

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);
```

```
1 void destruir (LISTA_CIR_DUP_ENC_NC l)
2 {
3     LISTA_CIR_DUP_ENC_NC aux;
4     int tam = l->inf;
5     do
6     {
7         aux = l;
8         l = l->prox;
9         free (aux);
10    }while (tam--);
11 }
```


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 I);
```

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

```
1 void inverter_lista (LISTA_CIR_DUP_ENC_NC l)
2 {
3     int i=tam(l);
4     if (i>1)
5     {
6         NODO *aux;
7         for (i++; i; l=l->ant,i--)
8         {
9             aux=l->ant;
10            l->ant=l->prox;
11            l->prox=aux;
12        }
13    }
14 }
```



Listas Circulares Duplamente Encadeadas com Nó Cabeçalho Exercícios

Listas Dup. Encad. Circ. com NC

Os espectadores mais atentos de nossas videoaulas 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 videoaula que versou sobre lista circulares duplamente encadeada com nó cabeçalho, poderiam ser mais eficientes?

```
1  typedef struct nodo
2  {
3      int inf;
4      struct nodo * ant;
5      struct nodo * prox;
6  }NODO;
7  typedef NODO * LISTA_CIR_DUP_ENC_NC;
8  void cria_lista (LISTA_CIR_DUP_ENC_NC *);
9  int eh_vazia (LISTA_CIR_DUP_ENC_NC);
10 int tam (LISTA_CIR_DUP_ENC_NC);
11 void ins (LISTA_CIR_DUP_ENC_NC, int, int);
12 int recup (LISTA_CIR_DUP_ENC_NC, int);
13 void ret (LISTA_CIR_DUP_ENC_NC, int);
14 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);
```

```
1 void ins (LISTA_CIR_DUP_ENC_NC l, int v, int k)
2 {
3     LISTA_CIR_DUP_ENC_NC aux, novo;
4     if (k < 1 || k > l->inf+1) {
5         printf ("\nERRO! Posição invalida para insercao.\n");
6         exit (1);
7     }
8     novo = (LISTA_CIR_DUP_ENC_NC) malloc (sizeof(NODO));
9     if (!novo) {
10        printf ("\nERRO! Memoria insuficiente!\n");
11        exit (2);
12    }
13    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 reimplente 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);
```

```
1 int recup (LISTA_CIR_DUP_ENC_NC l, int k)
2 {
3     int tamanho = l->inf;
4     if (k < 1 || k > tamanho){
5         printf ("\nERRO! Consulta invalida.\n");
6         exit (3);
7     }
8     if (k<=tamanho/2)
9         for (;k>0;k--)
10            l=l->prox;
11     else
12         for (;k<=tamanho;k++)
13            l=l->ant;
14     return (l->inf);
15 }
```

Listas Dup. Encad. Circ. com NC

Com base no que foi visto reimplente 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);
```

```
1 void ret (LISTA_CIR_DUP_ENC_NC l, int k)
2 {
3     int tamanho = l->inf;
4     if (k < 1 || k > tamanho)
5     {
6         printf ("\nERRO! Posição invalida para retirada.\n");
7         exit (4);
8     }
9     l->inf--;
10    if (k<=(l->inf)/2)
11        for (; k>0; k--, l=l->prox);
12    else
13        for (; k<=tamanho; k++, l=l->ant);
14    l->ant->prox = l->prox;
15    l->prox->ant = l->ant;
16    free (l);
17 }
```

Listas não inteiras e não homogêneas

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*.

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:

```
1  typedef struct
2  {
3      char nome [30];
4      char id [9];
5      char end [100];
6      float cr;
7      char earea [20];
8  } INF;
9  typedef struct node
10 {
11     INF inf;
12     struct node *prox;
13 } NODE;
```

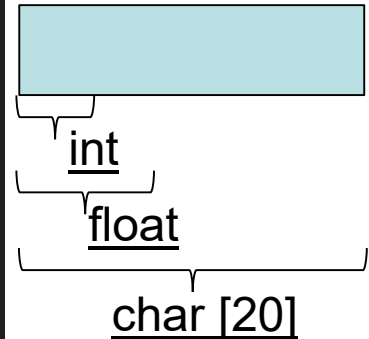
Listas não inteiras e não homogêneas

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;
```

union



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.