**Homework 2 Solution**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [Problem 1](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/2/solution.html#P1) | [Problem 2](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/2/solution.html#P2) | [Problem 3](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/2/solution.html#P3) | [Problem 4](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/2/solution.html#P4) | [Problem 5](http://www.cs.ucla.edu/classes/spring12/cs32/Homeworks/2/solution.html#P5) |

**Problem 1:**

#include <stack>

using namespace std;

const char WALL = 'X';

const char OPEN = '.';

const char SEEN = 'o';

class Coord

{

public:

Coord(int rr, int cc) : m\_r(rr), m\_c(cc) {}

int r() const { return m\_r; }

int c() const { return m\_c; }

private:

int m\_r;

int m\_c;

};

void explore(char maze[][10], stack<Coord>& toDo, int r, int c)

{

if (maze[r][c] == OPEN)

{

toDo.push(Coord(r,c));

maze[r][c] = SEEN; // anything non-OPEN will do

}

}

bool pathExists(char maze[][10], int sr, int sc, int er, int ec)

{

if (sr < 0 || sr > 9 || sc < 0 || sc > 9 ||

er < 0 || er > 9 || ec < 0 || ec > 9 ||

maze[sr][sc] != OPEN || maze[er][ec] != OPEN)

return false;

stack<Coord> toDo;

explore(maze, toDo, sr, sc);

while ( ! toDo.empty() )

{

Coord curr = toDo.top();

toDo.pop();

const int cr = curr.r();

const int cc = curr.c();

if (cr == er && cc == ec)

return true;

explore(maze, toDo, cr-1, cc); // north

explore(maze, toDo, cr, cc+1); // east

explore(maze, toDo, cr+1, cc); // south

explore(maze, toDo, cr, cc-1); // west

}

return false;

}

**Problem 2:**

(1,1) (2,1) (3,1) (1,2) (1,3) (1,4) (2,4) (3,4) (3,3)

**Problem 3:**

Make three changes to the Problem 2 solution:

* Change #include <stack> to #include <queue>
* Change stack<Coord> to queue<Coord>
* Change Coord curr = toDo.top(); to Coord curr = toDo.front();

**Problem 4:**

(1,1) (1,2) (2,1) (1,3) (3,1) (1,4) (2,4) (3,4) (3,5)

The stack solution visits the cells in a *depth-first* order: it continues along a path until it hits a dead end, then *backtracks* to the most recently visited intersection that has unexplored branches. Because we're using a stack, the next cell to be visited will be a neighbor of the *most* recently visited cell with unexplored neighbors.

The queue solution visits the cells in a *breadth-first* order: it visits all the cells at distance 1 from the start cell, then all those at distance 2, then all those at distance 3, etc. Beacuse we're using a queue, the next cell to be visited will be a neighbor of the *least* recently visited cell with unexplored neighbors.

**Problem 5:**

// eval.cpp

#include <string>

#include <stack>

#include <cctype>

#include <cassert>

using namespace std;

inline

bool isLetterOrCloseParen(char ch)

{

return ch == 'T' || ch == 'F' || ch == ')';

}

inline

int precedence(char ch)

// Precondition: ch is in "|&!("

{

static string ops = "|&!(";

static int prec[4] = { 1, 2, 3, 0 };

int pos = ops.find(ch);

assert(pos != string::npos); // must be found!

return prec[pos];

}

const int RET\_OK\_EVALUATION = 0;

const int RET\_INVALID\_EXPRESSION = 1;

int evaluate(const string& infix, string& postfix, bool& result)

// Evaluates a boolean expression

// Precondition: infix is an infix boolean expression consisting of the

// symbols T and F, the operators |, &, and !, and parentheses, with

// embedded blanks allowed for readability.

// Postcondition: If infix is a valid infix boolean expression, postfix is

// set to the postfix form of that expression, result is set to the value

// of the expression and the function returns zero. If infix is a

// malformed expression, the return value is 1. (In that case, postfix

// may or may not be changed and but result must be unchanged.)

{

// First convert infix to postfix

postfix = "";

stack<char> operatorStack;

char prevch = '|'; // pretend the previous character was an operator

for (size\_t k = 0; k < infix.size(); k++)

{

char ch = infix[k];

switch(ch)

{

case ' ':

continue; // do not set prevch to this char

case 'T':

case 'F':

if (isLetterOrCloseParen(prevch))

return RET\_INVALID\_EXPRESSION;

postfix += ch;

break;

case '(':

case '!':

if (isLetterOrCloseParen(prevch))

return RET\_INVALID\_EXPRESSION;

operatorStack.push(ch);

break;

case ')':

if ( ! isLetterOrCloseParen(prevch))

return RET\_INVALID\_EXPRESSION;

for (;;)

{

if (operatorStack.empty())

return RET\_INVALID\_EXPRESSION; // too many ')'

char c = operatorStack.top();

operatorStack.pop();

if (c == '(')

break;

postfix += c;

}

break;

case '|':

case '&':

if ( ! isLetterOrCloseParen(prevch))

return RET\_INVALID\_EXPRESSION;

while ( ! operatorStack.empty() &&

precedence(ch) <= precedence(operatorStack.top()) )

{

postfix += operatorStack.top();

operatorStack.pop();

}

operatorStack.push(ch);

break;

default: // bad char

return RET\_INVALID\_EXPRESSION;

}

prevch = ch;

}

// end of expression; pop remaining operators

if ( ! isLetterOrCloseParen(prevch))

return RET\_INVALID\_EXPRESSION;

while ( ! operatorStack.empty())

{

char c = operatorStack.top();

operatorStack.pop();

if (c == '(')

return RET\_INVALID\_EXPRESSION; // too many '('

postfix += c;

}

if (postfix.empty())

return RET\_INVALID\_EXPRESSION; // empty expression

// postfix now contains the converted expression

// Now evaluate the postfix expression

stack<bool> operandStack;

for (size\_t k = 0; k < postfix.size(); k++)

{

char ch = postfix[k];

if (ch == 'T')

operandStack.push(true);

else if (ch == 'F')

operandStack.push(false);

else

{

bool opd2 = operandStack.top();

operandStack.pop();

if (ch == '!')

operandStack.push(!opd2);

else

{ bool opd1 = operandStack.top();

operandStack.pop();

if (ch == '&')

operandStack.push(opd1 && opd2);

else if (ch == '|')

operandStack.push(opd1 || opd2);

else // Impossible!

return RET\_INVALID\_EXPRESSION; // pretend it's an invalid expression

}

}

}

if (operandStack.size() != 1) // Impossible!

return RET\_INVALID\_EXPRESSION; // pretend it's an invalid expression

result = operandStack.top();

return RET\_OK\_EVALUATION;

}

// Here's an interactive test driver:

// #include <iostream>

// #include <string>

// using namespace std;

//

// int main()

// {

// string s;

// while (getline(cin,s) && s != "exit")

// {

// string postfix;

// bool val;

// switch (evaluate(s, postfix, val))

// {

// case RET\_OK\_EVALUATION:

// cout << "Postfix is " << postfix << " and value is "

// << (val ? "true" : "false") << endl;

// break;

// case RET\_INVALID\_EXPRESSION:

// cout << "Malformed expression" << endl;

// break;

// default:

// cout << "Impossible return code" << endl;

// break;

// }

// }

// }