsmallest`
Most of the items, or all, in order	`sorted()`
Repeatedly "the best now" while items keep arriving	a heap

A Few More Tools

- **`heapreplace(h, x)`** pops the smallest, then pushes `x`. The returned value may be larger than `x`.
- **`heappushpop(h, x)`** pushes `x` first, then pops the smallest, so it returns `x` itself if `x` is already the smallest.

- `heapq.merge(*sorted_inputs)` lazily merges several already-sorted inputs into one sorted stream.

```
print(heapq.heappushpop([4, 6], 1), heapq.heappushpop([4, 6], 9))
print(list(heapq.merge([1, 4, 9], [2, 3, 10], [0, 5])))
```

```
1 4
[0, 1, 2, 3, 4, 5, 9, 10]
```

queue.PriorityQueue

The `queue` module from Data Structures in Practice 4 also has `PriorityQueue`. It is a heap wrapped in locks so several threads can share it safely. It uses the same tuple convention, with the same tie-breaking trap.

```
import queue

q = queue.PriorityQueue()
q.put((2, "b"))
q.put((1, "a"))
print(q.get(), q.qsize())
```

```
(1, 'a') 1
```

Use plain `heapq` in single-threaded code (it is lighter and gives you `h[0]` to peek) and `PriorityQueue` when threads share the queue.

Scheduling, and Cancelling Lazily

Schedulers, "next event" simulations, Dijkstra's shortest paths (Data Structures in Practice 8) and merging many sorted files are all the same pattern: repeatedly take the most urgent item while new ones keep arriving. One catch is that you cannot cheaply remove an item from the middle of a heap. The usual answer is **lazy deletion**: mark the entry as cancelled, and discard it when it eventually reaches the top. Exercise 3 builds this.

HEAP, SORTED LIST OR SET?

If you only ever need the best item and items keep arriving, a heap is right. If you need to search, remove arbitrary items or walk everything in order, a heap is the wrong shape: use a sorted list (with `bisect`), a dict or a tree (Data Structures in Practice 7).

Challenges

EXERCISE 1

Write a class `KthLargest(k)` with an `add(x)` method that accepts a stream of numbers and returns the current k-th largest value (or `None` until k values have arrived). Use a min-heap of at most k items. Feed it `4, 5, 8, 2, 10, 1, 9` with k = 3 and print each result.

View solution

EXERCISE 2

Write `heap_sort(items)` using only `heapify` and `heappop` (it must not change the input), and `merge_sorted(*lists)` that merges any number of sorted lists by keeping one candidate per list in a heap. Check the second against `heapq.merge`, and test it with an empty list and with no arguments.

View solution

EXERCISE 3

Build a `Scheduler` class with `add(priority, name)` (returning a ticket), `cancel(ticket)` and `next_task()` (returning `None` when empty). Lower numbers run first, equal priorities run in the order

added, and cancelled tasks never run. Cancel lazily. Show it working with five tasks, one of them cancelled.

View solution

QUICK REFERENCE

- **Heap:** a list where `h[0]` is always the smallest; the rest is only partly ordered
- **Costs:** peek constant; push and pop logarithmic; heapify linear
- **Max-heap:** store negatives (numbers) and flip the sign back
- **Payloads:** push `(priority, count, item)` so ties are first-in, first-out and items are never compared
- **Top k:** `nlargest` / `nsmallest`, or a size-k min-heap for streams
- **Others:** `heapreplace`, `heappushpop`, `heapq.merge`, `queue.PriorityQueue` for threads
- **Cancelling:** mark and skip when it surfaces (lazy deletion)
- **Not for:** searching, arbitrary removal or in-order walking

COMING NEXT

Data Structures in Practice 7 covers trees: binary trees, binary search trees, traversal and balance. A heap is a tree squeezed into a list; next the tree gets its own nodes.

Data Structures in Practice

Chapter 7 · Trees

A **tree** is a structure where each item (a *node*) points to zero or more *children*, and no node has more than one parent. Folders on a disk, the nesting of an HTML page and a company's reporting lines are all trees. A **binary tree** limits each node to at most two children, called left and right. Python has no built-in tree type, so you build one from small objects, just as in Data Structures in Practice 5, but with two pointers instead of one.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every snippet below was run and the output shown is what it printed. No timings are quoted, because they differ between machines. Heights and counts are exact, because they do not depend on the machine.

Nodes and Vocabulary

```
class Node:
    def __init__(self, value, left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

tree = Node(8, Node(3, Node(1), Node(6)), Node(10, None, Node(14)))
```

That builds this shape, which the rest of the chapter reuses:

```
        8
       / \
      3   10
     / \    \
    1   6    14
```

TERM	MEANING	IN THE PICTURE
Root	The top node, with no parent	8
Leaf	A node with no children	1, 6, 14
Depth of a node	Steps from the root down to it	6 is at depth 2
Height of the tree	Number of levels (this chapter counts nodes on the longest path)	3
Subtree	A node together with everything below it	3, 1 and 6 form one

Walking a Tree: Four Orders

A list has one obvious way to visit every item. A tree has several. The three depth-first orders differ only in when the node itself is visited relative to its two subtrees, and the fourth goes level by level using a queue (Data Structures in Practice 4).

```
def inorder(n):      # left, node, right
    if n is None:
        return []
    return inorder(n.left) + [n.value] + inorder(n.right)

def preorder(n):     # node, left, right
    if n is None:
        return []
    return [n.value] + preorder(n.left) + preorder(n.right)

def postorder(n):    # left, right, node
    if n is None:
        return []
    return postorder(n.left) + postorder(n.right) + [n.value]

from collections import deque
def levelorder(n):   # top to bottom, left to right
    out, q = [], deque([n] if n else [])
    while q:
        cur = q.popleft()
        out.append(cur.value)
        if cur.left:  q.append(cur.left)
        if cur.right: q.append(cur.right)
    return out
```

```
print(inorder(tree))
print(preorder(tree))
print(postorder(tree))
print(levelorder(tree))
```

```
[1, 3, 6, 8, 10, 14]
[8, 3, 1, 6, 10, 14]
[1, 6, 3, 14, 10, 8]
[8, 3, 10, 1, 6, 14]
```

Each has a typical use. Pre-order copies a tree (parents before children). Post-order deletes or totals a tree (children before parents). Level order finds the nearest thing first. In-order is special for search trees, as the next section shows. The recursive versions above also build many temporary lists; for large trees you would use a generator, or the explicit-stack version in Exercise 3.

Binary Search Trees

A **binary search tree** (BST) adds one rule: for every node, everything in its left subtree is smaller and everything in its right subtree is larger. The tree above obeys it. That rule makes searching a series of left-or-right decisions, each throwing away a whole subtree, which is binary search (Data Structures in Practice 1) on a linked structure.

```
class BST:
    def __init__(self):
        self.root = None
        self.size = 0

    def insert(self, v):
        if self.root is None:
            self.root = Node(v); self.size += 1; return True
        cur = self.root
        while True:
```

```
            if v == cur.value:
                return False              # duplicates ignored
            side = "left" if v < cur.value else "right"
            nxt = getattr(cur, side)
            if nxt is None:
                setattr(cur, side, Node(v)); self.size += 1; return True
            cur = nxt

    def __contains__(self, v):
        cur = self.root
        while cur:
            if v == cur.value: return True
            cur = cur.left if v < cur.value else cur.right
        return False

b = BST()
for v in [8, 3, 10, 1, 6, 14, 6]:
    print(v, b.insert(v))
print(inorder(b.root), b.size)
print(6 in b, 7 in b)
```

```
8 True
3 True
10 True
1 True
6 True
14 True
6 False
[1, 3, 6, 8, 10, 14] 6
True False
```

Notice the in-order walk returned the values *sorted*. That is a direct consequence of the rule (everything smaller, then the node, then everything larger), and it is why a BST can give you sorted output and range queries without a separate sort. Insertion and lookup follow one path from the root to a leaf, so their cost is proportional to the **height**, not to the number of items.

Balance: The Whole Point

The cost being "proportional to the height" is only good news if the height stays small. A tree with n items can be as short as about log2(n) levels, or as tall as n. The shape depends on the insertion order.

```
import random

def height(n):
    if n is None:
        return 0
    return 1 + max(height(n.left), height(n.right))

s = BST()
for v in range(1, 201):          # already sorted
    s.insert(v)
print(height(s.root))

vals = list(range(1, 201))
random.seed(1)
random.shuffle(vals)                  # same values, random order
r = BST()
for v in vals:
    r.insert(v)
print(height(r.root))
```

```
200
14
```

The same 200 values give a height of 200 when inserted in order (every new value goes right of the last, so the "tree" is a linked list) and 14 when shuffled (the exact figure varies with the shuffle, but stays close to the ideal of 8). Sorted input is the commonest real-world way to hit the bad case: data read from a sorted file or an auto-incrementing key.

DEEP TREES AND RECURSION

Python limits recursion to about a thousand nested calls. A recursive `height` on a chain of 5,000 nodes (the sorted case) raised `RecursionError` when tried. Skewed trees therefore break recursive

code as well as being slow. The iterative walks in Exercise 3 have no such limit.

There are two standard fixes. **Self-balancing trees** (AVL trees, red-black trees) rotate nodes during insertion to keep the height near log2(n); they are tricky to write, and rarely worth writing by hand. Or you can **build the tree from sorted data deliberately**, by taking the middle item as the root and recursing on each half, which is Exercise 1.

Do You Actually Need One in Python?

Mostly no. Python's built-in tools already cover what a BST offers, and they run as optimised C code rather than interpreted node-hopping.

YOU NEED	REACH FOR	WHY
Membership or lookup by key	`set` / `dict`	Hashing is constant time (Data Structures in Practice 3)
Sorted data, search and range queries	A sorted list with `bisect`	Binary search with no node objects; inserting is linear, but fast in practice for moderate sizes
Only the smallest or largest, repeatedly	`heapq`	Data Structures in Practice 6
Hierarchical data (folders, menus, parsed documents, game moves)	A tree of your own nodes	The data really is a tree
Constant fast inserts *and* sorted order for millions of items	A balanced-tree library	Not in the standard library; this is where a real tree pays off

THE PRACTICAL RULE

Trees earn their place when the *data is naturally hierarchical*. Using one purely as a sorted container is usually a sign to try a sorted list or a dict first.

Challenges

EXERCISE 1

Write `from_sorted(items)`, which builds a balanced binary search tree from a sorted list by using the middle item as the root and recursing on each half (pass index bounds rather than slicing). Write `height` and `inorder` helpers and check sizes 0, 1, 7, 15 and 1000: print the size, the height and whether the in-order walk equals the input.

View solution

EXERCISE 2

Write `range_query(node, lo, hi)`, returning in order every value between `lo` and `hi` inclusive in a search tree, without visiting subtrees that cannot contain a match. Then write `is_bst(node)` that correctly rejects a tree in which every parent beats its direct children but a deeper node is on the wrong side. Build such a tree to prove it.

View solution

EXERCISE 3

Write iterative generators `inorder_iter` and `preorder_iter` using an explicit stack, and `levels` returning a list of lists, one per level, using a deque. Check them on the chapter's example tree, on an empty tree, and on a right-leaning chain of 5,000 nodes that a recursive version could not handle.

View solution

QUICK REFERENCE

- **Binary tree:** each node has a value, a left child and a right child (either may be `None`)
- **Orders:** in-order (left, node, right), pre-order (node first), post-order (node last), level order (queue)
- **BST rule:** left subtree smaller, right subtree larger; in-order gives sorted output
- **Cost:** insert and search follow one path, so cost is the height
- **Balance:** shuffled input gives height near log2(n); sorted input gives height n
- **Validation:** check each node against a window of allowed values, not only its parent
- **Deep trees:** recursion hits Python's limit; use an explicit stack
- **In Python:** prefer `dict`, `set`, `bisect` or `heapq` unless the data is truly hierarchical

COMING NEXT

Data Structures in Practice 8 covers graphs: adjacency lists, breadth-first and depth-first search, and shortest paths on small graphs. A tree is a graph with no loops and one way in; next the loops are allowed.

Data Structures in Practice

Chapter 8 · Graphs

A **graph** is a set of *nodes* (also called vertices) joined by *edges*. Unlike a tree (Data Structures in Practice 7) there is no root, nodes may have any number of neighbours, and loops are allowed. Road maps, friend networks, web links, flight routes and the dependencies between tasks are all graphs. Edges can be **undirected** (a two-way road) or **directed** (a one-way street), and they can carry a **weight** such as a distance or a cost.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every snippet below was run and the output shown is what it printed. No timings are quoted, because they differ between machines. Orders, paths and costs are exact, because they depend only on the data.

Storing a Graph

There are two classic layouts. An **adjacency matrix** is a grid with one row and one column per node; cell (a, b) says whether an edge joins them. An **adjacency list** keeps, for each node, only the list of its neighbours. In Python the natural adjacency list is a dict of lists.

```
graph = {
    "A": ["B", "C"],
    "B": ["A", "D"],
    "C": ["A", "D", "E"],
    "D": ["B", "C", "F"],
    "E": ["C", "F"],
    "F": ["D", "E"],
    "G": [],                  # a node with no edges
}
print(graph["C"])
print(len(graph), sum(len(v) for v in graph.values()) // 2)

def add_edge(g, a, b):
    g.setdefault(a, []).append(b)
    g.setdefault(b, []).append(a)
```

```
g = {}
for a, b in [("A", "B"), ("A", "C"), ("B", "D")]:
    add_edge(g, a, b)
print(g)
```

```
['A', 'D', 'E']
7 7
{'A': ['B', 'C'], 'B': ['A', 'D'], 'C': ['A'], 'D': ['B']}
```

Seven nodes and seven edges. Because the graph is undirected, each edge is stored twice (once under each end), which is why the total length is halved to count edges. The picture the first dict describes:

```
    A --- B
    |     |
    C --- D        G (alone)
    |     |
    E --- F
```

	ADJACENCY LIST (DICT OF LISTS)	ADJACENCY MATRIX
Memory	One entry per node plus one per edge	One cell per *pair* of nodes
List a node's neighbours	Read its list directly	Scan a whole row
Is there an edge a-b?	Scan a's list (use sets of neighbours for constant time)	One lookup
Best for	Sparse graphs (few edges per node), which is most real graphs	Dense graphs, or small graphs where edge tests dominate

The memory row matters quickly. A chain of 1,000 nodes needs 1,000,000 matrix cells to hold only 1,998 stored list entries (each of its 999 edges stored twice). Unless the graph is nearly complete, use a dict.

Breadth-First Search

Breadth-first search (BFS) explores a graph in rings: the start, then everything one edge away, then everything two edges away. It is the queue-based walk from Data Structures in Practice 4 and the level-order walk from Data Structures in Practice 7, with one change: a graph can loop back, so you must remember which nodes have already been seen, in a set (Data Structures in Practice 3).

```
from collections import deque

def bfs_order(g, start):
    seen = {start}
    q = deque([start])
    order = []
    while q:
        cur = q.popleft()
        order.append(cur)
        for nb in g[cur]:
            if nb not in seen:
                seen.add(nb)          # mark when queued, not when visited
                q.append(nb)
    return order

print(bfs_order(graph, "A"))
```

```
['A', 'B', 'C', 'D', 'E', 'F']
```

G never appears, because no path leads to it from A. Marking a node as seen at the moment it is queued (rather than when it is taken off) stops it being queued twice. Each node and each edge is handled a fixed number of times, so the cost is proportional to nodes plus edges.

Why BFS finds shortest routes

Since BFS reaches nodes in order of distance, the first time it reaches a node is by a route with the fewest edges. To recover that route, store for each node where you came from (its *parent*)

and walk backwards from the goal.

```
def shortest_path(g, start, goal):
    parent = {start: None}          # doubles as the "seen" set
    q = deque([start])
    while q:
        cur = q.popleft()
        if cur == goal:
            path = []
            while cur is not None:
                path.append(cur)
                cur = parent[cur]
            return path[::-1]
        for nb in g[cur]:
            if nb not in parent:
                parent[nb] = cur
                q.append(nb)
    return None                             # unreachable

print(shortest_path(graph, "A", "F"))
print(shortest_path(graph, "A", "G"))
```

```
['A', 'B', 'D', 'F']
None
```

Depth-First Search

Depth-first search (DFS) goes as far as it can along one route before backing up and trying another. It is the same loop as BFS with the queue swapped for a stack (Data Structures in Practice 4), so the most recently discovered node is explored next.

```
def dfs_order(g, start):
    seen, order, stack = set(), [], [start]
    while stack:
```

```
        cur = stack.pop()
        if cur in seen:
            continue
        seen.add(cur)
        order.append(cur)
        for nb in reversed(g[cur]):   # reversed so the first neighbour is tried first
            if nb not in seen:
                stack.append(nb)
    return order

print(dfs_order(graph, "A"))
```

```
['A', 'B', 'D', 'C', 'E', 'F']
```

From A it dived to B, then D, and only later came back to C. Both searches visit exactly the same set of nodes; only the order differs. This version uses an explicit stack, so it has no recursion limit (Data Structures in Practice 7 showed why that matters).

QUESTION	USE	WHY
Fewest edges from A to B?	BFS	Reaches nodes in order of distance
Can A reach B at all? What is connected to A?	Either	Both visit everything reachable
Find every separate cluster; detect loops	Usually DFS	Simple to write with a stack; see Exercise 1
Put tasks in a valid order, explore every option (puzzles, mazes)	DFS	Follows one possibility to its end before trying others
Memory on a wide graph	DFS	BFS holds a whole ring in its queue at once

Weighted Edges: Cheapest, Not Fewest

BFS counts edges, which is the right measure only when every edge costs the same. With weights, the route with the fewest edges is not always the cheapest. **Dijkstra's algorithm** fixes

this with the heap from Data Structures in Practice 6: always expand the unfinished node with the smallest total cost so far. For weighted graphs, store `(neighbour, weight)` pairs.

```
import heapq

wgraph = {
    "A": [("B", 7), ("C", 2)],
    "B": [("D", 1)],
    "C": [("B", 3), ("D", 8), ("E", 5)],
    "D": [("F", 4)],
    "E": [("D", 2), ("F", 9)],
    "F": [],
}

def dijkstra(g, start):
    dist = {start: 0}
    heap = [(0, start)]
    while heap:
        d, cur = heapq.heappop(heap)
        if d > dist.get(cur, float("inf")):
            continue                # a stale entry: a cheaper route was found since
        for nb, w in g[cur]:
            nd = d + w
            if nd < dist.get(nb, float("inf")):
                dist[nb] = nd
                heapq.heappush(heap, (nd, nb))
    return dist

print(dijkstra(wgraph, "A"))

# the same graph with weights ignored: BFS by edge count
plain = {k: [n for n, _ in v] for k, v in wgraph.items()}
print(shortest_path(plain, "A", "F"))
```

```
{'A': 0, 'B': 5, 'C': 2, 'D': 6, 'E': 7, 'F': 10}
['A', 'B', 'D', 'F']
```

The fewest-edges route A, B, D, F has three edges but costs 7 + 1 + 4 = 12. The cheapest route is A, C, B, D, F: four edges, but 2 + 3 + 1 + 4 = 10, as the costs printed above show. Notice B costs 5 via C, not 7 directly.

> **DIJKSTRA NEEDS NON-NEGATIVE WEIGHTS**
>
> The method assumes that once a node is the cheapest unfinished one, nothing can make it cheaper. A negative edge breaks that assumption and the answers can be wrong. If your costs can be negative, a different algorithm is needed. Exercise 3 makes the function reject them.

Picking a Representation in Python

SITUATION	REACH FOR
Most graphs, sparse, unweighted	`dict` of lists; BFS with `deque`
Need to test "is there an edge?" often	`dict` of `set`s
Weighted edges	`dict` of lists of `(neighbour, weight)`, or a dict of dicts
A grid (maze, map, board)	No graph at all: compute neighbours from coordinates (Exercise 2)
Very large or specialised graphs	A graph library; the standard library has none

Challenges

EXERCISE 1

Write `components(g)` for an undirected graph, returning a list of the separate connected groups (each a sorted list), and `has_cycle(g)`, which says whether any group contains a loop. Remember that each undirected edge is stored twice, so walking back along the edge you came from must not count as a loop. Test with a graph that has three groups and no loop, then add one edge to create a loop, and check an empty graph.

View solution

EXERCISE 2

A maze is a list of strings where `#` is a wall, `S` the start and `G` the goal. Write `solve(grid)` using BFS, moving up, down, left and right, that returns the shortest path as a list of (row, column) positions, or `None` if the goal cannot be reached. Do not build a dict of neighbours: compute them from the coordinates. Print the path length and draw the route on the maze with `*`. Test a blocked maze and a maze where the start is next to the goal.

View solution

EXERCISE 3

Extend Dijkstra's algorithm into `cheapest(g, start, goal)`, returning `(cost, path)` or `None` if the goal is unreachable. Track a parent for each node to rebuild the path, stop as soon as the goal is taken from the heap, and raise `ValueError` if any edge weight is negative. Test it on the chapter's weighted graph, with start equal to goal, with an unreachable goal, and with a negative weight.

View solution

QUICK REFERENCE

- **Graph:** nodes joined by edges; edges may be directed and may carry weights
- **Storage:** a `dict` of lists (adjacency list) for sparse graphs; a matrix only for dense ones
- **Undirected:** store each edge under both ends
- **BFS:** queue (`deque`) plus a seen set; fewest edges first
- **DFS:** the same loop with a stack; follows one route to its end
- **Cost:** BFS and DFS are proportional to nodes plus edges
- **Paths:** keep a parent for each node and walk back from the goal
- **Weights:** Dijkstra with `heapq`; skip stale heap entries; no negative weights
- **Grids:** compute neighbours from coordinates instead of building a graph

COMING NEXT

Data Structures in Practice 9 builds a hash table by hand: collisions, load factor and a small working dictionary, so you can see why the dict you have used all course is so fast.

Data Structures in Practice

Chapter 9 · Hash Tables Built by Hand

You have used `dict` and `set` in every chapter of this course, and relied on one promise: looking up a key takes roughly the same time however many keys there are (Data Structures in Practice 1 and 3). This chapter builds that machinery yourself, so the promise stops being magic. A **hash table** is a list of slots plus a *hash function* that turns a key into a number; the number, reduced to the table's size, says which slot to look in.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every snippet below was run and the output shown is what it printed. No timings are quoted, because they differ between machines. Slot numbers, counts and chain lengths are exact, because they depend only on the data and on the hash function written here (Python's own string hashes are deliberately randomised between runs, so this chapter uses its own).

The Hash Function

A good hash function is **deterministic** (the same key always gives the same number), **fast**, and **spreads keys out**, so that different keys rarely land in the same slot. The usual recipe for text multiplies a running total by a small prime before adding each character, so that the position of each letter matters. The final number is reduced with `%` (modulo) to a slot index.

```
def simple_hash(text):
    h = 0
    for ch in text:
        h = (h * 31 + ord(ch)) % 1_000_003
    return h

for w in ["cat", "act", "tac", "apple"]:
    print(w, simple_hash(w), simple_hash(w) % 8)

def bad_hash(text):
    return sum(ord(c) for c in text)       # order of letters is ignored
print(bad_hash("cat"), bad_hash("act"), bad_hash("tac"))
```

```
cat 98262 6
act 96402 2
tac 114582 6
apple 28931 3
312 312 312
```

`cat` and `tac` both land in slot 6 even though they are different words. The poor function that merely adds up letter codes cannot tell the three anagrams apart at all: they all hash to 312. Which slot a key gets does not matter; what matters is that the keys use *all* the slots evenly.

Collisions and Chaining

Two different keys with the same slot is a **collision**. It is unavoidable: there are far more possible keys than slots, so a table must have a plan for it. The first plan is **separate chaining**: each slot (called a bucket) holds a small list of the `(key, value)` pairs that landed there. To look up a key, hash it, go to its bucket, and compare keys one by one along that short list.

```
class ChainedTable:
    def __init__(self, size=8):
        self.buckets = [[] for _ in range(size)]
        self.count = 0

    def _bucket(self, key):
        return self.buckets[simple_hash(key) % len(self.buckets)]

    def put(self, key, value):
        bucket = self._bucket(key)
        for i, (k, v) in enumerate(bucket):
            if k == key:
                bucket[i] = (key, value)     # replace an existing key
                return
        bucket.append((key, value))
        self.count += 1
        if self.count / len(self.buckets) > 0.75:
            self._grow()

    def get(self, key, default=None):
```

```
        for k, v in self._bucket(key):
            if k == key:
                return v
        return default

    def delete(self, key):
        bucket = self._bucket(key)
        for i, (k, v) in enumerate(bucket):
            if k == key:
                del bucket[i]
                self.count -= 1
                return True
        return False

    def _grow(self):
        old = self.buckets
        self.buckets = [[] for _ in range(len(old) * 2)]
        self.count = 0
        for bucket in old:
            for k, v in bucket:
                self.put(k, v)              # every key is re-hashed

    def __len__(self):
        return self.count

    def load_factor(self):
        return self.count / len(self.buckets)

    def longest_chain(self):
        return max(len(b) for b in self.buckets)
```

```
t = ChainedTable()
for i, w in enumerate(["cat", "act", "tac", "apple", "pear", "plum", "fig", "kiwi"]):
    t.put(w, i)
    print(w, len(t), len(t.buckets), round(t.load_factor(), 2))
print(t.get("fig"), t.get("nope"), t.get("nope", -1))
t.put("fig", 99)
print(t.get("fig"), len(t))
print(t.delete("fig"), t.delete("fig"), len(t))
print(t.longest_chain())
```

```
cat 1 8 0.12
act 2 8 0.25
tac 3 8 0.38
apple 4 8 0.5
pear 5 8 0.62
plum 6 8 0.75
fig 7 16 0.44
kiwi 8 16 0.5
6 None -1
99 8
True False 7
2
```

Read the output as a story. The table starts with 8 buckets. Adding the seventh key pushed the **load factor** (keys divided by buckets) to 7/8 = 0.88, past the 0.75 limit, so the table *doubled* to 16 buckets and re-hashed every key, which is why the load fell back to 0.44. That re-hash is needed because the slot is `hash % size`: when the size changes, nearly every key belongs somewhere new. It is the same trade as the list from Data Structures in Practice 2: an occasional expensive step buys a cheap average one, which is why adding keys is *amortised* constant time.

> **WHY THE LOAD FACTOR MATTERS**
>
> A bucket's average chain length is exactly the load factor. Keep the load factor below a fixed limit and chains stay short on average, so lookups cost a constant amount of work. Let it grow without bound and a lookup slides toward scanning a list, which is the linear cost you started the course trying to avoid.

A Bad Hash Function Ruins Everything

The table is only as good as its spreading. The next snippet hashes 2,000 distinct random six-letter words into 256 buckets with each function, and reports how many buckets were used and the longest chain. (It needs `simple_hash` and `bad_hash` from the first snippet.)

```
import random
random.seed(1)
```

```
words = ["".join(random.choice("abcdefghij") for _ in range(6)) for _ in range(2000)]
words = list(dict.fromkeys(words))               # drop repeats, keep order
print(len(words))

for name, fn in [("good", simple_hash), ("bad", bad_hash)]:
    counts = {}
    for w in words:
        b = fn(w) % 256                          # 256 buckets
        counts[b] = counts.get(b, 0) + 1
    print(name, len(counts), max(counts.values()))
```

```
2000
good 256 17
bad 44 119
```

The multiplying function used every one of the 256 buckets and its fullest bucket held 17 words, against an average of about 8. The letter-sum function used only 44 buckets and piled 119 words into one. Sums of six small letter codes all fall in a narrow band, so most buckets can never be chosen. With chains that long, a lookup is back to scanning a list: the table has the right shape but none of the speed.

Open Addressing

The second plan stores everything in the one flat list, with no inner lists. If a key's slot is taken, step to the next slot (**linear probing**), wrapping round at the end, until a free one is found. A lookup follows the same steps and stops at the first *never-used* slot, because the key cannot be any further along. This layout is friendly to memory caches and is the idea behind Python's own dict, though CPython uses a more elaborate probe order.

Deleting is the tricky part. If you simply emptied a slot, a lookup for a key that had been pushed past it would stop at the new hole and wrongly conclude the key is absent. The fix is a **tombstone**: mark the slot "deleted", so searches walk through it, while inserts may reuse it. Tombstones count towards how full the table is, so the code below tracks `used` (live plus tombstones) as well as `count` (live only).

```
EMPTY = object()                          # never used
DELETED = object()                        # used once, then removed (a tombstone)

class ProbeTable:
    def __init__(self, size=8):
        self.keys = [EMPTY] * size
        self.vals = [None] * size
        self.used = 0                     # live slots plus tombstones
        self.count = 0                    # live slots only

    def _find(self, key):
        n = len(self.keys)
        i = simple_hash(key) % n
        first_deleted = None
        for _ in range(n):
            k = self.keys[i]
            if k is EMPTY:                # the search ends at a never-used slot
                return (first_deleted if first_deleted is not None else i), False
            if k is DELETED:
                if first_deleted is None:
                    first_deleted = i     # a slot we may reuse for an insert
            elif k == key:
                return i, True
            i = (i + 1) % n               # linear probing: try the next slot
        return first_deleted, False

    def put(self, key, value):
        if (self.used + 1) / len(self.keys) > 0.6:
            self._grow()
        i, found = self._find(key)
        if not found:
            if self.keys[i] is EMPTY:
                self.used += 1
            self.count += 1
        self.keys[i] = key
        self.vals[i] = value

    def get(self, key, default=None):
        i, found = self._find(key)
        return self.vals[i] if found else default

    def delete(self, key):
        i, found = self._find(key)
```

```
        if not found:
            return False
        self.keys[i] = DELETED
        self.vals[i] = None
        self.count -= 1
        return True

    def _grow(self):
        old = list(zip(self.keys, self.vals))
        n = len(self.keys) * 2
        self.keys = [EMPTY] * n
        self.vals = [None] * n
        self.used = self.count = 0
        for k, v in old:
            if k is not EMPTY and k is not DELETED:
                self.put(k, v)
```

```
p = ProbeTable()
for w in ["cat", "act", "tac", "apple"]:
    i, _ = p._find(w)
    print(w, simple_hash(w) % 8, "stored at", i)
    p.put(w, 1)
print(p.delete("act"), p.get("tac"), p.get("act"), p.count, p.used)
p.put("act", 5)
print(p.get("act"), p.count, p.used)
```

```
cat 6 stored at 6
act 2 stored at 2
tac 6 stored at 7
apple 3 stored at 3
True 1 None 3 4
5 4 4
```

`cat` and `tac` both want slot 6; `tac` was pushed on to slot 7. After `act` was deleted, `tac` was still found and the live count dropped from 4 to 3, while `used` stayed at 4 because of the

tombstone. Putting `act` back reused that slot, so nothing grew. Open addressing needs a lower load limit than chaining (0.6 here), because probe sequences lengthen sharply as the table fills. Exercise 3 measures that.

	SEPARATE CHAINING	OPEN ADDRESSING
Where collided keys go	Into a list inside the bucket	The next free slot in the same flat list
Load factor	Can exceed 1; chains just lengthen	Must stay below 1; keep it well under
Deleting	Remove from the bucket's list	Needs a tombstone
Memory	Extra list object per bucket	One flat array
Weak spot	Pointer-heavy; many small lists	Clustering as the table fills

What Makes a Key Safe to Use

The whole scheme relies on a contract, the one behind the hashing rules in Data Structures in Practice 3: **keys that are equal must have equal hashes**, and a key's hash must never change while it is in the table. If a key could change, it would sit in a bucket its new hash no longer points to, and could never be found again. This is why lists and dicts are not allowed as keys, while numbers, strings and tuples of them are.

```
print(hash(5), hash(5.0), hash((1, 2)) == hash((1, 2)))

class K:
    def __init__(self, v): self.v = v
    def __eq__(self, other): return self.v == other.v
    def __hash__(self): return hash(self.v)

d = {K(1): "a"}
print(d[K(1)])                # a different object, but equal and same hash

class Bad:
    def __eq__(self, other): return True    # defines == but no __hash__

try:
    hash(Bad())
except TypeError as e:
```

```
    print(e)
```

```
5 5 True
a
unhashable type: 'Bad'
```

Note that `5` and `5.0` hash alike, because they are equal and so must find the same entry. A class that defines `__eq__` without `__hash__` is made unhashable by Python for exactly this reason: equality was changed, so the old hash could no longer be trusted. When you define both, base the hash on exactly the fields that `__eq__` compares.

> **CONSTANT TIME IS AN AVERAGE, NOT A GUARANTEE**
>
> If many keys share a hash, every operation degrades to scanning a chain or probe run. Real tables defend against this with well-mixed hashes (and Python randomises string hashes per run so outsiders cannot craft colliding keys on purpose). The practical lesson is the one the bad hash showed: the speed of a dict comes from spreading, not from the word "hash".

What You Have Rebuilt

IDEA	IN YOUR HAND-BUILT TABLE	IN REAL PYTHON
Key to slot	`simple_hash(key) % size`	`hash(key)` reduced to the table size
Collisions	Chains, or probing to the next slot	Probing (a more elaborate sequence)
Growth	Double and re-hash at a set load factor	Resizes automatically as it fills
Deletion	Remove from a chain, or a tombstone	A similar dummy-marker idea
Key rules	Equal keys, equal hashes; no mutation	Exactly the same, enforced by `__hash__`

You would never use a hand-built table in production Python: the built-in is written in C and heavily tuned. The value is understanding what it costs and why, which tells you when a dict is the right tool, why a list of keys is not, and what to look at when a lookup is unexpectedly slow.

Challenges

EXERCISE 1

Build a `HashSet` class using separate chaining and the chapter's `simple_hash`. Give it `add` (returning `True` if the item was new, `False` if already present), `in` support, `remove` (raising `KeyError` if absent), `len` and iteration. Start with 4 buckets and double whenever the item count exceeds twice the bucket count. Test duplicates, growth past the limit, removal, and removing a missing item.

View solution

EXERCISE 2

Make a chained table behave like a real dict by writing a `MyDict` class with `d[key] = value`, `d[key]` (raising `KeyError` when missing), `del d[key]`, `in`, `len`, `get`, `items` and iteration over keys. Then use it to count the words in a sentence, and confirm your counts equal those from a real `dict` doing the same job.

View solution

EXERCISE 3

Write a small open-addressing set of keys with linear probing in a fixed 16-slot table, with a switch that chooses between tombstones and simply emptying a slot on removal. First add several keys that share a starting slot, remove the first, and show that the naive version loses a key that is still present while the tombstone version does not. Then, in a 32-slot table, add 8, 16, 24 and 28 keys and print the load factor, the longest probe run and the average probe count to find each key, to see how probing cost grows as the table fills.

View solution

QUICK REFERENCE

- **Hash table:** a list of slots plus a hash function; slot = `hash(key) % size`
- **Good hash:** deterministic, fast, and spreads keys evenly over all slots
- **Collision:** two keys, one slot; unavoidable, so every table needs a plan

- **Chaining:** each bucket holds a short list of pairs
- **Open addressing:** one flat list; on a collision probe the next slot; delete with tombstones
- **Load factor:** keys divided by slots; keep it under a limit by doubling and re-hashing
- **Amortised:** an occasional full re-hash is repaid by many cheap inserts
- **Key contract:** equal keys have equal hashes, and a key's hash never changes inside a table
- **Weakness:** a poor hash turns lookups back into list scans

COMING NEXT

Data Structures in Practice 10 is the capstone: a decision guide for choosing a structure, and three worked case studies that use what the whole course has built.

Data Structures in Practice

Chapter 10 · Capstone: Choosing for Real Problems

Nine chapters have given you a toolbox: lists, tuples, sets, dicts, stacks, queues, linked lists, heaps, trees, graphs and hash tables. Real problems do not arrive labelled with the right one. This final chapter turns the toolbox into a habit: a short **decision guide** you can run in your head, then three worked case studies that each combine two or three structures from earlier chapters.

> **WHAT WAS RUN, AND WHAT WAS NOT**
>
> Every snippet was run and the output shown is exactly what it printed. The examples use small, fixed inputs and logical timestamps instead of the real clock, so the results are the same on every machine. No timings are quoted.

The Decision Guide

Do not start from the structure. Start from the **operations your program performs most often**, and ask what each must cost. Four questions settle most choices:

1. **How will items be found?** By position, by exact key, by range or prefix, or by "the smallest/largest one"?
2. **Where do items enter and leave?** Only at the ends, anywhere, or in priority order?
3. **Does order matter?** Insertion order, sorted order, or none at all?
4. **Are there relationships between items?** If items point at other items, you have a graph, whatever you call it.

YOU MOSTLY NEED TO...	REACH FOR	WHY
Keep items in order, read by position, append	`list`	Constant-time index and amortised append (Data Structures in Practice 2)
Look up by exact key	`dict`	Average constant-time lookup (Data Structures in Practice 9)
Ask "have I seen this?" or remove duplicates	`set`	Constant-time membership and set algebra (Data Structures in Practice 3)

YOU MOSTLY NEED TO...	REACH FOR	WHY
Take from one end, add to the other	`deque`	Constant-time at both ends; a list is slow at the front (Data Structures in Practice 4)
Undo, or walk back through history	list used as a stack	Last in, first out with `append` and `pop`
Always take the smallest or largest next	`heapq`	Logarithmic push and pop (Data Structures in Practice 6)
Search sorted data, or find by range or prefix	sorted list with `bisect`	Logarithmic search on sorted data (Data Structures in Practice 1)
Follow links, find routes, order dependencies	dict of neighbours (graph)	Adjacency lists with BFS or DFS (Data Structures in Practice 8)
Fixed records, usable as keys	`tuple`	Immutable and hashable (Data Structures in Practice 3)

TWO HABITS WORTH KEEPING

First, **start with the simplest structure that is correct** (usually a list or a dict) and change it only when you can name the operation that is too slow and how many items it runs on. Ten items never need a heap. Second, **combine structures**: most real designs are a dict for finding things plus something else for ordering them. The three case studies are all of that kind.

Case Study 1: A Rate Limiter

The problem. Allow each user at most 3 requests in any 10-unit window, and refuse the rest. **The operations:** find a user's recent requests by key (a dict), add a request at the newest end, and discard requests that have aged out from the oldest end. Items leave from the opposite end to where they enter, which is exactly a queue, so each user gets a `deque` of timestamps held in a dict.

```
from collections import deque, defaultdict

class RateLimiter:
    def __init__(self, limit, window):
        self.limit = limit
```

```
        self.window = window
        self.hits = defaultdict(deque)       # user -> timestamps, oldest first

    def allow(self, user, now):
        q = self.hits[user]
        while q and q[0] <= now - self.window:
            q.popleft()                      # forget requests outside the window
        if len(q) >= self.limit:
            return False
        q.append(now)
        return True

rl = RateLimiter(limit=3, window=10)
for t in [0, 1, 2, 3, 9, 10, 11, 12, 13]:
    print(t, rl.allow("ann", t))
print(rl.allow("bob", 3), len(rl.hits))
```

```
0 True
1 True
2 True
3 False
9 False
10 True
11 True
12 True
13 False
True 2
```

Requests at 0, 1 and 2 fill the allowance; 3 and 9 are refused, and the refused ones are *not* recorded. At time 10 the request from time 0 has aged out (`0 <= 10 - 10`), freeing one place, then 11 and 12 free the next two. Bob has his own deque, so Ann's traffic never affects him. Each timestamp is added once and removed once, so the cost per request is constant on average, however many requests a user has made in total. A list would work too, but `pop(0)` shifts every item (Data Structures in Practice 2).

Case Study 2: A Task Scheduler with Cancellation

The problem. Run tasks lowest priority number first, breaking ties by arrival order, and allow a task to be cancelled by name before it runs. A heap gives the next task cheaply, but a heap cannot remove an arbitrary item cheaply. The standard answer is **lazy deletion**: keep a dict from name to heap entry, and on cancellation just blank the entry's name. The heap discards blanked entries when they surface. A counter in the middle of each entry preserves arrival order for ties and stops Python ever comparing the names.

```
import heapq, itertools

class Scheduler:
    def __init__(self):
        self.heap = []
        self.live = {}                      # name -> its entry in the heap
        self.counter = itertools.count()    # tie-breaker: arrival order

    def add(self, name, priority):
        entry = [priority, next(self.counter), name]
        self.live[name] = entry
        heapq.heappush(self.heap, entry)

    def cancel(self, name):
        entry = self.live.pop(name, None)
        if entry is None:
            return False
        entry[2] = None                     # leave it in the heap, marked dead
        return True

    def pop(self):
        while self.heap:
            priority, _, name = heapq.heappop(self.heap)
            if name is not None:
                del self.live[name]
                return name
        return None

s = Scheduler()
for name, pr in [("email", 3), ("backup", 5), ("alert", 1), ("report", 3), ("cleanup", 4
    s.add(name, pr)
print(s.cancel("backup"), s.cancel("backup"), s.cancel("nothing"))
out = []
while True:
    n = s.pop()
```

```
    if n is None:
        break
    out.append(n)
print(out)
```

```
True False False
['alert', 'email', 'report', 'cleanup']
```

The first cancellation succeeds; cancelling again, or cancelling a name that never existed, reports `False`. The run order is by priority, with `email` ahead of `report` because it arrived first, and the cancelled backup never appears. The dead entry stayed in the heap until it reached the top and was skipped, so cancellation is constant time and each entry is still pushed and popped once. The price is a little wasted memory until dead entries surface, a trade worth knowing: sometimes the cheap fix is to delay the clean-up rather than do it immediately.

Case Study 3: Shortest Route

The problem. Given a set of two-way roads between named places, find a route with the fewest hops between two of them, or report that none exists. The relationships between places are the point, so this is a graph (Data Structures in Practice 8): a dict mapping each place to the set of its neighbours. Breadth-first search explores in rings of increasing distance, so the first time it reaches the goal it has used the fewest hops. A queue holds the frontier, and a dict doubles as the "already visited" set *and* the record of where each place was reached from, which lets the route be rebuilt backwards.

```
from collections import deque, defaultdict

roads = [("A", "B"), ("B", "C"), ("A", "D"), ("D", "E"),
         ("E", "C"), ("C", "F"), ("G", "H")]
graph = defaultdict(set)
for a, b in roads:
    graph[a].add(b)
    graph[b].add(a)                          # two-way roads
```

```
def shortest_route(start, goal):
    prev = {start: None}                # visited places, and how we got there
    queue = deque([start])
    while queue:
        node = queue.popleft()
        if node == goal:
            path = []
            while node is not None:
                path.append(node)
                node = prev[node]
            return path[::-1]
        for nxt in sorted(graph[node]):
            if nxt not in prev:
                prev[nxt] = node
                queue.append(nxt)
    return None

print(shortest_route("A", "F"))
print(shortest_route("A", "A"))
print(shortest_route("A", "H"))
```

```
['A', 'B', 'C', 'F']
['A']
None
```

There are two ways from `A` to `C`: through `B` (two hops) or through `D` and `E` (three). The search finds the shorter one first. Asking for a route to the place you are already at returns just that place, and asking for a route to an unconnected pair of places (`G` and `H` are an island) returns `None` after exploring everything reachable, visiting each place and road once. Without the `prev` check, the loop `A, B, C, E, D, A` would run forever; a visited record is what makes graph searches terminate.

One More Pattern: Prefix Search

A short coda showing the guide's "range or prefix" row. A hash table finds only *exact* keys, so it cannot answer "everything starting with `p`". Sorted data can, because all matches sit together, and `bisect` finds both ends of that block in logarithmic time.

```
import bisect

words = ["pear", "apple", "fig", "kiwi", "plum", "lime", "date", "cherry"]
index = sorted(words)

def starts_with(prefix):
    lo = bisect.bisect_left(index, prefix)
    hi = bisect.bisect_left(index, prefix + chr(0x10FFFF))   # just past every match
    return index[lo:hi]

print(starts_with("p"), starts_with("ch"), starts_with("z"))
```

```
['pear', 'plum'] ['cherry'] []
```

The price is that the list must stay sorted: inserting a new word costs a linear shift. A read-heavy index suits this; a constantly changing one may want a different design. Every choice in this chapter is a trade of this kind.

> **MISTAKES THE CASE STUDIES AVOID**
>
> Using a list as a queue (slow at the front); keeping a graph with no visited record (endless loops); deleting from the middle of a heap (not supported, so mark and skip instead); and comparing entries whose later fields cannot be compared (the counter in the scheduler guarantees ties never reach the name).

What the Course Gave You

You can now count the work a piece of code does, read the cost off the structure rather than guess, and build the structure from first principles when you need to understand it. The rule that

runs through all ten chapters is the same: **name the operations, find the structure that makes them cheap, and prefer the simplest one that does**.

Challenges

EXERCISE 1

Build an `LRUCache` with a fixed capacity: `get` returns the stored value (or a default) and counts as a use, `put` adds or updates and counts as a use, and when the cache is over capacity the least recently used key is dropped. Both operations must take constant time. Show the key order after a sequence of gets and puts that includes a read, an overflow and an update of an existing key.

View solution

EXERCISE 2

A leaderboard receives thousands of `(score, name)` results and only ever needs the best *k*. Write a `TopK` class that never holds more than *k* items, using a heap. Test it on a small hand-checked list including a tie, then on 5,000 random results, confirming the answer equals both a full sort and `heapq.nlargest`.

View solution

EXERCISE 3

Write `build_order(tasks)`, where `tasks` maps each task to the list of tasks that must finish first, returning a valid order to run them in. Use a dict of waiting counts and a queue of ready tasks. Raise `ValueError` naming the tasks involved if there is a cycle. Test a task chain with a shared step, a single dependency that is not itself listed as a task, an empty input, and a loop.

View solution

QUICK REFERENCE

- **Start from operations:** how items are found, where they enter and leave, whether order matters, whether items relate to each other

- **Simplest first:** a list or dict until a named operation on a large input is too slow
- **Queue-shaped traffic:** a `deque`, one per key in a dict (rate limiter)
- **Always the next best:** a heap; cancel with lazy deletion and use a counter for ties
- **Relationships:** a dict of neighbour sets; BFS for fewest hops; always record visited places
- **Exact keys:** hash table. **Ranges and prefixes:** sorted list with `bisect`
- **Combine structures:** a dict to find things plus another structure to order them

COURSE COMPLETE

That is all ten chapters of Data Structures in Practice. Well done: the best next step is to take a program you have already written and ask of each loop which operation dominates and whether a different structure would make it cheaper.