National University of Computer & Emerging Sciences

Karachi Campus

Data Structures (CS201)

Programming Assignment #3

Instructions

Each Problem of 15 points

Due Date: The assignment is due by 29th October 2008, 11:59 PM
Submission: The assignment has to be submitted via e-mail. You must submit the source code as well as the executable file. You should mail the assignment to cs201@nu.edu.pk, preferably from the institute mail account. With subject like: "Assignment # 3 – 07-0033"
Sample Input and Output files: The sample input and output files are available from Khi website in course section. Make sure that you have tested your programs against all the available input files and EXACT output file is produced.

Problem # 1 Infix to Postfix Conversion

Polish notation, also known as prefix notation, is a form of notation for logic, arithmetic, and algebra. Its distinguishing feature is that it places operators, whose arity is assumed known, to the left of their operands. The result is a syntax lacking parentheses or other brackets that can still be parsed without ambiguity. Similarly Reverse Polish Notation is postfix notation. Human beings are very proficient with infix notation while computer can process prefix or postfix expressions efficiently. We have already discussed the algorithms in class but for the sake of implementation I am giving the same over here.

Converting Expression from Infix to Postfix using STACK

To convert an expression from infix to postfix, we are going to use a stack.

Algorithm:

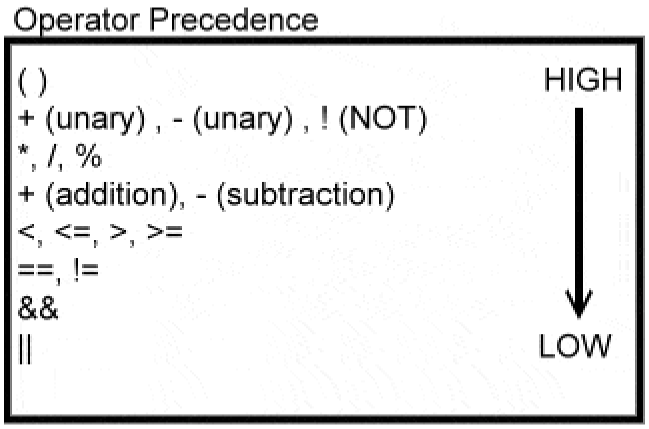
```
1) Examine the next element in the input.
2) If it is an operand, output it.
3) If it is opening parenthesis, push it on stack.
4) If it is an operator, then

        i) If stack is empty, push operator on stack.
        ii) If the top of the stack is opening parenthesis, push operator on stack.
        iii) If it has higher priority than the top of stack, push operator on stack.
        iv) Else pop the operator from the stack and output it, repeat step 4.

5) If it is a closing parenthesis, pop operators from the stack and output them until an
opening parenthesis is encountered. pop and discard the opening parenthesis.
```

6) If there is more input go to step 1
7) If there is no more input, unstack the remaining operators to output.

Following figure shows operator Precedence in descending order.


Operator Precedence
( )
+ (unary) , - (unary) , ! (NOT)
*, /, %
+ (addition), - (subtraction)
<, <=, >, >=
==, !=
&&
||
HIGH
LOW

Converting Expression from Infix to Prefix using STACK

It is a bit trickier algorithm, in this algorithm we first reverse the input expression so that a+b*c will become c*b+a and then we do the conversion and then again the output string is reversed. Doing this has an advantage that except for some minor modifications the algorithm for Infix->Prefix remains almost same as the one for Infix->Postfix.

Algorithm:

1) Reverse the input string.
2) Examine the next element in the input.
3) If it is operand, add it to output string.
4) If it is closing parenthesis, push it on stack.
5) If it is an operator, then

    i) If stack is empty, push operator on stack.
    ii) If the top of stack is closing parenthesis, push operator on stack.
    iii) If it has same or higher priority than the top of stack, push operator on stack.
    iv) Else pop the operator from the stack and add it to output string, repeat step 5.

6) If it is an opening parenthesis, pop operators from stack and add them to output string until a closing parenthesis is encountered. Pop and discard the closing parenthesis.
7) If there is more input go to step 2
8) If there is no more input, unstack the remaining operators and add them to output string.
9) Reverse the output string.

Evaluating postfix expression by using STACK

Algorithm:

1) Scan the Postfix string from left to right.
2) Initialise an empty stack.
3) If the scannned character is an operand, add it to the stack. If the scanned character is an operator, there will be atleast two operands in the stack.

    i. If the scanned character is an Operator, then we store the top most element of the stack(topStack) in a variable temp. Pop the stack. Now evaluate topStack(Operator)temp. Let the result of this operation be retVal. Pop the stack and Push retVal into the stack.

4) Repeat this step till all the characters are scanned.

5) After all characters are scanned, we will have only one element in the stack. Return topStack.

Problem # 2   Queuing Systems Simulation

The Shop-n-Carry (SC) is a big chain of departmental stores. The SC would like to improve its customer services. There has been a complained for some period that the waiting time to avail Point-of-Sale terminal services is unbearable for some areas in stores. SC is planning to improve the services by means of installing more POS terminals to those stores, which are in heavy use, and customers have to wait for making payments. You have been hired, as a consultant by the SC president to determine weather this idea is feasible or not. The management has decided that if the average waiting time of a user at a POS is greater than 3 minutes to get their services. They will install one or more POS terminal machine. It is provided that the arrival rate of POS customer is 0-4 minutes and the average service time is between 2 to 8 minutes. The POS is operational 24 X 6. Your job is to run a simulation program for SC POS for 24 hours, and calculate the average waiting time for customers on daily basis. Use the least count of the clock as discrete unit of minutes. Run the simulation for about 24 hours time and put the entire statistic in the output file which is as follows:

- Number of customers per day (– in a separate line)
- Average waiting time per day(– in a separate line)
- Average Customers per (– in a separate line)
- POS utilization time per day (Average Service Time)

Problem # 3 RoboCop

You are to implement the game of RoboCop, a multiplayer capture-the-thief game where two or more robots compete to be the first to traverse a maze and capture the thief.

In this assignment, you are required to implement the basic functionality of this game with just one robot.

The Maze: The maze consists of a two-dimensional grid of squares. A square can only be in one of two states: open or closed. Open squares are denoted with a period, "`.`", and closed (walled) squares are denoted with an asterisk, "*****". The thief is labeled "T", and is hiding from the cops. For simplicity, assume that the thief doesn't move. Each maze has an implicit row of walls around the outside. (The robot can not fall off the side of the maze.) Thus, a maze can have a passage along the edge of the maze.

The Robot: A robot can point in one of four directions:

```
1. North:   '^'
2. South:   'V'
3. East:    '>'
4. West:    '<'
```

As the robot navigates through the maze, it can make one of three possible moves:

```
1. move forward
2. turn left
3. turn right
```

A robot must choose its next move based only on information from the squares immediately surrounding it, and the information gathered from past moves. In other words, the robot ***cannot*** see the entire board on every move

Hints:

- Aim for the recursive solution as it is small and easy.
- When given a choice of directions to move, your robot must move in each direction before backtracking. Be careful! One of the directions - West, North, East, or South - will be the direction from which your robot entered the square.
- Your robot should always move to minimize the number of game turns it must take to find the flag. If your robot needs to turn around (turn 180 degrees), it should turn clockwise, otherwise it should turn in the direction that minimizes the number of game turns. (For example, if it is facing North, and wants to turn to face South, it should first turn to the East, and then turn to the South. If, however, it is facing North, and wants to turn to face West, it should turn immediately to the West. It should ***not*** turn East, South, and then West.)

Input: The file will start with the starting location of the robot, and the facing of the robot. This is followed by rows, columns of the maze and this is followed by the maze which is character 2-dimensional array. You can use Array2D <char> class to read this array.

```
9 4 N    // The robot's starting row, column, and facing [ N, S, E, W]
10 // number of rows in the maze
10 // number of columns in the maze
*******T**
*.......**
***.******
**..*....*
*..**.**.*
*.******.*
*........*
****.*****
****.....*
****.*****
```

Output: The output will consist of the path that the robot followed to locate the flag.

```
9 4 N
8 4 N
7 4 N
...
```

Bonus: Separate the logic of storage of maze and the Robot so that two robots can compete on the same maze to find the flag. Do store the current position of the robots on the board, so as to check for collision of the robots.

<The End>