Contents

CPSC 2430 Data Structures

Homework Assignments #4

22 points

Note: Students may complete this as either an individual or group project. For group projects (maximum 3 members), only one team member needs to submit the work.

1. Problem

You are tasked with building a **dictionary of cities**, where each entry stores the following information:

- City name
- Country name
- Population

The dictionary must support **efficient and dynamic insertion, deletion, and query operations** based on the city name and/or country name.

In this assignment, you will implement the dictionary class named **Dict** using a **Binary Search Tree (BST)**.

Each dictionary entry (or node) contains a city name, its corresponding country name, and population. Since most operations are based on both the city and country names, we will use the **combination of city name and country name as the key** for organizing nodes in the BST.

The comparison rule between two dictionary items is defined as follows:

1. Compare the city names in alphabetical order.
2. If the city names are identical, compare the country names in alphabetical order to break the tie.

Your BST-based dictionary must support all operations listed in **Table 1** below.

Function	**Description**
`Dict()`	**Default constructor:** Constructs an empty dictionary with no elements.
`~Dict()`	**Destructor:** Destroys all dictionary items and deallocates all dynamically allocated memory used by the Dict container.

Function	**Description**
`unsigned int size() const`	Returns the number of items currently stored in the dictionary.
`bool empty() const`	Returns true if the dictionary is empty; otherwise, returns false.
`bool insert(const std::string& city, const std::string& country, int population)`	Inserts a new city entry into the dictionary. Returns true if insertion is successful, increasing the dictionary size by one. Returns false without modifying the dictionary if: 1. Dynamic memory allocation fails, or 2. An entry with the same city and country names already exists.
`bool remove(const std::string& city, const std::string& country)`	Removes the dictionary item matching the given city and country names. Returns true if such an item exists and is successfully removed (decreasing the dictionary size by one). Returns false if no matching item exists.
`int get_population(const std::string& city, const std::string& country) const`	Returns the population of the specified city and country. Returns -1 if no matching dictionary item exists.
`int topk(int a, int b, std::string city_list[], std::string country_list[], int k) const`	Given an inclusive population range [a, b], retrieves up to *k* alphabetically smallest cities (according to the defined comparison rule) whose populations fall within that range. The matching city names are stored in city_list, and their corresponding country names are stored in country_list, aligned by index. The resulting list must be **sorted in alphabetically increasing order** based on the combination of city and country names. The function achieves this ordering by leveraging the BST's inherent search property rather than performing an explicit sort.

Function	Description
	The function returns the number of retrieved items.
`int bottomk(int a, int b, std::string city_list[], std::string country_list[], int k) const`	Given an inclusive population range [a, b], retrieves up to *k* alphabetically largest cities (according to the defined comparison rule) whose populations fall within that range. The matching city names are stored in city_list, and their corresponding country names are stored in country_list, aligned by index. The resulting list must be alphabetically ordered in **decreasing** order based on the combination of city and country names. The function achieves this ordering by leveraging the BST's inherent search property rather than performing an explicit sort. The function returns the number of retrieved items.
`bool next(const std::string& name, std::string& next_city, std::string& next_country, int& next_population) const`	Given an arbitrary city name (which can be a random name not included in the input file), finds the **first dictionary entry** whose city name is alphabetically greater than the given name. If found, stores the city name, country name, and population in the corresponding reference parameters and returns true; otherwise, returns false.

Notes

- You are **not required** to implement a copy constructor or assignment operator overloading.
- You may design your own BST node structure and add **private helper functions** in the `Dict` class to support the required public operations.

2. Input Data

You are provided with two city data files: **cities_unique.txt** and **morecities_unique.txt**, which differ in data size. **cities_unique.txt** contains 29 city entries and **morecities_unique.txt** contains 38601 city entries.

Each line in a file contains information for one city — the city name, country, and population — separated by commas. Sample entries are shown below:

```
Beijing,China,11106000
Bogota,Colombia,7772000
Buenos Aires,Argentina,12795000
Cairo,Egypt,11893000
Chicago,United States,8675982
```

You can use these data files to test and debug your `Dict` implementation.

3. Client/Driver Program: client.cpp

You are required to develop your own **client.cpp** file to test and debug your `Dict` implementation during development.

Additionally, a reference **client.cpp** is provided to perform comprehensive testing of your `Dict` implementation. It is strongly recommended that you use the reference client only after you are confident that your `Dict` implementation is functionally correct, as it is intended for final verification rather than initial debugging.

The TA will use the reference **client.cpp** to evaluate and grade your `Dict` implementation.

Notes: If you are unable to complete the implementation of any function in the `Dict` class, you must still include its definition in dict.cpp. Provide a single return statement in the function body based on its return type:

- For functions returning void, write `return;`
- For functions returning int, write `return -100;`
- For functions returning bool, write `return false;`

4. Using the Reference client.cpp

Please follow the instructions below to run the reference **client.cpp**. After compiling, the executable is named client. You can run it using the following command:

```
./client -i <input_file> [-p -n -t -b -r -c]
```

Options:

- -i <input_file>: Specifies the input city data file that the client program uses to build the dictionary (**cities_unique.txt** and **morecities_unique.txt**).
- If none of the options [-p -n -t -b -r -c] are specified, the client program will, by default, test the Dict constructor, destructor, insert, empty, and size functions.
- -p: Additionally tests the get_population operation.
- -n: Additionally tests the next operation.
- -t: Additionally tests the topk operation.
- -b: Additionally tests the bottomk operation.
- -r: Additionally tests the remove operation.
- -c: Tests **all** Dict operations.

Except for -c, you can specify any subset of [-p -n -t -b -r], and the client program will test the corresponding operations. If you use -c, you do **not** need to specify any other options.

Examples:

- To test `topk` and `remove` operations in addition to the default functions (constructor, destructor, insert, empty, and size):

  ```
  ./client -i morecities_unique.txt -t -r
  ```

- To test all functions:

  ```
  ./client -i morecities_unique.txt -c
  ```

- To test the default functions:

  ```
  ./client -i morecities_unique.txt
  ```

5. The Makefile

You are required to provide a Makefile for this assignment. You may adapt the Makefile used in previous homework assignments, as the modifications should be straightforward.

6. README file

You must include a README file in plain text format.

The README file should contain the following information:

- Your name. If this is a team project, list all team members and clearly describe each member's contribution to the assignment.
- The status of your `Dict` implementation. Specifically, report the testing status of each function or feature corresponding to the options described in **Section 4**, based on results from the reference client.cpp.

An example README file is shown below:

Team members: Mike Smith, Tyler Bourne.
Mike Smith's Contributions: Implemented next, ...
Tyler Bourne's Contributions: Implemented topk, ...

Unit	Status
Basic (constructor, destructor, insert, size, empty)	Passed
get_population	Passed
next	Passed
topk	Passed
bottom	Failed
remove	Incomplete

7. Submission

You need to submit the following files:

- dict.h
- dict.cpp
- client.cpp: the provided client program.
- Makefile

Before submission, you should ensure your program has been compiled and tested (extensively) in cs1.seattleu.edu. Your assignment receives zero if your code cannot be compiled and executed.

You can submit your program multiple times before the deadline. The last submission will be used for grading.

To submit your assignment, you should follow two steps below (assuming your files are on cs1.seattleu.edu):

- Pack all your files into a package named ***hw4.tar***

  ```
  tar -cvf hw4.tar dict.h dict.cpp client.cpp Makefile  README
  ```

- Submit the package ***hw4.tar*** as the fourth homework assignment ***HW4***

  ```
  /home/fac/zhuy/class/submit2430 HW4 hw4.tar
  ```

The message similar to the following one will be displayed if the submission is successful.

```
=====================Copyright(C)Yingwu Zhu=========================
Sun Mar 28 23:01:52 PDT 2021
Welcome testzhuy!
You are submitting hw1.tar for assignment HW1.
Transferring file.....
Congrats! You have successfully submitted your assignment! Thank you!
Email: zhuy@seattleu.edu

=====================================================================
```

8. Grading Criteria

Label	**Notes**
[1] Submission (1 pt)	All required files are submitted.
[2] Makefile (1 pt)	Makefile compiles the code and generates the executable named client.
[3] Presentation (1 pt)	Clean, well-commented code. No unsolicited output messages such as testing & debugging messages in your Dict implementation.
[4] Basic operations (7 pts)	Default Constructor: 1 pt empty(): 1 pt size(): 1 pt insert: 2 pts Destructor: 2 pts
[6] -p: get_population operation	2 pts if it passed the testing.
[7] -n: next operation	2 pts if it passed the testing.

[8] -t: topk operation	2 pts if it passed the testing.
[9] -b: bottomk operation	2 pts if it passed the testing.
[10] -r: remove operation	2 pts if it passed the testing.
[11] Readme (2 pts)	Name(s) and contributions (if it is a team project): 1 pt. Status of testing on the Dict implementation: 1 pt.
[11] Overriding policy	If the code cannot be compiled or executed (segmentation faults instantly, for instance), it results in zero point for Functionality. No further investigation will be conducted on your program.
[12] Late submission	Please refer to the Syllabus for details.