**PROJECT SPECIFIC TASKS IN C++**

|  |  |
| --- | --- |
| A | For this project you will implement a relatively simple spell checker for English text. The spell checker will incorporate a database of known words, built from a simple word list and organized for efficient searching  The controller will read words from an input text file, one by one, and pass them to the spell checker for processing. If the spell checker finds an exact match, it will simply indicate the word is spelled correctly. If not, the spell checker will provide the controller with a (possibly empty) list of suggested spellings. To select suggestions, the spell checker will need some rules for deciding whether a match is close enough. Suppose that W is the word whose spelling is being checked,and that C is a candidate for the suggested spelling list. Your spell checker will add C to the list of suggestions according to the following rules (in the order given)   * If the lengths of W and C differ by more than two, do not suggest C. * If W is a prefix of C, or if C is a prefix of W, then suggest * Compare the characters of W to those of C, one by one, until reaching the end of one or both strings. If there areno more than two mismatches, and the number of mismatches is less than the lengths of both W and C, suggest C.   These rules are somewhat arbitrary, and certainly too simple for a real spell checker; that's acceptable for this assignment since the primary point is the data structure design, not the matching algorithm  The primary data structures element of this project holds the word list against which spelling will be checked. There are a number of good ways this could be handled. Your implementation is under the following specific requirements:   * The spell checker must store the word list in an AVL tree * The spell checker must be an object. It will contain an AVL tree, but it will not simply be an AVL tree. * You must encapsulate the AVL tree, and its nodes, as C++ templates. * The words must be stored in the tree as C++ string objects. * The AVL tree interface must include only functions that are appropriate for a container, such as insert() and find(). The tree must not take any responsibilities that belong to the controller, such as reading in words to be checked, or to the spell checker itself, such as building the list of suggested alternative words. * For testing and for the demos, your tree must have the ability to display itself to a specified output stream in the manner described in the notes for a BST.   Your design must make appropriate use of classes. Aside from node objects used only within an encapsulating class, data members of classes must be private.  The use of STL containers, such as the vector, is allowed for this project, for example as a way to encapsulate the list of suggestions when carrying out a spell check  **Program execution:**  The program will be invoked with three command-line parameters:  spellcheck <word list file> <file to check> <log file>  The program will verify the existence of the two input files; if either is missing, it will print an error message and exit. |
| B | In this implementation, we'll write the beginnings of a chess program.  In specific, we'll write the part of the program that decides whether a given move is legal or not.   * **Write** a C++ definition of a class named board\_square, which is simply two ints in the range from 1 to 8 (an x coordinate and a y coordinate). * **Write** a C++ definition of a class named*chess\_piece*, with subclasses for the different kinds of chess pieces (king, queen, knight, bishop, rook, pawn), and a method named *is\_legal* that takes in two board\_squares and determines whether the piece can legally move from the first one to the second one. At any given time, a piece has a location (of type board\_square; this changes every time the piece moves), and a color (either "white" or "black", which never changes). * To avoid spending a lot of time on tedious I/O programming, we won't have the program read in chess positions or anything like that; we'll simply have a **main program** that creates a bunch of chess\_pieces (at least one of each kind) and tests the is\_legal method as appropriate on each one.  What are the Rules? No piece can move off the board; that is, the X and Y coordinates of each piece must always be in the range [1...8].  A King  can move one space in any direction: forwards, backwards, left, right, or on any of the four diagonals.  A Queen  can move as far as desired in a straight line in any direction: forwards, backwards, left, right, or on any of the four diagonals. (If there's a piece in the way along a particular straight line, she cannot move past it, but that's not relevant to this assignment.)  A Bishop  can move as far as desired in a straight line along any of the four diagonals, but not forwards, backwards, left, or right. (Again, he cannot move past another piece along the way, but that's not relevant to this assignment.)  A Rook  can move as far as desired in a straight line forwards, backwards, left, or right, but not diagonally. (See above about obstructed paths. There's also a special rule about "castling", which we'll ignore for this assignment.)  A Knight  can move in an L-shape, as though it were going two spaces forwards or backwards and one to either side (or, similarly, two spaces to the side and one space forwards or backwards. For example, if a knight were in the space below marked "K", it could move to any of the spaces marked "x":    A Pawn  can move one space forwards. (There are also rules about capturing and en passant, which are not relevant to this assignment.) For extra credit, implement the rule that on its first move, a pawn may move one or two spaces forwards; a pawn that has already moved at least once, regardless of whether that move was two spaces, can only move one space forwards.Note that "forward" means different things for the two players. A "white" pawn can only move from the top of the board (low Y coordinates) towards the bottom of the board (high Y coordinates), while a "black" pawn can only move from the bottom towards the top. |
| C | The problem is to implement a client - server user-level application using sockets API in C/C++. The Server application has to support at least five clients simultaneously. Server accepts strings from clients (even multiple strings from each client) and replies with reverse strings. For example, when client sends “CHANDIGARH”, Server replies with “HRAGIDNAHC”. Both server and client(s) have to output both sending & receiving strings on the terminal. The server and client processes should be run on different machines. During evaluation, you will be asked to setup up to 5 client processes on different machines and show the outputs.  Repeat the same problem using UDP sockets for implementing the client – server application |
| D | In this project, you should write a simple line editor. Keep the entire text on a file, one line in a key Enter. Start the program with entering EDIT file, after which a prompt appears along with the line number. If the letter ****I**** is entered with a number n following it, then insert the text to be followed before line n. If ****I**** is not followed by a number, then insert the text before the current line. If ****D**** is entered with two numbers ****n**** and ****m****, one n, or no number following it, then delete lines n through m, line n, or the current line. Do the same with the command ****L****, which stands for listing lines. If ****A**** is entered, then append the text to the existing lines. Entry ****E**** signifies exit and saving the text in a file. Here is an Example:  EDIT testfile  1>The first line  2>  3> And another line  4>I3  3> The second line  4> One more line  5> L  1>The first line  2>  3> The second line  4> One more line  5> And another line // This is now line 5, not 3;  5> D2 // line 5, since L was issued from line 5;  4> L //line 4, since one line was deleted;  The first line  2> The second line // this and the following lines  3> One more line // now have new numbers  4> And another line  4> E |