МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №4
по дисциплине «Программирование»
Тема: Динамические структуры данных

Студент гр. 0382 _____ Литягин С.М.

Преподаватель _____ Берленко Т.А.

Санкт-Петербург
2021

Цель работы.

Изучение динамических структур данных.

Задание.

Требуется написать программу, получающую на вход строку, (без кириллических символов и не более 3000 символов) представляющую собой код "простой" html-страницы и проверяющую ее на валидность. Программа должна вывести **correct** если страница валидна или **wrong**.

Html-страница, состоит из тегов и их содержимого, заключенного в эти теги. Теги представляют собой некоторые ключевые слова, заданные в треугольных скобках. Например, <**tag**> (где tag - имя тега). Область действия данного тега распространяется до соответствующего закрывающего тега </**tag**> который отличается символом /. Теги могут иметь вложенный характер, но не могут пересекаться:

<tag1><tag2></tag2></tag1> - верно;

<tag1><tag2></tag1></tag2> - не верно;

Существуют теги, не требующие закрывающего тега.

Валидной является html-страница, в коде которой всякому открывающему тегу соответствует закрывающий (за исключением тегов, которым закрывающий тег не требуется)

Во входной строке могут встречаться любые парные теги, но гарантируется, что в тексте, кроме обозначения тегов, символы < и > не встречаются. атрибутов у тегов также нет.

Теги, которые не требуют закрывающего тега:
, <hr>

Класс стека (который потребуется для алгоритма проверки парности тегов) требуется реализовать самостоятельно на базе **списка**. Для этого необходимо:

Реализовать **класс** CustomStack, который будет содержать перечисленные ниже методы. Стек должен иметь возможность хранить и работать с типом данных ***char****

Структура класса узла списка:

```
struct ListNode {
   ListNode* mNext;
   char* mData;
};
```

Объявление класса стека:

```
class CustomStack {
public:
// методы push, pop, size, empty, top + конструкторы, деструктор
private:
// поля класса, к которым не должно быть доступа извне
protected:  // в этом блоке должен быть указатель на голову
   ListNode* mHead;
};
```

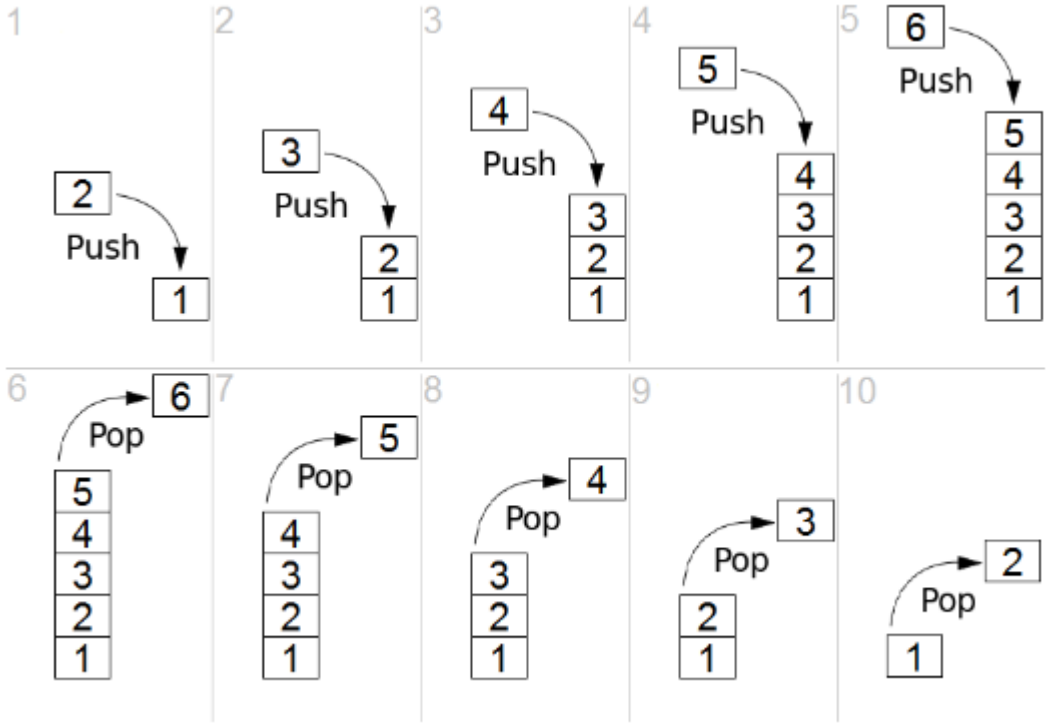
Перечень методов класса стека, которые должны быть реализованы:

- **void push(const char* tag)** - добавляет новый элемент в стек
- **void pop()** - удаляет из стека последний элемент
- **char* top()** - доступ к верхнему элементу
- **size_t size()** - возвращает количество элементов в стеке
- **bool empty()** - проверяет отсутствие элементов в стеке

Основные теоретические положения.

Стек - это структура данных, в которой хранятся элементы в виде последовательности, организованной по принципу LIFO (Last In — First Out). Такую структуру данных можно сравнить со стопкой тарелок или магазином автомата. Стек не предполагает прямого доступа к элементам, и список основных операций ограничивается операциями помещения элемента в стек и извлечения элемента из стека. Их принято называть PUSH и POP соответственно. Также, обычно есть возможность посмотреть на верхний элемент стека, не извлекая его (TOP) и несколько других функций, таких как проверка на пустоту стека и некоторые другие.

Пример добавления и удаления элементов из непустого стека (содержащего единицу):



Выполнение работы.

Реализация класса стека *CustomStack*:

В блоке *protected* объявлен указатель *ListNode* mHead = nullptr*.

Функция *void push(const char* tag)* добавляет новый элемент в стек. Создается новый элемент списка, поле *mNext* будет указывать на *mHead*, полю

mData присваивается содержимое переменной *tag*. Новый элемент становится на место головы списка.

Функция *void pop()* удаляет из стека последний элемент. Здесь проверяется, если стек пустой, то выводится *wrong* и программа завершает работу. Иначе – головой списка становится второй элемент списка, а первый – удаляется.

Функция *char* top()* возвращает данные, хранящиеся в поле *mData* головы списка. Если стек не имеет элементов, выводится *wrong*, иначе выводится содержимое *mData*.

Функция *size_t size()* возвращает количество элементов стека. Считает элементы списка, пока не встретит *nullptr*.

Функция *bool empty()* проверяет стек на пустоту. Если функция *size()* возвращает 0, то данная функция возвращает *true*. Иначе – *false*.

Реализация основной функции:

Объявляются переменные. Объявляется *CustomStack stack*. Считывание производится в переменную *text* с помощью функции *fgets()*.

Далее происходит поиск слов, заключенных в “<>”. Слово запоминается в переменную *buff*. Если первый символ не является “/”, то слово добавляется в стек *stack.push(buff)*. Иначе – сравнивается содержимое *buff+1* (т.е. так мы сравниваем слово без “/”) и *stack.top()*. Если слова сходятся, то удаляем элемент стека *stack.pop()*. Иначе выводим *wrong* и завершаем работу.

Когда алгоритм проверит всю строку, выполняем проверку стека на пустоту *stack.empty()*. Если пустой – то выводим *correct*. Иначе - *wrong*.

Разработанный программный код см. в приложении А.

Тестирование.

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№п/п	Входные данные	Выходные данные	Результат
1	<html><head><title>HTML Document</title></head><body> This text is bold, this is bold and italics </body></html>	correct	Программа работает верно
2	<html><head>fff pop opa <head></html>	wrong	Программа работает верно

Выводы.

В ходе работы были изучены принципы работы с динамическими структурами данных на языке C++. Был реализован стек на базе линейного списка, а также была написана программа, выполняющая проверку на валидность строку, представляющую собой код html-страницы.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <cstring>
#include <cstdlib>

using namespace std;

typedef struct ListNode{
    ListNode* mNext;
    char* mData;
}ListNode;

class CustomStack {
        public:
            void print(){
                ListNode* end = mHead;
                while(end != nullptr){
                    cout << end -> mData << endl;
                    end = end -> mNext;
                }
            }
            void push(const char* tag){
                ListNode* new_el = new ListNode;
                new_el -> mData = new char[strlen(tag)];
                strcpy(new_el -> mData,tag);
                new_el -> mNext = mHead;
                mHead = new_el;
            }
            void pop(){
                if(size() == 0){
                    cout << "wrong" << endl;
                    exit(0);
                }
                ListNode* tmp = mHead;
                delete mHead -> mData;
                mHead = mHead -> mNext;
                delete tmp;
            }
            char* top(){
                if(size() == 0){
                    cout << "wrong" << endl;
                    exit(0);
                }
                return mHead -> mData;
            }
            size_t size(){
                size_t size_s = 0;
                ListNode* end = mHead;
                while(end != nullptr){
                    size_s++;
                    end = end -> mNext;
                }
                return size_s;
```

```
            }
            bool empty(){
                if(size() == 0){
                    return true;
                }
                return false;
            }
        protected:
            ListNode* mHead = nullptr;
        };

int main() {
    char* text = new char[3000];
    fgets(text, 3000, stdin);
    CustomStack stack;
    int index = 0;
    int i = 0;
    int help = 0;

    while(help <= strlen(text)){
        if(text[help] == '<'){
            char* buff = new char[30];
            while(text[help] != '>'){
                i++;
                help++;
            }
            strncpy(buff, text+help - i +1, i-1);
            if(strcmp(buff, "br") && strcmp(buff, "hr")){
              if(buff[0]!='/'){
                stack.push(buff);
              }
              else{
                if(!strcmp(stack.top(), buff+1)) stack.pop();
                else{
                  cout << "wrong" << endl;
                  return 0;
                }
              }
            }
            buff = nullptr;
            i = 0;
        }
        help++;
    }
    if(stack.empty()) cout << "correct" << endl;
    else cout << "wrong" << endl;

    return 0;
}
```