Relatório Tarefa 1 - Listas Encadeadas

João Gabriel Peixoto Drumond Lopes

21 de março de 2023

1 Identificação

Grupo: João Gabriel Peixoto Drumond Lopes, matrícula 2210168

2 Objetivo

O trabalho tinha por objetivo a criação de um programa que servisse para gerir pacientes, e, de acordo com a sua ordem de chegada e a prioridade de seu atendimento, para determinar a ordenação deles. Haveria, portanto, uma função de remoção, que significaria atender o paciente em primeiro lugar na fila; uma função de inserção, levando em conta a ordenação; uma função de impressão dos pacientes na lista; e uma função de liberação de memória alocada ao fim da execução do programa, para evitar memory leaks.

3 Estrutura do Programa

O Tipo Abstrato de Dados utilizado para a resolução do problema e para a implementação do programa foi o de lista encadeada simples ordenada, com remoção, inserção, leitura e liberação de memória. Tem comportamento similar ao de uma fila/queue, contudo a ordenação impede que seja de fato First In First Out (mas assumindo que os pacientes cheguem de forma ordenada, o comportamento seria o mesmo)

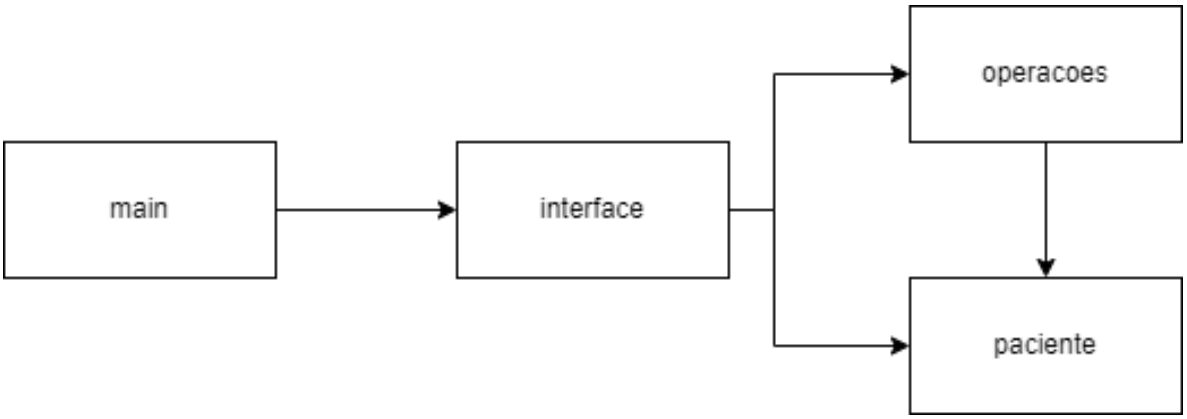

Figura 1: Relações entre os módulos do programa

O programa foi dividido em três módulos, fora a função *main*(), localizada no *main.c*. Os módulos são: *operacoes*, *paciente* e $interface$.

3.1 Operações (operacoes)

O módulo de manipulação do TAD tem quatro funções principais: *cria_lista*() , *chegada*(), *imprime*(), *atende*() e *libera_lista*(). Cria, lê, remove e libera.

- *cria_lista*(): Função simples, somente retorna NULL, e serve para anunciar a criação da lista.

- *chegada*(): Função recebe um ponteiro para o primeiro elemento de uma lista encadadeada de struct paciente*, l, um enum Priority, r, que determina a prioridade do paciente a ser criado, e um int, o, que determina a ordem na qual o paciente chegou.

  Inicialmente, é criado um novo elemento a partir da chamada para a função *cria_paciente*(), com parâmetros enum Priority r e int o, anteriormente citados. Caso não haja alocação de memória do novo nó, o programa é abortado precocemente, retornando NULL e emitindo um erro.

  Caso a lista esteja vazia, o elemento novo será o primeiro (e único) da lista encadeada. Caso contrário, percorre-se a lista até que se encontre outros de mesma prioridade, e de menor ordem de chegada. Caso h seja NULL, o elemento será inserido na primeira posição da lista. Se não, a inserção ocorre como normal, mesmo em caso de ser o último elemento da lista (já que p apontaria para NULL).

  Ao final da execução, retorna o ponteiro para o primeiro elemento da lista encadeada, que pode ter sido atualizado para um novo ou não.

- *imprime*(): A função de impressão é bem simples, e recebe um ponteiro para o primeiro elemento da lista encadeada. Ela é percorrida, então, até chegar em NULL, enquanto imprime a ordem de chegada dos pacientes e sua prioridade, determinada através de um *switch* case com as opções do enum Priority. Caso seja recebido um número que não pertence ao conjunto $\{0, 1, 2\}$, a cor da prioridade é substituída por uma mensagem de aviso.

  Tem tipo de retorno *void* e, portanto, não retorna nada.

- *atende*(): O atendimento, nesse TAD, consiste em atender o primeiro na lista, como uma fila. A função atende faz exatamente isso, removendo o primeiro elemento da lista, e a imprimindo atualizada. Recebe somente um ponteiro para o primeiro elemento da lista encadadeada, retornando uma versão modificada, após liberar a memória do elemento retirado.

- *libera_lista*() Essa função, a última do módulo de operações, não tem função intrínseca à situação presente. Sua não existência não impediria (provavelmente) a boa execução do programa. Contudo, havendo memória alocada no decorrer da execução, é importante que esse alocamento de memória da heap seja liberado, para não correr risco de ocorrerem memory leaks.

  Recebe o ponteiro para o primeiro elemento da lista encadeada e não retorna nada, sendo de tipo de retorno *void*.

3.2 Paciente (paciente)

Esse módulo tem como principal funcionalidade a definição do TAD, um tipo estruturado que possui como elementos um enum Priority, um int, e um ponteiro de struct paciente, que apontaria para o próximo elemento da lista encadeada. Além disso, nesse módulo é definido o enum Priority. Uma função também faz parte desse módulo, *cria_paciente*().

- struct paciente: O TAD principal da solução. Possui três elementos: um enum Priority, que pode assumir valores de 0,1 e 2, e que determina a prioridade do atendimento, Vermelho, Amarelo, Verde, respectivamente; um int o, que determina a ordem em que o paciente chegou; e um ponteiro para um próximo elemento de tipo struct paciente, servindo para a implementação de fato de uma lista encadeada simples.

- enum Priority: Esse tipo de enumeração é utilizado para representar os diferentes níveis de prioridade relatados na descrição do problema. VERMELHO, AMARELO, VERDE, 0,1, 2, respectivamente.

- *cria_paciente*(): Essa função é utilizada para alocação de memória de um struct paciente. Retorna um struct paciente*, e recebe um enum Priority r e um int o (prioridade e ordem). É utilizada para a função de chegada, que insere novos pacientes, por ela criados.

3.3 Interface (interface)

O módulo de interface é o mais objetivo e, ao mesmo tempo, o mais importante para a execução do programa. Contém somente uma função $main\_interface()$, que cria uma interface que permite ao usuário, através de um $scanf()$ e de um switch case, inserir novos pacientes (prioridade inserida pelo teclado, também), atender (remover) pacientes, e finalizar a execução do programa, liberando a memória alocada da lista.

O principal objetivo desse módulo foi retirar da main a função principal, deixando a main mais livre. Fosse talvez, mais interessante, caso houvesse uma generalização maior dessa função de interface, além da criação de um número maior de componentes que pudessem permitir uma reutilização futura desse código.

4 Solução

A solução do problema, como descrito anteriormente na parte da estrutura do programa, foi dividido em três módulos. A execução do programa, então, faz-se da seguinte maneira: na entrada de execução, na função $main()$, há somente uma chamada da função $main\_interface()$, fornecida pelo módulo interface, que terceiriza a execução efetiva do programa.

Inicialmente, é possível ver que esse módulo, devido ao seu protagonismo, dialoga com todos os outros módulos do programa. Como servidor em todos os casos, ele utiliza as funções disponibilizadas no módulo de operações, e o tipo estruturado presente no módulo paciente.

No início da função propriamente dita, é inicializada uma lista encadeada de struct paciente*, com valor inicial NULL, sem memória alocada. Três variáveis são então criadas, duas inicializadas. *opcao* serve para a interatividade do programa, armazenando a escolha de ação do usuário. Inicializada com -1, previne a execução de uma tarefa indesejada, já que um número fora do conjunto $\{0, 1, 2\}$ iria para o caso $default$ do switch case. A variável i é um inteiro, também, um contador para a ordem de chegada dos pacientes. O array de inteiros de tamanho 3, alocado estaticamente, $counters$, será utilizado posteriormente para armazenar o número de pacientes em cada grupo de prioridade.

Ao entrar no loop $do...while()$, o programa rodará sem parar, a não ser que o usuário forneça a opção 2, que aborta a execução. Caso seja fornecido 1, um paciente é atendido; 0, paciente inserido; qualquer outro número, reinicio do loop sem execução de qualquer tarefa. Quando o usuário fornece a opção 0, ele é perguntado o nível de prioridade do paciente (pode ser 0, 1 ou 2,e caso seja qualquer outro número, o $do...while()$ executa até uma inserção correta. Em caso de input aceitável, é adicionado 1 no contador de cada tipo de prioridade, tomando a propria priority como índice, e chamada uma função chegada, com a lista original, a prioridade e a ordem de chegada. Depois disso, a ordem de chegada é incrementada em 1.

Execução da função chegada: primeiramente, é alocado um novo struct paciente*, pela função $cria\_paciente()$ (essa função aloca um struct paciente*, insere os dados fornecidos e bota o ponteiro para o próximo elemento apontando para NULL). Caso seja NULL (erro na alocação), o programa é abortado e uma mensagem de erro é exibida. Caso contrário e se a lista for NULL (portanto, vazia), o novo elemento se torna a (cabeça) da lista, e a nova lista é retornada. Continuando a execução, em caso de lista não vazia, são inicializados dois cursores para percorrer a lista, mantendo um ponteiro para o elemento anterior.

O próximo $while()$ loop, caso encontre uma prioridade igual a inserida, continua procurando somente pelo número de chegada. Os cursores são sempre atualizados. Se o h for NULL, então o elemento deverá ser inserido no início da lista, o que então ocorre. Caso contrário, o elemento é inserido tomando a referência do elemento anterior e do atual (que se torna o próximo) e um ponteiro para a cabeça da lista atualizada é retornada.

Voltando para o switch da $main\_interface()$, caso o usuário escolha a opção 1, o primeiro paciente da lista é atendido pela função atende() e, então removido. Essa função retorna a lista atualizada. Caso a lista seja vazia, é impresso na tela: "Lista vazia". Caso contrário, é tomado o ponteiro para o segundo elemento, e posto na cabeça da lista. Então, a memória desse primeiro elemento é liberada, e é impressa a nova lista logo antes do retorno da função.

Por fim, com a escolha da opção 2, o programa exibe uma mensagem de Finalizando... e libera toda a memória alocada, por uma função padrão de liberação de memória de lista encadeada. Mantém-se sempre referência para o próximo elemento da lista, liberando o elemento atual a cada passada do

loop.

5 Conclusao

O programa parece atender às exigências propostas no enunciado. No entanto, existe uma maior elaboração que poderia existir, embora não faça sentido no contexto, mas são operações de manipução do tipo de lista encadeada, como funções para remover elementos específicos, inserir em pontos específicos da lista, entre outros.

A maior falha do programa encontra-se, provavelmente, na função $main\_interface()$, que realiza muitas funções de forma simultânea, e, embora já realize a chamada de funções de outros módulos, talvez pudesse ter mais disso (como para a leitura e verificação do valor inserido na prioridade).