Assignment #4

20190741 김 지수

Q1) Maze Searching

Note:

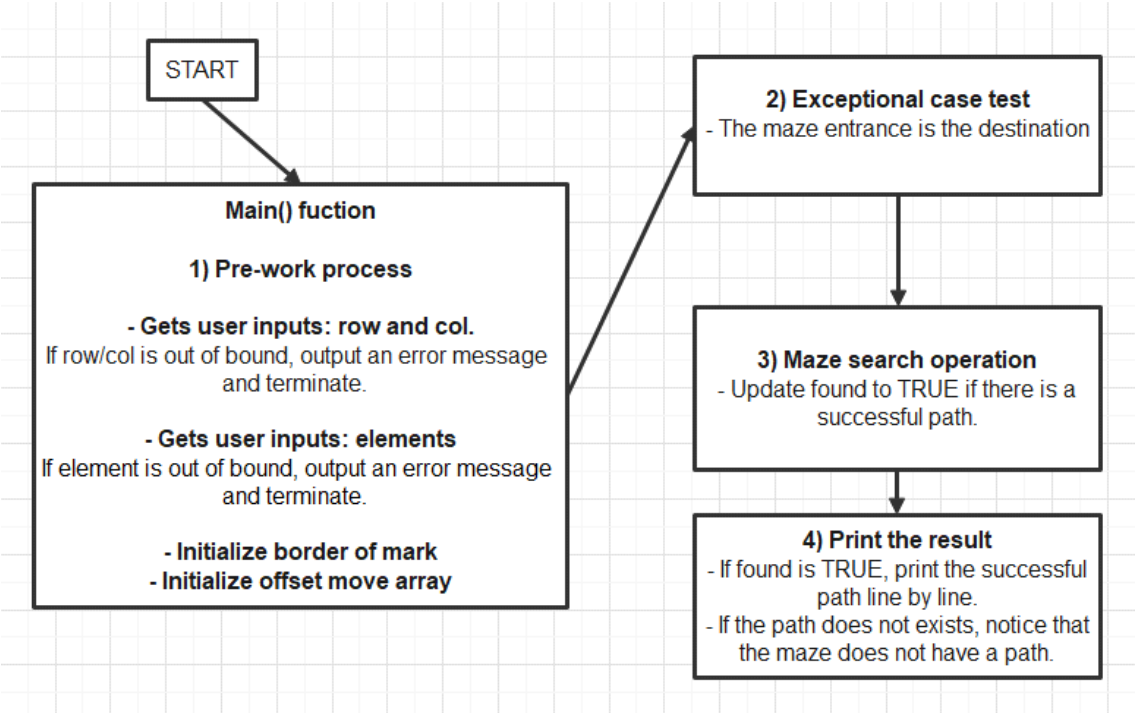
1) The program uses standard input/output with some requirements. Number of input rows and columns must between 1 to 20.(1<=length<=20) If the input number is out of range, the program will be terminated with error message. Notice that the output message was in Korean in the HW4 pdf file, it is translated into English in line with the requirement.

2) The user will input the number of rows and columns first, and then input each element of the matrix. It is assumed that each element of the matrix is inputted correctly by the user in line with the number of rows and columns that are previously entered by the user.

3) Note that the result of path could be different with others. Since the student does not want to allow the inefficiency of algorithm that searches maze, the searching order of direction is modified. It is inefficient to search North and Northeast first in the maze searching as it is opposite way from the destination, the program searches the East position first and Northeast position at last.

Q1-1. Code Description with Flow chart

The flow chart of the Q1 is as follows. When the program starts, the main() function gets user input about the number of rows and columns, and each corresponding element. In this process, the mark matrix is initialized. Next, the program initializes the border of maze to 1 in order to check the boundary of the maze. After the pre-work process, the program searches maze whether there is a successful path and prints the result accordingly. Each detailed information about the searching process will be explained at Q1-2 with pseudo code.



<Flow chart, Q1>

Q1-2. Main Algorithms with Pseudo code

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#define MAX_STACK_SIZE 500
#define MAX_ROW 20
#define MAX_COL 20
#define TRUE 1
#define FALSE 0

int maze[MAX_ROW+2][MAX_COL+2];
int mark[MAX_ROW+2][MAX_COL+2];

typedef struct {
            short int vert;
            short int horiz;
}offset;
offset move[8];

typedef struct {
            short int row;
            short int col;
            short int dir;
}element;
element stack[MAX_STACK_SIZE];

int top;
void push(element);
element pop();
void stackFull();
element stackEmpty();

void main() {
    int i, j, row, col, nextRow, nextCol, dir, found = FALSE, item;
    int exitRow, exitCol;
    element position;

    Gets number of rows(row) using standard input;
    if (input row is out of bound) {
        Print an error message and terminate;
    }

    Gets number of columns(col) using standard input;
    if (input col is out of bound) {
        Print an error message and terminate;
    }

    exitRow = row;
    exitCol = col;

    Get each element of maze by nested for loop and check if the maze element is either 0 or 1;
    If the element is not 0 or 1, the program prints an error message and terminate;
    Initialize the border of maze. Then maze[1][1] is the starting point and maze[row][col] is the destination;
```

```
    Set more greedy version of move like below;
    move[0].vert = 0; move[0].horiz = 1;     // East
    move[1].vert = 1; move[1].horiz = 1;     // SE
    move[2].vert = 1; move[2].horiz = 0;     // South
    move[3].vert = 1; move[3].horiz = -1;     // SW
    move[4].vert = 0; move[4].horiz = -1;     // West
    move[5].vert = -1; move[5].horiz = -1;     // NW
    move[6].vert = -1; move[6].horiz = 0;     // North
    move[7].vert = -1; move[7].horiz = 1;     // NE

    Set starting point;
    mark[1][1] = 1; top = 0;
    stack[0].row = 1; stack[0].col = 1; stack[0].dir = 0;

    Handle an exceptional case where the maze entrance is the final destination;
    if (row == 1 && col == 1 && maze[1][1] == 0) {
        printf("The path is: \n");
        printf("row col\n");
        printf("%2d%5d\n", row, col);
        exit(1);
    }
    while (stack is not empty) {
        position = pop();
        row = position.row; col = position.col; dir = position.dir;
        while (there are more moves from current position) {
            /* move in direction dir */
            nextRow = row + move[dir].vert;
            nextCol = col + move[dir].horiz;
            if (next position is the destination && next position is possible to move on)
                found = TRUE;
            else if (next position is legal move and haven't been there) {
                mark[nextRow][nextCol] = 1;
                save current position and next direction to the top of the stack;
                row = nextRow; col = nextCol; dir = 0;
            }
            else ++dir;
        }
    }
    if (there is a successful path) {
        Output the successful path;
    }
    else Print that the maze does not have a path;
}

void stackFull() {
    Output an error message and terminate program;
}

void push(element item) {
          if (stack is full) {
                    stackFull();
          }
          else insert item at the top of the stack;
}
```

```
element pop() {
        if (stack is empty)
                return stackEmpty();  /* error handle */
        else remove and return the element at the top of the stack;
}

element stackEmpty() {
        element empty;
        empty.row = -1; empty.col = -1; empty.dir = -1;
        return empty position(-1 is invalid but top controls the whole system);
}
```

<Pseudocode, Q1>

As this program is coded based on the maze searching program in the course material, the algorithm explanation is added about the different part compared to the original function. The original maze search function cannot handle the situation when the entrance is the final destination. That means, input row and col are both 1 and corresponding element is 0. This program handles the exceptional case by using additional if statement as seen in pseudocode.

Also note that, on the original function, the function says there is a successful path before the checking process, that is, whether the maze[nextRow][nextCol] is possible to move on. This could bring a serious problem. Thus, in this program the first if statement in the while loop checks whether the next position is the final destination that we could move on. As a result, the program operates correctly and the results are shown in the next part.

Q1-3. Test cases simulated

In order to show the output result within the page, the divided output cell is not in a same line. It means, each path <row, col> element will be output line by line. Note that a path resu lt start at <1, 1> and ends at <row, col>, if successful path exists, in line with the course mat erial chapter 3 slide 21. Also, at the below input table, it only contains the user input values. In the program, the detailed input instructions about input values will also be printed for guid ance purposes.

<Test results, Q1>

Input	Output
20 20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 1 1 0 1 0 1 1 0 1 0 1 0 1 1 0 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 1 0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0 1 0 0 1 1 1 1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0	The path is: row col 1 1 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 10 1 11 2 10 3 10 4 10 5 10 6 10 7 10 8 10 9 10 10 10 10 9 10 8 10 7 10 6 10 5 10 4 10 3 10 2 11 1 12 2 12 3 12 4 12 5 12 6 12 7 12 8 12 9 12 10 13 11 12 12 12 13 12 14 12 15 12 16 12 17 12 18 13 17 14 16 15 17 16 16 17 16 18 15 19 16 19 17 19 18 20 19 20 20

20 20 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 1 1 0 1 1 0 1 0 1 1 0 1 0 1 0 1 1 0 1 1 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 0 0 0 1 0 0 0 0 0 0 1 0 1 0 0 0 1 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 1 1 0 1 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 1 1 0 0 1 0 1 0 1 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0 1 0 0 1 1 1 1 0 0 0 1 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 0 1	The maze does not have a path.
1 1 0	The path is: row col 1 1
0	Input number of Rows is out of bound. Program terminates.
14 0	Input number of Columns is out of bound. Program terminates.
20 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	The path is: row col 1 1 2 1 3 1 4 1 5 1 6 1 7 1 8 1 9 1 10 1 11 1 12 1 13 1 14 1 15 1 16 1 17 1 18 1 19 1 20 1

9 6 0 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0 1 0 1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 0 1 0 0 0 0 1 0 1 1 1 1 1 1 0 0 0 0 0	The path is: row col 1 1 1 2 1 3 1 4 1 5 2 6 3 5 3 4 3 3 3 2 4 1 5 2 5 3 5 4 5 5 6 6 7 5 7 4 7 3 7 2 8 1 9 2 9 3 9 4 9 5 9 6
11 15 0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1 0 0 0 1 1 0 1 1 1 0 0 1 1 1 0 1 1 0 0 0 0 1 1 1 1 0 0 1 1 1 1 0 1 1 1 1 0 1 1 0 1 1 0 0 1 1 0 1 0 0 1 0 1 1 1 1 1 1 1 0 0 1 1 0 1 1 1 0 1 0 0 1 0 1 0 1 1 1 1 0 0 1 1 1 1 1 1 1 1 0 0 1 1 0 1 1 0 1 1 1 1 1 0 1 1 1 0 0 0 1 1 0 1 1 0 0 0 0 0 0 0 1 1 1 1 1 0 0 0 1 1 1 1 0 0 1 0 0 1 1 1 1 1 0 1 1 1 1 0	The path is: row col 1 1 2 2 2 3 2 4 3 5 3 4 4 3 5 3 6 2 7 1 8 2 9 3 9 4 9 5 8 5 7 6 7 7 8 8 9 8 10 9 10 10 9 11 9 12 9 13 9 14 9 15 10 15 11 15
3 3 0 0 0 0 1 1 0 0 0	The path is: row col 1 1 1 2 2 1 3 2 3 3

As simulating the many test cases like upper table, the program could be more reliable by checking the corner cases. As a result, there is no program error or incorrect result found for several test cases by comparing the test cases results with other students.

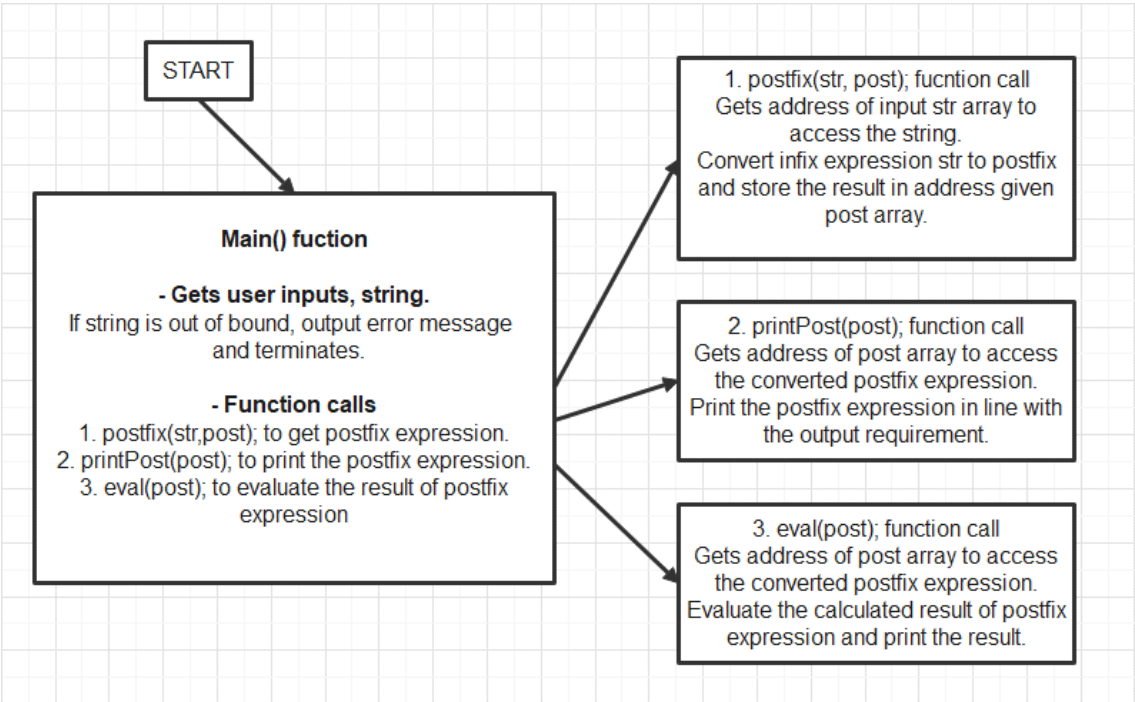
Q2) Infix to Postfix with unary minus operator added

notes)

1. If a length of the given string does not meet the requirements(1<= length <=20), program p rints the error message and terminates.
2. As the program uses standard input/output scanf(), it is assumed that the input expression string does not have blank space within the expression.
3. It is assumed that input string of this program only contains single digit operand, 6 operat ors, and 2 parenthesis in line with the Q2 input requirement. That means, as it is already ass umed about the input string, the program does not check if the string contains other operand s or operator that are out of the requirement.
4. Push/pop function is implemented separately based on their data type.
5. Q2-3 part also explains how the program works when the input string is only a single oper ator. However it is assumed that only a single bracket, either '(' or ')', could not be given sin ce it is not an expression. This is fully explained at Q2-3.

Q2-1. Code Description with Flow chart

The flow chart of the Q2 is as follows. When the program starts, main function gets user in puts, string. If the string is out of bound, the program outputs error message and terminates. Otherwise, if the input is valid, 3 Functions are called sequentially as seen in the <Flow chart, Q2> below. Specific information will be introduced at Q2-2 part.



<Flow chart, Q2>

Q2-2. Main Algorithms with Pseudo code

In this program, there are many functions that is purposely made for structured programmi ng. So, in this part, each function is explained below. As it is required not to use global varia ble, all functions below get the address of the needed data such as expr, top, stack, post, etc.

Function	Explanation
void eval(char *expr);	Eval function is almost the same as the program in the course material. However, as the program have to evaluate the proper value with the unary minus operator, the additional if statement handles this operation as seen in the <Pseudocode, Q2> below.
void pushInt(int item, int *top, int stack[]);	PushInt function pushes an integer value(item) at the top of the given stack if stack is not full. If the given stack is full, it calls stackFull error handling function.
int popInt(int* top, int stack[]);	PopInt function pops an integer value at the top of the given stack if stack is not empty. If the stack is empty, it directly handles the error because it is obvious that something went wrong about calculation result. It is better to notify the error than outputting a wrong result.
void pushPre(precedence item, int *top, precedence stack[]);	PushPre function pushes an precedence value(item) at the top of the given stack if stack is not full. If the given stack is full, it calls stackFull error handling function.
precedence popPre(int* top, precedence stack[]);	PopPre function pops an precedence value at the top of the given stack if stack is not empty. If the stack is empty, it directly handles the error because it is obvious that something went wrong since eos is always at the bottom of the stack.
void stackFull();	Prints error message and terminates execution.
void postfix(char *expr, char *post);	Postfix function is similar to the course material. However, in this function the additional if statement is added to identify the unary minus operator. Also, in this function, the final results are stored in the reference given post array rather than directly printing in order to reuse the final result in the eval function.
char convertToken(precedence token);	ConvertToken function converts the input precedence token into its corresponding character and returns it.
void printPost(char post[]);	PrintPost function gets the address of post fixed array and prints postfix expression in line with the output requirement.

Specifically, the main key idea of identifying whether the '-' is an unary minus operator or not is explained as follows. There are two cases : the given '-' is at the first index of the exp ression, or not. If it is first, then it is unary minus operator. Otherwise, before the '-' operat or, there can be '(', ')', '+', '-', '*', '/', '%', or an operand. In these cases the '-' is unary mi nus operator except for the situation at which either ')' or an operand is at right before it. In this algorithm, it is considered that even when the '-' is found like --, *-, /-, %-, the progra

m automatically considers it as an unary minus operator although there is no left parenthesis before the unary minus operator as we calculate 6--1 = 6-(-1) = 7 like python does.

```
>>> 6--1
7
>>> 6*-1
-6
>>> 6/-1
-6.0
>>> 6%-1
0
>>> 6+-1
```

The pseudocode of Q2 is as follows.

```
#define _CRT_SECURE_NO_WARNINGS
#define MAX_STACK_SIZE 100
#define MAX_EXPR_SIZE 20
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef enum { lparen, rparen, plus, minus, times, divide, mod, eos, unaryminus, operand } precedence;
precedence getToken(char *symbol, char *expr, int *n);
void eval(char *expr);
void pushInt(int item, int *top, int stack[]);
int popInt(int* top, int stack[]);
void pushPre(precedence item, int *top, precedence stack[]);
precedence popPre(int* top, precedence stack[]);
void stackFull();
void postfix(char *expr, char *post);
char convertToken(precedence token);
void printPost(char post[]);

void main() {
        char str[100000];
        char post[100000];
        int len;
        Gets input string str, using standard input;
        len = strlen(str);
        if (length of string is out of bound) {
                fprintf(stderr, "Input string size is out of bound. Program terminates.\n");
                exit(1);
        }

        call postfix(str, post); function to convert infix to postfix;
        call printPost(post); function to print postfix expression;
        call eval(post); function to calculate result using converted postfix expression;
}

precedence getToken(char *symbol, char *expr, int *n) {
        Gets a token and returns it to a corresponding precedence type equivalent;
}

void eval(char *expr) {
```

```
        /* evaluate a postfix expression and print the calculated result*/
        int top = -1, op1, op2, n = 0;
        int stack[MAX_STACK_SIZE];
        precedence token;
        char symbol;

        Gets a new token from the given expression;

        while (token is not the end of string) {
                if (token is operand) {
                        Push its integer value into the stack;
                }
                else if (token is unaryminus) {
                         Pop a operand and push calculated result into the stack;
                }
                else {
                        Pop 2 operands and push calculated result into the stack;
                }
                Gets a new token from the given expression;
        }
        Output(print) the final result;
}

void postfix(char* expr, char* post) {
        char symbol;
        int n = 0, top = -1, count = 0;
        int isp[] = {0, 19, 12, 12, 13, 13, 13, 0, 15};
        int icp[] = { 20, 19, 12, 12, 13, 13, 13, 0, 15};
        precedence stack[MAX_STACK_SIZE];
        precedence token;
        stack[++top] = eos;

        for (Gets a new token until the new token is end of string) {
                if (token == minus && n < 2) {
                        That token is unary minus operator;
                }
                else if (token == minus && !(expr[n - 2] >= 48 && expr[n - 2] <= 57) && expr[n-2] != ')') {
                        That token is unary minus operator;
                }

                if (token is operand) {
                        Update post array;
                }
                else if (token is rparen) {
                        Unstack and update post array until left parenthesis;
                        Discard the left parenthesis;
                }
                else {
                Unstack and update post array whose isp is greater than or equal to the current token's icp;
                Push current token;
                }
        }
        Update post array for remaining(s) and close by '\0';
}
```

```
void pushInt(int item, int* top, int stack[]) {
        if (stack is full) {
                stackFull();
        }
        else insert item at the top of the stack;
}

int popInt(int* top, int stack[]) {
        if (stack is empty) {
                Output an error message and terminate program;
        }
        else remove and return the element at the top of the stack;
}

void stackFull() {
        Output an error message and terminate program;
}

char convertToken(precedence token) {
        Convert the input precedence token into its corresponding character and return it;
}

void pushPre(precedence item, int* top, precedence stack[]) {
        if (stack is full) {
                stackFull();
        }
        else insert item at the top of the stack;
}

precedence popPre(int* top, precedence stack[]) {
        if (stack is empty) {
                Output an error message and terminate program;
        }
        else remove and return the element at the top of the stack;
}

void printPost(char post[]) {
        Print given postfix expression in line with the output requirement;
}
```

<Pseudocode, Q2>

Q2-3. Test cases simulated

As simulating the test cases could make the program more reliable by finding the corner cas es, cases are tested.

Input	Output
Input : -6	Postfix : 6# Result : -6
Input : (1-(-3)-5)	Postfix : 13#-5- Result : -1
Input : 3*2+4*(5-1)	Postfix : 32*451-*+ Result : 22

Input : 2+3	Postfix : 23+ Result : 5
Input : 2+3*4	Postfix : 234*+ Result : 14
Input : (2+3)*4	Postfix : 23+4* Result : 20
Input : -2+(-3)*4	Postfix : 2#3#4*+ Result : -14
Input : -2+-(-3)*4	Postfix : 2#3##4*+ Result : 10
Input : (2+3)*(4-1)/5	Postfix : 23+41-*5/ Result : 3
Input : -2*-3	Postfix : 2#3#* Result : 6
Input : -(-5+2*3%5+(2*3)-7)-5	Input string size is out of bound. Program terminates.
Input : -(-5+2*3%5+(-2*3))-7	Postfix : 5#23*5%+2#3*+#7- Result : 3
Input : 9	Postfix : 9 Result : 9
Input : ()	Postfix: Stack is empty. Cannot remove element.
Input : (((((()()()()())))))	Postfix: Stack is empty. Cannot remove element.
Input : +	Postfix : + Stack is empty. Cannot remove element.

Some additional corner cases are tested. The tricky input string could be just one operator. In this case, the program just says “Stack is empty. Cannot remove element.” and terminates rather than outputting the wrong result(return -1 when stack is empty) in order to identify th ere is a problem that a number of operand(s) does not properly match with that of operator (s). Also, as it is noted earlier, a single bracket is not considered as an expression. It is assu med that Parenthesis must be a pair. Thus, when the () is inputted, the result tells that there are no element to calculate as the stack is empty(that means no elements to calculate the res ult) and the postfix result is empty. As a result, there was no program error or incorrect res ult found for several test cases.