Relatório Projeto 1 - INF1010

Alunos:
João Ricardo Malta - 2112714
Vinícius Machado - 2111343

Enunciado:
1) Gere duas listas encadeadas, ordenadas, contendo 10 números inteiros, aleatórios, pertencentes ao intervalo [1,100];
Exiba as listas geradas;
Crie uma nova lista, intercalando os valores das listas geradas, sem repetição, mantendo a ordenação;
Exiba a lista intercalada.
Exemplo: L1 = { 1, 4, 7, 8 } /* lista gerada */
L2 = { 2, 4, 8, 10, 11, 12 } /* lista gerada */
L3 = { 1, 2, 4, 7, 8, 10, 11, 12 } /* lista intercalada */

2) Implemente um algoritmo com lista encadeada para calcular expressões aritméticas em notação pósfixa. Teste o algoritmo no cálculo das expressões abaixo e imprima o resultado de cada expressão:
a. 3 2 7 * + 1 -
b. 2 3 4 1 + * -
c. 5 1 2 + 4 * + 3 -
d. 7 5 6 2 / 8 * + 3 + *
e. 2 1 / 4 2 * + 6 5 - 8 2 / + *

O Enunciado contém 2 questões que buscam gerar dois algoritmos diferentes:
1 para concatenar listas encadeadas ordenadas em uma nova lista ordenada com os elementos presentes, sem repetir elementos iguais
1 para ler operações na notação pós fixada e retornar o resultado

1)
Para o primeiro exercício foram implementadas as funções e structs:

Struct lista
Representa a lista encadeada que armazena os números gerados aleatoriamente. Ele possui um elemento de número e um ponteiro para o próximo elemento da lista

lst_cria()
Esta função cria uma lista encadeada vazia e retorna um ponteiro para o início da lista.

void lst_imprime(Lista *lst)

Essa função recebe uma lista encadeada e cria um ponteiro para Lista, então ele faz um loop for fazendo com que o ponteiro receba o valor de lst e vai passando para o próximo nó de P enquanto ele for diferente de Null, e a cada repetição ele imprime o número do nó.

lst_insere(Lista ** l, int n)
Esta função insere um novo elemento em uma lista encadeada ordenada de forma crescente, mantendo a ordem. Caso o elemento já exista na lista, ele não será inserido novamente.
Recebe como parâmetro um ponteiro para ponteiro de lista
Primeiro ela cria uma nova lista e aloca espaço de memória para essa lista, faz uma verificação para garantir que a alocação de memória esteja correta e então ela faz com que o elemento do começo da lista seja o número passado.

Em seguida ela verifica se a lista passada está vazia ou se o elemento passado passado é menor do que o elemento da posição atual da lista, se for o caso, ele faz com que o próximo elemento de novo receba essa lista e a lista passada como parâmetro recebe o novo.
Se a primeira condição não for atendida, ele cria uma lista auxiliar e faz ela receber o conteúdo da lista passada como parâmetro. Então ela faz uma loop "while" que é atendido enquanto a lista aux não tiver acabado **E** enquanto o número do próximo elemento da lista for menor do que o número passado como parâmetro, enquanto essa condição for atendida, ele vai percorrer a lista auxiliar, elemento a elemento. Após isso, ele faz com que o próximo elemento da nova lista receba o próximo elemento da lista auxiliar e o próximo elemento da lista auxiliar receba a nova lista.

concatenaLista(*Lista lista1, *Lista lista2)
Esta função concatena duas listas encadeadas ordenadas (list1 e list2) em uma nova lista ordenada, garantindo que elementos repetidos sejam incluídos apenas uma vez. Ela recebe como parâmetro dois ponteiros para duas listas.
A função começa inicializando uma lista que vai ser retornada para o usuário com o resultado da concatenação. O programa irá rodar enquanto ambas as listas forem diferentes de NULL. Primeiro ele faz uma verificação para saber se a primeira lista não está vazia (diferente de NULL) e se verifica se ou a lista dois está vazia ou se o número da lista 1 é menor que o número da lista 2. Se a condição for atendida, ele verifica se a lista de retorno está vazia ou se o número da lista de retorno na posição atual é diferente do número da lista 1 na posição atual. Se ambas as condições forem atendidas, ele insere o número da lista 1 na lista de retorno e avança de posição na lista 1.

Caso a primeira condição não seja atendida, ele faz uma verificação para saber o oposto, se a lista dois não está vazia e se ou a lista um está vazia ou se o número da lista 2 é menor que o número da lista 1. Se a condição for atendida, ele verifica se a lista de retorno está vazia ou se o número da lista de retorno na posição atual é diferente do número da lista 2 na posição atual. Se ambas as condições forem atendidas, ele insere o número da lista 2 na lista de retorno e avança de posição na lista 2.

Se nenhuma das duas condições forem atendidas, é porque ambas possuem o mesmo elemento, então ele verifica se a lista de resultado está vazia ou se o elemento da posição atual da lista de retorno é diferente do elemento atual da lista 1 (que é igual ao da lista 2), se essa condição for cumprida, ele insere o elemento da lista 1 na lista 2. Após isso, ele avança de posição em ambas as listas
Após percorrer todas as listas, ele retorna a lista resultado

Na Main
São criadas três listas e depois elas são inicializadas com valores nulos. Pouco antes de inicializar é definido uma "seed" para a geração de números aleatórios, baseado no horário do computador (o tempo medido é utilizado na conta, então ele sempre muda).
Em seguida, é feito um loop for que se repete 10x, gerando 2 números inteiros aleatórios (n1 e n2) no intervalo de 0 a 100, e então n1 é inserido na lista 1 e n2 na lista 2.
Depois são impressos os elementos das listas 1 e 2.
por último, utilizamos a função de concatenar lista e fazemos a lista 3 receber o resultado dessa função, passando as listas 1 e 2, e então imprimindo os elementos da lista 3

2)
Para o segundo exercício foram necessárias a utilização de 2 structs, a struct elemento que contém um valor do tipo float e um ponteiro para o próximo elemento, e a struct pilha que representa a pilha em si e possui um único campo ponteiro que aponta para o elemento no topo da pilha. Para o funcionamento do código também implementamos as funções descritas abaixo:

Pilha *pilhaCria(void)
Esta função cria uma pilha vazia e retorna um ponteiro para a pilha recém-criada.

int pilhaVazia (Pilha *p)
Esta função verifica se a pilha está vazia. Ela recebe como parâmetro um ponteiro para a pilha e retorna 1 se a pilha estiver vazia ou 0 caso contrário.

void pilhaPush (Pilha *p, float valor)
Esta função insere um elemento (com valor `valor`) no topo da pilha. Ela recebe como parâmetros um ponteiro para a pilha e o valor do elemento a ser inserido.

float pilhaPop(Pilha *p)
Esta função remove o elemento do topo da pilha e retorna seu valor. Ela recebe como parâmetro um ponteiro para a pilha.

void pilhaLibera (Pilha *p)
Esta função libera a memória alocada para todos os elementos da pilha e a memória alocada para a própria pilha. Ela recebe como parâmetro um ponteiro para a pilha.

double calculadora(char operacao, double num1, double num2)
Esta função implementa uma calculadora simples que realiza operações aritméticas e exponenciação entre dois números (`num1` e `num2`) com base no operador fornecido (`operacao`). Ela retorna o resultado da operação como um número de ponto flutuante (double). Os operadores suportados são:

- `+`: Adição
- `-`: Subtração
- `*`: Multiplicação
- `/`: Divisão
- `^`: Exponenciação

Se a operação não for reconhecida ou se ocorrer uma divisão por zero, a função exibe uma mensagem de erro e encerra o programa.

Para o funcionamento da main, a expressão pósfixa é fornecida como uma sequência de tokens separados por espaços, primeiro criamos uma pilha vazia usando a função pilhaCria e armazenamos um ponteiro para ela na variável p.
Em seguida, declaramos um vetor de caracteres operação com tamanho máximo N para armazenar a expressão fornecida pelo usuário. Armazenamos essa informação no vetor utilizando a função scanf e usamos também a função strtok para dividir a expressão em tokens separados por espaços.

O próximo passo foi criar um um loop para processar cada token da expressão até que não haja mais tokens e verificamos o que o token é , se o for um dígito ele é convertido em float pela função atof e é inserido na pilha usando a função pilhaPush, se o token não for um dígito, verificamos se ele é um operador válido (+, -, *, /, ^) e se for, removemos os dois números superiores da pilha usando pilhaPop, realizamos a operação com esses números usando a função calculadora e inserimos o resultado de volta na pilha.

Após o loop, verificamos se a pilha está vazia, se não estiver vazia, significa que a expressão foi avaliada com sucesso, e o resultado final dela está no topo da pilha. Esse resultado é retirado da pilha usando `pilhaPop` e é exibido na tela. Por fim liberamos a memória e encerramos o programa

Abaixo estão os testes os resultados de cada teste:

Teste lista ordenada:

Lista 1

```
> make -s
> ./main

Lista 1
num = 18
num = 28
num = 45
num = 52
num = 70
num = 72
num = 81
num = 83
num = 88
num = 99
```

Lista 2

```
Lista 2
num = 18
num = 18
num = 22
num = 27
num = 28
num = 47
num = 49
num = 76
num = 85
num = 97
```

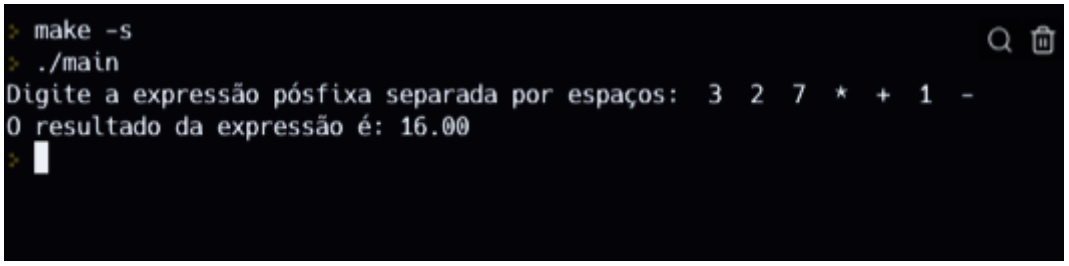
Lista 3 ordenada

```
Lista 3
num = 18
num = 22
num = 27
num = 28
num = 45
num = 47
num = 49
num = 52
num = 70
num = 72
num = 76
num = 81
num = 83
num = 85
num = 88
num = 97
num = 99
```

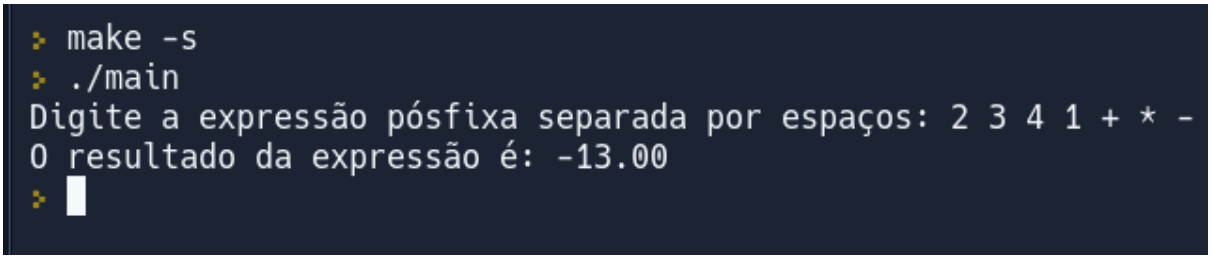
Teste calculadora Pósfixa:

a. 3 2 7 * + 1 -
b. 2 3 4 1 + * -
c. 5 1 2 + 4 * + 3 -
d. 7 5 6 2 / 8 * + 3 + *
e. 2 1 / 4 2 * + 6 5 - 8 2 / + *

Teste A



Teste B



Teste C

```
> make -s
> ./main
Digite a expressão pósfixa separada por espaços:  5  1  2  +  4  *  +  3  -
O resultado da expressão é: 14.00
>
```

Teste D

```
> make -s
> ./main
Digite a expressão pósfixa separada por espaços:  7  5  6  2  /  8  *  +  3  +  *
O resultado da expressão é: 224.00
>
```

Teste E

```
> make -s
> ./main
Digite a expressão pósfixa separada por espaços: 2 1 / 4 2 * +
6 5 - 8 2 / + *
O resultado da expressão é: 50.00
>
```