Argumentação de Corretude

Função selecionada: LIS_tpCondRet LIS_EsvaziarLista(LIS_tppLista pLista)

```
LIS_tpCondRet LIS_EsvaziarLista(LIS_tppLista pLista)
{
/* AE */

        tpElemLista * pElem;

        if (pLista == NULL)
        {
                return LIS_CondRetListaInexistente;
        }
        /*AI_1*/


        if (pLista->pOrigemLista == NULL)
        {
                return LIS_CondRetListaVazia;
        }
        /*AI_2*/

        pLista->pElemCorr = pLista->pOrigemLista;
        /* AI_ 3*/


        while (pLista->pElemCorr != NULL)
        {
                pElem = pLista->pElemCorr;
                /* AI_6 */

                pLista->pElemCorr = pLista->pElemCorr->pProx;
                /* AI_7 */

                LiberarElemento(pElem, pLista->ExcluirValor);
        }
        /*AI_4*/

        pLista->pElemCorr = NULL;
        /*AI_5*/

        LimparCabeca(pLista);

        return LIS_CondRetOK;

/* AS */
}
```

Argumentação de Sequência:

AE:

- Valem as assertivas estruturais da lista duplamente encadeada.
- Pode existir ou não uma lista. (pLista != NULL)
- A lista pode estar vazia ou não (pLista->Origem != NULL)

AS:

- A lista não existe, a estrutura possui valor nulo (pLista == NULL), então a função retorna condição de retorno LIS_CondRetListaInexistente.

- A lista existe porém a origem da lista possui valor nulo, ou seja, a lista está vazia (pLista->OrigemLista == NULL), então a função retorna condição de retorno LIS_CondRetListaVazia.

- A lista que antes era populada, teve seus elementos devidamente liberados, ou seja, a saída é a lista vazia e retorna condição de retorno LIS_CondRetOk.

AI_1:
Se a lista não existe (pLista == NULL), a função retorna condição de retorno LIS_CondRetListaInexistente, valendo a AS.

AI_2:
Se a lista já estiver vazia, ou seja, a origem da lista possui valor nulo (pLista->OrigemLista == NULL), a função retorna condição de retorno LIS_CondRetListaVazia, valendo a AS.

AI_3:
A lista existe e não está vazia, então o elemento corrente passa a apontar para a origem da lista (pLista->ElemCorr = pLista ->pOrigemLista)

AI_4:
Se a condição do laço de repetição (while) não foi satisfeita, então o elemento corrente recebe valor nulo (pLista->pElemCorr = NULL), garantindo assim, a liberação do elemento corrente.

AI_5:

O elemento corrente da lista já possui valor nulo (pLista->pElemCorr = NULL), então a função LimparCabeca é chamada para liberar a cabeça da lista, tornando assim, a lista vazia.

Argumentação de Seleção:

AE:

- Valem as assertivas estruturais da lista duplamente encadeada.
- A variável pLista pode ser NULL (nulo), ou seja, a Lista pode não existir.

AS:

Retorna condição de retorno LIS_CondRetListaInexistente do tipo LIS_tpCondRet.

1. AE && (Cond == True) + B => AS
   Pela AE, pLista é igual a NULL. Como (Cond == True), retorna condição de retorno LIS_CondRetListaInexistente do tipo LIS_tpCondRet, valendo assim, a AS.

2. AE && (Cond == False) => AS.
   Para (Cond == False), ou seja, pLista é diferente de NULL, logo nada ocorre, valendo a AS.

Argumentação de Seleção:

AE:

- Valem as assertivas estruturais da lista duplamente encadeada.
- A variável pOrigemLista do tipo estruturado LIS_tppLista pode ser NULL (nulo), ou seja, a Lista pode já estar vazia.

AS: Retorna condição de retorno LIS_CondRetListaVazia do tipo LIS_tpCondRet.

1. AE && (Cond == True) + B => AS

   Pela AE, pOrigemLista do tipo estruturado LIS_tppLista é igual a NULL. Como (Cond == True), retorna condição de retorno LIS_CondRetListaVazia do tipo LIS_tpCondRet, valendo assim, a AS.

2. AE && (Cond == False) => AS.

   Para (Cond == False), ou seja, pOrigemLista do tipo estruturado LIS_tppLista é diferente de NULL, logo nada ocorre, valendo a AS.

Argumentação de Repetição:

AE:

- A lista pLista existe e variável pLista->ElemCorr pode ser diferente de NULL, neste caso, o elemento corrente aponta para o primeiro nó. Caso esteja vazia, pLista->ElemCorr servirá para não entrar na repetição seguinte.

AS:

- A lista pLista com todos seus elementos liberados ou a repetição não será executada.

AINV:

- É válida antes de qualquer iteração – hipótese de indução.
- Existem dois conjuntos a tratar: “elementos a liberar” e “elementos já liberados”.
- pElemCorr aponta para “elementos a liberar”, ou seja, o próximo elemento da lista.

1) AE => AINV:

Pela AE, pElem aponta para o nó corrente da lista – nesse caso, o 1º elemento da lista. Como existem dois conjuntos e a repetição não foi iniciada, todos os elementos estão no conjunto “elementos a liberar” e o conjunto “elementos já liberados” está vazio a princípio, sendo que pElem aponta para um dos elementos a liberar, valendo, assim, AINV.

2) AE && (Cond == False) => AS

Pela AE, pElem aponta para o nó corrente da lista – nesse caso, o 1º elemento da lista. Para (Cond == False), a iteração não pode concluir o 1º ciclo. Então, pLista->ElemCorr é nulo, pois a lista está vazia. Assim, o laço de repetição não será executado (ou executa corretamente 0 iterações) valendo a AS.

3) AE && (Cond == True) + B => AINV

Pela AE, pElem aponta para o nó corrente da lista – nesse caso, o 1º elemento da lista. lista. Como (Cond == True), o nó corrente é diferente de nulo e a primeira iteração é executada corretamente (base da indução). Esse elemento (atual pElem) passa para o conjunto “elementos já liberados”. Em B, pLista->ElemCorr é reposicionado e passa a apontar para o próximo elemento da lista e AINV é válido.

4) AINV && (Cond == True) + B => AINV

A assertiva invariante continua valendo, pois B deve garantir que um elemento passe do conjunto “elementos a liberar” para o conjunto “elementos já liberados” e pLista->ElemCorr seja reposicionado para um novo elemento, e AINV é válido. Assim, acrescenta-se corretamente mais uma iteração.

5) AINV && (Cond == False) => AS

Para que (Cond == False), o nó corrente é nulo (pLista->ElemCorr == NULL). Como a condição é falsa, no último ciclo todos os elementos estarão no conjunto “elementos já liberados”, saindo corretamente após n iterações, valendo então a AS.

6) Término

A lista possui um número finito de elementos, e a cada ciclo o elemento corrente é excluído. Desta forma garante que, no término do laço, todos os elementos foram liberados (passam para o conjunto “elementos já liberados” e a repetição termina num número finito de ciclos.

Argumentação de Sequência (sequência da repetição):

AI_6:

Se (pLista -> ElemCorr != NULL), então entrou no laço de repetição (while) e a variável pElem (elemento a ser excluído no ciclo da repetição) passa a apontar para o elemento corrente da lista (pElem = pLista->pElemCorr).

AI_7:

Se pElem (elemento a ser excluído no ciclo da repetição) passar a apontar para o elemento corrente da lista, então o elemento corrente em si, passa a apontar para o próximo elemento da lista (atualização do elemento corrente) (pLista->pElemCorr = pLista->pElemCorr->pProx). É chamada a função LiberarElemento, onde o elemento contido (elemento corrente) em pElem é liberado.