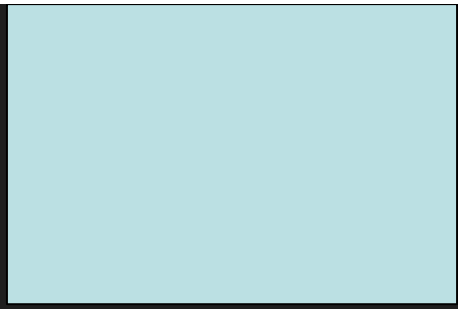
Listas Circulares

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

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

```
1   int recup (LISTA_CIRCULAR l, int k)
2   {
11  }
```

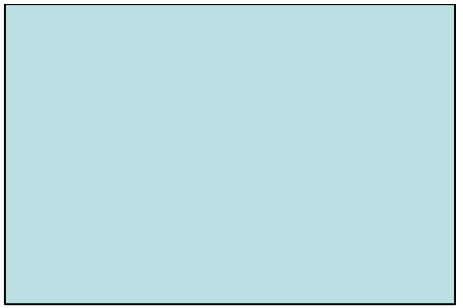


Listas Circulares

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

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

Listas Circulares



Dicas:

A posição é válida?

Todas as situações de remoção são tratadas da mesma forma?

```c
void ret (LISTA_CIRCULAR *pl, int k) {
   int tamanho = tam(*pl);
   if (k < 1 || k > tamanho) {
      printf ("\nERRO! Posição invalida para retirada.\n");
      exit (4);
    }
   if (tamanho==1) {
      free (*pl);
      *pl = NULL;
   } else {
      LISTA_CIRCULAR aux, aux2; /* NODO *aux; */
      int i;
      for (aux=*pl, i=k; i>1; i--, aux=aux->next);
      aux2 = aux->next;
      aux->next = aux2->next;
      if (k==tamanho)
         *pl=aux;
      free (aux2);
   }
}
```

Listas Circulares

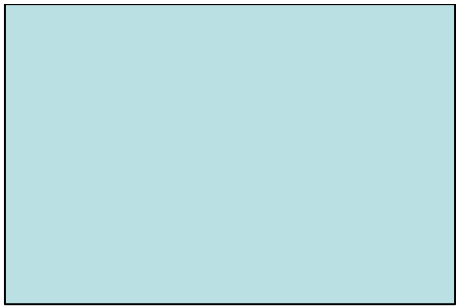
Com base no que foi visto implemente a operação destruir() que compõem o TAD LISTA_CIRCULAR.

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

```c
void destruir (LISTA_CIRCULAR l)
{
    if (l)
    {
        LISTA_CIRCULAR aux;
        for (aux=l->next; aux!=l ; aux = aux->next)
            free (aux);
        free (aux);
    }
}
```

```c
void destruir (LISTA_CIRCULAR *pl)
{
    if (*pl)
    {
        LISTA_CIRCULAR aux;
        for (aux=(*pl)->next; aux!=*pl ; aux = aux->next)
            free (aux);
        free (aux);
        *pl=NULL;
    }
}
```

Listas Circulares – Nó de Cabeçalho

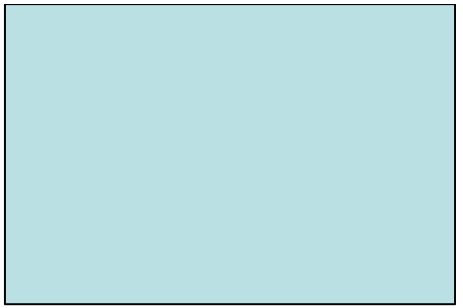


O conceito de ***nó de cabeçalho*** também pode ser empregado nas listas circulares.

A implementação de um TAD LISTA_CIRCULAR_COM_NC é sugerida como uma exercício de fixação.

Listas Duplamente Encadeadas

Listas Duplamente Encadeadas



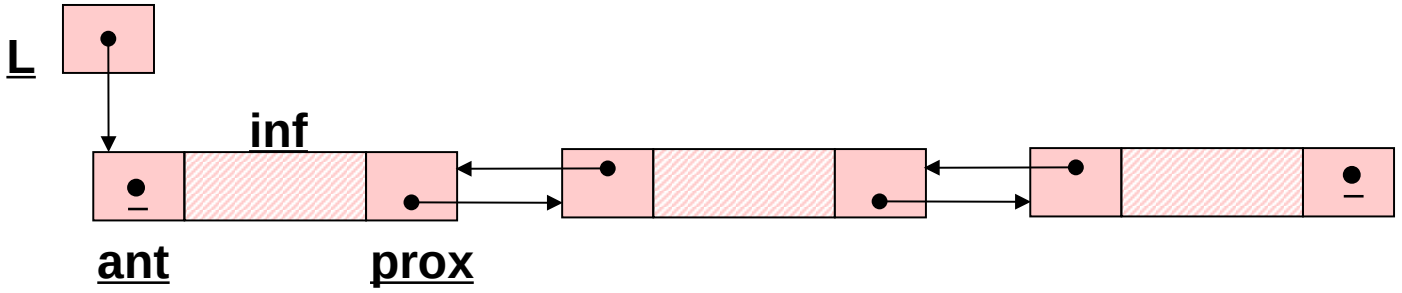
Como vimos, uma lista circular possui vantagem sobre uma lista linear. Contudo, esta ainda possui limitações.

Por exemplo, não podemos percorrê-la no sentido contrário o que impõe, por exemplo, a necessidade de se ter um ponteiro para o antecessor para inserirmos ou retirarmos um k-ésimo elemento.

Com o objetivo de sanar esta limitação surgiram as ***listas duplamente encadeadas***.

Listas Duplamente Encadeadas

Em uma ***lista duplamente encadeada*** os elementos possuem três campos: o campo ***inf*** o qual contém a informação, o campo ***ant*** que possui um ponteiro para o elemento antecessor e o campo ***prox*** que é uma referência para o próximo elemento.



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

Listas Duplamente Encadeadas

Com base no que foi visto implemente a operação cria_lista() que compõem o TAD LISTA_DUP_ENC.

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

```
1  void cria_lista (LISTA_DUP_ENC *pl)
2  {
3      *pl=NULL;
4  }
```

Listas Duplamente Encadeadas

Com base no que foi visto implemente a operação eh_vazia que compõem o TAD LISTA_DUP_ENC.

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

```c
int eh_vazia (LISTA_DUP_ENC l)
{
    return (!l);
}
```

Listas Duplamente Encadeadas

Com base no que foi visto implemente a operação tam() que compõem o TAD LISTA_DUP_ENC.

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

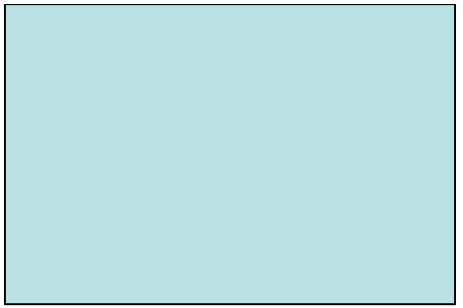
```c
int tam (LISTA_DUP_ENC l)
{
    int cont;
    for (cont=0; l; cont++, l = l->prox);
    return (cont);
}
```

Listas Duplamente Encadeadas

Com base no que foi visto implemente a operação ins() que compõem o TAD LISTA_DUP_ENC.

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

Listas Duplamente Encadeadas



Dicas:

A posição é válida?

Tem espaço na memória para armazenar mais um elemento?

Todas as situações de inserção são tratadas da mesma forma?

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

```
13      if (k==1) { /*situações um e dois*/
14          novo->ant = NULL;
15          novo->prox = *pl;
16          *pl = novo;
17          if ((*pl)->prox) /*situação dois*/
18              (*pl)->prox->ant=novo;
19      } else { /*situações três e quatro*/
20          LISTA_DUP_ENC aux;
21          for (aux=*pl; k>2; aux=aux->prox, k--);
22          novo->prox = aux->prox;
23          aux->prox = novo;
24          novo->ant=aux;
25          if (novo->prox) /*situação quatro*/
26              novo->prox->ant=novo;
27      }
28  }
```

Listas Duplamente Encadeadas

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

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