Assignment 3

2D Priority Queue for City S Meteorological Department

You are the chief algorithmist in the *City S Meteorological Department* (SMD). SMD periodically receives weather updates from a bunch of weather monitoring stations deployed in multiple regions of the city. The system is quite advanced, and alongside a probability of an upcoming disaster, it also monitors and outputs the relative ease with which mitigation can be delivered to a certain region. You had a weekend duty, and everything seemed normal. You are about to leave the office, when suddenly the weather monitoring log starts receiving a set of high priority disaster events (say a Typhoon hitting different parts of the city at different times). Each event is identified by:

- An **event ID (32-digit integer)**: This contains the region ID of a region (within the city) where the typhoon hits, along with some other important flags.
- **Event Priority (key1)**: This is an integer that indicates the priority or disaster level of the typhoon's impact in a particular region. Note that different regions may have different impacts. The lower the integer, the higher the impact.
- **Ease Priority (key2):** This is an integer that indicates the ease/convenience with which mitigation efforts can be implemented in that region. Again, the lower the number, the higher the ease of mitigation in that region.

Being a smart algorithmist, you decide within two minutes that you will implement a **binary min-heap in C** where **each heap node contains two keys (key1 and key2).** And of course, you will have to stay the night on loads of caffeine!

The structure of one node of the min-heap, as decided by you, is:

```
struct HeapNode {
    int key1;       // primary priority
    int key2;       // secondary priority
    char event_id[32];     //alphanumeric event ID, uppercase
};
```

Needless to say, the **Event Priority** is considered more important than the **Ease Priority.** Thus, you decide that a node A will have higher priority (goes higher in the heap) than node B if:

1. key1(A) < key1(B), OR
2. key1(A) == key1(B) AND key2(A) < key2(B), OR
3. key1 and key2 equal → break ties by lexicographic event_id (thus, if key1 and key2 are the same for both events, event ID starting with "a" will be higher in the heap than that starting with "c".

You must manually implement all comparisons and heap operations, without using any pre-existing heap-related libraries.

So far, so good, but just before you start your implementation, you notice that the log files you receive from the monitoring stations are also garbled, and some of the fields (event_id, key1, key2) are out of order or corrupted. You start writing an error resolver now, with high hopes that you will be able to resolve the situation fast.

Input Files

A. You will receive a directory or log named: `events_raw/` that contains **8–15 text files**, where the files are named in the convention **region_i_log.txt** (assume i is a value between 00 and 30) with **60–200 lines**, containing data in the following intentionally inconsistent format:

```
//Entries
id=E82,key1=5,key2=12 # correct order of fields (event_id,key1,key2)

key2=2,id=E77,key1=1  #fields out of order
id=E13,key1=error,key2=9 #error in key1
key1=9,key2=6,id=E77  # duplicate id E77
id=E46,key1=3         # key2 missing
id=,key1=7,key2=8     # corrupted id
```

Assume that two entries cannot be exact duplicates, i.e., all fields cannot be the same for two entries. Note that the fields are comma-separated. Rules to correct inconsistent data are:

1. **Discard** any line with a missing or empty `id`.
2. If `key1 is` missing or non-integer → set to 9999.
3. If `key2` is missing or non-integer → set to 9999.
4. If `event_id` is duplicated:
   - keep the one with the **smallest (id, key1, key2)** in heap-order sense
5. You clean the log files as indicated above, concatenate the data from all log files and save it as `cleaned_events.txt`.

B. **Query File:** You will receive: `queries.txt` with the following example format:

- `BUILD_HEAP`
- `INSERT E42 45 65`
- `INSERT E13 4 17`
- `EXTRACT_MIN`
- `DECREASE_KEY E42 4 2`

- ○ `PRINT_HEAP_ARRAY  # prints in heap order in final_output.txt`

Operations

The heap you implement is in the form of an array. You must implement:

1. `void insert(struct HeapNode *A, struct HeapNode x):` Method to insert a node in your heap (from the queries file)
2. `void decrease_key(struct HeapNode *A, char event_id[], int new_key1, int new_key2):` Updates priority values for an existing heap node with new values supplied;
3. `void print_heap_array(struct HeapNode *A, FILE* fp):` Prints the heap as an array (in the heap order as discussed in the class) in an output file;
4. `struct HeapNode extract_min(struct HeapNode *A):` Extracts the root of the current heap
5. `void build_heap(FILE* fp):` Builds heap from all entries in `cleaned_events.txt.`

Assume that your functions are called either while entering data from the cleaned log or while executing the instructions in the query file. Heap must be stored in an array:

```
struct HeapNode heap[MAXN];
int heap_size = VAL; //Size of heap with a max value that suits you.
```

Operations must:

- Modify the heap
- write snapshot **after each operation** into: `heap_log.txt`

Output Files

cleaned_events.txt

Sorted by (key1, key2, event_id).

Format:

```
event_id key1 key2
E13 1 2
E42 2 9
EXX 9999 9999
```

final_output.txt

Outputs of queries:

```
BUILD_HEAP → SUCCESS
EXTRACT_MIN → E42 4 2
DECREASE_KEY → SUCCESS
INSERT_KEY → SUCCESS
PRINT_HEAP_ARRAY → SUCCESS
E13, E423, E176, E12, E11,… # Should print elements in heap order.
```

Files Students must Submit

Submit a Single ZIP file named in the following Convention: `FirstName_LastName_DA3.zip`
The zip file must contain all your output files and have a `src` folder containing all your source code.

Source Code

1. `src/clean.c` — cleaning & merging
2. `src/heap.c` — heap implementation
3. `src/queries.c` — applies operations from queries.txt

Output Files

1. `cleaned_events.txt`
2. `heap_log.txt`
3. `final_output.txt`