1 Introdução

As características inerentes dos vetores podem torná-los inconvenientes para a solução de diversos problemas computacionais. Uma dessas cacterísticas envolve o seu tamanho. Uma vez alocado, um vetor nunca pode ter o seu tamanho alterado. Isso quer dizer que, utilizando vetores, não podemos escrever programas que gerenciam coleções cujo tamanho se adapta de acordo com a demanda. Neste material, iremos aprender a implementar um tipo de coleção cujo tamanho pode variar. Ela terá seu funcionamento baseado em vetores. Como veremos, aplicaremos uma espécie de "truque" para fazer o ajuste do tamanho sob demanda.

2 Desenvolvimento

2.1 (Ideia geral) A coleção que implementaremos é uma **lista**. Uma lista é uma coleção de valores ordenados que admite elementos duplicados. Os detalhes de sua implementação são independentes de sua definição. Há diferentes formas para se implementar uma lista. A implementação que segue baseia-se em vetores. Ela tem as seguintes características.

- **Um vetor** (por enquanto, de inteiros) para armazenar os itens de interesse.
- Uma variável que indica **quantos** elementos **existem** no vetor.
- Uma variável que indica **quantos** elementos **cabem** no vetor.
- Um método que adiciona um elemento recebido como parâmetro.
- Um método que remove um elemento recebido como parâmetro.

Utilizaremos as variáveis **quantidade** e **capacidade** para ajustar o tamanho de nossa lista conforme a demanda. Elas serão utilizadas da seguinte forma:

- Quando quantidade e capacidade são iguais, a coleção está "cheia". Neste caso, iremos alocar no novo vetor. Ele terá o dobro da capacidade atual. Copiaremos todos os elementos para ele e, por fim, ajustaremos a variável de referência para que o referencie.

- Quando quantidade <= capacidade / 4, o vetor está, no máximo, um quarto cheio. Iremos interpretar isso como desperdício de memória e aplicar a mesma técnica para reduzir seu tamanho pela metade.

2.2 (Classe e sua variáveis de instância) A estrutura de dados que definimos é, muitas vezes, chamada de **vetor dinâmico**. Esse será o nome de nossa classe. Veja a Listagem 2.2.1. Por padrão, ela terá capacidade igual a quatro.

Listagem 2.2.1

```
public class VetorDinamico {
   private int [] elementos;
   private int quantidade;
   private int capacidade;
   public VetorDinamico(){
      this.capacidade = 4;
      this.elementos = new int[this.capacidade];
   }
}
```

2.3 (Método para verificar se a coleção está cheia) Antes de adicionar um elemento, precisamos verificar se ele cabe. As variáveis quantidade e capacidade nos dão essa informação. Caso sejam iguais, a coleção está cheia. Vamos escrever um método que resolve somente esse problema. Veja a Listagem 2.3.1.

Listagem 2.3.1

```
public boolean estaCheio (){
      if (this.quantidade == this.capacidade)
         return true;
      else
         return false;
}
```

Nota: O método estaCheio pode ser reescrito utilizando-se muito menos código. Tente, por exemplo, remover a palavra **else**. Avalie o resultado e convença-se de que o funcionamento permanece o mesmo. A seguir, tente também fazer a sua implementação usando um operador ternário e, por fim, tente fazer uma implementação que sequer usa uma estrutura de seleção!

2.4 (Método para adição (sem redimensionamento)) O método que adiciona um elemento ao vetor somente pode fazê-lo depois de verificar que há espaço para ele. Essa verificação será feita, é claro, usando o método estaCheio. Veja a Listagem 2.4.1. Repare como utilizamos a variável **quantidade** para **resolver dois problemas**: além de indicar quantos elementos há na coleção, ela também representa a posição em que o próximo elemento deve ser colocado no vetor.

Listagem 2.4.1

```
public void adicionar (int elemento){
      if (!estaCheio()){
         this.elementos[this.quantidade] = elemento;
         this.quantidade++;
      }
 }
```

2.5 (Método para aumentar a capacidade) O que desejamos fazer quando o método adicionar for chamado e detectar que a coleção está cheia? No momento, apenas ignoramos o novo elemento. Ele não é adicionado. O que desejamos é alocar mais espaço para que ele também caiba. Como sabemos, contudo, vetores não podem ter o seu tamanho alterado. Neste momento, iremos:

- alocar um novo vetor com o dobro da capacidade do atual
- copiar todos os elementos para o novo vetor
- adicionar o novo elemento ao novo vetor
- ajustar a variável de referência **elementos** para que ela referencie o novo vetor.

O que acontecerá com o vetor antigo? Como não teremos nenhuma variável de referência fazendo referência a ele, ele será um objeto na heap que, por conta dessa característica, será coletado em algum momento pelo **Garbage Collector**.

O método que implementa os passos descritos aparece na Listagem 2.5.1. Ele será encapsulado para que as demais classes não possam alterar o estado dos objetos do tipo VetorDinamico explicitamente.

Listagem 2.5.1

```
private void aumentarCapacidade (){
      int [] aux = new int[this.capacidade * 2];
      for (int i = 0; i < this.quantidade; i++){
         aux[i] = this.elementos[i];
      }
      this.elementos = aux;
      this.capacidade *= 2;
}
```

Nota: Estude o método arraycopy da classe System e tente refazer a implementação do método aumentarCapacidade utilizando ele.

2.6 (Método para adição (com redimensionamento)) A seguir, podemos refazer a implementação do método de adição. Quando o vetor estiver cheio, ele o redimensionará e, as seguir, fará a adição. Veja a Listagem 2.6.1.

Listagem 2.6.1

```
public void adicionar (int elemento){
      if (estaCheio()){
         this.aumentarCapacidade ();
      }
      this.elementos[this.quantidade] = elemento;
         this.quantidade++;
}
```

2.7 (Teste) Vamos escrever uma classe para testar o que temos até então. Ela possui um método main em que alocamos um vetor dinâmico e adicionamos elementos aleatórios a ele em um loop infinito. A cada adição, exibimos o vetor com seu conteúdo. Note, contudo, que o código cliente não tem meios de acessar as variáveis de instância do vetor, nem mesmo com métodos getters. Por essa razão, iremos escrever um método na classe VetorDinamico que tem como finalidade exibir os valores de suas variáveis de instância. Veja as listagens 2.7.1 e 2.7.2.

Listagem 2.7.1

```
//na classe VetorDinamico
   public void exibir (){
      System.out.printf ("Qtde: %d, Cap: %d\n", this.quantidade, this.capacidade);
      for (int i = 0; i < this.quantidade; i++){
         System.out.print (this.elementos[i] + " ");
      }
      System.out.println();
}
```

Listagem 2.7.2

```
import java.util.Random;
public class TesteVetorDinamico {

   public static void main(String[] args) throws InterruptedException{
      VetorDinamico v = new VetorDinamico();
      Random gerador = new Random ();
      while (true){
         int elemento = 1 + gerador.nextInt(6);
         v.adicionar(elemento);
         v.exibir();
         Thread.sleep(5000);
         System.out.println("==================================");
      }
   }
}
```

2.8 (Método para verificar se a coleção está vazia) Conforme adicionamos elementos à coleção, fazemos com que ela "cresça". Contudo, pode ser de interesse remover elementos dela. Antes de remover precisamos verificar se, dê fato, há pelo menos um elemento a ser removido. Assim evitamos exceções do tipo **ArrayIndexOutOfBoundsException**. O método da Listagem 2.8.1 responde se a coleção está vazia. Note que a variável quantidade resolve bem esse problema.

Listagem 2.8.1

```
public boolean estaVazio (){
      return this.quantidade == 0;
}
```

2.9 (Método para remoção (sem redimensionamento)) O método da Listagem 2.9.1 remove um elemento da coleção. Por padrão, ele remove o último elemento inserido.

Listagem 2.9.1

```
public void remover (){
      if (!estaVazio())
         this.quantidade--;

}
```

Nota: A remoção consiste em apenas decrementar a variável quantidade. Isso ocorre pois quando uma nova adição acontecer, ela sobrescreverá o elemento que se encontrava na posição quantidade, antes do decremento.

2.10 (Método para reduzir a capacidade) Uma vez feita uma remoção, precisamos verificar se o desperdício ultrapassou o limite. Se tiver ocorrido, iremos reduzir a capacidade do vetor. O método da Listagem 2.10.1 resolve esse problema.

Listagem 2.10.1

```
private void reduzirCapacidade (){
       int [] aux = new int[this.capacidade / 2];
      for (int i = 0; i < this.quantidade; i++){
         aux[i] = this.elementos[i];
      }
      this.elementos = aux;
      this.capacidade /= 2;
}
```

2.11 (Método para remoção (com redimensionamento)) Uma vez que uma remoção tenha sido feita, verificaremos se o desperdício está “grande”, ou seja, se ele ultrapassa o limite que especificamos. Se for o caso, iremos reduzir a sua capacidade pela metade. Como a Listagem 2.11.1 mostra, cabe ao método remover fazê-lo.

Listagem 2.11.1

```
public void remover() {
      if (!estaVazio()) {
         this.quantidade--;
         if (this.capacidade > 4 && this.quantidade <= this.capacidade / 4) {
            this.reduzirCapacidade();
         }
      }
}
```

2.12 (Testando inserção e remoção)) O método da Listagem 2.12.1 gera um valor aleatório para decidir se desejamos inserir ou remover elementos e, a seguir, realiza a operação.

Listagem 2.12.1

```
import java.util.Random;
public class TesteVetorDinamico {

   public static void main(String[] args) throws InterruptedException{
      VetorDinamico v = new VetorDinamico();
      Random gerador = new Random ();
      while (true){
         int op = gerador.nextInt(2);
         switch (op){
            case 0:
               int elemento = 1 + gerador.nextInt(6);
               v.adicionar(elemento);
               break;
            case 1:
               v.remover();
               break;
         }

         v.exibir();
         Thread.sleep(5000);
         System.out.println("==================================");
      }
   }
}
```

2.13 (Um único método para redimensionamento) Repare na semelhança entre os método que lidam com o redimensionamento do vetor. A menos da capacidade desejada, eles são idênticos. Sendo assim, podemos escrever um único método que realiza as duas operações. Para que ele opere adequadamente, sua lista de parâmetros especifica um valor real que será multiplicado pela capacidade atual. Se a operação desejada for redução de capacidade, o valor será 0.5. Caso contrário, será 2. Veja a Listagem 2.13.1.

Listagem 2.13.1

```
private void redimensionar(double valor) {
      int[] aux = new int[(int)(this.capacidade * valor)];
      for (int i = 0; i < this.quantidade; i++) {
         aux[i] = this.elementos[i];
      }
      this.elementos = aux;
      this.capacidade = (int)(this.capacidade * valor);
   }
```

- Cabe aos métodos de adição e remoção utilzá-lo apropriadamente. Veja a Listagem 2.13.2.

Listagem 2.13.2

```
 public void remover() {
      if (!estaVazio()) {
         this.quantidade--;
         if (this.capacidade > 4 && this.quantidade <= this.capacidade / 4) {
           this.redimensionar(0.5);
         }
      }
}

public void adicionar(int elemento) {
      if (estaCheio()) {
        this.redimensionar(2);
      }
      this.elementos[this.quantidade] = elemento;
      this.quantidade++;
   }
```

- Execute o teste novamente para certificar-se que os métodos continuam funcionando. A seguir, os métodos aumentarCapacidade e reduzirCapacidade podem ser removidos.

2.14 (Vetores para tipos de dados quaisquer (Generics)) A coleção que implementamos até então apresenta características bastante interessantes. Contudo, ela possui uma grave limitação. Somente é capaz de lidar com números inteiros. E se quisermos armazenar valores reais? Alunos? Músicas?? Para cada novo tipo teríamos de escrever uma nova classe idêntica, apenas substituindo o tipo que ela manipula. Uma solução muito mais conveniente envolve o uso de um recurso conhecido como **Generics**. Com ele, podemos deixar aberto um ou mais tipos de dados que a classe irá manipular. Neste caso, ao invés de o tipo de dados manipulado pelo vetor ser fixo como int, ele será "genérico". Caberá ao código cliente especificar qual tipo de dados deseja manipular no momento em que obtiver uma instância da classe. Veja a definição da nova classe na Listagem 2.14.1. Implementamos somente alguns métodos com o intuito de ilustrar o recurso.

Listagem 2.14.1

```
public class TesteVetorDinamicoGenerico {
   public static void main(String[] args) {
      VetorDinamicoGenerico <Double> v1 = new VetorDinamicoGenerico <>();
      VetorDinamicoGenerico <String> v2 = new VetorDinamicoGenerico <>();
      VetorDinamicoGenerico <Musica> v3 = new VetorDinamicoGenerico <>();
      v1.adicionar(2.1);
      v2.adicionar("Oi");
      v3.adicionar(new Musica ("Fade to Black"));
      v1.exibir();
      v2.exibir();
      v3.exibir();
   }
}

class Musica {
   private String titulo;

   public Musica (String titulo){
      this.titulo = titulo;
   }
}
```

2.15 (ArrayList) A coleção que acabamos de implementar é clássica. As linguagens de programação – incluindo a linguagem Java – possuem implementações parecidas e muito mais sofisticadas. O que fizemos até agora foi um exercício didático, para entendermos o funcionamento dessa estrutura tão importante. A partir de agora, passaremos a utilizar uma classe da biblioteca da linguagem Java que faz essencialmente exatamente o que a classe VetorDinamicoGenerico faz. Para ilustrar o seu funcionamento, vamos implementar um pequeno gerenciador de músicas. Nele, o usuário pode armazenar as suas músicas, atribuir uma nota a elas e exibir a lista de músicas ordenadas pela nota. A Listagem 2.15.1 mostra a implementação da classe Música.

Listagem 2.15.1

```
class Musica {
   private String titulo;
   private int avaliacao;

   public Musica (String titulo){
      this.titulo = titulo;
   }

   public void setAvaliacao(int avaliacao) {
      this.avaliacao = avaliacao;
   }
   public String getTitulo() {
      return titulo;
   }
}
```

- Na Listagem 2.15.2, instanciamos uma lista de músicas (usando a classe ArrayList) e montamos um menu de opções, que iremos implementar a seguir.

Listagem 2.15.2

```
public class ListaDeMusicas {
   public static void main(String[] args) {
      List <Musica> musicas = new ArrayList <> ();
      int opcao;
      do{
         opcao = Integer.parseInt (JOptionPane.showInputDialog("0-Sair\n1-Inserir
música\n2-Avaliar música\n1-Ver lista de músicas ordenada\n"));
         switch (opcao){
            case 1:
               break;
            case 2:
               break;
            case 3:
               break;
            case 0:
               JOptionPane.showMessageDialog(null, "Até mais");
               break;
            default:
               JOptionPane.showMessageDialog(null, "Opcao invalida");
         }

      }while (opcao != 0);
   }

}
```

- A seguir, implementamos as operações de adição e avaliação. Veja a Listagem 2.15.3.

Listagem 2.15.3

```
public class ListaDeMusicas {

   public static void main(String[] args) {
      List <Musica> musicas = new ArrayList <> ();
      int opcao;
      do{
         opcao = Integer.parseInt (JOptionPane.showInputDialog("0-Sair\n1-Inserir
música\n2-Avaliar música\n1-Ver lista de músicas ordenada\n"));
         switch (opcao){
            case 1:
               String musicaInserir = JOptionPane.showInputDialog("Qual o nome da
música?");
               musicas.add(new Musica(musicaInserir));
               JOptionPane.showMessageDialog(null, "Música armazenada com
sucesso!");
               break;
            case 2:
               String musicaAvaliar = JOptionPane.showInputDialog("Qual música
deseja avaliar?");
               int nota = Integer.parseInt (JOptionPane.showInputDialog("Qual a
nota?"));

               for (int i = 0; i < musicas.size(); i++){
                  if (musicas.get(i).getTitulo().equals(musicaAvaliar)){
                     musicas.get(i).setAvaliacao(nota);
                     break;
                  }
               }
               break;
            case 3:
               break;
            case 0:
               JOptionPane.showMessageDialog(null, "Até mais");
               break;
            default:
               JOptionPane.showMessageDialog(null, "Opcao invalida");
         }

      }while (opcao != 0);
   }

}
```

- O método **sort** da classe Collections é capaz de ordenar uma coleção de itens como um ArrayList. No entanto, seu funcionamento se baseia em comparações, o que quer dizer que os itens existentes na lista precisam ser **comparáveis**. Precisamos, de alguma forma, especificar um critério de comparação entre eles. Isso pode ser feito por meio do uso da interface **Comparable**. Quando uma classe a implementa, ela assume um **contrato**. Ela está prometendo implementar o método definido pela interface. Ele se chama **compareTo**. Sua lista de parâmetros especifica um único objeto. Ele será comparado com o objeto this. Seu tipo de retorno é int e seu funcionamento é o seguinte:

- caso o objeto referenciado por **this** seja considerado menor do que o objeto recebido como parâmetro, ele devolve um valor negativo.

- caso o objeto referenciado por **this** seja considerado maior do que o objeto recebido como parâmetreo, ele devolve um valor positivo

- caso ambos os objetos sejam iguais, ele devolve o valor 0.

Cabe a nós, desenvolvedores, especificar o comportamento que desejamos para o método compareTo. Neste caso, desejamos realizar a ordenação com base na avaliação. Músicas com valor de avaliação maior devem vir primeiro na lista. Veja como implementar a interface Comparable na Listagem 2.15.4. A Listagem 2.15.5 mostra como realizar a ordenação com Collections.sort.

Listagem 2.15.4

```
class Musica implements Comparable <Musica> {
    private String titulo;
    private int avaliacao;

    public Musica (String titulo){
        this.titulo = titulo;
    }

    public void setAvaliacao(int avaliacao) {
        this.avaliacao = avaliacao;
    }

    public String getTitulo() {
        return titulo;
    }

    @Override
    public int compareTo(Musica m) {
        if (this.avaliacao < m.avaliacao)
            return -1;
        if (this.avaliacao > m.avaliacao)
            return 1;
        return 0;
```

```
    }
}
```

Listagem 2.15.5

```
case 3:
                Collections.sort(musicas, Collections.reverseOrder());
                JOptionPane.showMessageDialog(null, musicas);
                break;
```

- Note que o que aparece na tela é um tanto curioso. Claramente, uma lista de músicas é exibida. Porém, a representação textual de cada música parece confusa. De onde vem esse valor e o que ele quer dizer? Ocorre que toda classe herda da classe **Object** e, portanto, ganha de presente tudo aquilo (ou quase tudo) que ela define. Um dos métodos que ela define se chama **toString**. Ele é responsável por dizer como se dá a representação textual de um objeto. O valor que vemos até o momento é devolvido por sua implementação padrão, existente na classe Object. Caso desejemos personalizar o seu funcionamento, podemos redefini-lo na classe Musica, como mostra a Listagem 2.15.6.

Listagem 2.15.6

```
@Override
   public String toString() {
      return String.format("Título: %s, Nota: %d\n", this.titulo, this.avaliacao);
}
```

Referências

DEITEL, P. e DEITEL, H. **Java Como Programar**. 8ª Edição. São Paulo, SP: Pearson, 2010.

LOPES, A. e GARCIA, G. **Introdução à Programação – 500 Algoritmos Resolvidos**. 1ª Edição. São Paulo, SP: Elsevier, 2002.