CSE 100 // **PA2**

Implementing **dictionaries**, **autocomplete** and **spell checking**

Checkpoint deadline	**Friday, April 27 @ 11:59pm** → (**No late days** allowed for checkpoint)
Final deadline	**Friday, May the 4th be with you! @ 11:59pm**

Assignment Overview

Starter code: **https://goo.gl/xzDDWk**

In this assignment you will implement:

- a string lookup table using C++ STL implementations of a BST and HashTable.
- a Multi-way Trie or a Ternary Search Tree (TST) data structure for strings.
- autocomplete functionality seen in almost all text-based applications these days.
- spell checking functionality so you can be sure to never type *duck* wrong again!
- *(Extra Credit) a suffix array data structure for efficient pattern lookup in a string!*

From this PA onwards **your code will be graded upon submission!**

Set Up

Step 1 Get the starter code above.

Step 2 The reference dictionaries are located here **(please DO NOT upload them to Vocareum.)**
https://goo.gl/eTgyEH
You can copy them over to **ieng6** using **scp.** These are for your own benefit. We recommend you write tests using these to test your implementation.

Starter Code Overview

Provided files in repo : `Makefile, DictionaryTrie.h/cpp, DictionaryBST.h/cpp, DictionaryHashtable.h/cpp, util.h/cpp , test.cpp, SuffixArray.h/cpp`

`DictionaryBST.h` This file should contain the method *declarations* for building a dictionary using BST

`DictionaryBST.cpp` This file should contain the *implementation* for the method declarations provided in *`DictionaryBST.h`*

`DictionaryHashtable.h` This file should contain the method *declarations* for building a dictionary using *`HashTable`*

`DictionaryHashtable.cpp` This file should contain the *implementation* for the method declarations provided in *`DictionaryHashtable.h`*

`DictionaryTrie.h` This file should contain the method *declarations* for building a dictionary using a Multiway Trie or Ternary Search Trie. Any data structures that you plan to create can be added to this class. (ex. TrieNode)

`DictionaryTrie.cpp` This file should contain the *implementation* for the method declarations provided in *`DictionaryTrie.h`*

`util.h/cpp` This file contains a `load_dict` method that can load words into your dictionary data structure. It has been overloaded to support all three data structures (`DictionaryTrie`, `DictionaryBST`, `DictionaryHashtable`)

`SuffixArray.h` This file should contain the method *declarations* for building a suffix array. You'll only touch this file if you intend to complete the extra credit.

`SuffixArray.cpp` This file should contain the *implementation* for the method declarations provided in *`SuffixArray.h`*. You'll only touch this file if you intend to complete the extra credit.

`test.cpp` This file has a couple of basic checkpoint test cases . You are encouraged but not expected to add more in this.

`Makefile` contains instructions on how to build your files.

`Dictionary files` `freq1.txt, freq2.txt, freq3.txt (smaller dictionaries), shuffled_freq_dict.txt, shuffled_unique_freq_dict.txt, freq_dict.txt`

IMPORTANT NOTES

- **DO NOT HARDCODE ANY TEST CASE. You'll get a 0 (zero) on this PA if you do.**
- **DO NOT CREATE ANY new files to support the implementation of *DictionaryTrie***. If you want to create new classes to support any of the Dictionary classes (e.g. a TrieNode class), please place the class header and definition **in the existing** .h and .cpp
- **DO NOT EDIT ANY of the function signatures, or remove any of the provided code.** **However,** you are free to create any **private** member variables and member functions for the `DictionaryTrie` class to implement `predictCompletions` and `checkSpelling,` provided that you adhere to good object oriented design, and specifically use either a multi-way trie or a ternary trie to store the dictionary. In addition, you are free to add and implement any classes and methods in the provided `util.cpp/util.h` or `DictionaryTrie.h/cpp`.
- Make sure you have room in your account before copying files. Combined, the files for this PA are roughly 200 MB in size (because of the dictionaries).

CHECKPOINT [30 points]

At a glance Implement a Dictionary ADT using three different data structures

1) a balanced BST **2)** a Hash Table **3)** Multi-way Trie *or* Ternary Search Tree.

[5 points] **No segmentation faults**. That's all. Make sure you submit code that compiles and runs to completion, even if it doesn't always produce the expected output.

1 DictionaryBST [5 points]

✋ **Your code for this part must finish running each of our test cases in 60 seconds.**

This class must use a balanced binary search tree to store the words in the dictionary. The C++ STL set uses a balanced BST (a red-black tree, specifically) to store its elements, so all you need to do is use a C++ set to store the words in the dictionary. If implementing this class feels too simple, you're probably doing it right.

Implementation Checklist:

- Constructor
- Destructor
- `find` method (returns true if the word is in the dictionary and false otherwise)
- `insert` method (puts a word into the dictionary)

You may leave constructor/destructor empty depending on your implementation (see @531)

You will find it helpful to look in DictionaryBST.h for more thorough descriptions.

2 DictionaryHashtable [5 points]

✋ **Your code for this part must finish running ALL our test cases in 60 seconds.**

This class must use a hashtable data structure to store the words in the dictionary. The C++ STL unordered_set uses a hashset (i.e. a set implemented with a hashtable) to store its elements, so all you need to do is use an C++ unordered_set to store the words in the dictionary. If implementing this class feels too simple, you're probably doing it right.

Implementation Checklist:

- Constructor
- Destructor
- `find` method (returns true if the word is in the dictionary and false otherwise)
- `insert` method (puts a word into the dictionary)

You may leave constructor/destructor empty depending on your implementation (see @531)

You will find it helpful to look at `DictionaryHashtable.h` for more thorough descriptions.

3 DictionaryTrie [15 points]

✋ **Your code for this part must finish running each of our test cases in 60 seconds.**

This class **must** use either a **Multiway Trie or a Ternary Search Tree** to store the words in the dictionary, and **you must implement this data structure from scratch**.

Implementing this class will not be as trivial as implementing the other two. Note that you will likely need to implement additional classes and/or methods to support your Trie implementation (e.g. some kind of Node class). If you choose to do so, **you must** place their declarations and definitions **in the existing** .h and .cpp files.

You MUST NOT create any new files. If you add new files, your code will fail the test cases and you will get a 0.

Implementation Checklist:

- Constructor
- Destructor
- `find` method (returns true if the word is in the dictionary and false otherwise: it does not care about the word's frequency.)
- `insert` method (puts a word together with its frequency into the dictionary.
    - If a duplicate word inserted has a different frequency, update the frequency to the larger frequency but return false)

You may assume that no word in the dictionary will begin with, or end with, a space character.

You will find it helpful to look at `DictionaryTrie.h` for more thorough descriptions. Do not worry about implementing `predictCompletions` or `checkSpelling` for the checkpoint.

4 **Test your implementations** [Ungraded, but you should do it]

Add test cases to `test.cpp` and ensure that `make test` runs and works.

Tip: test your code on shuffled dictionaries (which may change the structure of a DictionaryTrie implemented as a Ternary Search Trie).

We have provided two dictionary files that you may use to test your implementation. (You may choose to not use them as well)

freq_dict.txt which contains about 4,000,000 English words and phrases (up to 5 words each) and their frequencies. Entries are limited to the lowercase letters a-z and the space character ' '.

unique_freq_dict.txt which contains about 200,000 words and their frequencies. Entries are limited to the lowercase letters a-z and the space character ' '.

Both dictionary files have the following format:

```
   freq_count_1 word_1
   freq_count_2 word_2
   ...
   freq_count_n word_n
```

In `util.cpp`, we have also provided a function named `load_dict` that loads the words from the stream (with frequencies if it is a `DictionaryTrie`) into the `Dictionary` object. We recommend you use it to load words from an open file. It is overloaded to take:

1. A reference to a `Dictionary` object (it is overloaded for all three `Dictionaries`)
2. An `istream`
3. *(optionally) a number of words to read in*

📬 **Required solution files**

- `DictionaryTrie.h/cpp`
- `DictionaryBST.h/cpp`
- `DictionaryHashtable.h/cpp`

DO NOT upload dictionary files to Vocareum.

Don't worry about `predictCompletions` and `checkSpelling` for now. You'll implement them for the Final Submission.

FINAL SUBMISSION [70 points]

At a glance Implement autocomplete and spell checking functionality.

[5 points] **No segmentation faults**. That's it. Make sure you submit code that compiles and runs to completion, even if it doesn't always produce the expected output.

[5 points] **No memory leaks**, i.e. the `valgrind --leak-check=full` report shows "`ERROR SUMMARY: 0`" when run on Vocareum. Make sure you submit code that cleans up after itself.

1 Implement autocomplete [30 points]

At a glance Implement the `predictCompletions` method in the `DictionaryTrie` class to enable autocomplete functionality.

✋ **Your code for this part must finish running each of our test cases in 60 seconds.**

Implementation Checklist **`predictCompletions`** returns a **`vector`** containing the **`num_completions`** most frequent legal completions of the prefix string ordered from most frequent to least frequent). See requirements list below and `DictionaryTrie.h` for more details.

An example of how `predictCompletions` works... Suppose `dictionary.txt` contains:

```
step 510
step up 500
steward 200
steer 100
stealth 100
```

If `prefix = "ste"` and `num_completions = 4`, `predictCompletions` can return either

```
<"step", "step up", "steward", "steer">
     OR
<"step", "step up", "steward", "stealth">
```

depending upon how it decided to break ties for equivalent frequencies.

Requirements

1. the empty prefix (" ") must return an empty vector. Also, if there are no legal completions, you should return an empty vector.
2. If the number of legal completions\\ is less than that specified by `num_completions`, your function should return as many valid completions as possible.
3. Invalid inputs should print an error message that reads "Invalid Input. Please retry with correct input" and return an empty vector. Invalid inputs include the following:
    a. a prefix is an empty string
    b. prefix is a string that's not in the dictionary\'

Our submission scripts use `shuffled_unique_freq_dict.txt` for `predictCompletions`.

We have provictionary file `smalldictionary.txt` for you to test the corner cases. We strongly encourage and recommend you to come up with more such smaller dictionaries to test various corner cases. This will help in debugging as well **(but your tests won't be graded.)**

Hints **Implementing `predictCompletions` (i.e., autocomplete):**

The autocomplete functionality requires two key steps:

1. Find the given prefix (use the same logic as the find function). This step allows you to progress up to some point in the trie.
2. Search through the subtree rooted at the end of the prefix to find the num_completion most likely completions of the prefix.

For step 2, an exhaustive search through the subtree is a simple approach. One possible exhaustive search implementation would use the breadth first search algorithm. You'll need a queue, which you can produce by using the C++ STL's `std::queue` data structure and always add nodes to the back (push_back) and remove them from the front (pop_front), and then keep track of the num_completion most frequent legal words you have seen along the way. This is a good starting point. While implementing exhaustive search is sufficient for this PA, brainstorming and thinking about techniques/algorithms that would prevent exhaustive search and return the completions faster is good food for thought.

2 Implement spell checking [30 points]

At a glance Implement the `checkSpelling` method in the `DictionaryTrie` class to enable spell checking functionality.

✋ **Your code for this part must finish running each of our test cases in 60 seconds.**

If the word is spelled correctly (i.e. it was found in the trie), simply return it. If the word is misspelled, however, you should return the best potential alternative (see below.)

In case of a misspelling, you must use the trie to come up with some suggestion of the word they may have intended, i.e. the *query*. The returned suggestion must be **the string in the trie with the <u>lowest</u> <u>Hamming distance</u> to the query**. The Hamming distance simply computes the number of characters in each string that are mismatched. If there's a tie, pick the word with the highest frequency. Finally, if you return one out of the five closest matches, but not the best match, you'll still get some points (but not all.)

As the Hamming distance is only defined for strings of same length, if there's no string of same length to the query, just return an empty string (i.e. no suggestion.) By definition, the Hamming distance between strings of different length is generally *infinity* (this may help in a recursive solution... be sure to just use a very large int, but not INT_MAX to avoid overflow.)

An example of how `checkSpelling` works… Suppose `dictionary.txt` contains:

```
stop 9001
star 250
atom 100
able 500
upon 420
step 510
step up 500
steward 200
steer 100
stone 130
steering 300
stealth 100
```

If `query = stoo`, `checkSpelling` should return `stop`, as their Hamming distance is equal to 1, vs at least 2 for all other 4-letter words. Moreover, if `query = stpmw`, `checkSpelling` should return `stone`, because both `stone` and `steer` have an equal Hamming distance to the query, but `stone` is more frequent.

An exhaustive search should suffice. A more involved implementation would use some form of dynamic programming; but whatever works. Our grading scripts will use the above set of short strings. **Reminder: if you hardcode any test cases you'll get a 0 (zero) on this entire PA.**

3 Test your implementations [Ungraded, but you should do it]

- Add test cases in **`test.cpp`** and ensure that the **`make test`** runs and works. You may use the same dictionary files and util functions as the checkpoint.

📫 Required solution files

- `DictionaryTrie.h/cpp`
- `DictionaryBST.h/cpp`
- `DictionaryHashtable.h/cpp`

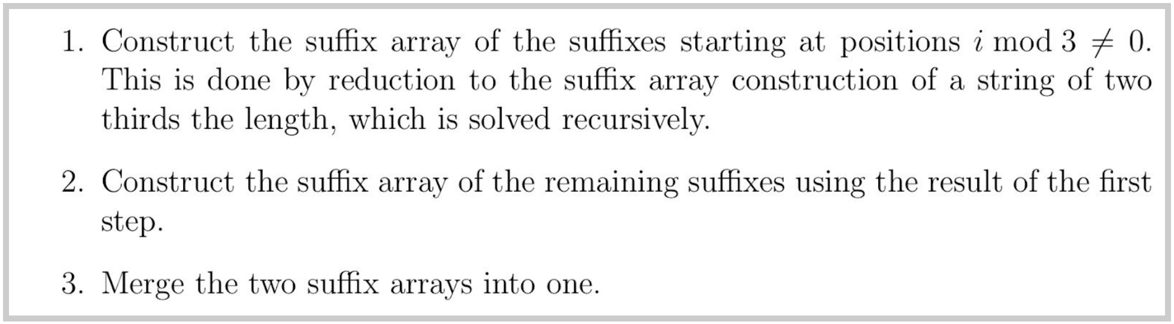
DO NOT upload dictionary files to Vocareum.

EXTRA CREDIT [10 points]

Suffix arrays and the DC3 algorithm

Suffix arrays *[1, see refs. below]* are an efficient data structure used to encode suffix trees, i.e. a trie with all the suffixes for a given string. Suffix trees or suffix tries *[2]* are often used in computational biology *[3] [4]*, as they enable efficient lookup of patterns in a string. That's handy if you're processing a long DNA sequence for applications such as contamination detection and sequence alignment. The suffix array **MUST** be represented as an `int` array storing the **indices of the suffixes in sorted order**. *[see 1]*

Now, it's trivial for one to build a suffix array by enumerating and then sorting all suffixes of a string. However, that's very costly: such algorithm would have time complexity of $\Theta(m^2\ log\ m)$, where m is the length of the string. The DC3 algorithm *[1] [5]* is a linear time procedure for building suffix arrays. It consists of three steps:



1. Construct the suffix array of the suffixes starting at positions $i \bmod 3 \neq 0$. This is done by reduction to the suffix array construction of a string of two thirds the length, which is solved recursively.
2. Construct the suffix array of the remaining suffixes using the result of the first step.
3. Merge the two suffix arrays into one.

Taken from [5].

Implementation checklist:

- [3 points] Fill out the method `buildSuffixArray` in `SuffixArray.cpp` (implement the brute force approach or the DC3 algorithm; unless you know better ways!)
- [5 points] Build the suffix array from **a very long** string (**`5000+`** characters) **in under 5s.**
- [2 points] Implement the `find` method, similar to how you did in the Checkpoint. Regular 60s timeout restriction applies to the test case for the `find` method.

Required solution files **DO NOT upload dictionary files to Vocareum.**

- `All files from before.`
- `SuffixArray.h/cpp`

Supplemental material

[1] Introduction to Suffix Arrays and the DC3 algorithm:
http://web.stanford.edu/class/archive/cs/cs166/cs166.1166/lectures/04/Slides04.pdf
[2] Lecture on "Suffix Tries" form UCSD's Coursera Course "Algorithms on Strings":
https://www.coursera.org/learn/algorithms-on-strings/lecture/V1AYj/herding-text-into-suffix-trie
[3] "Suffix Trees in Computational Biology".
https://homepage.usask.ca/~ctl271/857/suffix_tree.shtml
[4] "Applications of Suffix Trees"
http://www.cs.uku.fi/~kilpelai/BSA05/lectures/slides08.pdf
[5] Original paper for the DC3 algorithm:
https://www.cs.helsinki.fi/u/tpkarkka/publications/jacm05-revised.pdf