Assignment Overview

In this assignment you will:

- Implement either a multi-way trie (MWT) or ternary search tree (TST) data structure for strings.
- Implement auto-complete functionality seen in almost all text-based applications these days.
- Compare the empirical running time of MWTs or TSTs, Balanced Binary Search Trees and Hash Tables to their expected analytical running times.
- Research and compare the performance of different hash functions for strings.

Set Up

Step 0: Sign up for groups before **Wednesday October 17 11:59 pm**. After this deadline, group registration/updates will be closed and no one could sign up for a group or change his/her group.

1. **Go to this link and add your group information before the deadline.**
2. **Go to TritionEd and add yourself to the group with the same group id as in the Google spreadsheet first column. (i.e "PA2 Group XX" and XX is the number from the Google spreadsheet).**

After the deadline you could add yourself to any empty group on TritionED but be aware that by that time you could only do the homework on your own and you will not be able to join someone else for PA2.

Step 1: The reference dictionaries are located on Piazza resources section, TritonED and ieng6. You could either download the zip file from Piazza or TritonED as well as using the following instruction and download it from ieng6.

The files on **ieng6** are located in:
/home/linux/ieng6/cs100f/cs100f4/public/pa2/dictionaries.zip

You can copy these dictionaries to your working dir by using the following command.
Make sure you have enough space.

```
cp -r  /home/linux/ieng6/cs100f/cs100f4/public/pa2/dictionaries.zip ./
```

Step 2: (Optional) We have also provided a GUI to test the part 2 of the assignment. The GUI files are also available on Piazza resources section, TritonED and also on **ieng6** in **/home/linux/ieng6/cs100f/cs100f4/public/pa2/autoCompleter_GUI.zip** directory.

You can copy the dir to your working dir by using the following command

```
cp -r /home/linux/ieng6/cs100f/cs100f4/public/pa2/autoCompleter_GUI.zip ./
```

With this you are all set to start working on the assignment.

Provided files in repo : Makefile, DictionaryTrie.h/cpp, DictionaryBST.h/cpp, DictionaryHashtable.h/cpp, util.h/cpp , test.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 timer methods that could be used to time you insert and find functions take. It also 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)

test.cpp - This file has a couple of basic checkpoint test cases . You are expected to add more in this.
Makefile - contains instructions on how to build your files.

In addition, you will create two more cpp files called **benchdict.cpp** and **benchhash.cpp** for final submission. benchdict.cpp should contains methods that will test the running times of the three data structures implemented as a part of this assignment. You will implement 2 good hash functions of your choice in benchhash.cpp and compare the performance of these two hash functions by calculating the number of collisions each produces on a dictionary file.

Files provided in ieng6 : dictionary: freq1.txt, freq2.txt, freq3.txt(smaller dictionaries), shuffled_freq_dict.txt, shuffled_unique_freq_dict.txt, freq_dict.txt

autoCompleter_GUI: This contains code that generates a GUI (not required for use, but created for your benefit): Autocomplete.pro, main.cpp, mainwindow.h/cpp/ui, mylineedit.h/cpp, wordlist.h/cpp, Compile Guide.txt

NOTES (IMPORTANT):

- **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 files.

- **DO NOT add any dictionary to your repository**. This may cause your repository to hang up unexpectedly and make your life (and ours) very difficult.
- Don't add executables and the GUI files to your repository.
- 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).
- **DO NOT edit any of the function signatures, or remove any of the provided code. However,** you are free to create any member variables and member functions for the DictionaryTrie class to implement predictCompletions, provided that you adhere to good object oriented design, and specifically either use 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.
- Style matters! In this PA, style and comments will take up 3 out of total 30 points. Do follow minimal style guide to ensure you will have the points.

1. **Implement a Dictionary ADT using three different data structures: a balanced BST, a Hash Table, and a Multi-way Trie *or* Ternary Search Tree.**

DictionaryBST

This class must use a balanced binary search tree to store the words in the dictionary. However, this should be easy, because 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 will find it helpful to look at the files DictionaryBST.h for more thorough descriptions.

DictionaryHashtable

This class must use a hashtable data structure to store the words in the dictionary. However, this should be easy as well, because 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 will find it helpful to look at the files DictionaryHashtable.h for more thorough descriptions.

DictionaryTrie

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, place their declarations and definitions in the existing .h and .cpp files. **You should not create any new files.** If you add new files, your code will not work with the provided GUI and will fail the test cases.

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

Test your part 1 implementations

Implementation Checklist:

- Add *more* test cases in 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 much choose to not use them as well) :

1. 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 ' '.
2. 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 load the words from the stream (with frequencies if it is a DictionaryTrie) into the Dictionary object. We recommend you to 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

2. **Implement Auto-Complete by implementing the predictCompletions method in the DictionaryTrie class.**

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 edge cases checklist, hint section, 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.

Edge Cases Checklist: (**Read Carefully**)

- Ties may be broken in arbitrary order.
- You may assume that no word in the dictionary will begin with, or end with, a space character.

- 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 legal completions.
- 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.
- The empty prefix must return an empty vector.
- 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 prefix is an empty string, prefix is a string that contain non-dictionary characters (character not in the dictionary).
- All legal completions must be words (or phrases) that appear in the loaded dictionary.
- If a duplicate word inserted has a different frequency, update the frequency to the larger frequency
- Think of more edge cases yourself!

Our Submission scripts use shuffled_unique_freq_dict.txt for PredictCompletions.
We have provided a small dictionary 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.

Hints on implementing predictCompletions (i.e., 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. Find 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. 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. Checkout the starpoint if you are interested in coming up with an algorithm that is much faster than the exhaustive search algorithms.

Test your part 2 implementations.

Implementation Checklist:

- Add test cases in test.cpp and ensure that the autocomplete functionality works. You may use the same dictionary files and util functions as the checkpoint.
- Test your code with the provided GUI. This document has the instructions to test your program with the GUI. **Note:** We are NOT grading the GUI, or your ability to get the GUI working. The GUI is just for your benefit, because we thought it would be cool to have so you can see your own algorithm in action. It is okay if you do not test your program with the GUI.

3. **Benchmark Dictionaries**

Implementation Checklist:

- Create the benchdict program (i.e., translate the purple text below into code).
- Run benchdict on shuffled_freq_dict.txt. See below for details.
- Produce 1 graph (using google sheets, excel, gnuplot, etc) for *each* dictionary type. Use the most reliable run produced by the previous step. See below for details.
- Create a FinalReport.pdf that contains the three graphs and a discussion of the results. Answer the specific question in blue below. See below for details.

3.1 Create a program named benchdict that does the following:

It should take four arguments as input: ./benchdict min_size step_size num_iterations dictfile

- ■ min_size - The minimum size of the dictionary you want to test
- ■ step_size - The step size (how much to increase the dictionary size each iteration)

- num_iterations - The number of iterations (e.g. how many times do you want to increase your dictionary size)
- dictfile - The name of a dictionary file to use

Then it should run the following algorithm:

For each Dictionary class (DictionaryBST, DictionaryHashtable, DictionaryTrie):

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

For i = 0 to num_iterations:

1. Create a new dictionary object of that class and load *min_size + i*step_size* words from the *beginning* of the dictionary file (you will need to reset the istream to the start of the dictionary file at the start of each iteration). Use the load_dict function from util.cpp!
2. Read the next 100 words from the dictionary file and then compute the time to find those 100 words in the dictionary object. They will not be there. Using the same 100 words, repeat the find process many times and take the average time of all the runs. Be sure to time only the part when you are looking for the words in the dictionary.
3. Print the dictsize and the running time you calculated in step two to standard out, separated by a tab (\t)

Delete dictionary object (to avoid code to crash for lack of memory)

Edge Case Checklist:

- If there are fewer than min_size+i*step_size words in the file, benchdict should print a warning message. We will not test you on these corner cases as benchmarking does not make sense if there are no enough words. However, you must handle these corner cases reasonably by printing some warning/error message.

3.2 Run your program on the shuffled_freq_dict.txt dictionary

Here is an example (the ### will be numbers representing the running time to find 100 words) where:

min_size = 6000, step_size = 1000, and num_iterations = 5 (**you'll want to use more iterations and possibly a larger step size or starting value**):

```
DictionaryTrie
6000    ###
7000    ###
8000    ###
```

```
9000    ###
10000    ###
DictionaryBST
6000    ###
7000    ###
8000    ###
9000    ###
10000   ###
DictionaryHashtable
6000    ###
7000    ###
8000    ###
9000    ###
10000   ###
```

At minimum, you are required to have at least 15 iterations, minimum min_size 5000, and minimum min_step_size 100. We recommend you experiment on which numbers you use: some sizes and steps will be more informative than others.

A timer class has been provided to you inside the util.h/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 since begin_timer() was called, in nanoseconds.

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 is exists in memory simultaneously (even if both are allocated on the heap), your code is likely to crash due to a lack of memory.

3.3 Plot your results in a graph

Using the most reliable run of 3.2, produce a graph (using google sheets, excel, gnuplot, etc) for each dictionary type. Plots are between Dictionary Size and Run Time.

3.4 Finally, create a file named FinalReport.pdf in which you reason about the running times for your three dictionary files. Include your three graphs (one for each dictionary type) using the data you collected and reason about the curves you see. Then, specifically answer the following question:

- In class we saw that a Hashtable has expected case time to find of O(1), a BST worst case O(logN) and a MWT worst case O(K), where K is the length of the longest string. We didn't look at the running time of the TST, though the book mentions that its average case time to find is O(logN) (and worst case O(N)). Are your empirical results consistent with these analytical running time 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 (also referencing your graphs).

4. Research and compare the performance of different hash functions for strings

Implementation Checklist:

- Find two different hash functions for strings (online, in textbooks, etc) that are non-trivial (i.e., they don't just return the same value for every string). **Note:** It is OK if you find the implementation of the hash function, just make sure you understand what it is doing.
- Implement the two hash functions in a file named benchhash.cpp and record the source of each hash function in a comment above the function. **Note:** The input to each function should be a string and a table size, and the output should be an unsigned int between 0 and the table size (not including table size).
- Test your two functions by calculating the expected hash values of several string by hand and then making sure that your functions return the expected output. You will report these test cases in your writeup.
- Understand how to compare the performance of two hash functions. See below for details.
- Add actual code to benchhash.cpp to compare the performance of the two hash functions. See below for details.
- Add the following to the FinalReport.pdf:
  a. Describe how each hash function works, and cite the source where you found this information. *Your description of how the hash function works should be in your own words*.
  b. Describe how you verified the correctness of each hash function's implementation. Describe at least 3 test cases you used, what value you expected for each hash function on each test case, and the process you used to verify that the functions gave this desired output.

c. Run your benchhash program multiple times with different data and include a table that summarizes the results of several runs of each hash function. Format the output nicely--don't just copy and paste from your program's output.
d. Comment on which hash function is better, and why, based on your output. Comment on whether this matched your expectation or not.
e. Read the "Notes for FinalReport.pdf" to make sure you followed all the guidelines.

- That's it, you're done!! You are now ready to submit your final solution. See the grading overview below.

Mini lesson on how to compare the performance of two hash functions:

You will do this by calculating the number of collisions each hash function produces. In other words, we will count the number of times each slot in the table was hashed to. Each time a slot is hashed to, we will say that that slot experienced a "hit." From that information, you can construct a histogram showing how many slots received zero hits (no item ever hashed to that slot), one hit (1 item hashed to that slot and the slot never faced a collision), two hits (the slot faced 1 collision), and so forth. You can also calculate the average and maximum number of steps that a find would take.

For example, given a table size of 2000, a collection of 1000 words in a dictionary, and a hash function that computes a nonnegative integer <2000 (table size) from a string, we might discover that

#hits	#slots receiving that #hits
0	1042
1	932
2	15
3	7
4	3
5	1

Assume that the hash table uses separate-chaining for collision resolution. From that information, we would then calculate how many steps are required to find a certain amount of words. For example:

- 1 word would require 5 steps (1).

        - The intuition behind this is that one hash table slot received 5 hits! That means that slot has a linked list of 5 elements and therefore there is 1 word in the back of that linked list that will require 5 steps (i.e., comparisons) to find it.
    - 4 words (3 + 1) would require 4 steps
        - The intuition behind this is that 3 slots received 4 hits! That means that the 3 slots have a linked list of 4 elements and each linked list therefore contains 1 word in that back that would require 4 steps (i.e., comparisons) to find it. *However,* don't forget about the 1 slot that received 5 hits! There is also 1 word in that linked list that is the 4th element in that 5 element linked list. Thus, that word would *also* take 4 steps to find and thus we have a total of 3 + 1 words that would require 4 steps.
    - 11 words (7 + 3 + 1) would require 3 steps
        - Same intuition as above. We must take into consideration the last elements in the 7 slots with 3 element linked lists, the 3rd elements in the slots with the 4 element linked lists, and the 3rd element in the slot with the 5 element linked list.
    - 26 words (15 + 7 + 3 + 1) would require 2 steps
    - 958 words (932 + 15 + 7 + 3 + 1) would require 1 step

From that information, we would then calculate that the average number of steps in a search would be about 1.064, since
(958*(1 step) + 26*(2 steps) + 11*(3 steps) + 4*(4 steps) + 1*(5 steps)) / (1000) = 1.064.

The worst case is the maximum number of steps that would be needed to find a word: 5 for the example above.

How to actually add code to your benchhash.cpp file:

Your benchhash function should take two arguments as input: ./benchhash dictfile num_words

- dictfile -- The name of a dictionary file to use (string)
- num_words -- The number of words to be inserted from the dictionary in the hashtable (unsigned int)

We will then "simulate" the hash function by taking the following steps (i.e., we won't actually need to create a hash table and insert strings into it!):

1. Choose the size of the simulated hash table to be 2 * num_words.
2. Calculate the hash value for each string from a dictionary of size num_words. Keep track of the number of "hits" each slot receives by creatively using a vector.
3. For each hash function print:
    a. The number of slots that received each number of hits, as described above
    b. The average number of steps in a successful search
    c. The max number of steps that would be needed to find a word in a hashtable

Here is an example (with totally made-up values) of what your output would look like if the number of words to be inserted from the dictionary in the hashtable is 1000:

```
./benchhash freq1.txt 1000
Printing the statistics for hashFunction1 with hash table size 2000
#hits    #slots receiving the #hits
0           1999
1000      1
The average number of steps for a successful search for hash function 1 would be 500.5
The worst case steps that would be needed to find a word is 1000
Printing the statistics for hashFunction2
#hits    #slots receiving the #hits
0           1042
1           932
2           15
3           7
4           3
5           1
The average number of steps for a successful search  for hash function 2 would be 1.064
The worst case steps that would be needed to find a word is 5
```

Notes for FinalReport.pdf:

1. Changing the file extension of a file from .txt to .pdf does NOT convert the plain text file to a pdf file. You must create the .pdf document using one of the many word processors available to you. If you're at a loss for what to use, I'd recommend using google docs, MS word, or LaTeX for all you fancy people out there.
2. You will be required to create graphs to present your data. If you're at a loss for what to use, we recommend using google spreadsheets or excel to create graphs you can copy and paste, or hand-drawing really nicely and copying the image from MS paint (we won't judge)--whatever suits you.
3. Naturally you must make sure that your explanations are clear, your writing/typing is legible, and your graphs are neat, organized, and fitting.
4. 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.

Grading Overview (30 points)

- 5 points for the correctness of insert and find functions of DictionaryBST, DictionaryHashtable, and DictionaryTrie
- 8 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 points for the benchdict program
- 3 points for the benchhash program
- 7 points for FinalReport.pdf
- 2 points for code that is free of memory leaks
- **3 points** for Commenting, style and a good object-oriented design (see the minimal style guide). This time we ask you to write code with inline comments, function headers, file headers etc., and please DO check the style guide to see if you fulfill what it says.

Some caveats:

- **IMPORTANT:** Tables and graphs of experimental results, and transcripts in your FinalReport.pdf, must be genuine and not misleading. It may happen that some of your code or algorithms do not work correctly. In this case you must mention and explain this situation in documentation and reports. Faking data/ Copying somebody else's plot is a serious violation of the academic Integrity Policy of CSE 100.
- Code that does not compile will not receive credit.
- Any code that does not compile, or segfaults, or does not create any objects will NOT receive credit for the memory leak test
- You will receive 0 points for style, and FinalReport.pdf if you do not implement a ternary search tree or a multi-way trie as your DictionaryTrie.
- Naturally, at this point, you will get a 0 for code that was not submitted.

Required solution files: Makefile, DictionaryTrie.h/cpp, DictionaryBST.h/cpp, DictionaryHashtable.h/cpp, util.h/cpp, benchhash.cpp , test.cpp, benchdict.cpp, FinalReport.pdf

Starpoint

If you're gotten here and you still want more, we're providing this "star point" option. However, this is just a suggestion. Any significant extension or project will be considered for a star point. If you're interested, keep reading.

First, what is a "star point"? Star points are challenging, open-ended extensions designed to engage those who really want to learn more and go beyond the basic requirements. However, these are not extra credit. If you do "enough" starpoint and are "close enough" to the boundary, you may be moved up to the higher grade, but do these star point extensions because you are intellectually curious and want a challenge. Not for the grade. The course staff will not answer questions like "If I do this, will I get a star point?". If you are doing the star point extension just to get the starpoint, then you're doing it for the wrong reason. Only do it if you would be happy whether or not you get the point in the end.

We have two suggested star points for this assignment. You can choose to do one or both of these depending on your interest.

1. Implement Efficient PredictCompletions that beats the reference solution

For implementing AutoComplete we have provided you with a reference solution that implements this function using an exhaustive search technique. While doing an exhaustive search is good enough for finishing the PA, to earn the starpoint you need to design and implement an algorithm that consistently beats the running time by at least a factor of 10 on the specific test examples we provide, and *others with similar tree structure.* What we mean by "similar tree structure" is that 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.

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.

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.

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 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.
NOTE: You cannot use caching (i.e) you cannot 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. Implementations that use caching will not receive the StarPoint.
In StarPoint.pdf explain the algorithm that you used for PredictCompletions with an example. You will also submit all your Dictionaryclasses and implementation.

2. Implement SpellCorrection

Extend your DictionaryTrie class to support Spell Suggestions. Your program spellCheck.cpp will allow the user to type in words to check the spelling of. If the word is spelled correctly (i.e. it was found in the trie), inform the user. If the word is misspelled, However, you should inform the user that they were incorrect, and provide a potential alternative. You must use the trie to come up with *some*suggestion of the word they may have intended. You can use the trie to come up with *some*suggestion of the word they may have intended, by simply returning a word that was found somewhere on the path to the word they were typing ("yesterday" might be a suggestion for the word "yesterdayz", for example). If no word exists on that path, you should inform the user that you have no suggestions for them.The suggestions being provided by the above method is fairly weak, and will probably not find suggestions for some common misspelling s. However, this can be remedied by expanding the search in the tree. Instead of focusing on the path from the root to the last node, you can start looking down other branches.
To determine if a word is a good suggestion, you can use various metrics. One possible metric is the hamming distance, which simply computes the number of characters in each string are mismatched. This is a fairly weak metric for spell checking, but likely will produce slightly better results than the above. Another metric is the Levenshtein distance, which better handles strings of different lengths. Each of these metrics measure how different two strings are. Implement one of the above algorithms, and provide a handful of better suggestions by ranking them using the implemented metric.

Spell Check a File: It is not very useful to have a stand alone program for spell checking. It would be better if you could specify a file to be spell checked, and perform this operation on that file. You must produce a new file, which is a spell checked version of the original file.This program should take, as a command line arguments, the dictionary file name AND a plain text file to spell check. Your program should build a trie as normal using the dictionary. It should then check each word sequentially in the file. For every misspelled word, you should provide suggestions and ask the user which suggestion should be used. For ease of use, one of the suggestions should be the "incorrectly" spelled word. After you have finished checking the specified file, you should write a new file with the string "_checked" appended to the end of the file name. This file will be essentially a copy of the original file, but with all of the spellings "corrected" by the users choices.

You will turn in a makefile that generates the executable spellcheckfile , spellcheck , starpoint.pdf and your code as part of your submission.

In starpoint.pdf explain the algorithm you used to provide suggestions. Also explain how you tested your program.