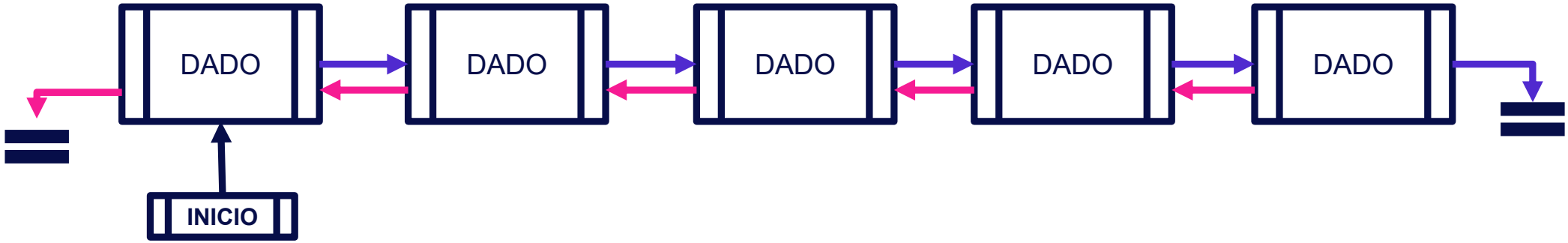
Estrutura de Dados 1

LISTA DUPLAMENTE ENCADEADA

Profª Juliana Franciscani

LISTA DUPLAMENTE ENCADEADA

- Estrutura de dados que armazena uma sequência de elementos chamados "nós".
- São úteis para quando o tamanho da coleção muda dinamicamente.



Estrutura de lista onde cada nó tem duas referências: uma para o próximo nó e outra para o nó anterior.

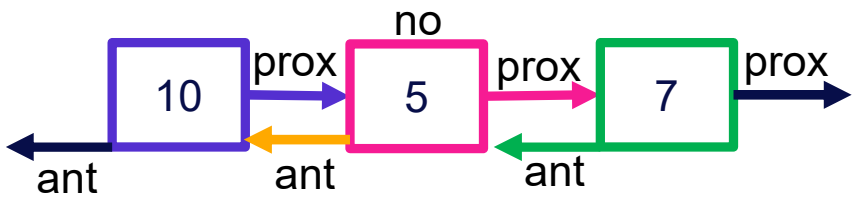
LISTA DUPLAMENTE ENCADEADA

- Campos de um nó:
  - Dado/Valor
  - Ponteiro para o próximo nó (prox ou next)
  - Ponteiro para o nó anterior (ant ou prev)

- Regras básicas:
  - O primeiro nó da lista aponta para NULO no ponteiro anterior
  - O último nó aponta para NULO no ponteiro próximo.
  - Para acessar um nó pode-se usar:

(no→prox)→ant ou

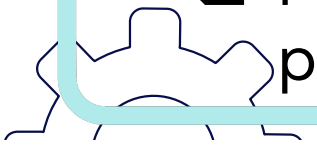
(no→ant)→prox



LISTA DUPLAMENTE ENCADEADA

- Vantagens:
  - Navegação em ambas as direções (anterior e próxima).
  - Facilita a remoção e inserção de nós em qualquer posição.
  - Usada em aplicações que exigem retrocesso, como editores de texto, e gerenciamento de navegação (web: manter um histórico de páginas visitadas)

- Desvantagens:
  - Maior uso de memória, pois cada nó armazena dois ponteiros.
  - Inserção e exclusão podem ser mais complexas, pois é preciso atualizar dois ponteiros.
  - Maior possibilidade de erros, especialmente em manipulação de ponteiros.

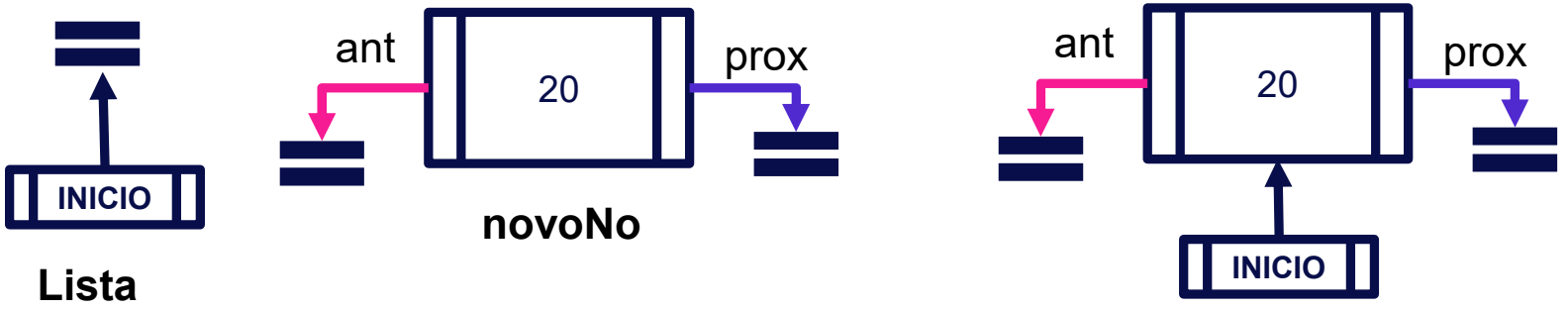


LISTA DINÂMICA

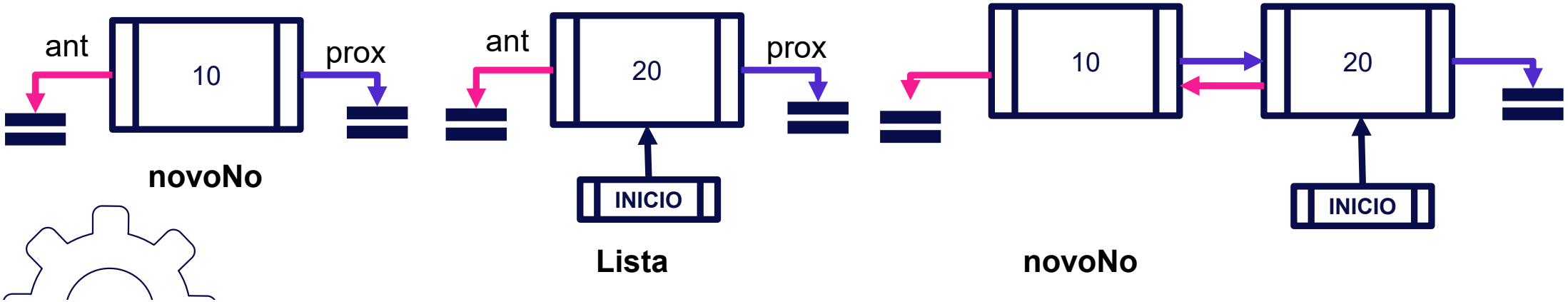
- Criar lista;
- Apagar lista;
- Inserir item;
- Acessar item (dado uma chave);
- Remover item (dado uma chave);
- Contar número de itens;
- Copiar lista;
- Imprimir lista;
- Imprimir lista inversa.

LISTA DUPLAMENTE ENCADEADA – Inserção

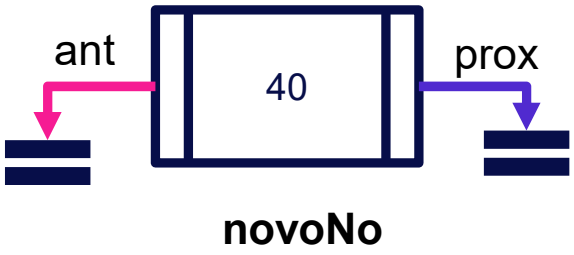
- Inserção em uma lista vazia: Valor 20

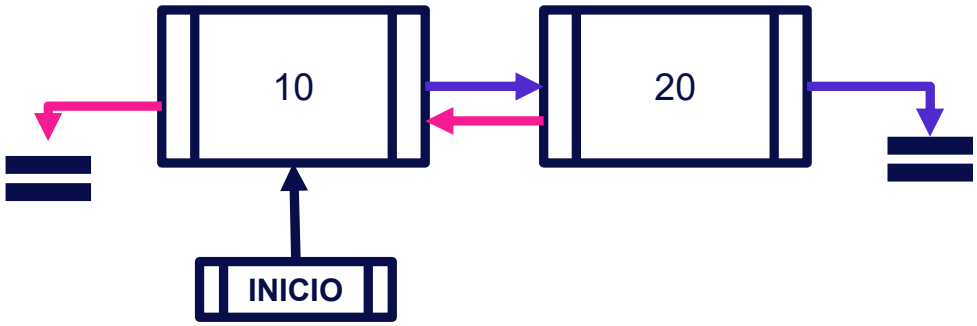


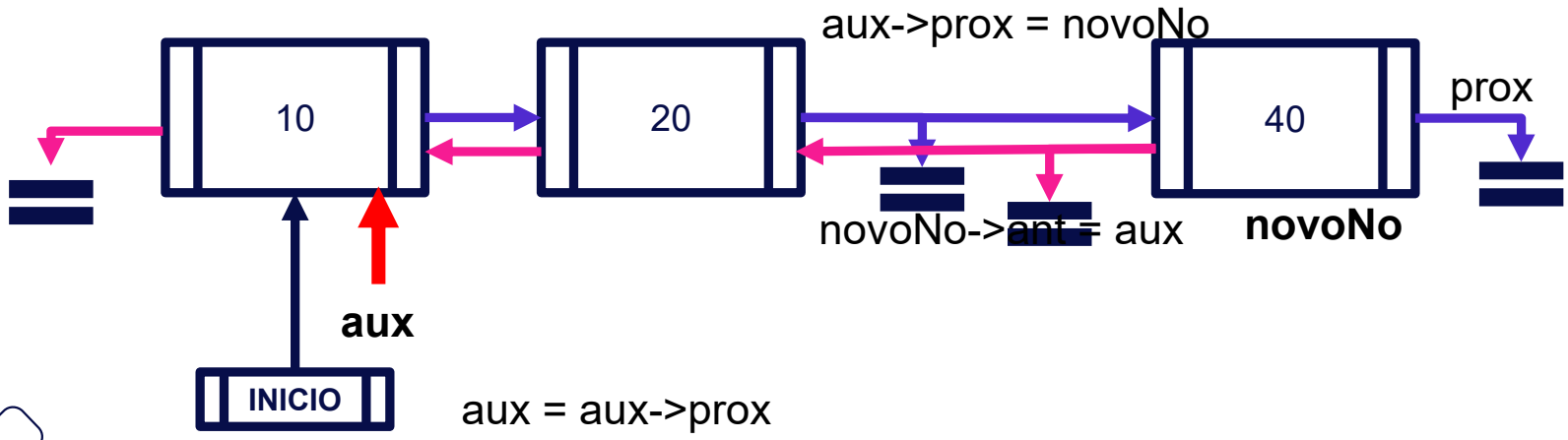
- Inserção no início em uma lista que já possui elemento: 10



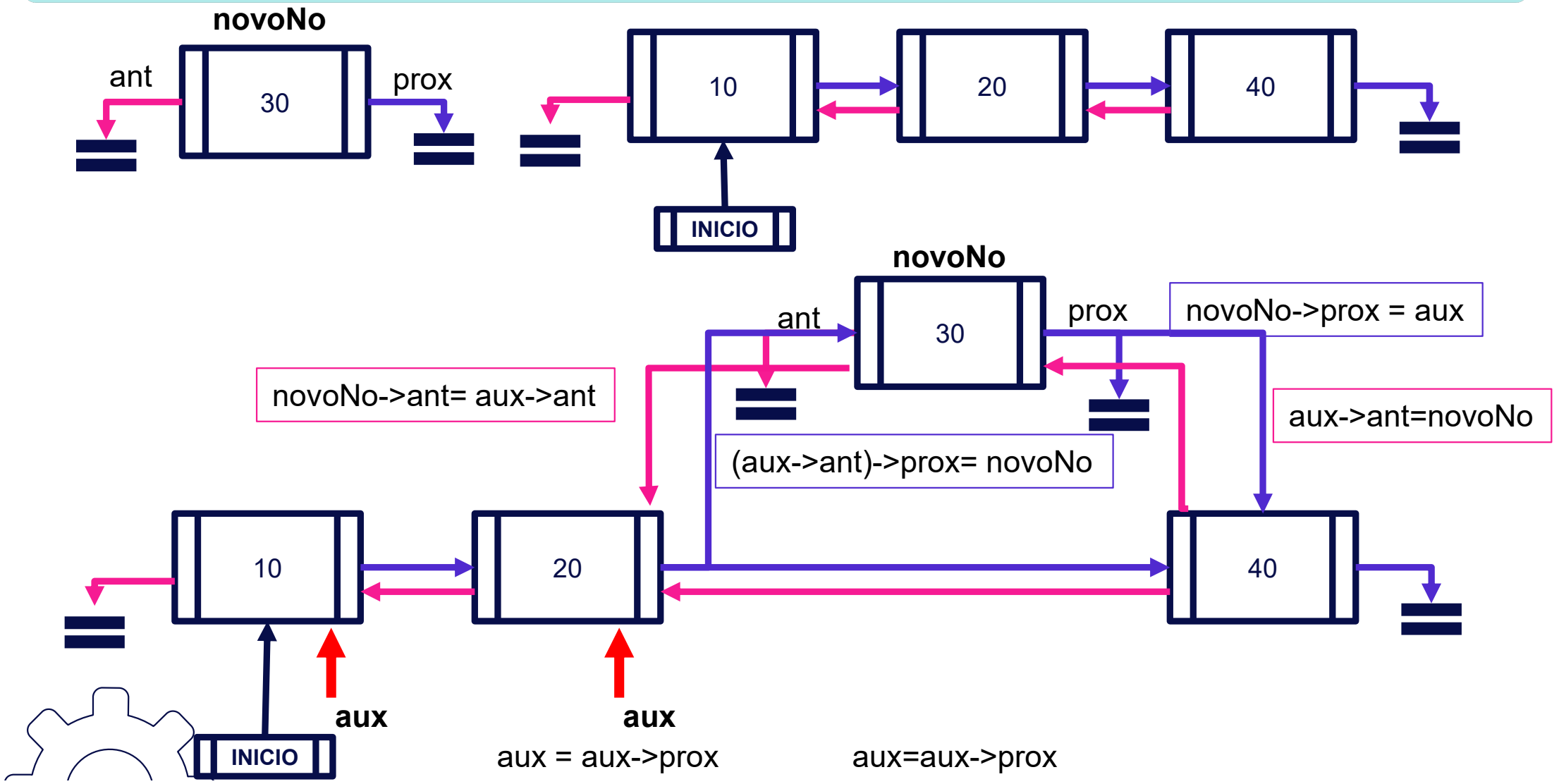
➢ Inserção no fim em uma lista que já possui elementos: 40





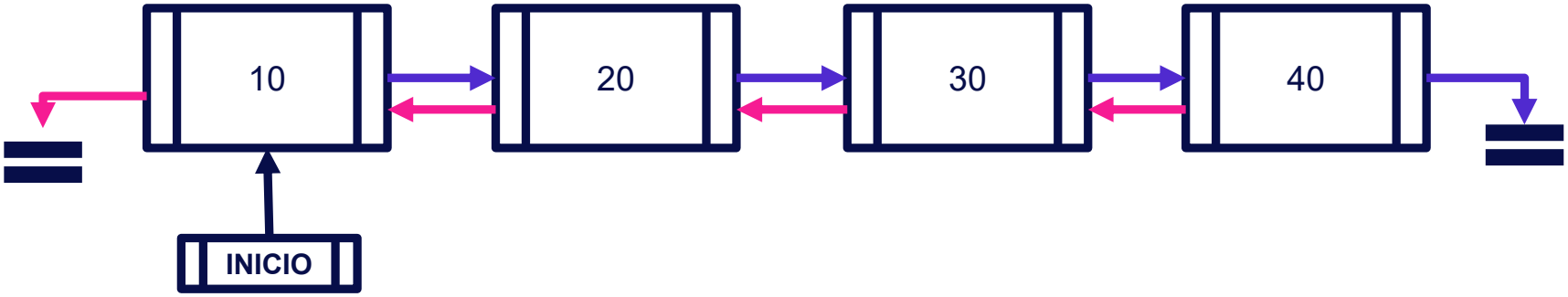


- Inserção ordenada em uma lista que já possui elementos: 30



LISTA DUPLAMENTE ENCADEADA – Inserção

- Lista Final



Criação da lista

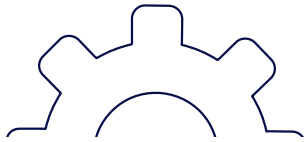
```
Lista* criarLista(){
    //ponteiro que irá apontar para uma estrutura Lista (que já é um ponteiro para struct NODE)
    Lista *inicioLista;
    inicioLista = new Lista; // alocação dinâmica da variável inicioLista
    //Se a lista (inicioLista) foi alocada corretamente, ela apontará para NULO
    if(inicioLista != nullptr){
        *inicioLista = nullptr;
        cout << "Lista Criada com sucesso!\n";
    }
    return inicioLista;
}
```

Liberação da lista

```
void liberarLista(Lista *inicioLista){
    if(inicioLista != nullptr){ // se a lista existir
        No *no; // cria um ponteiro do tipo No
        while((*inicioLista) != nullptr){ // verifica se a lista não está vazia (se existem nós)
            no = *inicioLista; // aponta o ponteiro no para o início da lista
            *inicioLista = no->prox; // o inicio da lista avança para o próximo nó
            delete no; // apaga o nó da memória...
        } // esse processo é feito até que o inícioLista seja nulo.
        delete inicioLista; // apaga o inicioLista da memória
    }
}
```

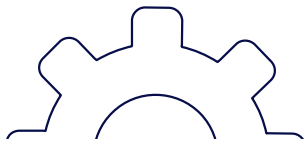
Inserção de Aluno no Início da lista

```
//função para inserção de um nó no início da lista
int inserirInicio(Lista* inicioLista, Aluno *alunoN){
    if(inicioLista == nullptr){
        cout << "Memória Insuficiente: Lista não foi alocada!\n";
        return 0;
    }
    No* novoNo;
    novoNo = new No;
    if(novoNo == nullptr){ // se a alocação não foi feita corretamente
        cout << "Memória Insuficiente: Nó (novoNo) não foi alocado!\n";
        return 0;
    }
    // atribuindo os valores do cadastro ao nó criado (novoNo)
    novoNo->aluno = *alunoN;
    novoNo->prox = nullptr;//atribuindo nulo ao prox do novoNo
    novoNo->ant=nullptr; // ###atribuindo nulo ao ant do novoNo
```



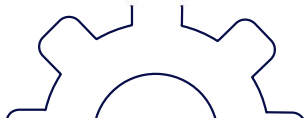
Inserção de Aluno no Início da lista

```
//se a lista estiver vazia, apontar o inicioLista para novoNo;
if(*inicioLista==nullptr){
    *inicioLista = novoNo;
    cout << "Novo nó inserido na lista com sucesso - primeiro elemento!\n";
    return 1;
}
novoNo->prox = (*inicioLista); //aponta o prox do novoNo para o inícioLista
(*inicioLista)->ant = novoNo; //###Apontando o ant do inicioLista para o novo
*inicioLista = novoNo; // apontando a cabeça da lista para o nó criado (novo
cout << "Novo nó inserido no início da lista com sucesso!\n";
return 1;
}
```



Inserção de Aluno no final da lista

```
//função para inserir um nó no fim da lista
int inserirFinal(Lista* inicioLista, Aluno *alunoN){
    if(inicioLista == nullptr){
        cout << "Lista não foi criada!\n";
        return 0;
    }
    No *novoNo; // cria variável no como um ponteiro para estrutura No
    novoNo = new No; // aloca dinamicamente um espaço na memória
    if(novoNo == nullptr){// verificar se o nó não foi alocado corretamente
        cout << "Erro na alocação na memória - nó não foi alocado!\n";
        return 0;
    }
    //valores inseridos no cadastro e são atribuidos ao novoNo
    novoNo->aluno = *alunoN;
    novoNo->prox = nullptr;
    novoNo->ant = nullptr; //##### ponteiro que aponta para o nó anterior
```



Inserção no final da lista...

```
//se a lista estiver vazia, apontar o inicioLista para novoNo;
if(*inicioLista==nullptr){
     *inicioLista = novoNo;
     cout <<"Novo nó inserido na lista com sucesso - primeiro elemento!\n";
     return 1;
}
// se a lista não estiver vazia... deverá percorrer até o último elemento
No *noAtual;  // variável noAtual é um ponteiro para estrutura No
noAtual = *inicioLista; // noAtual vai apontar  inícioLista aponta
// verifica se o próximo nó não é nulo
while(noAtual->prox != nullptr){
   noAtual = noAtual->prox; //atualiza noAtual para o proximo elemento
}// ao sair do while... significa que noAtual->prox é nulo
noAtual->prox = novoNo;  // noAtual->prox apontará para o novoNo;
novoNo->ant = noAtual; // ### novoNo->ant aponta para noAtual
cout<< "Novo nó inserido no final lista com sucesso!!\n";
return 1;
}
```

Inserção ordenada

```
//função para inserir um nó em uma lista ordenada
int inserirItem(Lista* inicioLista, Aluno *alunoN){
    if(inicioLista == nullptr){
        cout << "Memória Insuficiente: Lista não foi alocada!\n";
        return 0;
    }
    No* novoNo;
    novoNo = new No;
    if(novoNo == nullptr){ // se a alocação não foi feita corretamente
        cout << "Memória Insuficiente: Nó (novoNo) não foi alocado!\n";
        return 0;
    }
    novoNo->aluno = *alunoN;// atribuindo os valores do cadastro ao nó criado (n
    novoNo->prox = nullptr; // atribui nulo ao prox
    novoNo->ant = nullptr; //### atribui nulo ao anterior
    if((*inicioLista) == nullptr){//lista vazia, aponta o inicioLista para o nov
        *inicioLista = novoNo;
        cout << "Novo nó inserido na lista com sucesso - primeiro elemento!\n";
        return 1;
    }
```

Inserção ordenada...

```
No *noAtual;//ponteiro para percorrer a lista:noAtual->prox;noAtual->ant
noAtual = *inicioLista; // noAtual aponta o inicio da Lista

// se noAtual->mat >= a alunoN->mat não entra
// faz a repetição enquanto o noAtual for diferente null
//e a matr da lista < mat (novoNo)
while(noAtual->prox != nullptr && noAtual->aluno.matricula < alunoN->matricu
    noAtual = noAtual->prox;
}
//novoNo será inserido como primeiro elemento da lista
if(noAtual == *inicioLista && noAtual->aluno.matricula > alunoN->matricula){
    novoNo->prox = noAtual;
    noAtual->ant = novoNo;  // ###aponto o ant (noAtual) para o novoNo
    *inicioLista = novoNo;
    cout << "Novo nó inserido no início da lista com sucesso!\n";
    return 1;
}
```

Inserção ordenada

```
    //inserção no fim da lista, O atual->prox é nulo FIM DA LISTA
    if(noAtual->prox==nullptr && noAtual->aluno.matricula < alunoN->matricula){
        novoNo->ant = noAtual; // ##### NovoNo->ant aponta noAtual
        noAtual->prox = novoNo;
        cout << "Novo nó inserido no fim da lista com sucesso!\n";
        return 1;
    }

    //inserção quando não é no inicio da lista e nem no fim...
    //cuidado com a ordem pois senão pode perder o acesso do noAtual->ant
    novoNo->prox = noAtual;
    novoNo->ant = noAtual->ant; // ##### NovoNo aponta ant para o
    (noAtual->ant)->prox = novoNo;
    noAtual->ant = novoNo; //#### ant do noAtual aponta para o novoNo
    cout << "Novo nó inserido na lista com sucesso - Ordenado!\n";
    return 1;
}
```

Remoção de Aluno no início da lista

```
int removerInicio(Lista* inicioLista){
    if(inicioLista == nullptr){ //lista não foi criada corretamente
        cout << "Memória Insuficiente: Lista não foi alocada!\n";
        return 0;
    }
    if((*inicioLista) == nullptr){//lista vazia, não há o que remover
        cout << "Lista vazia!! Não há nós para remover!\n";
        return 0;
    }
    No *no = *inicioLista;
    if(no->prox==nullptr){ //único elemento, inicioLista deve apontar null
        *inicioLista=nullptr;
        cout<<"Único elemento removido com sucesso! Lista agora está vazia!";
        delete no;
        return 1;
    }
    *inicioLista = no->prox;
    (*inicioLista)->ant = nullptr;   //(no->prox)->ant = nullptr;
    delete no;
    cout << "Nó removido do início da lista com sucesso!\n";
    return 1;
}
```

Remoção de Aluno no final da lista

```
int removerFinal(Lista* inicioLista){
    if(inicioLista == nullptr){ //lista não foi criada corretamente
        cout << "Memória Insuficiente: Lista não foi alocada!\n";
        return 0;
    }
    if((*inicioLista) == nullptr){//lista vazia, não há o que remover
        cout << "Lista vazia!! Não há nós para remover!\n";
        return 0;
    }
    No *no = *inicioLista; // precisa só de um ponteiro!!!
    while(no->prox != nullptr)  // percorre até encontrar o nulo
        no = no->prox;
    if(no->prox==nullptr  && no->ant==nullptr){  //único elemento, inicioLista deve apo
        *inicioLista=nullptr;
        cout<<"Único elemento removido com sucesso! Lista agora está vazia!";
        delete no;
        return 1;
    }
   //se no foi movimentado no while... no->prox vai chegar até nulo
    (no->ant)->prox = nullptr;
    delete no; // desaloca o no...
    cout << "Nó removido do fim lista com sucesso!\n";
    return 1;
}
```

Remoção de determinado Item da lista

```
int removerItem(Lista* inicioLista, int matA){
    if(inicioLista == nullptr){ //lista não foi criada corretamente
        cout << "Memória Insuficiente: Lista não foi alocada!\n";
        return 0;
    }
    if((*inicioLista) == nullptr){//lista vazia, não há o que remover
        cout << "Lista vazia!! Não há nós para remover!\n";
        return 0;
    }

    No *no = *inicioLista;
    // percorre a lista até encontrar a matricula ou chegar no fim
    while(no != nullptr && no->aluno.matricula != matA){
        no = no->prox;
    }

    if(no == nullptr){//valor da matrícula não encontrado na lista
        cout << "Lista não contém o elemento que procura!\n";
        return 0;
    }
```

```
    //lista tem apenas o no a ser removido, o *inicioLista será nulo
    if(no->ant==nullptr && no->prox==nullptr){
        *inicioLista = nullptr;
        delete no;
        cout << "Nó encontrado e removido com sucesso - a lista está vazia!\n";
        return 1;
    }
    //nó a ser removido é o primeiro e a lista contém mais elementos.
    if(no == *inicioLista){ //ou if(no->ant == nullptr)
        *inicioLista = no->prox;  //Reposicionar inícioLista
        (no->prox)->ant = nullptr;
        delete no;
        cout << "Nó encontrado e removido da lista com sucesso - primeiro elemento!\n";
        return 1;
    }
   //remoção será no meio da lista, deve realocar os ponteiros
    if(no->prox!=nullptr){
        (no->ant)->prox = no->prox;
        (no->prox)->ant=no->ant;
    }
    else // O nó do fim da lista será removido
        (no->ant)->prox=nullptr;
    delete no;
    cout << "Nó encontrado e removido da lista com sucesso!\n";
    return 1;
}
```

Remoção de determinado Item da lista ...

Consultar Posição de um elemento na lista

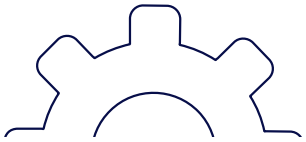
```
int consultarPosicao(Lista* inicioLista, int pos){
    if(inicioLista == nullptr){
        cout <<"Lista não criada!\n";
        return 0;
    }
    if(pos<=0){
        cout << "Posição informada inválida!\n";
        return 0;
    }
    if((*inicioLista) == nullptr){//lista vazia, não há o que consultar
        cout << "Lista vazia!!\n";
        return 0;
    }
    No *noAux = *inicioLista;
    int i = 1;
    while(noAux != nullptr && i < pos){
        noAux = noAux->prox;
        i++;
    }
    if(noAux == nullptr){
        cout << "Não há elementos na posição solicitada\n";
        return 0;
    }
    cout<<"\nDados do aluno na posição "<< pos;
    cout<< "\nNome: " << noAux->aluno.nome;
    cout<< "\nMatrícula: " << noAux->aluno.matricula;
    cout<< "\nNota: " << noAux->aluno.nota << endl;
    return 1;
`
```

Consultar um elemento na lista pela matrícula

```
int consultarMatricula(Lista* inicioLista, int mat){
    if(inicioLista == nullptr){
        cout <<"Lista não criada!\n";
        return 0;
    }
     if((*inicioLista) == nullptr){//lista vazia, não há o que consultar
        cout << "Lista vazia!!\n";
        return 0;
    }
    No *noAux = *inicioLista;
    while(noAux != nullptr && noAux->aluno.matricula != mat){
       noAux = noAux->prox;
    }
    if(noAux == nullptr){
        cout << "Matrícula não encontrada na lista!\n";
        return 0;
    }
    cout<<"\nDados do aluno de matrícula "<< mat;
    cout<< "\nNome: " << noAux->aluno.nome;
    cout<< "\nMatrícula: " << noAux->aluno.matricula;
    cout<< "\nNota: " << noAux->aluno.nota << endl;
    return 1;
}
```

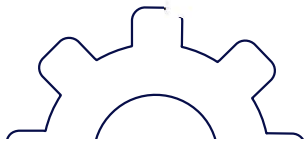
Exibir dados da Lista

```
void imprimirLista(Lista *inicioLista){
    No *noAux = *inicioLista;
    if(noAux==nullptr)
      cout << "Não há dados cadastrados na lista!\n";
    else{
          while(noAux!=nullptr) {
              cout<< "\nNome: " << noAux->aluno.nome;
              cout<< "\nMatrícula: " << noAux->aluno.matricula;
              cout<< "\nNota: " << noAux->aluno.nota << endl;
              noAux = noAux->prox;
          }
     }
}
```



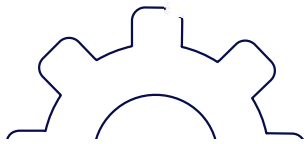
Exibir dados da Lista – inversa

```
void imprimirListaInversa(Lista *inicioLista){
    No *noAux = *inicioLista;
    if(noAux==nullptr)
      cout << "Não há dados cadastrados na lista!\n";
    else{
         while(noAux->prox!=nullptr)
              noAux = noAux->prox;
        do{
              cout<< "\nNome: " << noAux->aluno.nome;
              cout<< "\nMatrícula: " << noAux->aluno.matricula;
              cout<< "\nNota: " << noAux->aluno.nota << endl;
              noAux = noAux->ant;
         } while(noAux!=nullptr);
     }
}
```



Exibir dados da Lista – inversa

```
void imprimirListaInversa(Lista *inicioLista){
    No *noAux = *inicioLista;
    if(noAux==nullptr)
      cout << "Não há dados cadastrados na lista!\n";
    else{
         while(noAux->prox!=nullptr)
              noAux = noAux->prox;
        do{
              cout<< "\nNome: " << noAux->aluno.nome;
              cout<< "\nMatrícula: " << noAux->aluno.matricula;
              cout<< "\nNota: " << noAux->aluno.nota << endl;
              noAux = noAux->ant;
         } while(noAux!=nullptr);
     }
}
```



Referências

EDELWEISS, N,; GALANTE, R.. Estruturas de dados. Porto Alegre: Bookman, 2009.

SZWARCFITER, J. L.; MARKENZON, L. Estruturas de dados e seus algoritmos. 3. ed. Rio de Janeiro: LTC, 2010.

ZIVIANI, N. Projeto de algoritmos: com implementações em Pascal e C. 3. ed. rev. e ampl. São Paulo: Cengage Learning, 2011.

Aulas e vídeo aulas do professor André Backes: https://www.facom.ufu.br/~backes/