```
void ret (LISTA_ENC_EST *l, int pos) {
  if (pos<1 || pos> tam(l)) {
    printf("\nPosicao invalida!");
    exit (1);
  } else {
    int aux;
    if (pos==1)
    {
      aux = l->ind_pri_ele;
      l->ind_pri_ele = l->elementos[l-
   >ind_pri_ele].next;
      l->elementos[aux].next = l->ind_nodo_livre;
      l->ind_nodo_livre = aux;
    }
```

```
  else
  {
    int ind;
    for (ind=l->ind_pri_ele; --pos-1; ind=l->elementos[ind].next);
    aux = l->elementos[ind].next;
    l->elementos[ind].next = l->elementos[l->elementos[ind].next].next;
    l->elementos[aux].next = l->ind_nodo_livre;
    l->ind_nodo_livre = aux;
  }
 }
}
```

Alocação Encadeada

Contudo, esta abordagem ainda impõe limitações ou desvantagens.

Uma grande desvantagem é o fato de uma quantidade fixa de armazenamento permanecer alocada para a lista, mesmo quando a estrutura estiver utilizando uma quantidade menor ou talvez nenhum armazenamento.

Além disso, não mais do que essa quantidade fixa de armazenamento poderá ser utilizada, introduzindo, dessa forma, a possibilidade de estouro na representação.

UNIVASF

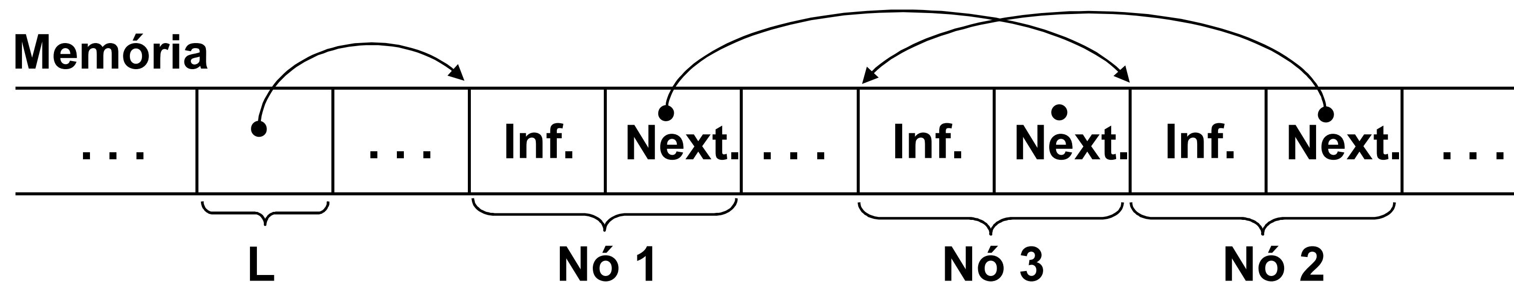
Alocação Encadeada

Com a utilização da alocação dinâmica de memória associada à alocação encadeada obtém-se a vantagem da alocação gradual para armazenamento da estrutura, também permitindo um crescimento indefinido da mesma (até os limites da memória disponíveis para o programa).

Na alocação encadeada dinâmica os nós de uma lista encontram-se aleatoriamente dispostos na memória e são interligados por ponteiros, que indicam a posição do próximo elemento da lista.

Alocação Encadeada

Logo, percebe-se que, como anteriormente, existe um custo agregado para se obter flexibilidade, referente à inclusão de um campo em cada nó (elemento), utilizado para armazenar o endereço de memória de seu sucessor, ou seja, um aumento no espaço de memória necessário para armazenar cada elemento, o esquema a seguir, ilustra esta estrutura:



UNIVASF

Alocação Encadeada

Observação: Como ocorria com a alocação encadeada estática, a alocação encadeada dinâmica requer o uso de um ponteiro que indica o endereço de seu primeiro nó.

Com estas informações, podemos definir o TAD LISTA_ENC como (considerando que o campo informação armazena um valor inteiro):

UNIVASF

```
typedef struct nodo
{
  int inf;
  struct nodo * next;
}NODO;
typedef NODO * LISTA_ENC;
void cria_lista (LISTA_ENC *);
int eh_vazia (LISTA_ENC);
int tam (LISTA_ENC);
void ins (LISTA_ENC *, int, int);
int recup (LISTA_ENC, int);
void ret (LISTA_ENC *, int);
```


UNIVASF

Alocação Encadeada

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

```
void cria_lista (LISTA_ENC *pl)
{
  *pl=NULL;
}
```


UNIVASF

Alocação Encadeada

Com base no que foi visto implemente a operação eh_vazia() que compõem o TAD LISTA_ENC.

```
int eh_vazia (LISTA_ENC l)
{
  return (l == NULL);
}
```

Alocação Encadeada

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

```
int tam (LISTA_ENC l)
{
  int cont;
  for (cont=0; l!= NULL; cont++)
    l = l->next;
  return (cont);
}
```


UNIVASF

Alocação Encadeada

Com base no que foi visto implemente a operação tam() que compõem o TAD LISTA_ENC. Porém, implemente a operação utilizando recursividade.

UNIVASF

Alocação Encadeada

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

```
typedef struct nodo
{
  int inf;
  struct nodo * next;
}NODO;
typedef NODO * LISTA_ENC;
void cria_lista (LISTA_ENC *);
int eh_vazia (LISTA_ENC);
int tam (LISTA_ENC);
void ins (LISTA_ENC *, int, int);
int recup (LISTA_ENC, int);
void ret (LISTA_ENC *, int);
```