1. Data Representation

13.1 User-defined Data Types

Purpose of user defined data types

- **To create a new data type**
- **To allow construction of data types not available in a programming language (extends flexibility of a programming language)**
- **Constructed by the programmer**

Composite

- **Collection of data that consists of multiple elements of different (or the same) data types which are grouped under a single identifier**
- **Can be user-defined or primitive**
- **Contain more than one data type in their definition**
- **Includes a record, set and class/object**

Record

- **Uses other data types in its definition to form a single new one**
- **Data types referenced can be primitive or user-defined**
- **Includes related items and a fixed number of items**

TYPE *record_name*
DECLARE *field_name1* : DATA TYPE
DECLARE *field_name2* : DATA TYPE
DECLARE *field_name3* : DATA TYPE
ENDTYPE

Set

- **Includes a list of unordered elements**
- **Theory operations (such as intersection or union) can be applied to its elements**
- **Includes the data type it stores ias part of its definition**
- **All elements are of the same data type**

TYPE <*identifier*> = SET OF DATA TYPE

DEFINE <*set_name*> = (*value1, value2, value3, ...*) : <*identifier*>

Non-Composite

- **Can be defined without referencing another data type**
- **Can be a primitive type available in a programming language or a user-defined type**
- **Contains only one data type in their definition**
- **Includes a pointer and any primitive/enumerated data type**
    - **➔ Enumerated Data Type – has an ordered list of all possible values**
    - **➔ Pointer Data Type – used to reference a memory location, stores addresses/memory locations and indicates the type of data stored there**

(Pseudocode for Non-Composite Data Types)

- **Enumerated**

  TYPE <*identifier*> = (*value1*, *value2*, *value3*, ...)

  TYPE Season = (Spring, Summer, Autumn, Winter)

- **Pointer**

  TYPE <*identifier*> = ^DATA TYPE

  TYPE IntPointer = ^INTEGER

13.2 File Organisation and Access

File Organisation Methods

Serial Files

- Records are stored one after the other in chronological order
- New records are appended to the end of the file
- When searching – every record needs to be checked until the record is found or all have been checked
- Examples include...
    - ➔ Creating unsorted/temporary transaction files
    - ➔ Creating data logging files

Sequential Files

- Files are stored and addressed one after the other
- A new version of the file has to be created to update it
- Files are stored with ordered records – records are stored in order of the key field
- New records are inserted in the correct position
- When searching – the key field is compared and every record is checked until it is found, or the key field of a current record is greater than the one being searched for

Random Files

- Records are stored in no particular order within the file (there is no sequencing)
- There is a relationship between the record key and its location within the file – the location of the record is found using a hashing algorithm
- Updates to the file can be carried out directly

File Access Methods

Sequential Access Process (For Sequential and Serial Files)

- Records are checked linearly until the desired record is found/end of file is reached
- Starts searching for records one after the other from the physical start of the file until the record is found or the end of the file is reached
- Most suitable when data is stored in a certain order based on a field – e.g. bank stored data records in ascending order of account number

Direct (For Sequential and Random Files)

- **Most suitable when a record is referenced by a unique address**
- **Allows a record to be found in a file without other records being read – records are found by using the key field of the target record (the record's location is found using a hashing algorithm)**

- **Sequential Files**

  ➔ **an index of all key fields is kept – the index is searched for the address of the file location where the target record is stored**
- **Random Files**
  ➔ **a hashing algorithm is used on the key field of the record to calculate address of the memory location where the target record is expected to be stored**
  ➔ **Linear probing or Search overflow can be used to find a record if it is not at the expected location**

Hashing algorithm

- **Used in direct access methods on random and sequential files**
- **Is a mathematical formula - performs a calculation on the key field of a record**
- **Result of calculation gives the address of the memory location where the record is located/should be stored**

Hash Value Duplicates (a calculated hash value is a duplicate of another value for a different record key)

Collisions

- **occurs when two values/data items in the key field for two records result in the same hash value (when passed through a hashing algorithm)**
- **Means the storage location identified by the algorithm may already be in use by another record – two records cannot occupy the same address**

Process of Collision Resolution

- **When storing a record**
  ➔ **Linear progression - record is stored is next free memory space after the one identified by the hashing algorithm**
  ➔ **Record is stored in the next free memory space in the overflow area**
- **When finding a record**

➔ **search the overflow area linearly until the matching record key is found (if not found, record is not in file)**

13.3 Floating-point Numbers, Representation and Manipulation

Changing the bit allocation

- **When the number of bits in the mantissa is raised, the precision/accuracy of the represented number increases – when the bit number is lowered, the accuracy is reduced**
- **When the number of bits in the exponent is raised, the range of possible numbers can be represented is increased – when the bit number is lowered, the range decreases**

Why binary numbers are stored in normalised form

- **To store a maximum range of numbers in a minimum number of bytes/bits**
- **Normalisation minimises the number of leading 0/1s represented (Numbers whose mantissa begins with 10 or 01 are normalised)**
- **Maximises the number of significant bits – increases precision/accuracy when storing very small/large numbers**
- **Avoids the possibility of many numbers having multiple representation**

Storing floating point numbers

- **Large numbers require a greater number of bits for the mantissa**

Overflow

- **Occurs following an arithmetic/logical operation – result is too large to be precisely represented in the available system**
- **Numbers cannot be stored accurately in certain computer systems if they require more bits than is available**

Underflow

- **Occurs following an arithmetic/logical operation – the result is too small to be precisely represented in the available system (number does not have enough bits to be represented)**

always refer to the loss in precision and state the number/digits would be truncated

Why A Binary Number Is Sometimes An Approximation

- **Real (decimal) numbers can have a fractional part**
- **Binary numbers have limited fractional representation (limited to powers of 2)**

- **Fixed length of storage means you can't store very large/small numbers – it's not possible to store all fractions with the level of precision that is provided**

Converting between denary and floating-point binary

$M \times 2^E$ **where *M is the mantissa* and *E is the exponent***

Both are always in two's complement (There is a binary point following the -1 in the mantissa)

Mantissa Representation:

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

Exponent Representation:

-128	64	32	16	8	4	2	1

- Floating-point binary → Denary

1. **Convert the number in the exponent to denary**
2. **Shift the mantissa binary point to the right by the number held in the exponent**
3. **Convert to binary, where numbers after the point are ½, ¼, ⅛and so on…**

E.g. 01011010 x 00000100

➔ **00000100 = 4**
➔ **0.1011010 - point shifts 4 places right**
➔ **01011.010 = 1 + 2 + 8 + ¼ = 11.25**

- Denary → Floating-point binary

1. **Convert the number into binary, separating fractions using a binary point**
2. **Move the binary point left until only one digit comes before it, counting the amount of places the point moves up (gives you the exponent)**

   Eg. 4.75

   - ➔ **4.5 = 4 + ½ + ¼ = 0100.11**
   - ➔ **0100.11 → 0.10011 - point is shifted 3 places left, hence the exponent is 3**
   - ➔ **3 = 0011, so the final number is 010011 x 0011**