**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. 2372 |  | Соколовский В.Д. |
| 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.

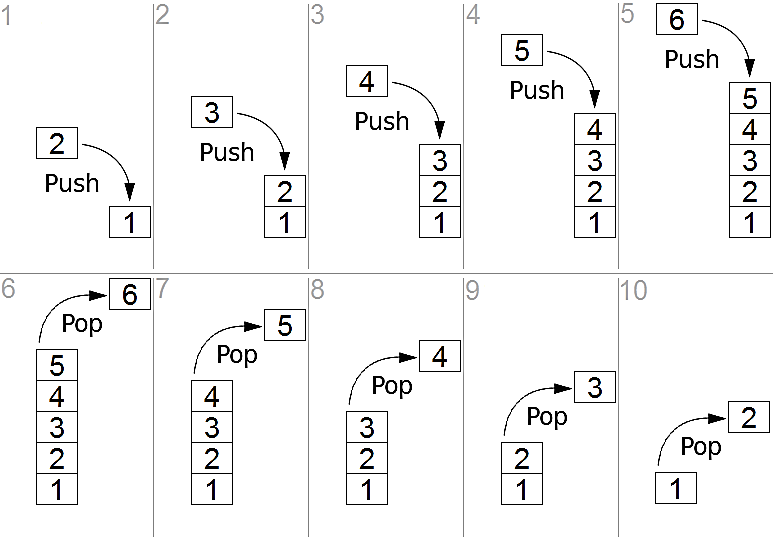
**Основные теоретические положения.**

Стек – это частный случай однонаправленного списка, добавление элементов в который и выборка из которого выполняется с одного конца, называемого вершиной стека. Другие операции со стеком не определены. При выборке элемент исключается из стека. Говорят, что стек реализует принцип обслуживания LIFO (последним пришел – первым ушел).

Основные операции над стеками:

1. чтение верхнего элемента;
2. добавление нового элемента;
3. удаление существующего элемента.

Графически его удобно изобразить в виде вертикального списка (рис. 1), например, стопки книг, где для того чтобы воспользоваться одной из них и не нарушить установленный порядок, нужно поднять все книги, которые лежат выше нее, а положить книгу можно лишь поверх всех остальных.

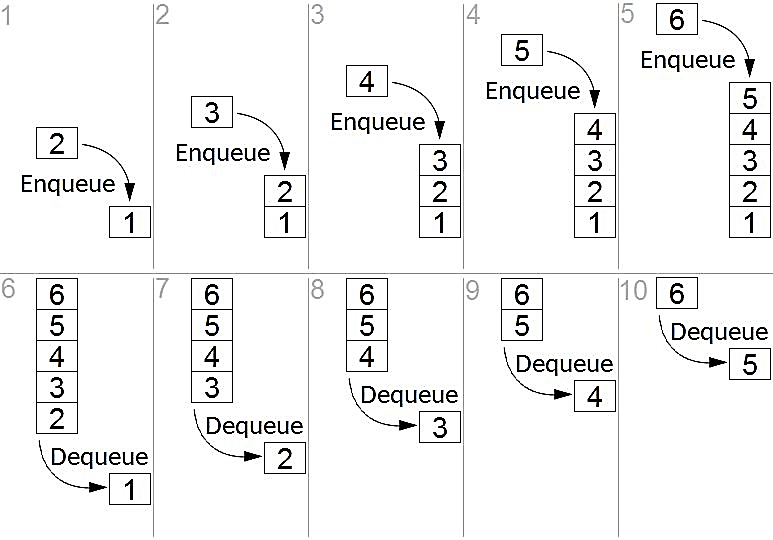


*Рис. 1.* Графическое представление стека

Стек чаще всего реализуется на основе обычных массивов, односвязных и двусвязных списков. В зависимости от конкретных условий выбирается одна из этих структур данных.

Очередь – частный случай однонаправленного списка, добавление элементов в который выполняется в один конец, а выборка – из другого конца. Другие операции с очередью не определены. При выборке элемент исключается из очереди. Говорят, что очередь реализует принцип обслуживания FIFO (первым пришел – первым ушел). В программировании очереди применяются при моделировании, диспетчеризации задач операционной системой, буферизованном вводе/выводе.

Графически ее удобно изобразить в виде вертикального списка (рис. 2), 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. Разумеется, будут появляться новые покупатели, которые будут занимать свое место в очереди в ожидании покупки.



*Рис. 2.* Графическое представление очереди

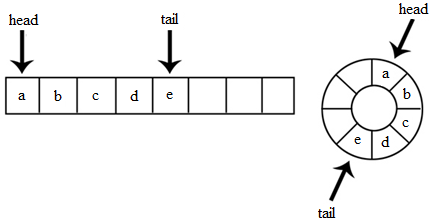
Основные операции над очередями:

1. чтение первого элемента;
2. добавление нового элемента;
3. удаление существующего элемента.

Если для стека в момент добавления или удаления элемента допустимо задействование лишь его вершины, то касательно очереди эти две операции должны быть применены так, как это регламентировано в определении этой структуры данных, т. е. добавление – в конец, удаление – из начала.

Выделяют два способа программной реализации очереди. Первый основан на базе массива, второй – на базе указателей (связного списка). Первый способ – статический, так как очередь представляется в виде простого статического массива, второй – динамический.

Кольцевой буфер так же известен, как кольцевая очередь или циклический буфер и является распространенной формой очереди. Это популярный, легко реализуемый стандарт, и хотя он представлен в виде круга (рис. 3) в базовом коде он является линейным. Кольцевая очередь существует как массив фиксированной длины с двумя указателями: один представляет начало очереди, другой – хвост.



*Рис. 3.* Кольцевая очередь

Операции с очередями работают следующим образом.

1. Два указателя, называемые и , используются для отслеживания первого и последнего элементов в очереди.
2. При инициализации очереди значения и устанавливаются равными –1.
3. При добавлении элемента постепенно увеличивается значение индекса и помещается новый элемент в положение, на которое указывает .
4. При снятии очереди с элемента возвращается значение, на которое указывает , и постепенно увеличивается индекс .
5. Перед постановкой в очередь проверяется, заполнена ли очередь.
6. Перед снятием очереди проверяется, пуста ли очередь.
7. При инициализации первого элемента устанавливается значение в 0.
8. При удалении последнего элемента сбрасываются значения и в –1.

Недостатком метода является его фиксированный размер. Для очередей, где элементы должны быть добавлены и удалены в середине, а не только в начале и конце буфера, реализация в виде связанного списка является предпочтительным подходом.

Обратная польская запись (нотация) (рис. 4) – форма записи математических и логических выражений, в которой операнды расположены перед знаками операций. Обратная польская запись имеет ряд преимуществ перед инфиксной записью при выражении алгебраических формул, одно из них то, что инфиксные операторы имеют приоритеты, которые произвольны и нежелательны.

|  |  |  |
| --- | --- | --- |
| Простое выражение | Прямая польская запись | Обратная польская запись |
| X + 3 \* Y  (X + 3) \* Y  1 + 2 | + X \* 3 Y  \* + X 3 Y  + 1 2 | X 3 Y \* +  X 3 + Y \*  1 2 + |

*Рис. 4.* Представление прямой и обратной польской записи

Обратная польская запись отлично подходит для вычисления выражений при помощи стека. Причем сам алгоритм достаточно прост. Необходимо просто прочитать обратную польскую запись слева направо. Если встречается операнд, то его нужно поместить в стек. Если встречается оператор, нужно выполнить заданную им операцию.

**Постановка задачи.**

Необходимо написать программу, которая выполняет следующее:

1. Реализует преобразование введенного выражения (если используются переменные, то пользователь должен их инициализировать). Ввод выражения должен быть реализовать двумя способами: с клавиатуры и с файла.

2. Реализует проверку на корректность простого выражения и выражения, записанного в прямой и обратной польских нотациях (на выбор пользователя). Ввод выражения должен быть реализовать двумя способами: с клавиатуры и с файла.

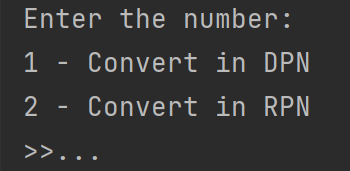
3. Реализует вычисления простого выражения и выражения, записанного в прямой и обратной польских нотациях (на выбор пользователя). Ввод выражения должен быть реализовать двумя способами: с клавиатуры и с файла.

4. Генерирует несколько (на выбор пользователя) вариантов проверочной работы по польской нотации (прямой и обратной). Задание и ответы к ним необходимо вывести в отдельные файлы (ответы должны быть максимально подробными).

Программа должна выводить и описывать все промежуточные действия.

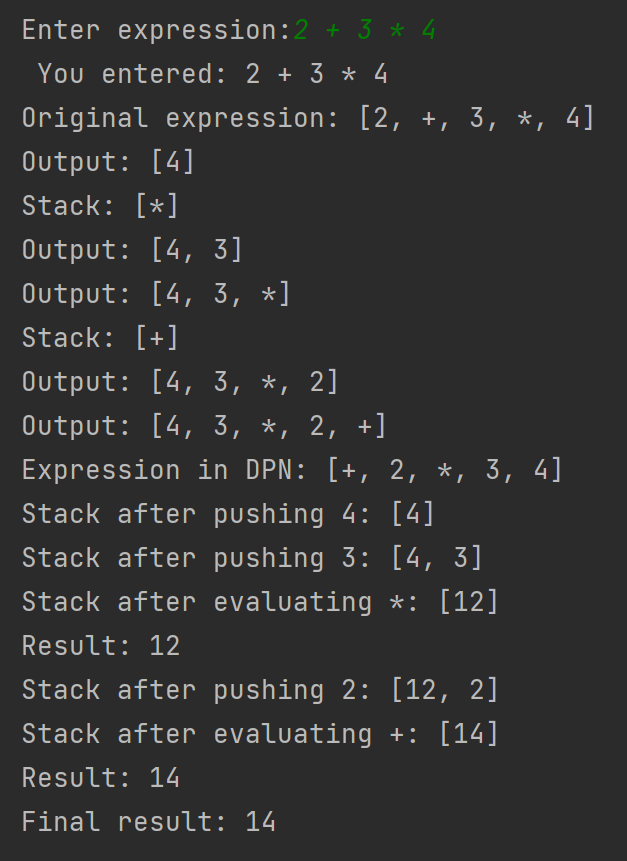
**Выполнение работы.**

1. **Пользователю предоставляется выбор между конвертацией просто выражения в обратную польскую нотацию или прямую польскую нотацию(рис.1):**

****

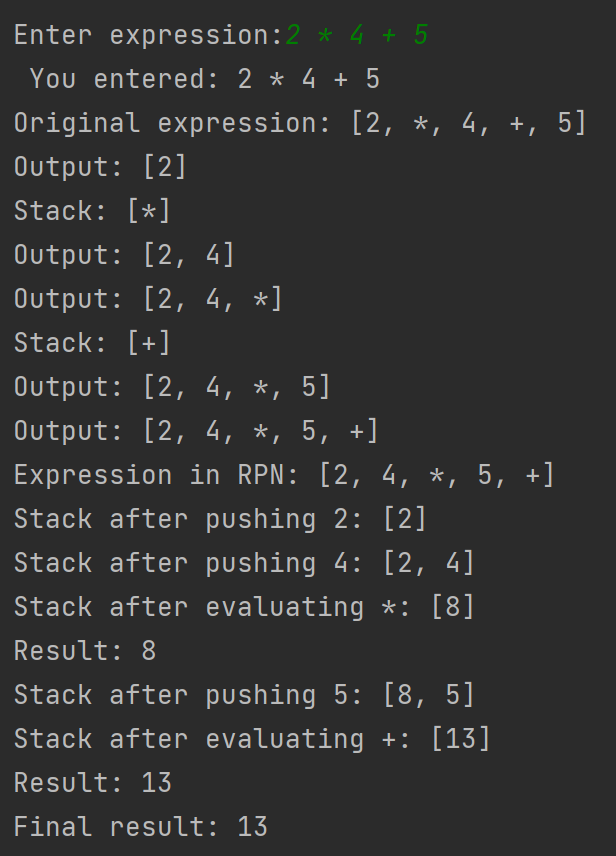
**Рис. 1**

1. **Реализует преобразование простого выражения в прямую польскую нотацию и выводит все промежуточные действия, а также выводит результат вычисления выражения. Ввод реализован с клавиатуры(рис.2):**

****

**Рис.2**

1. **Реализует преобразование простого выражения в обратную польскую нотацию и выводит все промежуточные действия, а также выводит результат вычисления выражения. Ввод реализован с клавиатуры(рис.3):**

****

**Рис.3**

**Выводы**.

В ходе данной практической работы я полученил практические навыки работы со стеками и очередями; изучил обратную и прямую польскую нотацию; провел сравнительный анализ этих структур данных.

Приложение А

**КОД ПРОГРАММЫ**

#include <iostream>

#include <string>

#include <vector>

#include <sstream>

#include <algorithm>

using namespace std;

void push(vector<string>& stack, string value){

stack.push\_back(value);

}

string pop(vector<string>& stack){

string value = stack.back();

stack.pop\_back();

return value;

}

void printStack(vector<string>& stack){

cout << "[";

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

cout << stack[i];

if(i < stack.size() - 1){

cout << ", ";

}

}

cout << "]" << endl;

}

void clearStack(vector<string>& stack){

stack.clear();

}

bool isOperator(string token){

return token == "+" || token == "-" || token == "\*" || token == "/";

}

int evaluate(string op, int a, int b){

if(op == "+"){

return a + b;

}

else if(op == "-"){

return a - b;

}

else if(op == "\*"){

return a \* b;

}

else if(op == "/"){

return a / b;

}

else{

throw runtime\_error("Unknown operator: " + op);

}

}

void printResult(int result){

cout << "Result: " << result << endl;

}

void convertToDPN(vector<string>& tokens){

vector<string> stack;

vector<string> output;

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

string token = tokens[i];

if(isdigit(token[0])){

output.push\_back(token);

cout << "Output: ";

printStack(output);

}

else if(isOperator(token)){

while(!stack.empty() && isOperator(stack.back()) && ((token == "\*" || token == "/") || (token == "+" || token == "-"))) {

string op = stack.back();

stack.pop\_back();

output.push\_back(op);

cout << "Output: ";

printStack(output);

}

stack.push\_back(token);

cout << "Stack: ";

printStack(stack);

}

else if(token == ")"){

stack.push\_back(token);

cout << "Stack: ";

printStack(stack);

}

else if(token == "("){

while(!stack.empty() && stack.back() != ")") {

string op = stack.back();

stack.pop\_back();

output.push\_back(op);

cout << "Output: ";

printStack(output);

}

if(stack.empty()){

throw runtime\_error("Error");

}

stack.pop\_back();

cout << "Stack: ";

printStack(stack);

}

else{

throw runtime\_error("Unknown token: " + token);

}

}

while(!stack.empty()){

string op = stack.back();

stack.pop\_back();

if(op == ")"){

throw runtime\_error("Error");

}

output.push\_back(op);

cout << "Output: ";

printStack(output);

}

reverse(output.begin(), output.end());

tokens = output;

}

int evaluateExpressionToDPN(vector<string>& tokens){

vector<string> stack;

int result = 0;

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

string token = tokens[i];

if(isOperator(token)){

string op = token;

int a = stoi(pop(stack));

int b = stoi(pop(stack));

result = evaluate(op, a, b);

push(stack, to\_string(result));

cout << "Stack after evaluating " << op << ": ";

printStack(stack);

printResult(result);

}

else{

push(stack, token);

cout << "Stack after pushing " << token << ": ";

printStack(stack);

}

}

return result;

}

string convertToRPN(vector<string>& tokens) {

vector<string> stack;

vector<string> output;

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

string token = tokens[i];

if(isdigit(token[0])){

output.push\_back(token);

cout << "Output: ";

printStack(output);

}

else if(isOperator(token)){

while(!stack.empty() && isOperator(stack.back()) && ((token == "\*" || token == "/") || (token == "+" || token == "-"))) {

string op = stack.back();

stack.pop\_back();

output.push\_back(op);

cout << "Output: ";

printStack(output);

}

stack.push\_back(token);

cout << "Stack: ";

printStack(stack);

}

else if(token == "("){

stack.push\_back(token);

cout << "Stack: ";

printStack(stack);

}

else if(token == ")"){

while(!stack.empty() && stack.back() != "(") {

string op = stack.back();

stack.pop\_back();

output.push\_back(op);

cout << "Output: ";

printStack(output);

}

if(stack.empty()){

throw runtime\_error("Error");

}

stack.pop\_back();

cout << "Stack: ";

printStack(stack);

}

else{

throw runtime\_error("Unknown token: " + token);

}

}

while(!stack.empty()){

string op = stack.back();

stack.pop\_back();

if(op == "("){

throw runtime\_error("Error");

}

output.push\_back(op);

cout << "Output: ";

printStack(output);

}

tokens = output;

return output.back();

}

int evaluateExpressionToRPN(vector<string>& tokens){

vector<string> stack;

int result = 0;

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

string token = tokens[i];

if (isOperator(token)) {

string op = token;

int b = stoi(pop(stack));

int a = stoi(pop(stack));

result = evaluate(op, a, b);

push(stack, to\_string(result));

cout << "Stack after evaluating " << op << ": ";

printStack(stack);

printResult(result);

} else {

push(stack, token);

cout << "Stack after pushing " << token << ": ";

printStack(stack);

}

}

return result;

}

int main() {

string s;

vector<string> tokens;

string token;

string input;

char choice;

do {

cout << "Enter the number: \n";

cout << "1 - Convert in DPN\n";

cout << "2 - Convert in RPN\n";

cout << ">>...\n";

cin >> choice;

cin.ignore();

cout << "\n\n";

switch(choice) {

case '1': {

cout << "Enter expression: ";

getline(cin, input);

cout << "You entered: " << input << endl;

stringstream ss(input);

while (ss >> token) {

tokens.push\_back(token);

}

cout << "Original expression: ";

printStack(tokens);

convertToDPN(tokens);

cout << "Expression in DPN: ";

printStack(tokens);

int result = evaluateExpressionToDPN(tokens);

cout << "Final result: " << result << endl;

clearStack(tokens);

cout << "\n\n";

break;

}

case '2': {

cout << "Enter expression: ";

getline(cin, input);

cout << "You entered: " << input << endl;

stringstream iss(input);

while (iss >> token) {

tokens.push\_back(token);

}

cout << "Original expression: ";

printStack(tokens);

convertToRPN(tokens);

cout << "Expression in RPN: ";

printStack(tokens);

int result = evaluateExpressionToRPN(tokens);

cout << "Final result: " << result << endl;

clearStack(tokens);

cout << "\n\n";

break;

}

default:{

cout << "Error" << endl;

return 0;

}

}

}

while(choice != 0);

return 0;

}