Listas dinâmicas

Conceitos, complexidade e implementação

Prof. Marcelo de Souza

45RPE – Resolução de Problemas com Estruturas de Dados

Universidade do Estado de Santa Catarina

Material de apoio

Leitura principal:

- Capítulo 7 de Goodrich et al. (2014)[1] – Listas e iteradores.

Leitura complementar:

- Capítulo 5 de Lafore e Machado (2004)[2] – Listas encadeadas.
- Capítulo 7 de Pereira (2008)[3] – Listas encadeadas.

[1] Michael T Goodrich et al. (2014). *Data structures and algorithms in Java*. 6ª ed. John Wiley & Sons.

[2] Robert Lafore e Eveline Vieira Machado (2004). *Estruturas de dados & Algoritmos em Java*. Ciência Moderna.

[3] Silvio do Lago Pereira (2008). *Estruturas de Dados Fundamentais: Conceitos e Aplicações*.

Listas dinâmicas

Ideia geral

Uma **lista dinâmica** permite operações (inserção e remoção) em posições arbitrárias.

- i.e., não somente nas extremidades, mas também no meio da estrutura.

Listas dinâmicas

Ideia geral

Uma **lista dinâmica** permite operações (inserção e remoção) em posições arbitrárias.

- i.e., não somente nas extremidades, mas também no meio da estrutura.

Operações:

- `get(i)`: retorna o `i`-ésimo elemento da lista.
- `set(i, e)`: substitui o `i`-ésimo elemento por `e`.
- `add(i, e)`: adiciona `e` na posição `i`, alterando a posição dos elementos seguintes.
- `remove(i)`: remove o `i`-ésimo elemento da lista.

Listas dinâmicas

Ideia geral

Uma **lista dinâmica** permite operações (inserção e remoção) em posições arbitrárias.

- i.e., não somente nas extremidades, mas também no meio da estrutura.

Operações:

- `get(i)`: retorna o `i`-ésimo elemento da lista.
- `set(i, e)`: substitui o `i`-ésimo elemento por `e`.
- `add(i, e)`: adiciona `e` na posição `i`, alterando a posição dos elementos seguintes.
- `remove(i)`: remove o `i`-ésimo elemento da lista.

Aplicações:

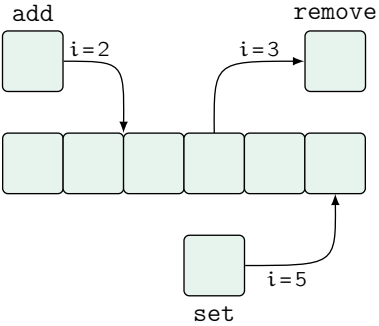
- Listas com ordenação (e.g., inserção ordenada).
- Listagem de arquivos em um diretório.

Listas dinâmicas

Exemplo de funcionamento

Seja uma lista dinâmica implementada usando uma lista sequencial simples, inicialmente vazia.

Método	**Retorno**	**Conteúdo da lista**
`add(0, A)`	–	(A)
`add(0, B)`	–	(B, A)
`get(1)`	A	(B, A)
`set(2, C)`	*erro*	(B, A)
`add(2, C)`	–	(B, A, C)
`add(4, D)`	*erro*	(B, A, C)
`remove(1)`	A	(B, C)
`add(1, D)`	–	(B, D, C)
`add(1, E)`	–	(B, E, D, C)
`get(4)`	*erro*	(B, E, D, C)
`add(4, F)`	–	(B, E, D, C, F)
`set(2, G)`	D	(B, E, G, C, F)
`get(2)`	G	(B, E, G, C, F)



(representação gráfica)

Listas dinâmicas

Implementação usando arranjos

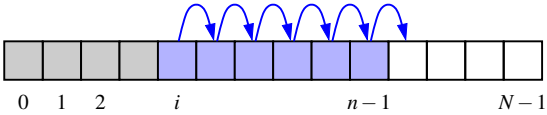
Problema 1: realocação de elementos nas operações.

Listas dinâmicas

Implementação usando arranjos

Problema 1: realocação de elementos nas operações.

- Na **inserção**, os elementos seguintes são movidos pra **trás**.

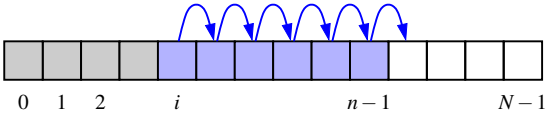


Listas dinâmicas

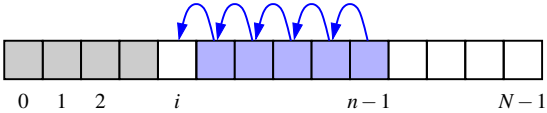
Implementação usando arranjos

Problema 1: realocação de elementos nas operações.

- Na **inserção**, os elementos seguintes são movidos pra **trás**.



- Na **remoção**, os elementos seguintes são movidos pra **frente**.



Listas dinâmicas

Implementação usando arranjos

Problema 2: capacidade limitada devido à estrutura fixa do arranjo.

Listas dinâmicas

Implementação usando arranjos

Problema 2: capacidade limitada devido à estrutura fixa do arranjo.

Solução:

- Mantém uma lista de **tamanho fixo**.
- Quando precisar de mais espaço, **dobra o tamanho** da lista.
  - Cria um novo arranjo com o dobro de capacidade.
  - Copia os elementos para o novo arranjo.
  - Atualiza a referência.

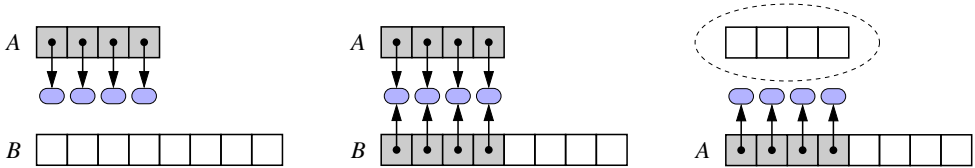
Listas dinâmicas

Implementação usando arranjos

Problema 2: capacidade limitada devido à estrutura fixa do arranjo.

Solução:

- Mantém uma lista de **tamanho fixo**.
- Quando precisar de mais espaço, **dobra o tamanho** da lista.
  - Cria um novo arranjo com o dobro de capacidade.
  - Copia os elementos para o novo arranjo.
  - Atualiza a referência.



Listas dinâmicas

Análise de complexidade (arranjos)

Listas dinâmicas

Análise de complexidade (arranjos)

Lista usando arranjo (I)

```
get(i):
  retorna data[i]
```

$\mathcal{O}(1)$

Listas dinâmicas

Análise de complexidade (arranjos)

Lista usando arranjo (I)

```
get(i):
  retorna data[i]
```

$\mathcal{O}(1)$

Lista usando arranjo (II)

```
set(i, e):
  data[i] ← e
```

$\mathcal{O}(1)$

Listas dinâmicas

Análise de complexidade (arranjos)

Lista usando arranjo (I)

```
get(i):                                  O(1)
  retorna data[i]

add(i, e):                               O(n)
  SE size == data.length:
    resize(2 * data.length)
  PARA k ← size - 1 ATÉ i:
    data[k + 1] ← data[k]
  data[i] ← e
```

Lista usando arranjo (II)

```
set(i, e):                               O(1)
  data[i] ← e
```

Listas dinâmicas

Análise de complexidade (arranjos)

Lista usando arranjo (I)

```
get(i):                                  O(1)
  retorna data[i]

add(i, e):                               O(n)
  SE size == data.length:
    resize(2 * data.length)
  PARA k ← size - 1 ATÉ i:
    data[k + 1] ← data[k]
  data[i] ← e
```

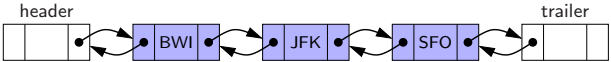
Lista usando arranjo (II)

```
set(i, e):                               O(1)
  data[i] ← e

remove(i):                               O(n)
  temp ← data[i]
  PARA k ← i ATÉ size - 2:
    data[k] ← data[k + 1]
  data[size - 1] ← null
  retorna temp
```

Listas dinâmicas

Implementação usando encadeamento

Usaremos uma **lista duplamente encadeada** para implementar a lista dinâmica.



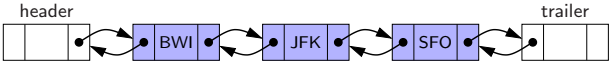
Listas dinâmicas

Implementação usando encadeamento

Usaremos uma **lista duplamente encadeada** para implementar a lista dinâmica.

Precisamos de um método **auxiliar** para suportar operações em posições arbitrárias:

- `searchNode`: busca o nodo de uma dada posição. ← operação $\mathcal{O}(n)$ sempre necessária!



Listas dinâmicas

Implementação usando encadeamento

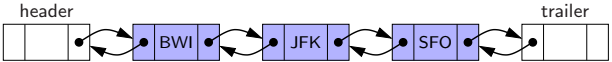
Usaremos uma **lista duplamente encadeada** para implementar a lista dinâmica.

Precisamos de um método **auxiliar** para suportar operações em posições arbitrárias:

- `searchNode`: busca o nodo de uma dada posição. ← operação $\mathcal{O}(n)$ sempre necessária!

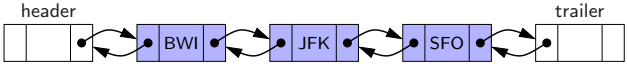
E também os métodos que implementam as **operações de listas**. Dada uma posição:

- `get`: retorna o elemento.
- `set`: substitui o elemento.
- `add`: adiciona um elemento.
- `remove`: remove o elemento.



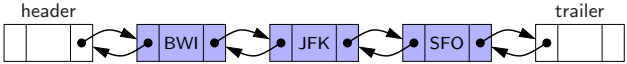
Listas dinâmicas

Análise de complexidade (encadeamento)



Listas dinâmicas

Análise de complexidade (encadeamento)



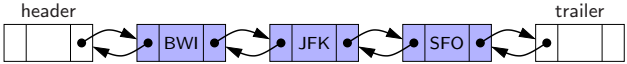
Lista usando encadeamento (I)

```
get(i):
  node ← searchNode(i)
  retorna node.getElement()
```

$\mathcal{O}(n)$

Listas dinâmicas

Análise de complexidade (encadeamento)



Lista usando encadeamento (I)

$\mathcal{O}(n)$

```
get(i):
  node ← searchNode(i)
  retorna node.getElement()
```

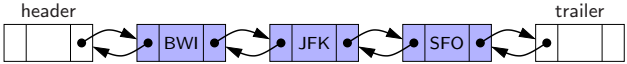
Lista usando encadeamento (II)

$\mathcal{O}(n)$

```
set(i, e):
  node ← searchNode(i)
  node.setElement(e)
```

Listas dinâmicas

Análise de complexidade (encadeamento)



Lista usando encadeamento (I)

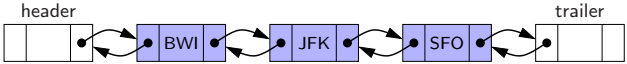
```
get(i):                                  O(n)
  node ← searchNode(i)
  retorna node.getElement()

add(i, e):                               O(n)
  node ← searchNode(i)
  addBetween(e, node.prev, node)
```

Lista usando encadeamento (II)

```
set(i, e):                               O(n)
  node ← searchNode(i)
  node.setElement(e)
```

Listas dinâmicas

Análise de complexidade (encadeamento)



Lista usando encadeamento (I)

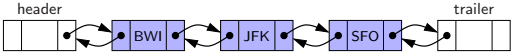
```
get(i):                                  O(n)
  node ← searchNode(i)
  retorna node.getElement()

add(i, e):                               O(n)
  node ← searchNode(i)
  addBetween(e, node.prev, node)
```

Lista usando encadeamento (II)

```
set(i, e):                               O(n)
  node ← searchNode(i)
  node.setElement(e)

remove(i):                               O(n)
  node ← searchNode(i)
  retorna remove(node)
```

Listas dinâmicas

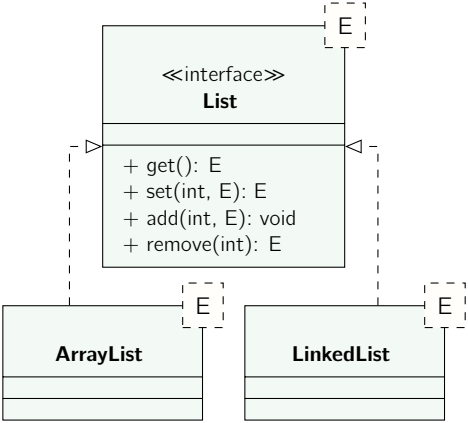
Análise de complexidade (resumo)



Operação	`ArrayList`	`LinkedList`
`get(i)`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`set(i, e)`	$\mathcal{O}(1)$	$\mathcal{O}(n)$
`add(i, e)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$
`remove(i)`	$\mathcal{O}(n)$	$\mathcal{O}(n)$

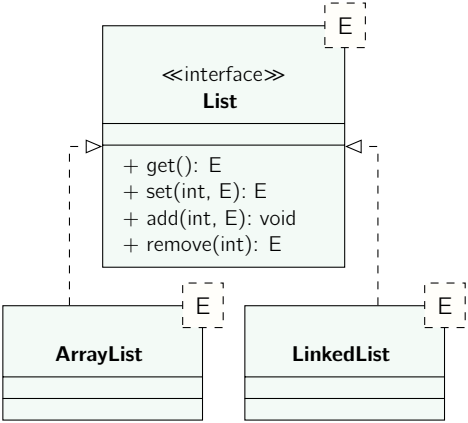
Listas dinâmicas

Implementação



Listas dinâmicas

Implementação



Implementação e uso:

```
data-structures
└─ list
    ├─ {List, ArrayList, LinkedList}.java
    └─ TestList.java
```

Veja também:

- `java.util.List`[1] (interface)
  - E suas diferentes implementações.
- Apêndice II

[1] Veja em https://docs.oracle.com/en/java/javase/24/docs/api/java.base/java/util/List.html.

Apêndices

Apêndice I

Implementação de listas (interface)

Interface `List`:

```
1 public interface List<E> {
2   int size();
3   boolean isEmpty();
4   E get(int i);
5   E set(int i, E e);
6   void add(int i, E e);
7   E remove(int i);
8 }
```

Usamos uma interface para definir nosso TAD lista. A interface só define a API (métodos), a qual pode ser implementada de diversas formas (e usando diferentes estruturas).

Apêndice I

Implementação de listas (arranjos)

```java
public class ArrayList<E> implements List<E> {
  public static final int CAPACITY = 16;
  private E[] data;
  private int size = 0;

  public ArrayList() { this(CAPACITY); }

  public ArrayList(int capacity) {
    data = (E[]) new Object[capacity];
  }

  public int size() { return size; }
  public boolean isEmpty() { return size == 0; }

  //...
}
```

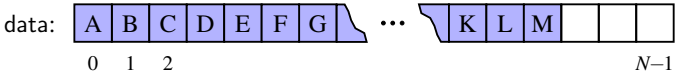
Apêndice I

Implementação de listas (arranjos)

```java
public class ArrayList<E> implements List<E> {
  public static final int CAPACITY = 16;
  private E[] data;
  private int size = 0;

  public ArrayList() { this(CAPACITY); }

  public ArrayList(int capacity) {
    data = (E[]) new Object[capacity];
  }

  public int size() { return size; }
  public boolean isEmpty() { return size == 0; }

  //...
}
```

Detalhes:

- `ArrayList` implementa a interface `List`.
- O *array* possui estrutura fixa (capacidade definida), apesar de usarmos a estratégia apresentada para contornar essa limitação.
- Não há referência para a "extremidade" da estrutura, pois os dados sempre estarão armazenados a partir do início do *array*. Logo, o último elemento estará em `size - 1`.



Apêndice I

Implementação de listas (arranjos)

Método `checkIndex`:

```
1 protected void checkIndex(int i, int n) {
2   if (i < 0 || i >= n)
3     throw new IndexOutOfBoundsException("Illegal index: " + i);
4 }
```

- Para se certificar de não acessar uma posição inválida.

Apêndice I

Implementação de listas (arranjos)

Método `checkIndex`:

```
protected void checkIndex(int i, int n) {
  if (i < 0 || i >= n)
    throw new IndexOutOfBoundsException("Illegal index: " + i);
}
```

- Para se certificar de não acessar uma posição inválida.

Método `resize`:

```
protected void resize(int capacity) {
  E[] temp = (E[]) new Object[capacity];
  for (int k = 0; k < size; k++)
    temp[k] = data[k];
  data = temp;
}
```

- Para implementar tamanho dinâmico.

Apêndice I

Implementação de listas (arranjos)

Método `get`:

```
1 public E get(int i) {
2   checkIndex(i, size);
3   return data[i];
4 }
```

Apêndice I

Implementação de listas (arranjos)

Método get:

```
public E get(int i) {
  checkIndex(i, size);
  return data[i];
}
```

Método set:

```
public E set(int i, E e) {
  checkIndex(i, size);
  E temp = data[i];
  data[i] = e;
  return temp;
}
```

Apêndice I

Implementação de listas (arranjos)

Método get:

```
public E get(int i) {
  checkIndex(i, size);
  return data[i];
}
```

Método set:

```
public E set(int i, E e) {
  checkIndex(i, size);
  E temp = data[i];
  data[i] = e;
  return temp;
}
```

- Tal como feito no método get, a posição a ser acessada é verificada.
- O elemento substituído é retornado.

Apêndice I

Implementação de listas (arranjos)

Métodos add:

```
public void add(int i, E e) {
  checkIndex(i, size + 1);
  if (size == data.length)
    resize(2 * data.length);
  for (int k = size - 1; k >= i; k--)
    data[k + 1] = data[k];
  data[i] = e;
  size++;
}

public void add(E e) {
  add(size, e);
}
```

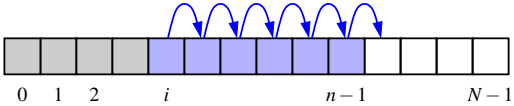
Apêndice I

Implementação de listas (arranjos)

Métodos add:

```
1  public void add(int i, E e) {
2    checkIndex(i, size + 1);
3    if (size == data.length)
4      resize(2 * data.length);
5    for (int k = size - 1; k >= i; k--)
6      data[k + 1] = data[k];
7    data[i] = e;
8    size++;
9  }
10
11 public void add(E e) {
12   add(size, e);
13 }
```

- A posição de inserção deve estar entre 0 e `size`.
- Se o limite do *array* foi atingido, sua capacidade é dobrada (linhas 3 e 4).
- As linhas 5 e 6 fazem a realocação dos elementos da posição `i` em diante.
- O método `add` sem posição definida insere o elemento no final da lista (i.e., no índice `size`).



Apêndice I

Implementação de listas (arranjos)

Método `remove`:

```
public E remove(int i) {
  checkIndex(i, size);
  E temp = data[i];
  for (int k = i; k < size-1; k++)
    data[k] = data[k+1];
  data[size-1] = null;
  size--;
  return temp;
}
```

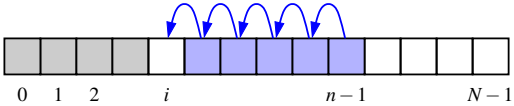
Apêndice I

Implementação de listas (arranjos)

Método `remove`:

```
1 public E remove(int i) {
2   checkIndex(i, size);
3   E temp = data[i];
4   for (int k = i; k < size-1; k++)
5     data[k] = data[k+1];
6   data[size-1] = null;
7   size--;
8   return temp;
9 }
```

- A posição de remoção deve estar entre 0 e `size - 1`.
- As linhas 4 e 5 fazem a realocação dos elementos para cobrir o espaço deixado pela remoção.
- A linha 6 limpa a antiga última posição.
- O elemento removido é retornado.



Apêndice II

Implementação de listas (encadeamento)

Nossa lista duplamente encadeada já fornece os métodos `addBetween` e `remove`:

```java
public class DoublyLinkedList<E> {
  // ...
  private void addBetween(E e, Node<E> pred, Node<E> succ) {
    Node<E> newest = new Node<>(e, pred, succ);
    pred.setNext(newest);
    succ.setPrev(newest);
    size++;
  }

  private E remove(Node<E> node) {
    Node<E> predecessor = node.getPrev();
    Node<E> successor = node.getNext();
    predecessor.setNext(successor);
    successor.setPrev(predecessor);
    size--;
    return node.getElement();
  }
  // ...
}
```

Apêndice II

Implementação de listas (encadeamento)

Método checkIndex:

```
protected void checkIndex(int i, int n) {
  if (i < 0 || i >= n) throw new IndexOutOfBoundsException("Illegal index: " + i);
}
```

Métodos searchNode:

```
protected Node<E> searchNode(int position) {
  if(position == 0) return header.getNext();
  if(position == size()) return trailer;
  int count = -1;
  Node<E> walk = header.getNext();
  while(walk != trailer) {
    count++;
    if(count == position)
      return walk;
    walk = walk.getNext();
  }
  return null;
}
```

Apêndice II

Implementação de listas (encadeamento)

Método `checkIndex`:

```
protected void checkIndex(int i, int n) {
  if (i < 0 || i >= n) throw new IndexOutOfBoundsException("Illegal index: " + i);
}
```

Métodos `searchNode`:

```
1  protected Node<E> searchNode(int position) {
2    if(position == 0) return header.getNext();
3    if(position == size()) return trailer;
4    int count = -1;
5    Node<E> walk = header.getNext();
6    while(walk != trailer) {
7      count++;
8      if(count == position)
9        return walk;
10     walk = walk.getNext();
11   }
12   return null;
13 }
```

- Caso a posição seja 0, trata-se do primeiro nodo; caso seja o tamanho da lista, trata-se do `trailer` (linhas 2 e 3).
- O método percorre a estrutura, incrementando um contador para identificar as posições (laço das linhas 6 a 11).
- Quando o contador é igual à posição buscada, o nodo correspondente é retornado (linhas 8 e 9).

Apêndice II

Implementação de listas (encadeamento)

Método `get`:

```
public E get(int position) {
  checkIndex(position, size);
  return(searchNode(position).getElement());
}
```

Método `remove`:

```
public E remove(int position) {
  checkIndex(position, size);
  return remove(searchNode(position));
}
```

Apêndice II

Implementação de listas (encadeamento)

Método `get`:

```
1 public E get(int position) {
2   checkIndex(position, size);
3   return(searchNode(position).getElement());
4 }
```

Método `remove`:

```
1 public E remove(int position) {
2   checkIndex(position, size);
3   return remove(searchNode(position));
4 }
```

Para esses métodos, basta:

1. Verificar se a posição desejada é válida;
2. Buscar pelo nodo da posição;
3. Fazer a operação desejada: retornar ou remover o elemento correspondente.

Apêndice II

Implementação de listas (encadeamento)

Método `set`:

```
1 public E set(int position, E e) {
2   checkIndex(position, size);
3   Node<E> n = searchNode(position);
4   addBetween(e, n, n.getNext());
5   remove(n);
6   return n.getElement();
7 }
```

Apêndice II

Implementação de listas (encadeamento)

Método `set`:

```
1 public E set(int position, E e) {
2   checkIndex(position, size);
3   Node<E> n = searchNode(position);
4   addBetween(e, n, n.getNext());
5   remove(n);
6   return n.getElement();
7 }
```

Método `add`:

```
1 public void add(int position, E e) {
2   checkIndex(position, size + 1);
3   Node<E> n = searchNode(position);
4   addBetween(e, n.getPrev(), n);
5 }
```

- A atualização é feita inserindo um novo nodo (com o novo elemento) após o nodo da posição buscada ("antigo" nodo).
- O antigo nodo é removido da estrutura e seu (antigo) elemento é retornado.

Apêndice II

Implementação de listas (encadeamento)

Método `set`:

```
1 public E set(int position, E e) {
2   checkIndex(position, size);
3   Node<E> n = searchNode(position);
4   addBetween(e, n, n.getNext());
5   remove(n);
6   return n.getElement();
7 }
```

- A atualização é feita inserindo um novo nodo (com o novo elemento) após o nodo da posição buscada ("antigo" nodo).
- O antigo nodo é removido da estrutura e seu (antigo) elemento é retornado.

Método `add`:

```
1 public void add(int position, E e) {
2   checkIndex(position, size + 1);
3   Node<E> n = searchNode(position);
4   addBetween(e, n.getPrev(), n);
5 }
```

- O novo nodo ocupará a posição do nodo buscado, que será "deslocado" para trás.
- Para isso, o novo nodo é inserido antes do nodo buscado.

Apêndice II

Implementação de listas (encadeamento)

Basta usarmos os métodos da classe `DoublyLinkedList` para implementar a lista dinâmica.

```
public class LinkedList<E> implements List<E> {
  DoublyLinkedList<E> list = new DoublyLinkedList<>();
  public int size() { return list.size(); }
  public boolean isEmpty() { return list.isEmpty(); }
  public String toString() { return list.toString(); }
  public E get(int i) { return list.get(i); }
  public E set(int i, E e) { return list.set(i, e); }
  public void add(int i, E e) { list.add(i, e); }
  public E remove(int i) { return list.remove(i); }
}
```

45RPE – Resolução de Problemas com Estruturas de Dados
Prof. Marcelo de Souza