**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ФКТИ**

отчет

**по лабораторной работе №1**

**по дисциплине «Алгоритмы и структуры данных»**

Тема: Списочные структуры

|  |  |  |
| --- | --- | --- |
| Студент гр. 3376 |  | Михайлов Н.Ф. |
| Преподаватель |  | Молдовян Д.Н. |

Санкт-Петербург

2024

**Лабораторная работа №1**

**СПИСОЧНЫЕ СТРУКТУРЫ**

**Цель работы:** изучить логику работы различных списочных структур, таких как двусвязный список, динамический массив и стек, реализовать их самостоятельно и с помощью них написать алгоритм, преобразующий обычную инфиксную запись в постфиксную (обратную польскую).

**Задание:** реализовать следующие структуры: двусвязный список*,*динамический массив *и* стек*.* Стек можно реализовать как на базе списка, так и отдельно. Использовать стек для реализации алгоритма сортировочной станции. Разрешённые символы в исходном выражении: +, -, \*, /, ^, sin, cos, (, ), 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Для упрощения разбиения входной строки на токены разрешается отделять каждый символ пробелом.

**Базовый теоретический материал:**

1. **Список** — это структура данных, представляющая собой упорядоченный набор элементов, где каждый элемент имеет своё положение (индекс). Элементы списка могут быть разных типов, и количество элементов может меняться. В зависимости от языка программирования, список может поддерживать такие операции, как добавление, удаление, поиск, сортировка и доступ к элементам по индексу.   
   **Списков можно выделить несколько типов:**
   1. **Односвязные** — это списки, где у каждого элемента есть ссылка только на следующий элемент.
   2. **Двусвязные списки** — это списки, где у каждого элемента есть ссылки не только не следующий, но и на предыдущий элементы.
   3. **Двусвязные кольцевые списки** — это двусвязные списки, которые в отличии от одно и двухсвязных списков не оканчиваются нулём, а ссылаются на какой-то из уже существующих элементов списка, тем самым зацикливаясь.
2. **Динамический массив** (он же **вектор**)— это структура данных, представленная массивом с возможностью при надобности увеличивать свою вместимость.
3. **Стек** —это структура данных, которая работает по логике «Первый вошёл, последний вышел». То есть элементы добавляются в начало стека (в вершину) и достаются тоже с вершины.  
   **Основные операции со стеком:**
   1. **Push** — добавление элемента на вершину стека.
   2. **Pop** — удаление элемента из вершины стека.
   3. **Peek (Top)** — просмотр элемента с вершины стека без его удаления.
   4. **isEmpty** — проверка стека на пустоту.

**Алгоритм решения задачи:**

1. Пользователем вводится строка equation.
2. С помощью функции Tokenize() строка разбивается на элементы (токены):
   1. Число (Number)
   2. Оператор (Operator)
   3. Функция (Function)
   4. Открывающая скобка (Open paren)
   5. Закрывающая скобка (Close paren)
3. Алгоритм работы функции Tokenize():
   1. В строке берётся каждый символ отдельно.
   2. Если это пробел — ничего не делаем.
   3. Если это число или точка (“.”) и при этом нынешний символ существует и предыдущий символ — это число, то приписываем этот символ в строчку, где пока что хранится строчка, в которой лежит что-то сборное (будь то функция или число, не цифра).
   4. Если же предыдущее условие не выполняется, а нынешний символ не пустой, то добавляем этот единственный символ в вектор, как одну единственную цифру.
   5. Если же нынешний символ — это буква, то идём по строке до тех пор, пока не наткнёмся на не букву. Добавляем получившуюся строку как sin, cos или другую функцию.
   6. Далее мы проверяем не будет ли этот символ оператором или открывающей/закрывающей скобкой. Если является, то добавляем в вектор с описанием того, что это.
   7. Возвращаем вектор с строчкой разбитой на токены.
4. Далее вы вызываем функцию ToPostfix(), которая создаст нам постфиксную запись из тех токенов, которые мы получили.
5. Алгоритм работы функции Tokenize():
   1. Проходим по всем полученным токенам.
   2. Проверяем токен на число. Если это число, то добавляем его в стек.
   3. Проверяем токен на оператора. Если это оператор, то мы смотрим больше ли приоритет этого оператора приоритета оператора, который уже лежит на вершине списка. Если да, то мы оператор с вершины списка добавляем в итоговый вектор, а этот оператор добавляем на вершину списка.
   4. Проверяем токен на открывающую скобку. Если это открывающая скобка, то добавляем её в стек.
   5. Проверяем токен на закрывающую скобку. Если это закрывающая скобка, то проверяем, если стек не пустой, и последний элемент в стеке не открывающая скобка, то добавляем последний элемент в стеке в итоговый вектор, а закрывающую скобку добавляем в стек.
   6. Переносим оставшиеся операторы из стека в итоговый вектор.
6. Последним шагом просто выводим итоговый вектор посимвольно на экран.

**Трудоёмкость алгоритма:**

1. GetOperatorPriority(char op)— O(1), т.к. количество проверок фиксировано.
2. Tokenize(const std::string& expression) — в функции единственный проход по строчке длины n, так что сложность O(n).
3. ToPostfix(const CustomVector<Element>& elements) — в функции единственный проход по вектору элементов длины m, так что сложность O(m).
4. В самой функции main функции под номером 2 и 3 вызываются последовательно, так что сложность O(n+m).

**Код алгоритма:**

**main.cpp**

#include <iostream>

//#include "CustomList.h"

#include "CustomVector.h"

#include "CustomVector.cpp"

#include "CustomStack.h"

// Типы элементов в строке

enum ElementType { NUMBER, OPERATOR, FUNCTION, OPEN\_PAREN, CLOSE\_PAREN };

// Структура для сопоставления элемента с его типом

struct Element {

std::string value;

ElementType type;

};

// Функция для определения приоритета операторов

int GetOperatorPriority(char op) {

if (op == '+' || op == '-') return 1;

if (op == '\*' || op == '/') return 2;

if (op == '^') return 3;

return 0;

}

// Функция для разбиения строки на элементы

CustomVector<Element> Tokenize(const std::string& expression) {

CustomVector<Element> elements;

std::string currentToken;

for (size\_t i = 0; i < expression.size(); i++) {

char c = expression[i];

if (isspace(c)) continue; // Пропускаем пробелы

// Если число (поддерживаем десятичные дроби)

if (isdigit(c) || (c == '.' && !currentToken.empty() && isdigit(currentToken.back()))) {

currentToken += c;

}

else {

if (!currentToken.empty()) {

elements.PushBack({ currentToken, NUMBER });

currentToken.clear();

}

//sin, cos

if (isalpha(c)) {

std::string func;

while (isalpha(expression[i])) {

func += expression[i++];

}

i--;

elements.PushBack({ func, FUNCTION });

}

// Операторы и скобки

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

elements.PushBack({ std::string(1, c), OPERATOR });

}

else if (c == '(') {

elements.PushBack({ std::string(1, c), OPEN\_PAREN });

}

else if (c == ')') {

elements.PushBack({ std::string(1, c), CLOSE\_PAREN });

}

}

}

if (!currentToken.empty()) {

elements.PushBack({ currentToken, NUMBER });

}

return elements;

}

// Отладочная печать разобранных элементов

void PrintTokens(const CustomVector<Element>& elements) {

for (const auto& element : elements) {

std::cout << "Token: " << element.value << " Type: "

<< (element.type == NUMBER ? "Number" :

element.type == OPERATOR ? "Operator" :

element.type == FUNCTION ? "Function" :

element.type == OPEN\_PAREN ? "Open Parenthesis" : "Close Parenthesis")

<< std::endl;

}

}

CustomVector<Element> ToPostfix(const CustomVector<Element>& elements) {

CustomStack<Element> operatorStack; // Стек для операторов

CustomVector<Element> output; // Вектор для результата

for (const auto& element : elements) {

if (element.type == NUMBER || element.type == FUNCTION) {

// Если это число или функция, добавляем его в output

output.PushBack(element);

}

else if (element.type == OPERATOR) {

// Обрабатываем оператор

while (!operatorStack.Empty() &&

GetOperatorPriority(operatorStack.Top().value[0])

>= GetOperatorPriority(element.value[0])) {

output.PushBack(operatorStack.Top());

operatorStack.Pop();

}

operatorStack.Push(element); // Добавляем текущий оператор в стек

}

else if (element.type == OPEN\_PAREN) {

//Добавляем в стек открывающуюся скобку

operatorStack.Push(element);

}

else if (element.type == CLOSE\_PAREN) {

// Обрабатываем закрывающую скобку

while (!operatorStack.Empty() && operatorStack.Top().type != OPEN\_PAREN) {

output.PushBack(operatorStack.Top());

operatorStack.Pop();

}

operatorStack.Pop(); // Удаляем открывающуюся скобку из стека

}

}

// Переносим оставшиеся операторы в выходной вектор

while (!operatorStack.Empty()) {

output.PushBack(operatorStack.Top());

operatorStack.Pop();

}

return output;

}

int main() {

std::string equation;

std::cout << "Введите строку для преобзования в обратную польскую запись: ";

std::cin >> equation;

CustomVector<Element> postfix = ToPostfix(Tokenize(equation));

// Печать результата

std::cout << "Обратная польская нотация: ";

for (const auto& elem : postfix) {

std::cout << elem.value << " ";

}

std::cout << std::endl;

}

**CustomList.h**

#pragma once

#include <iostream>

template <typename T>

class CustomList{

public:

CustomList() : head(nullptr), tail(nullptr), size(0) {};

~CustomList() {

Node\* current = head;

while (current != nullptr) {

Node\* nextNode = current->next;

delete current;

current = nextNode;

}

};

// Операции с элементами списка

void PushFront(const T& value);

void PushBack(const T& value);

void PopFront();

void PopBack();

T& Front() const { return head->data; };

T& Back() const { return tail->data; };

T& GetValue(unsigned t\_index) const {

Node\* t\_head = head;

for (size\_t i = 0; i < size; i++, t\_head = t\_head->next) {

if (t\_head->index == t\_index) return t\_head->data;

}

};

bool Empty() const { return size == 0; };

size\_t GetSize() const { return size; };

void Clear();

// Вспомогательные методы

void Print() const;

private:

struct Node {

T data;

size\_t index;

Node\* next;

Node\* prev;

Node(const T& data, Node\* prev = nullptr, Node\* next = nullptr, size\_t index = 0)

: data(data), prev(prev), next(next), index(index) {}

};

Node\* head;

Node\* tail;

size\_t size;

};

**CustomList.cpp**

#include "CustomList.h"

template <typename T>

void CustomList<T>::PushFront(const T& value) {

Node\* newNode = new Node(value);

if (head == nullptr) {

head = newNode;

tail = newNode;

}

else {

head->prev = newNode;

newNode->next = head;

head = newNode;

Node\* current = head->next;

while (current != nullptr) {

current->index++;

current = current->next;

}

}

head->index = 0;

size++;

}

template <typename T>

void CustomList<T>::PushBack(const T& value) {

Node\* newNode = new Node(value);

if (tail == nullptr) {

head = newNode;

tail = newNode;

newNode->index = 0;

}

else {

tail->next = newNode;

newNode->prev = tail;

newNode->index = size;

tail = newNode;

}

size++;

}

template <typename T>

void CustomList<T>::PopFront() {

if (head == nullptr) return;

Node\* temp = head;

head = head->next;

if (head != nullptr) {

head->prev = nullptr;

Node\* current = head;

while (current != nullptr) {

current->index--;

current = current->next;

}

}

else {

tail = nullptr;

}

delete temp;

size--;

}

template <typename T>

void CustomList<T>::PopBack() {

if (tail == nullptr) return;

Node\* temp = tail;

tail = tail->prev;

if (tail != nullptr) {

tail->next = nullptr;

}

else {

head = nullptr;

}

delete temp;

size--;

}

template <typename T>

void CustomList<T>::Clear() {

while (tail) {

Node\* t\_current = tail->prev;

delete tail;

tail = t\_current;

size--;

}

}

template <typename T>

void CustomList<T>::Print() const {

Node\* current = head;

bool first = true;

std::cout << "[";

for (size\_t i = 0; i < size; i++, current = current->next) {

if (first) { std::cout << " " << current->data; first = false; }

else std::cout << ", " << current->data;

}

std::cout << " ]" << std::endl;

}

**CustomVector.h**

#pragma once

#include <iostream>

#include <stdexcept>

template <typename T>

class CustomVector {

public:

//Конструтор вектора

CustomVector(size\_t initialCapacity = 2);

~CustomVector();

//Добавить элемент в конец массива

void PushBack(const T& value);

//Вырезать элемент с конца массива

void PopBack();

//Перегрузка оператора чтобы обращатся по индексам как обычно

T& operator[](size\_t index);

//Получения размера вектора

size\_t GetSize() const;

//Получение вместимости

size\_t GetCapacity() const;

//Печать массива

void Print() const;

//Класс функций, чтобы работал синтаксис типа (const auto& item : vector)

T\* begin() { return array; }

T\* end() { return array + size; }

const T\* begin() const { return array; }

const T\* end() const { return array + size; }

private:

T\* array; //Указатель на массив

size\_t size; //Размер массиа

size\_t capacity; //Вместимость массива

void Resize(size\_t newCapacity); //Функция изменения размера массива

};

**CustomVector.cpp**

#include "CustomVector.h"

template <typename T>

CustomVector<T>::CustomVector(size\_t initialCapacity) : size(0), capacity(initialCapacity) {

array = new T[capacity];

}

template <typename T>

CustomVector<T>::~CustomVector() {

delete[] array;

}

template <typename T>

void CustomVector<T>::Resize(size\_t newCapacity) {

T\* newData = new T[newCapacity];

for (size\_t i = 0; i < size; ++i) {

newData[i] = array[i];

}

delete[] array;

array = newData;

capacity = newCapacity;

}

template <typename T>

void CustomVector<T>::PushBack(const T& value) {

if (size == capacity) {

Resize(capacity \* 2);

}

array[size] = value;

size++;

}

template <typename T>

void CustomVector<T>::PopBack() {

if (size > 0) {

size--;

}

else {

throw std::out\_of\_range("Массив пуст!");

}

}

template <typename T>

T& CustomVector<T>::operator[](size\_t index) {

if (index >= size) {

throw std::out\_of\_range("Вышли за границы массива!");

}

return array[index];

}

template <typename T>

size\_t CustomVector<T>::GetSize() const {

return size;

}

template <typename T>

size\_t CustomVector<T>::GetCapacity() const {

return capacity;

}

template <typename T>

void CustomVector<T>::Print() const {

for (size\_t i = 0; i < size; ++i) {

std::cout << array[i] << " ";

}

std::cout << std::endl;

}

**CustomStack.h**

#pragma once

#include "CustomList.h"

#include "CustomList.cpp"

template <typename T>

class CustomStack {

public:

//Добавляет элемент

void Push(const T& value) { list.PushBack(value); }

//Удаляет верхний элемент

void Pop() { if (!list.Empty()) list.PopBack(); }

//Возвращает верхний элемент

T& Top() { return list.Back(); }

//Проверка на пустоту

bool Empty() const { return list.Empty(); }

//Возврашает размеры стэка

size\_t Size() const { return list.GetSize(); }

//Чистит стэк

void Clear() { list.Clear(); }

//Выводит стэк

void Print() const { list.Print(); }

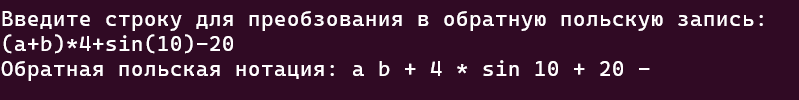
private:

CustomList<T> list;

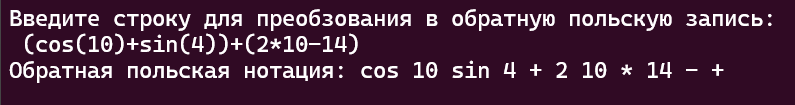
};

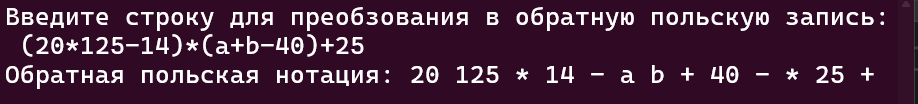
**CustomStack.cpp**

#include "CustomStack.h"

**Примеры работы программы:**

*рис.1. Ввод примера №1 и вывод ОПН.*

**

*рис.2. Ввод примера №2 и вывод ОПН.*

*рис.3. Ввод примера №3 и вывод ОПН.*

**Вывод:**

В ходе выполнения лабораторной работы было достигнуто понимание работы таких списочных структур как двусвязный список, динамический массив и стек, и была написана их реализация. Так же в работе был создан алгоритм по переводу инфиксной записи выражений в постфиксную (ОПН) и продемонстрирована его работа.