Módulo 2

Introdução à Programação II

Lição 3

Técnicas Avançadas de Programação

Versão 1.0 - Mar/2007

Autor
Rebecca Ong

Equipe
Joyce Avestro
Florence Balagtas
Rommel Feria
Rebecca Ong
John Paul Petines
Sun Microsystems
Sun Philippines

Necessidades para os Exercícios

Sistemas Operacionais Suportados

NetBeans IDE 5.5 para os seguintes sistemas operacionais:

- Microsoft Windows XP Profissional SP2 ou superior
- Mac OS X 10.4.5 ou superior
- Red Hat Fedora Core 3
- Solaris™ 10 Operating System (SPARC® e x86/x64 Platform Edition)

NetBeans Enterprise Pack, poderá ser executado nas seguintes plataformas:

- Microsoft Windows 2000 Profissional SP4
- Solaris™ 8 OS (SPARC e x86/x64 Platform Edition) e Solaris 9 OS (SPARC e x86/x64 Platform Edition)
- Várias outras distribuições Linux

Configuração Mínima de Hardware

Nota: IDE NetBeans com resolução de tela em 1024x768 pixel

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	500 MHz Intel Pentium III workstation ou equivalente	512 MB	850 MB
Linux	500 MHz Intel Pentium III workstation ou equivalente	512 MB	450 MB
Solaris OS (SPARC)	UltraSPARC II 450 MHz	512 MB	450 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Série 1.8 GHz	512 MB	450 MB
Mac OS X	PowerPC G4	512 MB	450 MB

Configuração Recomendada de Hardware

Sistema Operacional	Processador	Memória	HD Livre
Microsoft Windows	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	1 GB
Linux	1.4 GHz Intel Pentium III workstation ou equivalente	1 GB	850 MB
Solaris OS (SPARC)	UltraSPARC IIIi 1 GHz	1 GB	850 MB
Solaris OS (x86/x64 Platform Edition)	AMD Opteron 100 Series 1.8 GHz	1 GB	850 MB
Mac OS X	PowerPC G5	1 GB	850 MB

Requerimentos de Software

NetBeans Enterprise Pack 5.5 executando sobre Java 2 Platform Standard Edition Development Kit 5.0 ou superior (JDK 5.0, versão 1.5.0_01 ou superior), contemplando a Java Runtime Environment, ferramentas de desenvolvimento para compilar, depurar, e executar aplicações escritas em linguagem Java. Sun Java System Application Server Platform Edition 9.

- Para **Solaris**, **Windows**, e **Linux**, os arquivos da JDK podem ser obtidos para sua plataforma em http://java.sun.com/j2se/1.5.0/download.html
- Para **Mac OS X**, Java 2 Plataform Standard Edition (J2SE) 5.0 Release 4, pode ser obtida diretamente da Apple's Developer Connection, no endereço: http://developer.apple.com/java (é necessário registrar o download da JDK).

Para mais informações: http://www.netbeans.org/community/releases/55/relnotes.html

Colaboradores que auxiliaram no processo de tradução e revisão

Alexandre Mori
Alexis da Rocha Silva
Aline Sabbatini da Silva Alves
Allan Wojcik da Silva
André Luiz Moreira
Andro Márcio Correa Louredo
Antoniele de Assis Lima
Antonio Jose R. Alves Ramos
Aurélio Soares Neto
Bruno da Silva Bonfim
Bruno dos Santos Miranda
Bruno Ferreira Rodrigues
Carlos Alberto Vitorino de Almeida
Carlos Alexandre de Sene
Carlos André Noronha de Sousa
Carlos Eduardo Veras Neves
Cleber Ferreira de Sousa
Cleyton Artur Soares Urani
Cristiano Borges Ferreira
Cristiano de Siqueira Pires
Derlon Vandri Aliendres
Fabiano Eduardo de Oliveira
Fábio Bombonato
Fernando Antonio Mota Trinta
Flávio Alves Gomes
Francisco das Chagas
Francisco Marcio da Silva
Gilson Moreno Costa
Givailson de Souza Neves
Gustavo Henrique Castellano
Hebert Julio Gonçalves de Paula
Heraldo Conceição Domingues
Hugo Leonardo Malheiros Ferreira
Ivan Nascimento Fonseca
Jacqueline Susann Barbosa
Jader de Carvalho Belarmino
João Aurélio Telles da Rocha
João Paulo Cirino Silva de Novais
João Vianney Barrozo Costa
José Augusto Martins Nieviadonski
José Leonardo Borges de Melo
José Ricardo Carneiro
Kleberth Bezerra G. dos Santos
Lafaiete de Sá Guimarães
Leandro Silva de Morais
Leonardo Leopoldo do Nascimento
Leonardo Pereira dos Santos
Leonardo Rangel de Melo Filardi
Lucas Mauricio Castro e Martins
Luciana Rocha de Oliveira
Luís Carlos André
Luís Octávio Jorge V. Lima
Luiz Fernandes de Oliveira Junior
Luiz Victor de Andrade Lima
Manoel Cotts de Queiroz
Marcello Sandi Pinheiro
Marcelo Ortolan Pazzetto
Marco Aurélio Martins Bessa
Marcos Vinicius de Toledo
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Mauricio Azevedo Gamarra
Mauricio da Silva Marinho
Mauro Cardoso Mortoni
Mauro Regis de Sousa Lima
Namor de Sá e Silva
Néres Chaves Rebouças
Nolyanne Peixoto Brasil Vieira
Paulo Afonso Corrêa
Paulo José Lemos Costa
Paulo Oliveira Sampaio Reis
Pedro Antonio Pereira Miranda
Pedro Henrique Pereira de Andrade
Renato Alves Félix
Renato Barbosa da Silva
Reyderson Magela dos Reis
Ricardo Ferreira Rodrigues
Ricardo Ulrich Bomfim
Robson de Oliveira Cunha
Rodrigo Pereira Machado
Rodrigo Rosa Miranda Corrêa
Rodrigo Vaez
Ronie Dotzlaw
Rosely Moreira de Jesus
Seire Pareja
Sergio Pomerancblum
Silvio Sznifer
Suzana da Costa Oliveira
Tásio Vasconcelos da Silveira
Thiago Magela Rodrigues Dias
Tiago Gimenez Ribeiro
Vanderlei Carvalho Rodrigues Pinto
Vanessa dos Santos Almeida
Vastí Mendes da Silva Rocha
Wagner Eliezer Roncoletta

Auxiliadores especiais

Revisão Geral do texto para os seguintes Países:

- **Brasil** – Tiago Flach
- **Guiné Bissau** – Alfredo Cá, Bunene Sisse e Buon Olossato Quebi – ONG Asas de Socorro

Coordenação do DFJUG

- **Daniel deOliveira** – JUGLeader responsável pelos acordos de parcerias
- **Luci Campos -** Idealizadora do DFJUG responsável pelo apoio social
- **Fernando Anselmo -** Coordenador responsável pelo processo de tradução e revisão, disponibilização dos materiais e inserção de novos módulos
- **Regina Mariani -** Coordenadora responsável pela parte jurídica
- **Rodrigo Nunes -** Coordenador responsável pela parte multimídia
- **Sérgio Gomes Veloso -** Coordenador responsável pelo ambiente JEDI™ (Moodle)

Agradecimento Especial

John Paul Petines – Criador da Iniciativa JEDI™
Rommel Feria – Criador da Iniciativa JEDI™

1. Objetivos

Esta lição introduz algumas técnicas avançadas de programação. Veremos detalhes sobre recursão e tipos de dados abstratos.

Ao final desta lição, o estudante será capaz de:

- Definir e aplicar recursão na programação
- Diferenciar entre pilhas e filas
- Implementar pilhas e filas seqüenciais
- Implementar pilhas e filas encadeadas
- Usar as classes *Collection* existentes

2. Recursão

2.1. O que é recursão?

Recursão é uma técnica poderosa para resolução de problemas que pode ser aplicada quando a natureza do problema exigir repetição. Ainda que estes tipos de problemas sejam solucionáveis utilizando a iteração (ex. uso de instruções de repetição como *for, while e do-while*). De fato, a iteração é uma ferramenta mais eficiente se comparada à recursão, mas a recursão provê uma solução mais elegante para o problema. Na recursão, é permitido aos métodos chamarem a si mesmos. O dado passado no método como argumento é armazenado temporariamente em uma pilha antes que a chamada do método seja completada.

2.2. Recursão versus Iteração

Para uma boa compreensão da recursão, observe como a técnica de repetição pode variar.

Determinados problemas de natureza repetitiva requerem uso explícito de estruturas de controle de repetição. Por outro lado, para a recursão, a tarefa é repetida chamando um mesmo método sucessivas vezes. A idéia aqui é definir o problema em pedaços de instâncias menores de si mesmo. Considere o cálculo do fatorial de um determinado inteiro. Esta recursão pode ser descrita da seguinte maneira: fatorial(n) = fatorial(n-1) * n; fatorial(1) = 1. Por exemplo, o fatorial de 2 é igual ao fatorial(1)*2, o qual é igual a 2. O fatorial de 3 é 6, o qual é igual ao fatorial(2)*3.

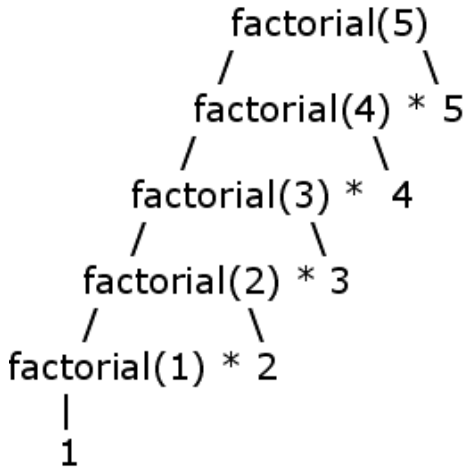


Figura 1: Exemplo de fatorial

Com a iteração, o processo termina quando a condição do laço falha. No caso de se utilizar recursão, o processo é finalizado uma vez que for encontrada uma condição particular em que este seja satisfeito. Por exemplo, como observado na definição recursiva do fatorial, o caso mais simples é quando a entrada é o 1. O 1 para esse problema é o caso básico.

O uso da iteração e da recursão pode em ambos os casos levar a laços infinitos se eles não forem usados corretamente.

A vantagem da iteração sobre a recursão é o desempenho. A iteração é mais rápida se comparada à recursão. Na recursão ocorre a passagem de parâmetros para o método, e isto pode consumir algum tempo de CPU. Entretanto, a recursão encoraja as boas práticas da engenharia de software pois, usualmente, resulta em um código mais fácil de compreender e promove a reutilização de uma solução previamente implementada.

Escolher entre iteração e recursão é a melhor maneira de balancear um bom desempenho e uma boa engenharia de software.

2.3. Fatorial: Um exemplo

O código a seguir mostra como calcular fatorial utilizando a técnica da iteração.

```
class FatorialDemo {
    public int factorial(int n) {
        int result = 1;
        for (int i = n; i > 1; i--) {
            result *= i;
        }
        return result;
    }
    public static void main(String args[]) {
        FatorialDemo fatorialDemo = new FatorialDemo();
        System.out.println(fatorialDemo.factorial(5));
    }
}
```

Aqui é a mesma classe entretanto o método utiliza a recursão:

```
class FatorialDemo {
    public int factorial(int n) {
       if (n == 1) {             /* saída do método */
          return 1;
       }
       /* Definição recursiva; Invoca a si mesmo */
       return factorial(n-1)*n;
    }
    public static void main(String args[]) {
        FatorialDemo fatorialDemo = new FatorialDemo();
        System.out.println(fatorialDemo.factorial(5));
    }
}
```

Considere a saída esperada de um determinado código de inteiros. Observe o que acontece no código para uma determinada entrada. Aqui estão umas amostras de inteiros com suas saídas correspondentes.

a) Entrada: 1

```
1
```

b) Entrada: 3

```
6
```

c) Entrada: 5

```
120
```

2.4. Imprimindo n em alguma base: Outro exemplo

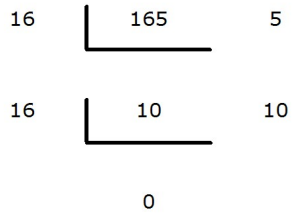
Agora considere o problema de imprimir um número decimal em uma base específica definida pelo usuário. Observe que a solução para isto é usar a divisão repetitiva e escrever o resto no inverso. O processo termina quando a divisão é menor que a base. Assuma que a entrada do número decimal é 10 e converteremos para a base 8. Aqui está a solução usando caneta e papel.

```
8 |  10      2
  |_____

8 |  1       1
  |_____

  0
```

Na solução, 10 é equivalente a 12 base 8. Aqui está outro exemplo. A entrada decimal é 165 para ser convertida em base 16.



165 é equivalente ao valor A5 na base 16. Note: A=10.

Essa é a solução iterativa para esse problema.

```
class DecToOthers {
   public static void main(String args[]) {
      int num = Integer.parseInt(args[0]);
      int base = Integer.parseInt(args[1]);
      printBase(num, base);
   }
   static void printBase(int num, int base) {
      int rem = 1;
      String digits = "0123456789abcdef";
      String result = "";
      /* o passo iterativo */
      while (num!=0) {
         rem = num%base;
         num = num/base;
         result = result.concat(digits.charAt(rem)+"");
      }
      /* Imprimindo o inverso do resultado */
      for(int i = result.length()-1; i >= 0; i--) {
         System.out.print(result.charAt(i));
      }
   }
}
```

Agora, essa é a recursão equivalente da solução acima.

```
class DecToOthersRecur {
   static void printBase(int num, int base) {
      String digits = "0123456789abcdef";
      /* Passo recursivo*/
      if (num >= base) {
         printBase(num/base, base);
      }
      /* Caso básico: num < base */
      System.out.print(digits.charAt(num%base));
   }
   public static void main(String args[]) {
      int num = Integer.parseInt(args[0]);
      int base = Integer.parseInt(args[1]);
      printBase(num, base);
   }
}
```

Para atingir uma melhor compreensão do código, observe o conjunto de entrada. Aqui estão alguns exemplos de entrada.

a) Num: 10, base: 2
```
1010
```
b) Num: 100, base: 8
```
144
```
c) Num: 200, base: 9
```
242
```

3. Tipos de Dados Abstratos

3.1. O que é um Tipo de Dado Abstrato?

Um tipo de dado abstrato (TDA) é uma coleção de elementos de dados providos com um conjunto de operações que são definidas nos elementos de dados. Pilhas, filas e árvores binárias são alguns exemplos de TDA. Nessa lição, aprenderemos sobre pilhas (stacks) e filas (queues).

3.2. Pilhas

Uma pilha é um conjunto de elementos de dados onde à manipulação desse elemento é permitida somente no topo (top) da pilha. A pilha é uma coleção de dados ordenado o qual a disciplina "último a entrar é o primeiro a sair" (do inglês: "last in, first out" - LIFO) é imposta. Pilhas são usadas para uma variedade de aplicações como o reconhecimento de padrões e a conversão entre as notações infixa, pós-fixa e pré-fixa.

As duas operações associadas às pilhas são a de adicionar (push) e a de remover (pop). Push simplesmente insere o elemento no topo da pilha e, por outro lado, pop remove o elemento do topo da pilha. Para compreender como as pilhas trabalham, imagine adicionar ou remover um prato de uma pilha de pratos sua natureza, instintivamente, sugere que seja removido somente o que está no topo da pilha porque do contrário, existe um perigo de que todos os pratos caiam.

Aqui está um exemplo ilustrando como uma pilha funciona.

n-1		
...		
6		
5	Jayz	***Top***
4	KC	
3	Jojo	
2	Toto	
1	Kyla	
0	DMX	***bottom***

Tabela 1: Ilustração da pilha

A pilha está cheia se o topo alcança a célula n-1. Se o topo é igual a n-1, é o indício de que a pilha está cheia.

3.3. Filas

A fila é outro exemplo de TDA. Fila é uma coleção ordenada de elementos o qual prima pela disciplina "primeiro a entrar é o primeiro a sair" (do inglês: "first-in, first-out" - FIFO). As aplicações da fila incluem escalonamento de serviços do sistema operacional, sorteamento topológico e o grafo transversal.

Enfileirar (enqueue) e Desenfileirar (dequeue) são operações associadas com as filas. Enqueue se refere ao modo de inserir no final da fila e, por outro lado, dequeue, o de remover o elemento do início da fila. Para recordar como uma fila trabalha, pense no significado literal de uma fila – uma linha. Suponha uma fila de um banco e estamos esperando para ser atendido. Se uma pessoa chegar ela deverá ficar no final da fila. Caso contrário haveria, provavelmente, algum desentendimento. Essa é a maneira que funciona a inserção (enqueue). Dessa forma, é

necessário que a primeira pessoa seja atendida para somente depois ser a vez da segunda pessoa da fila, e assim sucessivamente. Essa é a maneira que trabalha a remoção (dequeue).

Aqui está um exemplo de como a fila funciona.

0	1	2	3	4	5	6	7	8	9	...	n-1
		Eve	Jayz	KC	Jojo	Toto	Kyla	DMX			
		front						***end***	←	Insere	
									→	Deleta	

Tabela 2: Ilustração da fila

A fila está vazia se o final é menor do que o início. Por outro lado, a fila está cheia quando o final é igual a n-1.

3.4. Representação seqüencial e encadeada

TDA podem ser representados usando uma representação seqüencial ou encadeada. É fácil criar uma representação seqüencial com o uso de arrays. Entretanto, o problema com o uso de arrays está no seu limite de tamanho, o qual é fixo. O espaço de memória ou é desperdiçado ou não é o bastante com uso dos arrays. Considere o seguinte cenário. Deve ser criado um array com uma capacidade de armazenar um máximo de 50 elementos. Se o usuário inserir somente 5 elementos, 45 espaços estariam desperdiçados. Por outro lado, se o usuário desejar inserir 51 elementos, o espaço provido pelo array não suportaria.

Comparativamente à representação seqüencial, uma representação encadeada poderia ser ligeiramente mais difícil de se implementar, mas seria muito mais flexível. O encadeamento se adapta à necessidade de memória requerida pelo usuário. Uma explanação mais detalhada da representação encadeada será discutida na sessão a seguir.

Representação seqüencial de uma pilha de inteiros:

```
public class StackDemo {
   int top = -1;      /* inicialmente, a pilha está vazia */
   int memSpace[];    /* armazenamento para inteiros */
   int limit;         /* tamanho do espaço de memória */

   /* Recebe o tamanho inicial da pilha */
   public StackDemo(int size) {
      memSpace = new int[size];
      limit = size;
   }
   /* Adiciona um elemento */
   public boolean push(int value) {
      top++;
      if (top < limit) {
         memSpace[top] = value;
      } else {
         top--;
         return false;
      }
      return true;
   }
   /* Remove um elemento retornando a quantidade de elementos
      restantes *
   public int pop() {
      int temp = -1;
      if (top >= 0) {
```

```
            temp = memSpace[top];
            top--;
         } else {
            return -1;
         }
         return temp;
      }
      public static void main(String args[]) {
         StackDemo stackDemo = new StackDemo(3);
         stackDemo.push(1);
         stackDemo.push(2);
         stackDemo.push(3);
         stackDemo.push(4);
         System.out.println(stackDemo.pop());
         System.out.println(stackDemo.pop());
         System.out.println(stackDemo.pop());
         System.out.println(stackDemo.pop());
      }
   }
```

A figura a seguir mostra um trecho do método principal de uma pilha seqüencial.

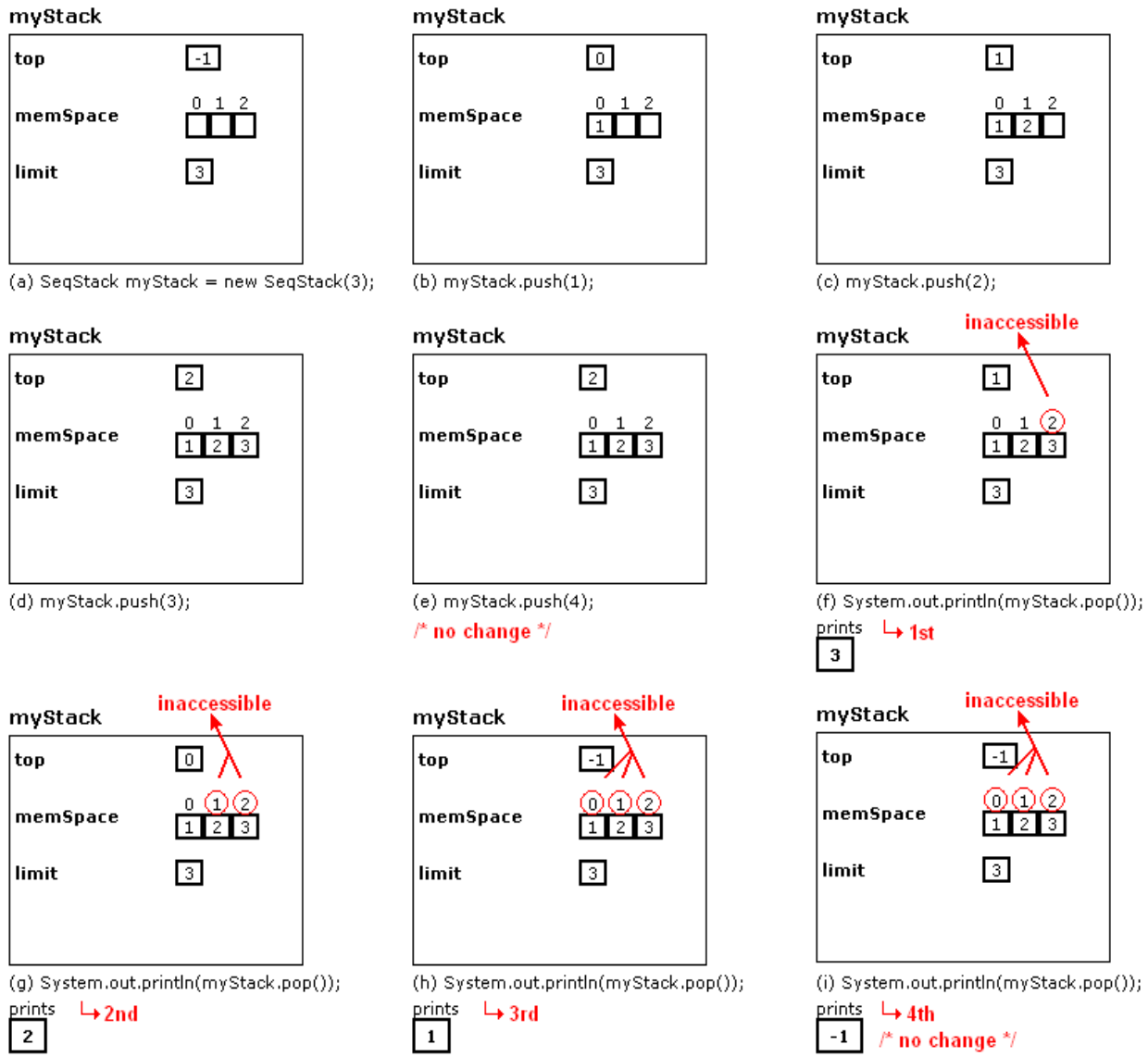


Figura 2: Trecho de uma pilha seqüencial

3.5. Lista encadeada

Antes de implementar a representação encadeada de uma pilha, vamos, primeiramente, estudar como criar uma representação encadeada. Em particular, nós devemos olhar para as listas encadeadas.

A lista encadeada é uma estrutura dinâmica, em oposição ao array, que é uma estrutura estática. A grande vantagem da lista encadeada é que a lista pode crescer ou diminuir de tamanho dependendo da necessidade do usuário. Uma lista encadeada é definida como uma coleção de nós, cada qual consiste de um elemento e um encadeamento ou ponteiro para o próximo nó na lista. A figura a seguir mostra como um nó aponta para outro.

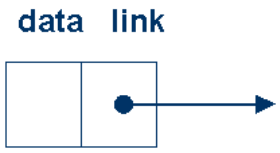


Figura 3: Um nó

Aqui está um exemplo de uma lista encadeada não vazia com três nós.

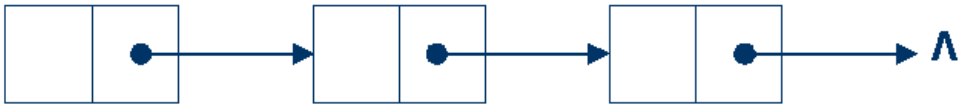


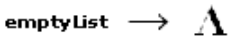
Figura 4: Uma lista encadeada não vazia com três nós

Esta é uma classe que demonstra como implementar um nó. Pode ser usada para criar uma lista encadeada:

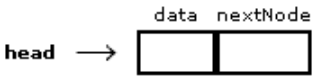
```
class Node {
   int data;         /* dados inteiro contidos em um nó */
   Node nextNode;    /* o próximo nó da lista */
}
class NodeDemo {
   public static void main(String args[]) {
      Node emptyList = null;     /* cria uma lista vazia */
      /* Nó cabeça para o primeiro nó da lista */
      Node head = new Node();
      /* inicializa o primeiro nó da lista */
      head.data = 5;
      head.nextNode = new Node();
      head.nextNode.data = 10;
      /* marca nulo para o final de fila */
      head.nextNode.nextNode = null;
      /* imprime elementos da lista */
      Node currNode = head;
      while (currNode != null) {
         System.out.println(currNode.data);
         currNode = currNode.nextNode;
      }
   }
}
```

Trecho da execução do método *TestNode*.

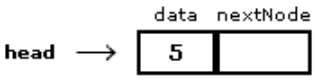
(a) Node emptyList = null;



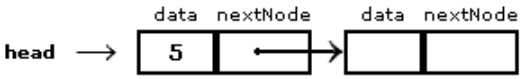
(b) Node head = new Node();



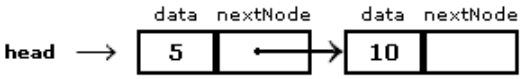
(c) head.data = 5;



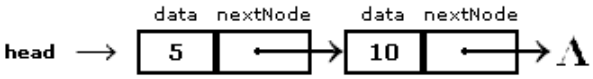
(d) head.nextNode() = new Node();



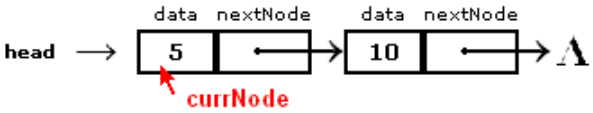
(e) head.nextNode.data = 10;



(f) head.nextNode.nextNode = null;



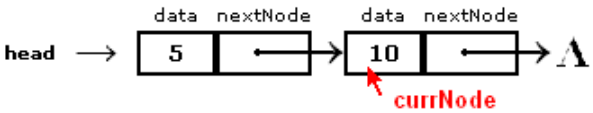
(g) Node currNode = head;



(h) while (currNode != null)
/* condition is true */

(i) System.out.println(currNode.data);
prints

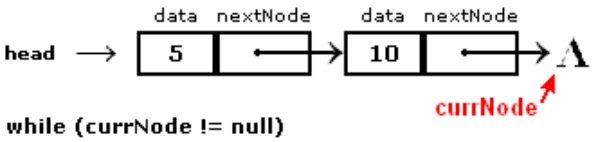
(j) currNode = currNode.nextNode;



(k) while (currNode != null)
/* condition is still true */

(l) System.out.println(currNode.data);
prints

(m) currNode = currNode.nextNode;



(n) while (currNode != null)
/* condition is now false */

Figura5: Percurso do nó de teste

3.6. Representação encadeada de uma pilha de inteiros

Agora que aprendemos sobre lista encadeada, estamos prontos para implementar uma representação encadeada de uma pilha.

```
public class DynamicIntStackDemo {
   private IntStackNode top;     /* cabeça ou topo da pilha */
   class IntStackNode {          /* nó classe */
      int data;
      IntStackNode next;
      IntStackNode(int n) {
         data = n;
         next = null;
      }
   }
   public void push(int n) {
      /* não há necessidade de verificar se está cheio */
      IntStackNode node = new IntStackNode(n);
      node.next = top;
      top = node;
   }
   public int pop() {
      /* dispara uma exceção se a lista estiver vazia */
      if (isEmpty())
         return -1;
      int n = top.data;
      top = top.next;
      return n;
   }
   public boolean isEmpty() {
      return top == null;
   }
   public static void main(String args[]) {
      DynamicIntStackDemo myStack = new DynamicIntStackDemo();
      myStack.push(5);
      myStack.push(10);
      /* Imprime os elementos da pilha */
      IntStackNode currNode = myStack.top;
      while (currNode!=null) {
         System.out.println(currNode.data);
         currNode = currNode.next;
      }
      System.out.println(myStack.pop());
      System.out.println(myStack.pop());
   }
}
```

Aqui está a saída esperada do código acima:

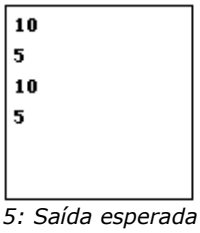


Figura 5: Saída esperada da pilha dinâmica

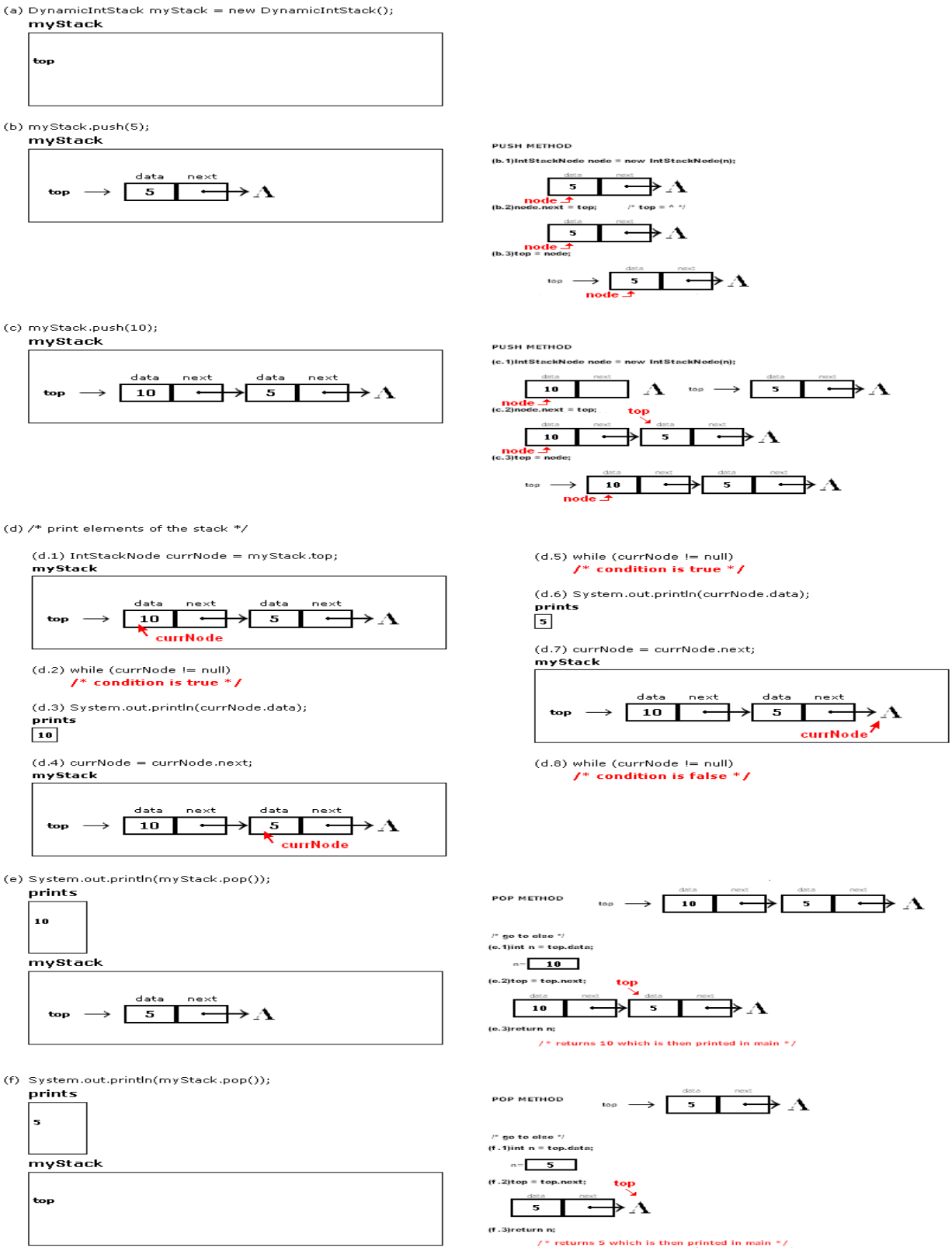


Figura 6: Trecho da pilha dinâmica

3.7. Java Collections

Os fundamentos dos tipos de dados abstratos mostrados como o básico das listas encadeadas, pilhas e filas. Esses tipos de dados abstratos já estão implementados e incluídos no Java. As classes pilha e lista encadeada estão disponíveis para uso sem exigir um completo entendimento desses conceitos. Entretanto, como cientista da computação, é importante que compreendamos os tipos de dados abstratos. Na verdade, uma explanação detalhada foi dada na sessão anterior. Com a liberação da versão do J2SE 5.0, a interface de Filas (*Queue*) foi disponibilizada. Para mais detalhes dessas classes e interfaces, por favor, veja a documentação da Java API.

Java provê outras coleções de classes e interfaces, as quais estão disponíveis no pacote *java.util*. Exemplos de classes *Collections* incluem *LinkedList*, ArrayList, *HashSet* e *TreeSet*. Essas classes são implementações de diferentes coleções e interfaces. A hierarquia das coleções de interface inicia-se com a interface *Collection*. Uma *Collection* é apenas um grupo de objetos que são conhecidos e seus elementos. Coleções podem permitir duplicidades e não exigir uma ordem específica.

O SDK não provê nenhuma implementação dessas interfaces, mas sub-interfaces diretas, a interface *Set* e a interface *List* estão disponíveis. Agora, qual é a diferença entre essas duas interfaces? Um *Set* é uma coleção não ordenada que não contém duplicidade. Por outro lado, um *List* é uma coleção ordenada de elementos onde duplicidades são permitidas. *HashSet, LinkedHashSet* e *TreeSet* são algumas implementações de classes conhecidas da interface Set. *ArrayList, LinkedList* e *Vector* são algumas implementações de classes conhecidas da interface List.

<Interface Raiz> *Collection*					
<interface> *Set*			*<interface>* *List*		
<Implementada pelas classes>			*<Implementada pelas classes>*		
HashSet	LinkedHashSet	TreeSet	ArrayList	LinkedList	Vector

Tabela 3: Java collections

A seguir está uma lista dos métodos de uma coleção providas na *API Collections do Java 2 Platform SE v1.4.1*. Em *Java 2 Platform SE v1.5.0*, esses métodos foram modificados para acomodar tipos genéricos. Tipos genéricos não serão discutidos ainda. É aconselhável considerar esses métodos primeiramente. Recomendo que os novos métodos da coleção sejam consultados uma vez que compreenda os tipos genéricos, os quais são discutidos no próximo capítulo.

Métodos da Collection
`public boolean add(Object o)`
Insere o objeto enviado como argumento na coleção. Retorna verdadeiro se o objeto foi adicionado com sucesso.
`public void clear()`
Remove todos os elementos da coleção.
`public boolean remove(Object o)`
Remove o objeto enviado como argumento na coleção. Retorna verdadeiro se o objeto se o objeto foi encontrado e removido com sucesso.
`public boolean contains(Object o)`
Retorna verdadeiro se a coleção contém o objeto enviado no argumento.
`public boolean isEmpty()`
Retorna verdadeiro se a coleção estiver vazia.
`public int size()`

Retorna o número de elementos na coleção.
`public Iterator iterator()`
Retorna um objeto do tipo Iterator que permite percorrer os elementos da coleção.

Tabela 4: Métodos da classe Collection

Por favor, consulte a documentação da API para uma lista mais completa dos métodos encontrados nas interfaces *Collection*, *List* e *Set*.

Java SDK apresenta diversas formas de implementação de uma lista encadeada. A classe *LinkedList* contém métodos que permitem que listas encadeadas sejam usadas como pilhas, filas ou algum outro TDA. O código a seguir mostra como usar a classe *LinkedList*.

```
import java.util.*;

class LinkedListDemo {
   public static void main(String args[]) {
      LinkedList list = new LinkedList();
      list.add(new Integer(1));
      list.add(new Integer(2));
      list.add(new Integer(3));
      list.add(new Integer(1));
      System.out.println(list + ", size = " + list.size());
      list.addFirst(new Integer(0));
      list.addLast(new Integer(4));
      System.out.println(list);
      System.out.println(list.getFirst() + ", " + list.getLast());
      System.out.println(list.get(2) + ", " + list.get(3));
      list.removeFirst();
      list.removeLast();
      System.out.println(list);
      list.remove(new Integer(1));
      System.out.println(list);
      list.remove(2);
      System.out.println(list);
   }
}
```

O *ArrayList* é uma versão redimensionável de um array. Ela implementa a interface *List*. Analise o código a seguir.

```
import java.util.*;

class ArrayListDemo {
   public static void main(String args[]) {
      ArrayList al = new ArrayList(2);
      System.out.println(al + ", size = " + al.size());
      al.add("R");
      al.add("U");
      al.add("O");
      System.out.println(al + ", size = " + al.size());
      al.remove("U");
      System.out.println(al + ", size = " + al.size());
      ListIterator li = al.listIterator();
      while (li.hasNext())
         System.out.println(li.next());
      Object a[] = al.toArray();
      for (int i=0; i<a.length; i++)
         System.out.println(a[i]);
   }
```

```
}
```

O *HashSet* é um tipo de implementação da interface Set que usa uma tabela *hash*. O uso de uma tabela *hash* permite buscar os elementos de forma fácil e rápida. A tabela usa uma fórmula que determina onde um objeto está armazenado. HashSet foi utilizado na seguinte classe:

```
import java.util.*;

class HashSetDemo {
   public static void main(String args[]) {
      HashSet hs = new HashSet(5);
      System.out.println(hs.add("one"));
      System.out.println(hs.add("two"));
      System.out.println(hs.add("one"));
      System.out.println(hs.add("three"));
      System.out.println(hs.add("four"));
      System.out.println(hs.add("five"));
      System.out.println(hs);
   }
}
```

O *TreeSet* é uma implementação da interface *Set* que usa uma árvore. Esta classe assegura que o conjunto será organizado em uma ordem ascendente. Observe como a classe *TreeSet* foi usada no seguinte código fonte.

```
import java.util.*;

class TreeSetDemo {
   public static void main(String args[]) {
      TreeSet ts = new TreeSet();
      ts.add("one");
      ts.add("two");
      ts.add("three");
      ts.add("four");
      System.out.println(ts);
   }
}
```

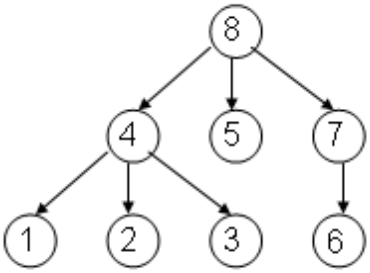


Figura 7: Exemplo de um TreeSet

Parceiros que tornaram JEDI™ possível

Instituto CTS
Patrocinador do DFJUG.

Sun Microsystems
Fornecimento de servidor de dados para o armazenamento dos vídeo-aulas.

Java Research and Development Center da Universidade das Filipinas
Criador da Iniciativa JEDI™.

DFJUG
Detentor dos direitos do JEDI™ nos países de língua portuguesa.

Banco do Brasil
Disponibilização de seus *telecentros* para abrigar e difundir a Iniciativa JEDI™.

Politec
Suporte e apoio financeiro e logístico a todo o processo.

Borland
Apoio internacional para que possamos alcançar os outros países de língua portuguesa.

Instituto Gaudium/CNBB
Fornecimento da sua infra-estrutura de hardware de seus servidores para que os milhares de alunos possam acessar o material do curso simultaneamente.