CS1010X: Programming Methodology I
Chapter 10: Data Structures and Multiple Representations

Lim Dillion
dillionlim@u.nus.edu

2025/26 Special Term 2
National University of Singapore

Chapter 10: Data Structures and Multiple Representations

Learning Objectives

By the end of this chapter, students should be able to:

- **Explain** the difference between *functional abstraction* (hiding process details) and *data abstraction* (hiding representation details), and **justify** why both are required in non-trivial programs.
- **Decompose** a program into (i) a *process layer* (control flow, algorithms) and (ii) a *data layer* (state representation, operations), using Nim as a case study.
- **Specify** an Abstract Data Type (ADT) via a clear contract: overview, invariants/-constraints, examples, and required operations.
- **Design** an ADT interface using **constructors**, **selectors**, **predicates**, and **printers/operations**.
- **Apply** the *abstraction barrier* principle to keep client code independent of internal representation, and **identify** barrier violations.
- **Implement** a **stack** ADT and **use** it to store history for undo, including correct handling of edge cases (e.g. empty history).
- **Explain** why multiple representations co-exist in real software projects and **compare** strategies to manage them (conversion layers, tagging, dispatch).
- **Build** a **complex-arithmetic package** with a stable interface, and **implement** it using both rectangular and polar representations without changing client arithmetic code.
- **Use** Python's `math` library (`hypot`, `sin`, `cos`, `atan`, `atan2`) correctly and **state** that angles are measured in **radians**.
- **Implement** the **set** ADT using three representations (unordered lists, ordered lists, binary search trees) and **analyse** time/space trade-offs.
- **Use** Python's built-in `set` type and **relate** it to the set ADT, including key constraints (hashability) and typical performance expectations (average-case).

10.1 From Functional Abstraction to Data Abstraction

In earlier chapters, we relied heavily on **functional abstraction**. A function:

- hides irrelevant details of *how* something is computed,
- exposes only an interface (parameters $\rightarrow$ return value),
- supports reuse and compositional reasoning (build big programs from small pieces).

However, most realistic programs are not just computations over isolated numbers and strings. They manage **state** and **structured information** (records, collections, histories, graphs, etc.). To write such programs cleanly, we need a comparable abstraction tool for **data**.

Abstraction Type	Hides	What does it do?
Functional abstraction	Internal steps of a computation (algorithmic details)	"How do we compute this result?"
Data abstraction	Internal representation of an object (storage details)	"How should this object be represented and manipulated?"

Table 10.1: Functional abstraction vs. data abstraction.

10.1.1 Why do we need both?

Consider a program that simulates a game, processes images, or manages student records. In each case we have:

- **Process concerns:** control flow, algorithms, iteration/recursion, input/output.
- **Data concerns:** what the objects are, what properties they have, how to store and update them.

If we mix these concerns freely, the program becomes:

- hard to extend (a small change breaks many parts),
- hard to optimise (representation changes require rewriting logic),
- hard to maintain (many places depend on the same low-level details).

The goal of this chapter is to show how **good data abstraction**:

- improves modularity,
- enables multiple representations,
- supports new features (like undo),
- and makes large programs manageable.

10.2 Case Study: The Game of Nim

Nim is a common game used in game theory, and its rules are:

- Two players take turns.
- The board consists of piles of coins.
- On your turn, remove any number of coins from exactly one pile.
- The player who takes the last coin wins.

Writing Nim cleanly requires designing:

1. **Functional (process) design:** the game loop.
2. **Data design:** the game state and history structures.

Functional Design. At a high level, we need a loop (or recursion) until the game ends. This is usually known as the game / event loop:

- Human makes a move.
- Update and display state.
- Check game over.
- Computer makes a move.
- Update and display state.
- Check game over.

Data Design. We must store:

- **current state:** number of coins in each pile,
- **history:** earlier states (for undo / replay / debugging),
- **(optional) move record:** which pile and how many were removed.

A common mistake is to jump directly into the game loop code and "figure out data later". It is often better to decide the representation of data, since it will have significant implications on how the process is designed. Therefore, to accurately design a data structure to contain the game data, we must:

1. Decide what information needs to be represented (state, history).
2. Write an interface (constructors/selectors/operations) for that representation.
3. Only then write the control flow (game loop) using the interface.

This keeps the process code clean and resilient to representation changes.

10.3 Data Abstraction for Nim Game State

10.3.1 Game state as an ADT

Conceptually, a Nim game state for two piles is the mathematical object:

$$(\text{pile}_1, \text{pile}_2)$$

with operations such as:

- create an initial state,
- read the size of a pile,
- remove coins from a chosen pile (producing a new state),
- compute total coins, check if the game is over,
- display the state.

We define a **game state ADT** interface:

- **Constructor:** `make_game_state(n, m)`
- **Selectors:** `size_of_pile(state, p)` for pile p
- **Operation:** `remove_coins_from_pile(state, k, p)`

The client code should only use these functions. It does not need to know the internal representation.

10.3.2 Representation 1: Encode as an integer

We can store (n, m) as $10n + m$. Thus, we can write the above operations as:

```
def make_game_state(n, m):
    return 10*n + m

def size_of_pile(state, pile_number):
    if pile_number == 1:
        return state // 10
    else:
        return state % 10

def remove_coins_from_pile(state, num_coins, pile_number):
    if pile_number == 1:
        return state - 10*num_coins
    else:
        return state - num_coins
```

Hidden assumptions. However, there are some hidden assumptions with our representation above, that are not obvious from the interface:

- It naturally supports only **two** piles (it is not straightforward to extend this to n piles).
- It breaks once pile sizes exceed 9 if we insist on base-10 decoding.

This is a classic reason to prefer clearer representations when future changes are likely.

10.3.3 Representation 2: Encode as a tuple

```python
def make_game_state(n, m):
    return (n, m)

def size_of_pile(state, p):
    return state[p-1]

def remove_coins_from_pile(state, num_coins, p):
    n, m = state
    if p == 1:
        return make_game_state(n - num_coins, m)
    else:
        return make_game_state(n, m - num_coins)
```

Advantages. This structure has a few advantages over the previous representation:

- The structure is explicit: a state is (n, m).
- Extending to more piles is natural: use a tuple/list of length k.
- The representation communicates fewer hidden constraints.

10.3.4 Generalising to k piles

If we want k piles, the ADT stays the same conceptually, but the constructor and selector become:

- `make_game_state(piles)` where `piles` is a list/tuple of pile sizes,
- `size_of_pile(state, p)` returns element $p - 1$.

Notice that the tuple representation is more flexible than the integer representation of piles.

Invariants should be stated

If your representation requires constraints, state them explicitly. For Nim state, typical invariants are:

- pile sizes are integers,
- pile sizes are non-negative,
- pile index p is valid (between 1 and number of piles).

10.3.5 ADT Support Operations for the Game Loop

The game loop depends on operations that:

- **query** the current game state (e.g. total coins, game-over),
- **present** the state (printer),
- **validate** moves (to keep rules checks out of the loop).

These are part of the **data abstraction layer** because they are properties of the *game state object*, and should remain correct even if the internal representation changes. Therefore, each function below should only use the ADT interface (selectors/operations), so the game loop remains representation-independent, and does not break if the representation changes.

To support both 2-pile and k-pile versions cleanly, we include a new selector:

```
def num_piles(state):
    """Returns the number of piles in this game state."""
    return 2
```

If you later generalise the representation to a tuple/list of length k, then `num_piles` becomes the only function that needs to know that detail.

We can also write some derived queries for the total size and game over states, as well as a way to display the game state.

```
def total_size(state):
    """Total number of coins across all piles."""
    total = 0
    for p in range(1, num_piles(state) + 1):
        total += size_of_pile(state, p)
    return total
def is_game_over(state):
    """True iff there are no coins left."""
    return total_size(state) == 0
def display_game_state(state):
    """Print the sizes of all piles (1-indexed)."""
    print()
    for p in range(1, num_piles(state) + 1):
        print("Pile " + str(p) + ": " + str(size_of_pile(state, p)))
    print()
```

These functions keep rule checks out of the game loop and out of the move policies:

```
def is_valid_pile(state, p):
    return 1 <= p <= num_piles(state)
def is_valid_move(state, num_coins, p):
    """A move is valid if it removes 1..size coins from valid pile."""
    if not is_valid_pile(state, p):
        return False
    if num_coins <= 0:
        return False
    return num_coins <= size_of_pile(state, p)
```

10.4 Functional Abstraction for Nim

Now that we have a clean game-state ADT interface, we can implement the game loop as **pure process code**.

10.4.1 Operations in the Process Layer: Human/Computer Moves

The functions `human_move` and `computer_move` are **process-level policies**:

- `human_move` includes user interaction (I/O) and validation,
- `computer_move` encodes a strategy (AI policy).

They should **not** depend on representation details; they should call only ADT operations.

We can get input by using:

```
def prompt(msg):
    return input(msg)
```

Then, we can implement the human move function:

```
def human_move(state):
    """Ask the human for a move and return the updated state."""
    while True:
        p = int(prompt("Which pile will you remove from? "))
        n = int(prompt("How many coins do you want to remove? "))
        if is_valid_move(state, n, p):
            return remove_coins_from_pile(state, n, p)
        print("Invalid move. Try again.")
```

The computer uses a simple policy: pick the first non-empty pile and remove 1 coin.

```
def computer_move(state):
    """A very simple strategy: remove 1 coin from the first non-empty
       pile."""
    for p in range(1, num_piles(state) + 1):
        if size_of_pile(state, p) > 0:
            print("Computer removes 1 coin from pile " + str(p))
            return remove_coins_from_pile(state, 1, p)
    # Should not happen if is_game_over is checked correctly.
    return state
```

10.4.2 Announcing the Winner

In our `play(state, player)` design, `player` is the *next* player to move. If the game is already over at the start of a turn, the next player loses.

```
def announce_winner(player):
    if player == "human":
        print("You lose. Better luck next time.")
    else:
        print("You win. Congratulations.")
```

10.4.3 Core Game Loop

Now, the loop should rely on operations that we have previously implemented, such as:

- `display_game_state(state)`
- `is_game_over(state)`
- `human_move(state)` (returns updated state)
- `computer_move(state)` (returns updated state)
- `announce_winner(player)`

If these services respect the abstraction barrier, the loop never needs to know how state is stored. As a result, using our data abstraction above, we can write the game loop either recursively or iteratively.

```
def play(state, player):
    display_game_state(state)
    if is_game_over(state):
        announce_winner(player)
    elif player == "human":
        play(human_move(state), "computer")
    elif player == "computer":
        play(computer_move(state), "human")
    else:
        print("Unknown player:", player)
```

```
def play_iterative(state):
    player = "human"
    while True:
        display_game_state(state)
        if is_game_over(state):
            announce_winner(player)
            break
        if player == "human":
            state = human_move(state)
            player = "computer"
        else:
            state = computer_move(state)
            player = "human"
```

Abstraction Barriers

The game loop is an example of **functional abstraction**: it should operate only on the *interface* of the game state (and related ADTs like stacks). If we change the representation of the game state (e.g. from tuples to lists), the loop should not change.

10.5 Extensibility of Data Structures

We shall see how good design allows for easy extension when requirements change. Suppose we wanted to support an undo operation, which lets a **human player** to undo their last move. Undo requires remembering previous states, so we need to modify our data representation.

10.5.1 History design

We maintain a history of game states such that:

- Before each human move, record the current state.
- When undo is requested:
  - pop the most recent state from history,
  - make it the current state again,
  - keep it the human's turn.

The natural structure is a **stack**, since we want to easily access the **previous** state.

10.5.2 Stack ADT: Specification

A stack, as covered in the previous chapter, is a **Last-In-First-Out** (LIFO) collection. We noted that we can directly use lists as stacks, but we shall write a more explicit stack ADT here.

- `make_stack()` returns a new empty stack.
- `push(s, item)` adds `item` to stack `s`.
- `pop(s)` removes and returns the most recently pushed item; returns `None` if empty.
- `is_empty(s)` returns `True` iff empty.

10.5.3 Stack ADT: Implementation

We implement the stack ADT using a list as the underlying representation. The abstraction barrier principle still applies: client code should only use `make_stack`, `push`, `pop`, and `is_empty`.

```
def make_stack():
    """Return a new empty stack."""
    return []  # internal representation: Python list
def is_empty(s):
    """Return True iff stack s is empty."""
    return len(s) == 0
def push(s, item):
    """Push item onto stack s (mutates s)."""
    s.append(item)
    # Return value is optional; we return s for convenience.
    return s
def pop(s):
    """Pop and return the top item of stack s (mutates s).
    Return None if s is empty."""
    if is_empty(s):
        return None
    return s.pop()
```

10.5.4 Using stack for Nim undo

Now that we have the stack ADT, we can use it to implement the undo functionality. The human player enters 0 to indicate undo. Notice that other than the functions below, we do not need to modify any other functions.

```
game_stack = make_stack()

def human_move(state):
    p = prompt("Which pile will you remove from? (0 for undo)")
    if int(p) == 0:
        return handle_undo(state)

    n = prompt("How many coins do you want to remove?")
    push(game_stack, state)  # save current state before changing it
    return remove_coins_from_pile(state, int(n), int(p))

def handle_undo(state):
    old_state = pop(game_stack)
    if old_state is not None:
        print("Undo!")
        display_game_state(old_state)
        return old_state   # human's turn continues in the loop
    else:
        print("No more previous moves!")
        return state
```

10.6 Data Structures: Specification vs Implementation

Having seen the above case study, we find that a good data structure design separates:

1. **Specification (contract):** what it represents, what operations it provides, and what guarantees hold.
2. **Implementation (representation):** how it is stored and how operations are executed efficiently.

A good specification typically contains:

- **Overview:** conceptual meaning of the structure.
- **Invariants / constraints:** what must always be true.
- **Interface:** list of required operations.
- **Contracts:** algebraic laws / expected relationships.
- **Examples:** concrete sample inputs and outputs.

On the other hand, implementation typically requires:

- choosing a representation (there are often multiple choices),
- implementing constructors/selectors/predicates/operations,
- ensuring all contracts and invariants are satisfied.

10.7 Multiple Representations in Real Software

A good data structure design not only allows for extensibility with changing requirements, but the ability to handle multiple representations of data. For more complex data types, **multiple representations are possible**. A set, for example, can be represented as:

- unordered list without duplicates,
- ordered list without duplicates,
- binary search tree without duplicates,
- (in Python) a hash table (built-in `set`).

Multiple representations can co-exist in real software since:

- Large projects live a long time; requirements and performance needs change.
- Different operations need different strengths (fast membership vs fast union).
- Teams may work independently and pick different representations.

Therefore, in order to manage multiple representations, we make use of:

1. **Abstraction barrier:** insist all clients use the interface, not representation details.
2. **Conversion layer:** convert between representations at module boundaries (e.g. tree $\leftrightarrow$ sorted list).
3. **Tagged data + dispatch (next chapter):** store a tag indicating representation type and choose the appropriate implementation at runtime.

The key engineering trade-off

Multiple representations can improve performance and flexibility, but increase complexity:

- more code paths,
- more invariants to maintain,
- more testing required.

Abstraction barriers are what keep this complexity manageable.

10.8 Case Study: Complex-Arithmetic Package

Complex numbers have two common mathematical views:

- **Rectangular form:** $z = x + iy$ (store the Cartesian coordinates (x, y))
- **Polar form:** $z = re^{iA}$ (store the magnitude-angle pair (r, A))

In real software, different teams might prefer different representations:

- Addition/subtraction are simplest in rectangular form.
- Multiplication/division are simplest in polar form.

Our goal is to write *client arithmetic code* (add/mul/etc.) that works **without knowing which representation is used**. This is exactly what **data abstraction** and the **abstraction barrier** are for.

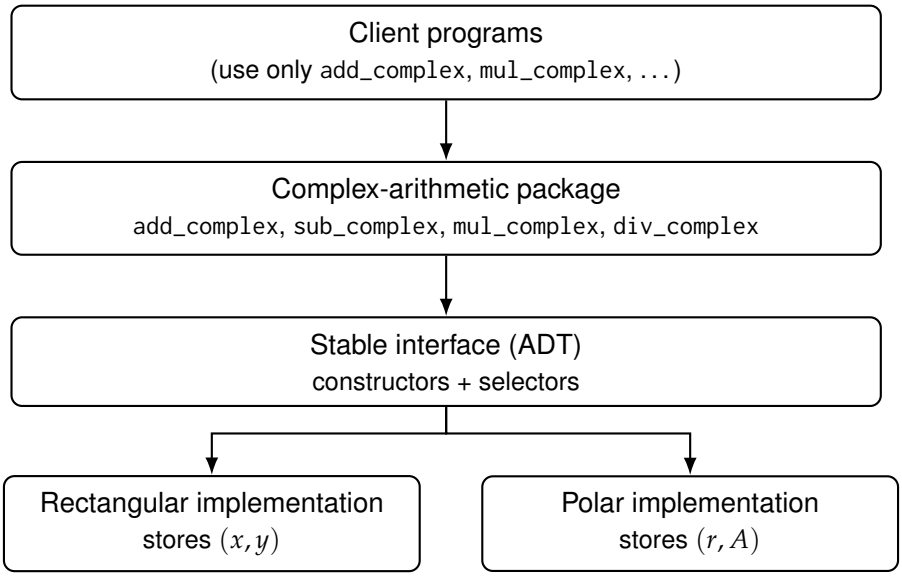


Figure 10.1: The abstraction barrier: arithmetic (clients) depend only on the interface, not on the representation.

What the barrier buys you

- You can **swap implementations** (rectangular vs. polar) without changing arithmetic code.
- You can **optimize** one layer (e.g. faster `magnitude`) without breaking users.
- You can evolve your system: as requirements change, you can change the representation safely.

10.8.1 Python `math` Library

In order to use the `math` library, you need to do `import math`. Useful functions are:

- `math.hypot(x, y)` returns $\sqrt{x^2 + y^2}$.
- `math.sin(a)`, `math.cos(a)` for $\sin a$ and $\cos a$ respectively.
- `math.atan(t)` computes $\arctan(t)$; `math.atan2(y, x)` computes angle of vector (x, y).

Prefer `atan2` over `atan(y/x)`

`atan(y/x)` fails when $x = 0$ and can lose quadrant information. `atan2(y,x)` is the standard robust choice.

10.8.2 Geometry of the two views (rectangular vs. polar)

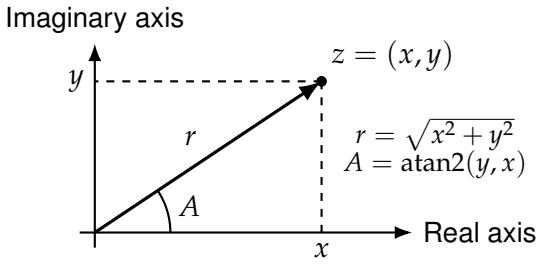


Figure 10.2: Rectangular coordinates (x, y) and polar coordinates (r, A) describe same point.

Angles and units

Python trigonometric functions use **radians**, not degrees. Use `math.radians(deg)` and `math.degrees(rad)` to convert.

10.8.3 ADT contract for Complex Arithmetic

We commit to the following interface, regardless of representation:

- **Constructors:**
  - `make_from_real_imag(x, y)`
  - `make_from_mag_ang(r, a)`
- **Selectors:**
  - `real_part(z)`, `imag_part(z)`
  - `magnitude(z)`, `angle(z)`

Representation invariants

Your implementation should ensure reasonable invariants such as:

- `magnitude`$(z) \geq 0$,
- angles are kept in a standard range (e.g. $(-\pi, \pi]$).

These are *implementation choices*; clients should not rely on how you enforce them.

10.8.4 Client arithmetic code (uses only the interface)

All arithmetic is written *only* in terms of constructors and selectors.

```
def add_complex(z1, z2):
    """Return z1 + z2."""
    return make_from_real_imag(real_part(z1) + real_part(z2),
                               imag_part(z1) + imag_part(z2))

def sub_complex(z1, z2):
    """Return z1 - z2."""
    return make_from_real_imag(real_part(z1) - real_part(z2),
                               imag_part(z1) - imag_part(z2))

def mul_complex(z1, z2):
    """Return z1 * z2."""
    return make_from_mag_ang(magnitude(z1) * magnitude(z2),
                             angle(z1) + angle(z2))
```

Why these formulas?

- Addition in rectangular form: $(x_1 + iy_1) + (x_2 + iy_2) = (x_1 + x_2) + i(y_1 + y_2)$.
- Multiplication in polar form: $(r_1 e^{iA_1})(r_2 e^{iA_2}) = (r_1 r_2) e^{i(A_1 + A_2)}$.

So we deliberately choose the constructor that makes the operation simplest.

Abstraction barrier: what clients must *not* do

A client must not assume `z[0]` is the real part. That is only true for one possible representation and will break for the other.

10.9 Multiple Representations

10.9.1 Complex Implementation A: Rectangular

Rectangular representation stores (x, y) directly. That is, for $z = x + yi$, it is stored as

$$z \equiv (x, y)$$

```python
import math

def make_from_real_imag(x, y):
    """Construct a complex number from rectangular coordinates."""
    return (x, y)

def real_part(z):
    """Return the real part x."""
    return z[0]

def imag_part(z):
    """Return the imaginary part y."""
    return z[1]

def magnitude(z):
    """Return r = sqrt(x^2 + y^2)."""
    return math.hypot(real_part(z), imag_part(z))

def angle(z):
    """Return A = atan2(y, x)."""
    return math.atan2(imag_part(z), real_part(z))

def make_from_mag_ang(r, a):
    """Construct from polar coordinates by converting to (x, y)."""
    return (r * math.cos(a), r * math.sin(a))
```

Key trade-off

This representation makes `real_part` and `imag_part` constant-time and trivial, but `magnitude` and `angle` require computation.

10.9.2 Complex Implementation B: Polar

Polar representation stores (r, A) directly. That is, for $z = re^{iA}$, it is stored as

$$z \equiv (r, A)$$

Normalizing is not strictly required for correctness, but it keeps angles in a standard range. Therefore, we can write the code as:

```python
import math

def normalize_angle(a):
    """Return an equivalent angle in (-pi, pi]."""
    return math.atan2(math.sin(a), math.cos(a))

def make_from_mag_ang(r, a):
    """Construct a complex number from polar coordinates."""
    # Invariant: r >= 0 (if r < 0, we can flip the angle by pi)
    if r < 0:
        r = -r
        a = a + math.pi
    return (r, normalize_angle(a))

def magnitude(z):
    """Return r."""
    return z[0]

def angle(z):
    """Return A."""
    return z[1]

def real_part(z):
    """Return x = r cos(A)."""
    return magnitude(z) * math.cos(angle(z))

def imag_part(z):
    """Return y = r sin(A)."""
    return magnitude(z) * math.sin(angle(z))

def make_from_real_imag(x, y):
    """Construct from rectangular coordinates by converting to (r, A).
    """
    r = math.hypot(x, y)
    a = math.atan2(y, x)
    return make_from_mag_ang(r, a)
```

Key trade-off

This representation makes `magnitude` and `angle` constant-time and trivial, but `real_part` and `imag_part` require computation.

10.9.3 Key lesson (and how to test it)

Both implementations satisfy the *same interface*. Therefore, **arithmetic code does not change** when we swap representations.

For example, if we were to write the code below, regardless of which representation we have used, it will give us the same result.

```
# Choose one implementation by importing/binding its constructors/
    selectors,
# then run the same client code.

z1 = make_from_real_imag(3, 4)
z2 = make_from_mag_ang(2, 1.0)

z3 = add_complex(z1, z2)
z4 = mul_complex(z1, z2)

print(real_part(z3), imag_part(z3))
print(magnitude(z4), angle(z4))
```

What should remain stable?

- The *meaning* of the results should be stable (same mathematical complex number).
- The *exact tuple contents* are allowed to differ across representations.

10.10 Sets as an Abstract Data Type

10.10.1 Specification

A **set** is an **unordered** collection of objects **without duplicates**.

- $\{3,1,2\} = \{1,2,3\}$
- duplicates are not allowed (e.g. $\{3,3,1,2\}$ is not a valid set)

Therefore, a set needs the following interfaces:

- Constructors:
  - `make_set()`
  - `adjoin_set(x, S)`
- Predicates:
  - `is_empty_set(S)`
  - `is_element_of_set(x, S)`
- Operations:
  - `union_set(S, T)`
  - `intersection_set(S, T)`
- Printer:
  - `print_set(S)`

We have the following contracts that a set must follow. For any sets S, T and object x:

- `is_element_of_set(x, adjoin_set(x, S))` is `True`.
- Membership in union:

```
is_element_of_set(x, union_set(S, T))
# is equivalent to
is_element_of_set(x, S) or is_element_of_set(x, T)
```

- No object is a member of the empty set.

Adjoining Sets

The adjoin of two sets is their *union*.
For any two sets A and B, their adjoin / union, denoted by $A \cup B$, is the set containing all elements that belong to A, or to B, or to both. In set-builder notation, we write:

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$$

10.10.2 Set Implementation 1: Unordered List

We can represent a set as a Python list with no duplicates in any order.

```
# Example representation:
S = [3, 1, 2]   # valid: unordered, no duplicates
```

```
def make_set():
    return []

def is_empty_set(S):
    return len(S) == 0

def is_element_of_set(x, S):
    for e in S:
        if e == x:
            return True
    return False

def adjoin_set(x, S):
    if is_element_of_set(x, S):
        return S
    return S + [x]

def union_set(S, T):
    U = S
    for x in T:
        U = adjoin_set(x, U)
    return U

def intersection_set(S, T):
    I = make_set()
    for x in S:
        if is_element_of_set(x, T):
            I = adjoin_set(x, I)
    return I
```

Now, we analyse the time complexity of operations in this implementation of a set.

Let $n = |S|$ and $m = |T|$.

Python list concatenation is not constant-time

In Python, the expression `S + [x]` does *not* append in place. It constructs a **new list** and copies all elements of `S`, then adds `x`. Therefore,

`S + [x]` costs $O(|S|)$.

This copying cost is a key reason why some "functional style" list-based implementations are slower than in-place versions.

Membership: `is_element_of_set(x, S)`. The function scans the list until it finds `x` (or reaches the end).

Best-case: $O(1)$ (first element), Worst-case: $O(n)$ (absent or last), Typical/average: $\Theta(n)$.

Adjoin: `adjoin_set(x, S)`.

- First, it performs a membership test: $O(n)$ worst-case.
- If $x \notin S$, it returns `S + [x]`, which copies S: $O(n)$.

Thus, whether x is present or absent,

$$\boxed{\texttt{adjoin\_set}(x, S) \text{ is } O(n) \text{ in the worst case.}}$$

Union: `union_set(S, T)`. The loop processes each of the m elements of T, calling `adjoin_set(x, U)` each time.

- **Best-case:** every $x \in T$ is already in S. Then U stays size n, and each adjoin is dominated by membership testing: $O(n)$. Hence total time $= m \cdot O(n) = O(mn)$.
- **Worst-case:** no element of T is in S. Then U grows through sizes $n, n+1, \ldots, n+m-1$. Each adjoin costs $O(|U|)$, so the total time is:

$$O(n) + O(n+1) + \cdots + O(n+m-1) = O\left(\sum_{i=0}^{m-1} (n+i)\right) = O(mn + m^2).$$

Therefore,

$$\boxed{\texttt{union\_set}(S, T) \text{ is } O(mn) \text{ best-case and } O(mn + m^2) \text{ worst-case.}}$$

Intersection: `intersection_set(S, T)`. The loop runs over the n elements of S. For each $x \in S$, it checks membership in T, which costs $O(m)$ worst-case:

$$n \cdot O(m) = O(nm).$$

Additionally, when x is in T, we adjoin it into I. In the worst case, all n elements of S are also in T, so I grows from size 0 to size n, and the total cost of building I by repeated adjoins is:

$$O(0) + O(1) + \cdots + O(n-1) = O(n^2).$$

So,

$$\boxed{\texttt{intersection\_set}(S, T) \text{ is } O(nm) \text{ best-case and } O(nm + n^2) \text{ worst-case.}}$$

Summary.

Operation	Time (worst-case)
`is_element_of_set(x,S)`	$O(n)$
`adjoin_set(x,S)`	$O(n)$
`union_set(S,T)`	$O(mn + m^2)$
`intersection_set(S,T)`	$O(nm + n^2)$

10.10.3 Set Implementation 2: Ordered List

We represent a set as a Python list that is:

- **sorted** in increasing order, and
- contains **no duplicates**.

This representation requires elements to be comparable (support <, ==).

```
# Example representation:
S = [1, 2, 3, 10]   # valid: ordered, no duplicates
```

```
def make_set():
    return []

def is_empty_set(S):
    return len(S) == 0

def is_element_of_set(x, S):
    """Return True iff x is in the ordered-list set S."""
    for e in S:
        if e == x:
            return True
        if e > x:
            # Early termination: once we pass x, it cannot appear
               later
            return False
    return False

def adjoin_set(x, S):
    """Return a new ordered-list set equal to S union {x}."""
    i = 0
    while i < len(S) and S[i] < x:
        i += 1
    # If x already present, return S unchanged
    if i < len(S) and S[i] == x:
        return S
    # Otherwise, insert x at position i
    return S[:i] + [x] + S[i:]

def union_set(S, T):
    """Return union as a new ordered-list set."""
    i, j = 0, 0
    U = []
    while i < len(S) and j < len(T):
        if S[i] == T[j]:
            U.append(S[i]); i += 1; j += 1
        elif S[i] < T[j]:
            U.append(S[i]); i += 1
        else:
            U.append(T[j]); j += 1
    # Append remaining tail (at most one of these is non-empty)
    return U + S[i:] + T[j:]
```

```python
def intersection_set(S, T):
    """Return intersection as a new ordered-list set."""
    i, j = 0, 0
    I = []
    while i < len(S) and j < len(T):
        if S[i] == T[j]:
            I.append(S[i]); i += 1; j += 1
        elif S[i] < T[j]:
            i += 1
        else:
            j += 1
    return I
```

Now, we analyse the time complexity of operations. Let $n = |S|$ and $m = |T|$.

Important list-slicing / concatenation costs

In Python, the following operations create **new lists** and copy elements:

- `S[:i]` copies i elements: $O(i)$; `S[i:]` copies $(n - i)$ elements: $O(n - i)$,
- `A + B` copies $|A| + |B|$ elements: $O(|A| + |B|)$,
- `U + S[i:] + T[j:]` copies all appended elements (linear in output size).

These costs matter whenever we build results without mutating in place.

Membership: `is_element_of_set(x, S)`. We scan from the beginning until:

- we find x, or
- we see an element $e > x$ (early termination), or
- we reach the end.

Best-case: $O(1)$, Worst-case: $O(n)$, Typical: often $< n$, early stopping (but still $\Theta(n)$).

Adjoin: `adjoin_set(x, S)`. There are two phases:

- find insertion index i by scanning: $O(n)$ worst-case,
- build the new list `S[:i] + [x] + S[i:]`: copying n existing elements overall, so $O(n)$.

Thus,

$$\boxed{\texttt{adjoin\_set}(x, S) \text{ is } O(n) \text{ worst-case.}}$$

Union: `union_set(S, T)`. The merge loop advances i or j each step, so it runs at most $n + m$ iterations. Each iteration does $O(1)$ work (comparisons and append). Appending the tails at the end copies at most $n + m$ elements overall. Therefore,

$$\boxed{\texttt{union\_set}(S, T) \text{ is } \Theta(n + m).}$$

Intersection: `intersection_set(S, T)`. Similarly, the merge loop advances i or j each step, for at most $n + m$ iterations. Each iteration does $O(1)$ work, and the output size is at most $\min(n, m)$. Therefore,

$$\boxed{\texttt{intersection\_set}(S, T) \text{ is } \Theta(n + m).}$$

Summary.

Operation	Time (worst-case)
`is_element_of_set(x,S)`	$O(n)$
`adjoin_set(x,S)`	$O(n)$
`union_set(S,T)`	$\Theta(n + m)$
`intersection_set(S,T)`	$\Theta(n + m)$

10.10.4 Set Implementation 3: Binary Search Tree (BST)

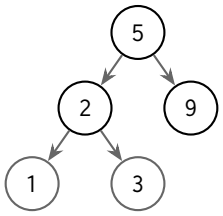
In the previous chapter, we already wrote the core BST operations (`bst_search`, `bst_insert`) and analysed their costs in terms of the tree height h. Here, we simply **reuse that BST implementation** to implement the **Set ADT**.

We represent a set as a BST with **no duplicates**:

- the empty set is `None`,
- a non-empty node is a tuple `(left, value, right)`.

Because the BST invariant is strict ($<$ on the left, $>$ on the right), duplicates are naturally excluded: attempting to insert an existing value simply returns the original tree.

```
# Example representation (using the
    previous chapter's representation):
S = ((None, 1, None),
     2,
     (None, 3, None))
T = (S, 5, (None, 9, None))
```



We assume `bst_search` and `bst_insert` are exactly as defined in the previous chapter.

```
def make_set():
    return None

def is_empty_set(S):
    return S is None

def is_element_of_set(x, S):
    # Reuse BST searching
    return bst_search(S, x)

def adjoin_set(x, S):
    # Reuse BST insertion (duplicates are ignored)
    return bst_insert(S, x)
```

Union and intersection. A simple (and very clear) strategy is: *traverse one tree and insert into another*. We use an in-order traversal because it is the canonical BST traversal (left, node, right), but any full traversal works.

```
def union_set(S, T):
    """Return union as a BST-set."""
    def insert_all(tree, acc):
        if tree is None:
            return acc
        left, val, right = tree
        acc = insert_all(left, acc)
        acc = adjoin_set(val, acc)
        acc = insert_all(right, acc)
        return acc

    return insert_all(T, S)

def intersection_set(S, T):
    """Return intersection as a BST-set."""
    def keep_common(tree, other, acc):
        if tree is None:
            return acc
        left, val, right = tree
        acc = keep_common(left, other, acc)
        if is_element_of_set(val, other):
            acc = adjoin_set(val, acc)
        acc = keep_common(right, other, acc)
        return acc

    return keep_common(S, T, make_set())
```

BST performance depends on height

All fundamental BST operations follow a single root-to-leaf path. Thus:

$$\texttt{bst\_search} = O(h), \qquad \texttt{bst\_insert} = O(h),$$

where h is the height of the tree.

- Balanced BST: $h = \Theta(\log n)$.
- Skewed BST: $h = \Theta(n)$ (worst case).

In this section, we state costs primarily in terms of height.

Complexity analysis. Let $n = |S|$, $m = |T|$, and let $k = |S \cap T|$. Let h_S and h_T denote the heights of the respective BSTs.

- **Membership:** `is_element_of_set(x, S)` calls `bst_search`, so it is $O(h_S)$.
- **Adjoin:** `adjoin_set(x, S)` calls `bst_insert`, so it is $O(h_S)$.

Union: `union_set(S, T)`. We traverse all m nodes of T (visiting each once), and for each visited value we insert into the accumulating tree. If the height of the accumulating tree is h_U, each insertion costs $O(h_U)$, so:

$$\boxed{\texttt{union\_set}(S,T) \text{ is } O(m \cdot h_U) \text{ (plus } O(m) \text{ traversal).}}$$

Balanced typical case: $O\big(m \log(n+m)\big)$. Worst-case skew: $O\big(m(n+m)\big)$.

Intersection: `intersection_set(S, T)`. We traverse all n nodes of S. For each value, we test membership in T (cost $O(h_T)$). If it is present, we insert it into the result tree (there are k such values). Hence:

$$\boxed{\texttt{intersection\_set}(S,T) \text{ is } O\big(n \cdot h_T + k \cdot h_I\big)}$$

where h_I is the height of the result tree. Balanced typical case: $O(n \log m + k \log k)$. Worst-case skew: $O(nm + k^2)$.

Why express costs in terms of h?

Unlike ordered lists (which guarantee $O(n+m)$ merging), a plain BST does not automatically stay balanced. To guarantee $\Theta(\log n)$ height, we need self-balancing trees (AVL / Red-Black), which are beyond our scope here.

Summary.

Operation	Time (height-based)
`is_element_of_set(x,S)`	$O(h_S)$
`adjoin_set(x,S)`	$O(h_S)$
`union_set(S,T)`	$O(m \cdot h_U)$
`intersection_set(S,T)`	$O(n \cdot h_T + k \cdot h_I)$

10.10.5 Comparing Set Representations

Representation	Membership	Union/Intersect	Notes
Unordered list	$O(n)$	$O(nm)$ (simple)	Simplest; good for small sets.
Ordered list	$O(n)$ (early stop)	$O(n+m)$	Requires comparability; great for frequent union/intersection.
BST	$O(h)$ (avg. $\log n$)	varies	Fast membership/insertion when balanced; worst-case $O(n)$.

Table 10.2: Trade-offs among set implementations.

10.11 Python sets

Python provides a built-in `set` type: an **unordered** collection of **unique** elements. Conceptually, a set supports operations such as: membership, insertion, removal, union, and intersection.

A Python set guarantees:

- **Uniqueness:** each element appears at most once.
- **Fast membership (average-case):** `x in S` is typically $O(1)$ on average.

A Python set does *not* guarantee:

- **Order:** do not rely on iteration order for correctness.

Unordered does *not* mean "random"

Python sets are **unordered** in the sense that their iteration/printing order is **not specified by the set abstraction** and is **not guaranteed** to follow insertion order. However, *unordered does not mean random*:

- Within a single run, a set's iteration order is typically **deterministic** given its current contents (because it is driven by the internal hash table layout).
- Small changes (adding/removing elements), different runs, or different Python versions may produce a different order.

Rule: You may *observe* a consistent order, but you must **not rely on it for correctness**. If you need a stable order, explicitly request one: `sorted(S)`.

Mathematical Set Operations

Operation	Notation	Set-Builder Definition	Description
Union	$A \cup B$	$\{x \mid x \in A \vee x \in B\}$	Elements belonging to **at least one** of the sets (OR).
Intersection	$A \cap B$	$\{x \mid x \in A \wedge x \in B\}$	Elements belonging strictly to **both** sets (AND).
Difference	$A \setminus B$	$\{x \mid x \in A \wedge x \notin B\}$	Elements in A, excluding those in B.
Symmetric Diff.	$A \Delta B$	$\{x \mid x \in (A \setminus B) \cup (B \setminus A)\}$	Elements in either A or B, but **not** in both (XOR).

10.11.1 Creating sets and basic usage

The most common operations for working with Python `set` objects are summarised in Table 10.3. Here $n = |S|$ is the number of elements currently in the set, and $m = |T|$ is the size of a second set (or other iterable) used in binary operations.

Operation	Syntax / Example	Time	Notes
Construct empty	`set()`	$O(1)$	Create an empty set.
Set literal	`{1, 2, 3}`	$O(n)$	Duplicates are discarded during construction.
From iterable	`set(iterable)`	$O(n)$ avg.	Inserts each element from the iterable; duplicates are ignored.
Membership	`x in S`	$O(1)$ avg.	Hash-table lookup; worst case can degrade, but average is constant-time.
Size	`len(S)`	$O(1)$	Size is stored in the set header.
Iteration	`for x in S: ...`	$O(n)$	Visits each element once. Order is consistent for a given run/state but **not guaranteed**.
Insert	`S.add(x)`	$O(1)$ avg.	Mutates in place. No effect if `x` already present.
Remove (error)	`S.remove(x)`	$O(1)$ avg.	Mutates in place. Raises `KeyError` if `x` not present.
Remove (no error)	`S.discard(x)`	$O(1)$ avg.	Mutates in place. Does nothing if `x` absent.
Pop arbitrary	`S.pop()`	$O(1)$ avg.	Removes and returns an arbitrary element; raises `KeyError` if empty.
Clear all	`S.clear()`	$O(n)$	Removes all elements.
Union	`S | T`	$O(n+m)$ avg.	Returns a new set. Method form: `S.union(T)`.
Intersection	`S & T`	$O(n+m)$ avg.	Returns a new set. Method form: `S.intersection(T)`.
Difference	`S - T`	$O(n+m)$ avg.	Returns a new set. Method form: `S.difference(T)`.
Symmetric diff	`S ^ T`	$O(n+m)$ avg.	Returns a new set. Method form: `S.symmetric_difference(T)`.
In-place updates	`S |= T`	$O(n+m)$ avg.	Mutates left operand. Analogous: `&=`, `-=`, `^=`.

Table 10.3: Basic `set` operations: syntax, (average) time complexity, and behaviour.

In-place updates vs. returning new sets

Operator forms such as `S | T` return a **new** set. The augmented assignments `S |= T`, `S &= T`, `S -= T`, `S ^= T` update the left-hand set **in place**. Choose in-place updates when you want to mutate an existing set object; choose the non-in-place operators when you want to keep the original sets unchanged.

10.11.2 Set methods

A set is an unordered collection with no duplicates, implemented as a hash table.

Method	Example	Time	Effect
`add(x)`	`s.add(10)`	$O(1)$ avg.	Insert element `x` into the set (no effect if already present).
`remove(x)`	`s.remove(10)`	$O(1)$ avg.	Remove `x`; raise `KeyError` if `x` not in the set.
`discard(x)`	`s.discard(10)`	$O(1)$ avg.	Remove `x` if present; do nothing if absent (no error).
`pop()`	`s.pop()`	$O(1)$ avg.	Remove and return an arbitrary element from the set.
`clear()`	`s.clear()`	$O(n)$	Remove all elements from the set.
`union(t)`	`s.union(t)`	$O(\|s\| + \|t\|)$	Return a new set with all elements that appear in `s`, `t`, or both.
`intersection(t)`	`s.intersection(t)`	$O(\|s\| + \|t\|)$	Return a new set containing only elements common to both sets.
`difference(t)`	`s.difference(t)`	$O(\|s\| + \|t\|)$	Return a new set of elements in `s` but not in `t`.
`symmetric_difference(t)`	`s.symmetric_difference(t)`	$O(\|s\| + \|t\|)$	Return a new set of elements that are in exactly one of `s` or `t` (but not both).
`issubset(t)`	`s.issubset(t)`	$O(\|s\|)$ to $O(\|s\|+\|t\|)$	Return `True` if every element of `s` is also in `t`.
`issuperset(t)`	`s.issuperset(t)`	$O(\|t\|)$ to $O(\|s\|+\|t\|)$	Return `True` if `s` contains every element of `t`.
`isdisjoint(t)`	`s.isdisjoint(t)`	$O(\min(\|s\|, \|t\|))$	Return `True` if the two sets have no elements in common.

Average-case vs. worst-case

The "$O(1)$ average" costs assume a **good hash function** and that the table is not too full. In adversarial or pathological cases, collisions can make operations slower. Most real programs rely on the average-case behaviour.

10.11.3 Hashability constraint

Elements of Python sets must be **hashable** (roughly: usable as dictionary keys). That means:

- the object has a hash value that does not change while it is in the set,
- equality is well-defined for hashing (if `a == b` then their hashes must match).

```
{(1, 2), (3, 4)}   # OK: tuples of hashable items
{[1, 2], [3, 4]}   # ERROR: lists are unhashable (mutable)
```

Immutability

Immutable values (integers, strings, and tuples containing only immutable items) are usually hashable. Mutable containers (like lists) are not hashable, because their contents can change, which would break the hash table's bookkeeping.

10.11.4 Connection to the Set ADT

In this chapter, we studied sets via multiple representations (unordered lists, ordered lists, BSTs) to highlight the core lesson of **data abstraction**:

- The **specification** of a set (membership, union, intersection, ...) stays stable.
- The **representation choice** changes constraints (ordering, comparability, hashability) and performance (linear vs. logarithmic vs. average constant-time).

Python's built-in `set` is one particularly efficient representation: a **hash-table-based** implementation.

10.12 Advanced: Hashing (Out of Syllabus)

Hash tables are the foundational data structure behind sets and dictionaries (hashmaps) in almost every modern programming language.

10.12.1 Hashing

A hash table relies on a function $h : U \to \{0, 1, \ldots, M-1\}$ that maps a universe of keys U to an integer index in an array of size M. To be effective, this function must satisfy three critical properties:

1. **Determinism:** For any inputs x, y, if $x = y$, then $h(x)$ must equal $h(y)$.
2. **Uniformity:** The function should map inputs as evenly as possible over the range $[0, M-1]$ to minimize clustering.
3. **Efficiency:** Computation should be $O(1)$ (or $O(k)$ for string length k) with a minimal constant factor.

10.12.2 The Collision Problem & The Birthday Paradox

A **collision** occurs when two distinct keys map to the same index:

$$h(x) = h(y) \quad \text{for } x \neq y$$

By the Pigeonhole Principle, collisions are inevitable if $|U| > M$. However, the **Birthday Paradox** proves that collisions occur much earlier than intuition suggests.

Mathematical Insight: The Birthday Paradox

Consider a hash table with M slots and n random insertions. The probability that *no* collision occurs is:

$$P(\text{no collision}) = \frac{M}{M} \times \frac{M-1}{M} \times \cdots \times \frac{M-(n-1)}{M} \approx e^{-\frac{n(n-1)}{2M}} \approx e^{-\frac{n^2}{2M}}$$

For a collision probability of 50%, we solve $0.5 \approx e^{-n^2/2M}$, which yields:

$$n \approx \sqrt{2M \ln 2} \approx 1.17\sqrt{M}$$

Implication: In a table of size $M = 10^6$, you only need ≈ 1170 elements to have a 50% chance of a collision. This is why collision handling is very important.

10.12.3 Collision Resolution Strategies

1. Separate Chaining

In this method, each bucket in the main array points to a secondary data structure (typically a linked list). Colliding elements are simply appended to the chain.

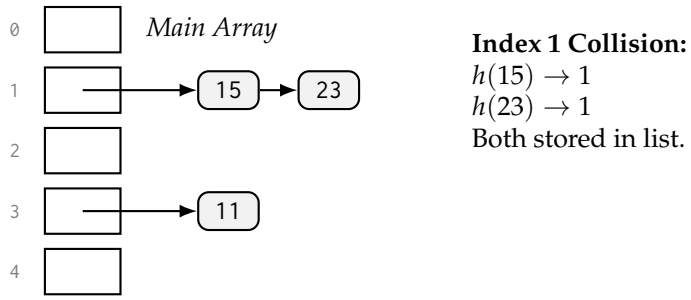


Figure 10.3: Separate chaining. Each bucket anchors a linked list of colliding elements. Performance degrades to $O(n)$ only if one chain becomes disproportionately long.

2. Open Addressing

In open addressing, all elements are stored directly in the main array. If the calculated slot $h(x)$ is occupied, the algorithm strictly follows a deterministic **probing sequence** until an empty slot is found.

Common probing sequences include:

- **Linear Probing:** $i, (i+1), (i+2), \ldots$
- **Quadratic Probing:** $i, (i+1^2), (i+2^2), \ldots$ (reduces primary clustering).

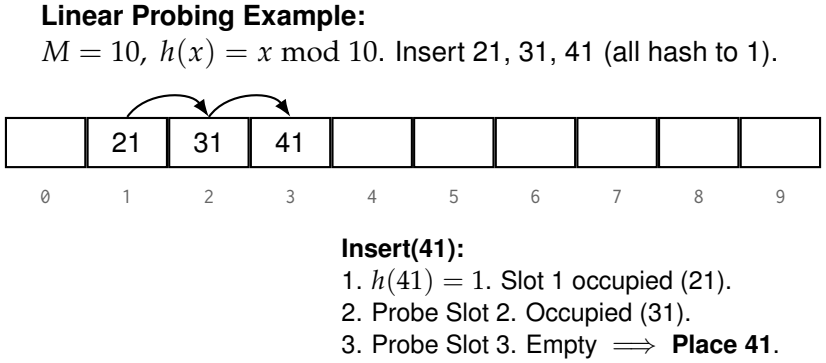


Figure 10.4: Open addressing. Collisions trigger a "probing" search for the next available slot. Deletion requires replacing items with a special `TOMBSTONE` marker to preserve the search path.

10.12.4 Load Factor (α)

The performance of any hash table is fundamentally governed by its **load factor**, defined as:

$$\alpha = \frac{n}{M}$$

where n is the number of stored elements and M is the number of buckets (array size).

- **In separate chaining:** α can technically exceed 1 (more items than buckets). However, as α grows, the average chain length grows, degrading lookup time to $O(\alpha)$.
- **In open addressing:** α must strictly be < 1. As α approaches 1, the probability of collisions skyrockets.

In open addressing, the expected number of probes required to find a key is roughly $\frac{1}{1-\alpha}$.

- At $\alpha = 0.5$, we expect 2 probes.
- At $\alpha = 0.9$, we expect 10 probes.
- At $\alpha = 0.99$, we expect 100 probes.

Because of this exponential degradation, hash tables impose a strict **resizing threshold** (typically when $\alpha \approx 0.66$ or 0.75). When this limit is reached, the table doubles in size ($M \to 2M$) and every element is **rehashed** to a new position.

Mathematical Derivation: Expected Probes

Let $\alpha = n/M$ be the load factor. We assume **uniform hashing**, where the probe sequence for any key is effectively random.
The probability that any random slot is occupied is α. The probability that a slot is **empty** is $1 - \alpha$.
Finding an empty slot is analogous to flipping a biased coin until we get "heads" (where "heads" = empty slot). This follows a **geometric distribution** with success probability $p = 1 - \alpha$.
The expected number of trials E for a geometric distribution is $\frac{1}{p}$. Therefore:

$$E_{\text{probes}} \approx \frac{1}{1-\alpha}$$

Intuitively,

- If the table is 90% full ($\alpha = 0.9$), only 10% of slots are empty ($p = 0.1$).
- On average, you must check $1/0.1 = 10$ slots to find one empty space.
- As $\alpha \to 1$, the denominator $\to 0$, causing $E \to \infty$.

10.12.5 Deletion and Tombstones

Deleting keys in **open addressing** is significantly more complex than insertion. We cannot simply mark a slot as `EMPTY` because doing so might break the probe chain for *other* elements that collided previously.

Broken Chains

Imagine inserting keys **A** and **B**, which both hash to index 5.

1. **Insert A:** Goes into index 5.
2. **Insert B:** Hashes to 5 (collision) $\rightarrow$ Probes to index 6.
3. **Delete A:** If we strictly empty index 5,
4. **Search B:** Hashes to 5. Sees `EMPTY`. IT **stops and returns "not found"**, which is incorrect!

The Solution: Tombstones. Instead of emptying the slot, we replace the deleted item with a special sentinel value often called a `TOMBSTONE` (or `DUMMY`).

The probe logic is then modified to distinguish between "new" empty slots and "recycled" tombstone slots:

1. **Search:** Treat a `TOMBSTONE` as **occupied**. Do not stop; continue probing.
2. **Insert:** Treat a `TOMBSTONE` as **empty**. You can overwrite it with the new value.

Note: Over time, a table can become cluttered with tombstones, which slows down searching (since searches step over them but do not stop). Re-hashing the table clears these tombstones.

10.12.6 Complexity Summary of Hash Tables

Operation	Average	Worst	Notes
Search	$O(1)$	$O(n)$	Degrades if many tombstones exist.
Insert	$O(1)$	$O(n)$	Amortized $O(1)$; occasional $O(n)$ spikes during resize.
Delete	$O(1)$	$O(n)$	Fast, but increases load of the table.

Python's `set` and `dict` use **open addressing** with the following specific tunings:

1. **Probing Schema:** To avoid the "clustering" issues of simple linear probing, Python uses a pseudo-random permutation based on the hash bits:

$$j = (5j + 1 + \text{perturb}) \pmod{M}$$

   The `perturb` variable shifts bits from the hash, ensuring that two keys with similar hashes probe vastly different sequences.

2. **Growth Rate:** CPython typically resizes the table when $\alpha > 2/3$.
3. **Power of Two:** The array size M is always a power of 2 (e.g. 8, 16, 32...). This allows the expensive modulo operation ($x \pmod{M}$) to be replaced by a fast bitwise AND operation (`x & (M-1)`).