ARIZONA STATE UNIVERSITY

CSE 310 SLN 21622 — **Data Structures and Algorithms** — Spring 2021

Instructor: Dr. Violet R. Syrotiuk

Project #1

Available Mon. 01/25/2021; milestone due Mon. 02/08/2021; complete project due Mon. 02/22/2021

Storm data, provided by the National Weather Service (NWS), contain a chronological listing, by state, of hurricanes, tornadoes, thunderstorms, hail, floods, drought conditions, lightning, high winds, snow, temperature extremes and other weather phenomena. The data also contain statistics on personal injuries and damage estimates. Data is available from 1950 to the present for the United States of America.

This goal of this project is to implement a *storm event application* that manages storm event data and uses it to answer queries meeting given selection criteria.

Note: This project **must** be completed **individually**, *i.e.*, you must write all the code yourself. Your implementation **must** use C/C++ and ultimately your code **must** run on the Linux machine `general.asu.edu`.

All dynamic memory allocation **must** be done yourself, *i.e.*, using either `malloc()` and `free()`, or `new()` and `delete()`. You **may not** use any external libraries to implement any part of this project, aside from the standard libraries for I/O and string functions (`stdio.h`, `stdlib.h`, `string.h`, and their equivalents in C++). If you are in doubt about what you may use, you must ask me for permission.

By convention, your program should exit with a return value of zero to signal that all is well; various non-zero values signal abnormal situations.

You **must** use a version control system as you develop your solution to this project, *e.g.*, GitHub. Your code repository **must** be *private* to prevent anyone from plagiarizing your work.

The rest of this project description is organized as follows. §1 describes the storm and fatality event data format, data structures, and the queries that your storm event application must support. §2 summarizes the requirements for the storm event application in this project. §3 describes experiments to evaluate the effectiveness of the algorithms used in the application. Finally, §4 describes the submission requirements for the milestone and the full project deadlines.

1 Storm Event Application: Data and Query Format

Your storm event application, named `storm`, reads command line argument `n`, indicating the number of years of storm data to manage. Following this are `n` years on the command line, each a distinct four digit year `YYYY`, $1950 \leq$ `YYYY` ≤ 2019. For each year `YYYY`, there are two input data files. The first is named `details-YYYY.csv`, and is a *comma separated value* (CSV) file containing the storm event data. The second is named `fatalities-YYYY.csv`, also a CSV file, containing the storm fatality data. These files should be stored in a directory/folder named `Data`. The format of these data files is described next.

1.1 The Storm and Fatality Event Data Format

For each year, each line of the file `details-YYYY.csv` except the first contains details of a storm event described by the following 13 fields, in order. The first line of the file contains the field names; skip it.

1. `event_id`: An identifier assigned by the NWS for a specific storm event; it links the storm event with the information in the `fatalities-YYYY.csv` file. Example: 383097.
2. `state`: The state name, in all capital letters, where the event occurred. Example: GEORGIA.
3. `year`: The four digit year for the event in this record. Example: 2000.
4. `month_name`: Name of the month for the event in this record. Example: January.
5. `event_type`: The event types permitted are listed in Table 1.
6. `cz_type:` Indicates whether the event is classified as county/parish (C), zone (Z), or marine (M).

Table 1: Valid Event Types

Event Type	Designator	Event Type	Designator
Astronomical Low Tide	Z	Hurricane (Typhoon)	Z
Avalanche	Z	Ice Storm	Z
Blizzard	Z	Lake-Effect Snow	Z
Coastal Flood	Z	Lakeshore Flood	Z
Cold/Wind Chill	Z	Lightning	C
Debris Flow	C	Marine Hail	M
Dense Fog	Z	Marine High Wind	M
Dense Smoke	Z	Marine Strong Wind	M
Drought	Z	Marine Thunderstorm Wind	M
Dust Devil	C	Rip Current	Z
Dust Storm	Z	Seiche	Z
Excessive Heat	Z	Sleet	Z
Extreme Cold/Wind Chill	Z	Storm Surge/Tide	Z
Flash Flood	C	Strong Wind	Z
Flood	C	Thunderstorm Wind	C
Frost/Freeze	Z	Tornado	C
Funnel Cloud	C	Tropical Depression	Z
Freezing Fog	Z	Tropical Storm	Z
Hail	C	Tsunami	Z
Heat	Z	Volcanic Ash	Z
Heavy Rain	C	Waterspout	M
Heavy Snow	Z	Wildfire	Z
High Surf	Z	Winter Storm	Z
High Wind	Z	Winter Weather	Z

7. `cz_name`: County/parish, zone or marine name assigned to the county or zone. Example: AIKEN.
8. `injuries_direct`: The number of injuries directly related to the weather event. Examples: 0, 56.
9. `injuries_indirect`: The number of injuries indirectly related to the weather event. Examples: 0, 87.
10. `deaths_direct`: The number of deaths directly related to the weather event. Examples: 0, 23.
11. `deaths_indirect`: The number of deaths indirectly related to the weather event. Examples: 0, 4, 6.
12. `damage_property`: The estimated amount of damage to property incurred by the weather event, *e.g.*, 10.00K = $10,000; 10.00M = $10,000,000. Examples: 10.00K, 0.00K, 10.00M; convert to integer.
13. `damage_crops`: The estimated amount of damage to crops incurred by the weather event *e.g.*, 10.00K = $10,000; 10.00M = $10,000,000. Examples: 0.00K, 500.00K, 15.00M; convert to integer.

The storm fatality data is also provided in a CSV file named `fatalities-YYYY.csv`. Each line of the file, except the first, contains information on fatalities that occurred within a storm event, described by the following 6 fields, in order. The first line of the file contains the field names; skip it.

1. `event_id`: An identifier assigned by NWS to denote a specific storm event; it links the fatality with the storm event in the `details-YYYY.csv` file. Example: 383097.
2. `fatality_type`: D represents a direct fatality, whereas I represents an indirect fatality. Example: D.
3. `fatality_date`: Date and time of fatality in MM/DD/YYYY HH:MM:SS format, 24 hour time. Example: 04/03/2012 12:00:00.
4. `fatality_age`: The age of the person suffering the fatality. Example: 38.
5. `fatality_sex`: The gender of the person suffering the fatality. Example: F.
6. `fatality_location`: The fatality locations permitted are listed in Table 2.

Fields may be empty. In this case you need to assign a sensible default value, or define an `UNDEFINED` field value of the proper type.

Table 2: Direct Fatality Location Table

Abbreviation	Location
BF	Ball Field
BO	Boating
BU	Business
CA	Camping
CH	Church
EQ	Heavy Equip/Construction
GF	Golfing
IW	In Water
LS	Long Span Roof
MH	Mobile/Trailer Home
OT	Other/Unknown
OU	Outside/Open Areas
PH	Permanent Home
PS	Permanent Structure
SC	School
TE	Telephone
UT	Under Tree
VE	Vehicle and/or Towed Trailer

1.2 Data Structure Format and Initialization

1.2.1 The `defns.h` File

A file named `defns.h` accompanies this project. It defines two structures that correspond to the event data format described in §1.1: `storm_event` contains 13 fields to store the details of each storm event, and `fatality_event` contains 6 fields to store the information on fatalities occurring within a storm event. *The only modifications that you are allowed to make to these structures is to use strings if you prefer them to character arrays.*

In addition, there is a structure `annual_storm` with 5 fields: an integer for the year, a pointer to an array of storm events for the given year, and a pointer to an array of the fatalities corresponding to the storm events for the given year. There is also an integer storing number of events of each type.

Of course, you should `#include defns.h` into any of your source files that require it. You are welcome to add other definitions into this file as you see fit.

1.2.2 Initialization of the Arrays

Your first job is to read the command line arguments and use them to initialize several arrays. Specifically, given command line arguments:

`storm n` $\texttt{YYYY}_1 \ldots \texttt{YYYY}_n$

you must first dynamically allocate an array of `struct annual_storm` of size `n`. For each year $\texttt{YYYY}_i$, $1 \leq i \leq \texttt{n}$, you must initialize the arrays of storm and fatality data for the year. This requires you to:

1. Dynamically allocate the array `storm_events` of size s_i where s_i is the number of storm events in the file named $\texttt{details-YYYY}_i\texttt{.csv}$, and then initialize each element of this array by reading each line of the file $\texttt{details-YYYY}_i\texttt{.csv}$. Set the field `no_storms` to s_i.
2. Dynamically allocate the array `fatality_events` size f_i where f_i is the number of fatality events in the file named $\texttt{fatalities-YYYY}_i\texttt{.csv}$, and then initialize each element of this array by reading each line of the file $\texttt{fatalities-YYYY}_i\texttt{.csv}$. Set the field `no_fatalities` to f_i.

You should find a method to return the number of lines in a file to use in the dynamic allocation of arrays.

1.3 Storm Data Query Input and Output Format

1.3.1 Query Input Format

In the following, <> bracket parameters to the query, while the other strings are literals. There are two queries that your storm application must be able to support:

1. `select <position> <YYYY> <damage_type> <sort>`, where

   - `position` is the literal `max`, the literal `min`, or a valid integer `k`,
   - `YYYY` is either a specific four digit year or the literal `all`,
   - `damage_type` is either `damage_property` or `damage_crops`, and
   - `sort` is either `insertion` or `merge`.

   If `position=max`, this query outputs the most expensive storm(s) causing damage to property or to crops (determined by parameter `damage_type`), in either a specific year or all years specified on the command line, with sorting implemented according to parameter `sort`. Similarly, if `position=min`, this query outputs the least expensive storm(s). Finally, if `position=k`, (`k` is assumed to be in proper range) this query outputs the k^{th} most expensive storm. See §1.3.2 for implementation details and §1.3.3 for the output format you must follow.

   Example queries:

   ```
   select max 1950 damage_property insertion
   select max all damage_crops merge
   select min 1981 damage_property insertion
   select 3 1972 damage_crops merge
   ```

2. `select <position> <YYYY> fatality <sort>`, where

   - `position` is the literal `max`, the literal `min`, or a valid integer `k`,
   - `YYYY` is either a specific four digit year or the literal `all`, and
   - `sort` is either `insertion` or `merge`.

   If `position=max`, this query outputs the storm(s) with the most fatalities, in either a specific year or all years specified on the command line, with sorting implemented according to parameter `sort`. See §1.3.2 for implementation details and §1.3.3 for the output format you must follow.

   Example queries:

   ```
   select max 1950 fatality insertion
   select min all fatality merge
   select 5 all fatality insertion
   ```

1.3.2 Requirements for Sorting

Given that the structures for storm and fatality events have many fields, to implement the `select` queries you do not want to be moving entire array entries when sorting. Instead, for the first query, you must dynamically allocate another array of `struct damage` of size that is the sum of `no_storms` for all years in the query. This structure contains three integer fields depending on the parameter `<damage_type>`:

```
struct damage{
    int damage_amount; // Amount of damage either to property or crops
    int year; // Year of storm
    int index; // Index position in storm_events array
};
```

Now, for each year y in the query, and for $1 \leq i \leq$ `no_storms` in year y:

1. If `<damage_type=damage_property>` then copy the field `damage_property` from the `storm_event` structure at position i of year y into the next position of the `damage_amount` field of the new array. If `<damage_type=damage_crops>` then copy the field `damage_crops` from the `storm_event` structure.
2. Store the year of the storm in the `year` field.
3. Store the index position i of the array `storm_events` for year y in the `index` field.

This simply concatenates a subset of the data from each year in the query in a single array.

If `sort=insert` then use Insertion Sort as the algorithm to sort this new array in non-decreasing order (*i.e.*, increasing order, but be aware that there may be duplicate values) by `damage_amount`; if `sort=merge` then use Mergesort algorithm for sorting.

For the second query, again you must dynamically allocate another array of `struct deaths` of size that is the sum of `no_fatalities` for all years in the query. This structure contains three integer fields depending on the two parameters `<deaths_direct>` and `<deaths_indirect>`:

```
struct deaths{
    int total_deaths; // Sum of direct and indirect deaths
    int year; // Year of storm
    int index; // Index position in storm_events array
};
```

Now, for each year y in the query, and for $1 \leq i \leq$ `no_storms` in year y:

1. Sum the fields `deaths_direct` and `deaths_indirect` from the `storm_event` structure at position i for year y and store this sum in the next position of the `total_deaths` field of the new array.
2. Store the year of the storm in the `year` field.
3. Store the index position i of the array `storm_events` for year y in the `index` field.

Again, this simply concatenates a subset of the data from each year in the query in a single array.

If `sort=insert` then use Insertion Sort as the algorithm to sort this new array in non-decreasing order (*i.e.*, increasing order, but be aware that there may be duplicate values) by `total_deaths`; if `sort=merge` then use Mergesort algorithm for sorting.

1.3.3 Query Output Format

For the selection query on damage to property or crops, there may not be a unique storm with the maximum, minimum, or `k`th damage amount. Hence secondary sorting by `year` and then `event_id` is required (using the sorting algorithm specified on the query). Extract all storm events with `damage_amount` equal to the maximum, minimum, or the amount at position `k`, by year. Then sort these storm events in increasing order, first by `year` and then by `event_id`.

You are now ready to generate the output! First, echo the query itself prefixed by `Query: <query>` followed by a newline. Then in increasing order by `year`, first print the `year`, tab indented, on a new line. Then, output each field in the `storm_event` structure (prefixed by field name) in increasing order by `event_id` indented by two tabs, followed by a newline. Each distinct event should be separated by a newline.

For example for the query: `select max 1950 damage_property insertion`, there are 7 events with a maximum property damage of $2,500,000. Therefore the output you should produce follows (only the first Event Id is provided for brevity):

```
Query: select max 1950 damage_property insertion
   1950
      Event Id: 10009718
      State: ILLINOIS
```

```
Year: 1950
Month: December
Event Type: Tornado
County/Parish/Marine: C
County/Parish/Marine Name: MADISON
Injuries Direct: 0
Injuries Indirect: 0
Deaths Direct: 0
Deaths Indirect: 0
Damage to Property: $2500000
Damage to Crops: $0
```

The fields in the events with identifiers 10009719, 10028673, 10063615, 10073786, 10086810, 10120419, in that order, should then follow.

For the selection query on fatalities, again there may not be a unique maximum, minimum, or `k`th number of fatalities. Hence secondary sorting by `year` and then `event_id` is required (using the sorting algorithm specified on the query). Extract all fatalities with `total_deaths` equal to the maximum, minimum, or the amount at position `k`, by year. Then sort these events in increasing order, first by `year` and then by `event_id`.

You are now ready to generate the output! First, echo the query itself prefixed by `Query: <query>` followed by a newline. Then in increasing order by `year`, print the `year`, tab indented, on a new line. Now, in increasing order by `event_id`, first print the `event_id` indented by two tabs, followed by a newline. Then extract all events with `event_id` prefixed by `Event Id:` from the `fatalities_events` array and print each field (prefixed by field name) in increasing order by `event_id` indented by three tabs, followed by a newline. (Note that there may be multiple fatality events with the same `event_id`.) Each distinct fatality event should be separated by a newline.

For example for the query: `select max 1950 fatality insertion`, there are 3 events with a maximum of 9 deaths, directly and indirectly. They are events with identifiers 10032626, 10032628, and 10126027.

Therefore the output you should produce follows (only the first Event Id is provided for brevity):

```
Query: select max 1950 fatality insertion
   1950
      Event Id: 10032626
         Fatality Type: D
         Fatality Date: 02/12/1950 14:00:00
```

The fields in the events with identifiers 10032628, and 10126027, in that order, should then follow.

In the case that the original field in the `fatalities-YYYY.csv` file was empty, do not print the fields. In this example, the age, sex, and location information were not recorded and are excluded from the output. In general, as years progress, the completeness of the information recorded by the NWS improves.

2 Program Requirements for Project #1

1. Write a C/C++ storm event application, named `storm`, that reads command line argument `n`, indicating the number of years of storm data to manage. Following this are `n` years on the command line, each a distinct four digit year `YYYY`, 1950 $\leq$ `YYYY` $\leq$ 2019. That is, the application is invoked as: `storm n` $\texttt{YYYY}_1 \ldots \texttt{YYYY}_n$. For example,

   ```
   storm 1 1950
   storm 3 1950 1951 1952
   ```

   For each year $\texttt{YYYY}_i$, $1 \leq i \leq n$, there are two input data files. The first is named $\texttt{details-YYYY}_i\texttt{.csv}$, and is a *comma separated value* (CSV) file containing the storm event data. The second is also a CSV

fle, named `fatalities-YYYY`$_i$`.csv`; it contains the storm fatality data. Initialize the data structures using these files as described in §1.2.2.

2. Implement the queries as described in §1.3. A subset of the queries and their parameters are to be implemented for the project milestone deadline (see §4.1 for details). Naturally, by the full project deadline your application must support all queries and all their parameters.

3. Once the arrays have been initialized, and queries implemented, your storm event application is ready to start processing queries. It reads `q`, the number of queries, from `stdin`, followed by `q` queries, one per line. The input format of queries follows the format described in §1.3.1. The output must be written to `stdout` in the format described in §1.3.3.

   Example query sequence.

```
2
select max 1950 damage_crops insertion
select 3 all fatality merge
```

Sample input files that adhere to the format described in §1.1 and §1.3 will be provided on Canvas; use them to test the correctness of your program.

3 Experimentation with the Sorting Algorithms

Run your `storm` application on data sets of increasing size on queries using Insertion sort as the sorting algorithm, repeating using Mergesort, timing the running time. (In general, later years have more storm events and, of course, multiple years have more events.)

Plot the running time of your application as a function of instance size on the x-axis, and running time in milliseconds (or seconds, as best fits the data) on the y-axis. The `clock` function is one function you may use to obtain the processor time used by your algorithm, though it may be easier to use the `time` command on the command line.

Do you observe a cross-over point in your graph? That is, based on your results, can you recommend which sorting algorithm is preferable for a given instance size?

4 Submission Instructions

Submissions are always due before 11:59pm on the deadline date.

1. The milestone deadline is Monday, 02/08/2021. See §4.1 for milestone requirements.
2. The full project deadline is Monday, 02/22/2021. See §4.2 for full project requirements.

It is your responsibility to submit your project well before the time deadline!!! Late projects are not accepted. Do not expect the clock on your machine to be synchronized with the one on Canvas!

An unlimited number of submissions are allowed. The last submission will be graded.

4.1 Requirements for Milestone Deadline

For the milestone deadline you must first complete step 1 in the Program Requirements described in §2, *i.e.*, reading in the data. You must be able to answer queries of the first type only, *i.e.*, selection according to `damage_type`, using `sort=insertion` only.

Using the submission link on Canvas for the milestone, submit a zip[1] file named `yourFirstName-yourLastName.zip` (do not put any spaces in your zip file name) that unzips into the following (and nothing else, *i.e.*, do not include extraneous files):

[1] **Do not** use any other archiving program except `zip`.

Project State (10%): In a folder (directory) named `State` provide a brief report (.pdf preferred) that answers the following:

1. Describe any problems encountered in your implementation for this project milestone.
2. Describe any known bugs and/or incomplete implementation in the project milestone.
3. While this project is to be completed individually, describe any significant interactions with anyone (peers or otherwise) that may have occurred.
4. Cite any external books, and/or websites used or referenced.

Implementation (40%): In a folder (directory) named `Code` provide:

1. In one or more files, your well documented C/C++ source code implementing the requirements for this project milestone.
2. A file named `makefile` (with no filename extension) that compiles your code and produces an executable named `storm`. Our TA will write a script to compile each student submission on `general.asu.edu`; therefore executing the command `make storm` in your `Code` directory must produce the executable `storm` also located in the `Code` directory.
3. The script assumes that the files containing the storm and fatality event data is in a folder (directory) named `Data` inside the `Code` directory.

Correctness (50%): The correctness of your program `storm` will be evaluated by running `select` queries on storm events according to `damage_type`, some of which will be provided to you on Canvas prior to the deadline for testing purposes.

The milestone is worth 30% of the total project grade.

4.2 Requirements for Complete Project Deadline

For the full project deadline, you must implement all the requirements for Project #1 as described in §2. Using the submission link on Canvas, submit a zip[2] file named `yourFirstName-yourLastName.zip` (do not put any spaces in your zip file name) that unzips into the following (and nothing else, *i.e.*, do not include extraneous files):

Project State (10%):

1. Follow the same instructions for Project State as in §4.1 except applied to all project requirements.
2. Include the results of your experimentation as described in §3 in your report, *i.e.*, you must describe the experiments you conducted, a plot of your results, and answer the question about cross-over points and instance sizes.

Implementation (40%): In a folder (directory) named `Code` provide:

1. In one or more files, your well documented C/C++ source code implementing *all* requirements for this project.
2. A file named `makefile` (with no filename extension!) that compiles your code and produces an executable named `storm`. Our TA will write a script to compile each student submission on `general.asu.edu`; therefore executing the command `make storm` in your `Code` directory must produce the executable `storm` also located in the `Code` directory.
3. The script assumes that the files containing the storm and fatality event data is in a folder (directory) named `Data` inside the `Code` directory.

Correctness (50%): The correctness of your program will be evaluated by testing all queries in §1.3 on storm events and fatalities, some of which will be provided to you on Canvas prior to the deadline for testing purposes.

[2] **Do not** use any other archiving program except `zip`.