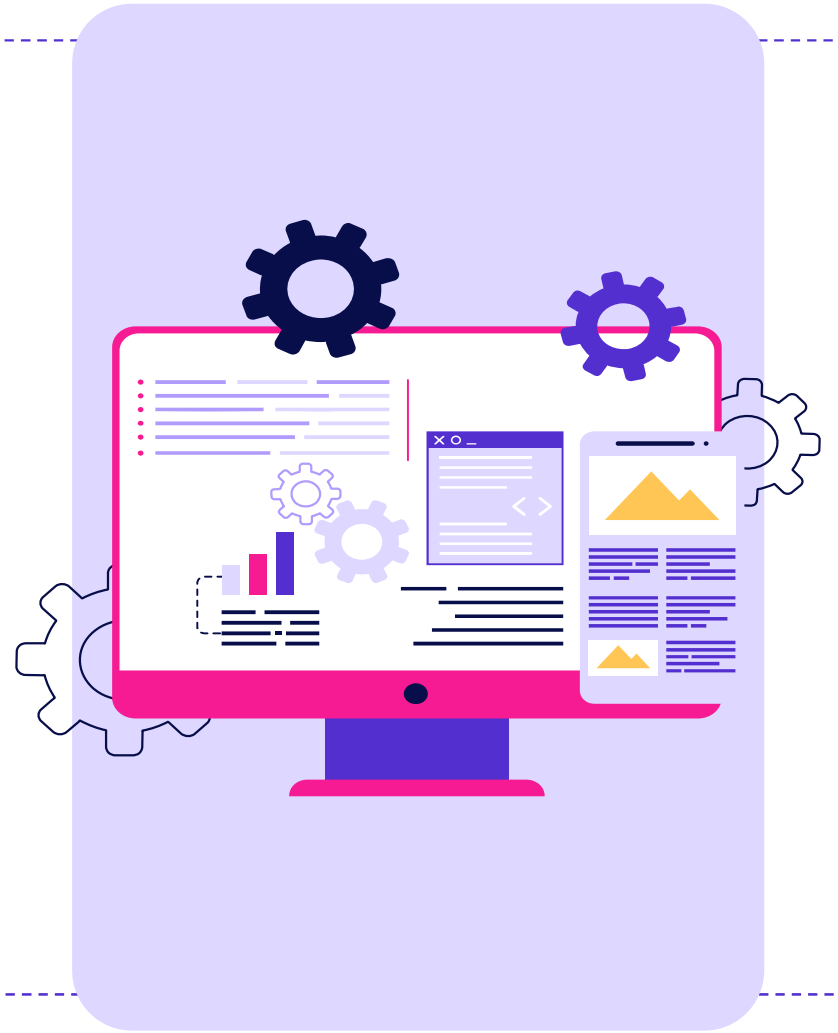
Estrutura de Dados 1

LISTA DINÂMICA



Profª Juliana Franciscani

Roteiro

01 O que é

02 TADs

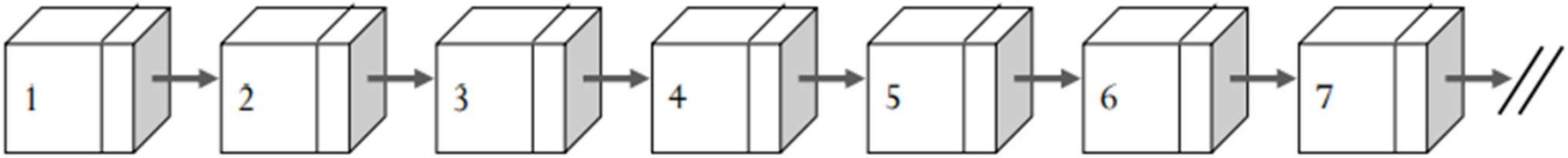
03 Condições (pré/pós)

04 Código

05 Exercícios

LISTA DINÂMICA

- Utiliza de Alocação dinâmica
- Aumenta e diminui em tempo de execução
- É uma estrutura encadeada, nós são interligados de acordo com a inserção.



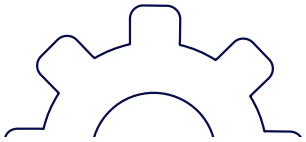
Principal vantagem em relação a lista estática é o ganho em desempenho em termos de velocidade nas inclusões e remoções de elementos.

LISTA DINÂMICA

- Criar lista;
- Apagar lista;
- Inserir item;
- Acessar item (dado uma chave);
- Remover item (dado uma chave);
- Contar número de itens;
- Verificar se a lista está vazia;
- Copiar lista;
- Imprimir lista.

Lista Dinâmica

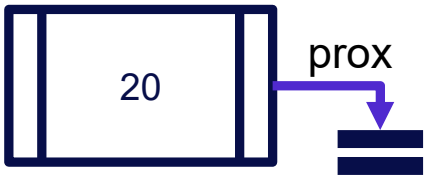
- A inserção ou remoção de um elemento não altera a posição dos outros elementos.
- Não é necessário definir o número máximo de elementos que a lista poderá ter.
- A memória é alocada dinamicamente, apenas para o número de nós necessários.
- A velocidade nas inclusões e remoções de elementos é maior do que nas listas sequenciais.



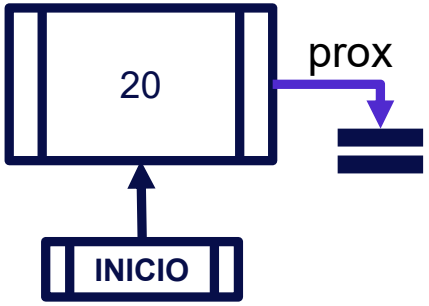
LISTA ENCADEADA – Inserção

- Inserção em uma lista vazia: Valor 20

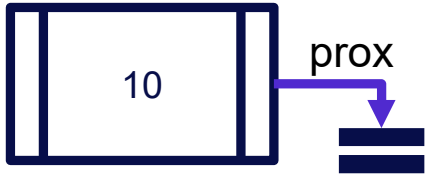
Lista



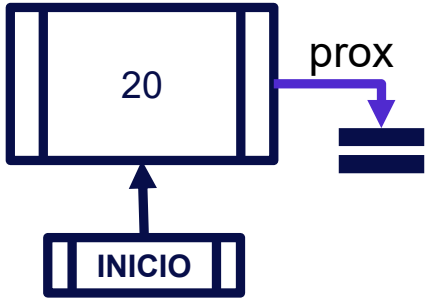
novoNo



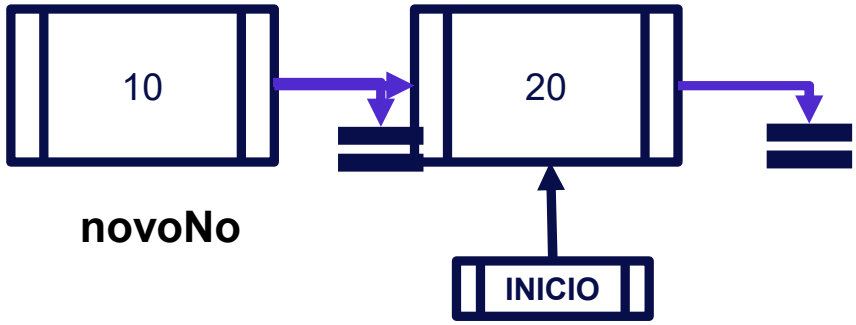
- Inserção no início em uma lista que já possui elemento: 10



novoNo



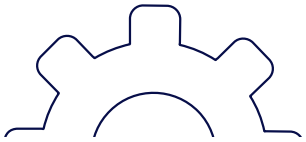
Lista



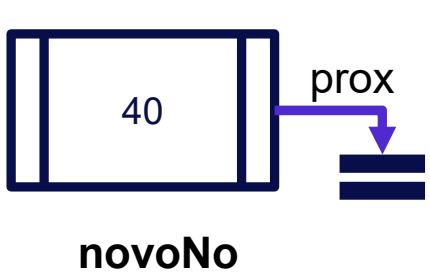
novoNo

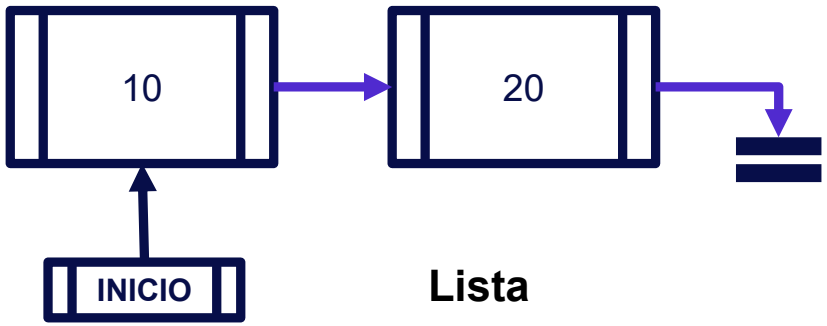
novoNo-> prox = *inicoLista

*inicioLista = novoNo

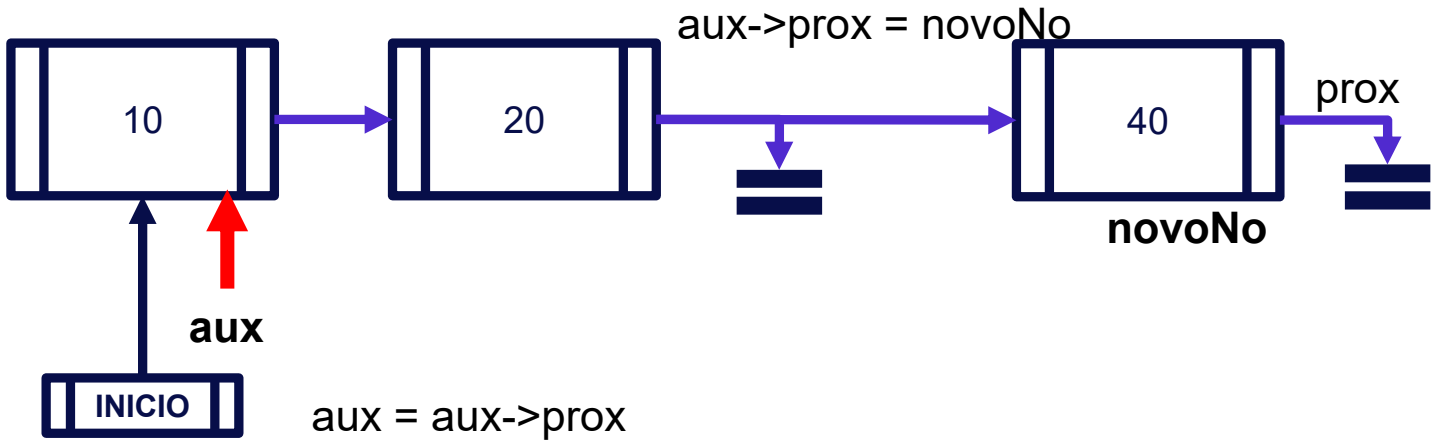


- Inserção no fim em uma lista que já possui elementos: 40

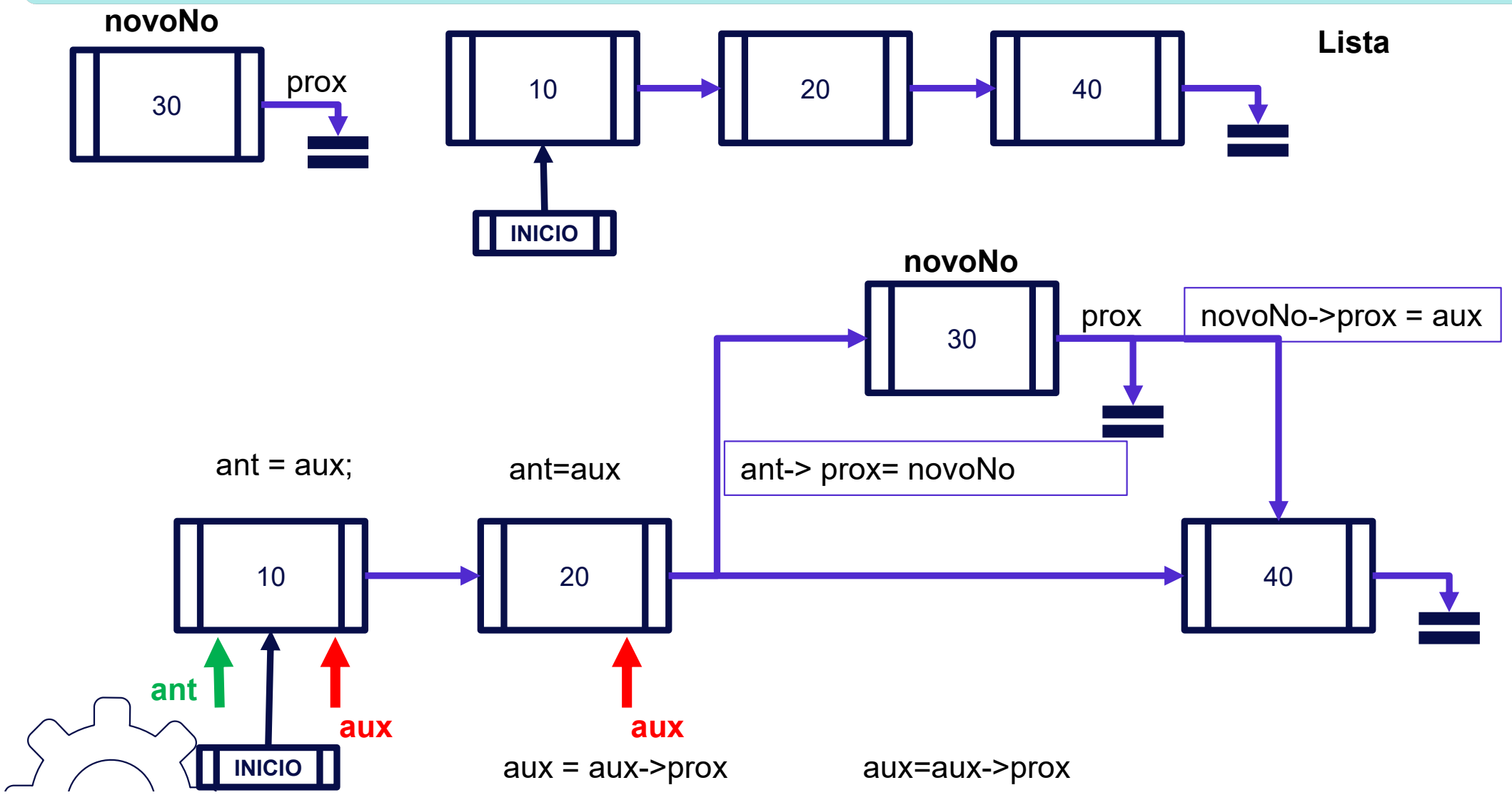




aux->prox é diferente de Nulo?



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



.cpp

```
Lista* criarLista(){
    Lista *inicioLista; //ponteiro apontará pa
                        //(que já é um pontei
    inicioLista = new Lista; // alocação dinâm
    if(inicioLista != nullptr){ //Se a lista (
        *inicioLista = nullptr; // inicioLista
        cout << "Lista Criada com sucesso!\n";
    }
    return inicioLista;
}
```

.h

```
struct ALUNO{
    int matricula;
    char nome[50];
    float nota;
};
typedef struct ALUNO Aluno;

struct NODE{
    Aluno aluno;
    struct NODE *prox;
};

typedef struct NODE *Lista;
typedef struct NODE No;

void cadastrarAluno(Aluno *alunoN);
Lista* criarLista();
```

Main

```
Aluno alunoN;
Lista *lista;
lista = criarLista();
```

.cpp

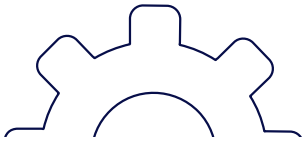
```
void liberarLista(Lista *inicioLista){
    if(inicioLista != nullptr){ // se a lista existir
        No *no; // cria um ponteiro do tipo No
        while((*inicioLista) != nullptr){ // verifica se a lista não está vazia
            no = *inicioLista; // aponta o ponteiro no para o início da lista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

.h

```
void liberarLista(Lista *lista);
```

Main

```
liberarLista(lista);
return 0;
```



.cpp

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;
    }
    //se o nó foi alocado na memória corretamente, daí pega os valores inseridos no cadastro e atribui a ele
    novoNo->aluno = *alunoN;
    novoNo->prox = nullptr;
    if((*inicioLista) == nullptr){//lista vazia
        *inicioLista = novoNo; // o inícioLista aponta para o nó criado
        cout << "Nó inserido na lista vazia!!\n";
        return 1;
    }
 // se a lista não estiver vazia... deverá percorrer até o último elemento
    No *aux;  // cria uma variável aux que é um ponteiro para estrutura No, para percorrer a lista
    aux = *inicioLista; // aux vai apontar para onde o inícioLista aponta
    while(aux->prox != nullptr){ // verifica se o próximo nó é nulo, se não for
       aux = aux->prox; // atualiza o ponteiro aux para o proximo elemento
    }// ao sair do while... significa que aux->prox é nulo
    aux->prox = novoNo;  // agora aux->prox apontará para o novo no;
    cout<< "Nó inserido na lista!!\n";
    return 1;
}
```

Inserção 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 = (*inicioLista); // apontando o prox do nó criado (novoNo) para o início
    *inicioLista = novoNo; // apontando a cabeça da lista para o nó criado (novoNo)
    cout << "Novo nó foi inserido na lista com sucesso!\n";
    return 1;
}
```

Inserção Ordenado Aluno Ordenado

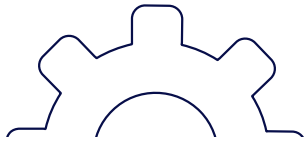
```
//função para inserir um nó em uma lista ordenada
int inserirItem(Lista* inicioLista, Aluno *alunoN){
    if(inicioLista == nullptr){
        cout << "Memória Insuficiente: Lista não foi alocada!\n";
        return 0;
    }

    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 (novoNo)
    novoNo->prox = nullptr;
    if((*inicioLista) == nullptr){//lista vazia, atribui nulo para o novoNo->prox e aponta o inicioLista para o novoN
        *inicioLista = novoNo;
        cout << "Novo nó foi inserido na lista com sucesso!\n";
        return 1;
    }

    No *noAnt, *noAtual; //cria dois ponteiros para que esses percorram a lista
    noAtual = *inicioLista; // noAtual aponta para o endereço que o inicioLista está indicando
```

Inserção Ordenado Aluno Ordenado cont.

```
// se o noAtual for nulo não entra na repetição
// se a matricula do aluno que está no noAtual também for >= a matrícula do novoAluno também não entra na repetição
while(noAtual != nullptr && noAtual->aluno.matricula < alunoN->matricula){
    noAnt = noAtual;
    noAtual = noAtual->prox;
} // faz a repetição enquanto o noAtual não for null e a matricula da lista seja menor que a matricula (novoNo) não

//verifica se o noAtual está no inicio da lista, pois se sim a inserção será no início
if(noAtual == *inicioLista){//insere início
    novoNo->prox = (*inicioLista);
    *inicioLista = novoNo;
    cout << "Novo nó foi inserido na lista com sucesso!\n";
    return 1;
}

//inserção quando não é no inicio da lista, necessário os dois ponteiros...
noAnt->prox = novoNo;
novoNo->prox = noAtual;
cout << "Novo nó foi inserido na lista com sucesso!\n";
return 1;

}
```

Remoção 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;
    *inicioLista = no->prox;

    delete no;
    cout << "Primeiro nó removido na lista com sucesso!\n";
    return 1;
}
```



Remoção 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 *noAnt, *noAux = *inicioLista;
     while(noAnt->prox != nullptr){
         noAnt = noAux;
         noAux = noAux->prox;
     } // percorre até encontrar o nulo

     if(noAux == (*inicioLista)) //se houver um elemento só na lista, noAux vai estar apontando para o
         *inicioLista = noAux->prox; // atribui nulo para o inicioLista (lista vazia)
     else //se noAux foi movimentado no while... noAux->prox vai chegar até nulo
         noAnt->prox = nullptr; // ou noAnt->prox=noAux->prox;

     delete noAux; // desaloca o noAux...
     cout << "Último nó removido na lista com sucesso!\n";
     return 1;
}
```

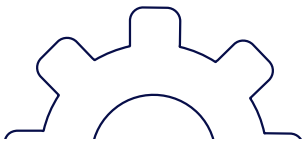
Remoção de um item da Lista

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

    if(noAux == nullptr){//valor da matrícula não encontrado na lista
       cout << "Lista não contém o elemento que procura
        return 0;
    }
    if(noAux == *inicioLista)
        *inicioLista = noAux->prox;
    else
        noAnt->prox = noAux->prox; //vai fazer a realocação dos ponteiros para poder remover o elemento.
    delete noAux;
    cout << "Nó encontrado e removido da lista com sucesso!\n";
    return 1;
}
```

concentrate, concentration, contravene

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;
         }
    }
}
```



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/