2. First 12 popped: (Stack)

(3, 5)

(3, 6)

(3, 4)

(2, 4)

(1, 4)

(1, 3)

(1, 2)

(1, 1)

(3, 3)

(4, 5)

(5, 5)

(5, 4)

4. (Queue)

(3, 5)

(4, 5)

(3, 4)

(3, 6)

(5, 5)

(3, 3)

(2, 4)

(6, 5)

(5, 4)

(1, 4)

(7, 5)

(5, 3)

In the stack, we always explored the last coordinate pushed (Last in, first out). On the other hand, for the queue, the oldest coordinate that was pushed was explored first (first in, first out). Also for the queue, we go out in rings. It will always give its optimal answer, but for stack, there is a bit of luck involved depending on the starting. In other words, if the end point was right next to the start point and was the last one searched for, then stack solves maze quickly.

END OF RESPONSE TO HOMEWORK.

#include "Set.h"

#include <iostream>

#include <string>

#include <stack>

#include <cctype>

#include <cassert>

using namespace std;

//SYNTAX IMPLEMENTATION SUPPORT

bool parenthesisCheck(string infix);

bool operatorcheck(string infix);

bool nearbychars(string infix, int index);

//END OF SYNTAX FUNCTIONS

int checkPrecedence(char ch);

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

{

if (infix.size() == 0) //infix must have stuff inside

return 1;

//

//return if char is not a lower case letter, space, |, &, or parenthesis

for (int i = 0; i < infix.size(); i++)

if (!isalpha(infix[i]) && !islower(infix[i]) && infix[i] != '&' && infix[i] != '|'

&& infix[i] != '!' && infix[i] != '(' && infix[i] != ')' && !isspace(infix[i]))

return 1;

int counterletter = 0; //every infix must have at least one letter

for (int i = 0; i < infix.size(); i++)

if (isalpha(infix[i]))

counterletter++;

if (counterletter == 0)

return 1;

//no consecutive letters allowed

int consecutiveletter = 0;

for (int i = 0; i < infix.size(); i++)

{

if (consecutiveletter > 1)

return 1;

if (isalpha(infix[i]))

consecutiveletter++;

if (infix[i] == '&' || infix[i] == '|')

consecutiveletter = 0;

}

if (consecutiveletter > 1)

return 1;

cerr << "no consecutive letters found" << endl;

if (!operatorcheck(infix))

{

cerr << "operator error" << endl;

return 1;

}

//at this point, operators are syntactically correct

//valid parenthesis order

if (!parenthesisCheck(infix))

{

cerr << "PARENTHESIS ERROR" << endl;

return 1;

}

/////////////////////////////////

//CONVERSION INTO POSTFIX

/////////////////////////////////

postfix = "";

stack<char> operatorStack;

for (int i = 0; i < infix.size(); i++)

{

switch (infix[i])

{

case 'a':

case 'b':

case 'c':

case 'd':

case 'e':

case 'f':

case 'g':

case 'h':

case 'i':

case 'j':

case 'k':

case 'l':

case 'm':

case 'n':

case 'o':

case 'p':

case 'q':

case 'r':

case 's':

case 't':

case 'u':

case 'v':

case 'w':

case 'x':

case 'y':

case 'z':

cerr << "postfix from letter: " << postfix << endl;

postfix += infix[i];

break;

case '(':

operatorStack.push(infix[i]);

break;

case ')':

// pop stack until matching '('

while (operatorStack.top() != '(' )

{

cerr << "postfix from ')': " << endl;

postfix += operatorStack.top();

operatorStack.pop();

}

operatorStack.pop(); //remove the '('

break;

case '&':

case '|':

case '!':

while (!operatorStack.empty() && operatorStack.top() != '('

&& checkPrecedence(infix[i]) <= checkPrecedence(operatorStack.top())) //Note: we are comparing precedence, not the operator characters themselves.

//Recall: ! > & > |.

{

postfix += operatorStack.top();

operatorStack.pop();

}

operatorStack.push(infix[i]);

break;

//space characters

default:

break;

}

}

cerr << "postfix before emptying loop: " << postfix << endl;

while (!operatorStack.empty())

{

postfix += operatorStack.top();

cerr << "postfix after emptying 1 stack: " << postfix << endl;

operatorStack.pop();

}

cerr << "postfix final: " << postfix << endl;

/////////////////////////////////

//END OF CONVERSION INTO POSTFIX

/////////////////////////////////

cerr << "entering true/false check" << endl;

/////////////////////////////////

//TRUEVALUE/FALSEVALUE ERROR

/////////////////////////////////

for (int i = 0; i < infix.size(); i++)

{

if ((!trueValues.contains(infix[i]) && !falseValues.contains(infix[i])) && isalpha(infix[i]))

{

cerr << "ANSWER ERROR: letter not found in both true and false values" << endl;

cerr << "infix[i]: " << infix[i] << endl;

return 2;

}

if (trueValues.contains(infix[i]) && falseValues.contains(infix[i]))

{

cerr << "ANSWER ERROR: letter found in both true and false" << endl;

return 3;

}

}

cerr << "begin evaluation" << endl;

/////////////////////////////////

//EVALUATING POSTFIX EXPRESSION

/////////////////////////////////

stack<bool> operandStack;

for (int i = 0; i < postfix.size(); i++)

if (isalpha(postfix[i]) && trueValues.contains(postfix[i]))

operandStack.push(true);

else if (isalpha(postfix[i]) && falseValues.contains(postfix[i]))

operandStack.push(false);

else if (postfix[i] == '!')

operandStack.top() = !operandStack.top();

else //ch is a binary operator

{

bool operand2 = operandStack.top();

operandStack.pop();

bool operand1 = operandStack.top();

operandStack.pop();

if (postfix[i] == '|' && !operand1 && !operand2)

operandStack.push(false);

else if (postfix[i] == '|' && (operand1 || operand2))

operandStack.push(true);

else if (postfix[i] == '&' && operand1 && operand2)

operandStack.push(true);

else

operandStack.push(false);

}

result = operandStack.top();

if (result)

cerr << "result is TRUE" << endl;

else cerr << "result is FALSE" << endl;

cerr << "no errors" << endl;

return 0;

}

int checkPrecedence(char ch)

{

if (ch == '!')

return 2;

else if (ch == '&')

return 1;

else if (ch == '|')

return 0;

return -1;

}

bool parenthesisCheck(string infix)

{

int front = 0, back = 0;

for (int i = 0; i < infix.size(); i++)

{

if (front == 0 && back == 0 && infix[i] == ')')

return false;

if (infix[i] == '(')

front++;

if (infix[i] == ')')

back++;

}

if (front != back)

return false;

if (front == 0 && back == 0) //no point in checking if no parenthesis to begin with

return true;

//int temp\_f = front; int temp\_b = back;

//At this point, we've dealt with most cases, except for certain special ones

front = 0, back = 0;

for (int i = infix.size() - 1; i >= 0; i--)

{

if (front == back && infix[i] == '(')

return false;

if (infix[i] == ')')

back++;

if (infix[i] == '(')

front++;

}

cerr << "parenthesis test 1 fine" << endl;

bool frontcheck = true; //bool backcheck = true;

for (int i = 0; i < infix.size(); i++)

{

if (infix[i] == '(')

{

if (!frontcheck)

return false;

front++;

}

else if (infix[i] == ')')

{

back++;

if (back > front)

return false; //not possible to have more ) than (

if (front > back) //false means not enough ) to support (

frontcheck = false;

else

frontcheck = true;

}

}

//invalid if no letters in between parenthesis

int letterCount = 0;

for (int i = 0; i < infix.size(); i++)

if (infix[i] == '(')

for (int j = i; infix[j] != ')'; j++)

if (isalpha(infix[j]))

letterCount++;

if (letterCount == 0)

return false;

cerr << "parentheses are fine" << endl;

return true;

}

bool operatorcheck(string infix) //syntax for operator

{

bool letter\_detect = false;

for (int i = 0; i < infix.size(); i++) //beginning cannot start with operator

{

if ((infix[i] == '|' || infix[i] == '&') && !letter\_detect)

return false;

if (isalpha(infix[i]))

letter\_detect = true;

}

cerr << "operator test 1 fine" << endl;

letter\_detect = false;

for (int i = infix.size() - 1; i >= 0; i--) //end cannot end with operator

{

if ((infix[i] == '|' || infix[i] == '&') && !letter\_detect)

return false;

if (isalpha(infix[i]))

letter\_detect = true;

}

cerr << "operator test 2 fine" << endl;

//at this point, we know infix has no starting or ending operator

//no consecutive & or | allowed

int operator\_count = 0;

for (int i = 0; i < infix.size(); i++)

{

if (operator\_count > 1)

return false;

if (infix[i] == '&' || infix[i] == '|')

operator\_count++;

else if (isalpha(infix[i])) //reset operator counter

operator\_count = 0;

}

//operator cannot focus on one letter/expression only

for (int i = 0; i < infix.size(); i++)

if ((infix[i] == '&' || infix[i] == '|') && !nearbychars(infix, i))

return false;

cerr << "operators are fine" << endl;

return true;

}

bool negatecheck(string infix)

{

int negatecounter = 0;

for (int i = 0; i < infix.size(); i++)

if (infix[i] == '!')

negatecounter++;

if (negatecounter == 0) //don't bother checking if no '!' found

return true;

bool letter\_detect = false;

//not possible for infix to end with ! or come after letter

for (int i = infix.size() - 1; i >= 0; i--)

{

if (infix[i] == '!' && !letter\_detect)

return false;

else if (isalpha(infix[i]))

letter\_detect = true;

if (infix[i] == '!')

letter\_detect = false;

}

//not possible for infix to have operator or ')' immediately follow !

for (int i = 0; i < infix.size(); i++)

if (infix[i] == '!')

{

for (int j = i; j < infix.size(); j++)

{

if (isalpha(infix[j]) )

break;

else if (infix[j] == '&' || infix[j] == '|' || infix[j] == ')')

return false;

}

}

return true;

}

bool nearbychars(string infix, int index) //for operator

{

bool frontchar = false;

bool backchar = false;

cerr << "checking nearby characters for operator" << endl;

for (int i = 0; i < infix.size(); i++)

if (infix[i] == '&' || infix[i] == '|')

{

//check if there's letter before operator

for (int j = i; j >= 0; j--)

{

if (isalpha(infix[j]))

{

frontchar = true;

break;

}

else if (infix[j] == '!' || infix[j] == '(')

return false;

}

if (!frontchar)

return false;

//check if there's letter after operator

for (int k = i; k < infix.size(); k++)

{

if (isalpha(infix[k]))

{

backchar = true;

break;

}

else if (infix[k] == ')')

return false;

}

if (!backchar)

return false;

}

cerr << "nearby function fine" << endl;

return true;

}

int main()

{

string trueChars = "tywz";

string falseChars = "fnx";

Set trues;

Set falses;

for (int k = 0; k < trueChars.size(); k++)

trues.insert(trueChars[k]);

for (int k = 0; k < falseChars.size(); k++)

falses.insert(falseChars[k]);

string pf;

bool answer;

assert(evaluate("w| f", trues, falses, pf, answer) == 0 && pf == "wf|" && answer);

assert(evaluate("(t))((b)", trues, falses, pf, answer) == 1);

assert(evaluate("y|", trues, falses, pf, answer) == 1);

assert(evaluate("n t", trues, falses, pf, answer) == 1);

assert(evaluate("nt", trues, falses, pf, answer) == 1);

assert(evaluate("()", trues, falses, pf, answer) == 1);

assert(evaluate("y(n|y)", trues, falses, pf, answer) == 1);

assert(evaluate("t(&n)", trues, falses, pf, answer) == 1);

assert(evaluate("(n)", trues, falses, pf, answer) == 0);

assert(evaluate("(n&(t|7)", trues, falses, pf, answer) == 1);

assert(evaluate("", trues, falses, pf, answer) == 1);

assert(evaluate("f | !f & (t&n) ", trues, falses, pf, answer) == 0

&& pf == "ff!tn&&|" && !answer);

assert(evaluate(" x ", trues, falses, pf, answer) == 0 && pf == "x" && !answer);

trues.insert('x');

assert(evaluate("((x))", trues, falses, pf, answer) == 3);

falses.erase('x');

assert(evaluate("((x))", trues, falses, pf, answer) == 0 && pf == "x" && answer);

trues.erase('w');

assert(evaluate("w| f", trues, falses, pf, answer) == 2);

falses.insert('w');

assert(evaluate("w| f", trues, falses, pf, answer) == 0 && pf == "wf|" && !answer);

cerr << "Passed all tests" << endl;

}