Capítulo 9

Tabelas de Espalhamento

"O primeiro passo para conseguirmos o que queremos na vida é decidirmos o que queremos"
— Ben Stein

Neste capítulo, implementaremos a estrutura de dados **Conjunto**.

9.1 - Introdução

Poderíamos simplesmente criar uma implementação desta estrutura que internamente encapsule uma `LinkedList` ou `ArrayList` como já fizemos anteriormente no caso da Fila e da Pilha.

Essa implementação é muito ineficiente computacionalmente: a cada inserção de palavra teremos que buscar dentro da Lista pela que está sendo adicionada, para evitar vocábulos repetidos. Também teremos de buscar dentro da Lista quando formos verificar se determinada palavra se encontra naquele vocabulário.

O problema de buscar em uma Lista é que fatalmente temos que percorrer ela por inteira e isso pode demorar muito: o consumo de tempo é linear em relação ao número de elementos que ela possui. O espaço de busca pode ser muito grande, a Lista pode ter milhões de elementos. Imagine uma Lista com as digitais de todos os brasileiros: são muitas digitais!

Em vez de buscar em todos os elementos do grupo, seria mais interessante conseguir restringir o espaço de busca, eliminando o maior número possível de elementos sem precisar "olhar" para eles.

Diferentemente das Listas, Pilhas e Filas, os Conjuntos não precisam manter os elementos em seqüência. Então tentaremos uma abordagem bem diferente aqui. Vamos armazenar os elementos do Conjunto espalhados! Isso mesmo,

espalhados, mas não aleatoriamente e sim com uma certa lógica que facilitará buscá-los.

A idéia é separar os elementos em categorias de acordo com certas características chaves do próprio elemento. Desta forma, para buscar um elemento, basta verificar qual é a categoria dele e assim eliminar todos os outros que estão nas outras categorias.

Essa idéia é fantástica e todo mundo já utilizou algo parecido no dia a dia. Um exemplo clássico em que essa técnica aparece é a agenda de telefones.

Nas agendas de telefones, os contatos são separados por categorias, normalmente, o que define a categoria do contato é a primeira letra do nome dele. Há uma página para cada letra do alfabeto. Desta forma, quando é preciso buscar um contato, vamos direto para a página referente a primeira letra do nome dele e assim eliminamos todos os contatos das outras páginas.

A técnica que estamos falando aqui é a técnica de **Espalhamento**. Ela consiste basicamente de duas partes: a **Função de Espalhamento** ("Função de Hash") e a **Tabela de Espalhamento** ("Tabela de Hash").

A Função de Espalhamento deve descobrir qual é a categoria de um determinado elemento. Para isso, ela deve analisar as características chaves do próprio elemento. Por exemplo, na agenda de telefone, a Função deveria ver qual é a primeira letra do nome do contato e devolver o número da página referente àquela letra.

A Tabela de Espalhamento deve armazenar as categorias e cada categoria deve ter um índice. Este índice, gerado com a Função de Espalhamento, é usado para recuperar a categoria rapidamente.

A Função deve ser determinística, ou seja, toda vez que aplicada a um mesmo elemento ela deve gerar o mesmo índice, caso contrário procuraríamos um nome na página errada da agenda.

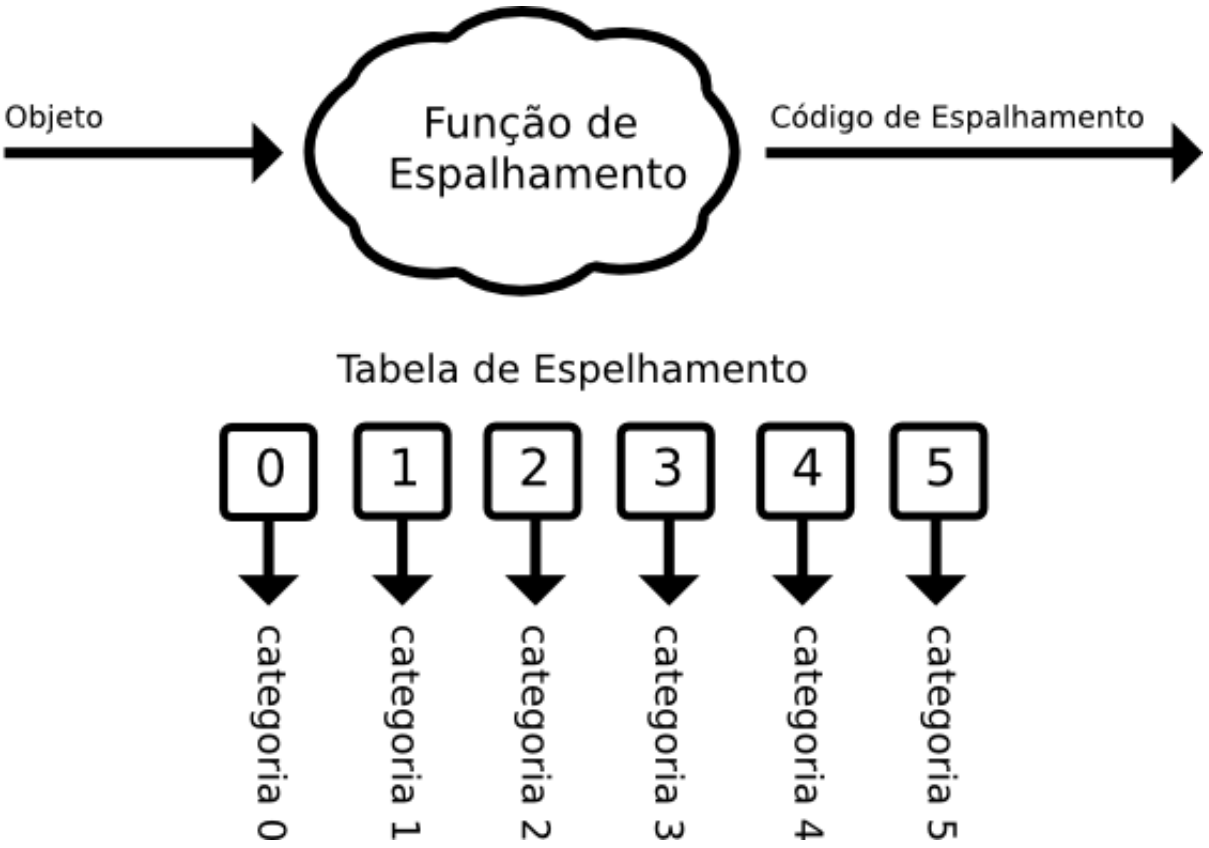


Figura 9.1: Técnica de Espalhamento

No mundo ideal, a Função deveria gerar índices diferentes para elementos diferentes. Mas, na prática, ela pode, eventualmente, gerar o mesmo índice para elementos distintos. Quando isto ocorre, dizemos que houve uma **colisão**. No exemplo da agenda de contatos, uma colisão acontece quando o nome de dois contatos distintos iniciam com mesma letra.

Para resolver o problema da colisão, muitas vezes, a Tabela é combinada com outra estrutura de dados. Isso ocorre da seguinte forma: cada posição da Tabela pode armazenar uma Lista, uma Árvore ou outra estrutura de dados em vez de apenas um elemento. Desta forma conseguimos armazenar mais de um elemento na mesma posição da Tabela.

Cuidados devem ser tomados para não permitir a repetição de elementos nas estruturas que forem combinadas com a Tabela.

Vamos criar uma classe para implementar um Conjunto de palavras para uma língua utilizando a técnica de Espalhamento.

```
package br.com.caelum.ed.conjuntos;

public class ConjuntoEspalhamento {
```

```
  public void adiciona(String palavra) {
    // implementação
  }

  public void remove(String palavra) {
    // implementação
  }

  public boolean contem(String palavra) {
    // implementação
  }

  public List<String> pegaTodas(){
    // implementação
  }

  public int tamanho() {
    // implementação
  }
}
```

Podemos também definir alguns testes:

```
package br.com.caelum.ed.conjuntos;

public class Teste {
  public static void main(String[] args) {
    ConjuntoEspalhamento conjunto = new ConjuntoEspalhamento();

    conjunto.adiciona("palavra");
    conjunto.adiciona("computador");
    conjunto.adiciona("apostila");
    conjunto.adiciona("instrutor");
    conjunto.adiciona("mesa");
    conjunto.adiciona("telefone");

    if (!conjunto.contem("apostila")) {
      System.out.println("Erro: não tem a palavra: apostila");
    }

    conjunto.remove("apostila");

    if (conjunto.contem("apostila")) {
      System.out.println("Erro: tem a palavra: apostila");
    }

    if (conjunto.tamanho() != 5) {
      System.out.println("Erro: o tamanho do conjunto deveria ser 5");
    }
  }
}
```

9.2 – Tabela de Espalhamento

A Tabela de Espalhamento pode ser implementada através de uma Lista de Strings. Neste caso, cada posição da Tabela poderá guardar somente uma palavra da língua. No mundo ideal, isto seria ótimo.

Mas, na prática, haverá colisões causadas pela Função de Espalhamento. Devemos combinar a Tabela com outra estrutura de dados para contornar as colisões.

Para combinar a Tabela com Listas, poderíamos definir um atributo do tipo Lista de Listas de Strings na classe `ConjuntoEspalhamento`.

Seria interessante que a Lista principal, aquela que armazena Listas, fosse um Vetor pois a princípio ela não sofrerá adição nem remoção de elementos.

As Listas secundárias, aquelas armazenadas na Lista principal, guardarão as palavras e elas sim sofrerão adição e remoção de elementos então seria interessante que essas fossem Listas Ligadas.

Lembrando que as classes do Java que implementam Vetor e Lista Ligada são `ArrayList` e `LinkedList` respectivamente.

```
package br.com.caelum.ed.conjuntos;

import java.util.ArrayList;
import java.util.LinkedList;

public class ConjuntoEspalhamento {

  private ArrayList<LinkedList<String>> tabela =
      new ArrayList<LinkedList<String>>();

  ...
}
```

Essa declaração java pode parecer complicada, mas você pode imaginar que aí criamos algo similar a uma matriz bidimensional, só que como são Collections do java, temos grandes vantagens e facilidades através de seus métodos.

Precisamos definir quantas posições a Tabela terá, ou seja, quantas Listas secundárias devemos ter, e inicializar cada uma delas. Isso pode ser feito no construtor da nossa classe. Inicialmente, teremos 26 posições na Tabela, uma para cada letra do alfabeto. Aqui vamos separar as palavras de acordo com a primeira letra delas: essa será nossa Função de Espalhamento. Assim como nas agendas de contatos de antigamente.

```
public class ConjuntoEspalhamento {
```

```
  private ArrayList<LinkedList<String>> tabela =
      new ArrayList<LinkedList<String>>();

  public ConjuntoEspalhamento() {
    for (int i = 0; i < 26; i++) {
      LinkedList<String> lista = new LinkedList<String>();
      tabela.add(lista);
    }
  }

  ...
}
```

Nós definimos o tipo do atributo `tabela` como `ArrayList` de `LinkedList`. Desta forma, estamos "amarrando" nossa implementação a tipos específicos de Listas. Se depois quisermos trocar os tipos das Listas teremos problemas. Podemos declarar e inicializar o atributo de forma mais genérica e elegante:

```
private List<List<String>> tabela = new ArrayList<List<String>>();
```


9.3 - Função de Espalhamento

A nossa Função deve observar a primeira letra das palavras e calcular o índice correto na Tabela. Por exemplo, palavras começadas com "a" devem ir para uma posição, palavras começadas com "b" devem ir para outra, e assim por diante até a letra "z".

Vamos fazer um método para implementar a Função de Espalhamento. Este método deve receber uma String (palavra) e devolver um int (índice associado a palavra).

```
private int calculaIndiceDaTabela(String palavra){
  return palavra.toLowerCase().charAt(0) % 26;
}
```

Observe que foram utilizados os métodos `toLowerCase()` e `chatAt(int)` da classe `String`. O primeiro devolve a palavra em letras minúsculas. O segundo devolve o caractere na posição passada como parâmetro. Além disso, foi feito uma operação de Resto da Divisão. Isso parece estranho mas faz todo o sentido visto que cada caractere tem um valor inteiro positivo.

Em vez de pegar o resto da divisão, poderíamos simplesmente subtrair o valor 65 desse caractere, que é o caractere `'A'`. Mas o que aconteceria se o primeiro caractere da String fosse um número ou um caractere especial?

Esta operação de resto ajusta o resultado da função de hash para não sair do intervalo fechado de `0` até `25`.

9.4 - Operações necessárias

Na seqüência, vamos implementar as operações que um Conjunto de palavras deve ter.

9.5 - Adicionar uma palavra

Para adicionar uma palavra no Conjunto, devemos aplicar a Função de Espalhamento para descobrir em qual posição da Tabela devemos adicionar. Depois, recuperamos a Lista que está nesta posição para guardar a palavra.

```
public void adiciona(String palavra) {
  int indice = this.calculaIndiceDaTabela(palavra);
  List<String> lista = this.tabela.get(indice);
  lista.add(palavra);
}
```

Este método ficou simples mas contém um erro. O requisito fundamental de um Conjunto é não ter elementos repetidos. Como as Listas permitem elementos repetidos se adicionarmos duas vezes a mesma palavra ela será inserida duas vezes.

Podemos evitar a repetição de palavras fazendo uma pequena verificação antes de adicionar uma palavra.

```
public void adiciona(String palavra) {
  if (!this.contem(palavra)) {
    int indice = this.calculaIndiceDaTabela(palavra);
    List<String> lista = this.tabela.get(indice);
    lista.add(palavra);
```

```
  }
}
```

Utilizamos o método `contem(String)` para saber se o Conjunto já contém a palavra. Se contém o `adiciona(String)` não faz nada.

9.6 – Remover uma palavra

Analogamente ao `adiciona(String)`, o método `remove(String)` deve achar a posição da Tabela onde está a Lista na qual a palavra a ser removida estaria. Depois, basta remover a palavra da Lista.

Antes de tentar remover poderíamos verificar se a palavra está no Conjunto. Nas Listas da API do Java, existe uma sobrecarga do método de `remover` que recebe o próprio elemento, além do qual recebe um índice. Isto auxilia no nosso método:

```
public void remove(String palavra) {
  if (this.contem(palavra)) {
    int indice = this.calculaIndiceDaTabela(palavra);
    List<String> lista = this.tabela.get(indice);
    lista.remove(palavra);
  }
}
```

9.7 – Verificar se uma palavra está ou não no Conjunto

Esta operação é simples, basta achar o índice da Tabela aplicando a Função de Espalhamento da palavra desejada e verificar se ela está na Lista correspondente.

```
public boolean contem(String palavra) {
```

```
  int indice = this.calculaIndiceDaTabela(palavra);
  List<String> lista = this.tabela.get(indice);

  return lista.contains(palavra);
}
```

Aqui está o grande truque para deixar nosso Conjunto mais rápido que uma simples lista: buscamos apenas nos elementos que se encontram naquela "página da agenda". Se o elemento não estiver lá, com certeza ele não se encontra em nenhuma outra página da agenda. O nosso espalhamento tem uma certa organização que facilita as buscas!

9.8 - Recuperar todas as palavras do Conjunto

As palavras estão armazenadas na Tabela. Então, para recuperar todas as palavras, precisamos percorrer todas as posições da Tabela. Em cada posição, há uma Lista, pegaremos todos os elementos de cada Lista e armazenaremos em uma única Lista e a devolveremos.

```
public List<String> pegaTodas() {
  List<String> palavras = new ArrayList<String>();

  for (int i = 0; i < this.tabela.size(); i++) {
    palavras.addAll(this.tabela.get(i));
  }

  return palavras;
}
```

Este método cria uma Lista do tipo `ArrayList`. Porém, observe que a referência é do tipo `List`. Isso é possível por causa do **Polimorfismo** (capacidade de referenciar de várias maneiras um mesmo objeto). Todo objeto do tipo `ArrayList` também é do tipo `List`.

Todas as Listas do Java disponibilizam uma operação que permite adicionar diversos elementos de uma vez só. Esta funcionalidade é implementada pelo método `addAll(Collection)`. Este método foi utilizado dentro do `for` para inserir na Lista de todas as palavras do Conjunto todas as palavras da Lista da posição `i` da Tabela.

9.9 - Informar o tamanho do Conjunto de palavras

Para esta operação, temos duas alternativas ou em cada vez que a operação for executada nós percorremos todas as Listas contanto os elementos ou fazemos de uma forma mais eficiente que é guardar em um atributo a quantidade de palavras presentes no Conjunto. Este atributo deve ser incrementado toda vez que uma palavra for adicionada e decrementado toda vez que uma palavra for removida.

```
public class ConjuntoEspalhamento {

  ...

  private int tamanho = 0;

  ...

  public int tamanho() {
    return this.tamanho;
  }
}
```

9.10 - Exercícios: Tabela de Espalhamento 1

1. Crie a classe `ConjuntoEspalhamento` para implementar Conjunto utilizando a técnica de Espalhamento. Faça esta classe no pacote **br.com.caelum.ed.conjuntos**. Coloque o atributo que implementa a tabela e o construtor que inicializa cada Lista da Tabela.

```
package br.com.caelum.ed.conjuntos;

import java.util.ArrayList;
import java.util.LinkedList;

public class ConjuntoEspalhamento {

  private List<List<String>> tabela = new ArrayList<List<String>>();

  public ConjuntoEspalhamento() {
    for (int i = 0; i < 26; i++) {
```

```
      LinkedList<String> lista = new LinkedList<String>();
      tabela.add(lista);
    }
  }
}
```

2. Faça o método que calcula qual é o índice da Tabela onde uma palavra deve ser armazenada. Este método deve classificar as palavras de acordo com a primeira letra.

```
private int calculaIndiceDaTabela(String palavra) {
  return palavra.toLowerCase().charAt(0) % 26;
}
```

3. Vamos implementar o método de adicionar palavras no Conjunto assim como foi discutido neste capítulo.

```
public void adiciona(String palavra) {
  if (!this.contem(palavra)) {
    int indice = this.calculaIndiceDaTabela(palavra);
    List<String> lista = this.tabela.get(indice);
    lista.add(palavra);
    this.tamanho++;
  }
}
```

Não esqueça de acrescentar o atributo `tamanho` para contabilizar o total de palavras.

```
private int tamanho = 0;
```

4. Para visualizar melhor o que está acontecendo como o Conjunto, vamos implementar agora o método que recupera todas as palavras da mesma forma que fizemos neste capítulo.

```
public List<String> pegaTodas() {
  List<String> palavras = new ArrayList<String>();

  for (int i = 0; i < this.tabela.size(); i++) {
    palavras.addAll(this.tabela.get(i));
  }

  return palavras;
}
```

Faça testes adicionando e imprimindo algumas palavras:

```
package br.com.caelum.ed.conjuntos;

import java.util.List;

public class TesteAdiciona {
  public static void main(String[] args) {
```

```
    ConjuntoEspalhamento conjunto = new ConjuntoEspalhamento();
    conjunto.adiciona("Rafael");
    conjunto.adiciona("Ana");
    conjunto.adiciona("Paulo");

    List<String> palavras = conjunto.pegaTodas();

    for (String palavra : palavras) {
      System.out.println(palavra);
    }
  }
}
```

Este teste deve imprimir o nome das três pessoas.

5. Implemente o `remove(String)`

```
public void remove(String palavra) {
  if (this.contem(palavra)) {
    int indice = this.calculaIndiceDaTabela(palavra);
    List<String> lista = this.tabela.get(indice);
    lista.remove(palavra);
    this.tamanho--;
  }
}
```

E teste:

```
public class TesteRemove {
  public static void main(String[] args) {
    ConjuntoEspalhamento conjunto = new ConjuntoEspalhamento();
    conjunto.adiciona("Rafael");
    conjunto.adiciona("Ana");
    conjunto.adiciona("Paulo");

    List<String> palavras = conjunto.pegaTodas();

    System.out.println("antes de remover");
    for (String palavra : palavras) {
      System.out.println(palavra);
    }

    conjunto.remove("Rafael");

    palavras = conjunto.pegaTodas();

    System.out.println("depois de remover");
    for (String palavra : palavras) {
      System.out.println(palavra);
    }
  }
}
```

6. Implemente o `contem(String)`

```
public boolean contem(String palavra) {
  int indice = this.calculaIndiceDaTabela(palavra);
  List<String> lista = this.tabela.get(indice);

  return lista.contains(palavra);
}
```

Faça um teste adicionando algumas palavras e testando se o método devolve `true` para elas. Teste também com palavras que não foram adicionadas. Neste caso, o método deve devolver `false`.

7. Implemente o método `tamanho()` e faça testes.

O Conjunto que implementamos não deve permitir repetição de palavras. Então, experimente adicionar palavras repetidas e depois verificar o tamanho do Conjunto.

9.11 – Diminuindo Colisões

Até o momento, não estamos tratando eficientemente as colisões. Devemos considerar dois problemas: o desbalanceamento da Tabela e a sobrecarga das Listas da Tabela.

O primeiro problema acontece quando a Função de Espalhamento não "espalha" direito os elementos. Por exemplo, na língua portuguesa há muito mais palavras começando com "a" do que com "z" então a nossa Função que considera apenas a primeira letra sobrecarrega muito mais a Lista da letra "a" do que a da letra "z" que fica praticamente vazia.

O segundo problema ocorre quando adicionamos muitas palavras. A Tabela tem tamanho fixo, ou seja, o número de Listas é constante. Ao adicionar `100000` palavras, calculando a melhor média possível, cada Lista interna da Tabela terá `100000/26 = 3846.2` elementos. Conforme as Listas vão ficando mais sobrecarregadas pior será o desempenho do sistema, chegando próximo do consumo de tempo linear.

Para resolver o primeiro problema, vamos fazer a Função de Espalhamento considerar melhor as características dos elementos do Conjunto. No nosso caso, a Função vai olhar para todas as letras da palavra em vez de olhar só para a primeira.

Para resolver o segundo, vamos fazer o tamanho da Tabela de Espalhamento ser dinâmico. Assim, se o Conjunto estiver com muitas palavras aumentamos este

tamanho caso contrário diminuímos.

9.12 - Espalhando Melhor

As tarefas de analisar as características das palavras e de gerar um índice válido para a Tabela de Espalhamento estão "amarradas" em um único lugar, no método `calculaIndiceDaTabela(String)`.

Desta forma, será difícil fazer alterações independentes em cada uma destas duas importantes tarefas. Então, vamos separá-las em dois métodos distintos. Estes métodos funcionarão da seguinte forma: o primeiro ficará encarregado de analisar uma dada palavra e gerar um código genérico que chamaremos de **Código de Espalhamento**, o segundo estará incumbido de a partir do Código de Espalhamento calcular um índice válido para a Tabela.

```
private int calculaCodigoDeEspalhamento(String palavra){
  int codigo = 1;
  for (int i = 0; i < palavra.length(); i++) {
    codigo = codigo + palavra.charAt(i);
  }
  return codigo;
}
```

Acima, o método que calcula o Código de Espalhamento faz um calculo sobre **todos** os caracteres que formam a String. Agora a palavra "caelum" não terá mais o mesmo resultado da função de hash que a palavra "campeão".

Apesar dessa significativa melhora, ainda existem outras colisões básicas que podem acontecer, como "caelum" com "caemul": uma simples troca de ordem das letras gera uma colisão. Não é escopo deste texto, mas aqui para melhorar ainda mais a função de hash, podemos dar um peso para cada posição da String, assim os anagramas não resultarão em colisão, como por exemplo:

```
private int calculaCodigoDeEspalhamento(String palavra){
  int codigo = 1;
  for (int i = 0; i < palavra.length(); i++) {
    codigo = 31 * codigo + palavra.charAt(i);
  }
  return codigo;
}
```

Agora, o método `calculaIndiceDaTabela(String)` simplesmente usa o método `calculaCodigoDeEspalhamento` para obter o Código de Espalhamento e a partir dele calcula um índice válido para a Tabela (isto é, dentro dos índices da tabela):

```
private int calculaIndiceDaTabela(String palavra) {
  int codigoDeEspalhamento = this.calculaCodigoDeEspalhamento(palavra);
  return codigoDeEspalhamento % this.tabela.size();
}
```

Observe como o Código de Espalhamento é ajustado para gerar um índice da Tabela. A operação Resto da Divisão devolve um número no intervalo fechado `[0, this.tabela.size() - 1]`.

Como foi dito, o Código de Espalhamento de um elemento é genérico e deve considerar apenas as características do próprio elemento. Eventualmente, este código poderia ser até um número negativo. Então, devemos tomar cuidado com este fato.

Do jeito que está, o método `calculaIndiceDaTabela(String)` poderia gerar um índice negativo para a tabela. Isso ocorreria se o Código de Espalhamento fosse negativo. Com uma pequena modificação evitamos este problema.

```
private int calculaIndiceDaTabela(String palavra) {
  int codigoDeEspalhamento = this.calculaCodigoDeEspalhamento(palavra);
  codigoDeEspalhamento = Math.abs(codigoDeEspalhamento);
  return codigoDeEspalhamento % tabela.size();
}
```

Através do método `Math.abs(int)` obtemos o valor absoluto do Código de Espalhamento.

9.13 – Exercícios: Tabela de Espalhamento 2

1. Vamos verificar se realmente a nossa Função de Espalhamento está melhor que a antiga. Para isso, vamos fazer um método na classe `ConjuntoEspalhamento` que imprime o estado da Tabela.

```
public void imprimeTabela(){
```

```
  for (List<String> lista : this.tabela) {
    System.out.print("[");
    for (int i = 0; i < lista.size(); i++) {
      System.out.print("*");
    }
    System.out.println("]");
  }
}
```

E o seguinte teste:

```
package br.com.caelum.ed.conjuntos;

public class TesteEspalhamento {

  public static void main(String[] args) {

    ConjuntoEspalhamento conjunto = new ConjuntoEspalhamento();

    for (int i = 0; i < 100; i++) {
      conjunto.adiciona("" + i);
    }

    conjunto.imprimeTabela();
  }
}
```

Observe que as palavras ficam concentradas em algumas Listas.

Agora, faça as modificações para melhorar a Função de Espalhamento.

```
private int calculaCodigoDeEspalhamento(String palavra) {
  int codigo = 1;
  for (int i = 0; i < palavra.length(); i++) {
    codigo = 31 * codigo + palavra.charAt(i);
  }
  return codigo;
}

private int calculaIndiceDaTabela(String palavra) {
  int codigoDeEspalhamento = this.calculaCodigoDeEspalhamento(palavra);
  codigoDeEspalhamento = Math.abs(codigoDeEspalhamento);
  return codigoDeEspalhamento % tabela.size();
}
```

Execute novamente o teste e veja a diferença.

9.14 – Tabela Dinâmica

O tamanho da Tabela será dinâmico, ou seja, quanto mais elementos no Conjunto maior a Tabela e quanto menos elementos menor a Tabela. Precisamos

de algum critério para decidir quando aumentar e quando diminuir o tamanho da Tabela.

A capacidade da Tabela é a capacidade das suas Listas. A Tabela está sobrecarregada quando o número de elementos é muito maior do que a capacidade. Então, vamos definir a carga da Tabela como sendo a razão entre o número de elementos e o número de Listas.

```
carga = tamanho / numero de Listas
```

Se esta carga ultrapassar o valor `0.75` significa que há uma alta chance de ocorrer colisões. Neste caso, aumentaremos o tamanho da Tabela.

Se a carga ficar menor do que o valor `0.25` significa que a Tabela tem muito espaço vazio. Neste caso, diminuiremos o tamanho dela.

Estes valores não foram chutados, eles são obtidos através de análise estatística avançada. Não faremos estas análises aqui pois foge do escopo deste texto.

Quando aumentarmos ou diminuirmos o tamanho da Tabela temos que redistribuir os elementos. Vamos criar um método que redimensiona a Tabela dado uma nova capacidade.

```
private void redimensionaTabela(int novaCapacidade){
  List<String> palavras = this.pegaTodas();
  this.tabela.clear();

  for (int i = 0; i < novaCapacidade; i++) {
    this.tabela.add(new LinkedList<String>());
  }

  for (String palavra : palavras) {
    this.adiciona(palavra);
  }
}
```

O que é feito neste método é o seguinte:

1. Guarda em uma Lista todas as palavras do Conjunto.

2. Limpa a tabela através do método `clear()` que toda Lista do Java tem.

3. Inicializa a Tabela com várias listas.

4. Guarda novamente as palavras no Conjunto.

Agora ao adicionar ou remover um elemento do Conjunto, verificamos se a

carga está acima ou abaixo do aceitável. Caso estiver, redimensionamos adequadamente.

```
private void verificaCarga() {
  int capacidade = this.tabela.size();
  double carga = (double) this.tamanho / capacidade;

  if (carga > 0.75) {
    this.redimensionaTabela(capacidade * 2);
  } else if (carga < 0.25) {
    this.redimensionaTabela(Math.max(capacidade / 2, 10));
  }
}
```

O método acima calcula a carga atual do conjunto. Se a carga for maior que `0.75` a novaCapacidade será o dobro da antiga. Se a carga for menor que `0.25` a novaCapacidade será a metade da antiga mas para evitar ficar com uma capacidade muito pequena definimos que o mínimo é 10.

O `verficaCarga()` deve ser utilizado antes de adicionar e após remover uma palavra.

9.15 - Exercícios: Tabela de Espalhamento 3

1. Neste ponto, nossa implementação deve estar sobrecarregando as Listas da Tabela deixando o desempenho ruim. Para verificar este fato faça o seguinte teste:

```
package br.com.caelum.ed.conjuntos;

public class TesteDesempenho {

  public static void main(String[] args) {
```

```
    long inicio = System.currentTimeMillis();

    ConjuntoEspalhamento conjunto = new ConjuntoEspalhamento();

    for (int i = 0; i < 50000; i++) {
      conjunto.adiciona("palavra" + i);
    }

    for (int i = 0; i < 50000; i++) {
      conjunto.contem("palavra" + i);
    }

    long fim = System.currentTimeMillis();
    System.out.println("Tempo: " + (fim - inicio) / 1000.0);
  }
}
```

Marque o tempo gasto!

Faça as melhorias adicionado o recurso de Tabela de Espalhamento dinâmica.

```
private void redimensionaTabela(int novaCapacidade) {
  List<String> palavras = this.pegaTodas();
  this.tabela.clear();

  for (int i = 0; i < novaCapacidade; i++) {
    this.tabela.add(new LinkedList<String>());
  }

  for (String palavra : palavras) {
    this.adiciona(palavra);
  }
}

private void verificaCarga() {
  int capacidade = this.tabela.size();
  double carga = (double) this.tamanho / capacidade;

  if (carga > 0.75) {
    this.redimensionaTabela(capacidade * 2);
  } else if (carga < 0.25) {
    this.redimensionaTabela(Math.max(capacidade / 2, 10));
  }
}
```

Não esqueça de invocar o `verificaCarga()` antes de adicionar uma palavra e depois de remover.

```
public void adiciona(String palavra) {
  if (!this.contem(palavra)) {
    this.verificaCarga();
    // RESTO DO CÓDIGO
  }
}
```

```
public void remove(String palavra) {
  if (this.contem(palavra)) {
    // RESTO DO CÓDIGO
    this.verificaCarga();
  }
}
```

Execute novamente o teste e veja a diferença.

9.16 – Generalização

A nossa implementação de Conjunto usando a técnica de Espalhamento funciona apenas para armazenar palavras (Strings). Estamos amarrando a estrutura de dados ao tipo de dado que ela vai guardar. Seria melhor generalizar o nosso Conjunto.

Assim como fizemos em capítulos anteriores, vamos trocar os tipos específicos por referências da classe `Object`. Diferentemente das outras estruturas que implementamos até o momento, a generalização do Conjunto será um pouco mais delicada.

O problema é o calculo do Código de Espalhamento. Este calculo depende do tipo do objeto. Para gerar um bom Código de Espalhamento o Conjunto deveria conhecer as características de todos os tipo de objetos. Isso seria inviável.

A solução é deixar alguém que conhece muito bem as características do objeto a ser armazenado no Conjunto gerar o Código de Espalhamento. Quem melhor conhece um objeto é a classe dele. Então, as classes dos objetos que precisam ser armazenados devem definir como gerar o Código de Espalhamento.

No Java, a classe `Object` define um método para gerar o Código de Espalhamento. Este método é o `hashCode()`. Eventualmente, você pode reescrever este método para implementar o seu Código de Espalhamento ou pode utilizar o que está feito em `Object`.

```
private int calculaIndiceDaTabela(Object objeto) {
  int codigoDeEspalhamento = objeto.hashCode();
  codigoDeEspalhamento = Math.abs(codigoDeEspalhamento);
  return codigoDeEspalhamento % this.tabela.size();
}
```

O método `calculaCodigoDeEspalhamento(Object)` não é mais necessário na classe que implementa o Conjunto.

```java
package br.com.caelum.ed.conjuntos;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class ConjuntoEspalhamentoGenerico {

  private List<List<Object>> tabela = new ArrayList<List<Object>>();

  private int tamanho = 0;

  public ConjuntoEspalhamentoGenerico() {
    for (int i = 0; i < 26; i++) {
      LinkedList<Object> lista = new LinkedList<Object>();
      tabela.add(lista);
    }
  }

  public void adiciona(Object objeto) {
    if (!this.contem(objeto)) {
      this.verificaCarga();
      int indice = this.calculaIndiceDaTabela(objeto);
      List<Object> lista = this.tabela.get(indice);
      lista.add(objeto);
      this.tamanho++;
    }
  }

  public void remove(Object objeto) {
    if (this.contem(objeto)) {
      int indice = this.calculaIndiceDaTabela(objeto);
      List<Object> lista = this.tabela.get(indice);
      lista.remove(objeto);
      this.tamanho--;
      this.verificaCarga();
    }
  }

  public List<Object> pegaTodos() {
    List<Object> objetos = new ArrayList<Object>();
    for (int i = 0; i < this.tabela.size(); i++) {
      objetos.addAll(this.tabela.get(i));
    }
    return objetos;
  }

  public boolean contem(Object objeto) {
    int indice = this.calculaIndiceDaTabela(objeto);
    List<Object> lista = this.tabela.get(indice);

    return lista.contains(objeto);
  }

  public int tamanho() {
    return this.tamanho;
  }
```

```
  private int calculaIndiceDaTabela(Object objeto) {
    int codigoDeEspalhamento = objeto.hashCode();
    codigoDeEspalhamento = Math.abs(codigoDeEspalhamento);
    return codigoDeEspalhamento % tabela.size();
  }

  private void redimensionaTabela(int novaCapacidade) {
    List<Object> objetos = this.pegaTodos();
    this.tabela.clear();

    for (int i = 0; i < novaCapacidade; i++) {
      this.tabela.add(new LinkedList<Object>());
    }

    for (Object objeto : objetos) {
      this.adiciona(objeto);
    }
  }

  private void verificaCarga() {
    int capacidade = this.tabela.size();
    double carga = (double) this.tamanho / capacidade;

    if (carga > 0.75) {
      this.redimensionaTabela(capacidade * 2);
    } else if (carga < 0.25) {
      this.redimensionaTabela(Math.max(capacidade / 2, 10));
    }
  }

  public void imprimeTabela() {
    for (List<Object> lista : this.tabela) {
      System.out.print("[");
      for (int i = 0; i < lista.size(); i++) {
        System.out.print("*");
      }
      System.out.println("]");
    }
  }
}
```

9.17 – EQUALS E HASHCODE

Repare que a nossa busca pelo elemento usa o método `contains` da `List`. A implementação de lista que estamos usando (assim como a maioria absoluta das coleções da API do Java) utiliza do método `equals` para saber se determinado elemento está contido naquela lista ou não.

Um problema aparece quando rescrevemos o `equals` e não rescrevemos o `hashCode`: pode ser que existam dois objetos, digamos `a` e `b`, que são considerados iguais pelo nosso método, mas possuem códigos de espalhamento diferentes. Isto

vai fazer com que a pesquisa por `a`, no caso de só existir `b` dentro do conjunto retorne `false`, sendo que na verdade deveria retornar `true` pois estamos levando em consideração o que o `equals` diz, e não o operador `==`.

Em outras palavras, devemos seguir o **contrato da classe Object**, que diz que se:

```
a.equals(b)
```

então:

```
a.hashCode() == b.hashCode()
```

Essa relação não é bidirecional! Se dois elementos têm o mesmo `hashCode`, eles não precisam (mas podem) necessariamente ser o mesmo... é o caso em que há colisão! Já sabemos que quanto menos colisão, melhor para uma distribuição e carga mais balanceada em nossa tabela.

Se não rescrevemos o `equals` não há problema: a implementação padrão do `hashCode` definido na classe `Object` é a de devolver um identificador único para aquele objeto. Como o `equals` definido na classe `Object` se comporta como o operador `==`, é fácil de perceber que o contrato da classe `Object` definido acima será respeitado.

9.18 – Parametrizando o Conjunto

Podemos utilizar o recurso de Generics para parametrizar a nossa classe.

```
package br.com.caelum.ed.conjuntos;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class ConjuntoEspalhamentoParametrizado<T> {
```

```
private List<List<T>> tabela = new ArrayList<List<T>>();

private int tamanho = 0;

public ConjuntoEspalhamentoParametrizado() {
  for (int i = 0; i < 26; i++) {
    LinkedList<T> lista = new LinkedList<T>();
    tabela.add(lista);
  }
}

public void adiciona(T objeto) {
  if (!this.contem(objeto)) {
    this.verificaCarga();
    int indice = this.calculaIndiceDaTabela(objeto);
    List<T> lista = this.tabela.get(indice);
    lista.add(objeto);
    this.tamanho++;
  }
}

public void remove(T objeto) {
  if (this.contem(objeto)) {
    int indice = this.calculaIndiceDaTabela(objeto);
    List<T> lista = this.tabela.get(indice);
    lista.remove(objeto);
    this.tamanho--;
    this.verificaCarga();
  }
}

public List<T> pegaTodos() {
  List<T> objetos = new ArrayList<T>();
  for (int i = 0; i < this.tabela.size(); i++) {
    objetos.addAll(this.tabela.get(i));
  }
  return objetos;
}

public boolean contem(T objeto) {
  int indice = this.calculaIndiceDaTabela(objeto);
  List<T> lista = this.tabela.get(indice);

  return lista.contains(objeto);
}

public int tamanho() {
  return this.tamanho;
}

private int calculaIndiceDaTabela(T objeto) {
  int codigoDeEspalhamento = objeto.hashCode();
  codigoDeEspalhamento = Math.abs(codigoDeEspalhamento);
  return codigoDeEspalhamento % tabela.size();
}
```

```
  private void redimensionaTabela(int novaCapacidade) {
    List<T> objetos = this.pegaTodos();
    this.tabela.clear();

    for (int i = 0; i < novaCapacidade; i++) {
      this.tabela.add(new LinkedList<T>());
    }

    for (T objeto : objetos) {
      this.adiciona(objeto);
    }
  }

  private void verificaCarga() {
    int capacidade = this.tabela.size();
    double carga = (double) this.tamanho / capacidade;

    if (carga > 0.75) {
      this.redimensionaTabela(capacidade * 2);
    } else if (carga < 0.25) {
      this.redimensionaTabela(Math.max(capacidade / 2, 10));
    }
  }

  public void imprimeTabela() {
    for (List<T> lista : this.tabela) {
      System.out.print("[");
      for (int i = 0; i < lista.size(); i++) {
        System.out.print("*");
      }
      System.out.println("]");
    }
  }
}
```

9.19 - API do Java

Na biblioteca do Java há uma classe que faz exatamente o que fizemos neste capítulo, implementar um Conjunto genérico utilizando a técnica de espalhamento. Esta classe é a `HashSet`.

Exemplo de uso:

```
package br.com.caelum.ed.conjuntos;

import java.util.HashSet;

public class TesteHashSet {
  public static void main(String[] args) {
    HashSet conjunto = new HashSet();
    conjunto.add("Rafael");
    conjunto.add("Rafael");
    conjunto.add("Ana");
```

```
    conjunto.add("Paulo");

    System.out.println(conjunto);
  }
}
```

9.20 - Exercícios: Tabela de Espalhamento 4

1. (Generalizando) Faça o método `calculaIndiceDaTabela` utilizar o método `hashCode` e substitua as referências do tipo `String` por `Object`. Assim como foi feito na seção de Generalização.

```
private int calculaIndiceDaTabela(Object objeto) {
  int codigoDeEspalhamento = objeto.hashCode();
  codigoDeEspalhamento = Math.abs(codigoDeEspalhamento);
  return codigoDeEspalhamento % tabela.size();
}
```

Execute o teste:

```
package br.com.caelum.ed.conjuntos;

public class TesteConjuntoGenerico {
  public static void main(String[] args) {
    ConjuntoEspalhamentoGenerico conjunto =
        new ConjuntoEspalhamentoGenerico();
    conjunto.adiciona("Rafael");
    conjunto.adiciona("Rafael");
    conjunto.adiciona("Ana");
    conjunto.adiciona("Paulo");

    System.out.println(conjunto.pegaTodos());
  }
}
```

2. Teste a classe `HashSet`.

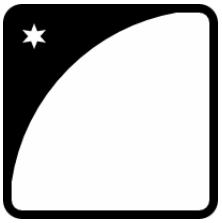
```
package br.com.caelum.ed.conjuntos;

import java.util.HashSet;

public class TesteHashSet {
  public static void main(String[] args) {
    HashSet conjunto = new HashSet();
    conjunto.add("Rafael");
    conjunto.add("Rafael");
    conjunto.add("Ana");
    conjunto.add("Paulo");

    System.out.println(conjunto);
  }
}
```

Você pode também fazer o curso CS-14 dessa apostila na Caelum



Querendo aprender ainda mais sobre estrutura de dados? Esclarecer dúvidas dos exercícios? Ouvir explicações detalhadas com um instrutor?

A Caelum oferece o **curso CS-14** presencial nas cidades de São Paulo, Rio de Janeiro e Brasília, além de turmas incompany.

Consulte as vantagens do curso *Algoritmos e Estruturas de Dados com Java*.

CAPÍTULO ANTERIOR:

Armazenamento sem repetição com busca rápida

PRÓXIMO CAPÍTULO:

Armazenamento Associativo

Facebook

Newsletter

Casa do Código

Twitter