CAPÍTULO 5

Listas Ligadas

"É impossível para um homem aprender aquilo que ele acha que já sabe"
— Epíteto

Vetores (listas implementadas com array) são uma excelente alternativa ao uso direto de arrays em Java, porém para algumas tarefas eles podem não ser eficientes, justamente por esse fato.

Adicionar um elemento na primeira posição de um Vetor, por exemplo, consome muito tempo, pois temos de deslocar todos os outros elementos uma posição para a frente. A performance dessa operação degrada a medida que a quantidade de elementos do nosso vetor cresce: ela consome tempo **linear** em relação ao número de elementos.

Analogamente, remover um elemento da primeira posição implica em deslocar todos os outros elementos que estão na sua frente para trás.

Em alguns casos, queremos uma implementação de Lista na qual a operação de adicionar ou a de remover um aluno na primeira posição seja computacionalmente eficiente. Conseguiremos isso através de uma **Lista Ligada**[/label].

Esta estrutura que vamos criar terá a inserção e remoção nas "pontas" da lista computacionalmente eficientes.

Lembrando que as operações da Lista para o nosso sistema de gerenciamento de alunos são:

1. Adiciona um dado aluno no fim da Lista.
2. Adiciona um dado aluno em uma dada posição.
3. Pega o aluno de dada posição.

4. Remove o aluno de dada posição.

5. Verifica se um dado aluno está armazenado.

6. Informar o número de alunos armazenados.

5.1 – Solução clássica de Lista Ligada

Para manipular uma estrutura sem ter de alterar todos seus vizinhos, precisamos de algo que não seja fixo. Uma array é uma estrutura fixa na memória, precisamos sempre empurrar todos seus vizinhos para um lado para conseguir espaço, ou para remover algum dos elementos, para não deixar espaços vazios.

Então a idéia aqui é ter uma forma de, dado um aluno, saber quem é o próximo, sem usar uma estrutura fixa. Podemos mudar a própria classe `Aluno`! Repare no código abaixo:

```
public class AlunoLista {

  private String nome;

  private int idade;

  private AlunoLista proximo;
}
```

A classe `AlunoLista` possui uma referência para um objeto de seu próprio tipo! É comum aqui o estudante ficar confuso em relação a isso, como se esse código fosse gerar um laço infinito, ou uma recursão infinita. Mas não é o caso, já que a declaração do atributo `proximo` é apenas uma referência, que não cria uma instância de `AlunoLista` na memória!

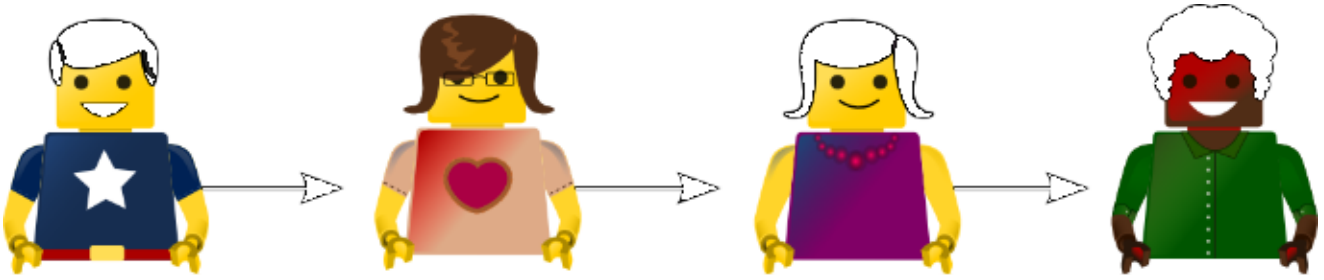


Figura 5.1: Lista Ligada

Repare que, apesar do efeito de um aluno estar "ao lado" do outro, na memória

é mais provável que eles não se encontrem um ao lado do outro, e sim em regiões bem diferentes da memória, só que cada um deles sabe dizer em que local se encontra o próximo aluno (pois temos a referência ao `proximo`).

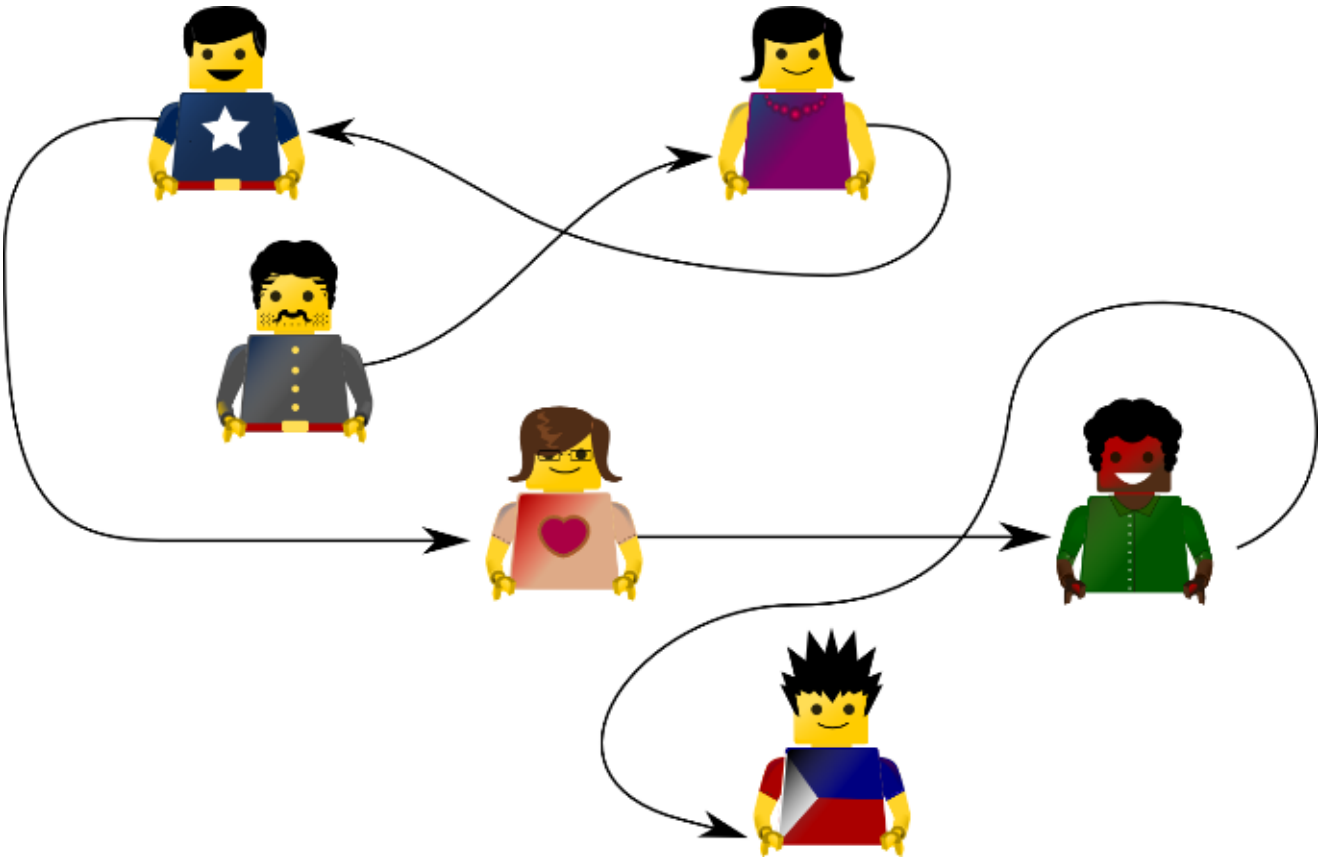


Figura 5.2: Lista Ligada na memória

O ruim de um código assim é que estamos misturando responsabilidades em uma classe muito estranha! A `AlunoLista` além de guardar informações sobre um aluno, guarda também uma Lista de alunos. Manipulá-la pode ficar estranho e trabalhoso. Precisaríamos sempre manter uma referência para o primeiro aluno na memória em algum outro lugar, o que também pode ser confuso!

Além disso, quando precisarmos ter uma Lista Ligada de peças de carro, precisaríamos criar uma classe PecaLista parecidíssima com a AlunoLista, e não estaríamos reaproveitando código (além de ficar suscetível a copiar e colar modificações, uma péssima prática de programação, que pode gerar muitos erros).

Em vez de usar essa estrutura engessada, vamos utilizar uma classe separada como célula , evitando a mistura da nossa classe de dados (a `Aluno`) da nossa estrutura.

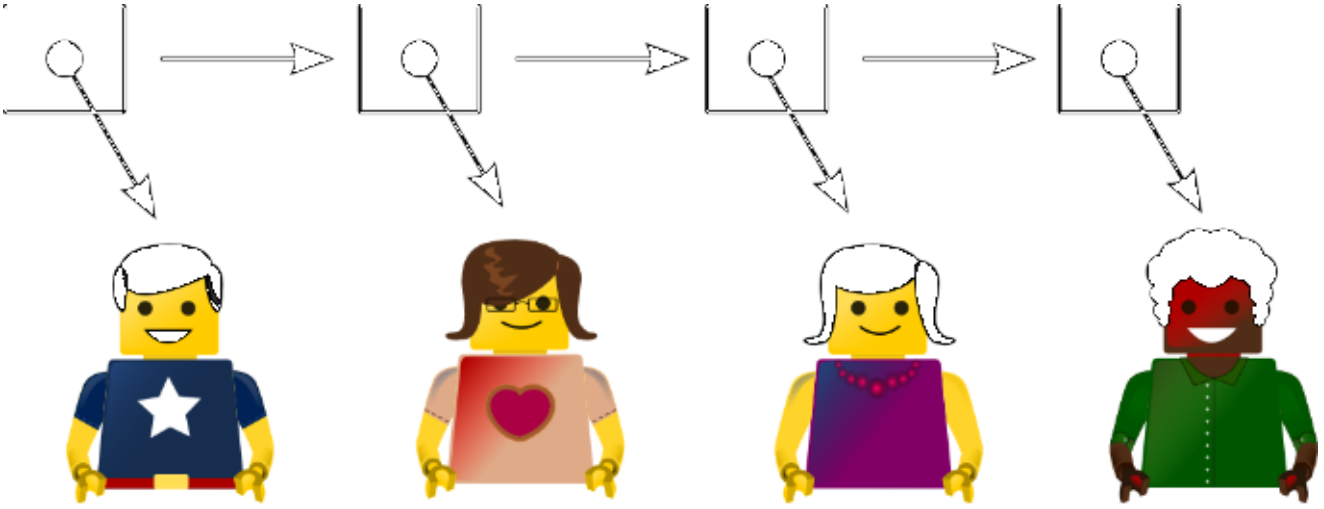


Figura 5.3: Lista Ligada com Célula

5.2 – Célula e Lista Ligada

Vamos então isolar a manipulação de nossa estrutura de dados, para isso teremos uma classe para representar uma célula. Ela deve possuir uma referência para o elemento a qual ela se refere, e uma referência para a próxima célula , que pode ser **null** caso essa seja a última célula da Lista.

A classe `Celula` a seguir possui apenas esses dois atributos, além de getters, setters e de construtores para facilitar nosso trabalho.

```
public class Celula {

  private Celula proxima;

  private Object elemento;

  public Celula(Celula proxima, Object elemento) {
    this.proxima = proxima;
    this.elemento = elemento;
  }

  public Celula(Object elemento) {
    this.elemento = elemento;
  }

  public void setProxima(Celula proxima) {
    this.proxima = proxima;
  }

  public Celula getProxima() {
    return proxima;
  }
```

```
  public Object getElemento() {
    return elemento;
  }
}
```

Essa classe `Celula` nunca será acessada diretamente por quem usar a classe `ListaLigada`, dessa forma estamos escondendo como a nossa classe funciona e garantindo que ninguém mexa na estrutura interna da nossa Lista.

A classe `ListaLigada` possui inicialmente apenas uma referência para a primeira célula e outra para a última. Através da primeira conseguimos iterar até chegar em qualquer posição necessária. Já a referência à última vai facilitar a inserção no fim da Lista.

```
public class ListaLigada {

  private Celula primeira;

  private Celula ultima;

}
```


5.3 - Definindo a interface

Com as operações definidas podemos esboçar a classe `ListaLigada`.

```
public class ListaLigada {

  private Celula primeira;

  private Celula ultima;

  public void adiciona(Object elemento) {}
```

```
  public void adiciona(int posicao, Object elemento) {}
  public Object pega(int posicao) {return null;}
  public void remove(int posicao){}
  public int tamanho() {return 0;}
  public boolean contem(Object o) {return false;}
}
```

Estamos discutindo sobre Lista Ligada especialmente porque queremos um tipo de Lista que as operações de adicionar e remover da primeira e da última posição sejam computacionalmente eficientes.

Então, vamos acrescentar mais três métodos na classe `ListaLigada`.

```
public void adicionaNoComeco(Object elemento) {}
public void removeDoComeco() {}
public void removeDoFim() {}
```

O método `adiciona()` insere no fim da Lista então não adicionaremos mais um método para realizar essa mesma tarefa.

5.4 – Testes

```
public class TesteAdicionaNoFim {
  public static void main(String[] args) {
    ListaLigada lista = new ListaLigada();

    lista.adiciona("Rafael");
    lista.adiciona("Paulo");

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

Saída esperada:

```
[Rafael, Paulo]
```

```
public class TesteAdicionaPorPosicao {
  public static void main(String[] args) {
    ListaLigada lista = new ListaLigada();
    lista.adiciona("Rafael");
    lista.adiciona(0, "Paulo");
    lista.adiciona(1, "Camila");

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

Saída esperada:

```
[Paulo, Camila, Rafael]
```

```
public class TestePegaPorPosicao {
  public static void main(String[] args) {
    ListaLigada lista = new ListaLigada();

    lista.adiciona("Rafael");
    lista.adiciona("Paulo");

    System.out.println(lista.pega(0));
    System.out.println(lista.pega(1));
  }
}
```

Saída esperada:

```
Rafael

Paulo
```

```
public class TesteRemovePorPosicao {
  public static void main(String[] args) {
    ListaLigada lista = new ListaLigada();

    lista.adiciona("Rafael");
    lista.adiciona("Paulo");
    lista.adiciona("Camila");

    lista.remove(1);

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

Saída esperada:

```
[Rafael, Camila]
```

```
public class TesteTamanho {
  public static void main(String[] args) {
    ListaLigada lista = new ListaLigada();

    lista.adiciona("Rafael");
    lista.adiciona("Paulo");

    System.out.println(lista.tamanho());

    lista.adiciona("Camila");

    System.out.println(lista.tamanho());
  }
}
```

Saída esperada:

```
2

3
```

```
public class TesteContemElemento {
  public static void main(String[] args) {
    ListaLigada lista = new ListaLigada();

    lista.adiciona("Rafael");
    lista.adiciona("Paulo");

    System.out.println(lista.contem("Rafael"));
    System.out.println(lista.contem("Camila"));
  }
}
```

Saída esperada:

```
true

false
```

```
public class TesteAdicionaNoComeco {
  public static void main(String[] args) {
    ListaLigada lista = new ListaLigada();

    lista.adicionaNoComeco("Rafael");
    lista.adicionaNoComeco("Paulo");

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

Saída esperada:

```
[Paulo, Rafael]
```

```
public class TesteRemoveDoComeco {
  public static void main(String[] args) {
    ListaLigada lista = new ListaLigada();

    lista.adiciona("Rafael");
    lista.adiciona("Paulo");

    lista.removeDoComeco();

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

Saída esperada:

```
[Paulo]
```

```
public class TesteRemoveDoFim {
  public static void main(String[] args) {
    ListaLigada lista = new ListaLigada();

    lista.adiciona("Rafael");
    lista.adiciona("Paulo");
```

```
    lista.removeDoFim();

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

Saída esperada:

```
[Rafael]
```

5.5 – Operações sobre uma Lista

Aqui vamos começar a aumentar o código da nossa `ListaLigada` baseado nas operações que precisamos realizar com ela.

Para facilitar a contagem de elementos e algumas operações, vamos definir um atributo **int** `totalDeElementos`.

5.6 – Adicionando no começo da Lista

Inserir no começo da Lista é bastante trivial, basta criarmos uma nova célula, e esta nova célula terá a referência `proxima` apontando para a atual `primeira` da lista. Depois atualizamos o atributo `primeira` para se referenciar a esta nova célula recém criada.

Ainda falta tratar o caso especial da Lista estar vazia. Neste caso, devemos atualizar a referência que aponta para a última célula também. Através do atributo `totalDeAlunos` podemos identificar este caso.

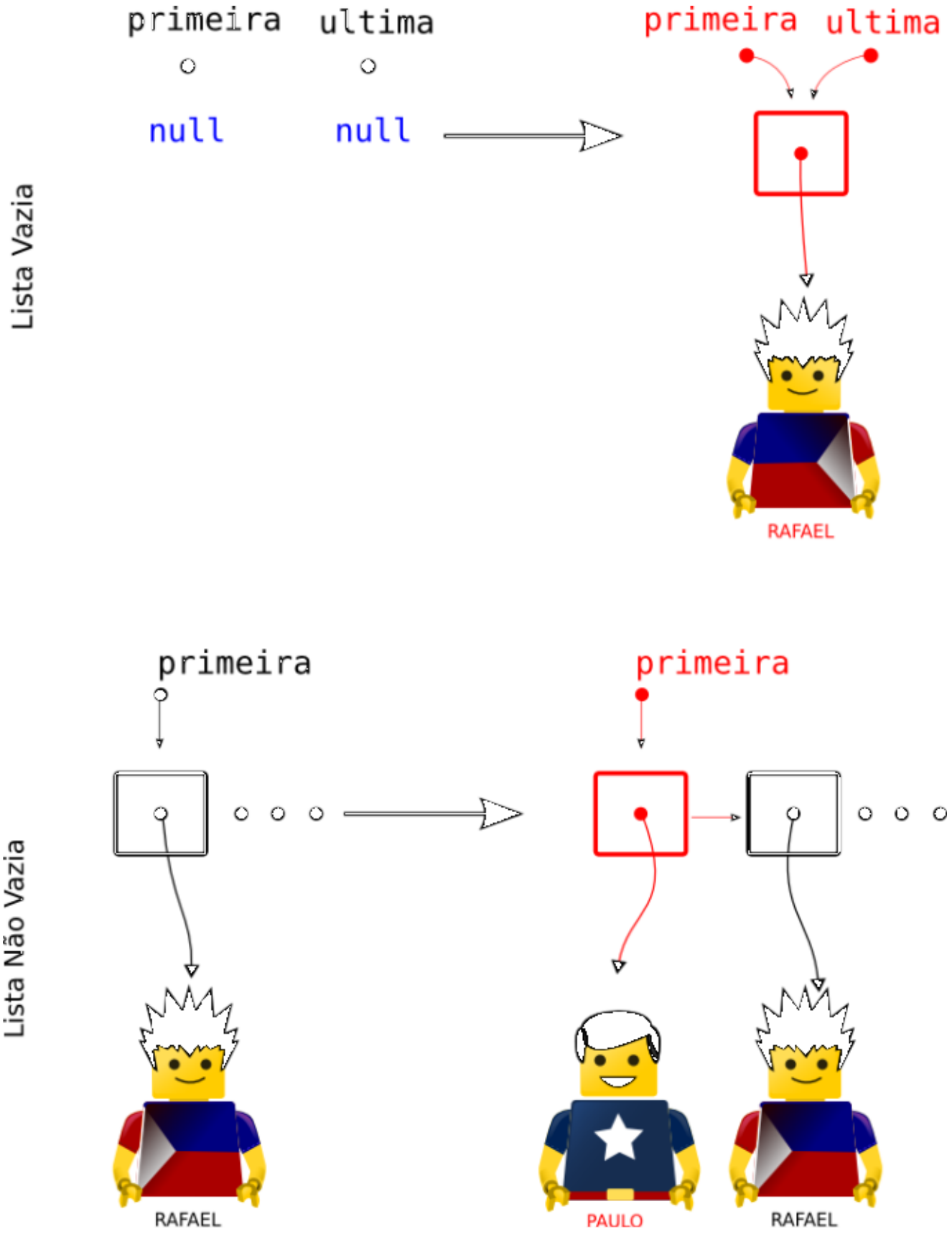


Figura 5.4: Os dois casos para adicionar no começo

```
public class ListaLigada {

  private Celula primeira;
```

```
  private Celula ultima;

  private int totalDeElementos;

  public void adicionaNoComeco(Object elemento) {
    Celula nova = new Celula(this.primeira, elemento);
    this.primeira = nova;

    if(this.totalDeElementos == 0){
      // caso especial da lista vazia
      this.ultima = this.primeira;
    }
    this.totalDeElementos++;
  }
}
```

5.7 - Adicionando no fim da Lista

Se não tivéssemos guardado a referência para a última célula precisaríamos percorrer célula a célula até o fim da Lista para alterar a referência `proxima` da última célula! Com um grande número de elementos isso ficaria lento, pois leva tempo linear.

No caso especial da Lista estar vazia, adicionar no começo ou no fim dessa lista dá o mesmo efeito. Então, se a Lista estiver vazia, chamaremos o método `adicionaNoComeco(Object)`.

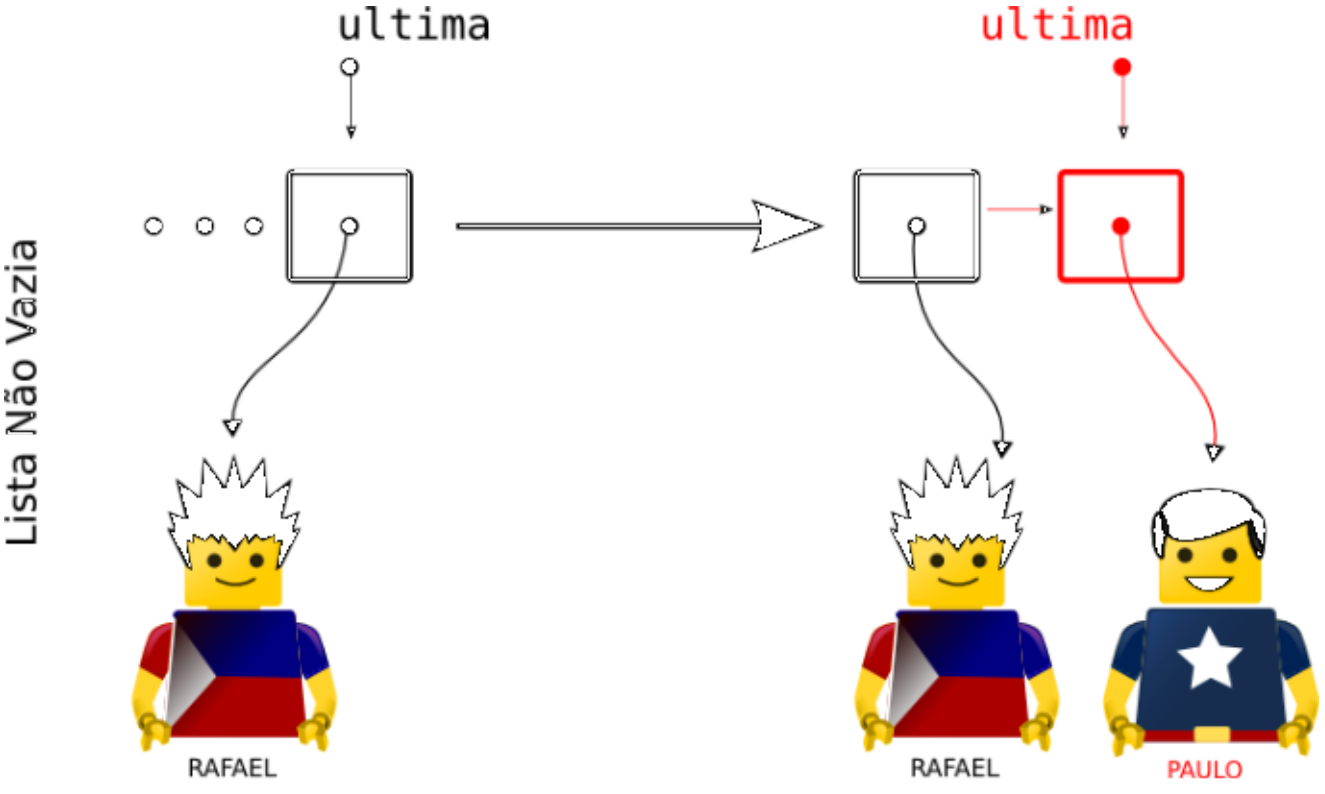


Figura 5.5: Adicionando no fim

```
public class ListaLigada {

  private Celula primeira;

  private Celula ultima;

  private int totalDeElementos;

  public void adiciona(Object elemento) {
    if (this.totalDeElementos == 0) {
      this.adicionaNoComeco(elemento);
    } else {
      Celula nova = new Celula(elemento);
      this.ultima.setProxima(nova);
      this.ultima = nova;
      this.totalDeElementos++;
    }
  }
}
```

Repare que aqui estamos fazendo que haja `n` células para `n` elementos, isto é, uma célula para cada elemento. Uma outra implementação clássica de lista ligada é usar uma célula **sentinela** a mais para indicar o começo da Lista, e outra para o fim, assim poderíamos simplificar um pouco alguns desses métodos, como o caso particular de inserir um elemento quando não há ainda elemento algum. Vale sempre lembrar que aqui estamos estudando uma implementação de uma estrutura de dados, e que há sempre outras formas de codifica-las, que podem ser mais ou menos elegantes.

5.8 - Percorrendo nossa Lista

Para poder rodar alguns testes, precisamos imprimir o conteúdo da nossa Lista. Como ainda não temos como acessar um elemento pela sua posição, vamos reescrever o método `toString()` da classe `ListaLigada` para que ele concatene todos os elementos de dentro da Lista Ligada em uma única `String`.

O nosso método utiliza-se de uma referência temporária para uma célula chamada atual, que vai apontando para a próxima a cada iteração, concatenando o elemento dessa célula:

```
public String toString() {

  // Verificando se a Lista está vazia
  if(this.totalDeElementos == 0){
    return "[]";
  }
```

```
  StringBuilder builder = new StringBuilder("[");
  Celula atual = primeira;

  // Percorrendo até o penúltimo elemento.
  for (int i = 0; i < this.totalDeAlunos - 1; i++) {
    builder.append(atual.getElemento());
    builder.append(", ");
    atual = atual.getProxima();
  }

  // último elemento
  builder.append(atual.getElemento());
  builder.append("]");

  return builder.toString();
}
```

Aqui estamos utilizando a classe `StringBuilder` que é muito útil para criar Strings potencialmente grandes, em vez de concatenar Strings pelo operador + [/label].

Também poderíamos ter utilizado um `while(atual != null)` em vez do `for` dentro do método `toString`, que as vezes pode ser mais legível.

5.9 - Adicionando em qualquer posição da Lista

A inserção no começo e no fim da Lista já foram tratadas nas seções anteriores. Aqui vamos nos preocupar em inserir em uma posição do interior da Lista (qualquer posição que não é a primeira e nem a última).

Para inserir um elemento em qualquer posição precisamos pegar a célula

anterior a da posição desejada, porque precisamos mexer na sua referência `proxima`. Para isso vamos criar um método auxiliar que pega determinada célula. Este método deve tomar cuidado com o caso da posição não existir. A verificação se a posição existe ou não é feita pelo método `posicaoOcupada(int)`.

```
private boolean posicaoOcupada(int posicao){
  return posicao >= 0 && posicao < this.totalDeElementos;
}

private Celula pegaCelula(int posicao) {
  if(!this.posicaoOcupada(posicao)){
    throw new IllegalArgumentException("Posição não existe");
  }

  Celula atual = primeira;
  for (int i = 0; i < posicao; i++) {
    atual = atual.getProxima();
  }
  return atual;
}
```

Os métodos são privados pois não queremos que ninguém de fora tenha acesso ao funcionamento interno da nossa estrutura de dados. É importante notar que o método `pegaCelula(int)` consome tempo linear!

Agora o método `adiciona(int, Object)` fica simples de ser feito. Basta pegar a célula anterior a posição onde a inserção será feita e atualizar as referências. A anterior deve apontar para uma nova célula e a nova célula deve apontar para a antiga próxima da anterior.

Devemos tomar cuidado com os casos particulares nos quais a posição para inserir é o começo ou o fim da Lista.

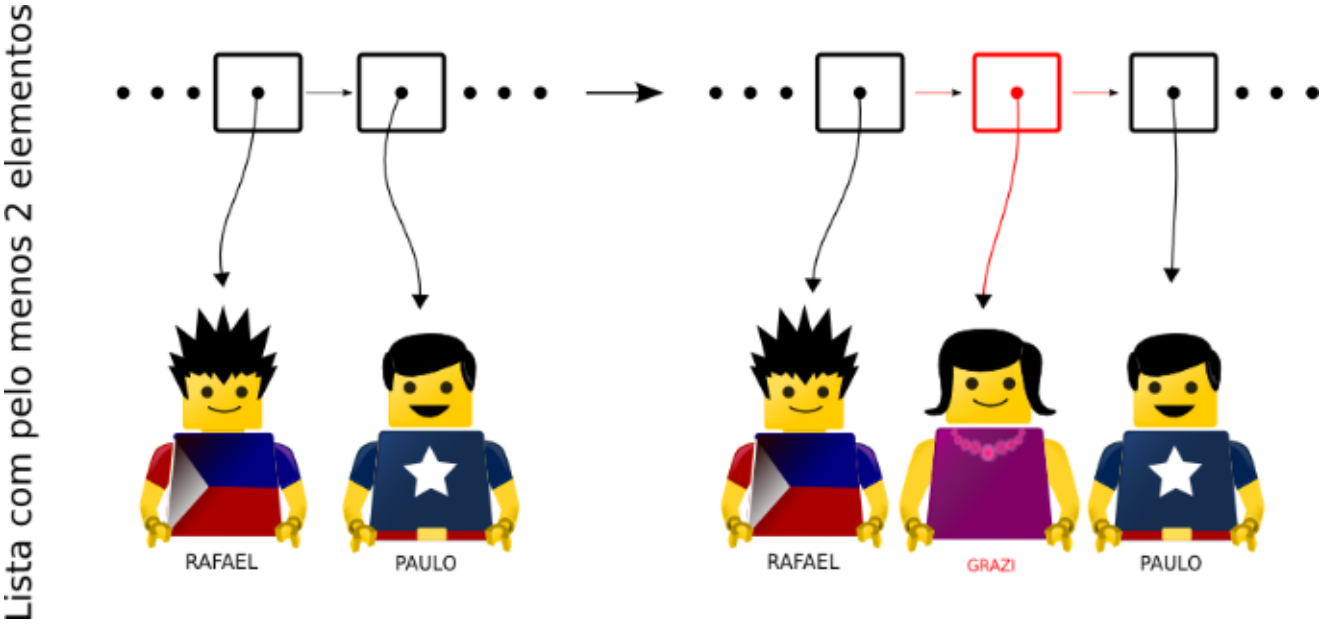


Figura 5.6: Adicionando no interior da Lista

```
public void adiciona(int posicao, Object elemento) {
  if(posicao == 0){ // No começo.
    this.adicionaNoComeco(elemento);
  } else if(posicao == this.totalDeElementos){ // No fim.
    this.adiciona(elemento);
  } else {
    Celula anterior = this.pegaCelula(posicao - 1);
    Celula nova = new Celula(anterior.getProxima(), elemento);
    anterior.setProxima(nova);
    this.totalDeElementos++;
  }
}
```

5.10 – Pegando um elemento da Lista

Para pegar um elemento é muito fácil: basta pegarmos a célula em que aquele elemento se encontra e acessar o elemento de dentro dela. Podemos utilizar o `pegaCelula(int)` previamente criado:

```
public Object pega(int posicao) {
  return this.pegaCelula(posicao).getElemento();
}
```

Perceba que este método consome tempo linear. Esta é uma **grande** desvantagem da Lista Ligada em relação aos Vetores. Vetores possuem o chamado **acesso aleatório** aos elementos: qualquer posição pode ser acessada em tempo constante. Apesar dessa grande desvantagem, diversas vezes utilizamos uma Lista e não é necessário ficar acessando posições aleatórias: comumente percorremos a lista por completa, que veremos como fazer mais adiante.

5.11 – Removendo do começo da Lista

Antes de tentar remover devemos verificar se a posição está ocupada. Não faz sentido remover algo que não existe. Depois, basta "avançar" a referência que aponta para a primeira célula.

Por fim, é importante perceber que a Lista pode ficar vazia. Neste caso, devemos colocar **null** na referência que aponta para a última célula.

Se não fizermos isso ficaríamos em um estado inconsistente, em que o atributo

`primeira` é `null` e o `última` não, ou seja, tem uma última mas não tem uma primeira. Isso não faria sentido.

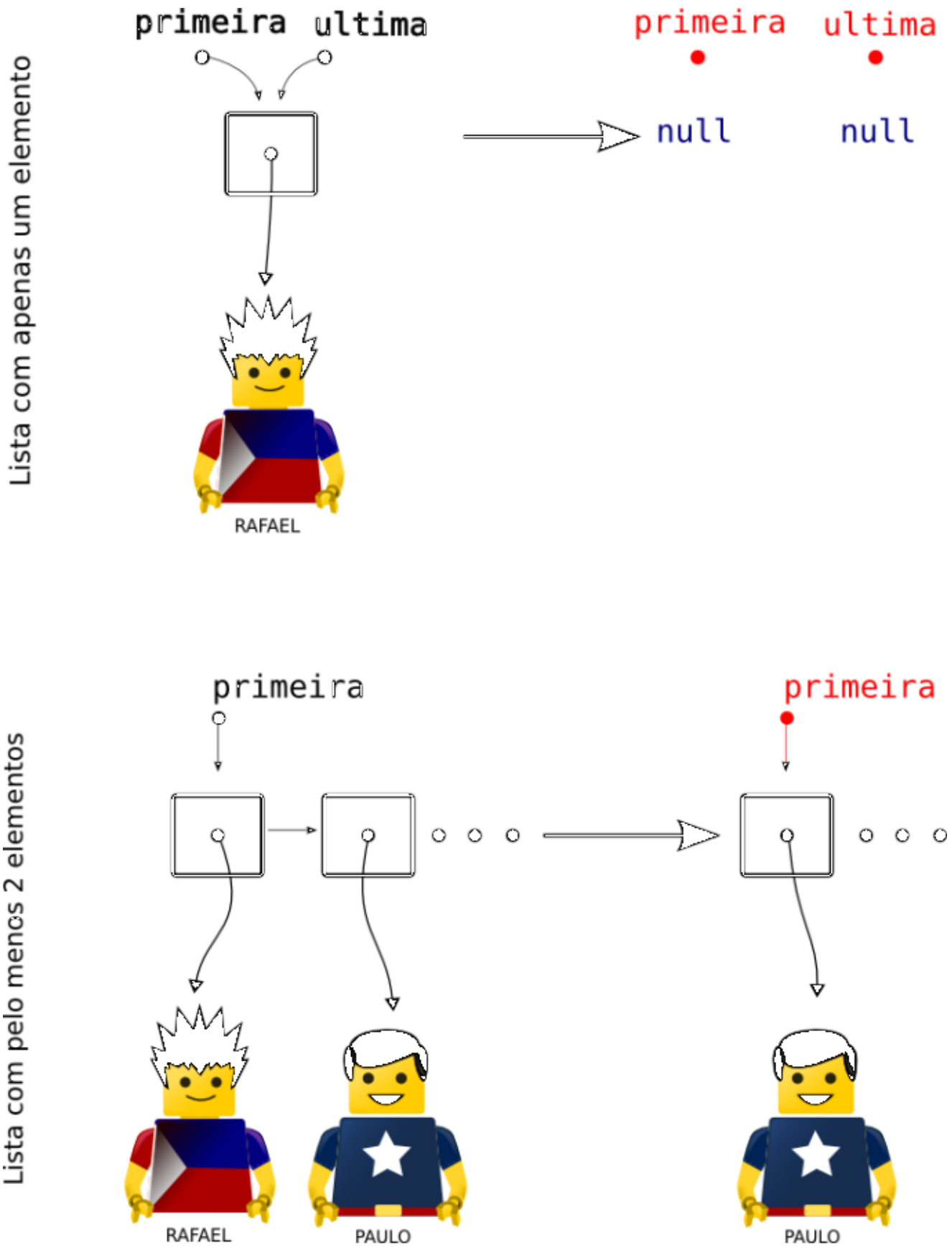


Figura 5.7: Os dois casos de remover do começo

```
public void removeDoComeco() {
  if (!this.posicaoOcupada(0)) {
    throw new IllegalArgumentException("Posição não existe");
  }

  this.primeira = this.primeira.getProxima();
  this.totalDeElementos--;

  if (this.totalDeElementos == 0) {
    this.ultima = null;
  }
}
```


5.12 - Removendo do fim da Lista

A primeira verificação a ser feita é se a última posição existe. Podemos fazer isso através do método já criado `posicaoOcupada(int)`.

Se a Lista estiver com apenas um elemento então remover do fim é a mesma coisa que remover do começo. Logo, podemos reutilizar o método `removeDoComeco()` para este caso.

Agora, se a Lista tem mais que um elemento então devemos pegar a penúltima célula; fazer a próxima da penúltima ser **null**; e fazer o atributo `ultima` apontar para a penúltima.

O problema aqui é como pegar a penúltima célula. Podemos fazer isso usando o `pegaCelula(int)` mas isso consumiria tempo linear. Como queremos consumo constate teremos que achar outra solução.

Então, em vez de fazer uma Lista Ligada simples vamos fazer uma Lista Duplamente Ligada. Ou seja, cada célula aponta para a sua anterior além de

apontar para a próxima:

```
public class Celula {

  ...

  private Celula anterior;

  ...

  public Celula getAnterior() {
    return anterior;
  }

  public void setAnterior(Celula anterior) {
    this.anterior = anterior;
  }
}
```

Com cada célula sabendo também quem é a sua anterior, fica fácil escrever o método `removeDoFim()`

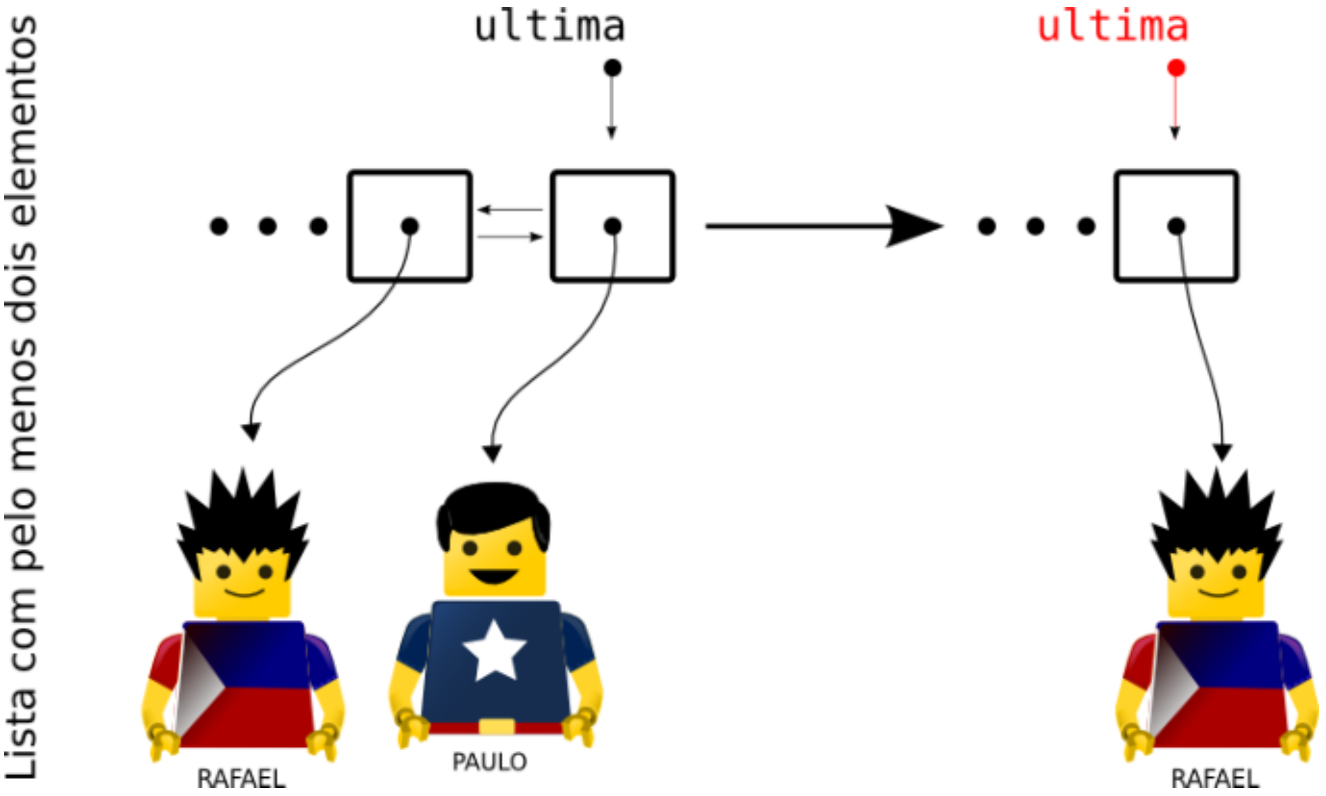


Figura 5.8: Removendo do fim da Lista

```
public void removeDoFim() {
  if (!this.posicaoOcupada(this.totalDeElementos - 1)) {
    throw new IllegalArgumentException("Posição não existe");
  }
  if (this.totalDeElementos == 1) {
    this.removeDoComeco();
```

```
  } else {
    Celula penultima = this.ultima.getAnterior();
    penultima.setProxima(null);
    this.ultima = penultima;
    this.totalDeElementos--;
  }
}
```

A modificação para Lista Duplamente Ligada implicará em pequenas modificações nos outros métodos que já tínhamos implementado. As modificações serão apresentadas na seção de Lista Duplamente Ligada.

5.13 – Removendo de qualquer posição

Inicialmente, devemos verificar se a posição está ou não ocupada. Se não estiver devemos lançar uma exceção. Caso contrário, devemos verificar se a remoção é do começo ou do fim da Lista se for um destes casos simplesmente chamamos os métodos que já fizemos.

Por fim, se a remoção é no interior da Lista devemos atualizar as referências das células relacionadas a célula que vamos remover (anterior e próxima). A próxima da anterior deve ser a próxima e a anterior da próxima deve ser a anterior.

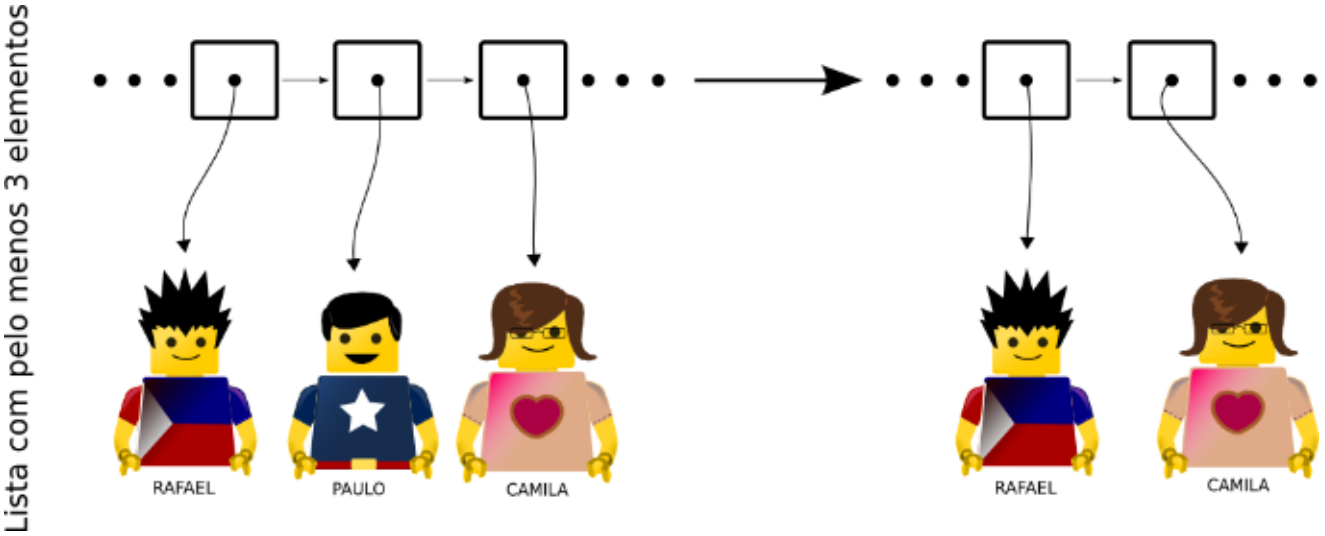


Figura 5.9: Removendo do interior da Lista

```
public void remove(int posicao) {
  if (!this.posicaoOcupada(posicao)) {
    throw new IllegalArgumentException("Posição não existe");
  }

  if (posicao == 0) {
    this.removeDoComeco();
```

```
  } else if (posicao == this.totalDeElementos - 1) {
    this.removeDoFim();
  } else {
    Celula anterior = this.pegaCelula(posicao - 1);
    Celula atual = anterior.getProxima();
    Celula proxima = atual.getProxima();

    anterior.setProxima(proxima);
    proxima.setAnterior(anterior);

    this.totalDeElementos--;
  }
}
```

5.14 – Verificando se um elemento está na Lista

Está operação deve percorrer a Lista e comparar com o método `equals(Object)` o elemento procurado contra todos os elementos da Lista.

```
public boolean contem(Object elemento) {
  Celula atual = this.primeira;

  while (atual != null) {
    if (atual.getElemento().equals(elemento)) {
      return true;
    }
    atual = atual.getProxima();
  }
  return false;
}
```


5.15 – O tamanho da Lista

Está operação não tem segredo, pois já temos um atributo que possui esta informação.

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

5.16 – Lista Duplamente Ligada

Para o método que remove o elemento do fim da Lista ter consumo de tempo constante, a última célula deve se referir a penúltima. Isso é possível se utilizarmos o esquema de Lista Duplamente Ligada, no qual as células possuem referências tanto para a próxima e quanto para a célula anterior.

Esta modificação implica alterações na implementação dos outros métodos, como já falamos anteriormente. Seguem:

5.17 – Adicionando no começo da Lista

Temos que considerar dois casos: Lista Vazia e Lista não Vazia.

Se a Lista está vazia então a nova célula será a primeira e a última. Além disso, ela não terá próxima nem anterior pois ela será a única célula.

Se a Lista não está vazia então devemos ajustar os ponteiros para a nova segunda (antiga referência `primeira`) apontar para a nova `primeira` e vice versa.

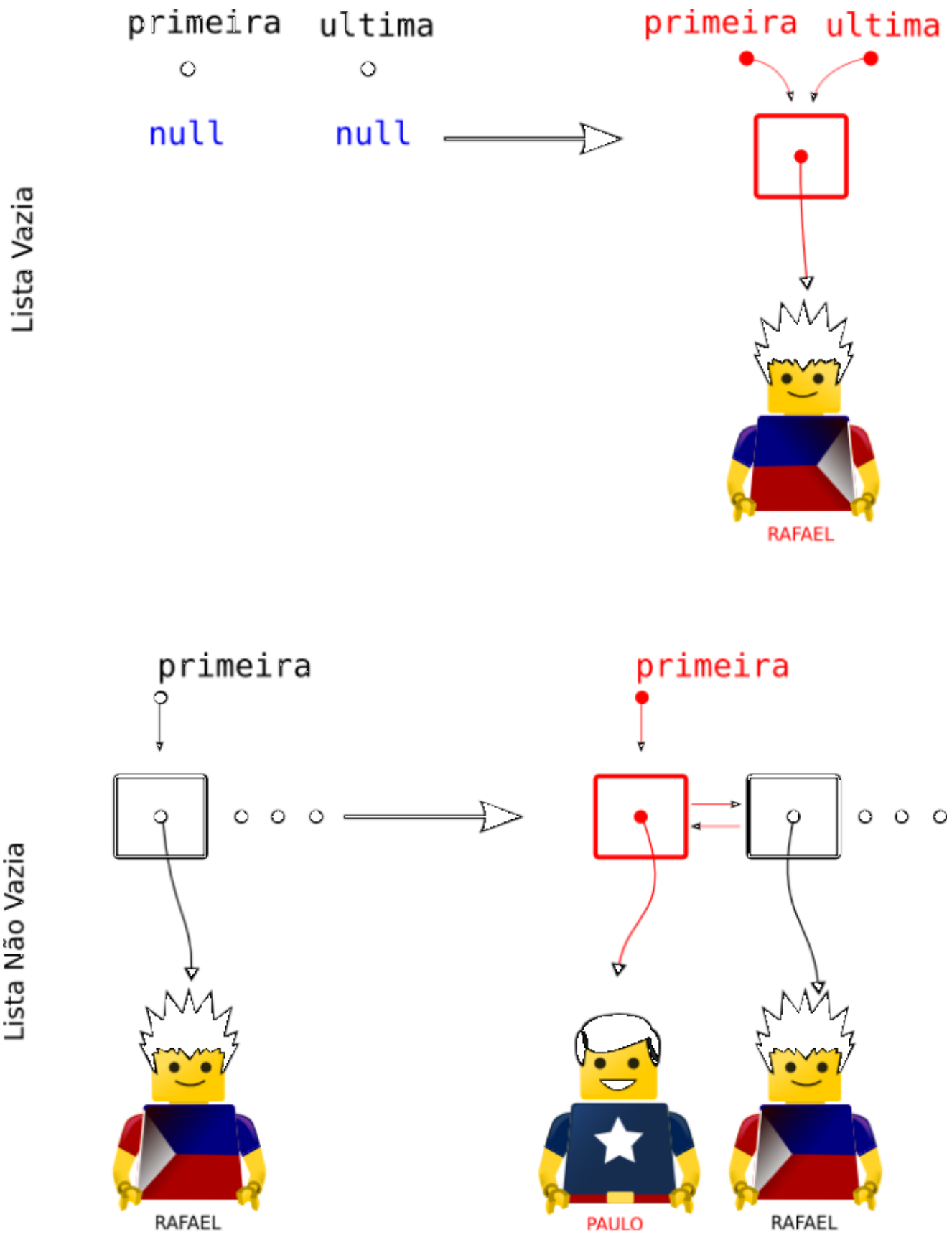


Figura 5.10: Adicionado no começo da Lista Duplamente Ligada

```
public void adicionaNoComeco(Object elemento) {
  if(this.totalDeElementos == 0){
    Celula nova = new Celula(elemento);
    this.primeira = nova;
```

```
    this.ultima = nova;
  } else {
    Celula nova = new Celula(this.primeira, elemento);
    this.primeira.setAnterior(nova);
    this.primeira = nova;
  }
  this.totalDeElementos++;
}
```


5.18 – Adicionando no fim da Lista

No caso em que a Lista está vazia, adicionar no fim é a mesma coisa que adicionar no começo.

Agora, caso a Lista não esteja vazia então devemos ajustar as referências de tal forma que a nova última célula aponte para a nova penúltima (antiga última) e vice versa.

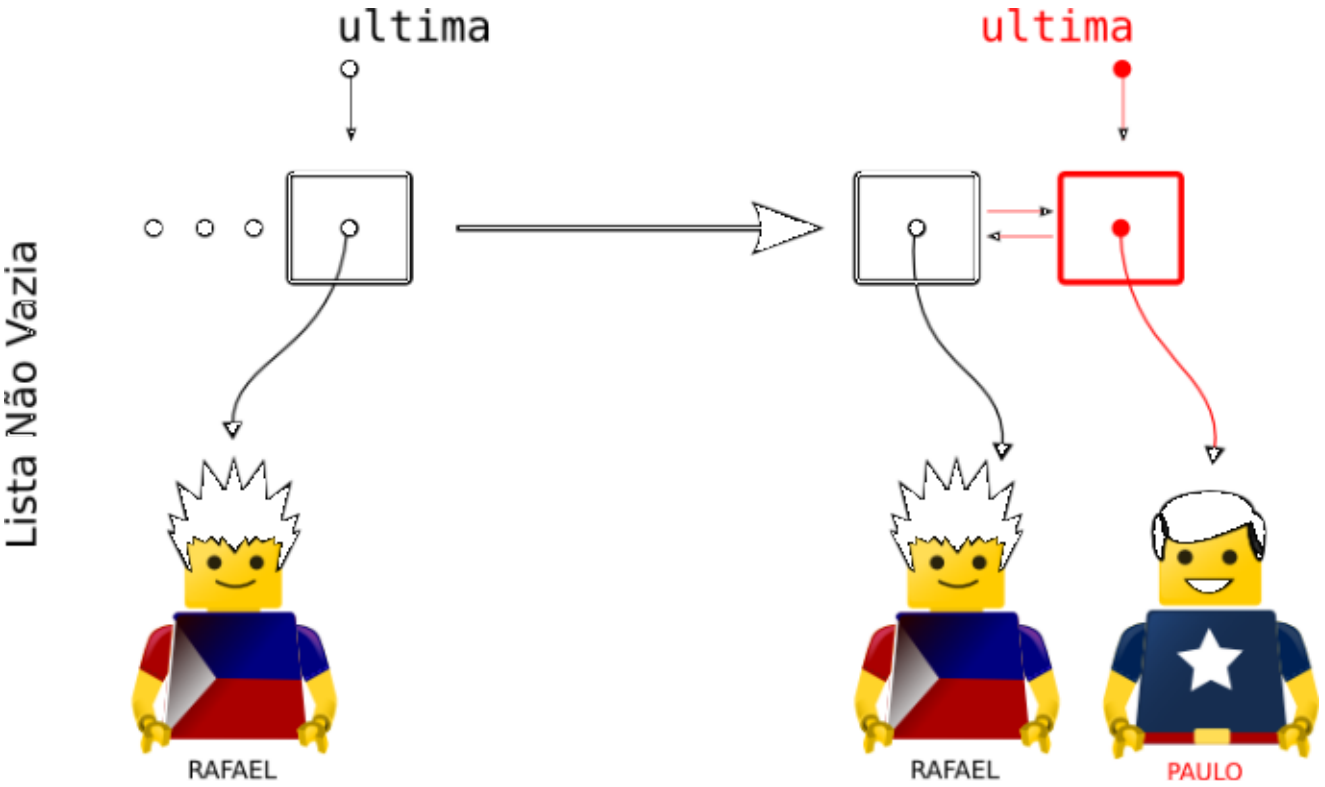


Figura 5.11: Adicionado no fim da Lista Duplamente Ligada

```
public void adiciona(Object elemento) {
  if (this.totalDeElementos == 0) {
    this.adicionaNoComeco(elemento);
  } else {
    Celula nova = new Celula(elemento);
    this.ultima.setProxima(nova);
    nova.setAnterior(this.ultima);
    this.ultima = nova;
    this.totalDeElementos++;
  }
}
```

5.19 – Adicionando em qualquer posição da Lista

Separamos os casos em que a inserção é no começo ou no fim porque podemos reaproveitar os métodos já implementados.

Sobra o caso em que a inserção é no meio da Lista, ou seja, entre duas células existentes. Neste caso, devemos ajustar as referências para a nova célula ser apontada corretamente pela duas células relacionadas a ela (a anterior e a próxima). E também fazer a nova célula apontar para a anterior e a próxima.

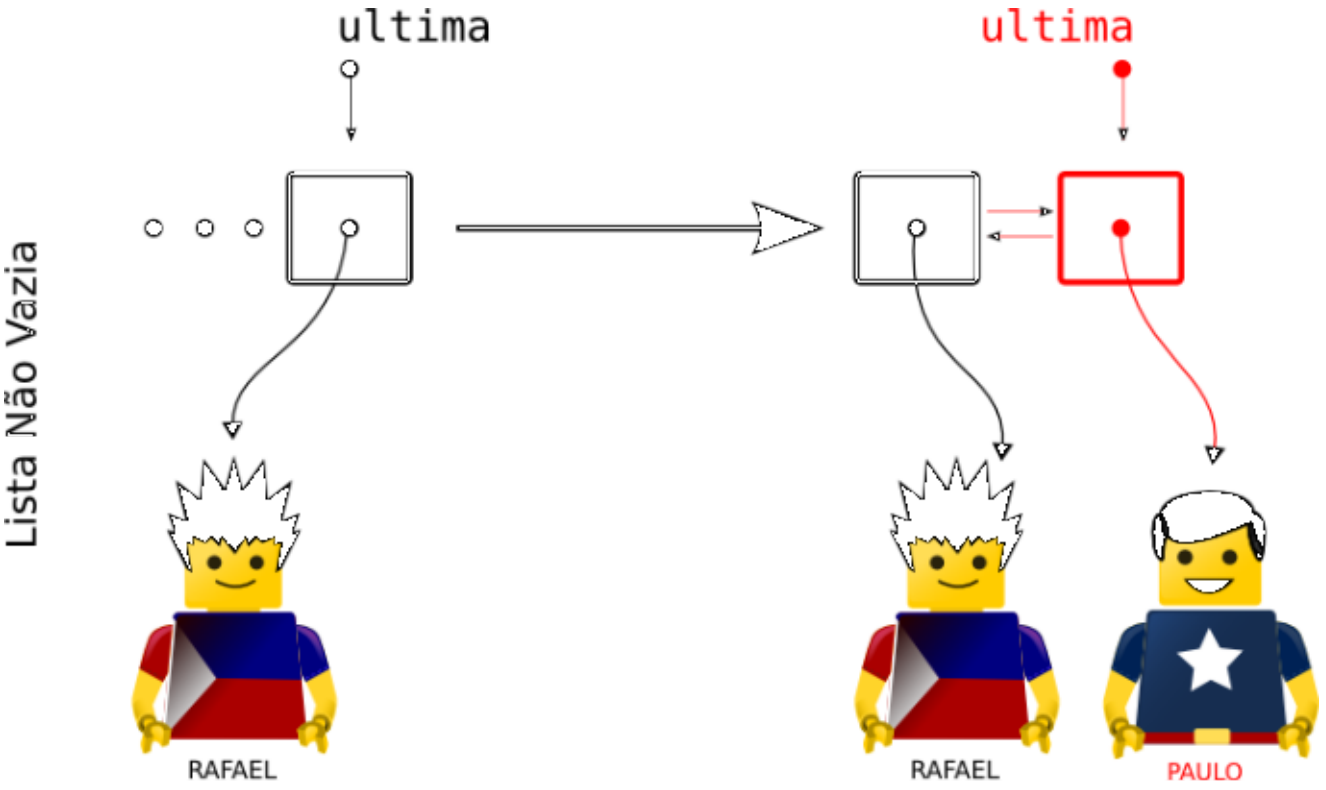


Figura 5.12: Adicionado no fim da Lista Duplamente Ligada

```
public void adiciona(int posicao, Object elemento) {
  if(posicao == 0){ // No começo.
    this.adicionaNoComeco(elemento);
  } else if(posicao == this.totalDeElementos){ // No fim.
    this.adiciona(elemento);
  } else {
    Celula anterior = this.pegaCelula(posicao - 1);
    Celula proxima = anterior.getProxima();
    Celula nova = new Celula(anterior.getProxima(), elemento);
    nova.setAnterior(anterior);
    anterior.setProxima(nova);
    proxima.setAnterior(nova);
    this.totalDeElementos++;
  }
}
```

5.20 – Removendo do começo da Lista

Esta operação é idêntica em ambos os tipos de Lista Ligada (simples ou dupla). Ela apenas deve avançar a referência primeira para a segunda célula e tomar cuidado com o caso da Lista ficar vazia pois, neste caso, a referência última deve ser atualizada também.

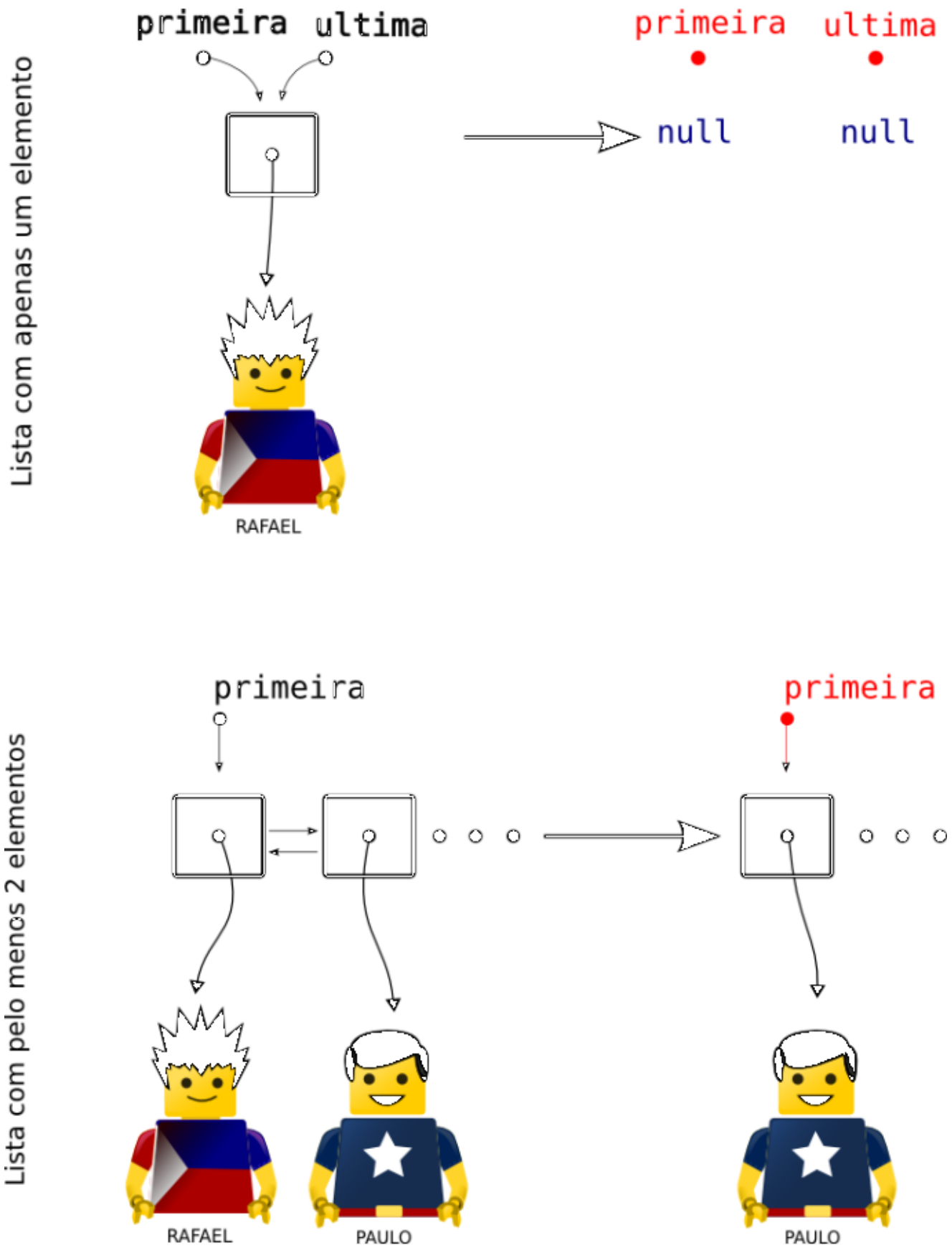


Figura 5.13: Removendo do começo da Lista Duplamente Ligada

Agora é a melhor hora de aprender algo novo

5.21 – Removendo do fim da Lista ou de qualquer posição

Estas duas operações já foram tratadas e implementadas anteriormente usando o esquema de Lista Duplamente Ligada.

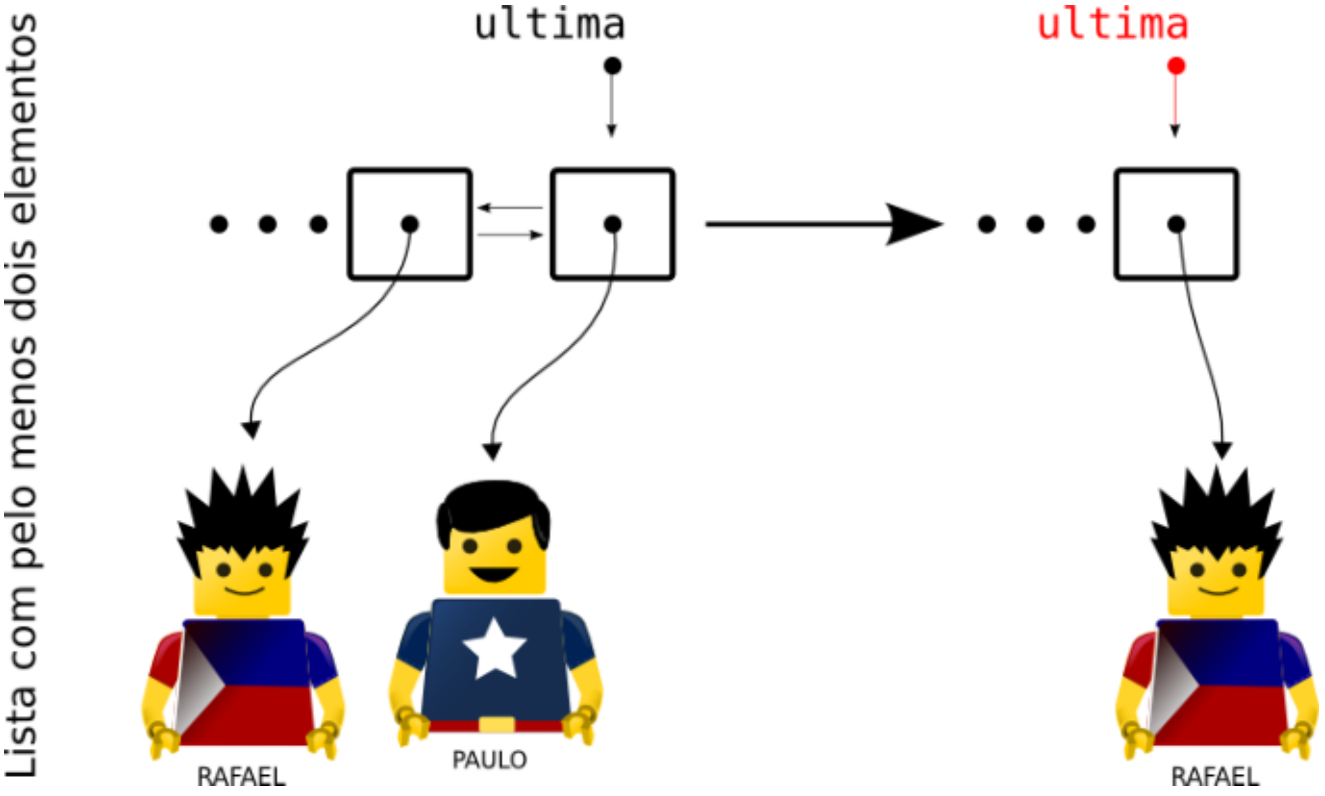


Figura 5.14: Removendo do fim da Lista Duplamente Ligada

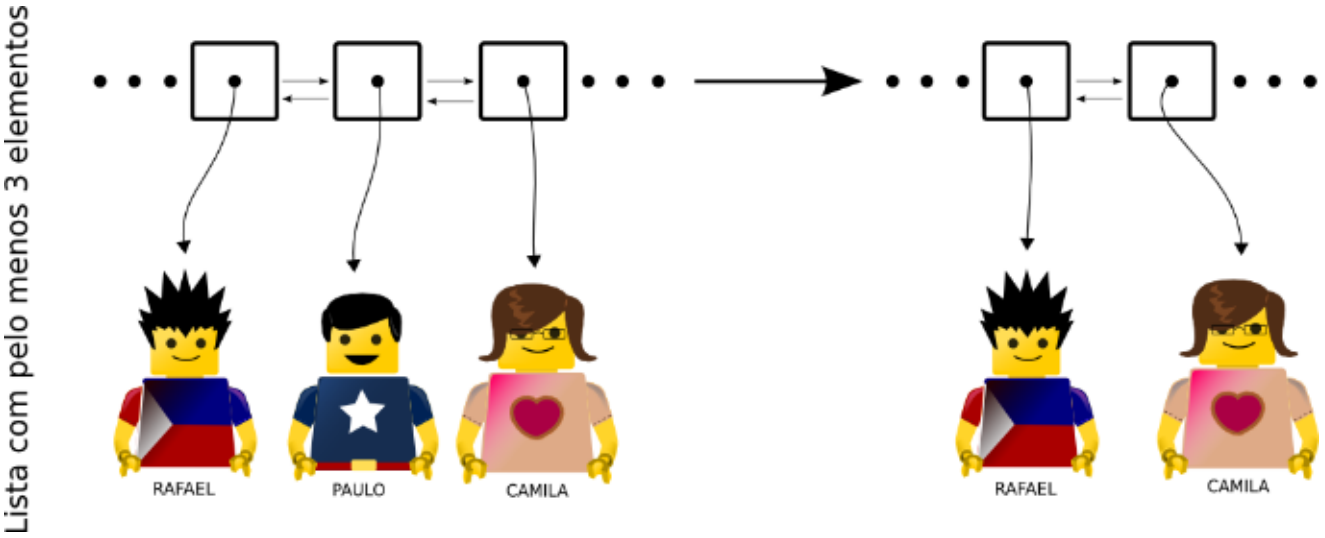


Figura 5.15: Removendo no interior da Lista Duplamente Ligada

5.22 - API

A classe `LinkedList` faz o papel da nossa Lista Ligada dentro da biblioteca do Java. Ela possui os mesmos métodos que a `ArrayList`, e adiciona alguns outros, como o `addFirst(Object)`, `removeFirst()`, `addLast(Object)` e `removeLast()`, que operam no começo e no fim da Lista em tempo constante.

```
public class Teste {

  public static void main(String[] args) {
    Aluno maria = new Aluno();
    maria.setNome("Maria");

    Aluno manoel = new Aluno();
    manoel.setNome("Manoel");

    Aluno joaquim = new Aluno();
    joaquim.setNome("Joaquim");

    LinkedList listaLigada = new LinkedList();

    listaLigada.add(maria);
    listaLigada.add(manoel);
    listaLigada.add(1, joaquim);

    for (int i = 0; i < listaLigada.size(); i++) {
      System.out.println(listaLigada.get(i));
    }
  }
}
```

5.23 – Exercícios: Lista Ligada

1. Implemente a estrutura Célula através de uma classe no pacote **br.com.caelum.ed.listasligadas**.

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

public class Celula {
  private Celula proxima;

  private Celula anterior;

  private Object elemento;

  public Celula(Celula proxima, Object elemento) {
    this.proxima = proxima;
    this.elemento = elemento;
  }

  public Celula(Object elemento) {
    this.elemento = elemento;
  }

  public void setProxima(Celula proxima) {
    this.proxima = proxima;
  }

  public Celula getProxima() {
    return proxima;
  }

  public void setAnterior(Celula anterior) {
    this.anterior = anterior;
  }

  public Celula getAnterior() {
    return anterior;
  }

  public Object getElemento() {
    return elemento;
  }
}
```

2. Implemente a classe `ListaLigada` com o "esqueleto" das operações. Utilize o pacote **br.com.caelum.ed.listasligadas**.

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

public class ListaLigada {

  private Celula primeira;
```

```
  private Celula ultima;

  private int totalDeElementos;

  public void adiciona(Object elemento) {
  }

  public void adiciona(int posicao, Object elemento) {
  }

  public Object pega(int posicao) {
    return null;
  }

  public void remove(int posicao) {
  }

  public int tamanho() {
    return 0;
  }

  public boolean contem(Object o) {
    return false;
  }

  public void adicionaNoComeco(Object elemento) {
  }

  public void removeDoComeco() {
  }

  public void removeDoFim() {
  }
}
```

3. Implemente **todos** os testes feitos na seção Teste. Coloque as classes no pacote **br.com.caelum.ed.listasligadas**.

4. Para testar as operações precisamos reescrever o método `toString()` para devolver os elementos da Lista em uma String.

```
public String toString() {

  // Verificando se a Lista está vazia
  if(this.totalDeElementos == 0){
    return "[]";
  }

  StringBuilder builder = new StringBuilder("[");
  Celula atual = primeira;

  // Percorrendo até o penúltimo elemento.
  for (int i = 0; i < this.totalDeElementos - 1; i++) {
    builder.append(atual.getElemento());
    builder.append(", ");
    atual = atual.getProxima();
```

```
    }

    // último elemento
    builder.append(atual.getElemento());
    builder.append("]");

    return builder.toString();
}
```

5. Implemente todos métodos para a Lista Ligada. A cada método implementado não esqueça de executar o teste apropriado.

- Adiciona no começo.

```
public void adicionaNoComeco(Object elemento) {
  if(this.totalDeElementos == 0){
    Celula nova = new Celula(elemento);
    this.primeira = nova;
    this.ultima = nova;
  } else {
    Celula nova = new Celula(this.primeira, elemento);
    this.primeira.setAnterior(nova);
    this.primeira = nova;
  }
  this.totalDeElementos++;
}
```

- Adiciona no fim.

```
public void adiciona(Object elemento) {
  if (this.totalDeElementos == 0) {
    this.adicionaNoComeco(elemento);
  } else {
    Celula nova = new Celula(elemento);
    this.ultima.setProxima(nova);
    nova.setAnterior(this.ultima);
    this.ultima = nova;
    this.totalDeElementos++;
  }
}
```

- Adiciona em qualquer posição. Esta operação depende do método `pegaCelula(int)` que por sua vez depende do método `posicaoOcupada(int)`.

```
private boolean posicaoOcupada(int posicao){
  return posicao >= 0 && posicao < this.totalDeElementos;
}

private Celula pegaCelula(int posicao) {
  if(!this.posicaoOcupada(posicao)){
    throw new IllegalArgumentException("Posição não existe");
  }

  Celula atual = primeira;
  for (int i = 0; i < posicao; i++) {
```

```
    atual = atual.getProxima();
  }
  return atual;
}

public void adiciona(int posicao, Object elemento) {
  if(posicao == 0){ // No começo.
    this.adicionaNoComeco(elemento);
  } else if(posicao == this.totalDeElementos){ // No fim.
    this.adiciona(elemento);
  } else {
    Celula anterior = this.pegaCelula(posicao - 1);
    Celula proxima = anterior.getProxima();
    Celula nova = new Celula(anterior.getProxima(), elemento);
    nova.setAnterior(anterior);
    anterior.setProxima(nova);
    proxima.setAnterior(nova);
    this.totalDeElementos++;
  }
}
```

- Remove do começo.

```
public void removeDoComeco() {
  if (!this.posicaoOcupada(0)) {
    throw new IllegalArgumentException("Posição não existe");
  }

  this.primeira = this.primeira.getProxima();
  this.totalDeElementos--;

  if (this.totalDeElementos == 0) {
    this.ultima = null;
  }
}
```

- Remove do fim.

```
public void removeDoFim() {
  if (!this.posicaoOcupada(this.totalDeElementos - 1)) {
    throw new IllegalArgumentException("Posição não existe");
  }
  if (this.totalDeElementos == 1) {
    this.removeDoComeco();
  } else {
    Celula penultima = this.ultima.getAnterior();
    penultima.setProxima(null);
    this.ultima = penultima;
    this.totalDeElementos--;
  }
}
```

- Remove de qualquer posição.

```
public void remove(int posicao) {
  if (!this.posicaoOcupada(posicao)) {
    throw new IllegalArgumentException("Posição não existe");
```

```
    }

    if (posicao == 0) {
      this.removeDoComeco();
    } else if (posicao == this.totalDeElementos - 1) {
      this.removeDoFim();
    } else {
      Celula anterior = this.pegaCelula(posicao - 1);
      Celula atual = anterior.getProxima();
      Celula proxima = atual.getProxima();

      anterior.setProxima(proxima);
      proxima.setAnterior(anterior);

      this.totalDeElementos--;
    }
  }
```

- Informar o tamanho.

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

- Pega elemento por posição.

```
  public Object pega(int posicao) {
    return this.pegaCelula(posicao).getElemento();
  }
```

- Verifica se um elemento pertence a Lista.

```
  public boolean contem(Object elemento) {
    Celula atual = this.primeira;

    while (atual != null) {
      if (atual.getElemento().equals(elemento)) {
        return true;
      }
      atual = atual.getProxima();
    }
    return false;
  }
```

6. Vamos comparar o consumo de tempo entre o Vetor e a Lista Ligada para saber em quais situações um é melhor que o outro.

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

    ArrayList<String> vetor = new ArrayList<String>();
    LinkedList<String> lista = new LinkedList<String>();
    int numeroDeElementos = 40000;

    // ADICIONADO NO COMEÇO
```

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

for (int i = 0; i < numeroDeElementos; i++) {
  vetor.add(0, "" + i);
}

long fim = System.currentTimeMillis();
System.out.println("Vetor adiciona no começo: " + (fim - inicio)
    / 1000.0);

inicio = System.currentTimeMillis();

for (int i = 0; i < numeroDeElementos; i++) {
  lista.add(0, "" + i);
}

fim = System.currentTimeMillis();
System.out.println("Lista Ligada adiciona no começo: " +
  (fim - inicio) / 1000.0);


// PERCORRENDO
inicio = System.currentTimeMillis();

for (int i = 0; i < numeroDeElementos; i++) {
  vetor.get(i);
}

fim = System.currentTimeMillis();
System.out
  .println("Vetor percorrendo: "
  + (fim - inicio) / 1000.0);

inicio = System.currentTimeMillis();

for (int i = 0; i < numeroDeElementos; i++) {
  lista.get(i);
}

fim = System.currentTimeMillis();
System.out.
  println("Lista Ligada percorrendo: "
  + (fim - inicio) / 1000.0);


// REMOVENDO DO COMEÇO
inicio = System.currentTimeMillis();

for (int i = 0; i < numeroDeElementos; i++) {
  vetor.remove(0);
}

fim = System.currentTimeMillis();
System.out
  .println("Vetor remove do começo: "
  + (fim - inicio) / 1000.0);
```

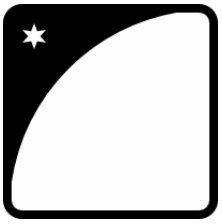
```
    inicio = System.currentTimeMillis();

    for (int i = 0; i < numeroDeElementos; i++) {
      lista.remove(0);
    }

    fim = System.currentTimeMillis();
    System.out.println("Lista Ligada remove do começo: "
      + (fim - inicio) / 1000.0);

  }
}
```

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:

Vetores

PRÓXIMO CAPÍTULO:

Pilhas

Blog Caelum

Cursos Online

Facebook

Newsletter

Casa do Código

Twitter