**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра информационных систем**

отчет

**по практической работе №3**

**по дисциплине «Программирование»**

Тема: Польские нотации. Стек и очередь

|  |  |  |
| --- | --- | --- |
| Студент гр. |  | Полуянов В. Н. |
| Преподаватель |  | Глущенко А. Г. |

Санкт-Петербург

2023

**Цель работы.**

Получение практических навыков работы со стеками и очередями; Изучение обратной и прямой польской нотации; Проведение сравнительного анализа этих структур данных.

**Основные теоретические положения.**

Стек является частным случаем однонаправленного списка, добавление элементов в который и выборка из которого выполняется с одного конца – вершины стека. Другие операции со стеком не определены. При выборке элемент исключается из стека. Стек реализует принцип обслуживания LIFO (Last In First Out).

Графически стек можно изобразить в виде вертикального списка (например, стопка книг), где для того, чтобы взять нижнюю книгу и не нарушить порядок, нужно поднять все книги, а положить книгу можно лишь поверх остальных книг.

Стек чаще всего реализуется на основе массивов или односвязных и двусвязных списков. Многие языки располагают встроенными средствами организации и обработки стеков. В C++ таким средством является stack.

Stack – контейнер, в котором добавление и удаление элементов осуществляется с одного конца. Использование стековых операций не требует описания, т.е. stack предоставляет набор стандартных функций.

Очередь – ещё один частный случай однонаправленного списка, добавление элементов в который выполняется в один конец, а выборка – из другого конца. В программировании очереди применяются при моделировании, диспетчеризации задач ОС, буферизованном вводе/выводе. Очередь реализует принцип обслуживания FIFO (First In First Out).

Графически очередь удобно изобразить в виде вертикального списка, например, очередь в магазине, где для того чтобы дойти до кассы и не нарушить установленный порядок, нужно дождаться, пока все покупатели перед вами не приобретут товар.

Если для стека в момент добавления или удаления допустимо задействовать лишь его вершины, то при реализации очереди добавлять нужно в конец, а удалять – из начала.

Обратная польская нотация – это форма записи математических и логических выражений, в которой операнды расположены перед знаками операций. Обратная польская нотация имеет ряд преимуществ перед привычной человеку инфиксной записью при выражении алгебраических формул, одно из них то, что инфиксные операторы имеют приоритеты, которые произвольны и нежелательны.

**Постановка задачи.**

Необходимо написать программу, которая:

1. Реализует преобразование введенного пользователем выражение;
2. Реализует проверку на корректность простого выражения и выражения, записанного в прямой и обратной польских нотациях;
3. Реализует вычисления простого выражения и выражения, записанного в прямой и обратной польской нотациях.

Программа должна выводить и описывать все промежуточные действия.

**Выполнение работы.**

Код программы представлен в приложении А.

1. При запуске программы пользователю выводится основное меню, предлагающее выбрать вид записи выражения или выключить режим пошагового отображения действий. (рис. 1).

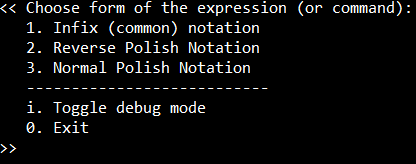


Рисунок . Главное меню

1. Следующий шаг зависит от введенной команды, если пользователь ввёл:
   1. “0”, то выполнение программы завершается.
   2. “1”, то ожидается ввод пользователем выражения в инфиксной форме, после чего оно переводится в прямую и обратную польскую нотацию и подсчитывается значение (рис. 2).

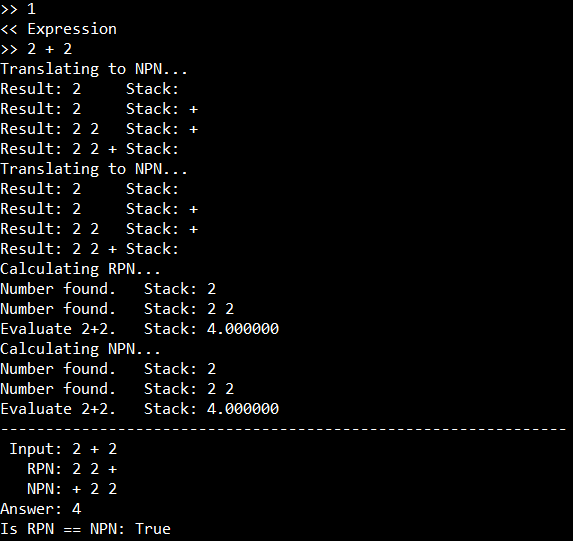


Рисунок 2. Задача 1

* 1. “2”, то ожидается ввод пользователем выражения в постфиксной форме, после чего подсчитывается значение (рис. 3).

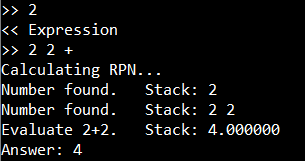


Рисунок 3. Задача 2

* 1. “3”, то ожидается ввод пользователем выражения в префиксной форме, после чего подсчитывается значение (рис. 4).

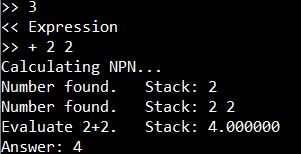


Рисунок 4. Задача 3

* 1. “i”, то переключается режим пошагового отображения (вкл./выкл.) (рис. 5).



Рисунок 5. Переключение режима вывода

**Выводы.**

В ходе работы были получены практические навыки работы со стеками и очередями, изучены обратная и прямая польские нотации.

Приложение А

рабочий код

#include <iostream>  
#include <iomanip>  
#include <unordered\_map>  
#include <vector>  
#include <stdexcept>  
#include "stack.h"  
bool inputAction(char &userAction) {  
 std::cout << "<< Choose form of the expression (or command):\n"  
 " 1. Infix (common) notation\n"  
 " 2. Reverse Polish Notation\n"  
 " 3. Normal Polish Notation\n "  
 << std::setw(28) << std::setfill('-') << '\n'  
 << std::setfill(' ') <<  
 " i. Toggle debug mode\n"  
 " 0. Exit\n"  
 ">> ";  
 std::cin >> userAction;  
 *// Error handler (i.e. more than one symbol input)* if (std::cin.fail() || std::cin.get() != '\n') {  
 std::cout << "RuntimeError: unknown command\n";  
 std::cin.clear();  
 std::cin.ignore(100000, '\n');  
 return false;  
 }  
 return true;  
}  
std::vector<std::string> splitString(std::string str, bool wsRemove = true) {  
 if (str.empty()) throw std::invalid\_argument("Empty string");  
 std::vector<std::string> tokens;  
 char operators[6] = {'(', ')', '+', '-', '\*', '/'};  
 if (wsRemove) str.erase(remove(str.begin(), str.end(), ' '), str.end());  
 std::string number;  
 for (char c : str) {  
 std::string s(1, c);  
 auto find = std::find(std::begin(operators), std::end(operators), c);  
 if (c == ' ') {  
 if (!number.empty()) {  
 tokens.push\_back(number);  
 number.clear();  
 }  
 } else if ( find != std::end(operators)) { *// Operator* if (!number.empty()) {  
 tokens.push\_back(number);  
 number.clear();  
 }  
 tokens.push\_back(s);  
 } else if (isdigit(c)) { *// Number* number += c;  
 } else { *// Unknown symbol error* throw std::runtime\_error("Unknown symbol: " + s);  
 }  
 }  
 if (!number.empty()) tokens.push\_back(number);  
 return tokens;  
}  
void printDebug(std::vector<std::string> result, Stack opStack, std::vector<std::string> tokens) {  
 int size = 7;  
 for (auto op : tokens) if (op != "(" && op != ")") size += op.length() + 1;  
 std::cout << "Result: ";  
 for (std::string token : result) {  
 std::cout << token << ' ';  
 size -= token.length() + 1;  
 }  
 std::cout << std::setw(size) << "Stack: ";  
 opStack.print();  
}  
double calculate(double a, double b, std::string op) {  
 switch (op[0]) {  
 case '+': return a + b;  
 case '-': return a - b;  
 case '\*': return a \* b;  
 case '/': {  
 if (b == 0)  
 throw std::runtime\_error("Division by zero: " + std::to\_string(a) + " / " + std::to\_string(b));  
 return a / b;  
 }  
 default: throw std::invalid\_argument("Wrong operator: " + op);  
 }  
}  
std::vector<std::string> toRPN(const std::string& expr, bool isDebugMode = false) {  
 std::vector<std::string> result;  
 Stack opStack;  
 std::vector<std::string> tokens = splitString(expr);  
 std::unordered\_map<std::string, int> precedences **{**{"+", 1}, {"-", 1}, {"\*", 2}, {"/", 2}**}**;  
 if (isDebugMode) std::cout << "Translating to NPN...\n";  
 for (std::string token : tokens) {  
 if (isdigit(token[0])) result.push\_back(token); *// Number: Add to result immediately* else if (token == "(") opStack.push(token); *// Open brace: Add to opStack* else if (token == ")") { *// Close brace: Read stack while not (* while (opStack.top() != "(") result.push\_back(opStack.pop());  
 if (opStack.isEmpty()) throw std::runtime\_error("Unclosed brace");  
 opStack.pop(); *// Delete open brace from stack* } else if (precedences.count(token) > 0) { *// Operator* while (!opStack.isEmpty() && opStack.top() != "(" && precedences[token] <= precedences[opStack.top()])  
 result.push\_back(opStack.pop());  
 opStack.push(token);  
 } else throw std::runtime\_error("Invalid syntax: " + token);  
 if (isDebugMode) printDebug(result, opStack, tokens);  
 }  
 while (!opStack.isEmpty()) {  
 if (opStack.top() == "(") throw std::runtime\_error("Unclosed brace");  
 result.push\_back(opStack.pop());  
 if (isDebugMode) printDebug(result, opStack, tokens);  
 }  
 return result;  
}  
std::vector<std::string> toNPN(const std::string& expr, bool isDebugMode = false) {  
 std::vector<std::string> result;  
 Stack opStack;  
 std::unordered\_map<std::string, int> precedences **{**{"+", 1}, {"-", 1}, {"\*", 2}, {"/", 2}**}**;  
 std::vector<std::string> tokens = splitString(expr);  
 std::reverse(tokens.begin(), tokens.end());  
 if (isDebugMode) std::cout << "Translating to NPN...\n";  
 for (std::string token : tokens) {  
 if (std::isdigit(token[0])) { *// Number* result.push\_back(token);  
 } else if (precedences.count(token) > 0) { *// Operator* while (!opStack.isEmpty() && opStack.top() != ")" && precedences[opStack.top()] > precedences[token])  
 result.push\_back(opStack.pop());  
 opStack.push(token);  
 } else if (token == ")") { *// Open brace* opStack.push(token);  
 } else if (token == "(") { *// Close brace* while (!opStack.isEmpty() && opStack.top() != ")")  
 result.push\_back(opStack.pop());  
 if (opStack.isEmpty()) throw std::runtime\_error("Unclosed brace");  
 opStack.pop();  
 } else throw std::runtime\_error("Invalid syntax: " + token); *// Error* if (isDebugMode) printDebug(result, opStack, tokens);  
 }  
 while (!opStack.isEmpty()) {  
 if (opStack.top() == ")") throw std::runtime\_error("Unclosed brace");  
 result.push\_back(opStack.pop());  
 if (isDebugMode) printDebug(result, opStack, tokens);  
 }  
 std::reverse(result.begin(), result.end());  
 return result;  
}  
double evaluate(std::vector<std::string> expr, bool isNPN = false, bool isDebugMode = false) {  
 if (isDebugMode) std::cout << "Calculating " << (isNPN ? "NPN" : "RPN") << "...\n";  
 Stack opStack;  
 if (isNPN) std::reverse(expr.begin(), expr.end());  
 for (std::string token : expr) {  
 if (isdigit(token[0])) {  
 opStack.push(token);  
 if (isDebugMode) {  
 std::cout << "Number found. \tStack: ";  
 opStack.print();  
 }  
 } else {  
 if (opStack.getSize() < 2) throw std::runtime\_error("Nothing to calculate");  
 double op1 = std::stod(opStack.pop()), op2 = std::stod(opStack.pop());  
 if (!isNPN) std::swap(op1, op2);  
 double calc = calculate(op1, op2, token);  
 opStack.push(std::to\_string(calc));  
 if (isDebugMode) {  
 std::cout << "Evaluate " << op1 << token << op2 << ". \tStack: ";  
 opStack.print();  
 }  
 }  
 }  
 return std::stod(opStack.pop());  
}  
int main() {  
 bool isDebugMode = true;  
 while (true) {  
 char userAction; *// Get command from user* if (!inputAction(userAction)) continue; *// Error occurred* if (userAction == '0') break; *// Exit command* bool isNPN = true;  
 switch (userAction) {  
 *// Infix to Polish Notation* case '1': {  
 try {  
 std::cout << "<< Expression\n>> ";  
 std::string expr;  
 std::getline(std::cin, expr);  
 std::vector<std::string> rpn = toRPN(expr, isDebugMode); *// Get Reverse PN* std::vector<std::string> npn = toNPN(expr, isDebugMode); *// Get Normal PN* double rpnAnswer = evaluate(rpn, false, isDebugMode); *// Evaluate RPN* double npnAnswer = evaluate(npn, true, isDebugMode); *// Evaluate NPN* std::cout << std::setw(64) << std::setfill('-') << ' ' << std::setfill(' ');  
 std::cout << "\n Input: " << expr << "\n RPN: ";  
 for (auto token : rpn) std::cout << token << ' ';  
 std::cout << "\n NPN: ";  
 for (auto token : npn) std::cout << token << ' ';  
 std::cout << "\nAnswer: " << rpnAnswer;  
 std::cout << "\nIs RPN == NPN: " << (npnAnswer == rpnAnswer ? "True" : "False") << '\n';  
 }  
 catch (std::invalid\_argument& e) { std::cerr << "Invalid argument. " << e.what(); }  
 catch (std::runtime\_error& e) { std::cerr << "Runtime error. " << e.what(); }  
 break;  
 }  
 *// Case algorithms are the same. The only difference is the only one value isRPN* case '2': isNPN = false; *// Calculate Reverse Polish Notation* case '3': { *// Calculate Normal Polish Notation* std::cout << "<< Expression\n>> ";  
 std::string expr;  
 std::getline(std::cin, expr);  
  
 try {  
 std::vector<std::string> npn = splitString(expr, false);  
 double answer = evaluate(npn, isNPN, isDebugMode);  
 std::cout << "Answer: " << answer << std::endl;  
 }  
 catch (std::invalid\_argument& e) { std::cerr << "Invalid argument. " << e.what(); }  
 catch (std::runtime\_error& e) { std::cerr << "Runtime error. " << e.what(); }  
 break;  
 }  
 *// Toggle debug mode* case 'i':  
 isDebugMode = !isDebugMode;  
 std::cout << "Debug mode switched to " << isDebugMode << '\n';  
 break;  
  
 *// Unknown command error* default: std::cout << "RuntimeError: unknown command\n";  
 }  
 system("pause");  
 }  
 return 0;  
}