```
****************************************************************************
***                      COMP 15  Project 2                              ***
****************************************************************************
            ______
         .-'      `-.
       .'            `.
      /                \
     ;      HERP        ;`
     |      DE          |;
     ;      GERP        ;|
     '\                / ;
      \`.            .' /
       `.`-._____.-' .'          May you find what you are searching for
         / /`_____.-'            in the right places
        / / /                                âM-^@M-^U Lailah Gifty Akita
       / / /
      / / /
     / / /
    / / /
   / / /
  / / /
 / / /
/ / /
\/_/

////////////////////////////////////////////////////////////////////////////
//                                                                        //
//               Gerp - It's Like grep But Something Is Off               //
//                                                                        //
////////////////////////////////////////////////////////////////////////////
```

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 implement a program that indexes and searches a file tree for strings. Your program will behave similarly to the Unix "grep" program, which can be used like this to search through all the files in a directory and look for some sequence of characters:

```
        grep -Rn Query DirectoryToSearch
```

Where "Query" is the target string, and "DirectoryToSearch" is the directory we want to look in for the "Query".

For example,

```
        grep -IRn ifstream /comp/15/files
```

Will produce:

```
/comp/15/files/lab1/main.cpp:45:        ifstream in;
/comp/15/files/extras/read_from_file.cpp:34: *        cin or an ifstream.
/comp/15/files/extras/read_from_file.cpp:40: *        of ifstream a C++ string.
/comp/15/files/extras/read_from_file.cpp:56:        ifstream input;
/comp/15/files/lab4/roster.cpp:51:    ifstream infile;
/comp/15/files/lab7/er.cpp:45:    ifstream infile;
```

This output tells you that "ifstream" occurs on line 45 /comp/15/files/lab1/main.cpp and on lines 34, 40 and 56 of /comp/15/files/hw3/read_from_file.cpp, for example, and it prints out the matching line after the colon.

How can we do something like this?

Your program will 1) build an index data structure and 2) use that index to respond to queries.

When you index files, you create a database containing information on all the files within the directory. Queries are searches that a user want you to do, e. g., "tell me all files that have the word "potato."

To create an index, you will read in the files and store information about them (such as their names, their relative paths, and their contents) in a data structure of your choice that is easily searched and/or queried.

We'll first describe the program, then the implementation details, and then the submission details.

Please read through the entire assignment before you start.

```
---------------------
Program specification
---------------------
```

Your program will traverse a file tree created using a module that we provide (see the interface and description of the file tree in "Using the FSTree and DirNodes" under "Implementation Details" below). It will need to 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 HWs) where the user can enter various commands to modify the search, and to quit the program.

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

```
        ./gerp DirectoryToIndex OutputFile
```

"DirectoryToIndex" determines which directory will be traversed and indexed. You could use "/comp/15/files", for example. "OutputFile" names the file to which query results should be sent.

Once the program indexes the specified directory it will print "Query? " on cout (NOT to the OutputFile) and wait for a command from the user. The possible query commands are the following:

```
    o AnyString
            A word (see "What is a Word?" under "Implementation
            Details") 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 this a case sensitive
            search so "we" and "We" are treated as different
            strings/words and should have different results.

o "@i AnyString" or "@insensitive AnyString"
            Preceeding 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.
    o "@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 when it reaches End-Of-File (EOF).

    o "@f newOutputFilename"
            This command causes the program to close the current output
            file.  Any future output should be written to the file
            named newOutputfilename.

    NOTE: Commands will be entered without the quotation marks ("")
```

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

```
            Usage:  gerp directory output_file

    and terminate the program (by returning EXIT_FAILURE from main() or by
    calling exit(EXIT_FAILURE)).

    BEWARE:  There are two output streams.
         o The "Query? " prompt always goes to cout.
         o The results of the query always go to the output file, which'll
           either be OutputFile or the file name in the last "@f" command.

    Treat a multi-word query as several indepedent 1-word queries.
    E. g.

         Query? We are the champions

    is the same as

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


    For this assignment, Standard Template Library (STL) implementations may
    be used. A description of STL implementations that you are allowed to use
    are listed in their own section "STL Usage" under "Implementation Details".

    To help you learn the interface and get a feel for the program, we
    have provided you with a working reference implementation.

----------------------
The Files to Implement
----------------------

    For this assignment we will not specify 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 combinations of files, functions, and classes you wish.
    We will, of course, evaluate your design.

    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
    producing an executable program named "gerp", which you can run by
    typing "./gerp"

    If you have a "clean" target in your Makefile, be sure you do not
    delete the .o files we give you to.

    You will want to write test methods to test the various parts of your
    program separately so that you do not have to debug compounded errors.
```

```
----------------------
Implementation Details
----------------------

    # 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 examples.

        When searching for the word "comp" using case insensitive
        sort, gerp should treat the following strings as the same as
        "comp":
            o "comp"
            o "comp."
            o "Comp"
            o "-comp"
            o "&&comp"
            o "comp?!"
            o "@#$comp?!"

        Also if any of the bulleted strings were submitted as a query,
        gerp should print 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: gerp should compare strings where all leading and trailing
              non-alphanumeric characters are striped. This includes both
              the queries and the the strings in the data files.

    # Output Formatting #
        If the word 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

        which consists of:
            1. The full pathname of the file (including the path from
               the command line), followed by a colon
            2. The line number within that file the query word appears
               on, followed by a colon and a space
            3. 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
        opened the file out.txt, you might see

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

            NOTE: THERE IS ONE NEW LINE AFTER THE LAST LINE.
                  EACH LINE THAT THE QUERY APPEARS ON ONLY PRINTS
                  ONCE.


        If the query is not found using the default search your
        implementation should print:

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

```
      If the query is not found using the insensitive search your
      implementation should print:

          query Not Found.

      NOTE: there is no whitespace before the messages, which begin
            in column 1.


  # Using FSTree and DirNodes #
      FSTree: We use a file-system tree to represent directories,
         subdirectories, and files. The data structure we 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 a
         computer.

         For example, a snapshot of a home directory might be represented
         in a n- ary tree like this:

              /h/mkorman
                      /comp11
                      /test/comp15
                              /exams /lab
                              /assignments
                              /hws /projects /comp160
                                      /hw


      YOU DO NOT HAVE TO WRITE THIS CLASS --- We did it for you.

      An FSTree is an n-ary tree consisting of DirNodes (which will
      be described below).  The FSTree class has the following
      public functions:

          o FSTree(string rootName)
                  This is the constructor for an FSTree.  It creates
                  a file tree of DirNodes where the root of the tree
                  is the directory that was passed as a parameter.
                  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.
          o ~FSTree()
                  The destructor deallocates all of the space
                  allocated when the tree was built.
          o void burnTree()
                  This function destroys the tree, frees any data that
                  was allocated, and makes the tree empty.
          o 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.

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

          o bool hasSubDir()
                  returns true if there are sub directories in the
                  current node (directory).
          o bool hasFiles()
                  returns true if there are files in the current node
                  (directory).
          o bool isEmpty()
                  returns true if there are no files or sub directories
```

```
                  in the current node (directory).
          o int numSubDir()
                  returns the number of sub directories.
          o int numFiles()
                  returns the number of files in the current node.
          o string getName()
                  returns the name of the current directory.
          o DirNode *getSubDir(int n)
                  returns a pointer to the nth subdirectory.
          o string getFile(int n)
                  returns nth file name
          o DirNode *getParent()
                  get parent directory node

      The DirNode class also contains the following public functions
      which are used to modify the contents and structure of the
      tree (we use these functions to initially build the FSTree).
      Although you have access to these functions, you should
      refrain from using them, as it may cause the tree to lose
      data/information.  We're only telling you about them, because
      you'll see them listed in the .h file, and you may be curious.

          o DirNode(string newName)
                  Constructor that initializes a DirNode named
                  newName.
          o void setName(std::string newName)
                  set the name of the current node.
          o void addFile(std::string newName)
                  Adds a file with the name "newName" to the current node.
          o void addSubDirectory(DirNode *newDir)
                  Adds a sub directory (newDir) to the current node.
          o void setParent(DirNode *newParent)
                  Sets parent node (directory) of the current node.

      In order to get a file's full path you will need to traverse
      the FSTree and concatenate the names of the directories you
      traverse to compile a file's full path (which is necessary to
      subsequently index the file).  You will then use this full
      path to open the file in an ifstream and index its contents.

  # STL Usage #
      For this assignment you will be allowed to use *ONLY* the following
      STL template implementations:
          o vector
          o queue
          o stack
          o set
          o unordered_set
          o functional

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

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

      Any other data structures you need, you must implement yourself.


  # Compiler Optimizations #
      When compiling your implementation of "gerp" you should
      compile it 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 run time.  This will help
      everyone during the testing phase because one will receive
      results faster.
```

```
    # Testing and Reference Implementation#
        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.

        *******************************************************************
        * NOTE:  YOUR VERSION OF GERP SHOULD BEHAVE EXACTLY AS THE        *
        *        REFERENCE MATERIAL WE SUPPLIED ("THE_GERP") UNDER ALL    *
        *        CIRCUMSTANCES.  THEREFORE YOU SHOULD EXTENSIVELY TEST THE *
        *        REFERENCE TO FIGURE OUT HOW IT BEHAVES.                  *
        *******************************************************************

        You can easily compare your output to the reference by
        redirecting output to two different files:

             ./the_gerp Directory ref_output.txt < commands.txt

        This command redirects the program's input so that the
        contents of "commands.txt" appear on cin just as if someone
        typed it in.

        You could run your program similarly:

             ./gerp Directory my_output.txt < commands.txt

        Then sort both text files using the unix "sort" command:

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

        After sorting you can use the "diff" command to find the
        differences between the two files:

             diff ref_output_sorted.txt my_output_sorted.txt

        "diff" will print the differences (if there are any) to the
        terminal.  If nothing prints out, the files are identical.

        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 strucure and algorithm
        choices find convenient.


    # Building/Indexing and Queries #
        When designing and implementing your program you should aim to
        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 the choices, and their effects
        in your README.

        To get full credit, your program will need to be able to index
        the largest file tree and be ready to query in under 10
        minutes.  Our implementation uses approximately 2.3 GB of RAM,
        once it has fully indexed the file tree.  On our system, your
        program can use about 7.5 -- 8 GB of memory before it crashes
        or is killed by the operating system.  This means that you
        will lose points for tests on the largest collection if you
        exceed the memory requirements.  You can get credit for tests
        on the smaller collections, of course, if your program works
        for those.

        You can check the memory usage of your program using the following
        unix command:
```

```
             echo "@q" | /usr/bin/time -v ./gerp [DirectoryToIndex]

        This command will start your program and it will complete its
        indexing procedure then immediately quit. In the output,
        "Elapsed (wall clock) time (h:mm:ss or m:ss)" is the duration
        that it took to index the directory.  "Maximum resident set
        size (kbytes)" is the peak memory usage of your program. Do
        not forget that the results are in KB and will need to be
        converted to GB (multiply by 1,000).  If you want to test our
        program's indexing speed you should replace "./grep" with
        "./the_grep".

        Also please note our reference implementation, "the_gerp", is
        by no means the fastest indexing or fastest querying solution
        to the problem.  We hope that you are able to build a faster
        implementation!

    # Other Implementation Details #

                       *DO NOT IMPLEMENT EVERTHING AT ONCE!*

        This may seem like a lot, but if you break it into pieces, it's
        perfectly manageable. Just do it one bit at a time.

        Each class you design should do one thing.  Clear abstractions with
        clearly delineated responsibilities are easier to design, implement,
        test, understand, and debug. Therefore, it is as important for you
        as for us --- you'll work more efficiently and have fewer bugs!

        You will add one function, then write code in your test file that
        performs one or more tests for that function.  Write a function,
        test a function, write a function, test function, ... This is
        called "unit testing."

        Follow the same testing approach for every class you write!

        You should add functionality little by little.

        *******************************************************************
        * NOTE:  YOU WILL NOT BE SUBMITTING A TESTING MAIN ALONG WITH      *
        *        YOUR GERP IMPLEMENTATION. HOWEVER YOU ARE REQUIRED        *
        *        TO DETAIL YOUR TESTING METHODS IN YOUR README             *
        *******************************************************************

        If you need help, TAs will ask about your testing plan and ask to
        see what tests you have written. They will likely ask you to
        comment out the most recent (failing) tests and ask you to
        demonstrate your previous tests.

        Be sure your files have header comments, and that those header
        comments include your name, the assignment, the date, the purpose of
        the particular file, and acknowledgements for any help you received.


--------
Makefile
--------

        In addition to the other required files you will need to submit a
        Makefile that compiles your program when the user types "make" in the
        terminal. The name of your fully compiled program should be
        "gerp".

------
README
------

    With your code files you will also submit a README file. You can format
    your README however you like. However it should have the following
    sections:

        A. The title of the homework and the author’s name (you)
        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 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 the the data structure that you used and justify why you
           used it. 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.

    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 as described below:


        o Part1 - Design + Using the FSTree and String Manipulation:
            - Design
                 Because HW 5 in-person grading is on-going, we will not do
                 in-person check-offs for your Project 2 design.  Rather, you
                 will turn a design.text file in with provide that
                 addresses these items:

                 * What data structure(s) will you be using?  Why these data
                   structures?
                   CAUTION:  If your answer is just vectors, you need to think
                   again!  Use the data structures we’ve been studying.
                 * What classes will you implement (presumably including one
                   for each data structure above)?  What public functions does
                   each class support?
                 * How do your classes interact, i. e., does one class contain
                   an instance of another, a pointer to an instance of
                   another, what functions in the other class will it call?
                 * How will you implement insert, search?
                 * Describe the space needs of your solution and the big-O
                   runtime of important operations (insert, search).
                 * You may optionally submit an image file with a
                   diagram of your implementation, but keep it under 1
                   MB in size.

            - FSTree and String Manipulation
               To complete this part you will need to write 2
               programs, each with its own main function: (1) a tree
               traversal method 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.
```

```
               You should write your tree traversal in a file called
               "FSTreeTraversal.cpp".  This program should take the
               highest directory as a command line argument (see
               below) and then print the full paths of all of the
               files accessible from that directory:

                   ./treeTraversal Directory

               Do not worry about the order that the file paths print
               in, just ensure that each one of them prints.

               Your string processing method should be defined in a
               file called "stringProcessing.cpp" and have a
               declaration in a file called "stringProcessing.h".  The
               function should be called "stripNonAlphaNum", it takes
               a string as a parameter and returns a string.  This
               function should remove all leading and trailing
               non-alphanumeric characters such that when
               "@##%#%#COMP-15!!!!!!!" is given as a parameter the
               function returns "COMP-15".

               In addition to the defining the stropNonAlphaNum
               function, you will want a main function in
               stringProcessing.cpp for testing.  Write complete a
               program that reads strings from cin (until reaching end
               of file), calls stripNonAlphaNum on each string and
               prints the result on cout, each result string on a line
               by itself.  Remember: stripNonAlphaNum should be a
               separate function.

               The provide command is:

                   provide comp15 proj2part1 FSTreeTraversal.cpp  \
                                             stringProcessing.h   \
                                             stringProcessing.cpp \
                                             README design.text

               You may include an image file with a diagram of your
               design (under 1 MB in size).
               Your README does not have to contain everything we
               normally ask.  For this phase, you can just write a
               very brief summary with any information you want the
               grader to see, e. g., if you have a nettlesome bug you
               have not been able to fix.

        o Part2 - Final Submission:
            For this part you will submit all of the files required
            (including a Makefile) to compile your gerp program.  The
            filenames for any testing modules you chose to submit should
            begin with "test".  The provide template is:

            provide comp15 proj2part2 README Makefile [YOUR FILE NAMES HERE]

            NOTE:  We cannot give you a complete provide command ---
            so make sure you submit everything we will need to run
            your program.  Maybe copy the files into another directoy,
            type "make" test the program, do a "make clean" and then
            provide every thing.  We should be able to use "make" or
            "make gerp" to build your program.
```

```
--------------
Helpful tricks
--------------

The data sets we're using include a very large collection.  Do not copy
them.  You can pass them directly to your program:

    ./gerp /comp/15/files/proj2-test-dirs/largeGutenberg output.txt

or you can link them into your directory like this:

    ln -s /comp/15/files/proj2-test-dirs ./test-dirs

which will create a "link" in your directory that refers to our directory.
You can use it as if it were a local directory without copying over
gigabytes of data.

We will test your solution on several directories, inluding smallGutenberg,
mediumGutenberg, and largeGutenberg.  You can see some test queries and
reference implementation output in /comp/15/files/proj2-sample-execution
You should match this output exactly (except that the order can be
different as explained above).


---------------
Data structures
---------------

You are welcome to use data structures of your choosing, those we've covered in
class and those we have not.  You are allowed to use the STL data structures
listed under "# STL usage #". All other data structures must be implemented
yourself. Do not forget to cite any source that you use in your
acknowledgements.

There are example uses of the 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.  I. e.,
you must monitor the load factor and expand the hash table if the load
factor is exceeded.

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
essentially constant time to search).

Finally, space 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.

If you are stuck, then you should get something that at least works
correctly on the small and medium collections.  But to get a very high
score, it will also need to work on the larger collection, and that means
choosing a strategy that considers space.

Hint:  Every copy of a string requires space proportional to the length of
the string.  (A 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?"  A vector is an ArrayList, remember.
```