ARIZONA STATE UNIVERSITY

CSE 310, SLN 82170 — **Data Structures and Algorithms** — Fall 2020

Instructor: Dr. Violet R. Syrotiuk

Project #2

Milestone due Thursday, 10/15/2020; complete project due Tuesday, 11/03/2020

This project implements `myAppStore` in which applications of various categories are indexed simultaneously by a hash table and by a search tree for optimal support of various queries and updates of your store.

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

You must build all dynamic data structures by yourself from scratch. All memory management must be handled using only `malloc` and `free`, or `new` and `delete`. That is, you may not make use of any external libraries of any type for memory management! You may only use 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, ask me.

Remember that you **must** use a version control system as you develop your solution to this project. Your code repository must be private to prevent anyone from plagiarizing your work.

1 The `myAppStore` Application

Applications for mobile phones are available from a variety of online stores, such as iTunes for Apple's iPhone, and Google Play for Android phones.

In this project you will write an application called `myAppStore`. First, you will populate `myAppStore` with data on applications under various categories. The data is to be stored simultaneously in both a *hash table* to support fast look-up of an application, and in a *search tree* to support certain queries.

Once you have populated `myAppStore` with application data, you will then process queries about the apps and/or perform updates in your store.

1.1 `myAppStore` Application Data Format

The `myAppStore` application must support n categories of applications. Allocate an array of size n of type `struct categories`, which includes the name of the category, and a pointer to the root of a search tree holding applications in that category. That is, there is a separate search tree for each category of applications. For example, if $n = 3$, and the three categories are "Games," "Medical," and "Social Networking," then you are to allocate an array of size 3 of `struct categories` and initialize each position to the category name and a pointer to the root of a search tree for applications in that category (initially `nil`).

```
#define   CAT_NAME_LEN   25

struct categories{
   char category[ CAT_NAME_LEN ]; // Name of category
   struct tree *root;  // Pointer to root of search tree for this category
};

// Dynamically allocate an array of size n of type struct categories
struct categories *app_categories = (struct categories *) malloc( n * sizeof( struct categories ) );
```

The search tree to be implemented is a *binary search tree* (BST). Each node of the binary search tree contains a record for the application and a pointer to the left and right subtrees, respectively; see `struct app_info`, and `struct tree`). The BST is to be ordered on the application name `app_name` field.

```
struct tree{ // A binary search tree
   struct app_info record; // Information about the application
   struct tree *left;  // Pointer to the left subtree
   struct tree *right;  // Pointer to the right subtree
};
```

For each application, its category, app name, version, size, units, and price are provided in that order in the data set.

```
#define   APP_NAME_LEN 50
#define   VERSION_LEN  10
#define   UNIT_SIZE    3

struct app_info{
   char category[ CAT_NAME_LEN ]; // Name of category
   char app_name[ APP_NAME_LEN ]; // Name of the application
   char version[ VERSION_LEN ]; // Version number
   float size; // Size of the application
   char units[ UNIT_SIZE ]; // GB or MB
   float price; // Price in $ of the application
};
```

First, you are to populate `myAppStore` with m applications. For each application, allocate a node of type `struct tree`. The node contains a structure of type `struct app_info`; initialize the structure. Now, search the array of categories, to find the position matching the category of the application. Insert the node as a leaf into the search tree for that application category.

In addition, you must insert into a hash table using the `app_name` as the key. Only the `app_name` and a pointer to the node just inserted into the search tree storing the full application record are to be stored in the hash table (not any of the other fields of the application). The hash table is to be implemented using *separate chaining*, with a table size k that is the first prime number greater than $2 \times m$. (You may find the boolean function in the file `prime.cc` provided to you useful; it returns *true* if the integer parameter is a prime number and *false* otherwise.) That is, a hash table of size k containing entries of type `struct hash_table_entry *` is to be allocated and maintained.

```
struct hash_table_entry{
   char app_name[ APP_NAME_LEN ]; // Name of the application
   struct tree *app_node; // Pointer to node in tree containing the application information
   struct hash_table_entry *next; // Pointer to next entry in the chain
};

// Declare hash table; dynamically allocate it as an array of size k of pointers
struct hash_table_entry **hash_table;
```

The hash function is computed as the sum of the ASCII value of each character in the application name, modulo the hash table size. For example, if a game is named `Sky` and the hash table size is 11, then the hash function value is: $(83 + 107 + 121) \bmod 11 = 311 \bmod 11 = 3$, because the ASCII value for `S` is 83, for `k` is 107, and for `y` is 121. That is, the `app_name` and a pointer to the node inserted into the search tree, is inserted at the head of the chain at position 3 of the hash table.

1.2 `myAppStore` Queries and Updates

Once `myAppStore` is populated with m applications, you are ready to process q queries and updates. When all queries and updates are processed, if requested your `myAppStore` application is to collect characteristics of the data structures constructed, and then terminate gracefully. Graceful termination means your program must deallocate all dynamically allocated data structures that you created before it terminates.

In the following, `<>` bracket variables, while the other strings are literals. There are 5 queries that `myAppStore` must be able to process:

1. `find app <app_name>`, searches the hash table for the application with name `<app_name>`. If found, it prints `Found Application: <app_name>` and then follows the pointer to the node in the search tree to print the record associated with the application (*i.e.*, the contents of the `struct app_info`, with each field tab indented and labelled on a separate line); otherwise it prints `Application <app_name> not found.` substituting the parameter `app_name` given in the command.

2. `find category <category_name>`, searches the array of categories to find the given `<category_name>`. If found, and the BST is not empty prints `Category: <category_name>`, then performs an in-order traversal of the search tree for that category, printing the tab indented name of the application (field `app_name` in the record), *i.e.*, this results in a list of applications of the given category in sorted order by app name. If the `<category_name>` is found but the tree is empty, then print `Category <category_name> no apps found.` If the `<category_name>` does not exist, prints `Category <category_name> not found.`, always substituting the parameter `category_name` given in the command.

3. `find price free`, steps through the array of categories in the order provided and, for each category, performs an in-order traversal of the search tree, printing the names of the applications in each category whose price is free on a separate line. Organize your output by category, *i.e.*, with the title `Free Applications in Category: <category_name>`. If no free applications are found print `No free applications found.`

4. `range <category_name> price <low> <high>`, for the given `<category_name>`, performs an in-order traversal of the search tree, printing the tab indented name of the each application whose price is greater than or equal to (float) `<low>` and less than or equal to (float) `<high>` on a separate line with the header `Applications in Price Range (<low>,<high>) in Category: <category_name>`. If no applications are found whose price is in the given range print `No applications found in <category_name> for the given price range (<low>,<high>).` substituting the parameters given in the command.

5. `range <category_name> app <low> <high>`, for the given `<category_name>`, performs an in-order traversal of the search tree, printing the tab indented names of the applications whose application name (`app_name`) is alphabetically greater than or equal to (string) `<low>` and less than or equal to (string) `<high>` with the header `Applications in Range (<low>,<high>) in Category: <category_name>`. If no applications are found whose name is in the given range print `No applications found in <category_name> for the given range (<low>,<high>).` substituting the parameters given in the command.

There is only one update that `myAppStore` must be able to process:

1. `delete <category_name> <app_name>`, first searches the hash table for the application with name `<app_name>`. Then it first deletes the entry from the search tree of the given `<category_name>`, and then also deletes the entry from the hash table. Finally, it prints `Application <app_name> from Category <category_name> successfully deleted.` If the application is not found it prints `Application <app_name> not found in category <category_name>; unable to delete.` substituting the parameters given in the command.

1.3 Sample Input

The following is sample input `myAppStore` must process. You may assume that the input is in the correct format. (The comments are not part of the input.)

```
3               // n=3, the number of app categories
Games           // n=3 lines containing the names of each of the n categories
Medical
Social Networking
4               // m=4, number of apps to add to myAppStore; here all in Games
Games           // Each field in app_info is provided in order; first the name of the category
Minecraft: Pocket Edition      // Name of the application
0.12.1          // Version number of the application
24.1            // Size of the application
MB              // Units corresponding to the size, i.e., MB or GB
6.99            // Price of the application
Games           // Start of record for the second app
FIFA 16 Ultimate Team
2.0
1.25
GB
0.00
Games           // Start of record for the third app
Candy Crush Soda Saga
1.50.8
61.3
MB
0.00
Games           // Start of record for the fourth app
Game of Life Classic Edition
1.2.21
15.3
MB
0.99
8                           // q=8, number of queries and/or updates to process
find app Candy Crush Soda Saga // List information about the application
find category Medical       // List all applications in the Medical category
find price free             // List all free applications
range Games app A F         // List alphabetically all Games whose name is in the range A-F
range Games price 0.00 5.00 // List all names of Games whose price is in the range $0.00-$5.00
delete Games Minecraft      // Delete the game Minecraft from the Games category
find category Games         // List all applications in the Games category
find app Minecraft          // Application should not be found because it was deleted
no report  // do not produce hash table and tree statistics
```

1.4 Sample Output

The following is sample output produced by `myAppStore`. (The comments are not part of the input.)

```
Found Application: Candy Crush Soda Saga   // Output of: find app Candy Crush Soda Saga
   Category: Games
   Application Name: Candy Crush Soda Saga
   Version: 1.50.8
```

```
   Size: 61.3
   Units: MB
   Price: $0.00

Category Medical no apps found.    // Output of: find category Medical

Free Applications in Category: Games   // Output of: find price free
   Candy Crush Soda Saga
   FIFA 16 Ultimate Team

Applications in Range (A,F) in Category: Games   // Output of: range Games app A F
   Candy Crush Soda Saga

Applications in Price Range ($0.00,$5.00) in Category: Games // Output of: range Games price 0.00 5.00
   Candy Crush Soda Saga
   FIFA 16 Ultimate Team
   Game of Life Classic Edition

Application Minecraft from Category Games successfully deleted. // Output of: delete Games Minecraft

Application Minecraft not found.   // Output of: find app Minecraft
```

2 Program Requirements for Project #2

1. Write a C/C++ program that implements all of the queries and updates described in §1.2 on data in the format described in §1.1. **You must build all dynamic data structures, *i.e.*, the hash table and the BSTs, by yourself from scratch. All memory management must be handled using only `malloc` and `free`, or `new` and `delete`.**
2. If the last command is `report`, then collect characteristics of the data structures you've built as described in §3 for your report. If the last command is `no report` then no statistics are collected.
3. Provide a `makefile` that compiles your program into an executable named `myAppStore`. This executable must run on `general.asu.edu`, compiled by a C/C++ compiler that is installed on that machine, reading input from `stdin` (of course, you may redirect `stdin` from a file in the prescribed format).

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

3 Characteristics of the Data Structures

For the binary search tree associated with each category: Print the category name, a count of the total number of nodes in the tree, the height of the tree, the height of the root node's left subtree, and the height of the root node's right subtree.

For the hash table: 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. In addition, compute and print the load factor α for the hash table, giving n and m.

Implement the `find app <app_name>` command by directly searching the BST instead of the hash table. The easiest way to do this may be to use the hash table to extract the `<category_name>` and then search the appropriate BST. Compare the time to find an ¡app_name¿ using the hash table, and by searching the BST for its category.

4 Submission Instructions

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 Canvas!

1. The milestone is due on Thursday, 10/15/2020. See §4.1 for requirements.
2. The complete project is due on Tuesday, 11/03/2020. See §4.2 for 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.

4.1 Requirements for Milestone Deadline

For the milestone deadline, the hash table and the array of categories with a `binary search tree` for each category must be implemented for `myAppStore`. You need only be able to support the query: `find app <app_name>`.

Submit electronically, before 11:59pm on Thursday, 10/15/2020 using the submission link on Canvas for the Project #2 milestone, a zip[1] file named `yourFirstName-yourLastName.zip` containing the following:

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

1. Describe any problems encountered in your implementation for this project milestone.
2. Describe any known bugs and/or incomplete query implementation for the project milestone.
3. While this project is to be complete 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.
5. A screen shot of your version control system showing commits over the development period.

Implementation (50%): In a folder (directory) named `Code` provide:

1. In one or more files, your well documented C/C++ source code implementing the hash table, and the array of categories with a binary search tree for each category. There is only one query to support for the project milestone.
2. A `makefile` that compiles your program to an executable named `myAppStore` on `general.asu.edu`. Our graders will write a script to compile and run all submissions on `general.asu.edu`; therefore executing the command `make myAppStore` in the `Code` directory must produce the executable `myAppStore` also located in the `Code` directory.

Correctness (40%): The correctness of your program will be determined by running it with input that adheres to the specified format, some of which will be provided to you on Canvas prior to the deadline for testing purposes. For the milestone deadline, these will only contain `find category <category_name>` queries.

Of utmost importance in this project is your memory management. You must build your dynamic data structures from scratch and implement graceful termination (see §1.2).

As described in §2, your program must read input from standard input. **Do not use file operations to read the input!**

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

[1] **Do not** use any other archiving program except `zip`. Do not include any files in your zip that are not directly related to the project!

4.2 Requirements for Complete Project Deadline

For the complete project, you must now add support of all additional queries and updates. Your program should then terminate gracefully (see §1.2).

Submit electronically, before 11:59pm on Tuesday, 11/03/2020 using the submission link on Canvas for the complete Project #2, a zip[2] file named `yourFirstName-yourLastName.zip` containing the following:

Project State (5%): Follow the same instructions for Project State as in §4.1.

Characteristics of Data Structures (15%): In a folder (directory) named `Data` provide a brief report (.pdf preferred) that reports the characteristics of the hash table and BSTs constructed, and an experiment to evaluate the implementation of `find`, for several data sets as described in §3. Clearly label your results by the data set used!

Interpret your results to answer the following questions:

1. How well do you think the hash function satisfies the assumption of simple uniform hashing?
2. Do the BSTs appear to be balanced, and does this impact any queries?
3. Which data structure best supports the `find` command? Are your results statistically significant?

Implementation (40%): Follow the same instructions for Implementation as in §4.1.

Correctness (40%): The same instructions for Correctness as in §4.1 apply except that the input files will exercise all queries and updates from §1.2 rather than a subset of them. In addition, the output is to include characteristics of the data structures constructed.

5 Marking Guide

The project milestone is out of 100 marks.

Project State (10%): Summary of project state, use of a zip file, and directory structure required (*i.e.*, a folder/directory named `State` and `Code` is provided).

Implementation (50%): 45% for the quality of implementation in your code including proper memory management, construction of the binary search trees, and query processing; 5% for a correct `makefile`.

Correctness (40%): 40% for correct output on several sets of sample input data, 5% for redirection from standard input.

The full project is out of 100 marks.

Project State (5%): Summary of project state, use of a zip file, and directory structure required (*i.e.*, a folder/directory named `State`, `Code`, and `Report` is provided).

Characteristics of Data Structures (15%): A brief report answering presenting the characteristics of the data structures collected, design of an experiment for the `find` command, and interpretation of your results.

Implementation (40%): 35% for the quality of implementation in your code including proper memory management, construction of the hash table and the binary search trees, and all query/update processing; 5% for a correct `makefile`.

Correctness (40%): 45% for correct output on several sets of sample input data; 5% for redirection from standard input.

[2] **Do not** use any other archiving program except `zip`. Do not include any files in your zip that are not directly related to the project!