**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. 3372 |  | Кипень В. А. |
| Преподаватель |  | Глущенко А. Г. |

Санкт-Петербург

2024

**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.

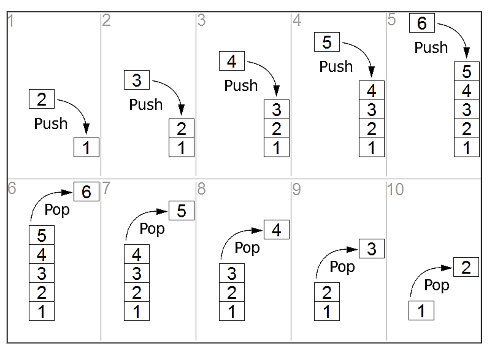


Рис. 1. Графическое представление стека

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.

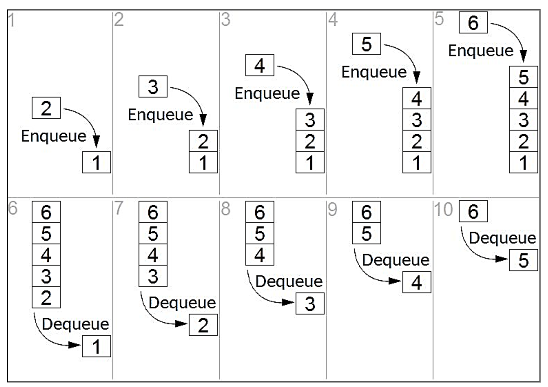


Рис. 2. Графическое представление очереди

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.

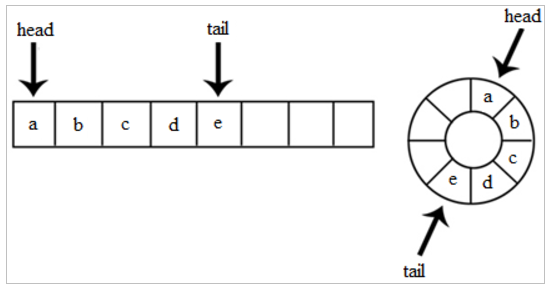


Рис. 3. Кольцевая очередь

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.

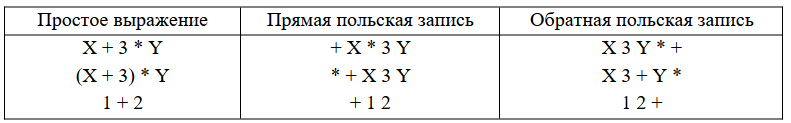


Рис. 4. Представление прямой и обратной польской записи

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.**

|  |  |
| --- | --- |
| Ввод пользователем и обработка данных | Работа алгоритма и вывод на экран |
| Меню | |
| При запуске программы перед пользователем появляется окно с меню, где он может выбрать пункт, который требуется выполнить | Меню:    Проверка на ввод символов, которые не входят в диапазон выбора: |
| Конвертация (с этапами) инфиксного выражения | |
| Пункт 1, необходимо ввести инфиксное арифметическое выражение (включая переменные), для его дальнейшей конвертации. | Этап инициализации переменных (если они есть)  Результат конвертации (постфиксная запись)    Результат конвертации (префиксная запись) |
| Проверка корректности арифметического выражения | |
| Пункт 2, проверка введённого арифметического выражения на корректность. |  |
| Вычисление результата арифметического выражения | |
| Пункт 3, вычисление результата арифметического выражения (введённого пользователем или взятого из памяти программы). |  |

**Выводы.**

Реализовано:

1. Преобразование введенного выражения (если используются переменные, то пользователь должен их инициализировать).
2. Проверка на корректность простого выражения и выражения, записанного в прямой и обратной польских нотациях (на выбор пользователя).
3. Вычисление простого выражения и выражения, записанного в прямой и обратной польских нотациях (на выбор пользователя).
4. Вывод и описывание всех промежуточных действий.

Приложение А

рабочий код

#include <iostream>

#include <string>

#include <map>

#include <algorithm>

using namespace std;

map<char, int> weight {

{'+', 2 },

{'-', 2 },

{'\*', 3 },

{'/', 3 },

{'(', 0 },

{')', 1 }

};

map<char, int> weightReversed {

{'+', 5 },

{'-', 4 },

{'\*', 6 },

{'/', 7 },

{'(', 1 },

{')', 0 }

};

struct Stack {

union {

int digit;

char letter;

} data;

Stack\* tail;

};

unsigned length(Stack\* beg) {

unsigned length = 0;

while (beg) {

++length;

beg = beg->tail;

}

return length;

}

bool isEmpty(Stack\* beg) {

if (beg == NULL || length(beg) == 0) return true;

return false;

}

Stack\* findItemByIndex(Stack\* beg, unsigned idx) {

int counter = 0;

while (beg) {

if (counter == idx) return beg;

beg = beg->tail;

counter++;

}

return NULL;

}

Stack\* peek(Stack\* beg) {

if (beg == NULL) return NULL;

while (beg->tail) beg = beg->tail;

return beg;

}

void pop(Stack\* &beg) {

Stack\* item = NULL;

int index = length(beg) - 1;

if (index < 0) return;

if ( !index ) {

item = beg -> tail;

delete beg;

beg = item;

return;

}

item = findItemByIndex(beg, index - 1);

Stack\* dItem = item->tail;

item->tail = dItem->tail;

delete dItem;

}

Stack\* addNode(Stack\* &beg) {

Stack\* item = new Stack;

item->tail = NULL;

if (beg == NULL) {

beg = item;

return item;

}

int index = length(beg);

Stack\* prevItem = beg;

--index;

while (prevItem->tail && (index--)) prevItem = prevItem->tail;

item->tail = prevItem->tail;

prevItem->tail = item;

return item;

}

Stack\* addDigit(Stack\* &beg, int data) {

Stack\* item = addNode(beg);

if (item) item->data.digit = data;

return item;

}

Stack\* addLetter(Stack\* &beg, int data) {

Stack\* item = addNode(beg);

if (item) item->data.letter = data;

return item;

}

void printNode(Stack\* curr, bool isDigit) {

cout << "\nItem\n"

<< " Data: " << (isDigit ? curr->data.digit : curr->data.letter) << "\n"

<< " Tail: " << curr->tail << "\n"

<< " Addr: " << curr << "\n";

}

void printStack(Stack\* beg, bool isDigit) {

if (!beg) {

cout << "\nThe stack is empty(((\n";

return;

}

while (beg) {

printNode(beg, isDigit);

beg = beg->tail;

}

}

int calculateInfixOperation(char op, int first, int second) {

switch (op) {

case '+':

return first + second;

case '-':

return first - second;

case '\*':

return first \* second;

case '/':

return first / second;

default:

return 0;

};

}

string getNumberFromString(string& expr, int& pos) {

string output = "";

for (pos; pos < expr.length(); pos++) {

char c = expr[pos];

if (isdigit(c)) output += c;

else {

pos--;

break;

}

}

return output;

}

void reverseString(string& inp) {

string output = "";

for (int i = inp.length() - 1; i >= 0; i--) output += inp[i];

inp = output;

}

string getNumberFromReverseString(string& expr, int& pos) {

string output = "";

for (pos; pos >= 0; pos--) {

char c = expr[pos];

if (isdigit(c)) output += c;

else {

pos++;

break;

}

}

reverseString(output);

return output;

}

void reverseNumbersInString(string& inp) {

string output = "";

string number = "";

for (int i = 0; i < inp.length(); i++) {

if (isdigit(inp[i])) {

number = getNumberFromString(inp, i);

reverseString(number);

output += number;

} else {

output += inp[i];

}

}

inp = output;

}

string fromInfixToPostfix(string& infixExpr) {

if (infixExpr.length() == 0) return "";

string output;

Stack\* opStack = NULL; *// стек операций*

for (int i = 0; i < infixExpr.length(); i++) {

char c = infixExpr[i];

if (isdigit(c)) {

string number = getNumberFromString(infixExpr, i);

output += number;

output += ' ';

cout << "\nFound number " << number << ", add it to the output line\n";

} else if (c == '+' || c == '-' || c == '\*' || c == '/') {

cout << "\nFound an operation sign " << c << ", checking priority\n";

if ( length(opStack) == 0 || (weight[peek(opStack)->data.letter] < weight[c]) ) {

addLetter(opStack, c);

cout << "\nCool cla$$, operation priority " << c << " > priority of top of the stack (or empty), add it to the operations stack\n";

}

else {

cout << "\nThat's a do$ada, the priority of the operation " << c << " <= priority of top of the stack, pull out all operations to the lowest priority\n";

while ( (peek(opStack) != NULL ? weight[peek(opStack)->data.letter] : -1) >= weight[c] ) {

output += peek(opStack)->data.letter;

output += ' ';

cout << "\nAdded an operation sign from the stack " << peek(opStack)->data.letter << " to the output line\n";

pop(opStack);

}

cout << "\nAnd consolidate success: priority " << c << " > priority of top of the stack (or empty), add it to the operations stack\n";

if ( length(opStack) == 0 || (weight[peek(opStack)->data.letter] < weight[c]) ) {

addLetter(opStack, c);

cout << "\nAdded an operation sign " << c << " to the operations stack\n";

}

}

} else if (c == '(') {

addLetter(opStack, c);

cout << "\nAdd open bracket to the operation stack\n";

} else if (c == ')') {

cout << "\nFound closed bracket, pull out all the operation signs from stack, until open bracket:\n";

while ( peek(opStack) != NULL && peek(opStack)->data.letter != '(' ) {

output += peek(opStack)->data.letter;

output += ' ';

cout << "\nAdded an operation sign from the stack " << peek(opStack)->data.letter << " to the output line\n";

pop(opStack);

}

cout << "\nRemove open bracket from the operation stack\n";

pop(opStack);

}

}

cout << "\nThe input line has ended, we throw all the remaining operation signs from the stack into the output line\n";

while (length(opStack) > 0) {

output += peek(opStack)->data.letter;

output += ' ';

cout << "\nAdded an operation sign from the stack " << peek(opStack)->data.letter << " to the output line\n";

pop(opStack);

}

return output;

}

string fromInfixToPrefix(string& infixExpr) {

if (infixExpr.length() == 0) return "";

string output = fromInfixToPostfix(infixExpr);

reverseString(output);

reverseNumbersInString(output);

return output;

}

string fromPrefixToPostfix(string& prefixExpr) {

if (prefixExpr.length() == 0) return "";

string output = prefixExpr + "";

reverseString(output);

reverseNumbersInString(output);

return output;

}

int calculatePostfix(string& postfixExpr) {

if (postfixExpr.length() == 0) return 0;

Stack\* stack = NULL; *// стек операндов*

string number;

for (int i = 0; i < postfixExpr.length(); i++) {

char c = postfixExpr[i];

if (isdigit(c)) {

number = getNumberFromString(postfixExpr, i);

addDigit(stack, stoi(number));

} else if (c == '+' || c == '-' || c == '\*' || c == '/') {

int num2 = peek(stack)->data.digit;

pop(stack);

int num1 = peek(stack)->data.digit;

pop(stack);

addDigit(stack, calculateInfixOperation(c, num1, num2));

}

}

return peek(stack)->data.digit;

}

int countOperators(const string& expr) {

int counter = 0;

counter += count(expr.begin(), expr.end(), '+');

counter += count(expr.begin(), expr.end(), '-');

counter += count(expr.begin(), expr.end(), '\*');

counter += count(expr.begin(), expr.end(), '/');

return counter;

}

int countNumbers(string& expr) {

int counter = 0;

for (int i = 0; i < expr.length(); i++) {

char c = expr[i];

if (isdigit(c)) {

string number = getNumberFromString(expr, i);

counter++;

}

}

return counter;

}

bool checkExpr(string& expr) {

if (expr.empty()) return false;

if (count(expr.begin(), expr.end(), '(') != count(expr.begin(), expr.end(), ')')) {

return false;

}

if (countNumbers(expr) - countOperators(expr) != 1) {

return false;

}

return true;

}

bool checkDictIncludes(char letter, map<char, int>& dict) {

return bool(dict[letter]);

}

string cutString(string& inp, unsigned startIdx, unsigned endIdx) {

string out;

for (int i = 0; i < inp.length(); i++) if (i >= startIdx && i < endIdx) out += inp[i];

return out;

}

string replaceVariablesInString(string& inp, map<char, int>& dict) {

string out = inp;

string inpBefore;

string inpAfter;

for (int i = 0; i < inp.length(); i++) {

if (checkDictIncludes(inp[i], dict)) {

inpBefore = cutString(out, 0, i);

inpAfter = cutString(out, i + 1, inp.length());

out = inpBefore + to\_string(dict[inp[i]]) + inpAfter;

}

}

return out;

}

void printMap(map<char, int> dict) {

for (const auto& [key, value] : dict)

cout << key << " " << value << "\n";

}

int main() {

setlocale(LC\_ALL, "Russian");

string infixExpr;

string postfixExpr;

string prefixExpr;

string test;

char actionType;

short unsigned choiseType;

short int workPoint;

map<char, int> variables {};

while(true) {

cout << "\nNavigation\n"

<< "1) Convert from infix to postfix and prefix:\n"

<< "2) Check expression\n"

<< "3) Calculate expression\n";

cin.clear(); *// Clearing the input stream from possible errors*

cin.sync();

cout << "Select point of work (number 1 to 3): ";

cin >> workPoint;

switch (workPoint) {

case 1: {

cout << "Write an infix expression:\n";

cin.clear();

cin.sync();

getline(cin, infixExpr);

cout << "Are you used variables in expression? (Y/N): \n";

cin >> actionType;

switch (actionType) {

case 'Y': {

string varLine;

cout << "Write all variables (new var. = new line, enter <S> to stop it) (example: a=24):\n";

cin.clear();

cin.sync();

while (true) {

getline(cin, varLine);

if (varLine.length() < 3) break;

int num = stoi(cutString(varLine, 2, varLine.length()));

variables[varLine[0]] = num;

}

infixExpr = replaceVariablesInString(infixExpr, variables);

break;

}

}

if (!checkExpr(infixExpr)) {

cout << "\nInvalid Input";

break;

}

postfixExpr = fromInfixToPostfix(infixExpr);

cout << "\nPostfix: " << postfixExpr << "\n";

prefixExpr = fromInfixToPrefix(infixExpr);

cout << "\nPrefix: " << prefixExpr << "\n";

break;

}

case 2: {

cout << "\nChoose the expression type:"

<< "\n 1 - Infix"

<< "\n 2 - Prefix"

<< "\n 3 - Postfix\n";

cin >> choiseType;

cout << "Write an expression:\n";

cin.clear();

cin.sync();

switch (choiseType) {

case 1: {

getline(cin, infixExpr);

if (!checkExpr(infixExpr)) {

cout << "\nInvalid Input";

infixExpr = "";

break;

}

cout << "Cool";

break;

}

case 2: {

getline(cin, prefixExpr);

if (!checkExpr(prefixExpr)) {

cout << "\nInvalid Input";

prefixExpr = "";

break;

}

cout << "Cool";

break;

}

case 3: {

getline(cin, postfixExpr);

if (!checkExpr(postfixExpr)) {

cout << "\nInvalid Input";

postfixExpr = "";

break;

}

cout << "Cool";

break;

}

default: {

cout << "\nYou entered an incorrect value\n";

break;

}

}

break;

}

case 3: {

cout << "\nChoose the expression type:"

<< "\n 1 - Infix"

<< "\n 2 - Prefix"

<< "\n 3 - Postfix\n";

cin >> choiseType;

cout << "Do you want to use old expression from memory< or write new? (O/N)\n";

cin >> actionType;

if (actionType == 'O') {

switch (choiseType) {

case 1: {

postfixExpr = fromInfixToPostfix(infixExpr);

cout << "\nResult: " << calculatePostfix(postfixExpr);

break;

}

case 2: {

string fakePostfix = prefixExpr;

reverseString(fakePostfix);

reverseNumbersInString(fakePostfix);

cout << "\nResult: " << calculatePostfix(fakePostfix);

break;

}

case 3: {

cout << "\nResult: " << calculatePostfix(postfixExpr);

break;

}

default: {

cout << "\nYou entered an incorrect value\n";

break;

}

}

break;

}

cout << "Write an expression:\n";

cin.clear();

cin.sync();

switch (choiseType) {

case 1: {

getline(cin, infixExpr);

if (!checkExpr(infixExpr)) {

cout << "\nInvalid Input";

break;

}

postfixExpr = fromInfixToPostfix(infixExpr);

cout << "\nResult: " << calculatePostfix(postfixExpr);

break;

}

case 2: {

getline(cin, prefixExpr);

if (!checkExpr(prefixExpr)) {

cout << "\nInvalid Input";

break;

}

string fakePostfix = prefixExpr + "";

reverseString(fakePostfix);

reverseNumbersInString(fakePostfix);

cout << "\nResult: " << calculatePostfix(fakePostfix);

break;

}

case 3: {

getline(cin, postfixExpr);

if (!checkExpr(postfixExpr)) {

cout << "\nInvalid Input";

break;

}

cout << "\nResult: " << calculatePostfix(postfixExpr);

break;

}

default: {

cout << "\nYou entered an incorrect value\n";

break;

}

}

break;

}

default: {

cout << "\n" << "You did not enter a number in the range from 1 to 3";

break;

}

}

cin.clear(); *// Clearing the input stream from possible errors*

cin.sync();

char stopFlag;

cout << "\n" << "Continue the program? (Y/N) ";

cin >> stopFlag;

if (stopFlag != 'Y' && stopFlag != 'y') break;

}

return 0;

}