PCC104 - Projeto e Análise de Algoritmos

Marco Antonio M. Carvalho

Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto

Conteúdo

Fonte

Este material é baseado nos livros

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
- S. Halim. *Competitive Programming*. 3rd Edition, 2013.
- Ian Parberry and William Gasarch. *Problems on Algorithms*. Second Edition, 2002.
- Ian Parberry *Lecture Notes on Algorithm Analysis and Complexity Theory*. Fourth Edition, 2001.

Licença

Genéricos em Java

Genéricos

O sistema de genéricos em Java é baseado no sistema utilizado pela linguagem C++.

Métodos Genéricos especificam em uma única declaração um conjunto de métodos de relacionados.

Classes Genéricas especificam em uma única declaração um conjunto de tipos relacionados.

Genéricos

Quando o compilador traduz o código para bytecode, os métodos genéricos têm seus argumentos substituídos por tipos de verdade – por padrão, o tipo Object é utilizado.

O tratamento dispensado pelo compilador às classes genéricas é semelhante ao dispensado aos métodos genéricos.

Entretanto, diferentemente do que ocorre em C++, não é criada uma cópia dos métodos e classes para cada tipo utilizado.

Classes Genéricas

Vejamos como exemplo uma classe que testa uma a classe pilha genérica hipotética.

Exceto pela sintaxe específica da linguagem Java, o uso de genéricos é semelhante ao da linguagem C++.

Para instanciarmos um objeto de uma classe genérica, precisamos informar qual tipo deve ser associado à classe.

No exemplo a seguir, declaramos dois objetos, associando às classes adaptadoras Double e Int.

StackTest.java

```
public class StackTest {
    private Stack< Double > doubleStack;
    private Stack< Integer > integerStack;

    public void testStacks(){
        doubleStack = new Stack< Double >(5);
        integerStack = new Stack< Integer >(10);

        doubleStack.push(1.1);
        doubleStack.push(2.1);
        doubleStack.push(3.1);

        doubleStack.pop();

        intStack.push(1);
        intStack.push(2);
        intStack.push(3);

        intStack.pop();
    }
```

Genéricos em Java vs. Genéricos em C++

- Em C++, *Standard Template Library*; em Java, *Java Collections Framework*.
- Library $\neq$ Framework.
- Genérico C++ = Interface Java.
- Contêiner C++ = Coleção Java.
- Em C++, performance é uma das exigências das interfaces, em Java não.
- Em Java, os algoritmos são organizados por coleção, em C++ não.
- Em C++, o gerenciamento de memória é por conta do usuário, em Java não.

Coleções

O **Java Collections Framework** contém estruturas de dados "pré-empacotadas", interfaces e algoritmos para manipular tais estruturas, estabelecendo uma relação muito próxima com genéricos.

Podemos utilizar as estruturas sem nos preocuparmos com detalhes da implementação interna.

Os algoritmos possuem excelente desempenho assintótico com bons tempos de execução e minimização do uso de memória.

Também fornece iteradores, para percorrermos os elementos de uma coleção, assim como a STL.

Algoritmos

O Java Collections Framework fornece vários algoritmos de alta performance para manipular elementos de uma coleção, alguns operam em listas apenas, outros em coleções em geral.

Todos os algoritmos são polimórficos, ou seja, podem ser aplicados a objetos de classes que implementam interfaces específicas, independente de detalhes internos.

Iteradores

O Java Collections Framework fornece apenas dois tipos de iteradores: Iterator e ListIterator.

Iteradores não podem ser comparados em Java, e não há iteradores de acesso aleatório.

A interface Iterator possui dois métodos principais: hasNext(), que indica se ainda existe um elemento a ser percorrido; e next(), que retorna o próximo objeto.

É necessário importar o pacote java.util.iterator.

Exemplo

```
Set<String> conjunto = new HashSet<>();
conjunto.add("item␣1");
conjunto.add("item␣2");
conjunto.add("item␣3");

// retorna o iterator
Iterator<String> i = conjunto.iterator();
while (i.hasNext()) {
  // recebe a palavra
  String palavra = i.next();
  System.out.println(palavra);
}
```

Coleções

Uma coleção é um objeto que mantém referências a outros objetos, normalmente, de um mesmo tipo.

As interfaces do Java Collections Framework declaram operações que podem ser realizadas genericamente em vários tipos de coleções.

Várias implementações destas interfaces são fornecidas pelo framework, no pacote java.util.

Podemos também criar nossas próprias implementações.

Interfaces

Collection	A raiz da hierarquia de coleções, a partir da qual todas as outras são derivadas.
Set	Uma coleção que não contém repetições.
List	Uma coleção ordenada que pode conter repetições.
Map	Associa chaves a valores e não pode conter chaves duplicadas.
Queue	Coleção FIFO que modela uma fila, embora outras políticas possam ser especificadas.

Interface Collection e Classe Collections

A interface *Collection* é a base da hierarquia de todas interfaces de coleções e contém operações realizadas em coleções inteiras (*bulk* operations):

- Adicionar elementos;
- Esvaziar;
- Comparar;
- Reter elementos.

Também contém operações que retornam iteradores (objetos Iterator), que nos permitem percorrer uma coleção.

Interface Collection e Classe Collections

A classe *Collections* fornece métodos que manipulam coleções polimorficamente, ou seja, implementa algoritmos para pesquisa e ordenação independente do tipo dos dados, entre outros.

Também fornece métodos adaptadores, que permitem que uma coleção seja tratada como sincronizada ou imutável.

Listas

A interface List é implementada por diversas classes:

- ArrayList: Lista implementada sobre um arranjo.
- LinkedList: Lista implementada sobre encadeamento.
- Vector: Lista implementada sobre um arranjo, porém, sincronizado.

Arraylists se comportam como os Vectors, no entanto, não são sincronizados e são mais rápidos.

ArrayList

ArrayLists são listas implementadas sobre arranjos dinâmicos, o que pode implicar em realocação de todos os elementos na memória.

O exemplo a seguir ilustra a utilização de ArrayLists.

Exemplo

```
import java.util.ArrayList;
import java.util.List;

public class ArrayListExample {
    public static void main(String args[]) {
        List<String> letters = new ArrayList<String>();

        //add example
        letters.add("A");
        letters.add("C");

        //inserts B between A and C
        letters.add(1,"B");

        //contains example
        System.out.println("Contains␣E␣?␣"+letters.contains("E"));
        System.out.println("Contains␣Z␣?␣"+letters.contains("Z"));

        //get example
        String e = letters.get(4);
```

Exemplo

```
        //tempList is empty?
        System.out.println("Is empty ? "+letters.isEmpty());

        //indexOf example
        System.out.println("First index of D = "+letters.indexOf("D")
            );
        System.out.println("Last index of D = "+letters.lastIndexOf("
            D"));

        //remove example
        String removed = letters.remove(3);

        //remove first occurrence of H
        boolean isRemoved = letters.remove("H");

        //size example
        System.out.println("Size = "+letters.size());

        //set example
        letters.set(2, "D");
    }
}
```

LinkedList

LinkedLists são listas duplamente encadeadas, com melhor complexidade para inserções e remoções.

A classe LinkedList implementa as interfaces List e Deque, podendo ser utilizada para criar listas, pilhas, filas e deques.

O exemplo a seguir ilustra a utilização de LinkedLists.

Exemplo

```
import java.util.LinkedList;
public class LinkedListDemo {
    public static void main(String[] args) {
        //Creation
        LinkedList list = new LinkedList();

        // Insertion by add method
        list.add(25);          // Inserting number  index: [0]
        list.add("java");      // Inserting String  index: [1]
        list.add(true);        // Inserting boolean index: [2]

        // Insert element at first and last position
        list.addFirst("ramu");
        list.addLast("kalu");

        //Printing Whole list in single line
        System.out.println("List␣contents␣"+list);
```

Exemplo

```
        //Getting the first & last element from the list
        System.out.println("Element at first position is "+list.
            getFirst());
        System.out.println("Element at last position is "+list.
            getLast());

        //Removing the element at first and last position
        list.removeFirst();
        list.removeLast();
    }
}
```

Vector

Assim como ArrayLists, os Vectors fornecem uma estrutura parecida com um vetor, que pode se redimensionar automaticamente.

Embora os comportamentos sejam similares, os Vectors são sincronizados, o que permitem operações que se valem do paralelismo de processamento.

Exemplo

```
import java.util.Vector;
public class VectorDemo {
      public static void main(String[] args) {

          //creation
          Vector v = new Vector();

          // Insertion by addElement method
          v.addElement(25);          // Inserting number  index: [0]
          v.addElement("java");      // Inserting String  index: [1]
          v.addElement(true);        // Inserting boolean index: [2]

          //Printing all the elements using Iterator
          Iterator it = v.iterator();
          while(it.hasNext()){
              System.out.println(it.next());
          }

          //Fetching element at particular index
          System.out.println("Element at index 2 is "+v.elementAt(2))
              ;
      }
}
```

Pilhas

A classe Stack estende a classe Vector para implementar a estrutura de dados pilha.

Não há alternativa quanto à estrutura subjacente à pilha, entretanto, podemos usar deques para simular pilhas, se necessário.

O exemplo a seguir demonstra alguns métodos da classe Stack.

Exemplo

```
import java.util.Stack;

public class StackDemo {
    public static void main(String[] args) {
        //creation
        Stack st = new Stack();

        // Insertion by push method
        st.push(25);
        st.push("java");
        st.push(true);

        //peek
        System.out.println(st.peek());

        //pop
        while(!st.empty()){
            System.out.println("Element: "+st.pop());
        }
    }
}
```

Filas

Como mencionado anteriormente, a classe LinkedList implementa a interface Queue, podendo ser também utilizada para criar filas.

As operações mais comuns são:

- add: enfileira um elemento;
- poll: desenfileira o elemento da frente da fila;
- peek: retorna uma referência ao objeto na frente da fila;
- clear: remove todos os elementos;
- size: retorna o número de elementos.

Exemplo

```
import java.util.LinkedList;

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

        //Creation
        Queue<Integer> integerQueue = new LinkedList<>(7);

        for(int i=0;i<7;i++){
            integerQueue.add(i);
        }

        for(int i=0;i<7;i++){
            Integer in = integerQueue.poll();
            System.out.println("Processing␣Integer:"+in);
        }
    }
}
```

Deque

Como mencionado anteriormente, a classe LinkedList também implementa a interface Deque, podendo ser também utilizada para criar deques (*double-ended queues*).

As operações mais comuns são:

- **getFirst:** retorna uma referência ao objeto no início do deque;
- **getLast:** retorna uma referência ao objeto no final do deque;
- **removeFirst:** remove o objeto no início do deque;
- **removeLast:** remove o objeto no final do deque;
- **addFirst:** insere um objeto no início do deque;
- **addLast:** insere um objeto no final do deque;
- **clear:** remove todos os elementos;
- **size:** retorna o número de elementos.

Exemplo

```
import java.util.Deque;
import java.util.Iterator;
import java.util.LinkedList;

public class DequeExample {
    public static void main(String[] args) {
        Deque<String> deque = new LinkedList<String>();

        // We can add elements to the queue in various ways
        deque.addFirst("Element 1 (Head)");
        deque.addLast("Element 2 (Tail)");

        System.out.println(deque + "\n");

        // Iterate through the queue elements.
        System.out.println("Standard Iterator");
        Iterator iterator = deque.iterator();
        while (iterator.hasNext())
            System.out.println("\t" + iterator.next());
```

Exemplo

```
        // Peek returns the head, without deleting it
        System.out.println("Peek " + deque.peek());

        // We can check if a specific element exists in the deque
        System.out.println("Contains element 3: " + deque.contains(
            "Element 3 (Tail)"));

        // We can remove the first / last element.
        deque.removeFirst();
        deque.removeLast();
    }
}
```

Filas de Prioridade

A classe PriorityQueue implementa a interface Queue e ordena os elementos de acordo com o método compareTo (Comparable) ou um objeto Comparator;

As inserções são ordenadas e as remoções são realizadas no início da estrutura, sendo que o primeiro elemento é o de maior prioridade.

As operações mais comuns são:

- offer: insere um elemento na posição apropriada de acordo com sua prioridade;
- poll: remove o elemento de maior prioridade;
- peek: retorna uma referência ao objeto de maior prioridade, sem removê-lo;
- clear: remove todos os elementos;
- size: retorna o número de elementos.

Exemplo

```
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;
import java.util.Random;

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

        //natural ordering example of priority queue
        Queue<Integer> integerPriorityQueue = new PriorityQueue<>(7);
        Random rand = new Random();

        for(int i=0;i<7;i++){
            integerPriorityQueue.add(new Integer(rand.nextInt(100)));
        }

        for(int i=0;i<7;i++){
            Integer in = integerPriorityQueue.poll();
            System.out.println("Processing Integer:"+in);
        }
    }
}
```

Conjuntos

O Java Collections Framework possui diversas implementações da interface Set, incluindo

- HashSet : armazena os elementos em tabelas hash, desordenados.
- TreeSet : armazena os elementos em árvores balanceadas, ordenados.
- LinkedHashSet : armazena os elementos em tabelas hash com encadeamento, em ordem de inserção.

Exemplo

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class HashSetExample {
    public static void main(String[] args) {
        Set<String> fruits = new HashSet<>();

        //add example
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Pear");
        fruits.add("Grape");
        fruits.add("Orange");

        //isEmpty example
        System.out.println("is␣empty?␣=␣"+fruits.isEmpty());
```

Exemplo

```
        //contains example
        System.out.println("fruits␣contains␣Apple␣=␣"+fruits.contains
            ("Apple"));

        //remove example
        System.out.println("Apple␣removed␣from␣fruits␣set␣=␣"+fruits.
            remove("Apple"));

        //size example
        System.out.println("fruits␣set␣size␣=␣"+fruits.size());

        //iterator example
        Iterator<String> iterator = fruits.iterator();
        while(iterator.hasNext()){
            System.out.println("Consuming␣fruit␣"+iterator.next());
        }

        //clear example
        fruits.clear();
        System.out.println("is␣empty?␣=␣"+fruits.isEmpty());
    }
}
```

Mapas

Quatro das várias classes que implementam a interface Map são:

- **HashMap:** armazenam os elementos em tabelas hash, desordenados.
- **TreeMap:** armazenam os elementos em árvores balanceadas, ordenados.
- **LinkedHashMap:** armazenam os elementos em tabelas hash, com encadeamento duplo.
- **MultiMap:** permite uma coleção de valores para uma mesma chave.

Exemplo

```
import java.util.Collection;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

public class MapExample {
    public static void main(String[] args) {
        Map<String, String> data = new HashMap<>();

        data.put("A", "A"); // put example
        data.put("C", "C");
        data.put("D", null); // null value
        data.put(null, "Z"); // null key

        String value = data.get("C"); // get example
        System.out.println("Key = C, Value = " + value);

        boolean keyExists = data.containsKey(null);
        boolean valueExists = data.containsValue("Z");

        System.out.println("data map size=" + data.size());
```

Exemplo

```
        Set<String> keySet = data.keySet();
        System.out.println("data map keys = " + keySet);

        Collection<String> values = data.values();
        System.out.println("data map values = " + values);

        data.clear();
        System.out.println("data map is empty =" + data.isEmpty());
        }
}
```

Dúvidas?