EECS 281: Project 1 - Letterman Reboot (Path Finding)

Due Thursday February 1, 2024 at 11:59 PM

Overview

The evil Spell Binder is loose, and it’s up to Letterman to save us! Letterman hasn’t been very active lately, and his power of changing one word into another by changing only one letter needs upgrading. Yes, in the old days he could change “tickle” into “pickle”, but can this new Letterman 2.0 change “evil” into “good”? Only you can say!

Your program will be given a dictionary of words, a “beginning” word and an “ending” word, and which types of conversions you are allowed to perform (such as changing one letter to another, adding or deleting a letter, etc.). Your goal is to convert the beginning word to another word, to another word, etc., eventually leading to the ending word, making one change at a time. You must use the given rules of conversion, and only use valid words from the dictionary along the way. For example, you could change “chip” into “chop”, but you couldn’t change “chip” into “chiz” because the word “chiz” doesn’t exist in the dictionary.

Learning Objectives

These are the skills and concepts encountered in this project:

- Text Dictionary: read, store, access, and write
- Breadth first search (BFS w/ queue)
- Depth first search (DFS w/ stack)
- Word and modification mode output
- Create custom data structures for efficient storage and access
- Use `getopt_long()` to handle command line inputs
- Use `std::cin` and `std::cout` to handle input and output
- Use `std::cerr` to handle error messages
- Use `std::vector<>` to store and access data
- Use `std::deque<>` to store and access data

Command Line Input

You must help Letterman navigate through the Spell Binder's word traps. You will be given a beginning and ending word, and a dictionary to search. The beginning and ending words, and any other necessary options, will be given on the command line when the program is run, while the dictionary will come from a file or direct input. In the spirit of the Unix tradition, your program will support command line options and use file redirection of standard input (`cin`).

Modifying Behavior

Programs often offer *options* that can change how they work at runtime. These are specified at the command line in the form of either "short options", which are a single character following a single hyphen (eg. `-o`) or "long options", which use full words following a double hyphen (eg. `--a-long-option`). Multiple short options can be combined (eg. `-al` is equivalent to `-a -l`). Both short and long options can also accept *arguments*, which allow a custom value to be associated with an option. Each option can independently prohibit, require, or optionally accept arguments.

Q: Why both short and long options?

A1: There are only 52 different single letter options, but an infinite number of long options can be created.

A2: Two programs can implement the same functionality using different options (eg. `-R` or `-r` for `--recursive`), but full words are more easily remembered.

A3: Short options are quick and easy to type at the command line, and when programs are used in scripts, long options make scripts more readable.

The solution to this problem must accept both short and long options. Some options will be implemented with no arguments, and others will be implemented with required arguments.

> Clarification: An option with a required argument is still optional and might not be specified at the command line, but if it is specified, an argument must be provided.

The complicated task of parsing options and arguments is made easier with a classic library `<getopt.h>` that provides both `getopt()` for handling short options only, and `getopt_long()` for handling short and long options. A helpful reference can be found at the getopt man page.

Letterman Options

Your program, `letter`, should accept the following case-sensitive command line options:

- `--help/-h`

  This options prohibits arguments and causes the program to print a brief help message of your own design, which describes what the program does and what each of the options are. The program should then `exit(0)` or `return 0` from `main()`.

- `--queue/-q`

  This option prohibits arguments and directs the program to use a search container that behaves like a queue, performing a breadth first search (exactly one of `stack` or `queue` must be specified, exactly once)

- `--stack/-s`

  This option prohibits arguments and directs the program to use a search container that behaves like a stack, performing a depth first search (exactly one of `stack` or `queue` must be specified, exactly once)

- `--begin <word>/-b <word>`

  This specifies the word that Letterman starts with. This option must be specified on the command line, and when it is specified a single word argument is required to follow it.

- `--end <word>/-e <word>`

This specifies the word that Letterman must reach. This option must be specified on the command line, and when it is specified a single word argument is required to follow it.

- `--output (W|M)/-o (W|M)`

  This option requires a single argument and specifies the output format. Legal options are `W` (word format) or `M` (modification format). If the output option is not specified, default output to word format. See the examples below regarding use.

- `--change/-c`

  This option prohibits arguments and allows Letterman to change one letter into another during word morphs.

- `--length/-l`

  This option prohibits arguments and allows Letterman to insert or delete a single letter during word morphs.

- `--swap/-p`

  This option prohibits arguments and allows Letterman to swap any two adjacent letters during word morphs.

Listing an option as `--stack/-s`, means that calling the program with `--stack` does the same thing as calling the program with `-s`.

You do not need to error check command line inputs, the AG will always specify them correctly, but you might find it helpful to add your own custom options for testing and debugging.

Legal Command Line Examples

Legal command line arguments must include exactly one of `--stack` or `--queue` (or their respective short forms `-s` or `-q`), and both `--begin` and `--end` (or their respective short forms `-b` and `-e`). If none are specified or more than one of stack or queue or is specified, or begin or end are duplicated, the program should print an informative message to standard error (`cerr`) and call `exit(1)`. A legal command line must also specify at least one of `--change`, `--length`, or `--swap` (or their respective short forms `-c`, `-l`, or `-p`).

```
$ ./letter --stack -b ship -e shot --length < infile
$ ./letter --begin ship --end shot -sl < infile
```

This will run the program using the stack search container and word output mode. The only modifications allowed on words are inserting/deleting letters, NOT changing one letter into another. The file “infile” on disk is redirected to `cin`.

```
$ ./letter -b ship -e shot -c --queue --output=W < infile | more
$ ./letter -e shot -b ship -q -co W < infile | more
```

This will run the program using the queue search container, word output mode, and letters can only be changed into other letters, no insertions, deletions, or swaps. Output is sent to the more program; press the spacebar after each page.

```
$ ./letter -b ship -e shot -c --length --queue --output M < infile > outfile
$ ./letter -qclo M --begin ship -e shot < infile > outfile
```

This will run the program using the queue search container, modification output mode, and letters can be changed, inserted, or deleted. Output is saved on disk in "outfile", overwriting that file if it already exists.

Illegal Command Line Examples

```
$ ./letter --queue -s -b ship -e shot -c < infile > outfile
```

Contradictory choice of routing

```
$ ./letter -b ship -e shot -c < infile | more
```

You must specify either stack or queue

```
$ ./letter -s -b ship -e shot < infile > outfile
```

You must specify at least one of change, length, or swap

Input (The Dictionary)

The program gets its dictionary from `cin`, in plain-text format. The dictionary can be in a file, and you redirect that file to `cin` when you run the program using `<` at the command line. There are two different types of dictionaries that the program needs to be compatible with: complex (C) and simple (S).

For both dictionaries, the first line will be a single character specifying the dictionary type 'C' or 'S'. Unlike the output mode, which is given on the command line (see below), this is part of the file. The second line will be a single positive integer N indicating the number of lines in the dictionary not counting the first line and lines that are comments (i.e. for simple dictionaries, the number of words, and for complex dictionaries, the number of word-generating lines).

We do not place a limit on the magnitude of N and neither should your code.

Comments may also be included in any input file. Comment lines begin with "//" (without quotes) in column 1, and are allowed anywhere in the file after the second line. When developing your test files, it is good practice to place a comment on line 3 describing the nature of the dictionary in the test file. Any dictionaries with noteworthy characteristics for testing purposes should also be commented. When reading the dictionary, you should discard all existing comments from the input file; do not save them in memory as part of your data structures.

Additionally, there may be extra blank/empty lines at the end of any input file: your program should ignore them. If you see a blank line in the file, you may assume that you have hit the end.

Simple Dictionary

The first type of dictionary that your program needs to handle is the simple dictionary. This is a simple text file specifying the words in the dictionary, one word per line. Each “word” will be a sequence of alphabetic characters.

Each word in the dictionary is unique; there will never be two copies of the same word.

Here is a valid input file:

p1-letterman/spec-simple.txt

```
S
10
// Just a short example dictionary. Although these words
// are in alphabetical order, that is not required.
chip
chop
junk
leet
let
shin
ship
shop
shot
stop
```

Complex Dictionary

The second type of dictionary that your program needs to handle is a complex dictionary. Like the simple dictionary, there will be one string per line. However, in this dictionary, each line could be a simple alphabetic string like the simple dictionary, or it could contain special characters. If a line contains special characters, then it will be used to generate alphabetic words that are a part of the dictionary. Each line will contain at most one special character (except in the case of insert-each, where a pair of square brackets counts as one special character).

Here are the special characters that may be included:

Reversal

If an ampersand (&) appears at the end of the word, then both the word and the reversal of the word are generated, in that order. An ampersand will not appear in the middle of a word.

Example: with “desserts&”, both “desserts” and “stressed” are added to the dictionary, in that order.

Insert-each

If a set of characters appears inside square brackets ([]), each character is inserted into the word, generating N words in the order of the letters, where N is the number of characters within the square brackets. There will not be square brackets without letters within them and there will not be duplicate letters.

Example: with “tre[an]d”, both “tread” and “trend” are added to the dictionary, in that order.
Example: with “c[auo]t”, all three of “cat”, “cut”, and “cot” are added to the dictionary, in that order.

Swap

If an exclamation point (!) appears, then the original string and the string with the two previous characters swapped are generated, in that order. An exclamation point will only occur if at least two characters precede it.

Example: with “bar!d”, both “bard” and “brad” are added to the dictionary, in that order.

Double

If a question mark (?) appears, then the original string and the string with the one previous character doubled are generated, in that order. A question mark will only occur if at least one character precedes it.

Example: with “le?t”, both “let” and “leet” are added to the dictionary, in that order.

Here is an example complex dictionary, with words similar to the previous simple dictionary:

p1-letterman/spec-complex.txt

```
C
7
//This generates the dictionary:
//chip, chop, junk, star, tsar, ship, shop,
//shot, stop, pots, let, leet
ch[io]p
junk
st!ar
```

```
sh[io]p
shot
stop&
le?t
```

Letterman Terminology

We use specific terminology throughout, and will try to do so in office hours.

Investigation

The process of searching the dictionary to find undiscovered words that are similar to the current word. To determine the current word, it must be fetched and removed from the search container (front of a queue-based SC, top of a stack-based SC). The search is a linear search of the dictionary given as input.

Discovery

When a word is added to the search container because it is similar to the current word. A discovered word is marked as discovered so that it will not be added to the search container again. Keeping track of how a word was discovered will be required to produce output in both output modes.

Implications

- Since every word can be discovered at most once, each word can be investigated at most once.
- Some words might be discovered but never investigated (they were still in the search container when the ending word was discovered).
- Some words might be similar to multiple words, so the search order is important to get the same results as the intended solution.
- Since words are removed from the search container when they are investigated, it cannot be used to store discovered or investigated words, and it cannot be used to store the path from the beginning word to the ending word.

Letterman's Morph Modes

There are a few ways that Letterman can convert words to other words.

- Change: Letterman can change a single letter of a word
  Example: Letterman can turn "pun" into "fun"

- Swap: Letterman can swap any single pair of adjacent letters in a word
  Example: Letterman can turn “brad” into “bard”
- Insert: Letterman can add a letter to a word at any position
  Example: Letterman can turn “stun” into “stunt”
- Delete: Letterman can remove a letter from a word at any position
  Example: Letterman can turn “boar” into “bar”

The modifications that Letterman is allowed to make will be determined by arguments on the command line.

Routing schemes

You are to develop two routing schemes to help Letterman get from the beginning word to the ending word:

- A queue-based routing scheme
- A stack-based routing scheme

In the routing scheme use a container of words that behaves like a queue or stack to direct the search. We will refer to as the “search container”, and it is best implemented with a deque. First, initialize the algorithm by adding the beginning word into the search container and marking the word as discovered. Then repeat the following steps until the end word is discovered or the search container is empty when an attempt to fetch the next word is made:

1. **Fetch:** store the next available word from the search container, this becomes the “current word”. If the search container is empty the search has failed and there is no series of words to foil the Spell Binder’s evil plan, the program should report failure and end, returning 0 from `main()` .
2. **Remove:** delete the next available word from the search container.
3. **Investigate:** add all available words from the dictionary to the search container that are sufficiently similar to the current word. A word is considered available if it has not already been discovered. The command line options given to the program will determine what “sufficiently similar” means. The order these words are to be added to the search container is the same order they were listed in the original dictionary.
    - Mark each word added to the search container when it is discovered, so that it will not be added again.
    - If the ending word is discovered, stop searching and output the solution, otherwise go back to step #1.

Output Format

The program will write its output to standard output (`cout`), and there are two possible output formats. The output format will be specified through a command line option '–output', or '-o', which will be followed by an argument of W or M (W for word output and M for modification output). See the section on Letterman Options for more details.

For both output formats, you will show the path of words you took from begin to end. In both cases you should first print the number of words in the morph, and include the beginning word (see examples below).

Word Output Mode

For this output mode (W), you should print each word in the morph. Starting at the beginning word, print the words in the morph until you reach and print the ending word.

For both the simple and complex dictionary sample inputs specified earlier, using the queue-based routing scheme and word (W) style output and trying to change "chip" into "stop", you should produce the following output:

p1-letterman/spec-queue-word-output.txt

```
Words in morph: 4
chip
chop
shop
stop
```

Using the same input file but with the stack-based routing scheme, you should produce the following output:

p1-letterman/spec-stack-word-output.txt

```
Words in morph: 4
chip
ship
shop
stop
```

Modification Output Mode

For this output mode (M), instead of printing each word, each line of output should be how to change from one word to the next. The number of words in the morph and the beginning word should always be displayed. The modification lines which follow show how the previous word is modified to move forward in the morph. Each modification has one of the following forms:

- `c,<position>,<letter>`
- `i,<position>,<letter>`
- `d,<position>`
- `s,<position>`

These four forms correspond to a letter changing (c), a letter being inserted (i), a letter being deleted (d), or two letters being swapped (s). The `<position>` always indicates the zero-based index of the modification (for swaps, the index of the first letter swapped), and when included, the `<letter>` is the new letter (either changed to or inserted).

If there is one change, but two possible places for the index, always specify where the first difference occurs. For example, if “put” changed into “putt”, the characters at positions 0, 1, and 2 are the same, the new letter occurs at index 3. Similarly for changing “putt” into “put”, the deletion occurs at index 3 (because indices 0 through 2 are the same characters).

The following are examples of correct output format in (M) modification mode that reflect the same solution as the word output format above.

Modification Mode Queue Solution

p1-letterman/spec-queue-modification-output.txt

```
Words in morph: 4
chip
c,2,o
c,0,s
c,1,t
```

In the above output, it is saying to begin with the word “chip”, change the letter at index 2 into o (producing chop), then change the letter at index 0 into s (producing shop), then change the letter at index 1 into t (producing stop).

Modification Mode Stack Solution

p1-letterman/spec-stack-modification-output.txt

```
Words in morph: 4
chip
c,0,s
```

```
c,2,o
c,1,t
```

No Solution

There is only one acceptable solution per routing scheme for each dictionary and begin/end word pair. If no valid morph exists (such as trying to change “ship” into “junk”), the program should simply display `No solution, X words discovered.` on a line by itself, instead of the “Words in morph” line. The X represents the number of words in the dictionary that were discovered as part of the search. A word has been “discovered” if it was ever added to the search container. For example, if you were trying to change “ship” into “junk”, simple dictionary with change mode on, the output would read `No solution, 7 words discovered.` . This output will be the same for either word or morph output mode, and for either stack or queue mode.

Errors You Must Check For

A small portion of your grade will be based on error checking. Your program will be provided with deliberately invalid inputs, and if your program does not reject them, you will lose points.

If your program receives any invalid input, it should print a message to `cerr` , then immediately `return 1;` from `main()` or call `exit(1);` The error message is highly recommended to help you debug, but not required; the AG will not evaluate the error message. If the input is valid (even if there is no solution), your program should `return 0;` from `main()` , and **SHOULD NOT** call `exit()` with any value.

You must check for the following errors:

- More or less than one `--stack/-s` or `--queue/-q` on the command line.
- No `--change/-c` , `--length/-l` , or `--swap/-p` on the command line.
- The `--output/-o` option is followed by an invalid string.
- Either the begin or end word is not specified, or does not exist in the dictionary.
- The `--change/-c` and/or `--swap/-p` flags are specified, but `--length/-l` is not, and the begin/end words do not match in length (creating an impossible situation).

In all of these cases, print an informative error message to `cerr` and call `exit(1);` . Read the file Error_messages.txt for standard error messages and when to use them.

You do not need to check for any other errors.

Assumptions you may make

- The first line of the dictionary will contain either a capital letter C or S and that letter will correctly reflect the dictionary type.
- The second line of the dictionary will contain a number, and the number of words in the file will match that number (the file will contain that many words/word generating lines).
- Comments will begin with // as the first two characters on a line, and can have any number of words on that line. No comment will appear before the number of words on line 2.
- The number of words on line 2 does NOT include comment lines.
- A dictionary will not contain any duplicate words.
- Other than comment lines, the dictionary will have one word per line (thus words will never contain blank spaces).
- For complex dictionaries, special characters will not appear incorrectly (for example, the reversal symbol will not appear in the middle of a word).
- You do not have to consider upper- and lower-case versions of words to be the same (for example "a" and "A" are different words).

Test Files

It is extremely frustrating to turn in code that you are "certain" is functional and then receive less than full credit. Your program will be graded for correctness primarily by running it on a number of test cases. If you have a single bug that causes many of the test cases to fail, you might get a very low score on the project even though you completed 95% of the work. Most of your grade will come from correctness testing. Therefore, it is imperative that you test your code thoroughly. To help you do this you will be required to write and submit a suite of test files that thoroughly test this project.

Your test files will be used to test a set of buggy solutions to the project. Part of your grade will be based on how many of the bugs are exposed by your test files. A bug is exposed by a test file if the test file causes a buggy solution to produce different output from the correct solution.

Each test file should be a dictionary input file. Each test file file should be named test-n-begin-end-options.txt, where $0 < n \leq 15$ for each test file. The begin and end indicate the begin/end words, and the options portion should include a combination of letters which correspond to command line options. Valid letters in the options portion of the filename are:

s: Run stack mode q: Run queue mode c: Run with change mode l: Run with length mode p: Run with swap mode w: Produce word output m: Produce modification output

The flags that you specify as part of your test filename should allow the AG to produce a valid command line. For instance, don't include both s and q, but do include one of them; include at least one of c, l and p; include at most one of w or m, but if you leave it off, the AG will assume word output mode. For example, a valid test file might be named test-1-ship-shot-scw.txt (change from ship to shot, stack mode, change mode, word output). Given this test file name, the AG would run your program with a command line similar to the following (long or short options may be used, such as –change instead of -c):

```
$ ./letter --begin ship -e shot --stack -c -o W < test-1-ship-shot-scw.txt > test-1-out.txt
```

Each dictionary may contain no more than 20 words. You may submit up to 15 test files (though it is possible to get full credit with fewer test files). The dictionaries the autograder runs with your solution are NOT limited to 20 words; your solution should not impose any size limits (as long as sufficient system memory is available). Your complex dictionary might start with 20 words but produce more; that is still a valid dictionary.

Input and Output Redirection

Input and output redirection is used in some of the above examples. While this makes it that input is read from a file and output is written to another file, file streams will not be used! The < redirects the file specified by the next command line argument to `cin` for the program. This is much easier than retyping the dictionary every time you run the program! The > redirects the output to `cout` to be printed to the file specified by the next command line argument. The vertical bar (|) pipes the output of your program to the input of the command that follows, such as more (which displays with page breaks). The operating system makes calls to `cin` to read the input file and it makes calls to `cout` to write to the output file. Come to office hours if this is confusing!

Runtime

The program must run to completion within 35 seconds of total CPU time (user + system). This is more time than you should need. See the time manpage for more information (this can be done in Unix by entering "man time" to the command line). We may test your program on very large dictionaries (up to several hundred thousand words). Be sure you are able to navigate to the end word in large dictionaries within 35 seconds. Smaller dictionaries should run MUCH faster.

Libraries and Restrictions

Unless otherwise stated, you are allowed and encouraged to use all parts of the C++ STL and the other standard header files for this project. You are not allowed to use other libraries (eg: boost, pthread, etc). You are not allowed to use the C++ smart pointers (shared or unique), or the C++11 regular expressions library (it is not fully implemented in the version of gcc used by the autograder) or the thread/atomics libraries (it spoils runtime measurements).

Submission to the Autograder

Do all of your work (with all needed files, as well as test files) in some directory other than your home directory. This will be your “submit directory”. Before you turn in your code, be sure that:

- Every source code and header file contains the following project identifier in a comment at the top of the file:
  `// Project Identifier: 50EB44D3F029ED934858FFFCEAC3547C68768FC9`
- The Makefile must also have this identifier
- You have deleted all .o files and your executable(s). Typing ‘make clean’ shall accomplish this. If make clean does not remove all of these files, you will lose points.
- Your makefile is called Makefile. Typing ‘make’ builds your code without errors and generates an executable file called `letter` `.
- Your Makefile specifies that you are compiling with the gcc optimization option -O3. This is extremely important for getting all of the performance points, as -O3 can often speed up code by an order of magnitude. You should also ensure that you are not submitting a Makefile to the autograder that compiles with the debug flag, -g, as this will slow your code down considerably. If your code “works” when you don’t compile with -O3 and breaks when you do, it means you have a bug in your code!
- Your test files are named test-n-begin-end-flags.txt, where n is an integer in the range 1-15. Up to 15 tests may be submitted.
- The total size of your program and test files does not exceed 2MB.
- You don’t have any unnecessary files or other junk in your submit directory and your submit directory has no subdirectories.
- Your code compiles and runs correctly using version 11.3.0 of the g++ compiler. This is available on the CAEN Linux systems (that you can access via login.engin.umich.edu). Even if everything seems to work on another operating system or with different versions of GCC, the course staff will not support anything other than GCC 11.3.0 running on CAEN Linux. In order to compile with g++ version 11.3.0 on CAEN you must put the following at the top of your Makefile:

```
PATH := /usr/um/gcc-11.3.0/bin:$(PATH)
LD_LIBRARY_PATH := /usr/um/gcc-11.3.0/lib64
LD_RUN_PATH := /usr/um/gcc-11.3.0/lib64
```

To run the generated executable, you need to run `module load gcc/11.3.0` once per session. You can add this line to your ~/.bashrc file if you don’t want to run it on every login.

⚠ Valgrind may report “still reachable” memory when compiling with GCC 11.3.0. This is not a concern on the autograder; you need only worry about “definitely lost” memory.

Turn in all of the following files:

- All your .h, .hpp, and .cpp files for the project
- Your Makefile
- Your test files

You must prepare a compressed tar archive or “tarball” (.tar.gz file) of all of your files to submit to the autograder. One way to do this is to have all of your files for submission (and nothing else) in one directory. The Makefile provided can prepare your submission by using either the `make fullsubmit` (all code, with test files) or `make partialsubmit` (all code, no test files). Alternatively, you can go into this directory and run this command:

```
$ tar czvf ./submit.tar.gz *.cpp *.h *.hpp Makefile test-*.txt
```

This will prepare a suitable file in your working directory.

Submit your project files directly to an autograder at https://eecs281staff.github.io/ag-status/. The autograders are identical and your daily submission limit will be shared (and kept track of) between them. You may submit up to three times per calendar day (more during the Spring). For this purpose, days begin and end at midnight (Ann Arbor local time). We will count only your best submission for your grade. If you would instead like us to use your **LAST** submission, see the autograder FAQ page, or use this form. If you use an online revision control system, make sure that your projects and files are **PRIVATE**; many sites make them public by default! If someone searches and finds your code and uses it, this could trigger Honor Code proceedings for you.

Scoring

Please be sure to read all messages shown at the top of the autograder results! The messages before the scoring of individual test cases are equally as important as the scores below, and will often explain outstanding issues the autograder finds.

- 80 points: Grades will be primarily based on the correctness of your algorithm implementation. Your program must have correct and working stack and queue algorithms and support both types of output modes. Additionally: Part of your grade will be derived from the runtime performance of your algorithms. Fast running algorithms will receive all possible performance points. Slower running algorithms may receive only a portion of the performance points. The same applies for use of system memory: programs using less memory will receive full points, solutions that use too much will lose points.

- 10 points: Memory leak check with valgrind. You can ensure that your program does not leak memory by running it locally or on CAEN with valgrind.
- 10 points: Test file coverage (effectiveness at exposing buggy solutions).

When you start submitting test files to the autograder, it will tell you (in the section called “Scoring student test files”) how many bugs exist, the number needed to start earning points, and the number needed for full points. It will also tell you how many are needed to start earning an extra submit/day!

Have fun coding!