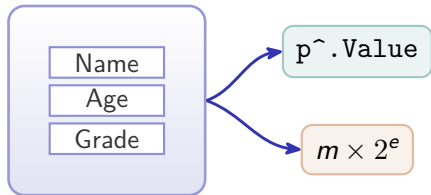


Data Representation

A-Level Computer Science ■ Topic 13

A-Level Computer Science



What we will cover

- 1 User-defined types
- 2 File organisation & access
- 3 Hashing
- 4 Floating-point numbers
- 5 Recap



Why define your own types?

Built-in types (INTEGER, STRING, ...) are too loose for real data.

Restrict

a STRING will hold nonsense; an enumerated type will not

Group

a taxi is a name *and* a capacity *and* a flag

Name

```
DECLARE Taxi :  
  Vehicle reads  
  as what it is
```

User-defined types make the compiler stricter and the code self-documenting.

Why they are needed

A built-in `STRING` lets you store nonsense in a field that should hold one of a few legal values; a user-defined type can restrict it.

Real entities are usually a **collection** of values of different types.

Enumerated type – a fixed named list

An **enumerated type** has values that are a fixed list of named constants.

A fleet of taxis

```
TYPE Vehicle = (M100, M230, T101,  
                T102, T120, T150)  
DECLARE MyTaxi : Vehicle  
MyTaxi ← T102
```

You cannot assign anything outside the list.
Internally the names are small integers.

Vehicle

M100

M230

T101

T102

T120

An enumerated type, drawn

TYPE Vehicle =

M100	M230	T101	T102	T120	T150
------	------	------	------	------	------

MyTaxi may only hold one of these named values

The values are **named constants** in a fixed order, so they can be compared and counted – but they are not strings and not integers.

An **enumerated type** lists its own allowed values by name, so the compiler can reject anything else – clearer than remembering that 3 means Wednesday.

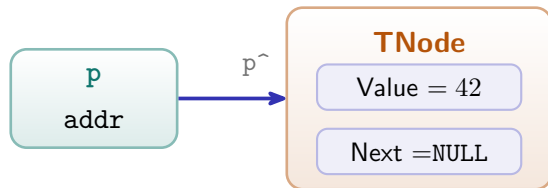
Pointer type – an address, not a value

A **pointer** holds the **memory address** of another variable (or NULL).

- builds dynamic structures: lists, trees
- passes a reference without copying
- **dereference** $p^\wedge$ reaches the target

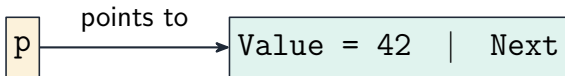
Point at a node

```
TYPE PNode = ^TNode  
DECLARE p : PNode  
p ← NEW TNode  
 $p^\wedge$ .Value ← 42
```



A pointer holds an address

a pointer holds an address



$p^{\wedge}$ dereferences — reach the node's fields, e.g. $p^{\wedge}.Value$

The variable stores **where** the data is, not the data — which is what lets a linked structure grow without moving anything.

A **pointer** does not hold the data; it holds **where the data is**. Follow it to reach the value — and a pointer to nothing is null.

Composite types – several values, one name

A **composite type** groups several values under one name.

record

fields of *different*
types in a TYPE
...ENDTYPE block

set

unordered, unique
values: add, remove,
membership, union

class

attributes *and*
methods – an
object is an instance

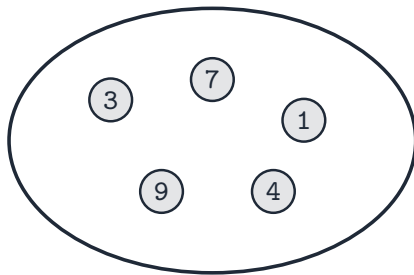
Enumerated and pointer are **non-composite**. Record, set and class are composite.

Two composite shapes

record Student

Name : STRING
Age : INTEGER
Grade : CHAR
Enrolled : BOOLEAN

one name, several fields of different types



unordered, every value unique

A **record** groups **different** types under one name, read by field; a **set** is unordered with no duplicates, and is only ever asked “**is this in it?**”

Choosing a type

Pick the type that matches the **kind of value**, not the first one that compiles.

a value from a fixed list

enumerated

a group of fields

record

an unordered unique collection

set

indirection / a linked structure

pointer

state *and* behaviour together

class



Three ways to lay records out

File organisation is the layout; **file access** is how you reach a record.

serial

records in the *order added*. Append is fast; search is slow. Logs.

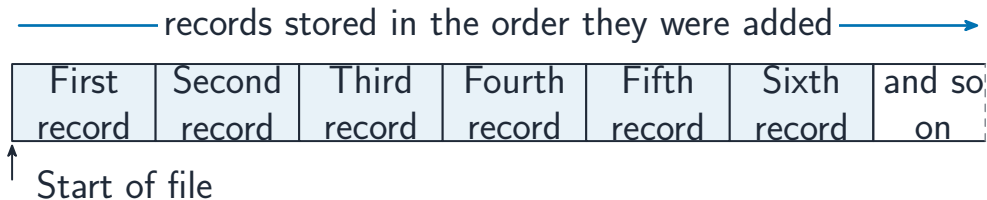
sequential

sorted on a *key*. Search can stop early; insert must shift.

random

position from the key (often a hash). Direct lookup is fast.

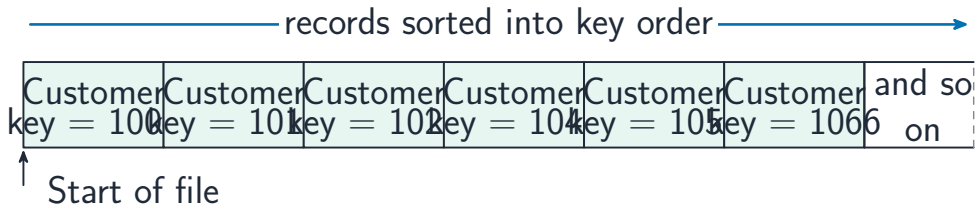
Serial – records in arrival order



New records go on the **end**. Simple to write, but any search reads from the start.

A **serial file** appends each record as it arrives, so writing is fast and finding one means reading from the start.

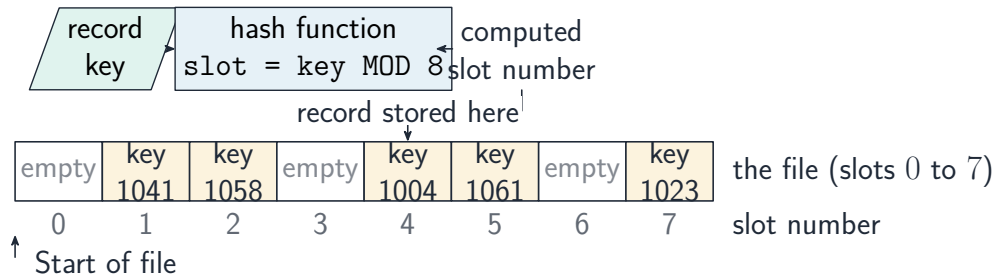
Sequential – ordered by a key



Records are kept in **key order**, so a search can stop early – but an insertion means rewriting the file.

A **sequential file** keeps records **sorted by a key**, so a search can stop early – but every insertion has to make room.

Random – the key computes the address



A **hash** of the key gives the record's position, so one read finds it – **direct access**, at the cost of managing collisions.

Sequential access vs direct access

- **sequential access** – read from the start, one record after another
- **direct access** – jump straight to a known position

Serial and sequential files **need** sequential access. A random file **allows** direct access.

Match the layout to the dominant operation: single-key lookups favour random; in-order reports favour sequential.

serial / sequential

walk from the start

random

hash, then jump

Converting

- **binary** → **denary**: read the mantissa (use two's-complement rules if negative) as a fraction, read the exponent as a signed integer, then multiply mantissa by 2^{exponent} .
- **denary** → **binary**: write the number as a binary fraction $\times$ a power of 2, then store the mantissa and exponent in the agreed formats.



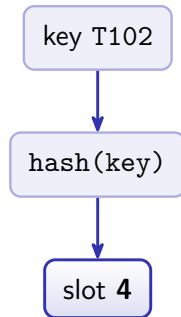
A hash function turns a key into an address

A **hash function** takes a record key and produces the **slot** where the record is stored.

A good one is fast, **deterministic**, and

spreads keys evenly. For N slots:

- modulo: $\text{key} \bmod N$
- folding: split, add the pieces, $\bmod N$
- string: sum character codes, $\bmod N$



Keep the **load factor** below about 70%.

Collisions – two keys, one slot

A **collision** is when two keys hash to the same address.

linear probing

use the next free slot, wrapping around. Simple, but keys *cluster*.

chaining

each slot points to a **linked list** .
No clustering; uses more memory.

rehashing

apply a second hash function. Spreads keys, more work.

Worked example – resolve a collision

Keys A and B both hash to slot 2

Linear probing: B goes in the next free slot (3).

Chaining: slot 2 holds a list $A \rightarrow B$.

To search: hash the key, read that slot; if the keys do not match, follow the resolution strategy until a match or an empty slot.



probing: B sits in 3

Empty slot = “not found”
when probing.



Mantissa $\times$ two-to-the-exponent

A **floating-point** number is binary scientific notation:

$$\text{value} = \text{mantissa} \times 2^{\text{exponent}}$$

Both fields are **two's complement** integers.

- more mantissa bits $\Rightarrow$ more **precision**
- more exponent bits $\Rightarrow$ more **range**

The mantissa is a binary *fraction*: the first bit after the point is $\frac{1}{2}$, then $\frac{1}{4}$, $\frac{1}{8}$, ...

mantissa

0.1010000

exponent

00000010

$$0.625 \times 2^2 = 2.5$$

Worked example – binary to denary

Mantissa 10110000, exponent 00000011

Exponent 00000011 = +3.

Mantissa starts with 1, so it is **negative**. Sign bit = -1, fraction $\frac{1}{4} + \frac{1}{8} = 0.375$:

$$m = -1 + 0.375 = -0.625$$

$$\text{value} = -0.625 \times 2^3 = -5.0$$

A negative mantissa follows two's-complement rules. Convert the exponent first – it is the easy part.

Worked example – store +2.5

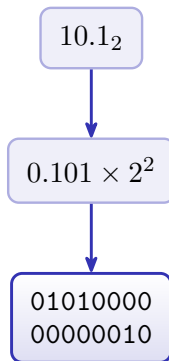
Write +2.5 in this 8+8-bit format

In binary $2.5 = 10.1$.

As a normalised fraction: $2.5 = 0.101 \times 2^2$.

mantissa 01010000 (sign 0, then .101)

exponent 00000010 ($= 2$)



Normalisation maximises precision

A number is **normalised** when the first significant bit sits **immediately after the binary point** – no wasted leading zeros.

- positive: starts 0.1 ...
- negative (two's complement): starts 1.0 ...

Shift the mantissa left and decrease the exponent (or right and increase it) until that pattern holds. The value is unchanged.

Normalise 0.0101

Shift left until it reads 0.1 ...:

$$0.0101 \rightarrow 0.101 \times 2^{-1}$$

mantissa 0.101, exponent -1

Each **left** shift subtracts one from the exponent.

Approximation and rounding errors

Many denary reals **cannot be stored exactly** in binary – e.g.

- **rounding errors** build up over many operations ($0.1 + 0.2$ is not exactly 0.3).
- **comparisons fail** – test $\text{ABS}(x - 0.3) < 1\text{e-}9$ instead of $x = 0.3$.
- subtracting two nearly-equal values loses precision.
- **overflow** (a result too large for the exponent's range) and **underflow** (a result too small, rounding to zero) occur when the exponent runs out of range.

What normalising does to the bits

before 0.0011010 exponent 4 (two wasted leading zeros)

↓ shift left 2, exponent - 2

after 0.1101000 exponent 2 (normalised — same value)

Shifting until the first two bits differ uses every mantissa bit for **precision** instead of for leading zeros.

Normalising shifts the mantissa until the first two bits **differ**, adjusting the exponent to match. That uses every bit of precision the mantissa has.

Why $0.1 + 0.2$ is not 0.3

Many denary reals **cannot be stored exactly** in binary – $0.1_{10} = 0.000110011\dots_2$ must be truncated.

rounding

errors build up over
many operations

overflow

result too large for
the exponent's range

underflow

result too small,
rounds to zero

Never test reals with `=`. Use a tolerance, or use **fixed-point** / BCD for money.

Floating-point numbers

sign

-1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$	$\frac{1}{128}$
0	1	0	1	0	0	0	0

implied

binary point

mantissa (8 bits)

sign

-128	64	32	16	8	4	2	1
0	0	0	0	0	0	1	0

place value

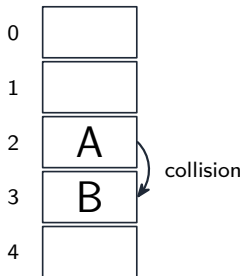
example bits

exponent (8 bits)

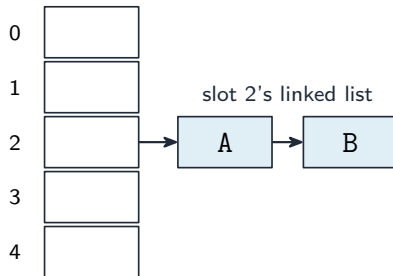
The place values of an 8-bit mantissa and an 8-bit exponent

Resolving a hash collision

linear probing



chaining



Resolving a hash collision: linear probing uses the next free slot; chaining keeps a linked list per slot

Exam tips

- For **floating-point** binary $\rightarrow$ denary, read the mantissa as a fraction and the exponent as a signed integer, then multiply; a **negative mantissa** follows two's-complement rules.
- **Normalise** by shifting until the first digit after the point differs from the sign bit – this maximises precision.
- Explain **rounding, overflow and underflow** errors and why 0.1 cannot be stored exactly.
- Compare file organisation (serial, sequential, direct) and how **hashing** finds a record quickly.



Topic 13 – key takeaways

1 Types

enumerated, pointer, record, set, class

2 Files

serial, sequential, random; sequential vs direct access

3 Hashing

key $\rightarrow$ address; probing / chaining / rehash

4 Float

$m \times 2^e$; normalise; rounding / overflow

Check yourself

- 1 Which file type gives direct access by key?
- 2 Linear probing vs chaining – what is the trade-off?
- 3 How does a normalised *positive* mantissa start?
- 4 Why can 0.1_{10} not be stored exactly?

Answers 1. random 2. probing clusters; chaining uses extra memory 3. $0.1 \dots$ 4. it is a repeating binary fraction