Especificação 3

Vitor Aguiar, Wiliam Masami, Yago Feitoza

03/07/2023

1 Problema

1.1 Enunciado

Existem duas alternativas à representação tradicional de expressões aritméticas, a notação posfixa e prefixa. Na posfixa o operador é expresso após seus operandos e na prefixa antes.

Alguns exemplos são:

infixa = a+b, posfixa = a b +, prefixa = + a b
infixa = (a+b)*c, posfixa = a b + c *, prefixa = + a b * c
infixa = a*((b-c)/d), posfixa= a b c - d / *, prefixa = * a / - b c d

1.2 Fundamento Teórico

A Notação Polonesa ou prefix notation é um método de operação aritmética em que o operador antecede os operandos. Por outro lado a Notação Polonesa Inversa ou RPN(Reverse Polish Notation) é uma notação matemática que admite que o operador vêm depois do operando, também sendo conhecida como postfix notation.

Operações aritméticas usando a notação prefixa e posfixa:

Primeiro passo: Varredura pela expressão.
Segundo passo: Se for operando, empurra na pilha.
Terceiro passo: Se for operador, retira os últimos dois items contidos na pilha e realiza sua operação.
Quarto passo: Empurra valor na pilha e continua varredura, até não sobrar mais operadores.

2 Resolução

Utilizando o ambiente da linguagem C:

```c
#ifndef MAIN_H
#define MAIN_H

typedef struct
{
    char expressao[200];
    int tamanho;

} Expressao;

typedef struct no
{
    char elemento[200];
    struct no *proximo;

} No;

typedef struct
{
    No *topo;

} Pilha;

/* Funções relacionadas ao TDA Expressao */
Expressao criar_expressao();                                                                    // Cria e retorna uma expressão vazia
void apagar_expressao(Expressao *expressao);                                                    // Apaga o conteúdo da expressão
void imprimir_expressao(const Expressao *expressao);                                            // Imprime a expressão na tela
void adicionar_elemento(Expressao *expressao, const char *elemento);                            // Adiciona um elemento à expressão
void converter_posfixa_para_prefixa(const Expressao *expressao_posfixa, Expressao *expressao_prefixa); // Converte uma expressão posfixa em prefixa
void converter_prefixa_para_posfixa(const Expressao *expressao_prefixa, Expressao *expressao_posfixa); // Converte uma expressão prefixa em posfixa
float resultado_expressao(const Expressao *expressao);                                          // Calcula e retorna o resultado da expressão

/* Funções relacionadas ao TDA Pilha */
Pilha *criar_pilha();                          // Cria e retorna uma pilha vazia
void apagar_pilha(Pilha *pilha);               // Libera a memória ocupada pela pilha
int pilha_vazia(const Pilha *pilha);           // Verifica se a pilha está vazia
void empilhar(Pilha *pilha, const char *elemento); // Empilha um elemento na pilha
void desempilhar(Pilha *pilha);                // Desempilha o elemento do topo da pilha
char *topo_pilha(const Pilha *pilha);          // Retorna o elemento do topo da pilha

/* Funções auxiliares */
int isOperador(char c);          // Verifica se um caractere é um operador (+, -, *, /)
int getPrioridade(char c);       // Obtém a prioridade de um operador (+, -, *, /)
void inverter_string(char *str); // Inverte uma string

#endif
```

Figura 1: main.h(Declarações do TDA)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "main.h"

Pilha *criar_pilha()
{
    Pilha *pilha = (Pilha *)malloc(sizeof(Pilha));
    pilha->topo = NULL;
    return pilha;
}

void apagar_pilha(Pilha *pilha)
{
    // Remove cada nó da pilha, liberando memória
    No *atual = pilha->topo;
    while (atual != NULL)
    {
        No *proximo = atual->proximo;
        free(atual);
        atual = proximo;
    }

    free(pilha);
}

int pilha_vazia(const Pilha *pilha)
{
    // Verifica se o topo da pilha é NULL
    // Se for, a pilha está vazia
    return (pilha->topo == NULL);
}

void empilhar(Pilha *pilha, const char *elemento)
{
    // Aloca memória para um novo nó
    No *novo_no = (No *)malloc(sizeof(No));

    // Copia o elemento para o campo 'elemento' do novo nó
    strcpy(novo_no->elemento, elemento);

    // Atualiza o campo 'proximo' do novo nó para apontar para o antigo topo da pilha
    novo_no->proximo = pilha->topo;

    // Atualiza o topo da pilha para apontar para o novo nó
    pilha->topo = novo_no;
}
```


Figura 2: func.c(Funções da TDA)

```c
void desempilhar(Pilha *pilha)
{
    // Verifica se a pilha está vazia
    if (pilha_vazia(pilha))
        return;

    // Remove o nó do topo da pilha e atualiza o topo para o próximo nó
    No *no_removido = pilha->topo;
    pilha->topo = pilha->topo->proximo;

    // Libera a memória ocupada pelo nó removido
    free(no_removido);
}

char *topo_pilha(const Pilha *pilha)
{
    // Verifica se a pilha está vazia
    if (pilha_vazia(pilha))
        // Se estiver vazia, retorna NULL
        return NULL;

    // Retorna o valor do campo 'elemento' do nó no topo da pilha
    return pilha->topo->elemento;
}

Expressao criar_expressao()
{
    // Cria uma nova estrutura de expressão e inicializa seu campo 'expressao' como uma string vazia
    Expressao expressao;
    expressao.expressao[0] = '\0';
    return expressao;
}

void adicionar_elemento(Expressao *expressao, const char *elemento)
{
    // Adiciona um elemento à expressão, concatenando um espaço em branco e o valor do elemento na string 'expressao->expressao'
    strcat(expressao->expressao, " ");
    strcat(expressao->expressao, elemento);
}

void imprimir_expressao(const Expressao *expressao)
{
    // Imprime a expressão armazenada na estrutura de expressão
    printf("%s\n", expressao->expressao);
}
```

Figura 3: func.c(Funções da TDA)

```c
void apagar_expressao(Expressao *expressao)
{
    // Limpa a expressão, definindo seu campo 'expressao' como uma string vazia
    expressao->expressao[0] = '\0';
}

int isOperador(char c)
{
    // Verifica se o caractere 'c' é um operador (+, -, * ou /)
    // Retorna 1 se for um operador e 0 caso contrário
    return (c == '+' || c == '-' || c == '*' || c == '/');
}

int getPrioridade(char c)
{
    // Retorna a prioridade do operador representado pelo caractere 'c'
    // Os operadores + e - têm prioridade 1, enquanto * e / têm prioridade 2
    // Qualquer outro caractere retorna prioridade 0
    if (c == '+' || c == '-')
        return 1;
    else if (c == '*' || c == '/')
        return 2;
    return 0;
}

void inverter_string(char *str)
{
    // Inverte a ordem dos caracteres na string 'str'
    int tamanho = strlen(str);
    for (int i = 0; i < tamanho / 2; i++)
    {
        char temp = str[i];
        str[i] = str[tamanho - 1 - i];
        str[tamanho - 1 - i] = temp;
    }
}
```

Figura 4: func.c(Funções da TDA)

```c
void converter_prefixa_para_posfixa(const Expressao *expressao_prefixa, Expressao *expressao_posfixa)
{
    // Cria uma pilha vazia
    Pilha *pilha = criar_pilha();

    char *expressao_prefixa_temp = strdup(expressao_prefixa->expressao); // Cria uma cópia temporária da expressão prefixa
    char *termo = strtok(expressao_prefixa_temp, " ");                   // Divide a expressão em termos separados por espaços

    while (termo != NULL)
    {
        if (isOperador(termo[0])) // Se o termo for um operador
        {
            while (!pilha_vazia(pilha) && isOperador(topo_pilha(pilha)[0]) && getPrioridade(topo_pilha(pilha)[0]) >= getPrioridade(termo[0]))
            {
                // Enquanto houver operadores no topo da pilha com prioridade maior ou igual ao termo atual
                adicionar_elemento(expressao_posfixa, topo_pilha(pilha)); // Adiciona o operador no topo da pilha à expressão posfixa
                desempilhar(pilha);                                       // Remove o operador do topo da pilha
            }
            // Adiciona o termo atual à pilha
            empilhar(pilha, termo);
        }
        else // Se o termo for um número, adiciona o número à expressão posfixa
            adicionar_elemento(expressao_posfixa, termo);

        // Avança para o próximo termo
        termo = strtok(NULL, " ");
    }

    while (!pilha_vazia(pilha))
    {
        adicionar_elemento(expressao_posfixa, topo_pilha(pilha)); // Adiciona os operadores restantes da pilha à expressão posfixa
        desempilhar(pilha);                                       // Remove os operadores da pilha
    }

    apagar_pilha(pilha);          // Libera a memória alocada para a pilha
    free(expressao_prefixa_temp); // Libera a memória alocada para a expressão prefixa temporária
}
```

Figura 5: func.c(Funções da TDA)

```c
void converter_posfixa_para_prefixa(const Expressao *expressao_posfixa, Expressao *expressao_prefixa)
{
    // Cria uma pilha para auxiliar na conversão
    Pilha *pilha = criar_pilha();

    // Cria uma cópia temporária da expressão posfixa
    char *expressao_posfixa_temp = strdup(expressao_posfixa->expressao);

    // Divide a expressão em termos separados por espaço
    char *termo = strtok(expressao_posfixa_temp, " ");

    while (termo != NULL)
    {
        if (isOperador(termo[0]))
        {
            // Se o termo for um operador, verifica a prioridade com o operador no topo da pilha
            while (!pilha_vazia(pilha) && isOperador(topo_pilha(pilha)[0]) && getPrioridade(topo_pilha(pilha)[0]) >= getPrioridade(termo[0]))
            {
                // Enquanto a prioridade do operador no topo da pilha for maior ou igual à do termo atual, desempilha e adiciona o operador à expressão prefixa
                adicionar_elemento(expressao_prefixa, topo_pilha(pilha));
                desempilhar(pilha);
            }
            // Empilha o operador atual
            empilhar(pilha, termo);
        }
        else
        {
            // Se o termo for um número ou operando, adiciona-o diretamente à expressão prefixa
            adicionar_elemento(expressao_prefixa, termo);
        }
        // Obtém o próximo termo da expressão posfixa
        termo = strtok(NULL, " ");
    }

    // Desempilha os operadores restantes na pilha e adiciona-os à expressão prefixa
    while (!pilha_vazia(pilha))
    {
        adicionar_elemento(expressao_prefixa, topo_pilha(pilha));
        desempilhar(pilha);
    }

    // Inverte a expressão prefixa para obter a notação correta
    inverter_string(expressao_prefixa->expressao);

    // Libera a memória alocada e apaga a pilha
    free(expressao_posfixa_temp);
    apagar_pilha(pilha);
}
```

Figura 6: func.c(Funções da TDA)

```c
float resultado_expressao(const Expressao *expressao)
{
    // Cria uma pilha para auxiliar no cálculo
    char operadores[] = "+-*/";

    Pilha *pilha = criar_pilha();
    // Cria uma cópia temporária da expressão
    char *expressao_temp = strdup(expressao->expressao);

    // Divide a expressão em termos separados por espaço
    char *termo = strtok(expressao_temp, " ");
    while (termo != NULL)···

    // O resultado final é o valor restante no topo da pilha
    float resultado_final = atof(topo_pilha(pilha));

    // Libera a memória alocada e apaga a pilha
    free(expressao_temp);
    apagar_pilha(pilha);

    return resultado_final;
}
```

Figura 7: func.c(Funções da TDA)

```c
while (termo != NULL)
{
    if (strchr(operadores, termo[0]) == NULL)
    {
        // Se o termo não for um operador, empilha o operando na pilha
        empilhar(pilha, termo);
    }
    else
    {
        // Se o termo for um operador, desempilha os dois operandos anteriores da pilha

        float operando2 = atof(topo_pilha(pilha));
        desempilhar(pilha);

        float operando1 = atof(topo_pilha(pilha));
        desempilhar(pilha);

        float resultado;
        switch (termo[0])
        {
        case '+':
            resultado = operando1 + operando2;
            break;
        case '-':
            resultado = operando1 - operando2;
            break;
        case '*':
            resultado = operando1 * operando2;
            break;
        case '/':
            resultado = operando1 / operando2;
            break;
        default:
            resultado = 0.0;
            break;
        }

        // Converte o resultado em uma string e empilha na pilha
        char str_resultado[20];
        sprintf(str_resultado, "%.2f", resultado);
        empilhar(pilha, str_resultado);
    }
    // Obtém o próximo termo da expressão
    termo = strtok(NULL, " ");
}
```

Figura 8: func.c(Dentro da função while)

```c
#include <stdio.h>
#include "main.h"

int main()
{
    printf("[Criando expressao posfixa]: ");
    Expressao expressao = criar_expressao();
    adicionar_elemento(&expressao, "5");
    adicionar_elemento(&expressao, "3");
    adicionar_elemento(&expressao, "*");

    imprimir_expressao(&expressao);

    float resultado = resultado_expressao(&expressao);
    printf("Resultado: %.2f\n", resultado);

    printf("\nRealizando conversao para prefixa... \n");
    Expressao expressao_prefixa = criar_expressao();
    converter_posfixa_para_prefixa(&expressao, &expressao_prefixa);
    imprimir_expressao(&expressao_prefixa);

    printf("Convertendo de volta para posfixa... \n");
    Expressao expressao_posfixa = criar_expressao();
    converter_prefixa_para_posfixa(&expressao_prefixa, &expressao_posfixa);
    imprimir_expressao(&expressao_posfixa);

    apagar_expressao(&expressao);
    apagar_expressao(&expressao_prefixa);
    apagar_expressao(&expressao_posfixa);

    return 0;
}
```

Figura 9: main.c(Atribuição dos valores)