Estruturas de Dados I

Listas Ligadas

Prof. Bruno Azevedo

Instituto Federal de São Paulo

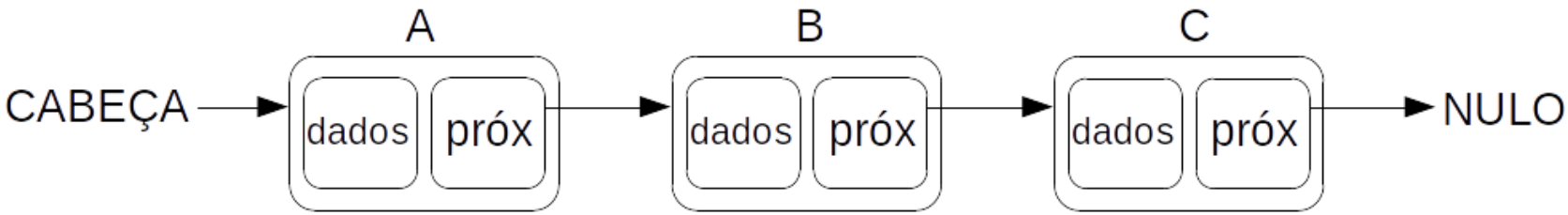
INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Catanduva

Lista Ligada

- Uma **lista ligada** é uma estrutura de dados na qual os elementos estão organizados em ordem linear. Também é conhecida como lista encadeada.
- Esta ordem linear não é determinada por índices sequenciais, como em um vetor, mas por **ponteiros** para o próximo objeto.
- Listas ligadas oferecem uma representação simples e flexível para conjuntos dinâmicos de dados.
- Pode possuir diversas formas: circular, ordenada, singularmente ou duplamente ligada.
- Essas formas não são necessariamente exclusivas. Ex: duplamente ligada e ordenada.

Lista Ligada

- Em uma lista singularmente ligada, cada elemento é um objeto contendo dados e um ponteiro que aponta para o próximo elemento da lista.



- Temos um ponteiro inicial (CABEÇA) que aponta para o primeiro elemento da lista.
- O ponteiro do último elemento aponta para NULL.

Lista Ligada

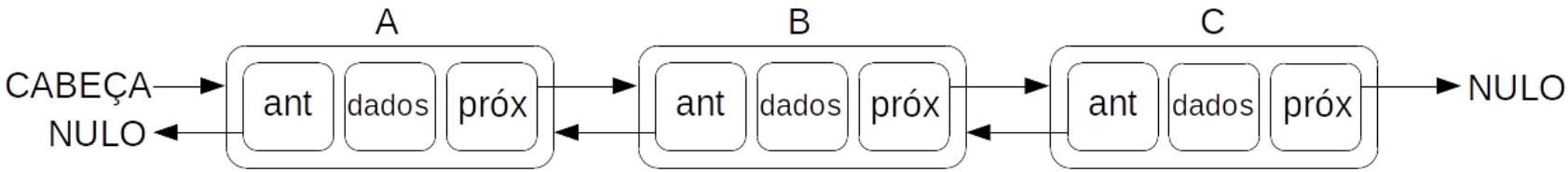
- Uma implementação típica de um nó da lista singularmente ligada usando struct.

```
struct No {
    int chave;
    No* prox;
};
```

- Temos um campo representando a chave. Podemos ter outros campos de dados.
- Temos um campo que identifica o próximo nó da lista. Se este for NULL, significa que este é o último nó da lista.
- Ou seja, cada nó está **ligado** com o próximo nó. Podem imaginar uma corda conectando cada nó ao próximo.
- Entretanto, um nó apenas conhece o próximo nó, ele não conhece o nó anterior.

Lista Duplamente Ligada

- A **lista duplamente ligada** é uma lista ligada onde cada nó contém dados e dois ponteiros.
- Um ponteiro apontando para o próximo nó, e um ponteiro apontando para o nó anterior.



- Como na lista singularmente ligada, temos um ponteiro CABEÇA que aponta para o primeiro nó da lista e o ponteiro 'prox' do último nó aponta para NULL.
- O ponteiro 'ant' do primeiro nó aponta para NULL (não existem elementos anteriores).

Lista Duplamente Ligada

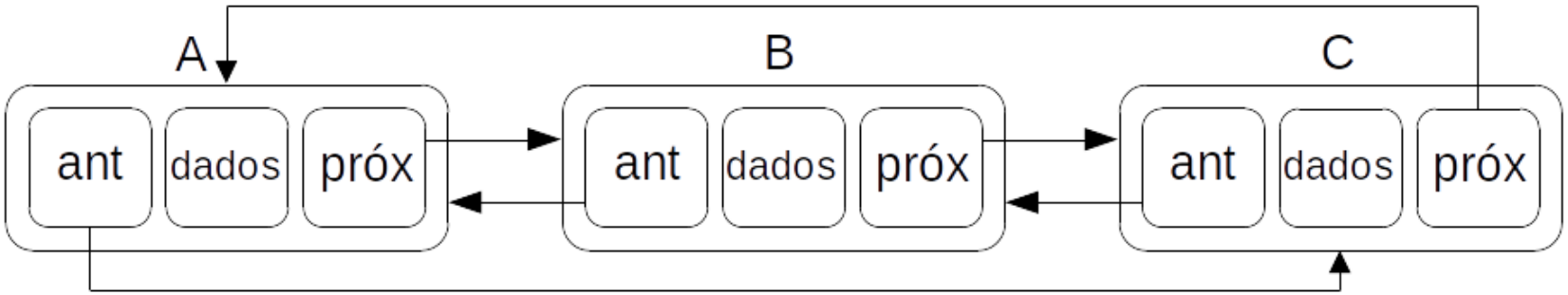
- Uma implementação típica de um nó da lista duplamente ligada usando struct.

```
struct No {
    int chave;
    No *ant, *prox;
};
```

- Na lista duplamente ligada, cada nó conhece seu sucessor e seu antecessor.

Lista Duplamente Ligada Circular

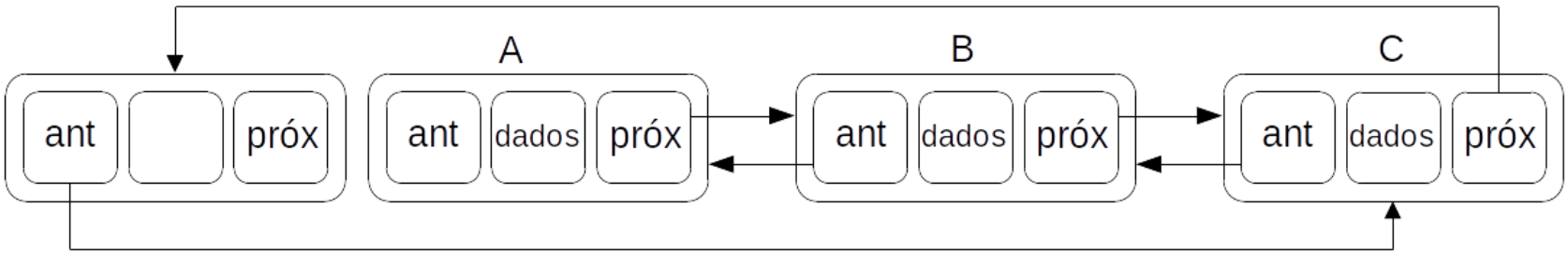
- A **lista duplamente ligada circular** é uma lista duplamente ligada que forma um ciclo, onde o último nó aponta para o primeiro nó e o primeiro nó aponta para o último nó, tornando-a circular.



- A implementação de um nó é a mesma da lista duplamente ligada. Um nó conterá dados e dois ponteiros.

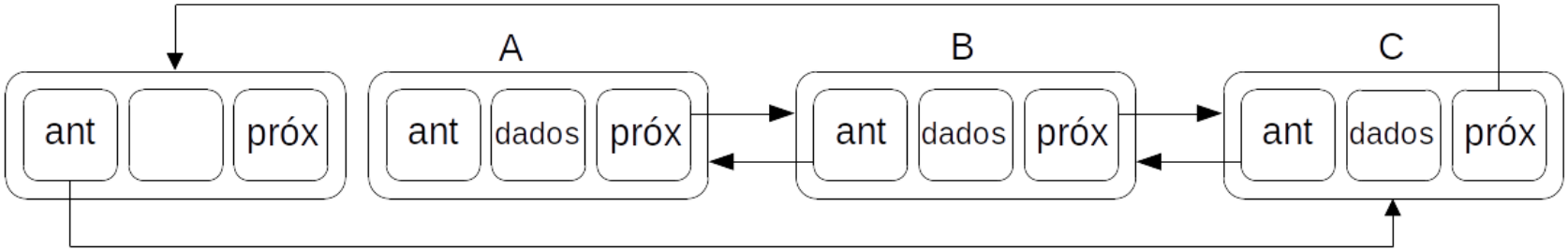
Lista Duplamente Ligada Circular com Sentinela

- Também podemos implementar a lista duplamente ligada circular usando uma **sentinela**.



- Uma **sentinela** é um objeto que permite simplificar a manipulação da lista.
- Podemos ter sentinelas de cabeça, de cauda, ou que conecte ambas, tornando a lista circular. Esta é a sentinela exemplificada na figura acima.

Sentinelas



- Uma sentinela de cabeça é um nó colocado no início da lista, antes do primeiro elemento.
- Uma sentinela de cauda é um nó colocado no fim da lista.
- A sentinela exemplificada, possui ambas as propriedades, já que é uma lista circular.
- Além de deixar o código mais simples, sentinelas também podem melhorar a eficiência de algumas operações, como por exemplo, a inserção no final da lista.

Revisando Operações em Conjuntos Dinâmicos de Dados

- Como já aprendemos, existem diversas operações que podemos querer efetuar em conjuntos dinâmicos de dados.
- Busca, Inserção, Remoção, Mínimo, Máximo, Sucessor, Predecessor, etc.
- Vamos pensar em como efetuar algumas operações básicas em listas ligadas. Mais especificamente, em **listas duplamente ligadas não-ordenadas**.

Busca em Listas Duplamente Ligadas

- A função **busca** retorna um ponteiro para o primeiro elemento com o valor 'chave' na lista ligada.
- Caso o valor não exista, retornará NULL.

```
No* busca(No* cabeca, int chave) {
    No* atual = cabeca;
    while((atual != nullptr) && (atual->chave != chave))
        atual = atual->next;
    return atual;
}
```

- Qual o pior caso, considerando uma lista de n elementos?

Busca em Listas Duplamente Ligadas

- A função **busca** retorna um ponteiro para o primeiro elemento com o valor 'chave' na lista ligada.
- Caso o valor não exista, retornará NULL.

```
No* busca(No* cabeca, int chave) {
    No* atual = cabeca;
    while((atual != nullptr) && (atual->chave != chave))
        atual = atual->next;
    return atual;
}
```

- Qual o pior caso, considerando uma lista de n elementos? Teremos que testar a condição do while $n + 1$ vezes, percorrendo a lista inteira e verificando quando o ponteiro for NULL na última vez.

Inserção em Listas Duplamente Ligadas

- A operação de inserção é feita inserindo um novo nó, com uma chave já definida na lista.
- Como a lista em questão é não-ordenada, a operação é simples e rápida.

```
void insereNo(No** cabeca, No* novo) {
    novo->prox = *cabeca;
    if(*cabeca != nullptr)
        (*cabeca)->ant = novo;
    *cabeca = novo;
    novo->ant = nullptr;
}
```

- Não precisaremos definir o novo->ant como nullptr se já o fizemos na criação do nó. Caso contrário, isso será necessário.
- Qual o pior caso, considerando uma lista já contendo n elementos?

Inserção em Listas Duplamente Ligadas

- A operação de inserção é feita inserindo um novo nó, com uma chave já definida na lista.
- Como a lista em questão é não-ordenada, a operação é simples e rápida.

```
void insereNo(No** cabeca, No* novo) {
    novo->prox = *cabeca;
    if(*cabeca != nullptr)
        (*cabeca)->ant = novo;
    *cabeca = novo;
    novo->ant = nullptr;
}
```

- Não precisaremos definir o novo->ant como nullptr se já o fizemos na criação do nó. Caso contrário, isso será necessário.
- Qual o pior caso, considerando uma lista já contendo n elementos? Tempo constante, sempre inserimos na primeira posição da lista.

Remoção em Listas Duplamente Ligadas

- A operação de remoção remove um nó da lista.
- Assumimos já termos um ponteiro para este nó.
- Caso contrário, precisaremos utilizar a função **busca** para obter essa referência.

```
void removeNo(No** cabeca, No* noARemover) {
    if(noARemover->ant != nullptr) // Se não é a cabeça
        noARemover->ant->prox = noARemover->prox;
    else
        *cabeca = noARemover->prox;
    if(noARemover->prox != nullptr)
        noARemover->prox->ant = noARemover->ant;
}
```

- Qual o pior caso, considerando uma lista já contendo n elementos?

Remoção em Listas Duplamente Ligadas

- A operação de remoção remove um nó da lista.
- Assumimos já termos um ponteiro para este nó.
- Caso contrário, precisaremos utilizar a função **busca** para obter essa referência.

```
void removeNo(No** cabeca, No* noARemover) {
    if(noARemover->ant != nullptr) // Se não é a cabeça
        noARemover->ant->prox = noARemover->prox;
    else
        *cabeca = noARemover->prox;
    if(noARemover->prox != nullptr)
        noARemover->prox->ant = noARemover->ant;
}
```

- Qual o pior caso, considerando uma lista já contendo n elementos? Tempo constante, já temos a referência desse nó.
- Caso seja necessário efetuar a busca pelo nó, utilizando a função **busca**, será o pior caso desta função (n + 1).

Remoção e Inserção em Listas Duplamente Ligadas com Sentinela

- Notem que se utilizarmos uma sentinela, o código de remoção é simplificado.
- Como existe a sentinela, não precisamos verificar se existe um próximo nó e um nó anterior.

```
void removeNo(No* noARemover) {
    noARemover->ant->prox = noARemover->prox;
    noARemover->prox->ant = noARemover->ant;
}
```

- Da mesma forma, o código de inserção também é simplificado.
- Inserimos no início, logo após a sentinela.

```
void insereNo(No* sentinela, No* novo) {
    novo->prox = sentinela->prox;
    sentinela->prox->ant = novo;
    sentinela->prox = novo
    novo->ant = sentinela;
}
```

Comparando a Eficiência com o Vetor

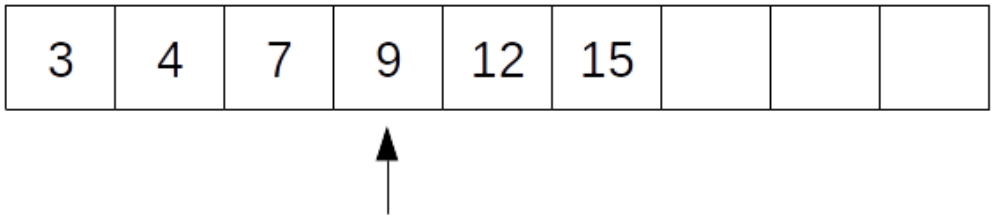
- O quanto eficiente é a lista ligada, comparada a um vetor, considerando a operação de inserção de um elemento?
- Nos exemplos anteriores, utilizamos uma lista duplamente ligada não-ordenada.
- Vamos tornar isso mais interessante e utilizar uma **lista duplamente ligada ordenada**.
- Ou seja, o elemento precisa ser inserido na posição correta.

Comparando a Eficiência com o Vetor

- É evidente que se tivermos que inserir no fim da lista, ou do vetor, o custo computacional será basicamente o mesmo.
- Teremos que efetuar $\sim n$ comparações em ambos os casos.
- Portanto, iremos avaliar o custo se o elemento precisar ser inserido na primeira posição e no meio da estrutura.

Comparando a Eficiência com o Vetor (Inserção no Meio)

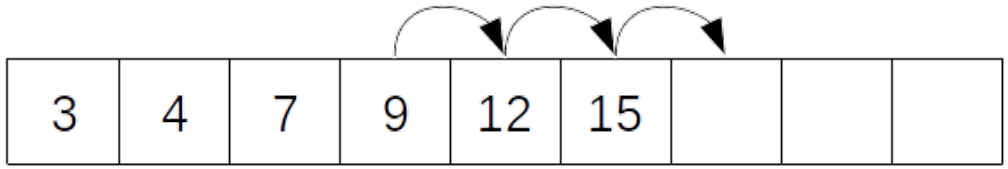
- Por exemplo, suponha que iremos inserir o valor 8 neste vetor abaixo.



- Precisamos comparar $\frac{n}{2} + 1$ vezes para encontrar a posição correta.

Comparando a Eficiência com o Vetor (Inserção no Meio)

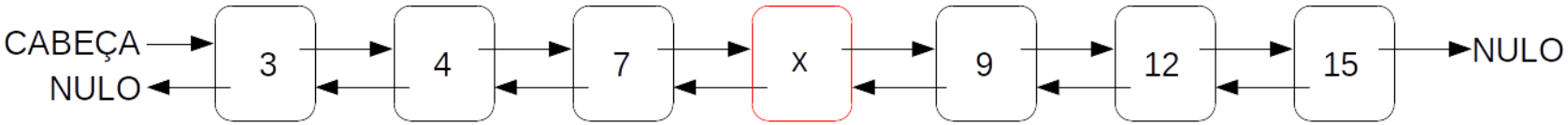
- Por exemplo, suponha que iremos inserir o valor 8 neste vetor abaixo.



- Precisamos copiar $\frac{n}{2}$ vezes para liberar o índice em que o elemento será inserido.
- Ou seja, precisamos de $\frac{n}{2} + 1$ comparações e $\frac{n}{2}$ cópias.

Comparando a Eficiência com o Vetor (Inserção no Meio)

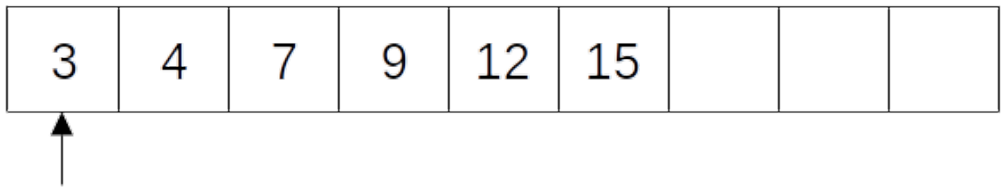
- Já em uma lista duplamente ligada, precisaremos comparar $\frac{n}{2} + 1$ vezes para encontrar a posição correta.



- Mas não será necessário efetuar nenhuma operação de cópia.
- Ou seja, neste caso particular, a lista ligada se mostra mais **eficiente**.

Comparando a Eficiência com o Vetor (Inserção no Início)

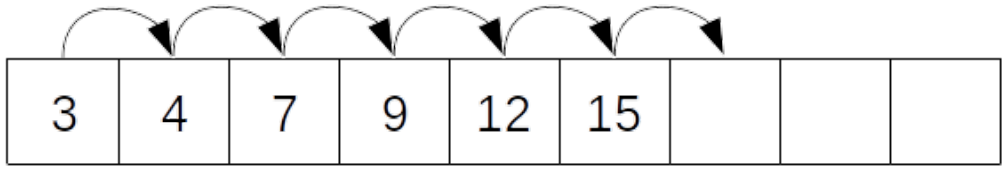
- Agora vamos utilizar um número que precisará ser inserido na primeira posição da estrutura.
- Por exemplo, suponha que iremos inserir o valor 1 neste vetor abaixo.



- Precisamos comparar apenas uma vez para encontrar a posição correta.

Comparando a Eficiência com o Vetor (Inserção no Início)

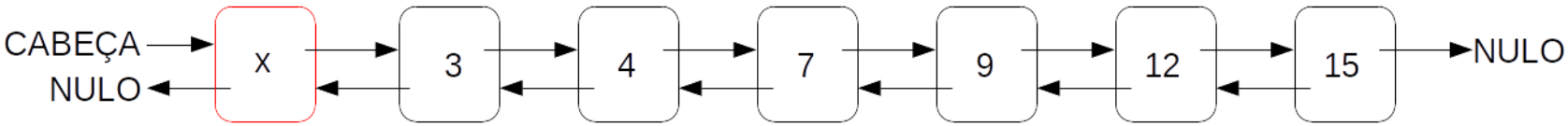
- Por exemplo, suponha que iremos inserir o valor 1 neste vetor abaixo.



- Precisamos comparar apenas uma vez para encontrar a posição correta.
- Mas serão efetuadas n cópias para liberar o índice da primeira posição.

Comparando a Eficiência com o Vetor (Inserção no Início)

- E uma lista duplamente ligada precisaremos comparar uma vez para encontrar a posição correta.



- Mas não será necessário efetuar nenhuma operação de cópia.
- Ou seja, neste caso particular, a lista ligada também se mostra mais **eficiente**.

Comparando a Eficiência com o Vetor (Busca)

- Vamos comparar a operação de busca.
- Como dissemos no início dessa discussão, a inserção de um elemento na última posição possui basicamente o mesmo custo para ambas estruturas.
- A inserção, neste caso, utilizaria a operação de busca efetuando comparações para encontrar a posição correta.
- Portanto, **no pior caso**, em ambas as estruturas ordenadas, precisamos percorrer ela inteira ($\sim n$ comparações) até descobrirmos que o elemento deve ser inserido depois da última posição.
- Podemos então assumir que são igualmente eficientes quando se trata de buscar um elemento?

Comparando a Eficiência com o Vetor (Busca)

- Vocês talvez lembrem da minha analogia de **caixa de ferramentas**.
- Vocês precisam possuir uma caixa de ferramentas grande, e saber utilizar as ferramentas que possuem.
- Mas eu também disse que as ferramentas não são apenas estruturas de dados, mas também **algoritmos** que utilizarão junto com as estruturas de dados.

Comparando a Eficiência com o Vetor (Busca)

- Por exemplo, vocês aprenderão sobre a **busca binária** que é capaz de encontrar um elemento em um vetor ordenado com custo de $\log_2 n$.
- Portanto, se não utilizarmos a abordagem ingênua de percorrer o vetor, mas utilizarmos o algoritmo de busca binária, temos um crescimento logarítimico versus um crescimento linear na busca da lista ligada (na qual não poderemos utilizar a busca binária de modo eficiente).
- Ou seja, o vetor não é "pior" que a lista ligada e "perde em todas as comparações".
- Temos que usar as ferramentas certas para o problema que temos em questão: estruturas e algoritmos.

Exercícios

- Utilize os códigos apresentados nos slides anteriores e implemente a lista duplamente encadeada. Crie um main onde fará inserções e remoções de elementos.
- Implemente a operação de inserção e de remoção em uma lista singularmente ligada. Devem executar em tempo constante.
- Implemente uma pilha usando lista ligada.
- Implemente uma fila usando lista ligada.
- (Desafio, mas nem tanto) Implemente uma função não-recursiva que inverta uma lista ligada simples de n elementos. Você consegue fazer essa operação com complexidade O(n)?
- Implemente a função anterior, agora sendo recursiva.