Project #3: Set the Controls for the Heart of the Sun

Due Aug 20 by 11:59pm **Points** 30 **Submitting** a file upload
File Types gz **Available** Jul 29 at 12am - Aug 25 at 3am 27 days

This assignment was locked Aug 25 at 3am.

Due date and time: *Sunday, 20th August, 2017 at 11:59pm*

Introduction

Search structures

In recent lectures, we've discussed several different data structures that can be used for the purposes of searching. Each of these data structures is organized differently, and each has different performance characteristics (measured in terms of time and memory usage). But they all share one goal: given a *key* that we're searching for, find that key and any information that's been associated with it.

Broadly, we say that a *set* is a kind of data structure that stores a collection of unique keys, allowing us to add new keys, remove existing keys, and determine whether a particular key is stored. We say that a *map* is a similar kind of data structure, except that we associate a value with each key, so that keys function as the tool that lets us identify other information uniquely and find it easily. In lecture, we discussed several approaches to solving the problems of implementing search structures like sets and maps.

- We talked about using single-dimension search structures like *arrays*, *vectors*, and *linked lists*, but all had significant limitations. Either they required linear time to search (by looking at every element in sequence), or they could be searched more quickly (e.g., using binary search) but required linear time to add or remove a key.
- We discovered that *binary search trees* offer the ability to do searches, insertions, and removals in logarithmic time, which is a significant improvement on the linear-time bound on at least some operations on single-dimension search structures. But there was a catch: We only got logarithmic-time operations if the tree stayed balanced, but it could fall out of balance in circumstances that weren't necessarily rare in practice (such as adding keys in ascending or descending order).
- We found that all was not lost, because there are well-known algorithms for detecting and correcting binary search trees that have fallen out of balance. In particular, we learned about a binary search tree variant called *AVL trees*, which guarantee logarithmic-time searches, adds, and removals, by rearranging the nodes in the tree whenever a significant imbalance is detected.
- We explored *skip lists*, a very different solution that allows a variant of binary search to be performed on a linked list. It employs randomness to achieve its goal of providing logarithmic-time searches, adds, and removals, by storing some keys in two places, a few others in three places, a handful of others in four

places, and so on. While skip lists don't actually guarantee logarithmic-time operations, if a good random number generator is used, skip lists have an extremely high probability of performing their operations in logarithmic-time, especially for large collections of keys.
- Finally, we discussed how *hash tables* offer the possibility of not having to search. Every key has a place where it "belongs," so if we only ever store keys where they belong, we'll never have to look anywhere except there. Of course, in practice, it doesn't turn out be quite as magical as that in practice, but it can perform very well, provided that we spread keys around so not too many of them belong in any one place.

As we often see when there are multiple solutions to the same kind of problem, there are important tradeoffs here. At first blush, hash tables with "good" hash functions appear to be the clear winner, as their search times tend toward Θ(1). But it's not always that simple. For example, you might not want to choose a hash table if you don't know enough about your keys to choose a suitable hash function. There's another reason, too, why you might prefer a balanced binary search tree or a skip list over a hash table: Since they impose ordering on their keys, binary search trees and skip lists can be iterated in ascending order of their keys in linear time. Hash tables can't, since they impose no meaningful ordering on their keys.

Spell checkers, as a case study of search structures

In this project, you'll build portions of a *spell checker*, similar to the one you might find in a word processor, a text editor, or an email client. A spell checker's job is relatively simple: Given a list of correctly-spelled words (which we'll call a *word set*) and the input text, a spell checker reports all of the *misspellings* (i.e., words that are not found in the word set) in the input. When a misspelling is found in the input, a good spell checker will also suggest words appearing in the word set that are somewhat like each of the misspelled words, since there's a reasonably good chance that one of them will be the word the writer intended.

As an example, suppose that the word set contains the following words:

```
bake  cake  main  rain  vase
```

If the input text contains the word **vake**, a good spell checker will state that **vake** is misspelled, since it does not appear in the word set, and will suggest that perhaps **bake**, **cake**, or **vase**was the word that was actually intended, since all three of these words are reasonably similar to the word **vake**, differing by only one letter each. (An even smarter spell checker might employ contextual clues to guess the word that is most appropriate in the sentence that surrounds it, but this is beyond the scope of our work here.)

Of course, a spell checker's task is centered around searching for words in a potentially large word set. An efficient search structure is critical to the performance of the spell checker, since it will be necessary to search not only for the words in the input, but possibly hundreds of potential suggestions that it might like to make for each misspelling. As you will see, a poor technique can render a spell checker — or any system that requires a large number of insertions and searches to be performed on a large search structure — effectively useless.

Getting started

Before you begin work on this project, there are a couple of chores you'll need to complete on your ICS 46 VM to get it set up to proceed.

Refreshing your ICS 46 VM environment

Even if you previously downloaded your ICS 46 VM, you will probably need to refresh its environment before proceeding with this project. Log into your VM and issue the command **ics46 version** to see what version of the ICS 46 environment you currently have stored on your VM. Note, in particular, the timestamp; if you see a version with a timestamp older than the one listed below, you'll want to refresh your environment by running the command **ics46 refresh** to download the latest one before you proceed with this project.

```
2017-05-12 02:25:25
Project #3 template added
```

Creating your project directory on your ICS 46 VM

A project template has been created specifically for this project, containing a similar structure to the **basic** template you saw in Project#0, but including a fair amount of code (both source code and compiled libraries) that is being provided as a starting point. So you'll absolutely need to use the **project3** template for this project, as opposed to the **basic** one.

Decide on a name for your project directory, then issue the command **ics46 start *YOUR_CHOSEN_PROJECT_NAME* project3** to create your new project directory using the **project3**template. (For example, if you wanted to call your project directory **p3**, you would issue the command **ics46 start p3 project3** to create it.) Now you're ready to proceed!

The project directory

Change into your project directory and take a look around. Having already completed earlier projects, what you will see will look somewhat familiar, though there are a couple of differences.

Once again, your project directory is capable of building (any or all of) three separate programs that you can run by issuing the commands **./run app**, **./run exp**, or **./run gtest**. A **provided** directory contains a set of source code for a mostly-complete spell checker; you will not be allowed to modify this code in any way, but you might find it instructive to read through as you work. An **include** directory contains declarations of ancillary tools that we needed to build the program, but that you are not likely to need yourself. The usual **app**, **core**, **exp**, and **gtest** directories are present, as well, and serve the same purpose as previously; **app** contains a **main.cpp** (and may not need anything else), **core** is where you'll do your primary work, **exp** is a place for you to write any experiments you need along the way, and **gtest** is where you can write Google Tests to validate that your code works as intended.

The program

Your work on this project begins with a partially-working spell checker. It allows a user to specify some configuration via the standard input (i.e., **std::cin**) and writes its output to standard output (i.e., **std::cout**). In short, it is capable of doing two different things:

1. Given a file containing a word set and another file containing input text, it can check spelling in the input text and report on misspellings, along with suggestions about potentially correct spellings.
2. Given a file containing a word set and another file containing input text, it can do the work of a spell checker without displaying a result, instead displaying the CPU time required to perform the task. This allows us to test different search structures to see what effect they have on performance.

Much of the program is already written and its complete source code has been provided. I will only be requiring you to implement two relatively small parts of it:

- A class called **WordChecker** that checks the spelling of words and makes appropriate suggestions when they're misspelled.
- Three class templates that derive from an existing class template **Set<T>**, which implements the concept of a *set* (i.e., a collection of search keys). Only a handful of operations is supported in each, and you are not required to implement anything not declared in **Set<T>**. There are actually four different class templates that you can write, but you are only required to choose three of them — you can choose any three of the four you'd like.
  - **BSTSet<T>**, which is a binary search tree with no attempt made to maintain balance.
  - **AVLSet<T>**, which is an AVL tree.
  - **SkipListSet<T>**, which is a skip list.
  - **HashSet<T>**, which is a hash table with separate chaining, implemented as a dynamically-allocated array of linked lists.

  Note that there is a member function called **isImplemented** in each of these class templates. For each of the class templates that you choose to implement, you'll need to modify this member function so that it returns **true**, so our test automation tools can easily detect which ones you chose to implement. Note, also, that there are some specific requirements and limitations in each class template; the comments in their header files specify them in detail.

If you're interested in seeing an example implementation of the **Set<T>** class template, check out **ListSet.hpp** in the **provided** directory in your project directory, which is a set implemented using an unsorted singly-linked list with head pointer.

Sanity-checking unit tests

As you saw in Project#0, a set of sanity-checking unit tests has been provided for each of the five classes you may implement in this project. Their role is not to test that the functionality works correctly, but instead to verify that your code remains compatible with the unit tests we'll be using as part of how we grade your work — for example, you haven't changed the signatures of the member functions, removed any of them, and so on. Initially, these sanity-checking unit tests will compile, since an empty implementation of every member function has been provided. You'll want to be sure that your code continues to compile with these sanity-checking unit tests.

Do not modify the sanity-checking unit tests, because this will invalidate the testing that they provide: Ensuring that your work compiles against our unit tests.

Insurance

You are also welcome to implement all four of the **Set<T>** class templates, even though only three of them are required. If you do, we'll offer you additional credit for it, which can be used to offset credit lost elsewhere in *this project*. In that sense, you can think of the fourth implementation as a form of "insurance" that can help you maintain a higher score on this project. But this is not an open offer of extra credit, so (a) your score on this project can't exceed 30 points out of 30, and (b) a fourth implementation in this project will not affect your scores on other projects.

The input and output, in detail

The program takes input in a particular format and writes its output in particular formats, as well, though the code for this has already been written. Still, you'll need to understand these formats in order to complete your work, so they are described below.

The input

The program reads four lines of input from the standard input (i.e., **std::cin**), without printing any kind of prompts; it simply waits until each line of input arrives before proceeding.

1. The first line selects a search structure. It can be any of the following:
   - **EMPTY**, which specifies an "empty set", a search structure that stores no keys and for which all searches return **false**. (This is used primarily in performance testing.)
   - **LIST**, which specifies a linked list implementation (which has been provided already).
   - **HASH ZERO**, which specifies a hash table implementation using a hash function that always returns zero.
   - **HASH SUM**, which specifies a hash table implementation using a hash function that sums the character codes for every character of a word.
   - **HASH PRODUCT**, which specifies a hash table implementation using a hash function that includes multiplication by prime numbers in its calculation.
   - **BST**, which specifies a binary search tree with no balancing techniques used.
   - **AVL**, which specifies an AVL tree.
   - **SKIPLIST**, which specifies a skip list.
2. The second line specifies the path to a file that contains the word set (i.e., the set of correctly-specified words). The word set file will be assumed to have one correctly-spelled word on each line.
   - You'll find a file called **wordset.txt** in your project directory, which is a large set of English words obtained from an open source wordlist project called **wordlist.sourceforge.net (http://wordlist.sourceforge.net/)** .
   - You might also want to create one or more smaller word sets to test with, since the provided word set has over 60,000 words.
3. The third line specifies the path to a file that contains the input text for which you want to check spelling. The file will be tokenized (i.e., broken into words, with punctuation and spaces ignored) and the words will be checked by your **WordChecker** class.
   - You'll find a file called **biginput.txt** in your project directory, which is one very large test case that contains a combination of words that are spelled correctly and incorrectly. This, too, was obtained from the open source wordlist project at **wordlist.sourceforge.net (http://wordlist.sourceforge.net/)** .

- You'll almost certainly want to create one or more smaller test inputs, so you can determine whether your implementations of the various **Set<T>** class templates and your **WordChecker** class are correct.

4. The fourth line specifies the format of the output you'd like to see, which allows you to choose between seeing a display of the misspellings and suggestions, or whether you're just interested in seeing how much time the spell-checking process takes. This line needs to say either:
   - **DISPLAY**, if you want a display of misspellings and suggestions.
   - **TIME**, if you want to see only timing results.

Display output

When you ask for a display of misspellings and suggestions, for each misspelling, you'll see the line of text, what word was misspelled, and suggested correct spellings. For example, if the input text is this:

```
This is a lne of text that has a missspelling in it.
```

you would see the following output:

```
This is a lne of text that has a missspelling in it.
     word not found: LNE
  perhaps you meant:
      LANE
      LEE
      LIE
      LINE
      LONE
      ONE

This is a lne of text that has a missspelling in it.
     word not found: MISSSPELLING
  perhaps you meant:
      MISS SPELLING
      MISSPELLING
```

Timing output

When you ask for timing output, the following experiment is run:

- The word set is loaded from the file and added into your chosen search structure.
- The input text is parsed, all words are compared against the word set, and all suggestions are generated. However, none of the misspellings or suggestions are displayed to he user.
- The word set is loaded from the file again, and instead added into the *empty set* search structure (i.e., one that doesn't store any keys and for which all searches return **false**).
- The input text is parsed again, all words again compared against the empty set, and suggestions generated.

Finally, three sets of times are shown: for everything (i.e., all of the work done using both the chosen search structure and the empty set), for just the empty set, and the difference. (The latter of these three timings is a reasonably good estimate of how much time was spent just in your chosen search structure, not including the time spent parsing the input or generating suggestions.) Each set of times indicates how much time was

spent loading the word set and adding the words into the search structure, how much time was spent performing the spell checking, and the total of these times.

The results, which are indicated as CPU time measured in microseconds (i.e., millionths of a second), look like this:

```
RESULTS
                LoadTime     SpellCheckTime     TotalTime
Everything       4040888usec         109778usec   4150666usec
Empty Set          36817usec            130usec     36947usec
Set Only         4004071usec         109648usec   4113719usec
```

This will give you some insight about how the various search structures perform relative to one another. Try your program using different search structures, word sets of different sizes and different organizations (e.g., sorted randomly, listed in ascending order), and input text of different kinds, and see what kind of insight you can gain about how the search structures compare to each other.

Generating suggestions for misspelled words

There are two popular text-mode spell checkers on Unix/Linux systems. One is called *ispell*; the other is a GNU "free software" program called *aspell*. They both use similar techniques for generating suggestions for misspelled words. While checking the spelling of an input file, if a word is not found in the word set, five techniques are used to generate possible suggestions. Each suggestion is looked up in the word set; any suggestion found in the word set is added to the list of suggestions, and no word should appear in the list of suggestions twice. The five techniques used are:

- Swapping each adjacent pair of characters in the word.
- In between each adjacent pair of characters in the word (also before the first character and after the last character), each letter from 'A' through 'Z' is inserted.
- Deleting each character from the word.
- Replacing each character in the word with each letter from 'A' through 'Z'.
- Splitting the word into a pair of words by adding a space in between each adjacent pair of characters in the word. It should be noted that this will only generate a suggestion if *both* words in the pair are found in the word set.

Your WordChecker class will need to generate suggestions using all five of these techniques, as well. (It should be noted that there are other ways to generate suggestions, including using algorithms that pay attention not only to the letters, but what the letters actually sound like. One such well-known algorithm is called the Soundex algorithm. You are not required to implement such algorithms for this project.)

Limitations

You are not permitted to modify any of the code in the **provided** directory, and note that the **gather** script (which gathers up your C++ source and header files for submission) will not include any of this code in your

submission. What you submit must be compatible with what was already provided.

Note, too, that some of the search structures have limitations on how you're permitted to implement them: portions of the C++ Standard Library that are off-limits, certain techniques that must be used, or performance guarantees (say, certain operations that must run in $O(\log n)$ time). See the comments in each of the files (e.g., **core/BSTSet.hpp**) for more details.

Deliverables

After using the **gather** script in your project directory to gather up your C++ source and header files into a single **project3.tar.gz** file. Submit the tar file (and only that file) to EEE Canvas. Refer back to Project #0 if you need instructions on how to do that.

Be aware that I'll be holding you to all of the rules specified in that document, including the one that says that you're responsible for submitting the version of the project that you want to be graded. We won't regrade a project simply because you submitted the wrong version accidentally. (It's not a bad idea to look at the contents of your tarball before submitting it; see Project #0 for instructions on how to do that.)

Can I submit after the deadline?

Yes, it is possible, subject to the late work policy for this course.