Listas Dup. Encad. Circ. com NC

Com base no que foi visto implemente a operação recup() que compõem o TAD LISTA_CIR_DUP_ENC_NC.

```
typedef struct nodo
{
    int inf;
    struct nodo * ant;
    struct nodo * prox;
}NODO;
typedef NODO * LISTA_CIR_DUP_ENC_NC;
void cria_lista (LISTA_CIR_DUP_ENC_NC *);
int eh_vazia (LISTA_CIR_DUP_ENC_NC);
int tam (LISTA_CIR_DUP_ENC_NC);
void ins (LISTA_CIR_DUP_ENC_NC, int, int);
int recup (LISTA_CIR_DUP_ENC_NC, int);
void ret (LISTA_CIR_DUP_ENC_NC, int);
void destruir (LISTA_CIR_DUP_ENC_NC);
```

```c
int recup (LISTA_CIR_DUP_ENC_NC l, int k)
{
    if (k < 1 || k > tam(l))
    {
        printf ("\nERRO! Consulta invalida.\n");
        exit (3);
    }
    for (;k>0;k--)
        l=l->prox;
    return (l->inf);
}
```

Listas Dup. Encad. Circ. com NC

Com base no que foi visto implemente a operação ret() que compõem o TAD LISTA_CIR_DUP_ENC_NC.

```
typedef struct nodo
{
    int inf;
    struct nodo * ant;
    struct nodo * prox;
}NODO;
typedef NODO * LISTA_CIR_DUP_ENC_NC;
void cria_lista (LISTA_CIR_DUP_ENC_NC *);
int eh_vazia (LISTA_CIR_DUP_ENC_NC);
int tam (LISTA_CIR_DUP_ENC_NC);
void ins (LISTA_CIR_DUP_ENC_NC, int, int);
int recup (LISTA_CIR_DUP_ENC_NC, int);
void ret (LISTA_CIR_DUP_ENC_NC, int);
void destruir (LISTA_CIR_DUP_ENC_NC);
```

UNIVASF

Listas Dup. Encad. Circ. com NC

Dicas:

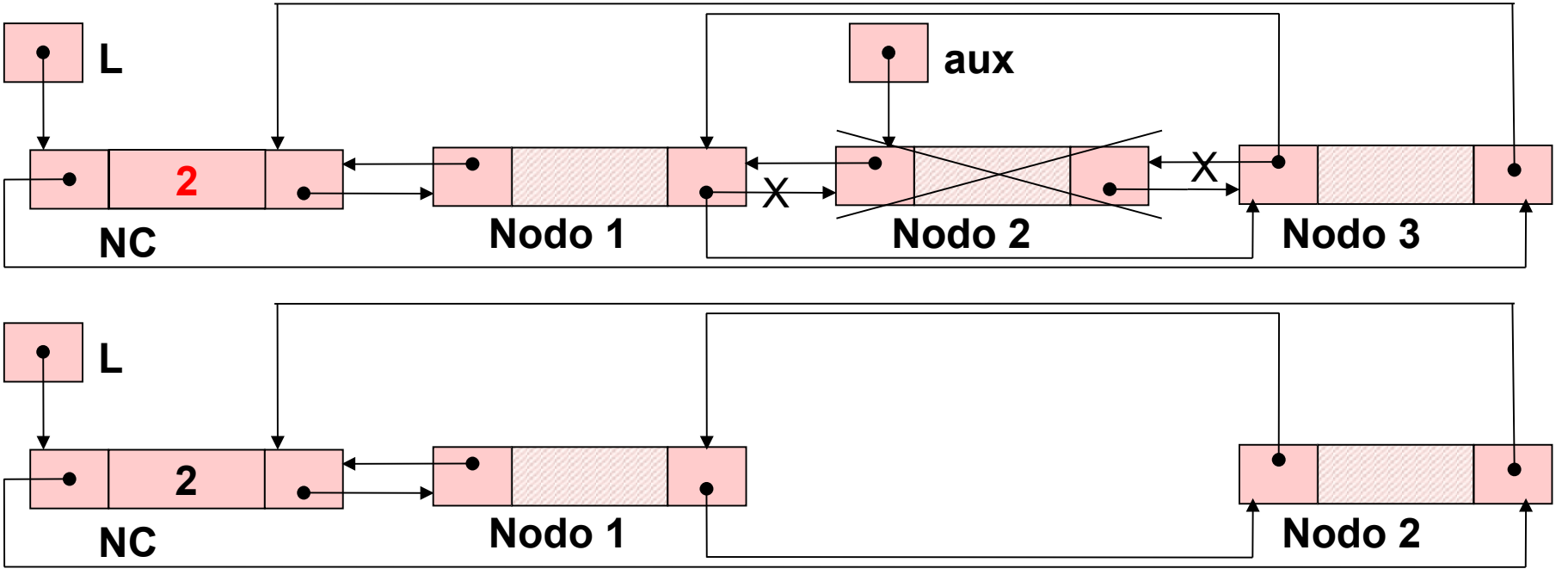
A posição é válida?

Todas as situações de remoção são tratadas da mesma forma?

UNIVASF

Listas Dup. Encad. Circ. com NC

Esquema do processo da retirada de um nó da lista circular duplamente encadeada com nó cabeçalho.



```c
void ret (LISTA_CIR_DUP_ENC_NC l, int k)
{
   if (k < 1 || k > tam(l))
   {
      printf ("\nERRO! Posição invalida para retirada.\n");
      exit (4);
   }
   l->inf--;
   for (; k>0; k--, l=l->prox);
   l->ant->prox = l->prox;
   l->prox->ant = l->ant;
   free (l);
}
```

Listas Dup. Encad. Circ. com NC

Com base no que foi visto implemente a operação destruir() que compõem o TAD LISTA_CIR_DUP_ENC_NC.

```
typedef struct nodo
{
    int inf;
    struct nodo * ant;
    struct nodo * prox;
}NODO;
typedef NODO * LISTA_CIR_DUP_ENC_NC;
void cria_lista (LISTA_CIR_DUP_ENC_NC *);
int eh_vazia (LISTA_CIR_DUP_ENC_NC);
int tam (LISTA_CIR_DUP_ENC_NC);
void ins (LISTA_CIR_DUP_ENC_NC, int, int);
int recup (LISTA_CIR_DUP_ENC_NC, int);
void ret (LISTA_CIR_DUP_ENC_NC, int);
void destruir (LISTA_CIR_DUP_ENC_NC);
```

```c
void destruir (LISTA_CIR_DUP_ENC_NC l)
{
    LISTA_CIR_DUP_ENC_NC aux;
    int tam = l->inf;
    do
    {
        aux = l;
        l = l->prox;
        free (aux);
    }while (tam--);
}
```

Listas Dup. Encad. Circ. com NC

Implemente, no **TAD LISTA_CIR_DUP_ENC_NC**, a seguinte operação:

void inverter_lista (LISTA_CIR_DUP_ENC_NC l);

a qual recebe uma referência para uma lista circular duplamente encadeada com nó cabeçalho e inverte a ordem de seus elementos.

UNIVASF

```c
void inverter_lista (LISTA_CIR_DUP_ENC_NC l)
{
    int i=tam(l);
    if (i>1)
    {
        NODO *aux;
        for (i++; i; l=l->ant,i--)
        {
            aux=l->ant;
            l->ant=l->prox;
            l->prox=aux;
        }
    }
}
```

Listas Circulares Duplamente Encadeadas com Nó Cabeçalho Exercícios

UNIVASF

Listas Dup. Encad. Circ. com NC

Os espectadores mais atentos de nossas aulas já devem ter se perguntado sobre a exploração das vantagens oriundas de cada abordagem.

Por exemplo:

As operações de inserção, recuperação e retirada, apresentadas na aula que versou sobre lista circulares duplamente encadeada com nó cabeçalho, poderiam ser mais eficientes?

UNIVASF

```c
typedef struct nodo
{
    int inf;
    struct nodo * ant;
    struct nodo * prox;
}NODO;
typedef NODO * LISTA_CIR_DUP_ENC_NC;
void cria_lista (LISTA_CIR_DUP_ENC_NC *);
int eh_vazia (LISTA_CIR_DUP_ENC_NC);
int tam (LISTA_CIR_DUP_ENC_NC);
void ins (LISTA_CIR_DUP_ENC_NC, int, int);
int recup (LISTA_CIR_DUP_ENC_NC, int);
void ret (LISTA_CIR_DUP_ENC_NC, int);
void destruir (LISTA_CIR_DUP_ENC_NC);
```

Listas Dup. Encad. Circ. com NC

Com base no que foi visto reimplemente a operação ins() que compõem o TAD LISTA_CIR_DUP_ENC_NC.

```
typedef struct nodo
{
    int inf;
    struct nodo * ant;
    struct nodo * prox;
}NODO;
typedef NODO * LISTA_CIR_DUP_ENC_NC;
void cria_lista (LISTA_CIR_DUP_ENC_NC *);
int eh_vazia (LISTA_CIR_DUP_ENC_NC);
int tam (LISTA_CIR_DUP_ENC_NC);
void ins (LISTA_CIR_DUP_ENC_NC, int, int);
int recup (LISTA_CIR_DUP_ENC_NC, int);
void ret (LISTA_CIR_DUP_ENC_NC, int);
void destruir (LISTA_CIR_DUP_ENC_NC);
```

UNIVASF

```c
void ins (LISTA_CIR_DUP_ENC_NC l, int v, int k)
{
    LISTA_CIR_DUP_ENC_NC aux, novo;
    if (k < 1 || k > l->inf+1) {
        printf ("\nERRO! Posição invalida para insercao.\n");
        exit (1);
    }
    novo = (LISTA_CIR_DUP_ENC_NC) malloc (sizeof(NODO));
    if (!novo) {
        printf ("\nERRO! Memoria insuficiente!\n");
        exit (2);
    }
    novo->inf = v;
```

```
14      if (k<=(l->inf)/2) {
15          for (aux=l; k>1; aux=aux->prox, k--);
16          novo->prox = aux->prox;
17          novo->ant = aux;
18          aux->prox = novo;
19          novo->prox->ant=novo;
20      } else {
21          for (aux=l; k<=l->inf; aux=aux->ant, k++);
22          novo->ant = aux->ant;
23          novo->prox = aux;
24          aux->ant = novo;
25          novo->ant->prox=novo;
26      }
27      l->inf++;
28  }
```

Listas Dup. Encad. Circ. com NC

Com base no que foi visto reimplemente a operação recup() que compõem o TAD LISTA_CIR_DUP_ENC_NC.

```
typedef struct nodo
{
    int inf;
    struct nodo * ant;
    struct nodo * prox;
}NODO;
typedef NODO * LISTA_CIR_DUP_ENC_NC;
void cria_lista (LISTA_CIR_DUP_ENC_NC *);
int eh_vazia (LISTA_CIR_DUP_ENC_NC);
int tam (LISTA_CIR_DUP_ENC_NC);
void ins (LISTA_CIR_DUP_ENC_NC, int, int);
int recup (LISTA_CIR_DUP_ENC_NC, int);
void ret (LISTA_CIR_DUP_ENC_NC, int);
void destruir (LISTA_CIR_DUP_ENC_NC);
```

```c
int recup (LISTA_CIR_DUP_ENC_NC l, int k)
{
    int tamanho = l->inf;
    if (k < 1 || k > tamanho){
        printf ("\nERRO! Consulta invalida.\n");
        exit (3);
    }
    if (k<=tamanho/2)
        for (;k>0;k--)
            l=l->prox;
    else
        for (;k<=tamanho;k++)
            l=l->ant;
    return (l->inf);
}
```

Listas Dup. Encad. Circ. com NC

Com base no que foi visto reimplemente a operação ret() que compõem o TAD LISTA_CIR_DUP_ENC_NC.

```
typedef struct nodo
{
    int inf;
    struct nodo * ant;
    struct nodo * prox;
}NODO;
typedef NODO * LISTA_CIR_DUP_ENC_NC;
void cria_lista (LISTA_CIR_DUP_ENC_NC *);
int eh_vazia (LISTA_CIR_DUP_ENC_NC);
int tam (LISTA_CIR_DUP_ENC_NC);
void ins (LISTA_CIR_DUP_ENC_NC, int, int);
int recup (LISTA_CIR_DUP_ENC_NC, int);
void ret (LISTA_CIR_DUP_ENC_NC, int);
void destruir (LISTA_CIR_DUP_ENC_NC);
```

UNIVASF

```c
void ret (LISTA_CIR_DUP_ENC_NC l, int k)
{
   int tamanho = l->inf;
   if (k < 1 || k > tamanho)
   {
      printf ("\nERRO! Posição invalida para retirada.\n");
      exit (4);
   }
   l->inf--;
   if (k<=(l->inf)/2)
      for (; k>0; k--, l=l->prox);
   else
      for (; k<=tamanho; k++, l=l->ant);
   l->ant->prox = l->prox;
   l->prox->ant = l->ant;
   free (l);
}
```

Listas não inteiras e não homogêneas

UNIVASF

Listas não inteiras e não homogêneas

Evidentemente, o campo ***inf*** de um nó numa lista não precisa necessariamente armazenar, apenas, um valor inteiro.

Pode-se, por exemplo, representar uma lista de strings, por uma lista encadeada, tornando assim, necessários nós contendo vetores de caracteres em seus campos ***inf***.

UNIVASF

Listas não inteiras e não homogêneas

Tais nós poderiam ser declarados como:

```
typedef struct node
{
        char inf [100];
        struct node *prox;
} NODE;
```

Listas não inteiras e não homogêneas

É provável que determinada aplicação exija nós com registros no campo *inf*.

Por exemplo, podemos necessitar de uma lista de registros de estudantes. Onde, cada registro, contém as seguintes informações: nome do estudante, número de identificação na faculdade, endereço, coeficiente de rendimento e área de especialização.

Os nós para tal implementação podem ser declarados como segue:

UNIVASF

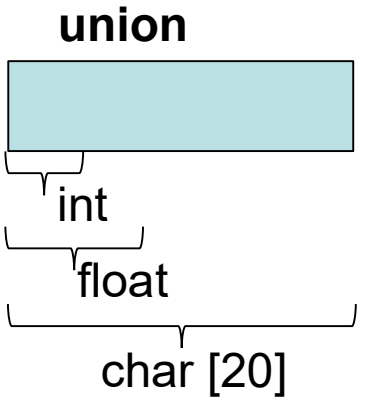
```c
typedef struct
{
    char nome [30];
    char id [9];
    char end [100];
    float cr;
    char earea [20];
} INF;
typedef struct node
{
    INF inf;
    struct node *prox;
} NODE;
```

Listas não inteiras e não homogêneas

Para representar listas não-homogêneas (as que contém nós de diversos tipos), podemos utilizar uma união.

Exemplo:

```
1   #define INTGR  1
2   #define FLT 2
3   #define STRING 3
4   typedef struct node  {
5       int etype;
6       union inf {
7           int ival;
8           float fval;
9           char sval[20];
10      } element;
11      struct node *prox;
12  } NODE;
```



Listas não inteiras e não homogêneas

O exemplo anterior define um nó cujos itens podem ser inteiros, números de ponto flutuante ou strings, dependendo do valor de *etype* correspondente.

Como uma união é suficientemente grande para armazenar seu maior componente, as funções ***sizeof*** e ***malloc*** podem ser usadas para alocar armazenamento para o nó. Evidentemente, fica sob a responsabilidade do programador usar os componentes de um nó, conforme for apropriado.

Listas não inteiras e não homogêneas

Com base no que foi visto, proponha a estrutura de dados e implemente a operação de inserção de um nó em uma lista circular duplamente encadeada não homogênea. Onde, os nós podem ter o campo com informação do tipo inteiro, ponto flutuante ou string.

A lista não deve possuir valores replicados.

Dica: Baseie-se na definição do nó vista anteriormente.

UNIVASF