ARIZONA STATE UNIVERSITY

CSE 310, SLN 30912 — **Data Structures and Algorithms** — Spring 2022

Instructor: Dr. Violet R. Syrotiuk

Project #2

Milestone due Sunday, 03/06/2022; complete project due Sunday, 03/27/2022

The Census Bureau is part of the U.S. Department of Commerce. Thomas Jefferson directed the first census in 1790. As required by the U.S. Constitution, a census has been taken every 10 years thereafter. In addition to censuses, numerous surveys on *e.g.*, demographics and economics, are conducted on an ongoing basis.

This project goal of this project is to implement an application that processes occupation and earnings data and uses it to answer queries satisfying given selection criteria. The data will be indexed by a max-heap, a hash table, and a binary search tree (BST), as appropriate to support the queries. The data sets used in this project have been obtained from the U.S. Census Bureau.

Note: This project **must** be completed **individually**. You **must** implement your solution in C/C++. Your code will be evaluated in Gradescope (Ubuntu 18.04 operating system, with version 7.5.0 of `gcc` and `g++`).

You **may not** use any external libraries to implement any part of this project, aside from the standard libraries for I/O, dynamic memory allocation, and string functions (*i.e.*, `stdlib.h`, `stdio.h`, `string.h`, and their equivalents in C++). In particular, you **may not** use the `vector` class. If you are in doubt about what libraries you may use, ask on `Ed Discussion`.

On the other hand, you **must** use the definitions of constants and structures in the `defns.h` file provided.

By convention, any program you write must exit with a **return value of zero** to signal normal completion; you may introduce non-zero return values to signal abnormal completion.

You **must** use a version control system as you develop your solution to this project, *e.g.*, GitHub or similar. 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 format of the earnings and occupation data, and the queries that your earnings application must support. §2 summarizes the requirements for the earnings application for this project. §3 describes measures to help evaluate the effectiveness of the data structures used in this application. Finally, §4 briefly describes the submission requirements for the milestone and the full project deadlines; see the marking guide on Canvas for details.

1 Data and Query Format

Your application will need to process *comma separated value* (CSV) files containing different forms of earnings and occupation data. The format of these data files is described next.

1.1 Data Format

A file name of the form `Occupation-Dist-All-YYYY.csv` contains a distribution of all U.S. year-round full-time workers by occupation for the four-digit year `YYYY`. Except for the first five (5) lines of the file, which contain information related to the provenance of the data and the field names (which must be skipped), each line contains five (5) fields, as defined in `struct SOC`, in order from left to right:

1. `occupation`: A worker occupation classified using the standard occupational classification (SOC).
2. `SOC_code`: The SOC code(s) associated with the occupation. This may be one code with format `xx-xxxx`, two codes with format `xx-xxxx & yy-yyyy`, or three or more codes with format `xx-xxxx, yy-yyyy, ..., & zz-zzzz`, where each `x`, `y`, and `z` can be any digit.
3. `total`: The number of workers in the occupation that are year-round full-time (YRFT), *i.e.*, the individual worked 50 or more weeks in the year `YYYY` (or is an elementary or secondary school teacher

who worked 37 or more weeks), including paid vacations, and worked 35 or more hours a week. The number of workers is rounded to the nearest 10.
4. `female`: The number of female workers that are year-round full-time.
5. `male`: The number of male workers that are year-round full-time.

The file `Earnings-1960-1999.csv` contains the number of and real median earnings of year-round full-time (YRFT) workers, by sex from the years 1960 to 2019. Except for the first eight (8) lines of the file, which contain information related to the provenance of the data and the field names (which must be skipped), each line contains nine (9) fields, as defined in `struct earnings`, in order from left to right:

1. `year`: The year for which values in the other fields in the given line is provided.
2. `male_total`: The number of male workers (in thousands).
3. `male_with_earnings`: The number of male workers (in thousands) with earnings.
4. `male_earnings`: An estimate of male median earnings (in dollars).
5. `male_earnings_moe`: The margin of error in earnings for male workers.
6. `female_total`: The number of female workers (in thousands).
7. `female_with_earnings`: The number of female workers (in thousands) with earnings.
8. `female_earnings`: An estimate of female median earnings (in dollars).
9. `female_earnings_moe`: The margin of error in earnings for female workers.

For some years, the data in some fields is not available. This is indicated by an "N" in the field.

1.2 Query Format

In this project, there are four (4) queries that your application must ultimately be able to support. You **must** implement the queries using the data structures specified in this section and in §2.

In the following queries, `<>` bracket parameters to the query, while the other strings are literals. In all cases, you must echo the query before answering it. In the sample output, tabs are displayed as three spaces. If a query cannot be answered, output `Query failed.`

1. `find max <worker> <n>`, where `worker` is one of the literals `total`, `female`, or `male`, and `n` is an integer ≤ 50.

   The object of this query is to determine the top `n` occupations in non-increasing order by number of workers in occupations for the year `YYYY`, as specified by the command line parameter to the application.

   To compute the answer to such a query, first build a max-heap on occupational data in the file named `Occupation-Dist-All-YYYY.csv`. Use the `worker` field as the key to order the max-heap. If there is a tie in the number of workers for an occupation, the max-heap should secondarily be ordered lexicographically by occupation.

   Once the heap is constructed, execute a sequence of `n` DELETE-MAX operations on the heap, each time printing tab indented values in the `occupation` and `worker` fields on a single line. The `occupation` must be followed by a colon. If ordered correctly, occupations with an equal number of workers should be deleted from the heap in alphabetical (lexicographical) order.

   Example query: `find max female 2` // find top 2 occupations for females

   Assuming the application was invoked with command line parameter `YYYY` = 1999, then the output produced is:

```
Query: find max female 2

Top 2 occupations in 1999 for female workers:
   Secretaries and administrative assistants: 2,409,830
   Elementary and middle school teachers: 2,143,750
```

2. `find ratio <YYYY> <ZZZZ>`, where $1960 \leq$ `YYYY`, `ZZZZ` ≤ 2019 are specific four-digit years such that `YYYY` $\leq$ `ZZZZ`.

   The object of this query is to determine the female-to-male earnings ratio over the given years `YYYY` through `ZZZZ`. (These results are depressing!)

   To compute the answer to such a query, first build an array of `struct earnings` using the data in the file named `Earnings-1960-2019.csv`. For each year in the range given (in increasing order by year), output the tab indented year, followed by a colon, and then the ratio of the estimate of median earnings for women over men as a percentage with one significant digit after the decimal point.

   Example query: `find ratio 2018 2019` // find female-to-male earnings ratio for 2018-2019

   The output produced is:

```
Query: find ratio 2018 2019

The female-to-male earnings ratio for 2018-2019:
   2018: 81.5%
   2019: 82.3%
```

3. `find occupation <soc_code>`, where `soc_code` is a standard occupational classification (SOC) code.

   The object of this query is to find the details associated with the occupation with the given standard occupational classification (SOC) code.

   To compute the answer to such a query, first call the hash function $h(k)$ with a SOC code of the form `xx-xxxx` treated as an integer $k =$ `xxxxxx`. The hash function returns the position $p = k \bmod m$, where m is the hash table size (see §2.4.1 for more details). Search the chain (*i.e.*, the linked list) at position p of the hash table for the given SOC code. If the SOC code is not found then output `Occupation with SOC code xx-xxxx not found.` Otherwise, output the tab indented occupation, followed by a colon, then `YRFT:` followed by the `total` field, followed by a comma, then `Female:` followed by the `female` field, followed by a comma, and finally `Male:` followed by the `male` field.

   Example query: `find occupation 53-6051` // find the occupation with the given SOC code

```
Query: find occupation 53-6051

The occupation with SOC code 53-6051:
   Transportation inspectors: YRFT: 32,580, Female: 4,590, Male: 27,990
```

4. `range occupation <low> <high>`, where `low` and `high` are double-quoted strings.

   The object of this query is to list all occupations within the range delimited by the double-quoted strings `low` and `high`.

   To compute the answer to such a query, perform an in-order traversal of the BST (see §2.4.1 for details) printing information for occupations whose `occupation` field is greater than or equal to `low` and less than or equal to `high`. Use the standard string comparison function (*i.e.*, `strcmp`) for ordering strings. If no occupations are found in the given range then output `No occupations found in the given range.` Otherwise, your application should produce for each occupation within the range of the search parameters, output in the same form as the `find occupation` query.

   Example query: `range occupation "Da" "De"` // all occupations alphabetically from Da to De

```
Query: range occupation "Da" "De"

The occupations in the range "Da" to "De":
   Dancers and choreographers: YRFT: 5,960, Female: 4,960, Male: 1,010
   Data entry keyers: YRFT: 325,930, Female: 271,630, Male: 54,310
   Database administrators: YRFT: 62,200, Female: 24,710, Male: 37,490
```

2 Program Requirements for Project #2

Write an application, named `earnings` using a modular design (see §2.1), in C/C++ that reads one command line argument consisting of a four-digit year `YYYY`, $1960 \leq$ `YYYY` ≤ 2020. The application is to process queries read from `stdin` and directs output generated to `stdout`. For example, here are two invocations of the executable `earnings`:

```
earnings 1999
earnings 1999 <queries.txt >output.txt
```

In both, the program processes data from 1999, but the first taking queries read from `stdin` and directs output to `stdout`. In the second, `stdin` is redirected from a file named `queries.txt` and `stdout` is redirected to a file named `output.txt` (neither of which changes your implementation).

2.1 Modular Design

Your project **must** use a modular design, *i.e.*, you must provide at least the following source code modules (each in a separate file):

1. A main program, which coordinates all other modules,
2. a module that provides any utility services,
3. a module that implements the max-heap and associated functions,
4. a module that implements the hash table and associated functions, and
5. a module that implements the binary search tree (BST) and associated functions.

Your `makefile` must compile all modules and link them into a single executable named `earnings`.

2.2 File defns.h

Provided for you is a file named `defns.h` in which structures have been defined for storing the occupational and earnings data with fields matching the format described in §1.1. You **must** use these structure definitions and the data structures described in this document to process the queries. This will require the use of forward references in your module design.

2.3 Milestone Requirements

For the milestone deadline you only need to support the two (2) queries, `find max` and `find ratio`. A new max-heap **must** constructed to answer *each* `find max` query and it must be freed once the query is answered. The array of `struct earnings` to answer a `find ratio` *must only be constructed once*; it should only be freed after all queries are answered. See §1.2 for details on how to compute the answer to each query for the milestone.

2.4 Full Project Requirements

By the full project deadline you must be able to support all four (4) queries. To answer `find occupation` and `range` queries, you **must** first build a binary search tree (BST) and a hash table. Each of these data structures *must only be constructed once*, and should be created together, *i.e.*, after a record is inserted into the BST, one or more records are then also inserted into the hash table. Details of the construction of the BST and hash table are described next.

2.4.1 Construction of the BST and Hash Table

In processing a file named `Occupation-Dist-All-YYYY.csv`, for each line of the file:

1. First, create a new record of `struct bst` and initialize the fields of the SOC structure within it using the fields in the line.
2. Now, insert the record as a leaf node into a BST ordered lexicographically by the `occupation` field of the SOC structure. Use the standard string comparison function (*e.g.*, `strcmp`) for ordering strings, and the standard construction of BSTs.
3. Recall that there may be one or more SOC codes associated with an occupation. *For each* SOC code c in the `SOC_code` field:
   (a) Create a new record of `struct hash_table_entry`, setting the `SOC_code` field to the SOC code c, set the `node` field to point to the `struct bst` record created in step 1, and initialize the `next` field to Null. (For occupations with more than one SOC code, there will be multiple entries in the hash table pointing to the record in the BST associated with that application; this is by design.)
   (b) Call the hash function on the SOC code c, and insert the hash table entry into the position returned.

By the end of processing this file, the BST will contain an entry for each occupation, ordered lexicographically by occupation field.

The hash table **must** be implemented using *separate chaining*, *i.e.*, the hash table is an array of pointers of `struct hash_table_entry` initialized to Null. The size, m, of the hash table should be the first prime number greater than three times the number of occupations. Use the function `TestForPrime()` provided in the file `prime.cc` to find such a prime number. The hash function $h(k)$ takes a key k that is a SOC code. You should treat a SOC code of the form `xx-xxxx` as an integer k = `xxxxxx`. The hash function should return the position k mod m *i.e.*, $h(k) = k \bmod m$. In the end, the hash table will have as many entries as there are SOC codes.

2.5 Experimentation with Data Structures

For the full project, you are also to design experiments to collect characteristics of the data structures constructed, and to answer the questions, in a report; see §3 for details.

2.6 Query Input Format

You application **must** read queries from `stdin`. First read `q`, the number of queries, from `stdin`. This is followed by `q` queries, one per line. As described in detail in §1.2, echo the query to `stdout`, process the query, and produce the required output. All output **must** be written to `stdout`.

An example set of queries for the full project is (for the milestone only `find max` and `find ratio` need to be supported):

```
4
find max female 2
find ratio 2018 2019
find occupation 53-6051
range occupation "Da" "De"
```

3 Experimentation

Design experiments to collect and output the characteristics of three data structures constructed in this project.

Max-heap: For a max-heap constructed to answer `find max` queries:

1. Print a count of the total number of nodes in the max-heap.
2. Print the total height of the max-heap, and the height of its left and right subtrees. Is it "height balanced"?
3. Is the max-heap an efficient implementation of the `find max` query? What is another implementation to compute the answer to a `find max` query? Would it be more or less efficient (does it depend on the value of `n`)?

Binary search tree: For the BST constructed to answer `range occupation` queries:

1. Print a count of the total number of nodes in the BST.
2. Print the height of the root of the BST, and the height of its left and its right subtree. Is it "height balanced"?
3. Is the BST an efficient implementation of the `range occupation` query? What is another implementation to compute the answer to a `range occupation` query? Would it be more or less efficient (does it depend on the value of `n`)?

Hash table: For the hash table constructed to answer `find occupation` queries:

1. Print a table that lists for each chain length ℓ, $0 \leq \ell \leq \ell_{max}$, the number of chains of length ℓ, up to the maximum chain length ℓ_{max} that your hash table contains.
2. Compute and print the load factor for the hash table. Do you consider the hash function to be a "good" one for the SOC codes?
3. Is the hash-table an efficient implementation of the `find occupation` query? What is another implementation to compute the answer to a `find occupation` query? Do you think it would it be more or less efficient? Why or why not?

4 Submission Deadlines

Submissions are always due before 11:59pm on the deadline date. Do not expect the clock on your machine to be synchronized with the one on Gradescope!

1. The milestone is due on Sunday, 03/06/2022. See §2 for the milestone requirements. The milestone is worth 30% of the total project grade.
2. The complete project is due on Sunday, 03/27/2022. See §2 for the full project requirements.

It is your responsibility to submit your project well before the time deadline!!! Late projects are not accepted.

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

Detailed marking guides will be posted on Canvas soon. Sample input files that adhere to the format described in §1.1 and §1.2 are provided on Canvas; use Gradescope to test the correctness of your program.