Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

# AULA 12

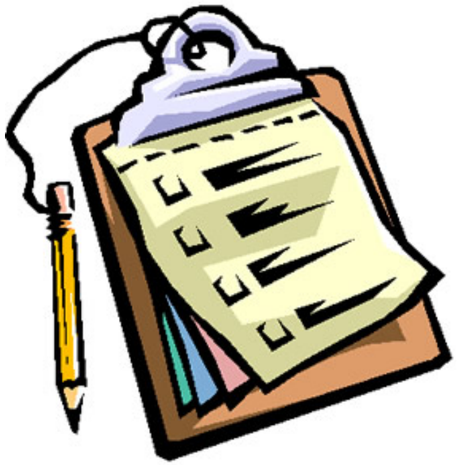
## Estrutura de dados básico I (EDB1)

Prof. Msc. Janiheryson Felipe (Felipe)

Natal, RN
2023

# OBJETIVOS DA AULA

- Apresentar os conceitos de lista smplesmente encadeada e sua implementação em memória.
  - Conhecer as listas smplesmente encadeada;
  - Implementar uma lista smplesmente encadeada; a partir do zero;



# LISTAS LIGADAS

# LISTAS LIGADAS (LINKED LIST)

Uma lista ligada é uma coleção não sequencial de itens de dados. É uma estrutura de dados dinâmica.

Para cada item de dados em uma lista encadeada, há um ponteiro associado que daria a localização de memória do próximo item de dados na lista encadeada.

# LISTAS LIGADAS (LINKED LIST)

Os itens de dados na lista ligada não estão em locais de memória consecutivos. Eles podem ser em qualquer lugar, mas o acesso a esses itens de dados é mais fácil, pois cada item de dados contém o endereço do próximo item de dados.

# LISTAS ENCADEADAS - VANTAGENS

- 1. As listas encadeadas são estruturas de dados dinâmicas. ou seja, eles podem crescer ou diminuir durante a execução de um programa.

- 2. As listas encadeadas têm utilização de memória eficiente. Aqui, a memória não é pré-alocada. A memória é alocada sempre que é necessária e é desalocada (removida) quando não é mais necessária.

# LISTAS ENCADEADAS - VANTAGENS

- 3. As inserções e exclusões são mais fáceis e eficientes. As listas vinculadas fornecem flexibilidade na inserção de um item de dados em uma posição especificada e na exclusão do item de dados da posição especificada.

- 4. Muitos aplicações complexas podem ser facilmente executados com listas encadeadas.

# LISTAS ENCADEADAS - DESVANTAGENS

- 1. Consome mais espaço porque cada nó requer um ponteiro adicional para armazenar o endereço do próximo nó.

- 2. Pesquisar um determinado elemento na lista é difícil e também demorado

# VETORES VS LISTA LISTA LIGADAS

| ARRAY | LINKED LIST |
|---|---|
| Size of an array is fixed | Size of a list is not fixed |
| Memory is allocated from stack | Memory is allocated from heap |
| It is necessary to specify the number of elements during declaration (i.e., during compile time). | It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time). |
| It occupies less memory than a linked list for the same number of elements. | It occupies more memory. |
| Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room. | Inserting a new element at any position can be carried out easily. |
| Deleting an element from an array is not possible. | Deleting an element is possible. |

# VETORES VS LISTA LISTA LIGADAS

| FEATURE | ARRAYS | LINKED LISTS |
|---|---|---|
| Sequential access | efficient | efficient |
| Random access | efficient | inefficient |
| Resigning | inefficient | efficient |
| Element rearranging | inefficient | efficient |
| Overhead per elements | none | 1 or 2 links |

# ESTRUTURA DE UMA LISTA LIGADA

- Uma lista ligada aloca espaço para cada elemento separadamente em seu próprio bloco de memória chamado "nó".
- A lista obtém uma estrutura geral usando ponteiros para conectar todos os seus nós, como os elos de uma cadeia.
- Cada nó contém dois campos; um campo "dados" para armazenar qualquer elemento e um campo "próximo" que é um ponteiro usado para vincular ao próximo nó.

# ESTRUTURA DE UMA LISTA LIGADA

- Cada nó é alocado no heap usando malloc() ou new, então a memória do nó continua a existir até que seja explicitamente desalocada usando free() ou delete(). A frente da lista é um ponteiro para o nó inicial.

- O início da lista encadeada é armazenado em um ponteiro "inicial" que aponta para o primeiro nó. O primeiro nó contém um ponteiro para o segundo nó. O segundo nó contém um ponteiro para o terceiro nó, ... e assim por diante.

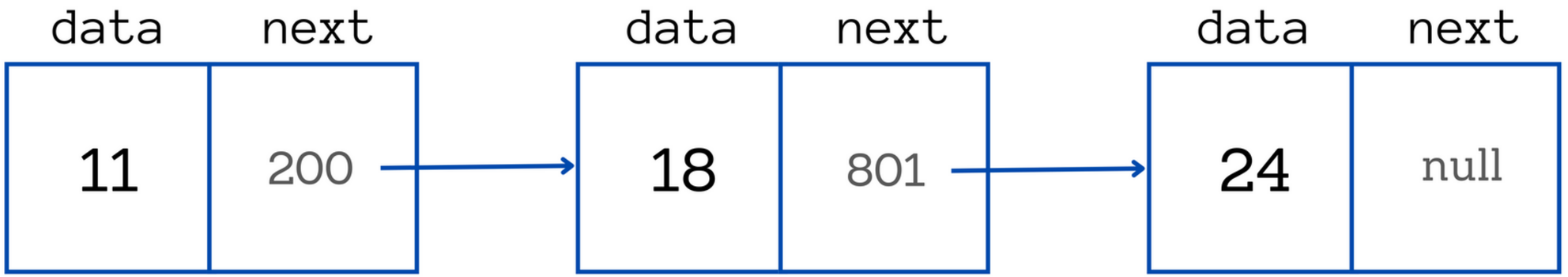
# ESTRUTURA DE UMA LISTA LIGADA

- O último nó da lista tem seu campo próximo definido como NULL para marcar o fim da lista. O código pode acessar qualquer nó na lista começando no início e seguindo os próximos ponteiros.

- Não há como retroceder uma busca;

# LISTAS ENCADEADAS (LINKED LIST)

Uma lista ligadas (= linked list = lista encadeadas)
cada célula contém um objeto (todos os objetos são do mesmo tipo) e o endereço da célula seguinte.



# CRIAR UMA NOVA LISTA

```c
typedef struct no{
    int valor;
    struct no *prox;
}No;
```

# INSERIR ELEMENTOS

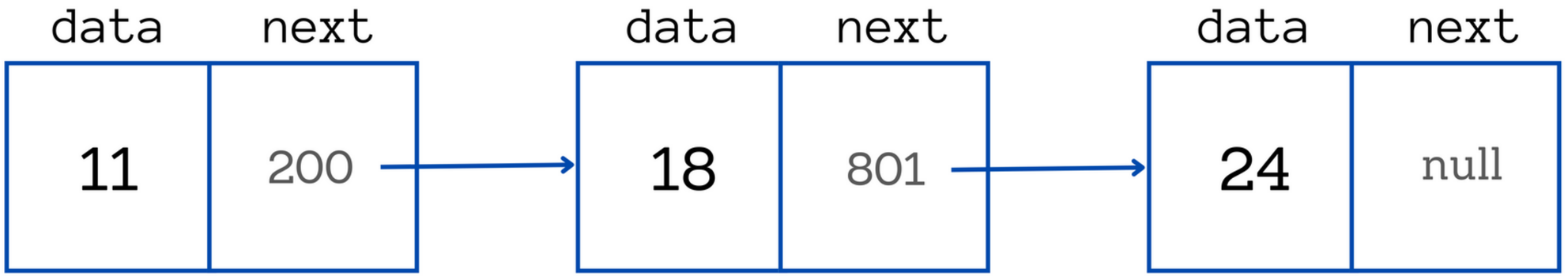
Uma das operações mais primitivas que podem ser feitas em uma lista ligada é a inserção de um nó:

- A memória deve ser alocada para o novo nó (de maneira semelhante ao que é feito ao criar uma lista).
- O novo nó conterá o campo de dados vazio e o campo proximo vazio.
- O campo de dados do novo nó é então armazenado com as informações lidas do usuário.
- O próximo campo do novo nó é atribuído a NULL.

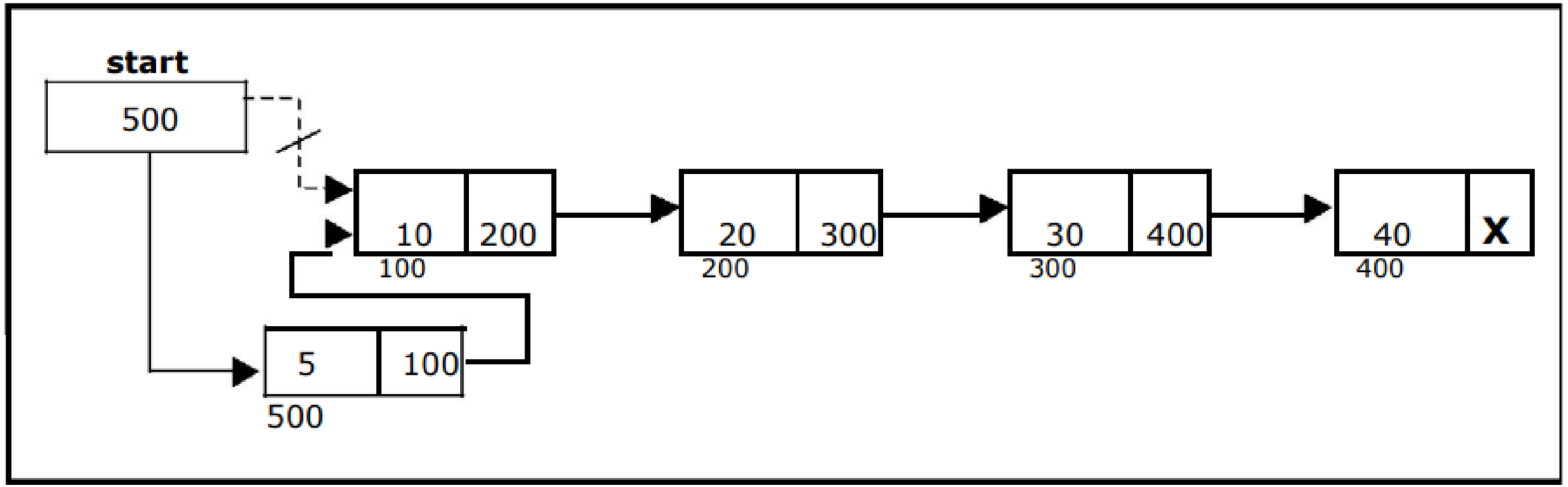
# INSERIR ELEMENTOS

O novo nó pode então ser inserido em três lugares diferentes, a saber:

- Inserindo um nó no início.
- Inserindo um nó no final.
- Inserindo um nó na posição intermediária.



# INSERIR UM NÓ - INICIO



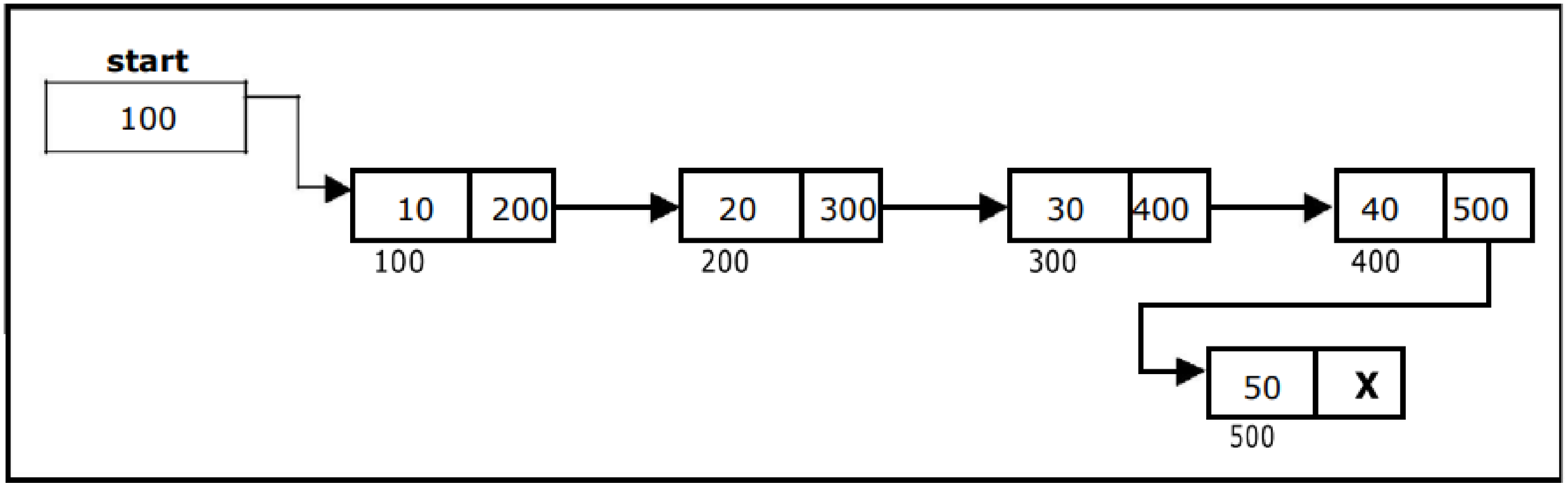
```c
void push_front(No **head, int x){
  //Criar um novo nó
  No *novoNo = (No*)malloc(sizeof(No));
  novoNo->valor = x;
  novoNo->prox = NULL;

  //Verificar se a lista esta vazia
  if(*head == NULL){
    *head = novoNo;

  //Caso a lista não esteja vazia;
  }else{
    novoNo->prox = *head;
    *head = novoNo;
  }
}
```

# INSERIR UM NÓ INÍCIO

# INSERIR UM NÓ - FIM



```c
//Insere um elemento no final da lista
void push_back(No **head, int x){
  //Criar um novo nó
  No *novoNo = (No*)malloc(sizeof(No));
  novoNo->valor = x;
  novoNo->prox = NULL;

  //Verificar se a lista esta vazia
  if(*head == NULL){
    *head = novoNo;
    |
  //Caso a lista não esteja vazia;
  }else{
    No *temp = *head;
    //Percorrer até o último elemento
    while (temp->prox != NULL){
      temp = temp->prox;
    }
    temp->prox = novoNo;
  }
}
```

# INSERIR UM NÓ FIM

# PROCURAR UM VALOR NA LISTA

```
list* find(int x, list *p){
    if(p == NULL){
      return NULL;
    }
   if(p->valor == x){
     return p;
   }else{
     return find(x, p->prox);
   }
}
```

# IMPRIMIR OS VALORES DA LISTA

```
void print(list *p){
   if(p == NULL){
      return;
   }else{
         printf("%d\n", p->valor);
         print(p->prox);
   }
}
```

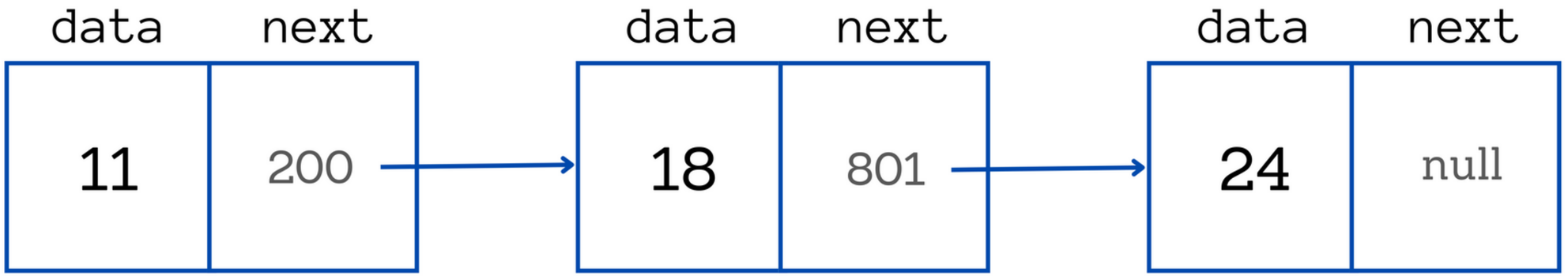
# TAMANHO DA LISTA

```
int size(list *p){
   if(p->prox == NULL){
      return 1;
   }else{
     return size(p->prox) + 1;
   }
}
```

# DELETAR ELEMENTOS

Um nó pode ser deletado em três lugares diferentes, a saber:

- Nó do início da lista.
- Nó do final da lista.
- Nó de uma posição intermediária.



# DELETAR ELEMENTOS (INICIO)

```
list *pop_front(int size, list *p){
  if(size < 2){
    free(p);
    return NULL;
  }else{
    list *temp = p->prox;
    free(p);
    return temp;
  }
}
```

# DELETAR ELEMENTOS (FIM)

```
void pop_back(int size, list *p){
   if(size < 2){
      free(p);
   }else{
      if(p->prox->prox == NULL){
         p->prox = NULL;
         free(p->prox);
      }else{
         return pop_back(size, p->prox);
      }
   }
```

# INSERIR ELEMENTOS

Uma das operações mais primitivas que podem ser feitas em uma lista ligada é a inserção de um nó:

- A memória deve ser alocada para o novo nó (de maneira semelhante ao que é feito ao criar uma lista).
- O novo nó conterá o campo de dados vazio e o campo proximo vazio.
- O campo de dados do novo nó é então armazenado com as informações lidas do usuário.
- O próximo campo do novo nó é atribuído a NULL.

# DÚVIDAS???