ARIZONA STATE UNIVERSITY

CSE 310, SLN 88400 — **Data Structures and Algorithms** — Fall 2018

Project #2

Available 10/03/2018; milestone due Wednesday, 10/17/2018; full project due Wednesday, 10/31/2018

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

Note: This project is to be completed individually. Your implementation *must* use C/C++ and your code *must* run on the Linux machine `general.asu.edu`. See §2 for full project requirements. No changes to these project requirements are permitted.

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!

Remember to use a version control system as you develop your solution to this project. Your code repository should 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 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
};
```

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

```
struct tree{ // A binary search tree
   struct app_info; // 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.

```
#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
};
```

You are to populate `myAppStore` with m applications. The information associated with each application is to be inserted as a leaf node into the search tree for the given category. That is, allocate a node of type `struct tree`. The node containing a structure of type `struct app_info`; initialize the structure. Search the array of categories, to find the position matching the category of the application. Use the corresponding pointer to the root of a search tree for the given category to start insertion of the application into the search tree. Now insert the node as leaf into the search tree for that application category.

In addition, you will 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 information are to be stored in the hash table (not any of the other application information). 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` 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
};
```

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, would be 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, your `myAppStore` application is to collect some characteristics of the data structures, and then terminate gracefully. Graceful termination means your program must deallocate

all dynamically allocated data structures created including the search tree associated with each application category, the array of categories, the chains in the hash table, and the hash table itself before it terminates.

In the following, <> bracket variables, while the other strings are literals. There are 5 valid 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 follows the pointer to the node in the search tree to print the information associated with the application (i.e., the contents of the struct app_info, with each field labelled on a separate line); otherwise it prints Application not found.

2. find category <category_name>, searches the array of categories to find the given <category_name>. If found, an in-order traversal of the search tree is performed, printing the names of the applications in the given category (i.e., this will result in a list of applications of the given category sorted alphabetically by app name); otherwise it prints Category not found.

3. find price free, for each category, performs an in-order traversal of the search tree, printing the names of the applications whose price is free. Organize your output by category. 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 names of the applications whose price is greater than or equal to (float) <low> and less than or equal to)(float) <high>. If no applications are found whose price is in the given range print No applications found for given range.

5. range <category_name> app <low> <high>, for the given <category_name>, performs an in-order traversal of the search tree, printing the 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>. If no applications are found whose name is in the given range print No applications found for given range.

There is only one valid 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 deletes the entry from the hash table, and also from the search tree of the given <category_name>. If the application is not found it prints Application not found; unable to delete.

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 provide 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 info on the second app
FIFA 16 Ultimate Team
```

```
2.0
1.25
GB
0.00
Games           // Start of info on the third app
Candy Crush Soda Saga
1.50.8
61.3
MB
0.00
Games           // Start of info on 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
```

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 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!**

2. Once you have processed all queries and/or updates, collect and output the characteristics of the data structures you've built as described in §3.

3. Provide a `Makefile` that compiles your program into an executable named `p2`. 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 Blackboard; use them to test the correctness of your program.

3 Characteristics of the Data Structures

For the binary search tree associated with each category:

1. Print the category name.

2. Perform a in-order traversal of the tree by application name. For each node, print the application name, the height of its left subtree, the number of nodes in its left subtree, the height of its right subtree, and the number of nodes in its right subtree.

For the hash table:

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.

4 Submission Instructions

All submissions are electronic. This project has two submission deadlines.

1. The milestone deadline is before 11:59pm on Wednesday, 10/17/2018. See §4.1 for requirements.
2. The complete project deadline is before 11:59pm on Wednesday, 10/31/2018. See §4.2 for requirements.

For the milestone deadline, the array of categories, with a *binary search tree* for each categories must be implemented. In addition, the four queries that operate on the search tree are to be implemented for the milestone.

For the full project deadline, the `hash table` implemented using separate chaining is introduced as well as support for the remaining query and the update in `myAppStore`. In addition, after processing all the queries and updates, you must print the characteristics of the binary search tree and hash table as described in §3.

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 Blackboard!

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

4.1 Requirements for Milestone Deadline

For the milestone deadline, the array of categories, with a `binary search tree` for each category must be implemented for `myAppStore`. In addition, the queries that operate on the tree are to be implemented. Specifically, you must support the following four queries: `find category <category_name>`, `find price free`, `range <category_name> price <low> <high>`, and `range <category_name> app <low> <high>`.

Submit electronically, before 11:59pm on Wednesday, 10/17/2018 using the submission link on Blackboard for the Project #2 milestone, a zip[1] file named `yourFirstName-yourLastName.zip` containing the following:

Project State (5%): In a folder (directory) named `State` provide a brief report (.pdf preferred) that addresses the following:

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.

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

1. In one or more files, your well documented C/C++ source code implementing the array of categories, and binary search trees for each category. There are four queries to be implemented for this project milestone.

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

2. A `Makefile` that compiles your program to an executable named `p2` 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 p2` in the `Code` directory must produce the executable `p2` also located in the `Code` directory.

Correctness (45%): 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 Blackboard prior to the deadline for testing purposes. For the milestone deadline, these will only contain queries of the form described above.

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.

4.2 Requirements for Complete Project Deadline

For the complete project, you must now add the implementation of the hash table using separate chaining, linking the entries to those in your binary search tree. The remaining `find app <app_name>` query and `delete <category_name> <app_name>` update must also be implemented.

In addition, once you have processed all queries and/or updates, collect and output the characteristics of the data structures you've built as described in §3. Your program should then terminate gracefully (see §1.2).

Submit electronically, before 11:59pm on Monday, 10/31/2018 using the submission link on Blackboard for the complete Project #2, a zip[2] file named `yourFirstName-yourLastName.zip` containing the following:

Project State (10%): Follow the same instructions for Project State as in §4.1.

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

Correctness (50%): 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 includes characteristics of the data structures constructed.

5 Marking Guide

The project milestone 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` and `Code` is provided).

Implementation (50%): 50% for the quality of implementation in your code including proper memory management, construction of the binary search trees, and query processing, 10% for a correct `Makefile`.

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

The full project 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).

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

Implementation (40%): 40% 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, 10% for a correct `Makefile`.

Correctness (50%): 45% for correct output on several files of sample input including characteristics of the data structures, 5% for redirection from standard input.