CSE 100 PA 3: Autocomplete and Benchmarking Performance

These days, apps like Google Chrome that use text have the ability to spellcheck and autocomplete--the ability to predict the word you are trying to type. In this assignment, you will implement this function and be able to test your code with a provided GUI. You will be able to see your Autocompletion happen on your computer!

Notice the differences of this assignment with both of the previous ones. You will be implementing the data structure and the algorithms as well as doing experiments, analyzing the results, and reporting the findings. You have a larger role in designing your implementation (even less pseudocode given). Make sure you finish the data structure and algorithms early enough to have enough time to conduct performance tests. As you are aware, time is short. You are able to start on the assignment in Part 0 and other testing tasks before understanding the data structures required. Just in case you wanted to study ahead, here are Prof. Kube's Slides on the MWT and TST. Plan your algorithm before you try to code. Keep a strong grasp of every task you are required to do for this assignment!

This PA will be partly an implementation assignment and partly a benchmarking assignment. You will implement a Dictionary using the multiway trie or the ternary trie. Your implementation will include a time-efficient auto-complete program that will find the x most frequently occurring completions of a given prefix, where x is the number desired. You will test the performance of your trie with two other dictionaries from the C++ STL.

Table of Contents:

File Information

The public starter files are located in /home/linux/ieng6/cs100v/public/pa3/

Copy the starter files to your workspace. There are two additional folders. You will need to copy those folders as well.

Provided Starter Files:

- **Makefile**: the usual makefile. You may need to add the benchtrie.cpp in.
- **DictionaryTrie.hpp/.cpp:** where you will implement the multiway or ternary trie AND any additional classes that are needed by the DictionaryTrie. Do not create new files for submission.
- **DictionaryBST.hpp/.cpp**: a dictionary that uses the Red Black Tree as the data structure.
- **DictionaryHashtable.hpp/.cpp**: a dictionary that uses the hashset as the data structure.
- util.hpp/.cpp: the provided tools for reading from file to any of the three dictionaries. Do not remove or edit any of the provided code, but you may add and implement new variables and functions to these files.
- **Benchtrie.cpp**: used for benchmarking and an example benchmark program
- **Dictionary (folder)**: the folder that contains large test input files for your Dictionaries.
- **autoCompleter_GUI (folder)**: contains the GUI files that can go with your AutoComplete algorithm.
- **createSpreadSheetChart.html:** html file that can be used to create a Google Chart out of a Google sheets page.

Notes:

1. Unlike previous PAs or proper C++ style, **any additional classes you implement must be placed in one of the files provided. If the class is related to the DictionaryTrie, it must be placed inside DictionaryTrie.cpp/.hpp.**
2. Any additional classes, functions, variables that does not need to be public must be private.
3. Don't use file names other than the ones given in the Makefile.

4. You will need to add benchdict.cpp when you're ready to compile it.
5. All style guidelines from previous courses apply.

Final Submission [1 Slip Day Allowed]

Due: Sunday, August 21th, 2016, 11:59PM

Required Submission Files:

util.hpp/.cpp, DictionaryBST.hpp/.cpp, DictionaryHashtable.hpp/.cpp, DictionaryTrie.hpp/.cpp, **FinalReport.pdf**, **benchdict.cpp**

Part 0: Dictionaries, Utilities, and Testing

There are three implementations of the Dictionary: the DictionaryHashtable, DictionaryBST, and the DictionaryTrie. The Hashtable uses the hashset, which is called the `unordered_set` in C++. The BST uses a balanced binary search tree, specifically the Red-Black Tree, which is called the `set` in C++. The DictionaryTrie is your implemented Dictionary. More about it in the next part.

Each data structure will have a constructor, destructor, find(), and insert(). In the hashtable and BST, insert puts a word into the dictionary and find returns true if the word is in the dictionary. We have provided the implementations of the Hashtable and the BST.

We have also provided an overloaded utility function named load_dict in the util file called util.cpp. load_dict takes a reference to a Dictionary object and an istream, and optionally, a number a words to read in (this is also overloaded), and load the words from the stream (with frequencies if it is a DictionaryTrie) into the Dictionary object. So once you create an empty Dictionary, you can use this method to load it with the words from an open file.

Write a program that tests the functions, find and insert, in the balanced BST and the Hashtable. Make sure your testers work, and then use them when you are implementing the Trie. (This is test-driven development, but made easier because you have two already-working data structures and the capabilities of the Trie are very similar.) We won't collect anything from this part, but it will be useful for getting familiar with the Util class and for you to test your implementations of your DictionaryTrie (with the exception of the predictCompletions).

We have provided two dictionary files for you to use to test your implementation: freq_dict.txt which contains about 4,000,000 English words and phrases (up to 5 words each) and their frequencies and unique_freq_dict.txt which contains about 200,000 words and their frequencies. **Each entry in either dictionary will be limited to the lowercase letters a-z and the space character ' ' (starting with a non-space character).**

We will be testing using shuffled dictionaries.

Part 1: Implementation

Your DictionaryTrie implementation will use the Multiway Trie or the Ternary Search Trie to store words in the dictionary, and you must implement this data structure from scratch.

DictionaryTrie's Find, Insert, and predictCompletions

The trie's abilities are slightly different. Insert will put a word *and* the word's frequency into the dictionary. Find will have the same function as previously and does not consider the frequency.

The trie will have the Autocomplete function, predictCompletions(). This function takes in as input a prefix string and a maximum number of completions. It returns a vector containing num_completions most frequent valid completions of the prefix string from most frequent to least frequent. All legal completions must be valid words in the dictionary. If the prefix itself is a valid word, it should be included in the list of returned words **if** its frequency is among the top num_completions most frequent valid completions.

The vector of completions must contain all of the most frequent completions, but when there are ties, it may break them up in any order. For example, if the prefix string is "ste" and only the words "step", "step up", "steward", "steer" and "stealth" are in the dictionary with frequencies (500, 500, 200, 100 and 100 respectively), when the user asks for 4 completions, the vector must include "step", "step up" and "steward", (in that order) but may include either the word "steer" or "stealth". In the case of tied frequencies, we will check to see if you have included either of the tied frequencies. You may assume that no word in the dictionary will begin with, or end with, a space character. You may also assumed that a valid input prefix is at least one character. You are responsible for handling invalid inputs, such as <=0 num_completions, empty strings, and strings that contain non-dictionary character.

If the number of legal completions is less than that specified by num_completions, your function should return as many valid completions as possible. If there are no legal completions, you should return an empty vector.

Reference Solution

We have provided you with a reference solution that implements this function using an exhaustive search technique. This is a fine starting point and will earn you many of the correctness points for this function. However, for the final submission you should consistently beat the running time by at least a factor of 10 on the specific test examples we provide, and

others with similar tree structure, meaning we will also test on similar-length prefixes that have subtrees of similar size to the examples we provide. If you beat the reference by at least a factor of 10 on our examples and you haven't done anything tricky to hard-code for just these solutions you should be fine. More details on timing your implementation is below in part 2.

To speed up your search, you may store additional information in each node (e.g. information about most frequent completion stored in the subtree of each child of the node), but this information must be a constant overhead over the information already stored in the node. In particular, **you may not simply cache (including using priority queues) all of the completions stored in the subtree of every node in the tree.** To be clear: you may not store multiple keys in a single node in the dictionary via some data structure or create multiple copies of the dictionary/keys via other data structures during insert. **You must design and implement an efficient Algorithm that will Search the Trie to return num_completions.**

Hints on implementing the autocomplete:

Note: If you don't want spoilers, skip to the next section on testing your code :)

The autocomplete functionality requires two key steps: 1) finding the given prefix (this should be easy as it uses the same logic as the find function). This step allows you to progress up to some point in the trie. 2) The next step is to 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 adding nodes to the back (push_back) and removing 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, and it is also the technique used in the reference code.

Unfortunately, this naive approach will not actually *beat* the reference solution, so you'll need to be more clever to earn your points for speed. You will have to design an efficient algorithm that will prevent us from exhaustively searching. As mentioned before, you are not allowed to cache multiple keys in a TrieNode or create multiple copies of the dictionary with other data structures or perform any other form of caching during insertion. We are not going to give you many hints on potential search algorithms. We expect you to be as creative as possible to come up with an efficient search algorithm. This is an opportunity to design your own algorithm and to implement it by choosing the appropriate supporting data structures (a key focus of this class).

Testing your code

Similarly to the previous tester you wrote for find and insert, write a tester that will test your Autocomplete's correctness. Use the input files and the utility functions provided. In addition, you are free to add and implement any classes and methods in the provided util.cpp/util.hpp files. **DO NOT, however, edit any of the function signatures, or remove any of the provided code.**

In summary, predictCompletions returns a vector of the most frequent legal completions (or as many completions as possible) ordered from most frequent to least frequent, that follows the timing requirements detailed in part 2 below.

Testing your code with the provided GUI

We have provide you with a GUI to make testing your code more fun! You can also use it to see what your finished auto-complete would look like.

The GUI code on ieng6 is located in /home/linux/ieng6/cs100s/public/pa3/autoCompleter_GUI directory. You do not need to modify the code in this directory, so don't spend your time trying to understand it. Instead focus on using the code given to you to test your implementation and compare it against a reference implementation. To run the programs given to you, make sure that you ssh into one the ieng6 servers with X11 forwarding. More information on how to set up X11 forwarding on your ssh client can be found on this link: Remote login with X11 forwarding

There are two main programs that we would like you to use both of which are available in the autoCompleter_GUI directory:

1) A pre-built reference program that uses our reference implementation of the DictionaryTrie as the back end and the GUI as its front end. This program is called "refautocomplete" and it has been compiled for ieng6. You won't be able to run it on any other machine. To run it simply type the name of the program at the command prompt as follows (you might have to change the permission of refautocomplete to make it an executable. You can do this by using 'chmod 777 refautocomplete'):

```
>./refautocomplete
```

It might take about a minute for the GUI window to pop up. This program uses the dictionary "freq_dict.txt" which is in the parent directory. After about a minute, you should see a window

with an interface similar to the Google search bar. Type words or phrases to see the reference autocomplete program in action.

If the program aborts abruptly, try running it with a smaller dictionary. **We have provided you with three smaller dictionaries on ieng6 under the folder cs100s/public/pa3/dictionary/ called freq1.txt, freq2.txt, freq3.txt**. Copy the dictionary of your choice to the parent directory of autoCompleter_GUI and rename it as "freq_dict.txt". Then run the reference GUI program as instructed above.

Note that even though you may run the GUI with smaller dictionaries, your DictionaryTrie should run successfully with the dictionary "freq_dict.txt" provided to you in the skeleton code and any other dictionary of similar size.

2)To run the GUI with your implementation of the DictionaryTrie follow the steps below:

a) Use qmake to generate a Makefile by typing in the following command:

> qmake Autocomplete.pro

Note: the Makefile in the working directory will be overwritten. If you have a Makefile to be used later in the current directory that is not for the GUI (aka not generated by the qmake), make sure to save it as something else or move it to another directory.

b) Compile as usual by running make

> make

3) Run the GUI executable

> ./Autocomplete&

It takes time to load the dictionary (roughly 1 minute), so please be patient if the GUI doesn't pop up at once. It also takes a while to exit (roughly 30 seconds). You may want to use the ampersand ("&") above to prevent your terminal from hanging up on you. Please find more detailed instructions in Compile Guide.txt.

Note that the Autocomplete program calls your implementation of predictCompletions();

We are NOT grading the GUI, or your ability to get the GUI working. The GUI is just for your benefit, and because we thought it would be cool to have, so you can see your own algorithm in action.

Part 2: Benchmarking and Analysis

Timing predictCompletions

The file benchtrie.cpp implements a program that times your implementation of the predictCompletions method in DictionaryTrie. You can use this program to see if your solution beats the reference solution. However, more importantly, you must use this program as a template for benchmarking in general and in writing benchdict.cpp.
To test the runtime of the reference solution, refbenchtrie has been provided to you. It will run 5 tests using our reference solution and output the times taken. The same 5 tests are already provided for you in the benchtrie main method. Note that the testers will take a long time to build the dictionary using the freq_dict.txt file.

For convenience, you can run the testagainstref script. This script will run both the refbenchtrie and benchtrie together. It should take 1 parameter which is the filename of the dict. E.g. ./testagainstref freq_dict.txt. This is only testing the runtime and NOT correctness. Make sure you have tests to ensure your solution is working correctly.

In terms of timing, these are the cases we are definitely testing for:

- The first 10 completions of each letter of the alphabet: We expect your implementation to be at least 10 times faster than the reference implementation when searching for the first 10 completions of ANY letter of the alphabet. This includes letters with many completions (e.g. "a" and "t"), and letters with relatively few completions (e.g. "q" and "x").
- The sum of the first 10 completions of every letter of the alphabet: We expect your implementation to be at least 100 times faster than the reference implementation when searching for the first 10 completions of ALL letters of the alphabet. What this means is if we looped from "a" to "z" and returned the first 10 completions of each of them, the total run-time should be 1/100th of the reference (total) run-time.
- If any test case takes longer than a minute to complete, you will likely lose some or all points for that case.
- Note that this isn't an exhaustive list of what we will test for in terms of timing: We may test for other prefixes. For *timing* purposes, we will not test on any prefixes longer than 5 characters (including spaces), and any prefix we test for timing purposes will be in the tree, and will have at least 100 completions. For example, if "troll" is not a prefix in the dictionary, or if it was but only has 2 completions, we will not test that prefix for timing. We expect such prefixes to beat the reference code by a factor of 2 when searching for the first 10 completions.

Benchmarking the three dictionaries

In this part, create a benchdict program to explore the running time of find in all of your dictionary classes. You must extend this program so that it does the following:

It should take four arguments as input:

```
./benchdict min_size step_size num_iterations dictfile
```

- The minimum size of the dictionary you want to test
- The step size (how much to increase the dictionary size each iteration)
- The number of iterations (e.g. how many times do you want to increase your dictionary size
- The name of a dictionary file to use

For each of your Dictionary classes:

Print a message to standard out to indicate which dictionary class you are benchmarking.

For num_iterations iterations:

1. Create a new dictionary object of that class and load min_size + i*step_size (where i is the current iteration) words from the dictionary file. If there are fewer than min_size+i*step_size words in the file, it should print a warning, but continue to run. Note that there is a load_dict function already implemented that allows you to load dictionaries of a certain size in the provided Utils class in util.cpp/hpp.

2. Compute the worst case time to find ten words in each of the three dictionaries by looking for words that are not present in the dictionary. Repeat this step many times (exactly how many is up to you) to get an *average* value for the run times.

A timer class has been provided to you inside the util.hpp/cpp files. Simply call void begin_timer() to begin the timer and int end_timer()to end it. The end_timer() function will return the duration in nanoseconds.

3. Print the dictsize and the running time you calculated in step two to standard out, separated by a tab (\t)

The idea is that this program will print out three tables. Here is an example (with totally made-up running times) of what your output would look like if min_size was 1000, step_size was 1000, and num_iterations was 5:

```
DictionaryTrie
1000    45
2000    49
3000    76
4000    100
5000    123

DictionaryBST
1000    41
2000    41
3000    745
4000    1000
5000    1293

DictionaryHashtable

1000    40
2000    400
3000    89
4000    900
5000    2000
```

Again, note that the timing numbers are completely made up above.

At minimum, you are required to have at least 15 iterations, minimum min_size 6000, and minimum min_step_size 1000. We recommend you experiment on which numbers you use: some sizes and steps will be more informative than others. The numbers above are likely NOT going to give you useful results.

Note: The dictionary file freq_dict.txt is very large, so if you plan on testing on the entirety of that dictionary, we highly recommend that you allocate all DictionaryBST, DictionaryHashtable, and DictionaryTrie objects on the heap. If any more than one dictionary data structure exists in memory simultaneously (even if both are allocated on the heap), your code is likely to crash due to a lack of memory. We also recommend that you delete each dictionary object before creating a new one in the same program.

FinalReport.pdf

Finally, create a file named FinalReport.pdf in which you reason about the running times for your three dictionary files.

Plot one graph with three lines for each dictionary using the data you collected and reason about the curves you see. Credits to Christoph for this method!

1. Save the output of ./benchdict into a TSV file (./benchdict ... ... > output.tsv)
2. Upload the TSV file onto your Google Drive.
    a. If you are working remotely without VNC, you might need to SCP the file to your computer first.
3. Open the TSV file using Google Sheets.
4. Rearrange the columns in this format with **any number of rows and 4 columns**:

Number	NameDictionary	NameDict	NameDict
#	#	#	#
#	#	#	#
...	...	...	...

Where # is a number.

5. Copy the URL of the Sheet.
6. Double click the provided HTML file called “createSpreadSheetChart.html” (should open in firefox/a web browser).
7. When prompted, paste the URL.
8. Copy the image of the graph using some screenshot tool.
9. Paste into wherever you are editing your FinalReport.

Then answer the following questions:

1. What running time function do you expect to see for the find method in each of your three dictionary classes as a function of N (number of words), C (number of unique characters in the dictionary) and/or D (word length), and why? This step requires mathematical analysis. Your function may depend on any combination of N, C, and D, or just one of them.
2. Are your results consistent with your expectations? If yes, justify how by making reference to your graphs. If not, explain why not and also explain why you think you did not get the results you expected.

3. Explain the algorithm that you used to implement the predictCompletions method in your dictionaryTrie class. What running time function do you expect to see for the predictCompletions as a function of N (number of words), C (number of unique characters in the dictionary) and/or D (word length), and why? This step also requires mathematical analysis. Your function may depend on any combination of N, C, and D, or just one of them.

NOTES FOR FinalReport.pdf:

1. You must create the .pdf document using one of the many word processors available to you, such as Google Docs, Microsoft Word, or LaTeX.
2. You are free to create the PDF content (the graph and the answers) any way you would like, including using paper, pencil, and camera. The instructions above for the automatic graph maker is a suggestion.
3. If you are using a camera, make sure to edit the photo so that your writing and drawings are visible.
4. Make sure that your explanations are clear, your writing/typing is legible, and your graphs are neat, organized, and fitting. Points will be deducted if the PDF is difficult to view.
5. Please be kind to the graders and be as concise and organized in your answers as possible. We want you to articulate, but we aren't looking for a novel by Charles Dickens.

That's it, you're done! You are now ready to submit your final solution.

Final Submission Task Summary:

- Implement predictCompletions to return num_completions most frequent legal completions of the prefix string ordered from most frequent to least frequent
- Improve the running time of your predictCompletions method to beat the given reference without compromising your memory and without using caching.
- Run benchmark tests on all your algorithms and submit your benchmarking code.
- Summarize your benchmarking results. Explain the algorithm you implemented for predictCompletions, and provide explanations for your results. Submit a writeup named "FinalReport.pdf"

Submitting on Vocareum - 1 Slip Day Allowed

Due: Sunday, August 21th, 2016, 11:59PM

Required Submission Files: util.hpp/.cpp, DictionaryBST.hpp/.cpp, DictionaryHashtable.hpp/.cpp, DictionaryTrie.hpp/.cpp, **FinalReport.pdf**, **benchdict.cpp**

Choose the files you want to submit. A script will run. **Make sure** that your submission at least:

1. Has all the correct names
2. Compiles
3. Runs without issues from the basic tester

Fix any errors that arise from Steps 1-3.
Only one of the programmers in a partnership need to submit, but both should verify that the submission works for both programmers.

You can submit once, ten times, any number of times before the deadline and only your last submission will be graded. If you submit within 24 hours after the deadline, you will be charged a slip day. Slip days cannot stack, so there will be no submissions allowed after 24 hours.

Don't rely on the Vocareum tester for test cases (though it's not a bad source for testing); your custom testers must check for corner cases.

Grading Guidelines

This PA is worth 30 points.

The checkpoint is worth 5 points:

1. 13 points for correctness of predictCompletions: It must return as many completions possible, up to num_completions, of the highest frequency completions in the DictionaryTrie, from highest frequency to lowest.
2. 1 point for beating the naive reference solution by any amount on any letter of the alphabet, and on valid prefixes in the trie up to 5 characters (including spaces)
3. 1 points for beating the naive reference solution by at least a factor of 10 on any letter of the alphabet, and by a factor of 2 on valid prefixes in the trie of up to 5 characters (including spaces)
4. 1 point for beating the naive reference solution by at least a factor of 100 when tested iteratively on all of the letters of the alphabet.
    a. You will lose 3 points if you cache instead of searching the subtree rooted at the prefix during predict-completions.

5. 3 points for the benchdict program
6. 5 points for FinalReport.pdf
7. 2 points for code that is free of memory leaks
8. 4 points for a good object-oriented design, good coding style, and informative, well-written comments.

Memory leaks will result in deductions. Each test case that results in a segfault will receive a 0. Compile and link errors will cause the entire assignment to receive a 0. Valgrind and GDB are highly recommended debugging tools to use.

Additional massive point deduction scenarios:

- predictCompletions returning an incorrect vector will result in a zero for the timing tests
- Using caching (for example, but not limited to, using priority queues in each of your nodes) will receive no credit for beating the reference.
- Avoiding the use of the multiway or ternary trie as the DictionaryTrie will receive no points.