Trabalho 1 de Estrutura de Dados

Ricardo P. da Silva Filho – 19/0054042
Adriano

Introdução:

Nas escolas, quando aprendemos o básico da matemática, somos introduzidos à expressões algébricas, e a lógica de ordem das operações na forma infixa (exemplo: 3+4=7). Porém, conforme o número de operandos, operadores, e separadores que formam a expressão aumentam, a mesma se torna complexa e a prioridade das operações mais difícil de implementar.

Para solucionar este problema, utilizaremos expressões algébricas na forma pós-fixa (exemplo: 3 4 + = 7). Esta forma elimina a necessidade do uso de separadores para definir a prioridade das operações porque a prioridade já esta expressa na ordem de escrita.

Este trabalho tem como objetivo implementar uma calculadora de expressões algébricas. O programa deve receber como entrada uma expressão na forma infixa, analisar se a expressão é valida, transformá-la para a forma pós-fixa e, por fim, calcular o resultado.

Implementação:

Para facilitar o desenvolvimento do algoritmo e do projeto como um todo, foi utilizado o git. Assim, foi possível modularizar as etapas de desenvolvimento do código fonte, além do controle de versões do trabalho, bem como quanto cada integrante do grupo contribuiu para sua conclusão.

No programa descrito, foi usada a estrutura de dados chamada de pilha. A estrutura da pilha contem um ponteiro para o elemento no seu topo e é formada por uma lista interligada que contem diversos elementos. Cada elemento possui um ponteiro para o próximo da lista e duas variáveis chamadas de “dadoc” e “dadof” para dados em formato char e dados em formato float respectivamente.

```
//structs
typedef struct elemento{
```

```
   char dadoc;
   float dadof;
   struct elemento * proximo;
}t_elemento;
----------------------------------------
typedef struct pilha{
   t_elemento * topo;
}t_pilha;
```

A pilha é uma estrutura de dados que ao receber um elemento o mesmo é armazenado no topo e ao remover um elemento, o elemento no topo é removido, ou seja, o primeiro elemento a entrar é o último a sair. Para essas ações nós implementamos as funções “empilhac” e “desempilhac” (para variáveis do tipo char) e as funções “empilhaf” e “desempilhaf” (para variáveis do tipo float).

Para empilhar, um elemento é inserido no início da pilha, modificando assim o endereço do primeiro elemento da estrutura e armazenado o endereço do elemento inicial anterior como o próximo deste novo elemento. Quando se desempilha, o endereço inicial da pilha torna-se o do segundo elemento e o dado do elemento removido é retornado pela função.

```
void empilhaC(char dadoc, t_pilha * p){ //empilha um char
   t_elemento * novoElemento= (t_elemento *) malloc(sizeof(t_elemento));
   novoElemento->dadoc=dadoc;
   if(p->topo==NULL){
      p->topo=novoElemento;
      p->topo->proximo=NULL;
   }
   else{
      novoElemento->proximo=p->topo;
      p->topo=novoElemento;
   }
```

```
char desempilhaC(t_pilha * p){ //desempilha um char
   char retorno;
   t_elemento * removido;
   if(p->topo== NULL){
      return -1;
   }
   else{
      retorno=p->topo->dadoc;
      removido=p->topo;
      p->topo=p->topo->proximo;
```

```
    }
    free(removido);
    return retorno;
}
```

o mesmo se aplica para as funções que trabalham com variáveis do tipo float

também implementamos as funções “criaPilha” que inicia uma pilha vazia, e “estaVazia” que lê o enderenço do primeiro elemento da pilha passada como parâmetro e verifica se ele existe ou não. Se não existir, a pilha está vazia.
s

```
t_pilha * criaPilha(){ //inicia uma pilha vazia
    t_pilha * p =(t_pilha * )malloc(sizeof(t_pilha));
    p->topo=NULL;
    return p;
}
-----------------------------------------------------------------------------------------------------
int estaVazia(t_pilha * p){ //retorna 0 para vazia e 1 para nao vazia
    if(p->topo==NULL){
        return 0;
    }
    else{
        return 1;
    }
}
```

A validação da expressão é dada através da função “ehValida” que percorre a expressão e checa se para cada separador de abertura “(” existe um separador de fechamento “)” e também se o número de operadores é coerente ao número de operandos. Para esta, e algumas outras funções, utilizamos os códigos numéricos da tabela ASCII.

```
int ehValida(t_pilha * p, char exp[]){ // retorna 1 para expressao valida e 0 para expressao
invalida
    int caracter,i=0,r;
    while(exp[i] != '\0'){
        caracter=exp[i];
        if(caracter>=40 && caracter<=57 && caracter!=46 && caracter!= 44){
            if(caracter==40){ // (
                empilhaC('(',p);
            }else if(caracter == 41){ // )
                r=desempilhaC(p);
                if(r==-1){
```

```
            return 0;
          }
        }
      }else{
        return 0;
      }
      i++;
    }
    caracter=exp[0];
    if(caracter>=42 && caracter<=45)return 0;
    caracter=exp[i-1];
    if(caracter>=42 && caracter<=45){
      return 0;
    }
    r=estaVazia(p);
    if(r==1){
      return 0;
    }else{
      return 1;
    }
}
```

Para a conversão de infixa para pós-fixa temos a funcao “posFixa”. Esta percorre a expressao dada, e utilizando novamente os códigos numéricos da tabela ASCII, identifica se o char é operando, operador ou separador. Ao encontrar um separador de abertura, esse é empilhado. Ao encontrar um separador de fechamento, o separador de abertura é desempilhado. Ao encontrar um operador, alem de ser empilhado, o mesmo passa por uma análise de prioridade. Por exemplo, em uma equação algébrica a multiplicação e a divisão tem prioridade igual entre si e maior em relação a soma e subtração. Assim o algorítimo converte e organiza a expressa-o na forma pós-fixa de forma com que esta respeite a ordem das operações da expressão original.

```
void posFixa(char exp[]){
    t_pilha * pilha=criaPilha();
    int i=0, caracter, j=0, prio_p, prio, p_vazia;
    char aux_pos;
    while(exp[i] !='\0'){
      caracter=exp[i];
      if(caracter>=48 && caracter<=57){ // operandos
        posfixa[j]=exp[i];
        j++;
      }
      else{
        if(caracter==41){ // )
          aux_pos=desempilhaC(pilha);
```

```
            while(aux_pos!='('){
                posfixa[j]=aux_pos;
                j++;
              aux_pos=desempilhaC(pilha);
            }
          }
          else{
            if(caracter==40){ // (
              empilhaC(exp[i], pilha);
            }
            else{ // sinal
              do{
                  p_vazia=estaVazia(pilha);
                  if(p_vazia==1){
                    prio_p=prioridade(pilha->topo->dadoc);
                    prio=prioridade(exp[i]);
                    if(prio_p>=prio){
                      aux_pos=desempilhaC(pilha);
                      posfixa[j]=aux_pos;
                      j++;
                    }
                  }
                  p_vazia=estaVazia(pilha);
                  if(p_vazia==1){
                    prio_p=prioridade(pilha->topo->dadoc);
                    prio=prioridade(exp[i]);
                  }

}while(p_vazia==1 && prio_p>=prio);
              empilhaC(exp[i],pilha);
            }
          }
        }
        i++;
    }
    p_vazia=estaVazia(pilha);
    while(p_vazia == 1){ // desmpilhaos sinais
      aux_pos=desempilhaC(pilha);
      posfixa[j]=aux_pos;
      j++;
      p_vazia=estaVazia(pilha);
    }
    posfixa[j]='\0';
    resolve();
}
```

Por fim, temos a função “resolve” que empilha os operandos, e ao desempilhar

dois por vez os combinam com o próximo operador da expressão pós-fixa e faz o calculo de uma variável do tipo float. Ao fim do processo o resultado é exibido na tela.

```
void resolve(){
   t_pilha * pilha=criaPilha();
   float t1,t2,aux_vet,resul;
   int i=0,j=0,caracter,count;
   while(posfixa[i] != '\0'){
      caracter=posfixa[i];
      if(caracter>=48 && caracter<=57){
         aux_vet=vet_num[j];
         empilhaF(aux_vet,pilha);
         count=contDigitos(aux_vet);
         i=count+i;
         j++;
      }

else{
         t2=desempilhaF(pilha);
         t1=desempilhaF(pilha);
         resul=calcular(t1,t2,posfixa[i]);
         empilhaF(resul,pilha);
         i++;
      }
   }
   resul=desempilhaF(pilha);
   printf("Resultado = %.1f \n", resul);
}
```

Estudo de Complexidade:

Função	Análise
criaPIlha	3
estavazia	2
EmpilhaF	5
desemplhaF	5
empilhaC	5
desempilhaC	5
ehValida	$14n+7$
prioridade	2
guardarNumeros	$11n^2+2n+2$
posFixa	$6n^2+39n+8$

contDigitos	$3n+4$
calcular	3
resolve	$3n^2+13n+7$
main	$17n^3+55n^2+21n+4$

Total: $O(n^3)$

Testes Executados

Para a verificação do código, foram realizados os seguintes testes:

1.
1+2*2-4/2 = 3 – Este teste checa se o código obedece a ordem de preferência das operações.
2.
(1 + 2) ∗ 3) = expressão inválida – Este teste checa se o código identifica expressões inválidas
3.
(40+80)*(60-50)/((30-20)*(20+20)) = 3 – Este último teste checa se o algoritmo consegue fazer
cálculos com parênteses e números de mais de um algarismo.

Conclusão:

O grupo acredita que este trabalho englobou boa parte da matéria dada em aula e serviu como um excelente método de revisão e aprendizado prático. O trabalho se prova mais desafiador na parte lógica, os algorítimos de conversão para a forma pós-fixa e de verificação da validade da expressão se destacaram como os mais difíceis para implementar. Porém o desafio nos ajudou a desenvolver ideias e ferramentas que podem nos servir em todas as áreas da computação. Procuramos seguir as instruções dadas na proposta e acreditamos termos feito um bom trabalho.

Bibliografia:

- Site do professor Alchieri – https://cic.unb.br/~alchieri/disciplinas/graduacao/ed/ed.html
- CJDinfo – http://www.cjdinfo.com.br/utilitario-tabela-caracteres
- Tutorials Point – https://www.tutorialspoint.com/cprogramming/