Relatório Tarefa 1 - Listas Encadeadas

João Gabriel Peixoto Drumond Lopes

March 25, 2023

1 Identificação

Grupo: João Gabriel Peixoto Drumond Lopes, matrícula 2210168

2 Objetivo

O trabalho tinha por objetivo criar um TAD para gerir um sistema de pacientes. Seria possível: inseri-los na lista; removê-los, seja pelo atendimento do primeiro da fila, a desistência de um específico, ou a transferência de um paciente de uma prioridade determinada; imprimir a lista de todos os pacientes e liberar a memória ao finalizar o programa.

3 Estrutura do Programa

O TAD utilizado para a resolução do problema foi uma lista simplesmente encadeada. A implementação foi feita utilizando um tipo estruturado, chamado de *paciente*, que guarda um ponteiro para o próximo elemento da lista. Não foi utilizada uma estrutura que pudesse, por exemplo, armazenar diretamente um ponteiro para o head da lista.

As operações implementadas foram as de inserção, remoção, leitura e destruição (liberação de memória).

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

3.1 Singly linked list (singlinklist)

O módulo do TAD, *singlinklist* tem sete funções: *cria_paciente*(), *cria_lista*() , *chegada*(), *imprime*(), *atende*(), *desiste_ou_transfere*() e *libera_lista*(). Cria, insere, lê, remove e libera. Além disso, possui a definição de dois tipos, *struct paciente* e *enum Priority*.

- *cria_paciente*: A função tem como objetivo a alocação de memória para um novo nó da lista encadeada. Recebe um enum Priority e um int o, e põe esses dados na nova memória alocada. O atributo next é inicializado como NULL, para evitar problemas de não inicialização. Retorna um struct paciente*.
- *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, emitindo um erro e encerrando a execução com código de saída 1.

  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 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.

- $desiste\_ou\_transfere()$: A função de desistir ou transferir é uma função de remoção de um item específico da lista encadeada. Caso a flag *desiste* esteja "ligada", o programa irá, através de um operador ternário, fazer a comparação de acordo com a ordem de chegada do paciente fornecida pela chamada da função. Caso contrário, a comparação ocorrerá com a prioridade do paciente, sendo uma operação de transferência do primeiro de uma determinada prioridade. Existem salvaguardas para evitar problemas como remoção do primeiro item da lista, e o int* *priority_ptr* tem como objetivo permitir que a lista de contagem de quantos pacientes tem em cada prioridade seja atualizada após uma chamada para essa função.

- *libera_lista*() Essa função, a última do módulo de operações, não tem sentido intrínseco à situação presente. Sua não existência não impediria (exceto em raros casos) 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*.

- 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.

3.2 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), remover (atender, transferir, desistir) pacientes, imprimir a lista atual e finalizar a execução do programa, liberando a memória alocada.

Além disso, também realiza a manipulação de um array de 3 elementos, $counter[]$, que permite ao programa exibir quantos pacientes estão presentes no momento de cada prioridade.

O principal objetivo desse módulo foi retirar da main a função principal, deixando-a mais livre. Seria talvez, mais interessante, haver 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 dois módulos. A execução do programa, então, faz-se da seguinte maneira: na entrada de execução, na *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 singlinklist.

No início da função propriamente dita, é inicializada uma lista encadeada de struct paciente*, com valor inicial NULL, sem memória alocada. Cinco variáveis são então criadas, três 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 $\{0, 1, 2, 3, 4, 5\}$ 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. As variáveis *priority_temp* e *priority_ptr* são utilizadas para haver uma referência para a prioridade que deverá ser atualizada no array de contadores.

Ao entrar no loop *do...while*(), o programa rodará sem parar, a não ser que o usuário forneça a opção 5, que aborta a execução. Caso seja fornecido 0, paciente inserido; 1, paciente é atendido; 2, paciente é transferido; 3, paciente desiste; 4, impressão da lista; 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.

No caso das opções 2 e 3, um elemento escolhido é removido da lista. A única diferença encontra-se na flag *desiste*, 0 na opção 2 e 1 na opção 3. A opção 4 permite a impressão de todo o conteúdo da lista encadeada.

Por fim, com a escolha da opção 5, o programa exibe uma mensagem de Finalizando... e libera toda a memória alocada. A função de liberação funciona chamando sempre a função atende. Isso causa a impressão da lista várias vezes, mas com a flag *imprimir*, é possível determinar que essa impressão não ocorra. Ao fim de sua execução, é exibida a mensagem "Memoria liberada".

Em todas as funções de remoção, a lista atualizada é impressa ao fim da execução. Contudo, isso não ocorre nos casos em que a lista não sofre alteração.

O funcionamento do programa foi condizente com os exemplos de operações presentes no enunciado, exibindo o mesmo output.

5 Conclusao

O programa parece atender às exigências propostas no enunciado. Algumas funções de manipulação de lista encadeada não foram necessárias, como inserção em início, final, ponto específico, ou uma função explícita de busca.

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, a realização dessa operação mais vezes (como para a leitura e verificação do valor inserido na prioridade). Além disso, uma função de busca poderia diminuir a complexidade das funções de remoção de elemento específico.