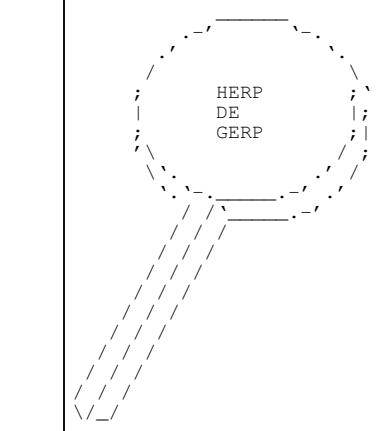
```
**************************************************************************
***                        COMP 15  Project 2                          ***
**************************************************************************
```




```
                                  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. It will behave similarly to running "grep" as shown below. The following Unix command will 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 -Rn ifstream /comp/15/files
```

Will produce:

```
/comp/15/files/hw3/read_from_file.cpp:35: *        cin or an ifstream.
/comp/15/files/hw3/read_from_file.cpp:50:        ifstream input;
/comp/15/files/lab04/Unjumbler.cpp:54:        ifstream infile(filename);
/comp/15/files/proj1/Parser.h:55:        std::ifstream orders;
/comp/15/files/lab08/USE_SETUP!/ImageEngine.cpp:35:     ifstream inputFile;
```

This output tells you that "ifstream" occurs on lines 35 and 50 of /comp/15/files/hw3/read_from_file.cpp, and it prints out the line after the colon.

How can we do something like this? Your program will build an index data structure and use it to to respond to queries like this.

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
```

"DirectoryToIndex" determines which directory will be traversed and indexed. You could use "/comp/15/files", for example.

Once the program indexes the specified directory it will print "Query? " and wait for a command from the user. The possible query commands are the following:

- o AnyString
  A simple string (sequence of non-whitespace characters) 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: Commands will be entered without the quotation marks ("")

If the user did not specify exactly one command line argument or print this message on cerr:

```
        Usage:  gerp directory
                     where:  directory is a valid directory
```

and terminate the program (by returning EXIT_FAILURE from main() or by calling exit(EXIT_FAILURE)). NOTE: the "where" above is indented 12 spaces, and "Usage" has no indentation.

For this assignment you may need to use the Standard Template Library (STL) implementations. 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 define 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 the entire program as "./gerp".

    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
----------------------

    # Output Formatting #
        For this assignment there will only be one specified output format.

        You will print query results to standard output (cout) as follows.
        For example if you queried "we" on our small_test sample data set
        your program should print:

            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.

        The generic output format is as follows:

            FileNameWithPath:LineNum: Line

        Where "FileNameWithPath" is the name and path of the file
        where the queried string appears, "LineNum" is the line number
        of the file on which the queried string appears, and "Line" is
        a reproduction of the line on swhich the queried string
        appears.


    # Using the FSTree and DirNodes #

        The FSTree is the class that we built to help you traverse the
        directory you will have to search, and index.

        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 later). The FSTree class has the following public
        methods:

            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.
            o ~FSTree() -
                    This is the destructor it deallocates all of the data that
                    may have been allocated when the tree was built.
            o void burnTree() -
                    This function destroys the tree and frees any data that
                    was allocated.
```



```
            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 the building block for the FSTree class. It is
        our representation of folders. It 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.
            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 the following STL
        implementations:
            o vector
            o queue
            o stack
            o set
```

To use these STL implementations 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.

Testing

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".

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

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

This command redirects the program's input so that the contents of "commands.txt" appear on cin just as if someone typed them in. Similarly, it redirects the program's output so that whatever is printed on cout gets saved in "ref_output.txt".

You could run your program similarly:

```
./gerp Directory < commands.txt > my_output.txt
```

Then sort both text files using the unix "sort" command as shown below:

```
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.

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.

As a point of reference "the_gerp" indexes all of the files in our largest file tree in approximately 5--6 minutes, and queries almost instantly. Your program will need to be able to index the largest file tree and be ready to query in under 10 minutes. Our implementation takes approximately 5gb of RAM, once it has fully indexed the file tree. Your program can use a max of 10gb of RAM. If you go over the build time or RAM usage you will get a 0 for the functionality portion of your grade.

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. 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.

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.

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
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 3 parts as described below:

       o Part1 - Plan:
           To complete this part, come to a TA's office hours and
           discuss your plan of attack.  Please bring a diagram,
           detailing the different parts of your program, and how
           they interact with each other.  This is your program's
           "architecture."  Also please come in with a list of all
           the special cases you can think of for the search (we
           suggest that you test these out "the_gerp" so that you know
           how the program should behave when it comes to these special
           cases).  This meeting is informal, however it is important
           that you give it a fair amount of meaningful thought.  We
           encourage you to come in early to discuss your plans so that
           you can make the most of the two weeks. To further incentivize
           you to come in early we will be giving out rubber ducks, on a
           first come first serve basis to students who come in with a
           plan where it is evident that they have given it some
           meaningful thought (maximum of one rubber duck per student).
           This part is due on November 17th, however we suggest that
           you come in earlier for the reasons specified above.

       o Part2 - Using the FSTree:
           To complete this part you will need to write a tree traversal
           method that prints out the full paths of each file in the tree
           on separate lines. You should write this method 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. This part will be due on
           November 17th and the provide command is:

               provide comp15 proj2part2 FSTreeTraversal.cpp

       o Part3 - Final Submission:
           For this part you will submit all of the files required (including
           a Makefile) to compile your gerp program.  This is due on the
           November 29th and the provide command is:

           provide comp15 proj2part3 README Makefile [YOUR FILE NAMES HERE]
```