```
**************************************************************************
***                      COMP 15  Homework 4                           ***
**************************************************************************

                ,@@@@@@@,
        ,,,.   ,@@@@@@/@@,  .oo8888o.
     ,&%%&%&&%,@@@@@/@@@@@@,8888\88/8o
    ,%&\%&&%&&%,@@@\@@@/@@@@88\88888/88'
    %&&%&%&%/%&&%@@\@@/ /@@@88888\88888'       Knowing trees, I understand
    %&&%/ %&%%&&@@\ V /@@' `88\8 `/88'         the meaning of patience.
    `&%\ ` /%&'    |.|        \ '|8'           Knowing grass, I can
        |o|        | |         | |             appreciate persistence.
        |.|        | |         | |                              - Hal Borland
     \\/ ._\//_/__/  ,\_//__\\/.  \_//__/_

//////////////////////////////////////////////////////////////////////////
//                                                                      //
//            Tree-Mendous Alphabetizer - Binary Search Tree            //
//                                                                      //
//////////////////////////////////////////////////////////////////////////


In this assignment you will implement an alphabetizer.  In order to
build your alphabetizer you will have to implement a binary search tree
which stores words ("strings" in C++).

We'll first describe the program, and then the requirements of the
classes involved.  Please read through the entire assignment before
you start.


---------------------
Program specification
---------------------

Your alphabetizer will order input strings based on their ASCII
character values (using simple boolean comparisons like ">", and "<").
Your alphabetizer should be able to store a list of words and print
them in either correct alphabetized order or reversed order.

Your program will run in 2 different modes, the "interactive" mode, in
which you can manually type in the words and specify the print order;
and an "automated" mode, in which it will read a list of words from a
file and print those words in a given order.

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

        ./alphabetizer order filename

"order" can either be an "f" --- for "forward" alphabetized order, or
"r" --- for reversed order.  The "filename" specifies the file
containing the words to print in the specified order.  The "words" in
the file are just the strings of non-whitespace characters:  you are
not expected to distinguish lines or worry about non-alphabetical
characters.
```

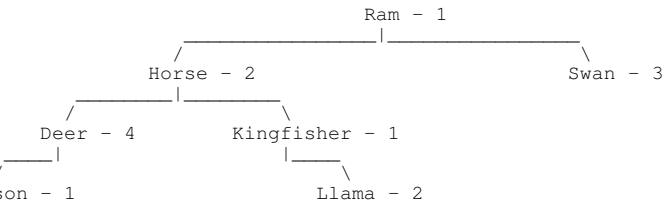


```
Your program will start in "interactive" mode if no command line
parameters are specified:

        ./alphabetizer

If the program is started in "interactive" mode then the first string
read from standard input (cin) will be the order command ("f" or "r"),
then you will read in the list of words as before.  The program should
read in words until it reaches end of file.  Once it reaches end of
file, the program will print list of words in the specified order.  For
example "interactive" mode will take the following input

        f
        Dog
        Cat Fish Aardvark
        Ram
        Dog Ant

and print the following (more on printing syntax later)

        [ cin, f : Aardvark, Ant, Cat, Dog, Dog, Fish, Ram ]

If the user did not specify the correct number of command line parameters
(0 or 2) or specified invalid command line parameters (e.g. order is not
either "f" or "r"), print this message on cerr:

        Usage:  alphabetizer [order filename]

and terminate the program (by returning 1 from main() or by calling
exit(1)).


------------------------
The Classes to Implement
------------------------

For this assignment, you will write two classes, each with an
associated .h and .cpp file (interface and implementation).

     o The StringBST class will have its class definition in a file
       named StringBST.h and its implementation in a file named
       StringBST.cpp.

     o The Alphabetizer class will have its class definition in a file
       named Alphabetizer.h and its implementation in a file named
       Alphabetizer.cpp.

You will want to write test programs to test both of these classes.

We'll describe the (public) interfaces of the classes first, then give
some implementation specifics, and finally submission instructions.
```

Below are the descriptions for the two classes you have to write.

Your StringBST class must have the following interface (all the following members are public):

- Define two constructors for the StringBST class:
  - The default constructor takes no parameters and initializes an empty binary search tree.
  - The second constructor takes an array of strings and an integer specifying the size of the array as parameters and creates a binary search tree with strings inserted in the order in which they appear in the array.
- Define a destructor that destroys/deletes/recycles the heap-allocated data associated with the current binary search tree.
- An "isEmpty" function that takes no parameters and returns a boolean value that is true if this specific instance of the StringBST class is empty and false otherwise.
- A "clear" function that takes no parameters and has a void return type. It makes the current binary search tree into an empty binary search tree.
- A "size" function that takes no parameters and returns an integer value that is the number of strings in the binary search tree.
- A "print" function that takes no parameters and returns nothing. It prints the binary search tree in order. For example, consider a tree we might draw like this:



The tree above will print out *exactly* as follows (all on one line):

```
[[[[[] Bison 1 []] Deer 4 []] Horse 2 [[] Kingfisher 1 [[] Llama 2 []]]] Ram 1 [
[] Swan 3 []]]
```

Notice that the print function prints the number of times each element was added to the tree after the element. Empty trees print out as []. A non-empty tree prints out as 4 items in square brackets separated by a single space: the left subtree, the word in the node, the number of times that word has been inserted into the tree, and then the right subtree.

You'll find this very useful for debugging.

- A "add" function that takes an element (string) and adds it to the binary search tree in the correct location based on the behavior of binary search trees. Your tree should be able to handle adding duplicate elements. For example in the example tree above Horse, Swan, Deer, and Llama were all added multiple times (the number that follows them indicates how many times).
- A "remove" function that takes a target (string) as a parameter and returns a boolean. It will find the target and then proceed to remove one instance of it from the tree. The boolean returned reflects whether or not the function was able to find and remove the target. For example if we removed "Swan" from the example tree above, we would still have 2 instances of "Swan" in the resulting tree. However if we removed "Bison" from the tree we would no longer have any instances of "Bison" in the tree. If we removed "Bird" then the remove function should return "false", and NOT throw an exception.
- A "getMin" function that takes no parameters and returns a string. It returns the left most (smallest) string in the binary search tree. If the tree is empty it throws a C++ "runtime_error" exception with the message "getMin:empty_tree".
- A "getMax" function that takes no parameters and returns a string. It returns the right most (largest) string in the binary search tree. If the tree is empty it throws a C++ "runtime_error" exception with the message "getMax:empty_tree".
- A "removeMin" function that takes no parameters and returns nothing (not a boolean). It removes the left most (smallest) string in the binary search tree. Similarly to "remove", "removeMin" only removes one instance of the left most string.
- A "removeMax" function that takes no parameters and returns nothing (not a boolean). It removes the right most (largest) string in the binary search tree. Similarly to "remove", "removeMax" only removes one instance of the right most string.
- A "contains" function that takes a string as a parameter, and returns a boolean. It will search the binary search tree, and if it finds the query returns true, otherwise it will return false.

```
**************************************************************************
*  NOTE:  ALL OF YOUR TREE FUNCTIONS MUST BE WRITTEN USING A             *
*         RECURSIVE ALGORITHM (NO LOOPS!... EXCEPT IN THE SECOND        *
*         CONSTRUCTOR).                                                  *
*                                                                        *
*         AS YOU MAY HAVE NOTICED, WE HAVE DEFINED THE PUBLIC            *
*         FUNCTIONS SO MANY OF THEM CANNOT EFFECTIVELY BE WRITTEN        *
*         RECURSIVELY.  WRITE PRIVATE HELPER FUNCTIONS THAT UTILIZE      *
*         A RECURSIVE ALGORITHM.                                         *
*                                                                        *
*         THE NAMES OF YOUR FUNCTIONS/METHODS (SHOWN ABOVE IN QUOTES)    *
*         AS WELL AS THE ORDER AND TYPES OF PARAMETERS AND RETURN        *
*         TYPES SHOULD BE *EXACTLY* AS SPECIFIED ABOVE.                  *
*                                                                        *
*         THE ONLY PRIVATE DATA MEMBER THAT YOU CAN HAVE IN YOUR         *
*         TREE IS A POINTER TO THE ROOT OF THE TREE.                     *
*                                                                        *
*         YOU MAY NOT HAVE ANY PUBLIC DATA MEMBERS.                      *
**************************************************************************
```

```
Your Alphabetizer class must have the following interface (all the
following members are public):

- Define one constructor for the Alphabetizer class.  The default
  constructor takes no parameters and initializes an the object.

- Define a destructor that destroys/deletes/recycles any
  heap-allocated data you may have used in the Alphabetizer.

- A "run" function that takes no parameters.  This function will
  launch a process somewhat like the command loop you wrote in
  HW3, reading in commands from the standard input (cin).  Here
  are the commands:

    o a leading "f" or "r" which indicates whether the list of words
      that follows will be printed in forward alphabetical order or
      reverse.  Note: a second "f" or "r" are not order commands but
      valid words to be added to the binary search tree (see next
      bullet point).
    o a word causes the word to be added to the binary search tree.
    o Upon reachin the end of file on cin, the function will
      proceed to print the list that was stored in correct
      alphabetized order or reverse order, whichever was requested.

      The function should print the words in the order that the user
      selected, in the following format:

      [ cin, X : FirstWord, SecondWord, ... , LastWord ]

      Where the "cin" is the origin of the words (cin) and "X" is
      either "f" or "r" depending on the order which the words
      were printed.  If a word appears multiple times in the
      original list it should be printed the same number of times
      in the final sorted list.

[UPDATED:  PLEASE NOTE CHANGE IN RUN FUNCTION SPECIFICATION]

- A second "run" function that takes an input file name (string)
  and an order (string).  This function will read in the list of
  words from the input file stream, and then print them in the
  order specified.  The function should print the words in the
  order that the user selected, in the following format:

  [ Filename, X : FirstWord, SecondWord, ... , LastWord ]

  Where the "Filename" is the name of the origin file of the words
  and "X" is either "f" or "r" depending on the order which the words
  were printed.  If a word appears multiple times in the original list
  it should be printed the same number of times in the final sorted
  list.

**************************************************************************
*  NOTE:  THE NAMES OF YOUR FUNCTIONS/METHODS (SHOWN ABOVE IN QUOTES)  *
*         AS WELL AS THE ORDER AND TYPES OF PARAMETERS AND RETURN      *
*         TYPES SHOULD BE *EXACTLY* AS SPECIFIED ABOVE.                *
*                                                                      *
*         YOU MAY NOT HAVE ANY PUBLIC DATA MEMBERS.                    *
**************************************************************************

You may add private methods.  We particularly encourage the use of
private member functions that help you produce a more modular
solution.  Using private helper functions also help in the better
organization of your code and will help us understand what you
have written.
```



```
    In addition to these two classes you will also need to
    write/implement a main() for your final submission (not testing).
    This main() will be in an eponymous file (main.cpp).  It should
    handle the checks of the command line arguments and if all the
    checks pass it should call the specified run function in the
    Alphabetizer class, otherwise it should execute the specified
    error.


----------------------
Implementation details
----------------------

    Implement the binary search tree using nodes and pointers.  You
    may not use any STL classes:  you must implmement the data
    structures yourself.

    The files "StringBST.h" and "Alphabetizer.h" will contain your class
    definitions only.  The files "StringBST.cpp" and "Alphabetizer.cpp" will
    contain your implementations.

    You saw how to throw an exception in the previous two homeworks.
    Your exceptions MUST be exactly as specified in the function
    descriptions above.

    Create a one or more programs to test your classes.

                *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 can start implementing the classes in either order, however we
    suggest that you implement the StringBST class first.  The
    Alphabetizer class doesn't need a binary search until it starts to
    interpret and execute commands: You can write and debug the
    command reading logic without a binary search tree.  Similarly,
    you can write and debug the binary search tree class without the
    Alphabetizer implemented.

    First, just define a class, #include the .h file in a .cpp file,
    define an empty main function (just return 0), and compile.  This
    tests whether your class definition is syntactically correct.

    If you start with the binary search tree class (follow a similar pattern
    if you start with the Alphabetizer class):

    Implement just the default constructor first. Add a single variable of
    type binary search tree to your test main function, and compile, link,
    and run.

    Then you have some choices. You could add the destructor next,
    but certainly you should add a print function for debugging soon.

    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!

    For both classes, you should add functionality little by little.
```

```
    **************************************************************************
    *  NOTE:  YOU WILL BE SUBMITTING TESTING MAINS ALONG WITH YOUR           *
    *         BINARY SEARCH TREE AND ALPHABETIZER IMPLEMENTATIONS            *
    **************************************************************************

    BSTTesting.cpp should contain testing functions for your StringBST
    class, and AlphaTesting.cpp should contain tests for your
    Alphabetizer class

    You should use the introduction to file input and output and the
    example program that reads strings from a file that we provided
    you with in HW 3.  Study the example, but don't use that code
    exactly: use the information and patterns it shows you to write
    your own command processing code.

    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.


------
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 outline of the data structures and algorithms that you used.
           Given that this is a data structures class, you need to always
           discuss the data structure that you used and justify why
           you used it.  The algorithm overview may not be relevant
           depending on the assignment.

           For this assignment in addition to describing any algorithms
           that you think are interesting you are required to explain the
           algorithm of insert and remove.

        G. 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 that describes the content of the section.
```



```
--------------------
Submitting Your Work
--------------------

    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.

    The command is:

         provide comp15 hw4 BSTTesting.cpp AlphaTesting.cpp StringBST.h   \
                            StringBST.cpp Alphabetizer.h Alphabetizer.cpp \
                            main.cpp README

    Note:  There must not be an space after the '\' and the newline!
```