# Estrutura de Dados

## 1 Introdução

Na primeira aula vimos rapidamente duas `estruturas de dados` : `listas` e `Strings` . A partir delas, e também do próprio nome do conteúdo, podemos inferir o conceito básico (adaptado da wikipedia):

> Uma **estruturas de dados** é uma coleção tanto de valores (e seus relacionamentos) quanto de operações (sobre os valores e estruturas correspondentes). É uma implementação concreta de um `tipo abstrato de dados` ou um `tipo de dado` básico ou primitivo.

Ainda na primeira aula vimos os principais `tipos primitivos` que são implementados na maioria das linguagens de programação. Na `aula 03` vimos também sobre `estruturas de dados abstratas` , as quais correspondem aos `tipos abstratos de dados` . Porém, além dos tipos abstratos, temos também formas de estruturar esses dados. As principais estruturas são: `vetor` (ou *array*), `lista` , `pilha` , `fila` e `árvore` .

A partir da conceituação que tivemos, essas estruturas normalmente são acompanhadas de operações sobre elas mesmas. Por exemplo:

- Regras para a adição ou remoção de um elemento.
- Maneiras de se percorrer a estrutura.

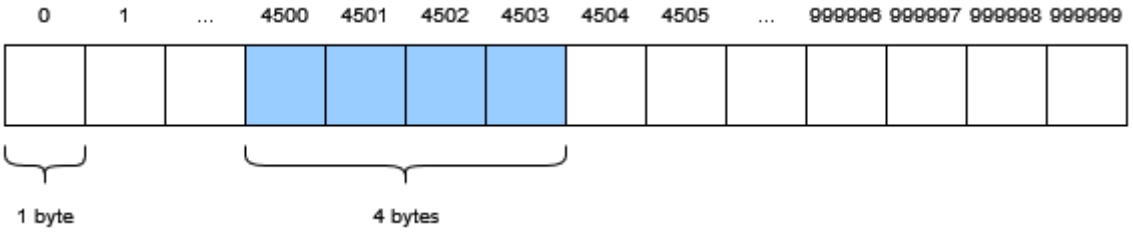
Na última aula, vimos superficialmente algumas dessas estruturas, e até utilizamos em exemplos o `ArrayList` e o `HashMap` . Em um curso de `Ciência da Computação` é necessário estudar sobre essas estruturas de forma mais aprofundada e, por isso, comumente se usa a linguagem de programação `C` . Porém, como estamos em um curso de `Análise e Desenvolvimento de Sistemas` , nosso foco está em conhecer as principais estruturas, saber como funcionam e como utilizá-las. Por isso, continuaremos usando o `Java` .

## 1.1 Conceitos para lembrar

Existem alguns conceitos básicos que já vimos e precisamos lembrar. O primeiro deles é do que realmente se trata uma `variável`. Grosso modo, uma `variável` é um **espaço de memória** `alocado` (ou seja, destinado/reservado) para que um valor de um determinado tipo seja armazenado.

Cada linguagem tem as especificações do quanto cada tipo de dado ocupa na memória em `bytes`. Por exemplo, um valor `inteiro com sinal` pode requisitar 4 `bytes` para ser armazenado. Isto significa que uma `variável` do tipo `inteiro com sinal` pode armazenar um valor qualquer entre -2.147.483.648 a 2.147.483.647, incluindo o 0. Porém, se a `variável` for do tipo `inteiro sem sinal`, os valores possíveis em 4 `bytes` variam de 0 a 4.294.967.295.

Além do valor, uma `variável` também tem outras informações, como seu nome e `endereço`. Esse `endereço` é um número que diz onde o primeiro `byte` da `variável` se encontra na memória. Digamos que temos uma memória feita de 1 milhão de células, e cada célula tenha 1 `byte`. Os endereços possíveis variam de 0 a 999.999. Se declaramos uma variável inteira que requisite 4 `bytes`, e seu endereço seja 4500, então essa variável vai ocupar as células 4500, 4501, 4502 e 4503. O `endereço`, entretanto, corresponde ao **primeiro *byte***. O mesmo princípio se aplica a qualquer outro tipo de dado.




De forma similar, uma `estrutura de dados` vai consistir de uma maneira de armazenar um conjunto de dados na `memória`, com suas respectivas formas de acesso e manipulação. Ainda, é importante lembrar também que cada linguagem de programação também possui um conjunto de `estruturas de dados` nativamente implementado. Para sabermos quais são, e como utilizá-las, precisamos recorrer à documentação da linguagem.

# 2 Vetor

Um `vetor` é uma estrutura de dados que armazena uma coleção de elementos de tal forma que cada um dos elementos possa ser identificado por, pelo menos um índice. Por padrão um `vetor` é uma `estrutura de dados unidimensional`, já que possui apenas um único índice para cada elemento. Quando os elementos possuem dois ou mais índices, então temos uma `matriz`.

O `vetor` é a `estrutura de dados` mais simples, geralmente homogêneas, ou seja, do mesmo tipo de dados. Além disso, é uma estrutura de tamanho fixo, ou imutável. Isso significa que, quando é declarado, o `compilador` ou `interpretador` irá alocar a quantidade de memória requisitada, nem mais, nem menos. Se o `vetor` não estiver sendo usado por completo, um `vetor` menor pode ser criado, e os elementos do primeiro são copiados no segundo. Da mesma forma, caso seja necessário mais espaço, um novo `vetor` com mais memória alocada é criado e os elementos do primeiro são copiados no segundo. Exemplo: ver `classe Exemplo01` no pacote `aula08.codigos.exemplos.vetor`.

Uma vantagem do `vetor` está na busca de elementos, pois não há necessidade de se procurar elemento por elemento, caso se saiba de antemão o índice a ser buscado.

# 3 Lista

Uma `lista` é uma `estrutura de dados` que consiste em uma `sequência` de zero ou mais itens, os quais podem estar ordenados ou não. Existem alguns tipos de implementação geral de listas:

- Estática sequencial.
- Estática encadeada.
- Dinâmica encadeada simples.
- Dinâmica duplamente encadeada.

O que todas as implementações têm em comum são algumas operações:

- Iniciar a lista.
- Inserir elemento (ao fim da lista).

- Remover elemento (e reorganizar a lista caso o elemento removido não seja o último).
- Buscar/consultar elemento.

## 3.1 Lista estática sequencial

Uma `lista estática sequencial` é uma lista com um tamanho (ou quantidade de elementos) finita e pré-definida. O termo `sequencial` significa que todos os elementos serão alocados contiguamente na memória, ou seja, o endereço de cada elemento é exatamente após o endereço do elemento anterior. Dependendo da linguagem, o endereço alocado é para uma referência ao elemento.

Esse tipo de lista pode ser implementado exatamente como um `vetor` (ou `array`). Entretanto, é preciso lembrar que seu uso se dará pelo menos com as operações comuns em listas. Exemplos mais detalhados: `classes Aluno`, `VetorAluno`, e `TesteVetorAluno` no pacote `aula08.codigos.exemplos.vetor`.

Normalmente a estrutura é criada com uma referência para o primeiro elemento da lista, e a partir da lista é possível acessar os demais elementos com o valor dos índices.

Em `Java` temos o `ArrayList`. A documentação para sua implementação de acordo com o `Java 17` pode ser encontrada aqui. Geralmente iniciamos um `ArrayList` assim:

```
ArrayList<Aluno> alunos = new ArrayList<Aluno>();
```

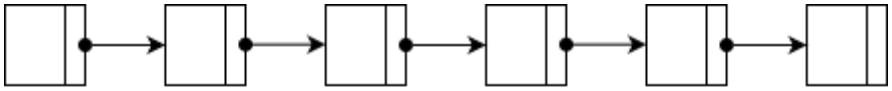
A partir disso estamos utilizando um de seus `construtores`, o qual irá criar um `array` de capacidade **inicial** de dez elementos. A `JVM` vai alocar dez espaços de memória contíguos onde em cada espaço haverá uma referência para algum objeto. À medida em que vamos adicionando novos elementos, o tamanho do `array` é aumentado automaticamente. Esse aumento acontece como já foi dito anteriormente, ou seja, um novo `array` com maior capacidade é criado, e os elementos existentes são copiados para o novo `array`. A diminuição não acontece automaticamente, porém existe o método `timeToSize()` que para realizar essa diminuição.

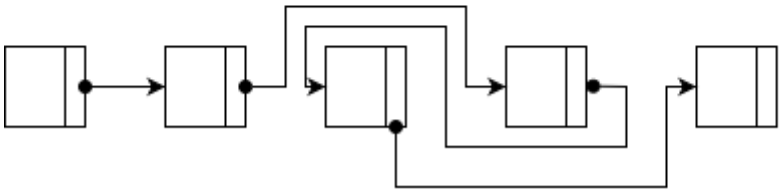
Caso queiramos criar um `array` com um tamanho mínimo especificado, podemos utilizar outro `construtor`:

```
ArrayList<Aluno> alunos = new ArrayList<Aluno>(100); // indicando que o array deve
```

## 3.2 Lista estática encadeada

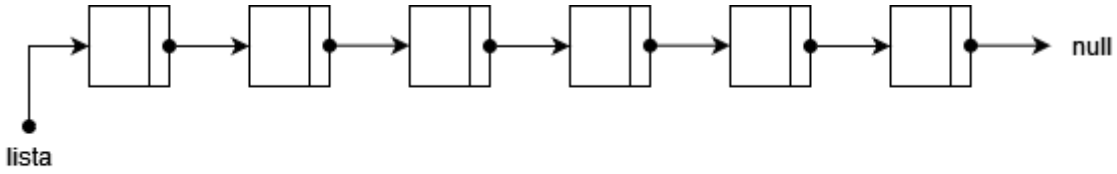
Uma `lista estática encadeada` é bastante parecida com uma `estática sequencial`. A diferença principal é que os elementos da lista (ou a referência aos elementos) não estarão alocados de forma contígua na memória. Isto faz com que cada elemento (ou referência) da lista precise ter uma referência para o elemento seguinte.

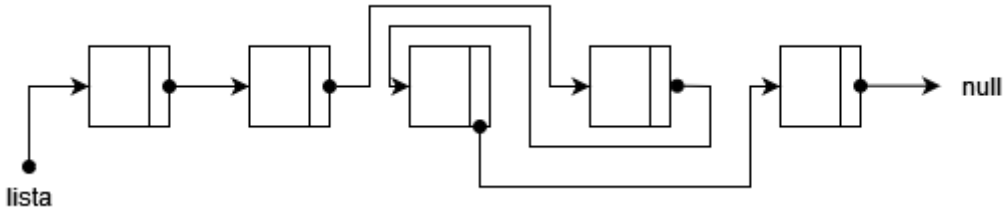




## 3.3 Lista dinâmica encadeada simples

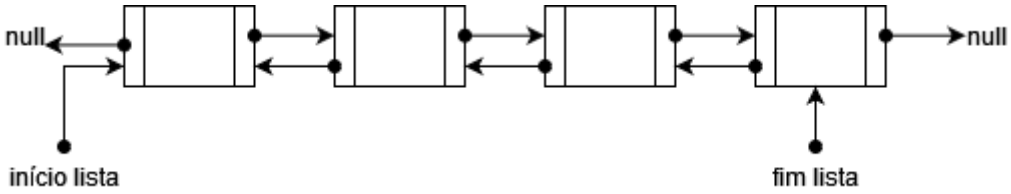
Em uma `lista dinâmica encadeada simples` a quantidade de elementos é indeterminada e, sempre que houver a necessidade uma nova alocação na memória ocorre. Esse novo elemento alocado é então acrescentado à lista. O `encademanto simples` significa que cada elemento terá uma referência para o elemento seguinte, de forma que só é possível percorrer a lista do início para o fim.




## 3.4 Lista dinâmica duplamente encadeada

Uma `lista dinâmica duplamente encadeada` é praticamente o mesmo da dinâmica encadeada simples. A principal diferença é que cada elemento (ou referência) da lista possui duas outras referências, um para o elemento seguinte e o outro para o elemento anterior. A partir disso é possível percorrer a lista em qualquer sentido, ou seja, do início para o fim ou do fim para o início. Por causa disso, uma implementação costuma ter uma referência para o início e outra para o fim da lista.




Em `Java` temos o `LinkedList` , o qual implementa a lista duplamente encadeada. Na prática, seu uso é bastante similar ao do `ArrayList` , com as diferenças óbvias de que existem `métodos` para adição e remoção tanto no início quanto no fim, além dos custos computacionais de alocação dinâmica. Exemplo:

```
LinkedList alunos = new LinkedList();
```

A documentação para sua implementação de acordo com o `Java 17` pode ser encontrada aqui