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. 3341 _____ Рябов М.Л.

Преподаватель _____ Глазунов С.А.

Санкт-Петербург

2024

Цель работы

Написать программу, реализующую моделирование работы стека на базе списка. Для этого необходимо создать класс CustomStack с методами push, pop, top, size, empty, которые будут работать с элементами типа int. Программа должна обрабатывать команды из потока ввода stdin и выполнять соответствующие действия согласно протоколу:

- push: добавление целого числа n в стек.
- pop: удаление последнего элемента из стека и вывод его значения.
- top: вывод верхнего элемента стека.
- size: вывод количества элементов в стеке.
- empty: показывает, пустой ли стек или нет.

При возникновении ошибок (например, вызов метода pop или top при пустом стеке), программа должна проигнорировать команду и\или завершиться.

Примечания:
- Указатель на голову стека должен быть защищенным (protected).
- Необходимо использовать предоставленную структуру ListNode.
- Не требуется подключение дополнительных заголовочных файлов.
- Не нужно использовать using для пространства имен std.

Задание

Вариант 5

Моделирование стека.

Требуется написать программу, моделирующую работу стека на базе списка. Для этого необходимо:

Расстановка тегов.

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.

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. аттрибутов 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**.

Основные теоретические положения

Стек (stack) - это абстрактная структура данных, которая представляет собой коллекцию элементов, организованных по принципу Last In First Out (LIFO). Это означает, что элементы добавляются и удаляются из стека только с одного конца, называемого вершиной стека.

Вот основные теоретические положения о стеке:

1. Операции со стеком:

- push(): добавляет элемент на вершину стека.
- pop(): удаляет и возвращает элемент с вершины стека.
- top(): возвращает элемент, находящийся на вершине стека, без его удаления.
- empty(): проверяет, пуст ли стек.
- size(): возвращает количество элементов в стеке.

2. Вершина стека:

- Вершина стека - это элемент, добавленный последним. Она представляет последний добавленный и первый удаляемый элемент стека.

3. Реализация стека:

- Стек можно реализовать с помощью статического массива, динамического массива или связного списка.
- В C++ стандартная библиотека содержит класс std::stack, который представляет стек.

4. Применение стека:

- Стек широко используется в программировании. Некоторые примеры использования стека:

- Рекурсивные вызовы функций.

- Обработка операций в обратной польской записи (постфиксной нотации).

- Управление операциями возврата (backtracking).

- Обработка операций undo/redo.

- Решение задач на графах (DFS - Depth First Search).

5. Важность стека:

- Использование стека позволяет эффективно управлять данными, сохраняя порядок их добавления и удаления.

- Стек обеспечивает простой доступ к последнему добавленному элементу и удобство его обработки.

Стек - это важная структура данных, которая играет ключевую роль во многих алгоритмах и программах. Понимание его основных принципов и операций поможет в разработке эффективных и легко поддерживаемых программ.

Выполнение работы

Ход работы по коду:

1. Создается класс CustomStack, содержащий методы для работы со стеком и управления элементами:

- push(int data): добавляет элемент на вершину стека.
- pop(): удаляет элемент с вершины стека.
- top(): возвращает значение элемента на вершине стека.
- size(): возвращает количество элементов в стеке.
- empty(): проверяет, пуст ли стек.

2. Реализованы методы:

Функция char** getTags(char* str, int* capacityTags):

- Эта функция принимает строку str и указатель на переменную capacityTags, которая будет хранить количество тегов.

- Выделяется память под массив указателей tags, который будет содержать найденные теги.

- Затем происходит перебор символов в строке str. Если встречается символ <, то начинается поиск тега.

- Для каждого найденного тега выделяется память под строку tag, в которую копируется содержимое тега.

- Если размер строки tag превышает ее текущую емкость, память под строку увеличивается с помощью realloc.

- После завершения поиска тега он добавляется в массив tags, а количество тегов увеличивается.

- В конце функция возвращает массив tags.

Функция int checkStatusTag(char* tag):

- Эта функция принимает строку tag, представляющую тег.

- Проверяет, является ли тег
 или <hr>. Если да, возвращает 2.

- Иначе проверяет, является ли тег открывающим или закрывающим. Если закрывающий, возвращает 1, иначе 0.

Функция int compareTags(char* stackTag, char* tag):

- Принимает две строки: stackTag

- верхний элемент стека и tag - текущий тег.

- Создает новую строку newTag, из которой убирает символ / (если он есть) из тега.

- Сравнивает верхний элемент стека с новым тегом и возвращает результат сравнения.

Функция int successHTML(char* str):

- Принимает строку str, представляющую HTML-код.

- Получает все теги из строки с помощью функции getTags() и сохраняет их в массив tags.

- Создает стек stack.

- Перебирает все теги: если тег открывающий, помещает его в стек; если закрывающий и соответствует верхнему элементу стека, удаляет его из стека; если не соответствует, возвращает 0.

- По завершении проверки всех тегов, если стек пустой, возвращает 1 (успешное завершение), иначе 0.

5. В main() функции происходит считывание строки, вызов основной функции successHTML() и в зависимости от возвращаемого значения выводит wrong или correct

Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	<html><head></ title><title>HTML Document</title></head><b ody> Semi-bold textas also italic. </body>< /html>	wrong	Проверка на наличие одного закрывающего тега, не имеющего открытого
2.	<html><head><title>HTML Document</title></head><b ody> Semi-bold textas also italic. </body>< /html>	correct	Проверка на валидность HTML разметки

Выводы

Цель программы была успешно достигнута. Был создан класс CustomStack, реализующий моделирование работы стека на базе списка. Программа обрабатывает теги из потока ввода stdin и выполняет соответствующие действия согласно протоколу, включая добавление элементов в стек, удаление последнего элемента, вывод верхнего элемента, вывод количества элементов и проверка на наличие элементов. При возникновении ошибок, таких как вызов метода pop или top при пустом стеке, программа обрабатывает данные случаи, что предотвращает ее некорректное поведение. Все требования к реализации были выполнены.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.c

```
#define END_STR '\0'
#define SIZE_BUFFER 3000

class CustomStack{

    size_t sizeStack = 0;

    public:

        CustomStack(){
            this->mHead = NULL;
        }

        void push(const char* tag){
            this->sizeStack += 1;
            if (this->sizeStack - 1 == 0){
                this->mHead = createNode(tag);
                return;
            }


            ListNode* newElem = createNode(tag);
            newElem->mNext = this->mHead;
            this->mHead = newElem;
            return;
```

```
}

void pop(){
    if(this->sizeStack == 0)
        return;

    if(this->sizeStack == 1){
        free(mHead);
        this->mHead == NULL;
        this->sizeStack -= 1;
        return;
    }

    ListNode* tmp = this->mHead;
    this->mHead = this->mHead->mNext;
    this->sizeStack -= 1;
    free(tmp);
    return;
}

char* top(){
    if(sizeStack == 0)
        return NULL;

    return this->mHead->mData;
}

size_t size(){
    return this->sizeStack;
}
```

```
    bool empty(){
        if(this->sizeStack == 0)
            return true;
        else
            return false;
    }

    void print(){
        cout << "\n\nStack have " << this->sizeStack << " elements" << endl;

        if(this->sizeStack == 0)
            return;

        ListNode* cur = this->mHead;

        while(cur->mNext != NULL){
            cout << cur->mData << endl;
            cur = cur->mNext;
        }
        cout << cur->mData << endl;
    }

private:
    ListNode* createNode(const char* data){
        ListNode* tmp = (ListNode*)malloc(sizeof(ListNode));
        tmp->mNext = NULL;
        tmp->mData = (char*)data;
        return tmp;
    }
```

```
    protected:
        ListNode* mHead;

};

char** getTags(char* str, int* capacityTags){
    char** tags = (char**)malloc(sizeof(char*)*100);
    char* tag;

    int sizeStr = strlen(str);
    for(int i = 0; i < sizeStr; i++)
    {

        if(str[i] == '<')
        {
            int j = i;
            int size = 0, capacity = 1;
            char* tag = (char*)malloc(sizeof(char) * capacity);

            while(str[j] != '>')
            {
                tag[size++] = str[j];
                if(size >= capacity)
                {
                    capacity *= 2;
                    tag = (char*)realloc(tag, sizeof(char)*capacity);
                }

                j++;
```

```
            }
            tag[size++] = str[j];
            tag[size] = '\0';
            tags[(*capacityTags)++] = tag;
        }
    }
    return tags;
}

int checkStatusTag(char* tag){
    if(!strcmp(tag, "<br>") || !strcmp(tag, "<hr>"))
        return 2; //<br> or <hr>

    char backSlash = '/';
    char* isClose = strchr(tag, backSlash);
    if(isClose != NULL)
        return 1; // close
    else
        return 0; // open
}

int compareTags(char* stackTag, char* tag){
    if(stackTag == NULL)
        return -1;

    int sizeTag = strlen(tag);
    int size = 0;
    char* newTag = (char*)malloc(sizeof(char) * (sizeTag - 1));

    for(int i = 0; i < sizeTag; i++){
```

```
        if(tag[i] == '/')
            continue;
        newTag[size++] = tag[i];
    }
    newTag[size] = '\0';
    return strcmp(stackTag, newTag);
}

int successHTML(char* str){
    int size = 0;
    int status;
    char** tags = getTags(str, &size);
    CustomStack stack;

    for(int i = 0; i < size; i++){
        status = checkStatusTag(tags[i]);
        if(status == 0)
            stack.push(tags[i]);

        else if(status == 1 && compareTags(stack.top(), tags[i]) == 0)
            stack.pop();

        else if (status == 1 && compareTags(stack.top(), tags[i]) != 0)
            return 0;
    }

    if(stack.empty())
        return 1;
    else
        return 0;
```

```
}

int main(){
    char* str = (char*)malloc(sizeof(char) * SIZE_BUFFER);
    cin.getline(str, 3000);
    int success = successHTML(str);
    if(success)
        cout << "correct" << endl;
    else
        cout << "wrong" << endl;
}
```