Módulo 3

Estruturas de Dados

Lição 2

Stack

Versão 1.0 - Mai/2007

Autor
Joyce Avestro

Equipe
Joyce Avestro
Florence Balagtas
Rommel Feria
Reginald Hutcherson
Rebecca Ong
John Paul Petines
Sang Shin
Raghavan Srinivas
Matthew Thompson

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
Anna Carolina Ferreira da Rocha
Antonio Jose R. Alves Ramos
Aurélio Soares Neto
Bárbara Angélica de Jesus Barbosa
Bruno da Silva Bonfim
Bruno dos Santos Miranda
Bruno Ferreira Rodrigues
Carlos Alexandre de Sene
Carlos Eduardo Veras Neves
Cleber Ferreira de Sousa
Everaldo de Souza Santos
Fabrício Ribeiro Brigagão
Fernando Antonio Mota Trinta
Frederico Dubiel
Givailson de Souza Neves
Jacqueline Susann Barbosa
João Paulo Cirino Silva de Novais
João Vianney Barrozo Costa
José Augusto Martins Nieviadonski
José Ricardo Carneiro
Kleberth Bezerra G. dos Santos
Kefreen Ryenz Batista Lacerda
Leonardo Leopoldo do Nascimento
Lucas Vinícius Bibiano Thomé
Luciana Rocha de Oliveira
Luís Carlos André
Luiz Fernandes de Oliveira Junior
Luiz Victor de Andrade Lima
Marco Aurélio Martins Bessa
Marcos Vinicius de Toledo
Marcus Borges de S. Ramos de Pádua
Maria Carolina Ferreira da Silva
Massimiliano Giroldi
Mauricio da Silva Marinho
Mauro Cardoso Mortoni
Mauro Regis de Sousa Lima
Namor de Sá e Silva
Nolyanne Peixoto Brasil Vieira
Paulo Afonso Corrêa
Paulo Oliveira Sampaio Reis
Pedro Antonio Pereira Miranda
Renato Alves Félix
Renê César Pereira
Reyderson Magela dos Reis
Ricardo Ulrich Bomfim
Robson de Oliveira Cunha
Rodrigo Fernandes Suguiura
Rodrigo Vaez
Ronie Dotzlaw
Rosely Moreira de Jesus
Seire Pareja
Silvio Sznifer
Tiago Gimenez Ribeiro
Vanderlei Carvalho Rodrigues Pinto
Vanessa dos Santos Almeida

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
- **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

Uma s*tack* (pilha) é uma ordem linear de elementos obedecendo a regra: "o último que entrar é o primeiro a sair", é conhecida como listas **LIFO** (*Last In First Out*).

É semelhante a um conjunto de caixas em um armazém, onde só a caixa de topo pode ser retirada e não há acesso às outras caixas. Quando adicionamos uma caixa, é sempre é colocada no topo.

Stack é utilizada em reconhecimentos de padrões, listas e árvores transversais, avaliação de expressões, soluções de recursividade e muito mais. As duas operações básicas para manipulação de dados em uma *stack* são "*push*" e "*pop*", ou seja, inserção e retirada de elementos do topo da *stack* respectivamente.

Ao final desta lição, o estudante será capaz de:

- Explicar os conceitos básicos e operações em uma *stack* ADT
- Implementar uma *stack* ADT usando representação seqüencial e de ligação
- Discutir aplicações de *stack*: Os problemas de reconhecimento de padrões e conversões do tipo ***infix*** para ***postfix***
- Explicar como múltiplas *stacks* podem ser armazenadas utilizando ***array*** de uma dimensão
- Realocação de memória durante um transbordamento (*estouro*) de um *array* com múltiplas s*tacks* utilizando algoritmos ***unit-shift policy*** e ***Garwick's***

2. Operações em *Stack*

Como já mencionado, Interfaces (*Application Programming Interface* ou API) são usadas para implementar ADT em Java. Esta é a interface Java para *stack*:

```
public interface Stack {
  public int size(); // retorna o tamanho da stack
  public boolean isEmpty(); // verifica se está vazia
  public Object top() throws StackException;
  public Object pop() throws StackException;
  public void push(Object item) throws StackException;
}
```

StackException é uma extensão de RuntimeException:

```
class StackException extends RuntimeException{
  public StackException(String err){
    super(err);
  }
}
```

As *stacks* possuem duas implementações possíveis - *array* unidimensional seqüencialmente alocado ou uma lista linear encadeada. Entretanto, a implementação que será usada é uma interface *Stack*.

A seguir as operações de uma *stack*:

- Verificar o tamanho
- Verificar se está vazia
- Pegar o elemento do topo sem excluí-lo
- Inserir um novo elemento (*push*)
- Apagar um elemento do topo (*pop*)

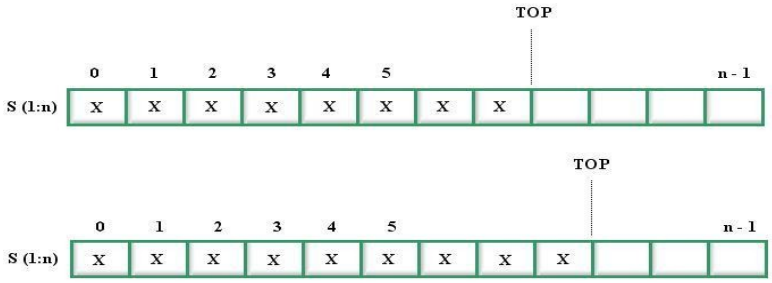

Figura 1. Operação PUSH

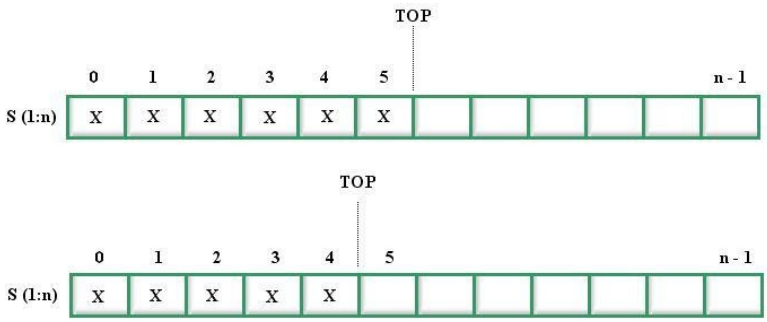

Figura 2. Operação POP

3. Representação seqüencial

A alocação seqüencial de uma *stack* faz uso de *arrays*, conseqüentemente o tamanho é estático. A *stack* está vazia se o topo=-1 e cheia se o topo=n-1. Tentar apagar um elemento de uma *stack* vazia causa um ***underflow*** enquanto a inserção de um elemento em uma *stack* cheia causa um **overflow**. A figura a seguir mostra um exemplo de uma *stack* ADT:

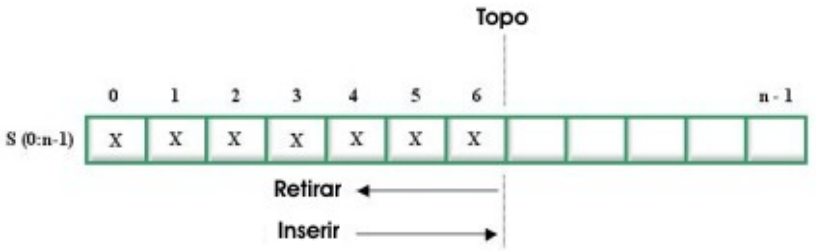

Figura 3. Retirar e inserir

A seguir, a implementação de uma *stack* usando a representação seqüencial:

```
public class ArrayStack implements Stack {
  // Array usado para implementar a stack
  Object S[];

  // Inicializa a stack em vazio
  int top = -1;

  // Inicializa a stack para o padrão 0
  public ArrayStack() {
    this(0);
  }

  // Inicializa a stack para ser o comprimento recebido
  public ArrayStack(int c) {
    S = new Object[c];
  }

  // Implementação do método size
  public int size() {
    return (top+1);
  }

  // Implementação do método isEmpty
  public boolean isEmpty() {
    return (top < 0);
  }

  // Implementação do método top
  public Object top() {
    if (isEmpty()) throw new StackException("Stack empty.");
    return S[top];
  }

  // Implementação do método pop
  public Object pop() {
    Object item;
    if (isEmpty()) throw new StackException("Stack underflow.");
    item = S[top];
```

```
    S[top--] = null;
    return item;
  }

  // Implementação do método push
  public void push(Object item) {
    if (size() == s.length)
      throw new StackException("Stack overflow.");
    S[++top]=item;
  }
}
```

Podemos testar esta representação com a seguinte classe:

```
public class TestArrayStack {
  public static void main(String [] args) {
    ArrayStack stack = new ArrayStack(4);
    stack.push("1");
    stack.push("2");
    stack.push("3");
    stack.push("4");
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
  }
}
```

Como resultado, lembre-se que a *stack* armazena "Empilhando" os dados, e retirá-os do último ao primeiro elemento armazenado.

4. Representação Encadeada

Uma lista acoplada de *nodes* poderia ser utilizada para implementar uma *stack*. Na representação acoplada, um *node* com a estrutura definida a seguir será usada:

```
class Node {
  private Object info;
  private Node link;
  public Node(Object o, Node n) {
    info = o;
    link = n;
  }
}
```

A figura seguinte mostra uma *stack* representada como uma lista linear encadeada:

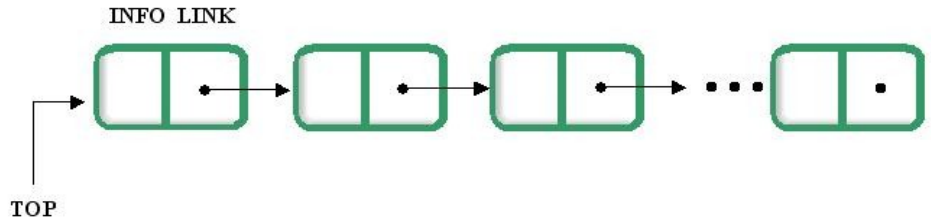

Figura 4. Representação Encadeada

O código Java a seguir implementa a *stack* utilizando representação encadeada:

```
public class LinkedStack implements Stack {
  private Node top;

  // O número de elementos na stack
  private int numElements = 0;

  // Implementaçao do método size
  public int size() {
    return (numElements);
  }

  // Implementaçao do método isEmpty
  public boolean isEmpty() {
    return (top == null);
  }

  // Implementaçao do método top
  public Object top() {
    if (isEmpty()) throw new
      StackException("Stack empty.");
    return top.info;
  }

  // Implementaçao do método pop
  public Object pop() {
    Node temp;
    if (isEmpty())
      throw new StackException("Stack underflow.");
    temp = top;
```

```
      top = top.link;
      return temp.info;
    }

    // Implementaçao do método push
    public void push(Object item) {
      Node newNode = new Node(item);
      newNode.link = top;
      top = newNode;
    }
  }
```

Podemos testar esta representação com a seguinte classe:

```
public class TestArrayStack {
  public static void main(String [] args) {
    LinkedStack stack = new LinkedStack();
    stack.push("1");
    stack.push("2");
    stack.push("3");
    stack.push("4");
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
    System.out.println(stack.pop());
  }
}
```

5. Aplicação Exemplo: Problema do Reconhecimento de Padrão

Dado o conjunto L = { wcw^R | w ⊂ { a, b }+ }, onde $\mathbf{w^R}$ é o reverso de **w**. **L** define uma linguagem que contém um conjunto infinito de strings palíndromos. **w** não pode ser uma *string* vazia. Exemplos são *aca, abacaba, bacab, bcb* e *aacaa*.

O algoritmo seguinte pode ser usado para resolver o problema:

1. Pegue o próximo caractere **a** ou **b** da *string* de entrada e insira na *stack*; repita até o símbolo **c** ser encontrado.
2. Pegue o próximo caractere **a** ou **b** da *string* de entrada, abra a *stack* e compare. Se os dois símbolos batem, continue, de outra maneira, pare – a *string* não está em L.

A seguir estão os estados adicionais nos quais a *string* de entrada pode ser encontrada para não estar em L:

1. O fim da *string* foi atingida mas o **c** não foi encontrado.
2. O fim da *string* foi atingido mas a *stack* não está vazia.
3. A *stack* está vazia mas o fim da *string* não foi atingido ainda.

Os exemplos a seguir ilustram como o algoritmo trabalha:

Entrada	***Ação***	***Stack***
abbabcbabba	**------**	(bottom) --> (top)
abbabcbabba	Entre a	a
bbabcbabba	Entre b	ab
babcbabba	Entre b	abb
abcbabba	Entre a	abba
bcbabba	Entre b	abbab
cbabba	Descarte c	abbab
babba	Abra, compare b e b --> ok	abba
abba	Abra, compare a e a --> ok	abb
bba	Abra, compare b e b --> ok	ab
ba	Abra, compare b e b --> ok	a
a	Abra, compare a e a --> ok	-
-	**Sucesso**	

Tabela 1: Execução do algoritmo

Entrada	***Ação***	***Stack***
abacbab	**------**	(bottom) --> (top)
abacbab	Entre a	a
bacbab	Entre b	ab
acbab	Entre a	aba
cbab	Descarte c	aba

bab	Abra, compare a e b --> **não batem, na string**	ba

Tabela 2: Execução do algoritmo

No primeiro exemplo, a *string* será aceita enquanto que no segundo não.

A seguir temos uma classe Java utilizada para implementar o padrão *recognizer*:

```
public class PatternRecognizer{
  ArrayStack S = new ArrayStack(100);

  public static void main(String[] args) {
    PatternRecognizer pr = new PatternRecognizer();
    if (args.length < 1)
      System.out.println("Usage: PatternRecognizer <input string>");
    else {
      boolean inL = pr.recognize(args[0]);
      if (inL)
        System.out.println(args[0] + " is in the language.");
      else
        System.out.println(args[0] + " is not in the language.");
    }
  }

  boolean recognize(String input) {
    int i=0; // Indicador de caractere corrente

    // Enquanto c não é encontrado, entre com um caractere na stack
    while ((i < input.length()) && (input.charAt(i) != 'c')) {
      S.push(input.substring(i, i+1));
      i++;
    }

    // O fim da string foi atingido mas o c não foi encontrado
    if (i == input.length()) return false;

    // Descarte o c, mova para o próximo caractere
    i++;

    // O ultimo character é c
    if (i == input.length()) return false;

    while (!S.isEmpty()) {
      // Se o character de entrada e o outro no topo da stack não baterem
      if (!(input.substring(i,i+1)).equals(S.pop()))
        return false;
      i++;
    }

    // A stack está vazia mas o fim da string não foi alcançada ainda
    if (i < input.length()) return false;

    // O fim da string foi alcançado mas a stack não está vazia
    else if ((i == input.length()) && (!S.isEmpty())) return false;

    else return true;
```

```
        }
    }
```

5.1. Aplicação: Infix to Postfix

Uma expressão está na forma ***infix*** se toda sub-expressão a ser avaliada está no formato **operando-operador-operando**. Por outro lado, a forma ***postfix*** é aquela em que a sub-expressão a ser avaliada está na forma **operando-operando-operador**. Estamos acostumados a avaliar expressões *infix*, porém é mais apropriado para os computadores avaliarem expressões na forma *postfix*.

Existem algumas propriedades que precisamos tomar nota neste problema:

- O **grau** de um operador é o número de operandos que ele tem.
- O ***rank*** de um operando é 1. O *rank* de um operando é 1 menos seu grau. O *rank* de uma seqüência arbitrária de operandos e operadores é as somas dos *ranks* dos operandos e operadores individuais.
- Se **z = x | y** é uma *string*, então **x** é o topo de **z**. E **x** é um próprio topo se **y** não é uma *string* nula.

Teorema: uma expressão *postfix* é bem moldada se o *rank* de todos os topos são maiores ou iguais a 1 e o *rank* da expressão é 1.

A tabela a seguir mostra a ordem de precedência dos operadores:

Operador	***Prioridade***	***Propriedade***	***Exemplo***
^	3	Associação a direita	a^b^c = a^(b^c)
* /	2	Associação a esquerda	a*b*c = (a*b)*c
+ -	1	Associação a esquerda	a+b+c = (a+b)+c

Tabela 3: Ordem de precedência

Exemplos:

Expressão Infix	***Expressão Postfix***
a * b + c / d	a b * c d / -
a ^ b ^ c - d	a b c ^ ^ d -
a * (b + (c + d) / e) - f	a b c d + e /+* f -
a * b / c + f * (g + d) / (f – h) ^ i	a b * c / f g d + * f h – i ^ / +

Tabela 4: Expressão Infix e Postfix

Regras para conversão de *infix* para *postfix*:

1. A ordem dos operandos nas duas formas são as mesmas se os parênteses estiverem ou não presentes na expressão *infix*.
2. Se a expressão *infix* não contém parênteses, então a ordem dos operadores na expressão *postfix* está de acordo com sua prioridade.
3. Se a expressão *infix* contém sub expressões em parênteses, a regra 2 se aplica do mesmo

modo para as sub-expressões.

E a seguir estão os números prioritários:

- Icp (x de ·) - número da prioridade quando o simbólico (*token*) x é um símbolo de entrada (prioridade de entrada)
- Isp (x de ·) - número de prioridade quando o simbólico (*token*) x está na *stack* (prioridade de entrada na *stack*)

Token, x	***icp(x)***	***isp(x)***	***Rank***
Operando	0	-	+1
+ -	1	2	-1
* /	3	4	-1
^	6	5	-1
(	7	0	-

Tabela 5: Números prioritários

Agora o algoritmo:

1. Pega o próximo símbolo (*token*) **x**.
2. Se x é operando então sai x
3. Se x é (, então insere x na *stack*.
4. Se x é), então retira elementos da *stack* até que (seja encontrado, mais uma vez apagar o (, Se topo = 0, o algoritmo termina.
5. Se x é um operador, então, enquanto icp(x) < isp(stack(top)), remove os elementos da *stack*; caso contrário; se icp(x) > isp(stack(top)), então insere x na *stack*.
6. Retorna ao passo 1.

Como no exemplo, vamos fazer a conversão de **a + (b * c + d) - f / g ^ h** usando a forma *postfix*:

Símbolo de entrada	***Stack***	***saída***	***Observações***
a		a	sai a
+	+	a	entra +
(	+(	a	entra (
b	+(	ab	sai b
*	+(*	ab	icp(*) > isp(**(**)
c	+(*	abc	sai c
+	+(+	abc*	icp(+) < isp(*), sai * icp(+) > isp(**(**), insere +
d	+(+	abc*d	sai d
)	+	abc*d+	sai +, sai (
-	-	abc*d++	icp(-) < isp(+), pop +, push -

f	-	abc*d++f	sai f
/	-/	abc*d++f	icp(/)>isp(-), push /
g	-/	abc*d++fg	sai g
^	-/^	abc*d++fg	icp(^)>isp(/), push ^
h	-/^	abc*d++fgh	sai h
-		abc*d++fgh^/-	retira ^, retira /, retira -

Tabela 6: Forma postfix

6. Tópicos avançados de *stacks*

6.1. Múltiplas stacks usando um array unidimensional

Duas ou mais *stack* podem coexistir em um *array* **S** comum de tamanho **n**. Nesta abordagem temos uma melhora na utilização da memória.

Se duas *stack* compartilham determinado array S, elas crescem e diminuem dentro do *array* **S**, sendo que os finais estão frente a frente separados por um intervalo definido. A figura a seguir mostra o comportamento de duas *stack* quando compartilham um mesmo *array* **S**:

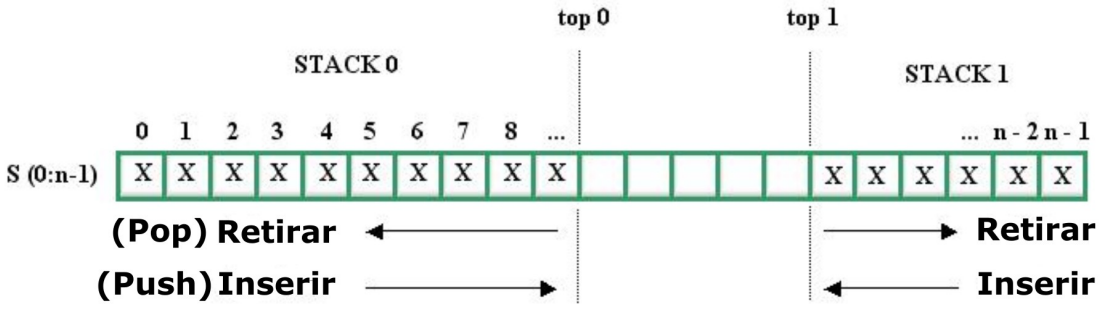

Figura 5. Duas stack coexistindo em um mesmo array

Na inicialização, a *stack* 1 está com o topo marcado como sendo -1, desta forma, top1=-1 e para a *stack* 2 será top2 = n.

6.2. Três ou mais stacks no mesmo array S:

Se três ou mais *stack* compartilham um mesmo array, elas precisarão de um indicador do endereço para os vários topos e bases destas, os apontadores de bases definem o inicio das *stack* **m** dentro do array S com tamanho igual a n, a notação disto será determinada por B[i]:

$$B[i] = \lfloor n/m \rfloor * i - 1 \qquad 0 \le i < m$$

$$B[m] = n-1$$

Os pontos B[i] determinam o espaço de uma *stack*. Para inicializar a *stack*, os topos serão marcados como sendo a ponto base da outra *stack* formando células,

$$T[i] = B[i], \qquad 0 \le i \le m$$

Por exemplo:

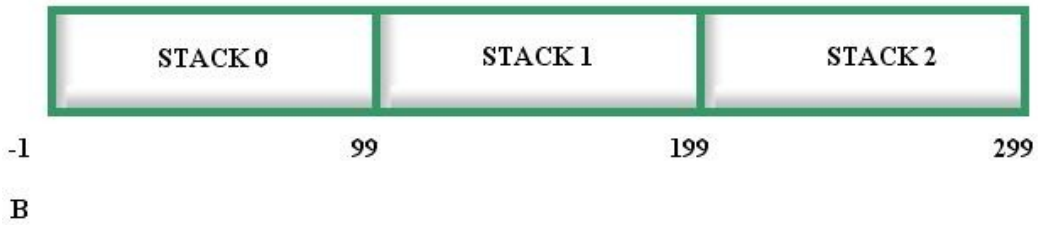

Figura 6. Três stacks no mesmo array

6.3. Três possíveis estados

O diagrama abaixo mostra as três possibilidades possíveis: *stack* vazia, cheia, cheia mas não lotada. *Stack* **i** está cheia se T[i] = B[i+1]. Não está cheia se T[i] < B[i+1] e está cheia se T[i] = B[i].

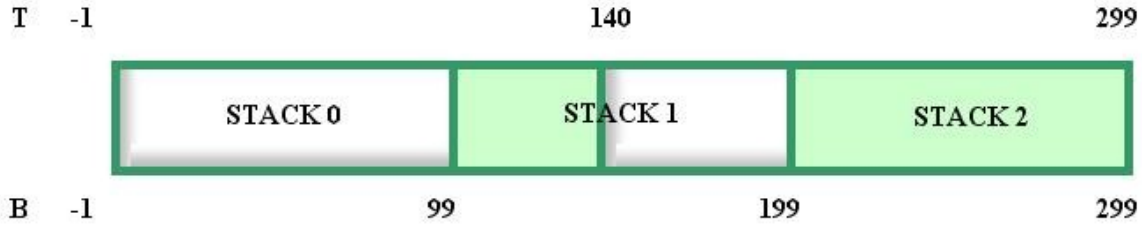

Figura 7. Três estados das stack (vazia, não vazia mas não cheia, cheia)

a parte do código Java a seguir mostra a implementação das operações de *push* e *pop* (inserir e remover) para a classe Mstack:

```
class MStack {
  int m = 3;   // número de stacks, por padrão 3
  int n = 300; // tamanho do array S
  Object[] S;
  int B[], T[], oldT[];

  //int B[] = {-1,90,210,310,400,499};
  //int T[] = {79,210,270,345,423};
  //int oldT[] = {85,195,254,360,415};

  // Construtor padrão
  public MStack() {
    S = new Object[n];
    B = new int[m+1];
    T = new int[m];
    oldT = new int[m];
  }

  // Construtor com número de stacks e tamanho
  public MStack(int numStacks, int s) {
    m = numStacks;
    n = s;
    S = new Object[n];
    B = new int[m+1];
    T = new int[m];
    oldT = new int[m];
  }

  // Returna o tamanho da stack i
  public int size(int i) {
    return (T[i]-B[i]);
  }

  // Checa se a stack está vazia
  public boolean isEmpty(int i) {
    return ((T[i]-B[i])==0);
  }

  // Returna o topo da stack i
  public Object top(int i) throws StackException {
```

```
    if (isEmpty(i)) throw new StackException("Stack empty.");
    return S[T[i]];
  }

  // inserindo elementos na stack i
  public void push(int i, Object item) {
    if (T[i]==B[i+1]) MStackFull(i);
    S[++T[i]]=item;
  }

  // retirando elementos na stack i
  public Object pop(int i) throws StackException {
    Object item;
    if (isEmpty(i))
      throw new StackException("Stack underflow.");
    item = S[T[i]];
    S[T[i]--] = null;
    return item;
  }
}
```

O método *MStackFull* captura uma possível condição de *overflow*.

```
  public void MStackFull(int i) {
    // garwicks(i);
    // unitShift(i);
  }
```

Veremos os métodos descritos a seguir.

6.4. Realocação de memória no caso de estouro (Stack Overflow)

Quando as *stack* coexistem em um mesmo *array* é possível que uma determinada *stack* fique cheia enquanto a *stack* adjacente ainda esteja vazia ou "não cheia". Neste cenário será preciso realocar memória para que seja possível disponibilizar mais espaço para a *stack* cheia. Para fazer isso procuramos a *stack* que possui endereços vazios.

Para fazer isto, procuramos nas *stacks* sobre a *stack* **i** (endereço-primário) pela mais próxima *stack* com células disponíveis, chamamos stack **k**, e então trocaremos a *stack* **i**+1 acima da *stack* **k** uma célula, até a célula esteja disponível para a *stack* **i**. Se todas as *stacks* sobre a *stack* **i** estão cheias, então procuramos as *stacks* abaixo até a mais próxima *stack* com espaço livre, chamamos de *stack* **k**, e então trocamos as células uma unidade para baixo. Isto é conhecido como o método de ***unit-shift***. Se **k** possui um valor inicial igual a **-1**, então as seguintes implementações de código para o método *unitShift* que será chamado pelo método ***MStackFull***:

```
/* capturando estouro de stack (stack overflow) usando o policiamento unidade
de troca(Unit-Shift) e retorna true se obteve sucesso, caso contrário false */

public void unitShift(int i) throws StackException {
  int k=-1; // Pontos da mais proxima stack com espaço livre

  // Procura a stack acima (endereço)
  for (int j=i+1; j<m; j++)
    if (T[j] < B[j+1]) {
      k = j;
      break;
    }
```

```
    // Troca os itens da stack k para fazer frente a stack i
    if (k > i) {
      for (int j=T[k]; j>T[i]; j--)
        S[j+1] = S[j];

      // Ajusta os pontos de topo e base
      for (int j=i+1; j<=k; j++) {
        T[j]++;
        B[j]++;
      }
    } else if (k > 0) { // Procura a stack abaixo se nenhum for achado
      for (int j=i-1; j>=0; j--)
        if (T[j] < B[j+1]) {
          k = j+1;
          break;
        }
      for (int j=B[k]; j<=T[i]; j++)
        S[j-1] = S[j];

        // Ajusta os ponteiros da base e to topo
        for (int j=i; j>k; j--) {
          T[j]--;
          B[j]--;
        }
    } else  // Não obteve sucesso, todas as stack estão cheias
      throw new StackException("Stack overflow.");
  }
```

6.5. Realocação de Memória usando o algoritmo de Garwick

O algoritmo de *Garwick* é uma maneira mais eficaz de se realocar os espaços quando a *stack* se torna cheia. Ele realoca a memória em dois passos: primeiro, um tamanho fixo de espaço é dividido entre todas as *stack*; e, segundo, o espaço restante é distribuído nas *stack* de acordo com a necessidade. A seguir vemos o algoritmo:

1. Elimina todas as células que não estão sendo usadas de todas as *stack* e considera esse espaço não-usado como sendo um espaço válido para a realocação.
2. Realoca de 1 a 10% do espaço livre válido igualmente entre as *stack*.
3. Realoca o espaço restante disponível nas *stack* em proporção com o recente crescimento, onde o recente crescimento será medido como sendo a diferença entre T[j] – oldT[j], onde oldT[j] é o valor de T[j] ao final da ultima realocação. Uma diferença negativa (positiva) significa que a *stack* j realmente foi diminuída (aumentada) em tamanho desde a última realocação.

6.6. Implementação de Knuth para o Algoritmo de Garwick

A implementação de *Knuth* organiza os espaços para que sejam distribuídos igualmente entre as *stacks* em 10%. Os 90% restantes serão particionados de acordo com o crescimento recente. O tamanho da *stack* (crescimento cumulativo) também é usado como medida de necessidade para a distribuição dos 90%. Quanto maior a *stack*, maior a quantidade de espaço que será alocada.

A seguir vemos o algoritmo:

1. Reunindo estatísticas sobre o uso da stack:

Tamanho da *stack* = T[j] - B[j]

Nota: +1 se a *stack* estiver sobrecarregada

differences = T[j] – oldT[j] if T[j] – oldT[j] >0

else 0 [a diferença negativa será substituída por 0]

Nota: +1 se a *stack* estiver sobrecarregada

freecells = tamanho total – (soma dos tamanhos)

incr = (soma das diferenças)

Nota: contador recebe +1 para a célula que sobrecarregou a *stack*.

2. Calculo da alocação de fatores

   α **= 10% * freecells / m**

   β **= 90%* freecells / incr**

   onde:

   - m= número de *stack*

   - α é o número de células que cada *stack* possui dos 10% do espaço válido alocado

   - β é o número de células que a *stack* terá por unidade incrementada do uso da stack dos 90% do espaço livre restantes.

3. Calculando o novo endereço

   σ - espaço livre teoricamente alocado para as *stack* 0, 1, 2, ..., j - 1

   τ - espaço livre teoricamente alocada para as *stack* 0, 1, 2, ..., j

   Número real total de células livres alocadas na *stack* j $= \lfloor \tau \rfloor - \lfloor \sigma \rfloor$

   inicialmente, **(new)B[0] = -1** e σ **= 0**

   for j = 1 to m-1:

   $\tau = \sigma + \alpha +$ **diff[j-1]*** β

   B[j] = B[j-1] + size[j-1] + $\lfloor \tau \rfloor - \lfloor \sigma \rfloor$

   $\sigma = \tau$

4. Trocando as *stack* para suas novas coordenadas.

5. Atribuindo oldT = T

Considere o seguinte exemplo. 5 *stack* coexistindo num vector de 500 posições. O estado das *stacks* é mostrado na figura abaixo:

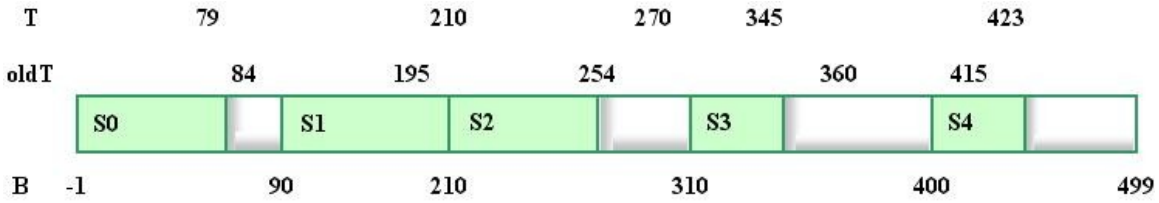

Figura 8. Estado das stack antes da re-alocação

1. Reunindo estatísticas sobre o uso da *stack*

   Tamanho das *stack* = T[j] - B[j]

   Nota: +1 se a *stack* estiver sobrecarregada

   Diferenças = T[j] – OLDT[j] if T[j] – OLDT[j] >0

   else 0 [a diferença negativa é substituída por 0]

   Nota: +1 se a *stack* estiver sobrecarregada

   freecells = tamanho total – (soma dos tamanhos)

   incr = (soma das diferenças)

fator	***Valor***
Tamanho das stack	Tamanho = (80, 120***+1***, 60, 35, 23)
Diferenças	Diferença = (0, 15***+1***, 16, 0, 8)
freecells	500 - (80 + 120***+1*** **+** 60 + 35 + 23) = **181**
incr	0 + 15+*1* + 16 + 0 + 8 = **40**

Tabela 7: Fator e valor

2. Calculando a alocação dos fatores

α = 10% * freecells / m = 0.10 * 181 / 5 = **3.62**

β = 90%* freecells / incr = 0.90 * 181 / 40 = **4.0725**

3. Calcula o novo endereço

   B[0] = -1 and σ **= 0**

   for j = 1 to m:

   τ **=** $\sigma + \alpha$ **+ diff(j-1)***β

   B[j] = B[j-1] + size[j-1] + $\lfloor \tau \rfloor - \lfloor \sigma \rfloor$

   $\sigma = \tau$

   j = 1: τ = 0 + 3.62 + (0*4.0725) = 3.62

   B[1] = **B**[0] + size[0] + $\lfloor \tau \rfloor - \lfloor \sigma \rfloor$

   = -1 + 80 + $\lfloor 3.62 \rfloor - \lfloor 0 \rfloor$ = **82**

$\sigma = 3.62$

j = 2: $\tau = 3.62 + 3.62 + (16*4.0725) = 72.4$

B[2] = **B**[1] + size[1] + $\lfloor\tau\rfloor - \lfloor\sigma\rfloor$

$= 82 + 121 + \lfloor 72.4\rfloor - \lfloor 3.62\rfloor =$ **272**

$\sigma = 72.4$

j = 3: $\tau = 72.4 + 3.62 + (16*4.0725) = 141.18$

B[3] = **B**[2] + size[2] + $\lfloor\tau\rfloor - \lfloor\sigma\rfloor$

$= 272 + 60 + \lfloor 141.18\rfloor - \lfloor 72.4\rfloor =$ **401**

$\sigma = 141.18$

j = 4: $\tau = 141.18 + 3.62 + (0*4.0725) = 144.8$

B[4] = **B**[3] + size[3] + $\lfloor\tau\rfloor - \lfloor\sigma\rfloor$

$= 401 + 35 + \lfloor 144.8\rfloor - \lfloor 141.18\rfloor =$ **439**

$\sigma = 144.8$

Para checar, NEWB(5) deverá ser igual a 499:

j = 5: $\tau = 144.8 + 3.62 + (8*4.0725) = 181$

B[5] = **B**[4] + size[4] + $\lfloor\tau\rfloor - \lfloor\sigma\rfloor$

$= 439 + 23 + \lfloor 181\rfloor - \lfloor 144.8\rfloor =$ **499** [OK]

4. Troca as stack para suas novas coordenadas.

B = (-1, 82, 272, 401, 439, 499)

T[i] = B[i] + size [i] ==> T = (0+80, 83+121, 273+60, 402+35, 440+23)

T = (80, 204, 333, 437, 463)

oldT = T = (80, 204, 333, 437, 463)

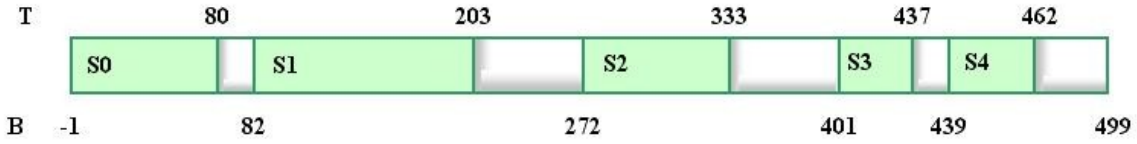

Figura 9. Estado das stacks após uma realocação

Existem algumas técnicas para melhorar a utilização das *stack*. Primeiro, saiba qual *stack* é a maior, faça isso primeiro. Segundo, o algoritmo pode emitir um comando de parada quando o espaço livre se tornar menor que um valor mínimo especificado que não 0, quer dizer um *minfree* (mínimo livre), onde o usuário possa especificar esse valor mínimo.

Além de *Stacks*, o algoritmo pode ser capacitado para realocar espaço para mais dados em outros

tipos de estruturas (por exemplo *queue*, tabelas seqüenciais ou a combinação destas).

O método a seguir implementa um algoritmo *Garwick's* que será chamado pelo método ***MStackFull***:

```
// Método Garwick's
public void garwicks(int i) throws StackException {
  int diff[] = new int[m];
  int size[] = new int[m];
  int totalSize = 0;
  double freecells, incr = 0;
  double alpha, beta, sigma=0, tau=0;

  // calculo de fatores de distribuição
  for (int j=0; j<m; j++) {
    size[j] = T[j]-B[j];
    if ((T[j]-oldT[j]) > 0)
      diff[j] = T[j]-oldT[j];
    else
      diff[j] = 0;
    totalSize += size[j];
    incr += diff[j];
  }

  diff[i]++;
  size[i]++;
  totalSize++;
  incr++;
  freecells = n - totalSize;
  alpha = 0.10 * freecells / m;
  beta  = 0.90 * freecells / incr;

  // se todas as stack estiverem cheias
  if (freecells < 1)
    throw new StackException("Stack overflow.");

  // cálculo para novas bases
  for (int j=1; j<m; j++) {
    tau = sigma + alpha + diff[j-1] * beta;
    B[j] = B[j-1] + size[j-1] + (int)Math.floor(tau) - (int)Math.floor(sigma);
    sigma = tau;
  }

  // Restabeleça tamanho da stack que teve overflowed para o seu antigo valor
  size[i]--;

  // cálculo para o novo endereço de topo
  for (int j=0; j<m; j++)
    T[j] = B[j] + size[j];
  oldT = T;
}
```

Para testar esta classe insira o seguinte método principal:

```
  public static void main(String args[]) {
    MStack stack = new MStack(5, 500);
    System.out.println(stack.n);
    for (int i=0; i<stack.m; i++){
      System.out.println("B: " + stack.B[i] +
```

```
            "  T: " + stack.T[i] + "  oldT: " + stack.oldT[i]);
        }
        stack.push(1, "a");
        for (int i=0; i<stack.m; i++){
          System.out.println("B: " + stack.B[i] +
            "  T: " + stack.T[i] + "  oldT: " + stack.oldT[i]);
        }
      }
```

7. Exercícios

1. Converta as expressões seguintes da forma ***infix*** para ***postfix*** e mostre a stack resultante.

```
a) a+(b*c+d)-f/g^h
b) 1/2-5*7^3*(8+11)/4+2
```

2. Converta as expressões seguintes para a forma ***postfix***:

```
a) a+b/c*d*(e+f)-g/h
b) (a-b)*c/d^e*f^(g+h)-i
c) 4^(2+1)/5*6-(3+7/4)*8-2
d) (m+n)/o*p^q^r*(s/t+u)-v
```

3. Estratégias de realocação para *overflow* de *stack* para os números 3 e 4:

a) Desenhe um diagrama mostrando o estado atual da *stack*.

b) Desenhe um diagrama mostrando o estado da *stack* depois da implementação do *unit-shift policy*.

c) Desenhe um diagrama mostrando o estado da *stack* depois de usar o algoritmo *Garwick's* mostrando como as novas bases foram computadas.

4. Cinco *stack* coexistindo em um array de 500 posições. Uma inserção é tentada na *stack* 2. O estado de computação está definido por:

```
OLDT(0:4) = (89, 178, 249, 365, 425)
T(0:4) = (80, 220, 285, 334, 433)
B(0:5) = (-1, 99, 220, 315, 410, 499)
```

5. Três stack coexistem em um array de tamanho 300. Uma inserção é tentada na *stack* 3. O estado de computação está definido por:

```
OLDT(0:2) = (65, 180, 245)
T(0:2) = (80, 140, 299)
B(0:3) = (-1, 101, 215, 299)
```

7.1. Exercícios para Programar

1. Escreva uma classe de Java que verifica se os parênteses e chaves estão equilibrados em uma expressão aritmética.

2. Crie uma classe Java que implementa a conversão bem-formada de uma expressão *infix* e seu *postfix* equivalente.

3. Implemente a conversão de *infix* para *postfix* usando uma implementação encadeada de *stack*. A classe solicitará uma entrada do usuário e verificará se a entrada está correta. Mostre a produção e conteúdo da stack em toda interação.

4. Crie uma definição de classe Java de uma *stack* múltipla em um array dimensional. Implemente as operações básicas em *stack* (*push* e *pop*, etc) para ser aplicável em múltipla *Stack*. O nome da classe será *MStack*.

5. Uma loja de livro tem estantes com divisórias ajustáveis. Quando uma divisória fica cheia, a divisória será ajustada para obter mais espaço. Crie uma classe Java que relocará o espaço da estante usando o algoritmo de *Garwick*.

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.