Daniel Yoh

CSCI 313-22

#1. Comment ALL of the code in the file “stack linked list.txt” and “postfix.txt”

The code in “stack linked list.txt” creates a stack linked list. First, header guards are declared with #ifndef and #define. Then <iostream> and <cassert> are also included, along with namespace std. A header file “stackADT.h” is brought in so a stack is defined for us to be able to use.

Code is written so that linked lists are able to be used, declaring them as structures in a template. They are given ‘info’ so they can hold data, and *‘link’ so that they can point to the address of the next node. Then the stack code is written, which is inherited from “stackADT.h”. The assignment operator is overloaded, and various member functions are written for a variety of purposes, such as checking if the stack is empty, removing all elements in the stack, and pushing or popping elements into/from the stack. Then the stack is given a default constructor, a copy constructor, and a destructor to complete the code for its class. The pointer to the stack top is declared private, along with the copyStack function.

The default constructor sets the stack top to point to null because there’s no element above it. More functions are defined, allowing the user to check if the stack is empty, full, push a node into the stack, etc. For the initializeStack function, a temporary pointer is used to delete the node. It is used to point to the top of the stack, then before it’s deleted, stackTop points to the next node so that the code can loop and the pointer points to all of the next nodes until all nodes have been deleted. For the code that pushes a new node into the stack, a new node is created, its data is set, then its link is set to point to the top of the stack. Then the stackTop is set

to the new node. For the top() function, assert is used to check if the stack is empty. If so, the program is terminated. Otherwise, it returns the data of the top node in the stack.

For the copyStack function, three new node pointers are declared. It checks if the stack is empty, and if not, then it runs initializeStack which makes the stack empty. Then it checks if otherStack has an empty top pointer. If so, it sets the current stack top to point to null. Otherwise, it sets current to point to otherStack. New node stackTop is created, data is passed in, and it points to null because there's no element after it. Last is set to stackTop, and current is pointed towards the next node, where it will loop in a while loop to copy the remainder of the stack with new nodes until it reaches the final node, when current will be pointing at nullptr.

Finally, a copy constructor, destructor, and assignment operator overload are coded. And #endif is used to mark the end for the earlier two header guard codes #ifndef and #define.

For "postfix.txt", the program calculates postfix expressions. Iostream, iomanip, fstream, and "mystack.h" are included. They allow for input and output streams, operations on iostream objects, files to be opened and altered, and allows for the creation and usage of stacks. Four void-type functions are declared to be used later, and they allow expressions and operands to be read from files and evaluated, discarded, or printed after they are converted into postfix expressions.

A bool, char, stack containing doubles, ifstream, and ofstream are declared. A text file is opened through infile, and the program checks if it is openable. If not, the program exits with a message. Then another file, RpnOutput, is opened, where the postfix expressions will be written to. Fixed, showpoint, and setprecision(2) are used on outfile to show floating point numbers with decimals up to hundredths place instead of in scientific notation. Infile assigns to ch the

first character, then a while loop is used to loop through all of the chars in the text file to the end. The stack is emptied, expressionOK is set to true, and char is written to outfile. Then evaluateExpression is called.

In evaluateExpression, a double is declared, then a while loop with a switch-case statement. The while loop iterates through every character in the text file until it reaches the equal sign. Ch is passed into switch(), and for the case where the char is a number, that number is assigned to num, which is then written to the outfile. Then it checks if the stack is full (size over 100) before pushing that number onto the top. If it is full, the program terminates with an error message. The switch-case statement breaks, continuing the loop. The default case, where char is not a number and thus most likely an operator (the only two characters that would be present in an equation would be a number and an operator), calls evaluateOpr.

In evaluateOpr, two new doubles are used. The stack is checked, and if it is empty, then it sets expression to not OK, and 'Not enough operands' is written to outfile. In the else statement, when stack is not empty, op2 is assigned the top of the stack, and the top of the stack is popped. The same check on the empty/non-empty stack is performed, and if it's not empty. op1 is assigned the top of stack, which is popped. A switch case statement is used to account for every operator (addition, subtraction, multiplication, division). Depending on the case, different results of the expressions are pushed to the top of the stack. A unique if statement is used for division for dividing by 0. The default cause is for illegal operators, and then isExpOK is set to false.

Back in evaluateExpression, an if statement checks if the expression is valid. If so, the next character is written from the input file into char, then into outfile. If char is not a number, then it writes a space to outfile. If the expression is not valid, it is discarded with discardExp. In

discardExp, a simple while loop is used to traverse the expression until the = sign and write it to outfile.

Back in the main function, after evaluating an expression, printResult is called. It uses multiple if statements to check if the stack is empty, sets a double equal to the top of the stack (which, at this moment in the program, should be the result of evaluateExpression), pops it, and writes the result to the outfile. It can also write an error to outfile, saying that there's too many operands or the expression simply contains an error depending on the error in the written equations in the original infile. Back in main, infile is set to the next character, where it will loop until all the expressions are evaluated. Then it will close the infile and outfiles.