ARIZONA STATE UNIVERSITY

CSE 310, All Sections — **Data Structures and Algorithms** — Fall 2015

Project #2

Available 09/23/2015; milestone due Friday, 10/16/2015; full project due Wednesday, 10/28/2015

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

As before, 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 queries and updates. A *binary search tree* is to be implemented for the milestone deadline, and a *2-3 tree* is to be implemented for the full project deadline.

Once `myAppStore` has been populated with data about applications, it will then process queries about the data and/or updates in the `myAppStore`.

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 of that category. That is, each category maintains a search tree of applications in that category. For example, if $n = 3$, then the three categories might be "Games," "Medical," and "Social Networking" each with a pointer to the root of a search tree for applications of that category.

```
#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
};
```

For the milestone deadline, the tree is a binary search tree ordered on the name of the application. Hence, 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 name, category, version, size, units, and price are provided.

```
#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 first be inserted as a leaf node into the search tree for the given category. That is, allocate a node of type `struct tree`, containing a structure of type `struct app_info`, and 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, insert into a hash table keyed by `app_name`. 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 the full application information). The hash table is to be implemented using *open addressing with linear probing*, with a table size k that is the first prime number greater than $2 \times m$. 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
};
```

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, are to be inserted into position 3 of the hash table if position 3 is empty.

If there is a collision, the hash table is probed linearly until an empty position is found. With *linear probing*, the positions of the table are simply probed in sequence in a circular fashion. That is for the example, if there was a collision in position 3, check positions $4, 5, \ldots, 10, 0, \ldots 2$ successively for an empty position. If the hash table is full, then `myAppStore` should exit gracefully and print `Hash table full, myAppstore exiting.`

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 must terminate gracefully. This means it must deallocate all dynamically allocated data structures created including the search tree associated with each application category, the array of categories, and the hash table.

Valid queries of `myAppStore` include:

- `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.`

- `find category category_name`, searches the array of categories to find the given category. 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 by name); otherwise it prints `Category not found.`
- `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.`
- `range category price low high`, for the given `category`, performs an in-order traversal of the search tree, printing the names of the applications whose price is greater than or equal to `low` and less than or equal to `high`. If no applications are found whose price is in the given range print `No applications found for given range.`
- `range category app low high`, for the given `category`, 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 `low` and less than or equal to `high`. If no applications are found whose name is in the given range print `No applications found for given range.`

Valid updates to `myAppStore` include:

- `delete category app_name`, first searches the hash table for the application with name `app_name`, and retrieves its category. Then it deletes the entry from the hash table, and also from the search tree of the given category. If the application is not found it prints `Application not found; unable to delete.`

1.3 Sample Input

The following is a set of sample input you `myAppStore` application must process. (The comments are not part of the input.)

```
3               // First initialize the categories of categories, n=3
Games           // The next n=3 lines contains the names of the n categories
Medical
Social Networking
4               // Number of apps to add to myAppStore, m=4; here all in Games
Games
Minecraft: Pocket Edition
0.12.1
24.1
MB
6.99
Games
FIFA 16 Ultimate Team
2.0
1.25
GB
0.00
Games
Candy Crush Soda Saga
1.50.8
```

```
61.3
MB
0.00
Games
Game of Life Classic Edition
1.2.21
15.3
MB
0.99
8                            // Number of queries and/or updates, q=8
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
```

2 Program Requirements for Project #2

1. Write a C/C++ program that initializes `myAppStore` as described in §1.1, and then processes queries and updates as described in §1.2. **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. Provide a `Makefile` that compiles your program into an executable named `p2`. This executable must read from standard input directly, or from a file redirected from standard input (this should require no change to your program).

Sample input files for `myAppStore` will be provided on Blackboard; use them to test correctness.

3 Submission Instructions

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

1. The milestone deadline is before midnight on Friday, 10/16/2015.
2. The complete project deadline is before midnight on Wednesday, 10/28/2015.

The difference between the milestone and full deadline is only the implementation of the search tree. For the milestone deadline, a `binary search tree` is to be implemented. For the full project deadline, a `2-3 tree` is to be implemented. That is, the structure `tree` given on page 1 is used for the milestone deadline. The structure definition changes as follows for the full project deadline:

```
struct tree{ // A 2-3 tree
   struct app_info; // Information about the application
   struct tree *left;  // Pointer to the left subtree
   struct tree *mid; // Pointer to the middle subtree
   struct tree *right;  // Pointer to the right subtree
  };
```

A 2-3 tree is a type of balanced search tree; operations for insertion and deletion into a 2-3 tree, along with how it is rebalanced will be discussed in class well in advance of the deadline.

3.1 Requirements for Milestone Deadline

For the milestone deadline, your project must meet the requirements in §2 by implementing the `myAppStore` using a binary search tree for `struct tree`.

Submit electronically, before midnight on Friday, 10/16/2015 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 (.txt, .doc, .docx, .pdf) that addresses the following:

1. Describe any problems encountered in your implementation for this project milestone.
2. Describe any known bugs and/or incomplete command implementation for the project milestone.
3. Describe any significant collaboration with anyone (peers or otherwise) and/or clearly reference any external code bases used.

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

1. Your well documented C/C++ source code implementing `myAppStore` using a *binary search tree.*
2. A `Makefile` that compiles your program to an executable named `p2` on the Linux machine `general.asu.edu`. 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 your program with several sets of input, some of which will be provided to you on Blackboard prior to the deadline for testing purposes. Of utmost importance in this project is your management of dynamic memory. As described in §2, your program must be able to run commands read from standard input directly, or from a file redirected from standard input. **You must not use file operations to read the input!**

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

3.2 Requirements for Complete Project Deadline

For the complete project deadline, your project must meet the requirements in §2 by implementing the `myAppStore` using a 2-3 tree for `struct tree`.

Submit electronically, before midnight on Wednesday, 10/28/2015 using the submission link on Blackboard 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 §3.1.

Implementation (50%): Your well documented C/C++ source code implementing `myAppStore` using a *2-3 tree.* In addition, provide a `Makefile` as described in §3.1.

Correctness (45%): The same instructions for Correctness as in §3.1 apply except that now it is expected that your search tree is implemented as a 2-3 tree instead of a binary search tree.

4 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.

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

Implementation (50%): 40% for your code including proper memory management, 10% for a correct `Makefile`.

Correctness (45%): 40% for correct output from input files, lack of memory leaks, and graceful termination; 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.

Implementation (50%): 40% for your code including proper memory management, 10% for a correct `Makefile`.

Correctness (45%): 40% for correct output from input files, lack of memory leaks, and graceful termination; 5% for redirection from standard input.