Random Access & Direct Address

Random Access (relative) files

- A random access file is implemented using a **relative file**. *(That's a "logical" (conceptual) concept, not a physical concept, with Windows/Linux OS. Physically, a file is just a stream of bytes).* That is, relative to the front of the file, which record is being referred to: the 1st one, the 10th one, …
- To refer to ("point to") a particular record in the file, the **relative key** or relative record number (**RRN**) is specified – i.e., 1, 2, . . . N. Traditionally, relative files (unlike arrays) **start their RRNs** (EXCLUDING the Header Record) **at 1, not 0** - i.e., 1st, 2nd, 3rd, . . . record. *(But this is just a "logical" concept).*
- Languages like Java, C#, C++, C implement *(a physical concept)* random access files' location referencing by specifying the
  relative BYTE number (**RBN**) of the 1st byte of the record
  rather than their RRN.
- a `seek` command (or some variation) is used to "move" the file position pointer to the correct byte location (i.e., the 1st byte of the desired record location) in the file. *[NOTE: Opening a file sets the file position pointer to the 1st byte of the file, byte 0. Reading from or writing to a file moves the file position pointer to the byte just after the record read/written].*
- **RBNs start at 0, not 1** (unlike RRNs which start at 1, not 0).
- Random access files need a **mapping algorithm** to map some field in the record (the **key**) to an RRN (generally).

DIRECT ADDRESS (DA) file structure

- **Direct address** is the simplest mapping algorithm to map key values (a field in each record) to RRNs. This project uses **id as the primary key** – i.e., the record with id 12 is stored in relative location 12, the record with id 39 is stored at RRN 39. There will never be an id 0, and there is no RRN 0 (per se). *[NOTE: There IS no location set aside in the file for RRN 0, unlike for an array where location 0 exists, but the application may just choose not to use it].*
- DA files need **fixed-length** record **locations** (where the size is knowable),
  - so **fixed-length records** are typically used,
  - so **fixed-length fields** are typically used

FIXED-LENGTH records/fields

- countryData is a **TEXT** file of records
- each record contains a bunch of **"strings"**
  - but since Java/C#/C++/C native strings are **variable-length**, by definition, you must truncate or pad each field to make it a **fixed- length "string"** of the specified size (see specs)
  - each "string" could be implemented as either
    - a char array (of the fixed size)
    - OR a native string (String data type) of the exact specified size
- **alphanumeric fields** (code, name, continent) must be
  - truncated on the RIGHT if they're too long
  - or space-filled on the RIGHT if they're too short
  - designated field-sizes are:
    - 3 chars for **code** *[they're always 3 capital letters, so no problem]*
    - 18 chars for **name** *[I was going to be ethnocentric and say 13 to not have to truncate United States – but then there's the United States Minor Outlying Islands, so we need at least 15 to make names unique]*
    - 13 chars for continent *[so no truncating needed North America & South America]*
- **numeric fields** (id, size, population, lifeExp) must be:
  - 0-filled on the LEFT if they're too small in value
  - Have a .0 appended on the right for lifeExp if it's just an integer in RawData
  - Truncating numeric fields isn't appropriate (you lose important data) – so the designer must decide on field size ahead of time, based on the largest value any record could have (now and in future) for that field:
    - 3 digits for **id** *[there are only 239 countries now, and there'd need to be a lot of breakups before we need > 999 (Sudan, Yugoslavia, Czechoslovakia, Belgium? Great Britain?, Canada?]*
    - 8 digits for **size** *[Russian Federation is 17,075,400 – and even if they annexed Ukraine with 603,700, 8 digits suffices]*
    - 10 digits for population *[China is 1,277,558,000 in the RawData file, and growing; today it's 1.357 billion, but it' a long time before it hits 10 billion (9,999,999,999 + 1) needing 11 digits]*
    - 4 characters for **lifeExp** including the decimal point *[Andorra is the best at 83.5 – humans are a long way from a 100.0 average]*

DOING Random Access

- The input (RawData) file is just a **serial file** in terms of CountryData file's key (id) - i.e., records are not in id order. So CountryData file is created using **random access**, not sequential access – i.e., dataStorage.insert (1 Country) uses random access.
  - This requires that a **seek** to the correct location in the file is done
    - before ANY writing a record to the file
    - or before ANY reading a record from the file.
  - A seek needs a **byte-offset** value (the RelativeByteNumber) as a parameter, which is the number of bytes beyond the 1st byte in the file (which is byte 0). (This is a physical concept, not a logical concept with Java/C#/C++/C/…, so you HAVE TO live with starting with byte 0).
- Similarly, DataStorage's select (by id) and its delete (by id) use random access.
- But its selectAl (by id) uses sequential access – NO `seek`-ing should be done
  EXCEPT 1 `seek` to byte 0 at the start of the method to get to the record at RRN 1.