

Data representation

A2

Read online at pastopic.com/cambridge/computer-science-9618/notes/data-representation

Last checked 30 September 2026

Read first: [Information representation](#), Data types and structures

© 2026 Pastopic. Free for personal study. Please share the link, not the file.

What you need to know

This is the A Level topic (Paper 3). Binary, hexadecimal, character sets, images, sound and compression are the AS topic **Information representation**. By the end of this topic you should be able to:

1. **Explain why programmers need user-defined data types**, when the built-in types of a language can't describe the data well.
2. **Define and use non-composite user-defined types**: enumerated and pointer types.
3. **Define and use composite user-defined types**: set, record and class/object.
4. **Choose and design a suitable user-defined type** for a given problem, including the data type of each field.
5. **Describe the three ways of organising a file**, serial, sequential (in key-field order) and random (by record key), and choose the right organisation and access method for a given problem.
6. **Describe the two ways of accessing a file**: sequential access for serial and sequential files, and direct access for sequential and random files.
7. **Explain hashing algorithms** and use them to work out where a record is written to or read from in a random or sequential file, including what to do when two keys give the same address (a collision).
8. **Describe the format of a binary floating-point number**, with a mantissa and an exponent both in two's complement, and explain what happens when bits are moved between the mantissa and the exponent.
9. **Convert** binary floating-point numbers to denary and denary numbers to binary floating-point.
10. **Normalise** a floating-point number and give the reasons for normalising.
11. **Explain the consequences of approximation**: why a binary representation is sometimes only close to the real number, how rounding errors come about, and how overflow and underflow happen.

Understand it

Why user-defined types?

A language gives you **primitive** types such as INTEGER, REAL, CHAR, STRING, BOOLEAN and DATE. Real problems need more: a flight has a number, a destination and a date that belong together; a day of the week can only be one of seven values. A **user-defined data type** is a new type that the programmer builds from

existing types, so that the program's data matches the problem. It makes a language more flexible, keeps related data together under one name, and stops impossible values being stored.

User-defined types come in two kinds:

- **Non-composite:** defined **without referring to any other type**. The user-defined ones are **enumerated** and **pointer**. (Primitive types are non-composite too.)
- **Composite:** defined **using other data types** (primitive or user-defined), grouped under one identifier. Examples: **record**, **set**, **class/object**. (Arrays and lists are composite as well.)

A type definition only describes a shape. It uses `TYPE`. A variable of that type is then created with `DECLARE`, exactly as for built-in types.

Enumerated types

An **enumerated** type lists **every possible value** in order:

```
TYPE Season = (Spring, Summer, Autumn, Winter)
DECLARE ThisSeason : Season
ThisSeason ← Summer
```

- The values are written without quotes: they are not strings, they are the values of the new type.
- The list is **ordered** (ordinal), so `Spring < Summer` is `TRUE` and you can step to the next value.
- The list holds **all** the values the type can take, with **no duplicates**, and all of them are of the same kind.

A field suits an enumerated type when it can only take a small, fixed set of values (a video format, a body style, a taxi's code).

Pointer types

A **pointer** holds a **memory address**, not a data value. Its type says what kind of data is stored at that address:

```
TYPE TIntPointer = ^INTEGER           // points to an INTEGER
DECLARE Score : INTEGER
DECLARE ScorePtr : TIntPointer        // no ^ when declaring the variable
ScorePtr ← ^Score                     // ScorePtr now holds the address of Score
OUTPUT ScorePtr^                      // the value stored at that address
```

The `^` in the `TYPE` line makes it a pointer type. `^Score` gives the address of `Score`, and `ScorePtr^` (caret after) follows the pointer to the value. A pointer can point to a user-defined type too: `TYPE TSeasonPointer = ^Season.`

Records

A **record** groups a **fixed number of related items**, which can be of different types, under one name. Each item is a **field**:

TYPE Booking

 DECLARE BookingRef : STRING

 DECLARE Guests : INTEGER

 DECLARE Arrival : DATE

 DECLARE Deposit : REAL

 DECLARE Paid : BOOLEAN

 DECLARE Room : RoomKind // an enumerated type defined earlier

ENDTYPE

DECLARE Booking1 : Booking

Booking1.BookingRef ← "HX2291"

Booking1.Arrival ← 14/08/2026

Booking1.Room ← Double

Fields are reached with **dot notation**: Booking1.Guests. A whole record can be copied in one go (Booking2 ← Booking1), and you can have an array of records: DECLARE Hotel : ARRAY[1:200] OF Booking, then Hotel[5].Paid ← TRUE.

Choosing each field's type:

Data	Type	Why
Codes mixing letters and digits, names, addresses	STRING	text
Telephone numbers	STRING	leading zeros, spaces or +; never used in arithmetic
Dates	DATE	
Counts, years, minutes, positions	INTEGER	whole numbers
Money, measurements with decimals	REAL	fractional part
Yes/no, complete/not complete	BOOLEAN	two values
A single letter such as a grade or gender	CHAR	one character
One of a fixed list of values	the enumerated type	

A field can also be a **subrange** (DECLARE Copies : 1..10) or an array (DECLARE Marks : ARRAY[1:6] OF INTEGER).

Sets

A **set** holds a collection of values of **one** data type, in **no particular order**, with no value repeated. You can apply set operations to it: **union**, **intersection**, **difference** and membership (is this value in the set?). In pseudocode you first give the set type, then define a set of that type with its values:

```
TYPE LetterSet = SET OF CHAR
DEFINE Vowels ('a', 'e', 'i', 'o', 'u') : LetterSet
```

The set type says which data type its members have (SET OF CHAR, SET OF INTEGER); DEFINE names the set, lists its values in brackets and gives its type after the colon.

Classes and objects

A **class** is a composite type that holds both data (attributes) and the methods that work on it; an **object** is one instance of a class. Here you only need to recognise it as a composite type; writing classes belongs to **Further programming**.

How records are organised in a file

A file of records can be arranged in three ways.

- **Serial**: records are stored **one after another in the order they arrive** (chronological order). A new record is simply **appended** at the end. Good for transaction files, logs and data-logging, where you store everything now and process it later.
- **Sequential**: records are stored one after another **in order of a key field** (for example account number). A new record must go in its correct place, so updating usually means writing a **new version** of the file. Good for files processed record by record in order, such as a monthly payroll or bank statements run.
- **Random**: records are stored **in no particular order**; each record's **address is calculated from its key** by a hashing algorithm. A record can be written, read or updated directly, without touching the others. Good when single records must be found quickly, such as looking up one customer.

How records are accessed

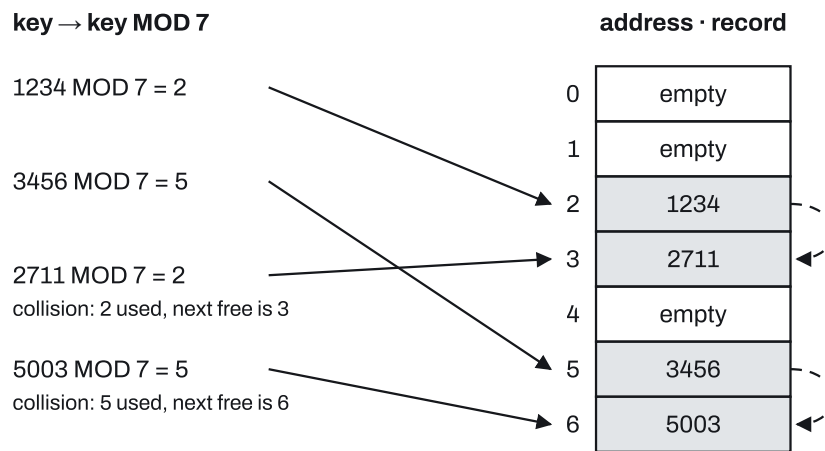
- **Sequential access**: start at the **beginning** of the file and read the records **one by one**.
 - In a **serial** file you keep going until you find the record or reach the end of the file.
 - In a **sequential** file you can stop early: once you pass the place where the key should be (the keys are in order), you know it is not there.
- **Direct access**: go straight to the record **without reading the others**.
 - In a **sequential** file, an **index** of key fields holds each record's address; search the index, then jump to the address.
 - In a **random** file, apply the **hashing algorithm** to the key to calculate the address.

Serial files can't use direct access, because there is no order and no formula linking a key to a place.

Hashing

A **hashing algorithm** is a calculation applied to a record's **key field**; the result is the **address** where the record is stored (or should be found). A common one is $\text{key} \bmod n$, the remainder after dividing by the number of addresses n . Another is **folding**: split the key into parts, add them, then take $\bmod n$ (key 2711: $27 + 11 = 38$,

and $38 \bmod 7 = 3$).



Dashed: a collision moves the record on to the next free address.

Different keys can give the **same** address: a **collision**. Ways to deal with it:

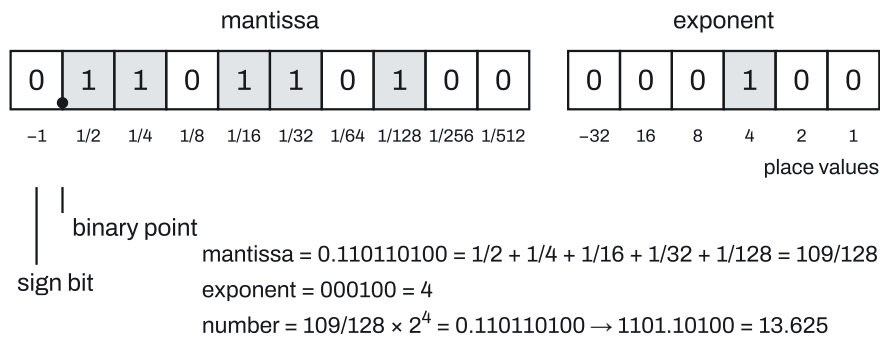
1. **Next free address** (linear probing, "closed hashing"): starting from the hashed address, move through the following addresses one at a time and store the record in the first free one, wrapping round to address 0 after the last.
2. **Overflow area** ("open hashing"): store the record in the first free space in a separate overflow area, searched linearly.
3. **Chaining**: each address holds a link to a list of all the records that hash there.

Reading a record uses the same algorithm: go to the hashed address; if the key there isn't the one you want, search on (the next addresses, or the overflow area) until you find it. If you reach an empty space first, the record is not in the file.

Floating-point numbers

Integers in binary can't hold fractions or very large and very small numbers in a few bits. A **floating-point** number stores a number like scientific notation, $M \times 2^E$:

- the **mantissa** M holds the significant digits, as a two's complement **fraction**: the binary point sits straight **after the first bit**, so the first bit is worth -1 and the rest are worth $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \dots$;
- the **exponent** E is a two's complement **integer** saying how many places to move the binary point: right if E is positive, left if negative.



Reading the diagram: mantissa $0.110110100 = \frac{1}{2} + \frac{1}{4} + \frac{1}{16} + \frac{1}{32} + \frac{1}{128} = \frac{109}{128}$, exponent $000100 = 4$.
Moving the point 4 places right gives $1101.10100 = 8 + 4 + 1 + \frac{1}{2} + \frac{1}{8} = 13.625$.

A **negative** number has a mantissa starting with 1. For -13.625 : $13.625 = 01101.101$; flip the bits and add 1 at the last place to get 10010.011 ($-16 + 2 + \frac{1}{4} + \frac{1}{8} = -13.625$). Moving the point 4 places left gives mantissa 1001001100 with the same exponent 000100 .

Normalisation

The same number can be written many ways: 0.0011010×2^6 and 0.1101000×2^4 are both 13. A **normalised** mantissa has no wasted leading digits:

- a **positive** normalised mantissa starts **01**;
- a **negative** normalised mantissa starts **10**.

If the first two bits are the **same** (00 or 11), it is not normalised. To normalise, move the bits **left** (drop the repeated leading digit, add 0s at the right end) and **subtract 1 from the exponent for each place moved**. For example 0001101000 with exponent 000110 (6) becomes 0110100000 with exponent 000100 (4): both are 13.

Why normalise:

- it gives the **greatest precision** for the bits available, because no mantissa bits are wasted on leading 0s or 1s;
- every number has **one unique** representation, which makes comparing numbers simple;
- it lets the widest range of numbers be stored accurately in the fixed number of bits.

Exact powers of two need care when negative: $-4 = -1 \times 2^2$, so the mantissa is 1000000000 and the exponent 2. (1100000000×2^3 is also -4 , but starts 11, so it is not normalised.)

Range and precision

With a fixed total number of bits, it is a trade-off:

- **more mantissa bits** $\rightarrow$ more **precision** (accuracy): more significant digits are kept;
- **more exponent bits** $\rightarrow$ more **range**: bigger and smaller numbers can be reached.

So taking bits from one to give to the other raises one and lowers the other. For a 10-bit mantissa and 6-bit exponent:

	Mantissa	Exponent	Value
Largest positive	0111111111	011111	$\frac{511}{512} \times 2^{31}$
Smallest positive (normalised)	0100000000	100000	$\frac{1}{2} \times 2^{-32} = 2^{-33}$
Most negative	1000000000	011111	$-1 \times 2^{31} = -2^{31}$

With a 12-bit mantissa and 4-bit exponent instead, the largest positive number is only $\frac{2047}{2048} \times 2^7 = 127.9375$, but each number is stored to 11 binary places.

Approximation, rounding errors, overflow and underflow

A floating-point number is often only **close** to the real number:

- **Fractions that are not a sum of $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \dots$** never end in binary. 0.1 in binary is 0.000110011001100... forever, so it is cut off: in the 10/6 system it is stored as 0.099853515625. 37.4 in a 12/4 system becomes 37.375, because .4 lies between .375 and .40625.
- **Too many significant bits:** if a number needs more bits than the mantissa has, the least significant bits are **truncated** (lost), so **precision is lost**.

These small differences are **rounding errors**. They build up over many calculations, and two values that should be equal may compare as unequal.

- **Overflow:** a result (for example multiplying two large numbers) is **too large** for the exponent to represent. The most significant bits are lost and the result is wrong.
- **Underflow:** a result (for example dividing a tiny number by a large one) is **too close to zero** to represent with the smallest exponent; it may be stored as 0.

Key facts and formulas

There is no formula list for 9618, so everything here is "Learn it".

Formula	Status
Enumerated type: <code>TYPE <name> = (<value1>, <value2>, ...)</code>	Learn it
Pointer type: <code>TYPE <name> = ^<data type>; address of x: ^x; value pointed to: P^</code>	Learn it
Record type: <code>TYPE <name> ... DECLARE <field> : <type> ... ENDTYPE</code>	Learn it
Record variable: <code>DECLARE <var> : <record type>; field: <var>.<field> ← <value></code>	Learn it

Formula	Status
Set: TYPE <name> = SET OF <type> then DEFINE <set> (<values>) : <name>	Learn it
Floating-point value = $M \times 2^E$ (mantissa a two's complement fraction, point after the first bit; exponent a two's complement integer)	Learn it
Mantissa place values: $-1, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \dots$; exponent with e bits: $-2^{e-1}, \dots, 2, 1$	Learn it
Normalised mantissa: positive starts 01, negative starts 10	Learn it
Moving the mantissa one place left: exponent -1	Learn it
m -bit mantissa, e -bit exponent: largest = $(1 - 2^{-(m-1)}) \times 2^{2^{e-1}-1}$	Learn it
Smallest positive normalised = $\frac{1}{2} \times 2^{-2^{e-1}}$	Learn it
More mantissa bits → more precision; more exponent bits → more range	Learn it
Hashing: address = key MOD n (for n addresses 0 to $n - 1$)	Learn it

How to do it

All 69 tagged questions (2021–2025) are Paper 3 questions. Most of them have several parts of different types (50 of the 69 mix types), so the counts add up to more than 69.

Question type	Questions
Convert denary to normalised floating-point	21
Describe types of data type	17
Declare an enumerated type	15
Explain precision, range and approximation problems	15
Convert floating-point to denary	11
Declare a record type	11
Normalise a binary or floating-point number	10
Declare a record variable and assign its fields	9
Describe file organisation and access	9

Question type	Questions
Declare a set or pointer type	7
Hashing and collisions	6
Largest number, overflow and underflow	3

1. Convert denary to normalised floating-point (21 questions)

1. **Write the positive number in binary**, whole part and fraction part: $5.75 = 101.11$ (the fraction from $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{16}, \dots$).
2. **Positive number**: put a 0 in front (0101.11). **Negative number**: give the positive one a leading 0, then flip every bit and add 1 at the **last** place: $-5.75: 0101.11 \rightarrow 1010.00 \rightarrow 1010.01$. Check: $-8 + 2 + \frac{1}{4} = -5.75$.
3. **Move the binary point** to just after the first bit: 0.10111 or 1.01001. The number of places moved **left** is the exponent (here 3).
4. **Fill the mantissa**: copy the bits and pad with **0s on the right** to the full length. **Fill the exponent**: write it in two's complement with the right number of bits, padding on the **left**.
5. **Check it is normalised** (starts 01 or 10) and check by converting back.

For an 8-bit mantissa and 8-bit exponent: $+5.75 \rightarrow 01011100\ 00000011$ and $-5.75 \rightarrow 10100100\ 00000011$. The exponent is the same for $+x$ and $-x$ (unless x is a power of 2); only the mantissa changes.

Variants:

- **Numbers smaller than $\frac{1}{2}$** move the point **right**, so the exponent is **negative**: $0.1875 = 0.0011 = 0.11 \times 2^{-2}$, exponent $-2 = 11111110$ in 8 bits.
- **The fractions method** is also accepted: write the mantissa as a fraction and show $\text{fraction} \times 2^E$.
- **"Show how it would be stored in two systems"**, or **a number that doesn't fit**: when the mantissa is too short, keep the leftmost bits, drop the rest, and say precision is lost (see type 4).

2. Describe types of data type (17 questions)

Give short points, each a separate fact. Useful points:

- **User-defined type (purpose)**: a new type built by the programmer from existing types; allows data types the language doesn't have; extends the language's flexibility.
- **Non-composite**: defined without referring to another data type; can be primitive or user-defined; examples: enumerated, pointer.
- **Composite**: refers to other data types in its definition; a collection of items (same or different types) under a single identifier; examples: record, set, class/object.
- **Enumerated**: user-defined, non-composite; an **ordered** list of **all possible values**; no duplicates; all of the same type.

- **Pointer**: user-defined, non-composite; stores a **memory address**; its definition gives the type of data at that address.
- **Record**: composite; a **fixed** number of **related** items (fields), possibly of different types; the types used may be primitive or user-defined.
- **Set**: composite; an **unordered** collection of values of **the same type**; set operations (union, intersection) can be applied; the definition includes the type of its members.

Always give the example if asked ("Example: an enumerated type such as `TYPE Size = (Small, Medium, Large)`").

3. Declare an enumerated type (15 questions)

1. Write `TYPE`, the name given in the question, then `=`.
2. In round brackets, list every value exactly as given, separated by commas, **without quotes**.

`TYPE Fuel = (Petrol, Diesel, Electric, Hybrid)`: one mark for `TYPE Fuel =`, one for the list.

Variants: two related types (school days and weekend days) each get their own `TYPE` line. "Which field could be an enumerated type?" Pick the field with a small fixed list of possible values and say so as the reason.

4. Explain precision, range and approximation problems (15 questions)

- **Changing the split** (same total bits): more mantissa bits → **precision increases**, exponent has fewer bits → **range decreases**; and the reverse. Say which way each changes and give the new bit counts ($12/4 \rightarrow 10/6$: precision down, exponent now 6 bits so range up).
- **A number that can't be stored exactly**, say why:
 - its fraction isn't a finite sum of $\frac{1}{2}, \frac{1}{4}, \dots$ (like .2 or .1), so the nearest value is stored, a **rounding error**;
 - or it needs more significant bits than the mantissa has (count them: 0.11011001101 has 11 bits after the point, so its normalised mantissa needs 12 bits including the sign bit; a 10-bit mantissa holds only 9 after the point), so the least significant bits are **truncated** and **precision is lost**;
 - or it needs a bigger exponent than exists: **overflow**.
- **"Describe an alteration"** so it can be stored: move bits from the exponent to the mantissa (for example $10/6 \rightarrow 11/5$), keeping the total the same, as long as the exponent still reaches.
- **Why normalise**: maximum precision for the bits; a unique representation for each number; fewest redundant leading 0s or 1s.
- **Why binary is only an approximation**: real numbers can have fractional parts that binary can't represent exactly with a fixed number of bits, so the stored value is as close as possible.

5. Convert floating-point to denary (11 questions)

1. **Exponent** in denary (two's complement): $000101 = 5$, $111101 = -3$.
2. **Move the point**: write the mantissa as M . rest (point after the first bit) and move it E places right (left if E is negative, copying the sign bit in at the front).

3. **Add the place values.** For a negative number the leftmost bit is worth -2^n : $101100.1 = -32 + 8 + 4 + \frac{1}{2}$. Or flip and add 1, work out the positive value and put a minus sign in front.
4. **Or the fractions method:** mantissa as a fraction times 2^E .

For example 01011000 with exponent 11111110 (-2): $0.1011 \rightarrow 0.001011 = \frac{1}{8} + \frac{1}{32} + \frac{1}{64} = \frac{11}{64} = 0.171875$.

Don't forget the **minus sign** on a negative answer.

6. Declare a record type (11 questions)

1. TYPE <Name> on the first line and ENDTYPE on the last.
2. One DECLARE <FieldName> : <type> line per item listed, with a sensible field name.
3. Pick each type with the table in "Understand it": codes and phone numbers STRING, dates DATE, counts INTEGER, money REAL, yes/no BOOLEAN.
4. If the question has just made an enumerated type, **use it** for the matching field.

Marks go for TYPE/ENDTYPE, DECLARE on every field, and the types being right.

7. Normalise a binary or floating-point number (10 questions)

A plain binary number such as 0.0000010110011 or 110100.0101:

1. Move the point so the number reads 0.1 . . . (positive). Count the places moved: **right** gives a **negative** exponent, **left** a positive one.
2. Write the mantissa with a leading 0 and pad (or truncate) to the mantissa length; write the exponent in two's complement.

0.0000010110011 in a 10/6 system: the first 1 is 6 places after the point, so shift 5: $0.10110011 \times 2^{-5} \rightarrow$ mantissa 0101100110, exponent 111011.

A stored pattern that isn't normalised (first two mantissa bits the same):

1. Shift the mantissa left until it starts 01 or 10, filling with 0s at the right.
2. Subtract the number of places from the exponent.

"State why it isn't normalised": the first two bits of the mantissa are the same, 00 or 11.

8. Declare a record variable and assign its fields (9 questions)

1. DECLARE Car1 : Car: a new variable name, the record type after the colon (not the other way round).
2. One assignment per field: `Car1.Model ← "Roadster"`.
 - STRING values in double quotes, CHAR in single quotes ('M');
 - numbers, BOOLEAN (TRUE), dates (12/12/2026) and enumerated values (Hybrid) without quotes.
3. Use the field names exactly as the type defines them, and don't add DECLARE to an assignment.

Variants: "change the field's type to the enumerated type" → `DECLARE BodyStyle : Body`; a field limited to a range → `DECLARE Copies : 1..10`; a field holding many values → an array field, `DECLARE Codes : ARRAY[1:Copies] OF INTEGER`.

9. Describe file organisation and access (9 questions)

Answer about **records**, not files ("the records are stored one after another").

- **Serial:** records in the order they arrive; new ones appended at the end. Use: transaction or log files, temporary unsorted files.
- **Sequential:** records in order of a key field; a new version of the file is written to insert or update. Use: master files processed in key order.
- **Random:** records in no order; the address is calculated from the key with a hashing algorithm; updates made in place.
- **Sequential access:** start at the beginning, read each record in turn until the one needed is found, or the end of file (serial), or the key is passed (sequential).
- **Direct access:** find a record without reading the others: through an index of keys (sequential files) or hashing the key (random and sequential files).

Choosing: a record referenced by its address, or single records found quickly → **direct** access, random organisation. Records kept in key order and mostly all processed in a run → **sequential**. Just collecting records as they come → **serial**.

10. Declare a set or pointer type (7 questions)

Set: two statements.

```
TYPE NumberSet = SET OF INTEGER
DEFINE Squares (1, 4, 9, 16) : NumberSet
```

Marks: the `TYPE ... = SET OF <type>` line, `DEFINE` with the set's name, the values in brackets, and : `<type name>` at the end. Characters go in single quotes: ('+' , '-' , '*').

Pointer: `TYPE TDayPointer = ^Day`, with the name given and the type of data it points to after ^ (a built-in type or a user-defined one from the question). A common mistake is to write another enumerated type instead.

11. Hashing and collisions (6 questions)

1. **Compute hash values:** the remainder after dividing the key by n . $2468 \bmod 7$: $2468 = 352 \times 7 + 4$, remainder 4.
2. **Explain hashing:** a formula applied to the key field; the result is the address where the record is stored or found; used for direct access to (reading from and writing to) random and sequential files.
3. **Explain a collision:** two different keys give the same hash value, so the address is already in use.

4. **Deal with a collision**: say what happens **when storing** (next free address from the hashed one, or the first free space in an overflow area) and, if asked, **when reading** (search on linearly from the hashed address, or through the overflow area, until the key matches; reaching an empty space means the record isn't in the file).

12. Largest number, overflow and underflow (3 questions)

- **Largest positive**: mantissa 0 followed by all 1s; exponent 0 followed by all 1s. Its value: $(1 - 2^{-(m-1)}) \times 2^{\text{largest exponent}}$, for example $\frac{127}{128} \times 2^{127}$ for 8/8. Leave it as powers of 2 if asked.
- **Overflow**: after a calculation (for example multiplying two large numbers), the result is too large for the range of the system; the most significant bits are lost.
- **Underflow**: after a calculation, the result is too small (too close to zero) to be represented.

Worked examples

The bold codes show where marks are earned, as in Cambridge mark schemes. The number is how many marks.

- **M1** method mark: for a correct method, even if the numbers slip.
 - **A1** accuracy mark: for a correct answer; it needs the M mark before it.
 - **B1** mark for a correct result on its own, with no method needed.
-

Example 1

A cycle-hire shop keeps data about its bikes.

(a) Write a pseudocode statement to declare the enumerated type `BikeKind` for the kinds of bike: Road, Mountain, Hybrid, Electric. **[2]**

(b) Write pseudocode statements to declare the composite data type `Bike` with: a bike code (letters and digits), the kind of bike, the date bought, the daily hire charge, the number of gears and whether the bike is out on hire. Use the type from part (a). **[4]**

(c) Write pseudocode to declare a variable `ThisBike` of type `Bike`, and set it to hold an electric bike with code `EB0412` that is not on hire. **[3]**

(a)

```
TYPE BikeKind = (Road, Mountain, Hybrid, Electric)
```

B1 for `TYPE BikeKind =`, **B1** for the bracketed list without quotes.

(b)

```

TYPE Bike
    DECLARE BikeCode : STRING
    DECLARE Kind : BikeKind
    DECLARE DateBought : DATE
    DECLARE DailyCharge : REAL
    DECLARE Gears : INTEGER
    DECLARE OnHire : BOOLEAN
ENDTYPE

```

B1 TYPE Bike and ENDTYPE; **B1** DECLARE used for every field; **B1** BikeKind used for the kind; **B1** the other five types right (the code is STRING because it has letters and a leading 0).

(c)

```

DECLARE ThisBike : Bike
ThisBike.BikeCode ← "EB0412"
ThisBike.Kind ← Electric
ThisBike.OnHire ← FALSE

```

B1 the declaration; **B1** the string in quotes; **B1** Electric and FALSE without quotes.

Example 2

Numbers are stored with a 12-bit mantissa and a 4-bit exponent, both in two's complement.

(a) Calculate the normalised floating-point representation of -37.375 . Show your working. **[4]**

(b) The same number is stored in a system with an 8-bit mantissa and an 8-bit exponent. Give the value that is stored and explain the problem. **[3]**

(a) $37.375 = 32 + 4 + 1 + \frac{1}{4} + \frac{1}{8} = 100101.011$. With a leading 0: 0100101.011. **M1** for the positive binary.

Flip and add 1: $1011010.100 + 0.001 = 1011010.101$. Check: $-64 + 16 + 8 + 2 + \frac{1}{2} + \frac{1}{8} = -37.375$. **M1** for the two's complement.

Move the point 6 places left: 1.011010101, exponent 6. **M1** for the exponent.

Mantissa 101101010100, exponent 0110. **A1**

(b) Only 8 mantissa bits: 1.0110101, dropping the last two bits 01. **M1**

Stored: mantissa 10110101, exponent 00000110, which is $1011010.1 = -37.5$. **A1**

The number needs 10 bits in its mantissa and only 8 are available, so the least significant bits are truncated and precision is lost. **B1**

Example 3

Numbers are stored with a 10-bit mantissa and a 6-bit exponent, both in two's complement.

(a) Calculate the denary value of mantissa 1011001000, exponent 000101. Show your working. **[3]**

(b) Mantissa 1111010000, exponent 000101 is not normalised. State why, and write it in normalised form. **[3]**

(a) Exponent $000101 = 5$. **M1**

1.011001000×2^5 : move the point 5 places right: 101100.1000. **M1**

$-32 + 8 + 4 + \frac{1}{2} = -19.5$. **A1**

(Fractions method: mantissa $= -1 + \frac{1}{4} + \frac{1}{8} + \frac{1}{64} = -\frac{39}{64}$, and $-\frac{39}{64} \times 32 = -19.5$.)

(b) The first two bits of the mantissa are the same (11), so it is not normalised. **B1**

Move the mantissa 3 places left so it starts 10: 1010000000. **B1**

Exponent $5 - 3 = 2$: 000010. **B1**

Check: before, $1.111010000 \times 2^5 = 111101.0000 = -3$; after, $1.010000000 \times 2^2 = 101.0 = -3$.

Example 4

A random file has addresses 0 to 10. The hashing algorithm is $\text{key} \bmod 11$. Collisions are handled by storing the record at the next free address.

(a) Records with keys 4512, 3087, 6129, 2003 and 7710 are written in this order to an empty file. Give the address of each record. **[3]**

(b) Describe how the record with key 6129 is found, and what happens when a search is made for key 5025. **[3]**

(c) Explain why this file organisation is better than a sequential file when single records must be found quickly. **[2]**

(a)

Key	Key MOD 11	Address
4512	2	2
3087	7	7
6129	2	3 (2 is taken: collision)
2003	1	1
7710	10	10

B1 for the hash values, **B1** for spotting the collision, **B1** for 6129 at address 3.

(b) $6129 \bmod 11 = 2$, so read address 2 first (**1**). The key there is 4512, not 6129, so move on to the next address, 3, where the key matches (**1**). For 5025, $5025 \bmod 11 = 9$, and address 9 is empty, so the record is not in the file (**1**).

(c) The address is calculated straight from the key, so the record is read directly without reading any other records (**1**); in a sequential file you would have to read from the start (or search an index) (**1**).

Common mistakes

- **Using DECLARE to define a type.** Enumerated, pointer, set and record types start with TYPE. DECLARE makes a variable. Examiners mention this in almost every session.
- **Swapping the variable and the type.** DECLARE Pet1 : Pet makes a variable Pet1 of type Pet; DECLARE Pet : Pet1 is wrong.
- **Quotes in the wrong places.** Enumerated values are unquoted in the TYPE line and when assigned. Strings need double quotes, characters single quotes; leaving them off strings loses the mark.
- **Adding DECLARE to an assignment,** or inventing new field names instead of the ones in the type.
- **Answering "pointer" with another enumerated type.** A pointer type always has ^ before the type it points to.
- **Describing an enumerated type as just "non-composite".** Add what makes it different: an ordered list of all its possible values.
- **Saying "files" are stored one after another,** when it is the **records** in a file.
- **Mixing up open and closed hashing.** Searching on through the file for the next free space is closed hashing; an overflow area is open hashing. If unsure, just describe the method.
- **Not showing working in floating-point conversions.** Several marks are for working: the binary, the two's complement step, the exponent or the number of places moved.
- **Forgetting the minus sign** when converting a negative floating-point number to denary.
- **Getting the exponent wrong when normalising:** moving the mantissa left makes the exponent **smaller**; a number below $\frac{1}{2}$ has a **negative** exponent, in two's complement.
- **Flipping without adding 1,** or adding 1 in the wrong place: for a number with a fraction part the 1 is added at the **last** bit, not the units bit.
- **Answers about precision without the numbers.** Say how many bits the number needs and how many the mantissa has, and then the consequence: truncation, loss of precision.

How to get the marks

- **"Show your working"** in floating-point: write the binary number, the two's complement version for a negative, the exponent in denary and binary (or "point moved 6 places"), then the final bits in the boxes. The final answer mark often needs every bit exactly right.

- **"Calculate the denary value"**: the exponent, the point moved, the sum of place values, the answer as a decimal or fraction.
- **"Write pseudocode statements"**: use the exact names in the question, the right keywords (`TYPE`, `ENDTYPE`, `DECLARE`, `DEFINE`, `SET OF`, `^`) and the pseudocode arrow `←` for assignment. One mark is often for the keyword line and one for the list or the fields.
- **"Use the most appropriate data type"** means every field is judged; a phone number as `INTEGER` or a date as `STRING` costs a mark.
- **"Describe"** and **"Explain"** questions are marked one point per fact. Write separate short points; three marks need three different points. For definitions, give the category (user-defined, composite or non-composite) and then the feature that makes the type different.
- **"Outline two methods"** (collisions): each method needs its own sentence saying where the record goes.
- **"State the most suitable method"** needs just the name: direct, sequential or serial.
- **"Explain the problem"** with storing a number: the cause (not enough mantissa bits / fraction not exact / exponent too small) and the consequence (truncation, loss of precision, rounding error, overflow).

Quick check

1. Write pseudocode to declare a set type `LetterSet` of characters, and define the set `Vowels` holding the letters a, e, i, o, u.
2. Write pseudocode to declare a pointer type `TTempPointer` that points to a `REAL`, a variable `TempPtr` of that type, and make it point to the variable `Reading`.
3. Using an 8-bit mantissa and an 8-bit exponent in two's complement, write the normalised representation of $+0.1875$.
4. A 12-bit mantissa, 4-bit exponent system holds mantissa 101000000000 and exponent 0011. Find the denary value.
5. For a 10-bit mantissa and 6-bit exponent, give the largest positive number that can be stored, in binary and as a power-of-2 expression.
6. Keys 518, 934, 207 and 653 are stored in that order in a random file with addresses 0 to 7, using key `MOD 8` and the next free address for collisions. Give each record's address.

Answers

1. TYPE LetterSet = SET OF CHAR then DEFINE Vowels ('a', 'e', 'i', 'o', 'u') : LetterSet.
2. TYPE TTempPointer = ^REAL, DECLARE TempPtr : TTempPointer, TempPtr ← ^Reading.
3. $0.1875 = 0.0011 = 0.11 \times 2^{-2}$. Mantissa 01100000, exponent $-2 = 11111110$.
4. Exponent 0011 = 3. Mantissa $1.01000000000 = -1 + \frac{1}{4} = -\frac{3}{4}$. Value $-\frac{3}{4} \times 2^3 = -6$.
5. Mantissa 0111111111, exponent 011111: $\frac{511}{512} \times 2^{31}$ (or $(1 - 2^{-9}) \times 2^{31}$).
6. $518 \bmod 8 = 6 \rightarrow$ address 6. $934 \bmod 8 = 6$, taken $\rightarrow$ address 7. $207 \bmod 8 = 7$, taken; after the last address go back to 0, which is free $\rightarrow$ address 0. $653 \bmod 8 = 5 \rightarrow$ address 5.

Past questions to try

- **9618/33/M/J/23 Q1**: the same number in two systems, and what goes wrong in the smaller one.
- **9618/33/M/J/21 Q1**: every floating-point skill in one question, including why 513 can't be stored.
- **9618/32/O/N/22 Q4**: an enumerated type, a record using it, and assigning a record's fields.
- **9618/32/M/J/24 Q7**: direct access in sequential and random files.