Segundo Trabalho Prático

Estruturas de Dados 1

Prof. Paulo Henrique Ribeiro Gabriel

O objetivo deste trabalho é implementar operações recursivas para percorrer lista encadeadas. Para isso, é necessário adaptar o TAD desenvolvido em aula e estendê-lo, criando novas funcionalidades. Implemente essa atividade sem compartilhar código com colegas e nem pesquisar na internet.

Listas e Recursão

Considere uma lista encadeada *simples*, ou seja, não é uma lista circular nem duplamente encadeada. Além disso, essa lista é acessada por um único ponteiro para o primeiro nó. Não há, portanto, ponteiros para o último nó, nem para qualquer outro nó intermediário, apenas para o primeiro. Essa lista armazenará apenas números *inteiros*.

Para simplificar a notação, vamos dizer aqui que o primeiro elemento da lista é a Cabeça e os demais formam a Cauda da lista. (Essa notação é mais usual em linguagens como Prolog ou Haskell, mas podemos adotá-la sem problemas.)

O objetivo desse trabalho é inserir um conjunto de valores inteiros, fornecidos via entrada padrão, em uma lista encadeada. Em seguida, deve-se executar diversas operações sobre os valores dessa lista. Todas as operações estão descritas a seguir e devem ser implementadas de **maneira recursiva**.

1. Operação *pertence*: dada uma lista $\mathcal{L}$ e um valor X, verificar se esse X pertence a $\mathcal{L}$. Imprimir 1 caso pertença, 0 caso não pertença. **Dica:** Se o X é o primeiro elemento da lista, então X pertence a $\mathcal{L}$; caso contrário, X pode pertencer à Cauda da lista.

2. Operação *último*: dada uma lista $\mathcal{L}$, encontrar o último elemento dela. **Dica:** se a lista tem apenas um elemento, esse elemento é o último; caso contrário, o último elemento da lista é o último elemento de sua Cauda.

3. Operação *soma*: dada uma lista $\mathcal{L}$, somar todos os seus elementos. **Dica:** se a lista é vazia, a soma é zero; caso contrário, a soma dos elementos de $\mathcal{L}$ é a soma dos elementos da Cauda mais o elemento da Cabeça.

4. Operação *soma ímpares*: dada uma lista $\mathcal{L}$, somar todos os seus elementos ímpares. **Observação:** números negativos também podem ser pares ou ímpares.

5. Operação *n-ésimo*: dada uma lista $\mathcal{L}$ e um inteiro N, encontrar o n-ésimo elemento de $\mathcal{L}$. **Dica:** o primeiro elemento da lista é a Cabeça de $\mathcal{L}$; o n-ésimo elemento de uma lista é o $(n-1)$-ésimo elemento de sua Cauda.

6. Operação *comprimento*: dada uma lista $\mathcal{L}$, contar o número de elementos de $\mathcal{L}$. **Dica:** o número de elementos de uma lista vazia é zero; o número de elementos de uma lista não vazia é o número de elementos da Cauda mais um.

Note que todas essas operações podem sem implementadas de diversas maneiras. Porém, para esse trabalho, elas devem ser implementas com recursão, seguindo as dicas dadas nesse documento. Implementações não recursivas serão **desconsideradas**.

Formato da Entrada

A entrada do programa será composta por duas linhas. Na primeira, temos os valores de X e de N que são usados como entrada na das operações *pertence* e *n-ésimo*, respectivamente (ver descrição). Já na segunda linha temos os valores da lista em si. Note que a ordem em que esses valores são mostrados a seguir deve ser a mesma em que eles aparecerão na lista. Um exemplo de entrada é o seguinte:

```
6 4
3 10  1 -8 -5  6  9 -1 -1  9
```

Formato da Saída

A saída será composta pelos resultados de cada uma das seis operações listadas anteriormente, sendo um resultado por linha. No caso da entrada acima, o resultado será:

```
1
9
23
15
-8
10
```

Critérios de Avaliação

O projeto será avaliado principalmente levando em consideração:

1. Processamento correto das entradas e saídas do programa;
2. Realização das tarefas descritas;
3. Bom uso das técnicas de programação;
4. Boa endentação e uso de comentários no código;
5. Boa estruturação e modularização do código.

Observações

1. O trabalho deve ser feito em **grupos de duas ou três pessoas** e em Linguagem C. Não serão aceitos trabalhos individuais nem desenvolvidos por grupos com mais de três discentes.

2. Não deverá ser utilizada qualquer variável global nem bibliotecas com funções prontas (a não ser aquelas para entrada, saída e alocação dinâmica de memória).

3. Todas as submissões são checadas para evitar plágio; portanto, evite problemas e implemente o seu próprio código.

4. Ainda sobre plágio, a detecção de cópia de parte ou de todo código-fonte, de qualquer origem, implicará reprovação direta no trabalho. Compartilhem ideias, modos de resolver o problema, mas não o código. Qualquer dúvida entrem em contato com o professor.

5. Comente o seu código com uma explicação rápida do que cada função ou trecho importante de código faz (ou deveria fazer). Os comentários e a boa modularização do código serão checados e valem nota.

6. Todas as funções devem ser implementadas em um único arquivo `.c`; porém, deve-se ficar atendo ao bom uso de TAD.

7. Entradas e saídas devem ser lidas e escritas a partir dos **dispositivos padrão**, ou seja, use as funções `scanf` e `printf`.

8. Lembre-se de respeitar estritamente o formado de entrada e saída. Uma quebra de linha a mais ou a menos resultará em erro no caso de teste.

9. O professor pode tirar dúvidas sobre o enunciado, ou sobre a lógica por trás do programa. No entanto, ele não olhará código-fonte em busca de erros! Procure o monitor da disciplina nesse caso.

10. Esse trabalho considera o uso de listas encadeadas e de recursão. O desrespeito a qualquer uma dessas especificações resultará em perda de pontos.