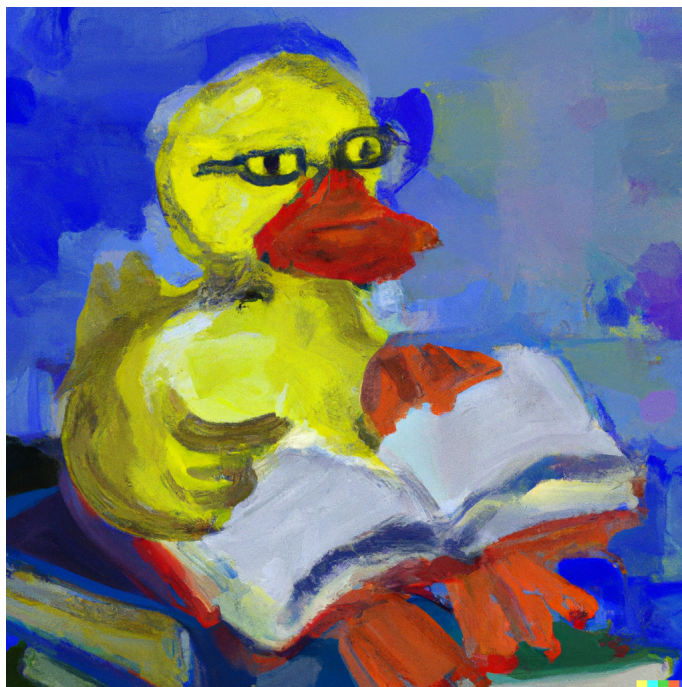


CS 15 Project 4: gerp



“May you find what you are searching for in the right places.”
- Lailah Gifty Akita

Contents

Introduction	2
Background: grep	2
Program Design Overview	3
Files to Implement	4
Pair Programming	4
Starter Files	4
Program Specification	5
Introduction	5
Command Loop	5
Beware!	6
Output Formatting	6
What is a Word?	7
Building the Index	7
FSTree	8
DirNode	8
STL Usage	9
Compiler Options	10
Time and Space Constraints	10
Reference Implementation and Testing	11
Introduction	11
Redirecting Input/Output in unix	11
Using Diff	12
README	13
Submitting Your Work	14
Part 1: Design, FSTree, and String Manipulation	14
Design	14
Directory Traversal	14
String Manipulation	15
Part I README	15
Submitting Phase 1	15
Part 2 - Final Submission	15

Helpful Tips	16
Testing	16
Hashing and Hash Tables	16
Runtime	16
Space	17
Hints	17

Introduction

We’re all familiar with web search engines, and we also have tools for searching our personal computers. Have you ever wondered how the Mac Spotlight works, for example? We’ll look at one approach now! In this assignment you will design and implement a program that indexes and searches files for strings. Your program will behave similarly to the `unix grep` program, which can search through all the files in a directory and look for a sequence of characters.

Background: `grep`

Here’s an example of calling `unix grep`:

```
grep -IRn Query DirectoryToSearch
```

In this example:

- `grep` is the program we’re calling
- `Query` is the target string we’re searching for
- `DirectoryToSearch` is the directory where we will look for the `Query`
- the option `I` means to search all files, including binary files
- the option `R` means to traverse all files under the provided directory recursively (including following symbolic links)
- the option `n` means to print each output line from `grep` with the line number of the match of the “`Query`” within the matched file.

For example,

```
grep -IRn #include /comp/15/files
```

will produce something like

```

/comp/15/files/hw1/CharArrayList.cpp:12:#include "CharArrayList.h"
/comp/15/files/hw1/simple_exception.cpp:27:#include <iostream>
/comp/15/files/hw1/simple_exception.cpp:28:#include <string>
/comp/15/files/hw1/simple_exception.cpp:29:#include <stdexcept>
/comp/15/files/hw1/unit_tests.h:15:#include "CharArrayList.h"
/comp/15/files/hw1/unit_tests.h:16:#include <cassert>
/comp/15/files/lab2/LinkedList.cpp:13:#include "LinkedList.h"
/comp/15/files/lab2/LinkedList.cpp:14:#include <sstream>
/comp/15/files/lab2/LinkedList.cpp:15:#include <string>
/comp/15/files/lab2/LinkedList.h:15:#include "Planet.h"
/comp/15/files/lab2/Planet.cpp:11:#include "Planet.h"
/comp/15/files/lab2/Planet.cpp:12:#include <sstream>
/comp/15/files/lab2/Planet.cpp:13:#include <string>
/comp/15/files/lab2/Planet.h:14:#include <string>
/comp/15/files/lab2/planet-driver.cpp:14:#include "LinkedList.h"
/comp/15/files/lab2/planet-driver.cpp:15:#include "Planet.h"
/comp/15/files/lab2/planet-driver.cpp:16:#include <iostream>
/comp/15/files/lab2/unit_tests.h:6:#include "LinkedList.h"
/comp/15/files/lab2/unit_tests.h:7:#include "Planet.h"
/comp/15/files/lab2/unit_tests.h:8:#include <cassert>
/comp/15/files/lab2/unit_tests.h:9:#include <iostream>
/comp/15/files/hw2/CharLinkedList.cpp:12:#include "CharLinkedList.h"
/comp/15/files/lab0/welcome.cpp:7:#include <iostream>
/comp/15/files/lab1/ArrayList.cpp:15:#include "ArrayList.h"
/comp/15/files/lab1/ArrayList.cpp:16:#include <sstream>
/comp/15/files/lab1/ArrayList.h:14:#include <string>
/comp/15/files/lab1/unit_tests.h:19: * Be sure to #include any .h files as necessary
/comp/15/files/lab1/unit_tests.h:37:#include "ArrayList.h"
/comp/15/files/lab1/unit_tests.h:38:#include <cassert>
/comp/15/files/lab1/unit_tests.h:39:#include <iostream>
/comp/15/files/lab1/unit_tests.h:40:#include <string>
/comp/15/files/compilation_tests/hw1_test.cpp:13:#include <iostream>
/comp/15/files/compilation_tests/hw1_test.cpp:14:#include "CharArrayList.h"
/comp/15/files/compilation_tests/hw2_test.cpp:14:#include "CharLinkedList.h"
/comp/15/files/compilation_tests/hw2_test.cpp:13:#include <iostream>

```

The first line of this output tells you that `#include` can be found on line 12 of the file located at `/comp/15/files/hw1/CharArrayList.cpp`, and that the line itself is `#include "CharArrayList.h"`. Although there are many optional flags you might use with `grep` (see <https://linux.die.net/man/1/grep> for the details), our `gerp` program is designed to replicate the core functionality above, with a twist!

Program Design Overview

How might we build something like this? We will start by limiting the constraints of our solution—whereas `grep` produces near-instantaneous results of matches at each run of the program, we will take a two-step approach with `gerp`:

- 1 Build a data structure that indexes a set of files
- 2 Use that index to respond to queries

That is, when `gerp` is run, it will require the user to provide a directory up front. Your first task will thus be to process the files of that directory (including subdirectories). You will then enter a loop where the user provides queries that your program will (quickly) answer. The queries are effectively searches for words in the indexed files. Described in English, a query might be: “Tell me all the files that have the word ‘potato’ inside the directory `/comp/15`.”

While processing the files, you might find it useful to store information about those files such as their names, their relative paths on the filesystem, and information about their contents, in one or more data structure(s) **that is/are easily searched and queried**. The choice of data structure(s) for this assignment is up to you!

That said, we have provided you with modules to help with indexing files (see **Building the Index** section below). Also, for this assignment, you may use certain classes from the C++ Standard Template Library (STL). Descriptions of the STL implemetations that you are allowed to use are listed in their own section named **STL Usage**.

To help you learn the interface and get a feel for the program, we have also provided you with a working reference implementation. For help on how to test your work against the reference, see the section

Testing and Reference Implementation.

Files to Implement

We will not specify most of the files or functions you will need to write. Instead, your program is required to function as described in this specification. You may accomplish this task using any combination of files, functions, and classes you wish. We will, of course, evaluate your design.

However, in addition to writing `.h` and `.cpp` files for your classes, you will need to write a `main()` for your program, and write a `Makefile`. The default `make` action should be to compile and link the entire program, and to produce an executable program named `gerp`, which you can run by typing `./gerp` in the terminal. If you have a `clean` target in your `Makefile`, be sure you do not delete the `.o` files we give to you.

You will want to write code to test the various parts of your program separately so that you do not have to debug compound errors. To that end, you are encouraged to use the `unit_test` framework to unit test your classes; however, testing details will be up to you. As usual, you will also be required to submit a `README`. See the `README` section for details.

Pair Programming

For this project, you will be working together with a partner! You will be responsible for choosing your partner. It is essential that you and your partner have compatible working schedules, because you are required to work together for all stages of the project. For more information on our pair programming guidelines, see our [reference document](#).

Starter Files

To copy the starter files, run the following command on the server

```
/comp/15/files/proj_gerp/setup
```

Note that you should **not** run `cp` except in the following situation.

Sometimes strange things happen with the provided object (`.o`) files. For example, students may accidentally try to re-compile the files or add incorrect (and unnecessary) `Makefile` rules for them. If you see something weird going on with any of the provided `.o` files, copy the original files back into your directory with:

```
cp /comp/15/files/proj_gerp/*.o .
```

This is generally the first thing a TA would try.

Program Specification

Introduction

Your program will be run from the command line like this:

```
./gerp DirectoryToIndex OutputFile
```

where `DirectoryToIndex` determines which directory will be traversed and indexed, e.g. `/comp/15/files`, and `OutputFile` names the file to which the query results will be sent.

If the user did not specify exactly two command line arguments (in addition to the program name), print this message to `std::cerr`:

```
Usage: ./gerp inputDirectory outputFile
```

and terminate the program by returning `EXIT_FAILURE` from `main`.

After being called, your program will first traverse a file tree created using a module provided by the course staff (details are in the section titled **Building the Index** below). It will index each file that it finds in the tree. After indexing all of the files, it will enter a command loop (similar to the “interactive” modes that you have implemented in previous homeworks) where the user can enter various commands to modify the search, and to quit the program.

Command Loop

Specifically, your program will print `Query?` followed by a single space to `std::cout` (**NOT** to the `OutputFile`), and then wait for a command from the user. The possible query commands are:

- `AnyString`

A word (see the **What is a Word?** section below) is treated as a query. The program will take this string and print all of the lines in the indexed files where `AnyString` appears. Note that this is a case sensitive search, so `we` and `We` are treated as different strings/words, and so should have different results.

- `@i AnyString` or `@insensitive AnyString`

Preceding a query string by `@i` or `@insensitive` causes the program to perform a case insensitive search on the string that was passed. For example, `we` and `We` would be treated as the same string/word and will have the same results.

- `@q` or `@quit`

These commands will completely quit the program, and print:

Goodbye! Thank you and have a nice day.

This statement should be followed by a new line. Note that the program should also quit if it reaches End-Of-File (EOF).

- `@f newOutputFilename`

This command causes the program to close the current output file. Any future output should be written to the file named `newOutputFilename`.

Beware!

Beware! There are **two** output streams.

- The “Query? ” prompt always goes to `std::cout`
- The result of the query always goes to the output file, which will **either** be the `OutputFile` **or** the file named in the last `@f` command.

Also, treat a multi-word query as several independent 1-word queries - e.g.

```
Query? We are the champions
```

is the same as

```
Query? We
```

```
Query? are
```

```
Query? the
```

```
Query? champions
```

Note, as seen when running the reference implementation, that this will actually appear as:

```
Query? we are the champions
```

```
Query? Query? Query? Query?
```

Output Formatting

If the word (see the **What is a Word?** section below) in a query is found in the index, then, for each line it appears in, you will print to the designated output file a line of the form:

```
FileNameWithPath:LineNum: Line
```

Where:

1. `FileNameWithPath` is the full pathname of the file (including the path from the command line), followed by a colon
2. `LineNum` is the line number within that file that the query word appears on, followed by a colon and a space
3. `Line` is the full text of the line from the file
4. A newline

For example, if you ran

```
$ ./gerp small_test out.txt
Query? we
```

which queries `we` on our `small_test` directory, and then sends output to the file `out.txt`, the file `out.txt` might read:

```
small_test/test.txt:5: we are the champions
small_test/test.txt:6: we we we
```

NOTE: There is one newline after the last line. Also, each line that the query appears in only prints once. If the query is not found using the default search, then print:

```
query Not Found. Try with @insensitive or @i.
```

If the query is not found using the insensitive search, then print:

```
query Not Found.
```

What is a Word?

It is important to outline what a word is when dealing with a word search engine. **We will define a word as a string that starts and ends with an alphanumeric (letter or number) character.** This means that you will need to do a little string parsing to determine the output of your `gerp` implementation. To help you with this nuance we have included a couple of examples.

When searching for the word `comp` using case insensitive search, `gerp` should treat the following strings as `comp`:

- `comp`
- `comp.`
- `Comp`
- `-comp`
- `&&comp`
- `comp?!`
- `@#comp?@!`

If any of the bulleted strings were submitted as a query, `gerp` should print the lines in files that contain any of the strings on the list (however, it should print them as they exist in the file, not a processed version).

Note that `gerp` should only compare strings where all leading and trailing non-alphanumeric characters are stripped. This includes both the queries and the strings in the data files.

Note that words can contain non alphanumeric characters in the middle. For instance, `comp&!$15` is considered all one word. It should not be split into two queries as with spaces.

Building the Index

You do **NOT** need to write the `FSTree` or `DirNode` classes described below - we have implemented them for you. However, they will be critical to the success of your project!

FSTree

We will use a file-system tree to represent directories, subdirectories, and files. The data structure we will use is an **n-ary** tree, so called because a node of the tree could have any number of children. The main usage of this class is to help you navigate through folders and directories inside the computer. For example, a snapshot of a home directory might be represented in an n-ary tree like this:

```
/h/mkorman
  /coursework
    /comp11
    /comp15
      /exams
      /labs
      /assignments
      /hws
    /comp160
      /hw
  /public_html
```

Specifically, a **FSTree** is an n-ary tree which consists of **DirNodes** (which are described below). The **FSTree** class has the following public functions:

- **FSTree**(std::string rootName)
This is the constructor of the **FSTree**. It creates a file tree of **DirNodes** where the root of the tree is the directory that is passed as the parameter **rootName**. If there is an error opening directories or files, the constructor will fail and halt your program. Be careful and do not run this on just any directory - if a directory structure has a loop in it, the constructor can run forever!
- ~**FSTree**()
The destructor deallocates all of the space allocated when the tree was built.
- **DirNode** ***getRoot**()
This function returns the root of the tree. Normally, we do not want to return the private members of an object or class, however in this case it is necessary so that you can traverse the tree and index its contents.

DirNode

The **DirNode** class is the key building block of the **FSTree** class. It is our representation of a folder. Each **DirNode** instance has: a name, a list of files in the directory, and a list of subdirectories. It contains the following public methods:

- **bool** **hasSubDir**()
Returns true if there are any sub-directories in this directory.
- **bool** **hasFiles**()
Returns true if there are files in this directory.
- **bool** **isEmpty**()
Returns true if there are no files or sub-directories in this directory.

- `int numSubDir()`
Returns the number of sub directories in this directory.
- `int numFiles()`
Returns the number of files in this directory.
- `std::string getName()`
Returns the name of this directory.
- `DirNode *getSubDir(int n)`
Returns a pointer to the `n`th subdirectory.
- `std::string getFile(int n)`
Returns the `n`th file name in this directory.
- `DirNode *getParent()`
Get the parent directory of this directory.

The `DirNode` class contains other public functions that are necessary for our implementation of `FSTree`. You should **not** have to use them. In order to get a file's full path, you will need to traverse the `FSTree` and concatenate the names of the directories you encounter along the way. You will then use this full path to open the file in an `std::ifstream` and index its contents.

STL Usage

For this assignment, you will be allowed to use **ONLY** the following STL implementations:

- `vector`
- `queue`
- `stack`
- `set`
- `list`
- `functional`

You may also find certain function(s) in the `cctype` library useful. You are also allowed to use `pair` from `utility`. That being said, you could easily make your own `struct`. If you have questions about using a particular library or class, clarify with a TA first. In general, if using something makes all or part of the assignment trivial, we will likely say no (e.g. you can't use `unorderedmap`).

You are not required to use any particular item of the STL. If you feel that one or more of these would be useful, you will need to learn about their respective interfaces. You can find more information about them at:

<http://www.cplusplus.com/reference>

Any other data structures that you need you must implement yourself. Moreover, you **may not** use any kind of parallel programming or concurrency in your implementation.

Compiler Options

When compiling your implementation of `gerp`, you should compile with the flag `-O2` (That's a capital letter 'O', not the numeral zero). This will optimize your program for the system that it is compiling on, which will result in an implementation with a faster runtime. This will help during the testing phase because you will receive your results faster.

Time and Space Constraints

When designing and implementing your program you should have it build its index and run queries as quickly as possible. You may find that there is a trade-off between the two (e.g. a program that builds an index quickly may not search as fast). It is important to document your design choices, your justification of those choices, and their effects in your `README`.

To get full credit, your program will need to be able to index and run queries on the following directories within the following amounts of time:

- `tinyData` under 15 seconds
- `smallGutenberg` under 15 seconds
- `mediumGutenberg` under 2 minutes
- `largeGutenberg` under 8 minutes

Roughly, this means your solution can be about 15 times as slow as the reference (which already is not the best solution). Furthermore, your program can use a maximum of **6 GB of RAM**. If you take too much time or use too much memory on a test, you will not receive points for that test.

You should check the time and memory usage of your program using the following command:

```
gerp_perf [DirectoryToIndex] [OutputFile]
```

This command will display your program's memory usage and execution time of building the index. It will also display these metrics for the reference implementation. `[OutputFile]` will store the non-`cout` output of your program as usual. Note, to use this program, you will need to have added `use -q comp15` to your `.cshrc` file as you would have done to use `unit_test`.

Before submitting to Gradescope, please make sure you have run your implementation on larger directories. If Gradescope times out during a submission, it likely means your implementation is too slow so it is good to get it working decently well before submitting. If Gradescope times out, it will **still count as a submission which we generally will not give back**.

Reference Implementation and Testing

Introduction

In order to help you with your testing and to get familiar with the user interface expectations of this project, we have provided you with a fully compiled reference implementation called `the_gerp`. By the end of the project your `gerp` implementation should behave exactly the same as `the_gerp`. **Your version of `gerp` should behave exactly as the reference in all circumstances. Therefore, you should extensively test the reference in order to figure out how it behaves.**

Testing Directories

In order to help you test your implementation, we have provided you with some directories. The folder `/comp/15/files/proj-gerp-test-dirs` contains all of the relevant directories, which are

- A Directory `tinyData`, which you can use to quickly test your work.
- Small, medium and large subsets of Project Gutenberg. These are samples from a massive free online library—for more information on this, see the [Project Gutenberg Website](#). Indexing/querying the medium/large subsets will give you a chance to test the time and space efficiency of your code. For this datasets, be sure that your submission matches the expectations described earlier (indexing in under 8 minutes, no more than 6GB memory usage).

Note: This directory contains all the testing directories, so you should not copy it. It is large and will take up your storage quota. Instead, `gerp` takes a directory as a command line argument, so you don't need the directory locally; just run `gerp` specifying, e. g., `/comp/15/files/proj-gerp-test-dirs/mediumGutenberg` and the program should work!

Redirecting Input/Output in unix

In order to test your implementation against the reference, using redirects to send input files to and from your program and the reference will be useful. To send data to the `std::cin` of a unix program, do the following:

```
./programname < fileforcin.txt
```

In our case, `gerp` takes multiple arguments: a directory to query, and a file to send the program's output to. So, to send a file to the reference implementation's `std::cin`, you would do:

```
./the_gerp Directory ref_output.txt < commands.txt
```

This command will send `commands.txt` to the standard input stream for the reference implementation. You can likewise do the same with your implementation. In order to redirect a program's `std::cout` to a file, do the following:

```
./programname > fileforcin.txt
```

Again, in our case,

```
./the_gerp Directory ref_output.txt > ref_std_out.txt
```

To combine redirection of input and output, you can do the following:

```
./the_gerp Directory ref_out.txt < cmds.txt > ref_std_out.txt
```

Recall that, even though `gerp` sends **most** data to the the output file, it will send some to `std::cout` - namely, the `Query?` lines, which will then be redirected to, in this case, `ref_std_out.txt`.

Using Diff

Once you have run the same input on the reference and on your implemenation, you can test that your and the reference's outputs (the contents of `ref_out.txt` above) are equivalent by:

1. Sorting the output
2. Using `diff` to compare the sorted files

Sorting is necessary because, while your program must produce output for all occurrences of the query, the order in which multiple lines appear is not specified. Print out multiple lines in whatever order your data structure and algorithm choices find convenient. To sort the output, you can use the `unix sort` command.

```
sort ref_output.txt > ref_output_sorted.txt
sort my_output.txt > my_output_sorted.txt
```

After sorting, use the `diff` command to find any differences. For example:

```
diff ref_output_sorted.txt my_output_sorted.txt
```

`diff` will print the differences in the file, if there are any. If nothing prints out, then the files are identical.

Keep in mind that, to compare the redirected `std::cout` streams of your implementation and the reference, you should **not** sort the output files before running `diff`.

README

You will be required to submit a `README` along with your code. Your `README` should have the following sections:

- A The title of the homework and the author names (you and your partner)
- B The purpose of the program
- C Acknowledgements for any help you received, including references to outside sources you consulted (though there is no need list C++ references like `cplusplus.com`).
- D The files that you provided and a short description of what each file is and its purpose
- E How to compile and run your program
- F An “architectural overview,” i.e., a description of how your various program modules relate. For example, the `FSTree` implementation keeps a pointer to the root `DirNode`.
- G An outline of the data structures and algorithms that you used. Given that this is a data structures class, you need to always discuss any data structures that you used and justify why you used them. For this assignment it is imperative that you explain your data structures and algorithms in detail, as it will help us understand your code since there is no single right way of completing this assignment.
- H Details and an explanation of how you tested the various parts of your classes and the program as a whole. You may reference the testing files that you submitted to aid in your explanation.
- I. Please let us know approximately how many hours you spent working on this project. This should include both weeks one and two.

Each of the sections should be clearly delineated and begin with a section heading which describes the content of the section.

Submitting Your Work

You will be submitting your work in 2 parts:

Part 1: Design, FSTree, and String Manipulation

Design

As part of your design checkoff, you must do two things:

1. Provide answers to the questions in the file `gerp_design_checkoff.txt`, included in the starter files. When you're done, submit your answers to the assignment titled "gerp Design Checkoff" on Gradescope. **You must submit this file prior to your design checkoff meeting (described below).**
2. Sign up for an in-person design checkoff meeting with a TA. *Both you and your partner must attend the meeting.* You can find the design checkoff form [linked here](#). At that meeting, among other things, you will be required to address the answers that you submitted with your design checkoff questions. You should also bring drawings that capture your design.

FSTree and String Processing

To complete this part you will need to write 2 functions:

1. a directory traversal function that prints out the full paths of each file in the tree on separate lines
2. a function that strips all leading and trailing non-alphanumeric characters from a given string

Directory Traversal

You will be implementing this in `traverseDirectory` in `processing.h` and `processing.cpp`. The function takes a directory as an argument and then prints the full paths of all the files accessible from that directory. Do not worry about the order that the file paths print in, just ensure that each one of them prints. For example, if you have a directory named `Foo`, with:

- files `a.cpp`, `b.cpp`, and `c.cpp` inside it
- a subdirectory `Bar`, with:
 - files `x.cpp`, `y.cpp`, `z.cpp`

Then one possible output for running `traverseDirectory(Foo)` would be:

```
Foo/Bar/x.cpp
Foo/Bar/y.cpp
Foo/Bar/z.cpp
Foo/a.cpp
Foo/b.cpp
Foo/c.cpp
```

Again, the order here does not matter. You may assume that the directory name has no trailing /.

String Manipulation

You will be implementing this in `stripNonAlphaNum` in `processing.h` and `processing.cpp`. The function takes a string and removes all leading and trailing non-alphanumeric characters. For example, when `stripNonAlphaNum("@##!!#!@!#COMP-15!!!!!!")` returns "COMP-15".

Part I README

Your README does not need to contain everything we normally ask for. For this phase, you can just write a very brief summary with any information you'd like the grader to see (e.g. if you have a nettlesome bug you have not yet been able to fix). Additionally, make sure to write both author's names (i.e., you and your partner).

Submitting Phase 1

Note: Only one of you or your partner should submit your code each time you submit – otherwise you may receive a grade for the wrong submission. Make sure both names are in the submitted README. You will need to submit at least the following files:

```
processing.h
processing.cpp
README
```

You should only include other C++ files you write if your solution to `stripNonAlphaNum` or `traverseDirectory` depend on them. You do not need to submit `DirNode` or `FSTree`. You must submit them using Gradescope to the assignment `proj_gerp_phase1`. Submit these files directly, do not try to submit them in a folder. **Make sure to add your partner on Gradescope each time you submit.**

Part 2 - Final Submission

For this part you will submit all of the files required (including a Makefile) to compile your `gerp` program. Make sure to include any testing files. **Once again, only one of you or your partner should submit each time you submit. Furthermore, make sure you add your partner for each submission.** You will need to submit the following files:

```
Makefile
README
(... all the C++ .h, .cpp files that implement your program)
(... all your testing files)
```

You must also submit:

- Any unit tests you write for your program. This may be done in a file called `unit_tests.h`, using the `unit_test` framework that we have used on past assignments. Alternatively, you can create your own testing `main` functions, e.g., submit a file called `Gerp_tests.cpp` with a `main` function that calls out to tests. Whatever testing files you use, you must submit them!
- Any command files (excluding the command files provided).
- Note, you only need to include testing input files. You do not need to include output files of any kind (`stdout` or `stderr`).

You should submit your final implementation to the assignment `proj_gerp` on Gradescope.

Note: Make sure you submit everything we will need to run your program. Maybe copy the files into another directory, type `make` test the program, do a `make clean` and then submit everything. We should be able to use `make` or `make gerp` to build your program. You do not need to submit any provided files (`FSTree.h`, `FSTree.o`, `DirNode.h`, `DirNode.o`).

Note, you only need to include testing input files. For example, files you are searching or files with queries. You do not need to include output files of any kind (`stdout`, `stderr`, or output files where you store the file:line occurrences of words).

Before submitting your work, please make sure your code and documentation (including sections of the README) conform to the course style guide.

Helpful Tips

Testing

We will test your solution on several directories, including `smallGutenberg` and `mediumGutenberg` and `largeGutenberg`. You can see some test queries and reference implementation output in `/comp/15/files/proj-gerp-sample-execution`. You should match this output exactly (except that the order can be different as explained above).

Hashing and Hash Tables

There are example uses of the `std::hash` facility in the files we've given you. See `hash_test.cpp` and `hashExample.cpp`. If you use a hash table, then you **MUST** have it dynamically resize. You must monitor the load factor and expand if the load factor is exceeded.

Runtime

The runtime of your query processing must depend on the size of the output, not on the size of the input (that is, the index structure should be very close to **constant time** to search).

Space

Finally, space usage will be an issue. Your program will have enough memory to store all the data in the `largeGutenberg` collection once or twice, but **not** 5 or 6 times. If you are not cognizant of space usage, then your program will work fine for the small or maybe even the medium collection, but will fail on the large collection when it runs out of memory. You might find the provided file `exceedmem.cpp` helpful regarding memory issues; it shows different examples of situations you might encounter when attempting to allocate too much memory.

If you are stuck, then you should get something that at least works correctly on the small data collections. To work on larger collections, you will need to choose a strategy that considers space.

Hints

- Hint: Every copy of a string requires space proportional to the length of the string. (An `std::string` is an `ArrayList` of characters.)
- Hint: **Do not store a pointer to an element in a vector.** A good final exam question would be: “Why should you not store the address of an element in a vector?” An `std::vector` is an `ArrayList`, remember.
- Hint: Pay attention to how you pass variables to functions. For example, consider the following two function signatures:
`void foo_a(string word)`
`void foo_b(string &word)`
Passing a `string` to `foo_b`, which uses a reference parameter, will be more space and time efficient than passing a `string` to `foo_a`. See why? Leverage this to your advantage!