Módulo 3

Estruturas de Dados

Lição 9

Árvores de Pesquisa Binária

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

Outras aplicações das árvores binárias (ADT[1]) são a **pesquisa** e a **classificação** através da aplicação de certas regras aos valores dos elementos armazenados em uma árvore binária. Considere como exemplo a árvore binária apresentada.

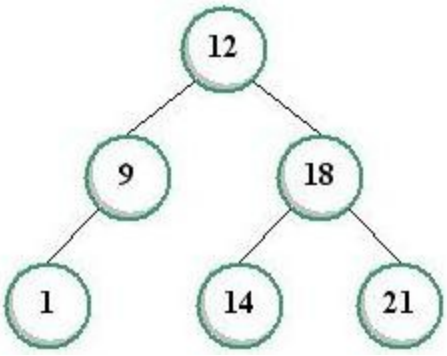

Figura 1. Árvore Binária (Binary Tree)

O valor de cada *node* na árvore é maior que o valor do *node* à sua esquerda (se existir) e é menor que o valor do *node* à sua direita (se existir). Em árvores de Pesquisa binária, esta propriedade sempre deverá ser satisfeita.

Ao final desta lição, o estudante será capaz de:

- Discutir as **propriedades de uma árvore de pesquisa binária**
- Realizar as **operações** em árvores de pesquisa binária
- Aprimorar a pesquisa, inserção e remoção em árvores de Pesquisa binária mantendo o **balanceamento** utilizando árvores **AVL**

[1] ADT – sigla em inglês Abstract Data Type, Tipo de Dado Abstrato

2. Operações em Árvores de Pesquisa Binária

As operações mais comuns em uma Árvores de Pesquisa Binária (*Binary Search Tree* ou **BST**) são as inserções de novas chaves, remoção de uma chave existente, pesquisa por uma chave e recuperação de dados numa ordem de classificação. Nesta seção, as três primeiras operações serão apresentadas. A quarta operação poderá ser realizada pela leitura da **BST** seguindo uma ordem.

Nas três operações, assume-se que a árvore de pesquisa binária não está vazia e que não armazena valores duplicados. Além disso, é utilizada a estrutura de *node* BSTNode (Esquerda, Info, Direita), como ilustrado abaixo:

Figura 2. BSTNode

Em Java,

```
class BSTNode {
      int info;
      BSTNode left, right;

      public BSTNode(){
      }
      public BSTNode(int i) {
            info = i;
      }
      public BSTNode(int i, BSTNode l, BSTNode r) {
            info = i;
            left = l;
            right = r;
      }
}
```

A árvore de pesquisa binária utilizada nesta lição tem uma cabeça de lista (list head) como ilustrado na figura abaixo:

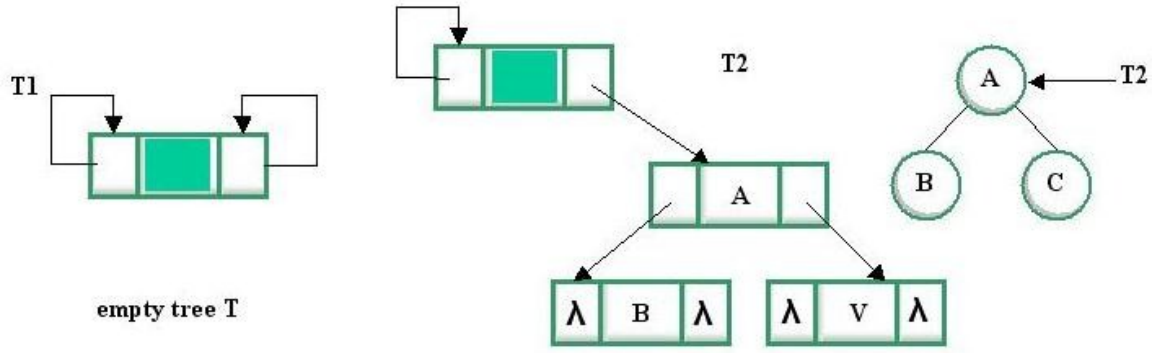

Figura 3. Representação da Árvore de Pesquisa Binária T2

Se a árvore de pesquisa binária está vazia, os ponteiros da esquerda e da direita do *header* da lista apontam para si mesmos; senão, o ponteiro da direita apontará para a raiz da árvore. Na sequência está a definição da classe de uma **BST** usando esta estrutura:

```
public class BST {
```

```
  BSTNode bstHead = new BSTNode();

  // Criar uma BST vazia
  public BST() {
    bstHead.left = bstHead;
    bstHead.right = bstHead;
  }
  // Criar uma BST com raiz r, ponteiro para bstHead.right
  public BST(BSTNode r) {
    bstHead.left = bstHead;
    bstHead.right = r;
  }
}
```

2.1. Pesquisando

Na Pesquisa por um valor, digamos **k**, três condições são possíveis:

- k = valor que está no *node* (pesquisa com sucesso)
- k < valor que está no *node* (pesquisa pela subárvore da esquerda)
- k > valor que está no *node* (pesquisa pela subárvore da direita)

O mesmo processo é repetido até que uma correspondência seja encontrada ou que se atinja um *node* folha. Neste caso, a pesquisa não teve sucesso.

A seguir está a implementação Java para o algoritmo acima:

```
// Pesquisa por k, retorna o node que contém k se encontrado
public BSTNode search(int k) {
  BSTNode p = bstHead.right;  // node raiz

  // Se a árvore está vazia, retorna null
  if (p == bstHead)
    return null;

  // Compare
  while (true) {
    if (k == p.info)
      return p;            // sucesso na Pesquisa
    else if (k < p.info)   // vá pela esquerda
      if (p.left != null)
        p = p.left;
      else
        return null;       // não encontrou
    else                   // vá pela direita
      if (p.right != null)
        p = p.right;
      else
        return null;       // não encontrou
  }
}
```

2.2. Inserção

Na inserção de um valor na árvore será realizada uma pesquisa para encontrar o local apropriado para o novo valor. A seguir está o algoritmo:

1. Comece a Pesquisa no node da raiz. Declare um *node* p e faça-o apontar para a raiz.
2. Faça a comparação:

```
if (k == p.info) return false       // se encontrou a chave, não permite inserção
else if (k < p.info) p = p.left     // pela esquerda
else p = p.right                    // se (k > p.info) pela direita
```

3. Insira o *node* (p agora aponta para o novo pai do node para inserir):

```
newNode.info = k
newNode.left = null
newNode.right = null
if (k < p.info) p.left = newNode
else p.right = newNode
```

Em Java,

```
// Insere k na árvore de pesquisa binária
public boolean insert(int k) {
  BSTNode p = bstHead.right;  // node raiz
  BSTNode newNode = new BSTNode();

  // Se a árvore está vazia, torna o novo node raiz
  if (p == bstHead) {
    newNode.info = k;
    bstHead.right = newNode;
    return true;
  }

  // Procura o local certo para inserir k
  while (true) {
    if (k == p.info)          // chave já existe
      return false;
    else if (k < p.info)      // pela esquerda
      if (p.left != null)
        p = p.left;
      else
        break;
    else if (p.right != null) // pela direita
       p = p.right;
    else
      break;
  }

  // Insere a nova chave no local apropriado
  newNode.info = k;
  if (k < p.info)
    p.left = newNode;
  else
    p.right = newNode;
  return true;
}
```

2.3. Eliminação

Eliminar uma chave da árvore de pesquisa binária é um pouco mais complexo que as outras duas operações discutidas. A operação inicia por encontrar a chave para deletar. Se não encontrar, o algoritmo simplesmente retorna dizendo que a exclusão falhou. Se a Pesquisa retornar uma chave encontrada, então existe necessidade de eliminar o *node* que contém a chave que procuramos.

Entretanto, excluir não é tão simples como remover um *node* encontrado que tenha um parente apontando pra ele. Existe também a possibilidade que ele seja o parente de alguns outros *nodes* na **BST**. Neste caso, existe a necessidade desses filhos serem "adotados" por outros *nodes*, e também ajustar os ponteiros que apontavam para o *node* removido. E, no processo de atribuir apontadores, a propriedade **BST** da ordem dos valores das chaves deve ser mantida.

Existem dois casos gerais para se considerar na exclusão de um *node* **d**:

1. *node* **d** é externo (folha):

Ação: Atualize o ponteiro filho do parente p:
Se d é um filho da esquerda, ajuste p.left=null
do contrário, ajuste p.right=null

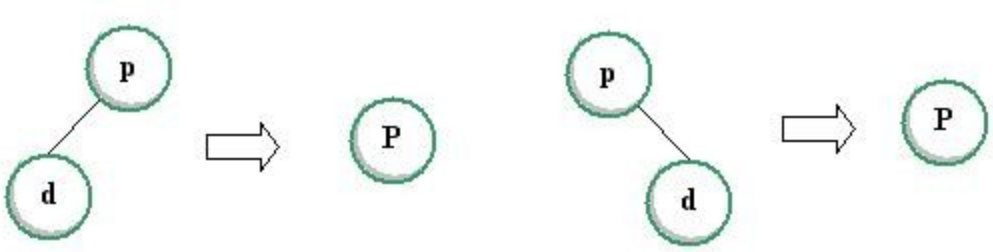

Figura 4. Eliminar um node folha

2. *node* **d** é interno (Existem dois sub-casos):

Caso 1: Se **d** tem uma subárvore à esquerda,

1. Obtenha o *node* predecessor **ip**. *Node* predecessor é definido como o *node* mais à direita na subárvore a esquerda do *node* corrente, o qual neste caso é **d**. Ele contém a chave precedente se o BST é atravessado *node*
2. Obtenha o parente de **ip**, diga **p_ip**
3. Substitua d com **ip**
4. Remova **ip** de sua antiga localização para ajustar os apontadores:
   a. Se **ip** não é uma folha *node*:
      1. Ajuste o filho da direita de **p_ip** para apontar para o filho da esquerda de **ip**
      2. Ajuste o filho da esquerda de **ip** para apontar para o filho da esquerda de **d**
   b. Ajuste o filho da direita de **ip** para apontar para o filho da direita de **d**

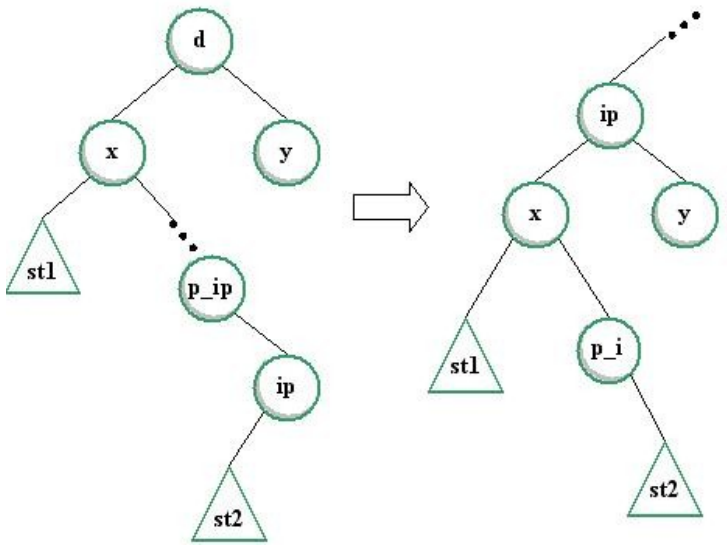

Figura 5. Excluindo um node interno com uma subárvore à esquerda

Caso 2: Se d não tem filho a esquerda, mas possui uma subárvore à direita.

1. Obtenha o sucessor inorder, **is**. O sucessor Inorder é definido como o node mais à esquerda na subárvore da direita do node atual. Ele contém a próxima chave se a APB estiver sendo varrida em ordem direta.
2. Obtenha o pai de **is**, digamos **p_is**.
3. Substitua **d** por **is**.
4. Remova **is** de sua antiga locação pelo ajuste dos ponteiros:
   a. Se **is** não é um node folha:
      1. Ajuste o filho da esquerda de **p_is** para apontar para o filho da direita de **is**
      2. Ajuste o filho da direita de **is** para apontar para o filho da direita de **d**
   b. Ajuste o filho da esquerda de **is** para apontar para o filho da esquerda de **d**

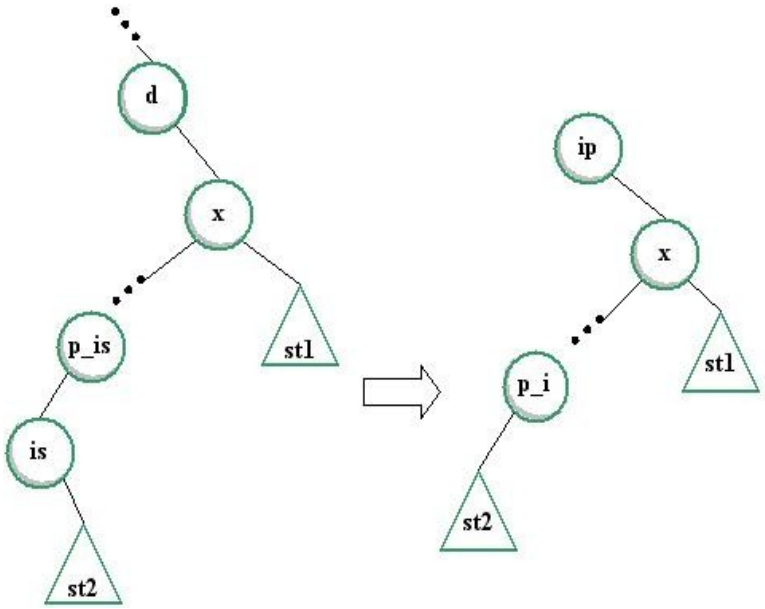

Figura 6. Remoção de um node interno sem subárvore à esquerda

O seguinte código Java implementa este procedimento:

```
// Retorna true se a chave foi removida com sucesso
public boolean delete(int k) {
  BSTNode delNode = bstHead.right;  // o node raiz
  boolean toLeft = false;           // direção do pai
  BSTNode parent = bstHead;         // pai do node removido

  // Pesquisa do node a remover
  while (true) {
    // delNode aponta para o node que será removido
    if (k == delNode.info)
      break;
    else if (k < delNode.info)         // Vá pela esquerda
      if (delNode.left != null) {
        toLeft = true;
        parent = delNode;
        delNode = delNode.left;
      } else
        return false;                  // não encontrado
    else if (delNode.right != null) { // Vá pela direita
        toLeft = false;
        parent = delNode;
        delNode = delNode.right;
    } else
      return false;                    // não encontrado
  }

  // Caso 1: Se delNode está na extremidade, atualiza o pai e remove o node
  if ((delNode.left == null) && (delNode.right == null)) {
    if (toLeft)
      parent.left = null;
    else
      parent.right = null;
  }

  // Case 2.1: Se delNode é interno e tem filho a esquerda
  else if (delNode.left != null) {
    BSTNode inPre = delNode.left;    // predecessor inorder
    BSTNode inPreParent = null;      // pai do predecessor inorder

    // Procura pelo sucessor inorder de delNode
    while (inPre.right != null) {
      inPreParent = inPre;
      inPre = inPre.right;
    }

    // Substitui delNode por inPre
    if (toLeft)
      parent.left = inPre;
    else
      parent.right = inPre;

    // Remove inSuc de seu local de origem

    // Se inPre não é um node folha
    if (inPreParent != null) {
```

```
            inPreParent.right = inPre.left;
            inPre.left = delNode.left;
          }
          inPre.right = delNode.right;
        }

        // Case 2.2: Se delNode é interno e não tem filho a esquerda, mas a direita
        else {
          BSTNode inSuc = delNode.right; // successor inorder
          BSTNode inSucParent = null;    // pai do sucessor inorder

          // Procura o sucessor inorder de delNode
          while (inSuc.left != null) {
            inSucParent = inSuc;
            inSuc = inSuc.left;
          }

          // Substitui delNode por inSuc
          if (toLeft)
            parent.left = inSuc;
          else
            parent.right = inSuc;

          // Remove inSuc de seu local de origem

          // Se inSuc não é um node folha
          if (inSucParent != null) {
            inSucParent.left = inSuc.right;
            inSuc.right = delNode.right;
          }
          inSuc.left = delNode.left;
        }
        delNode = null; // limpeza da memória (garbage collection)
        return true;    // retorna sucesso
      }
```

2.4. Complexidade no tempo de resposta em uma BST

As três operações discutidas fazem pesquisas pelo *node* que contém uma chave determinada. Por isso, concentraremos a atenção em relação a pesquisa que possui o melhor desempenho. O algoritmo de pesquisa em uma **BST** gasta $O(\log_2 n)$ vezes em média, quando a árvore de pesquisa binária está balanceada, quando a altura está em $O(\log_2 n)$. Entretanto, este nem sempre é o caso num processo de pesquisa. Considere, por exemplo, o caso onde os elementos inseridos na **BST** estão ordenados (ou em ordem inversa), então a **BST** resultante terá todo seus *nodes* da esquerda (ou direita) apontando para ***NULL***, resultando numa árvore degenerada. Neste caso, o tempo de pesquisa deteriora para O(n), equivalendo a uma pesquisa seqüencial, como no caso da árvore seguinte:

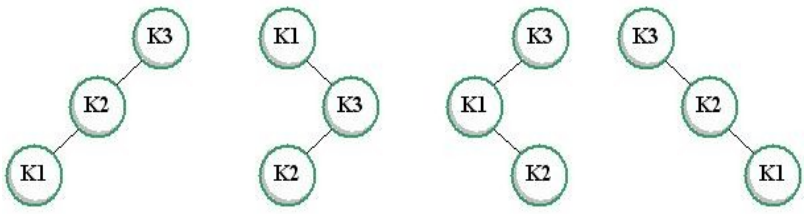

Figura 7. Exemplos de Árvores com pior caso de pesquisa

3. Árvore Binária de Pesquisa Balanceada

É o balanceamento pobre de uma Árvore Binária a faz ter uma performance de O(n). Por isso, deve-se assegurar de que a Pesquisa gaste $O(\log_2 n)$, de modo que o balanceamento possa ser mantido. Uma árvore balanceada só será criada se metade dos registros inserida depois de um dado registro r com chave k tenha chaves menores do que k e, similarmente, metade das chaves maiores do que k. Isso quando não há tratamento de balanceamento durante a inserção. No entanto, em situações reais, as chaves são inseridas em ordem aleatória. Portanto, há a necessidade de manter o balanceamento à medida que chaves são inseridas ou apagadas.

O **Balanceamento de um *node*** é um fator muito importante na manutenção de balanceamento. Ele é definido como a diferença de altura das subárvores de um *node*, i.e., a altura da subárvore à esquerda menos a altura da subárvore à direita.

3.1. Árvore AVL

Uma das árvores de Pesquisa binárias mais comumente utilizadas é a árvore AVL. Foi criada por *G.* ***A****del'son-****V****el'skii* e *E.* ***L****andis*, de onde advém o nome AVL. Uma árvore AVL é balanceada quando a diferença de altura entre as subárvores de um *node*, para cada *node* da árvore, é no máximo 1. Considere as árvores abaixo onde os *nodes* estão identificados com os fatores de balanceamento:

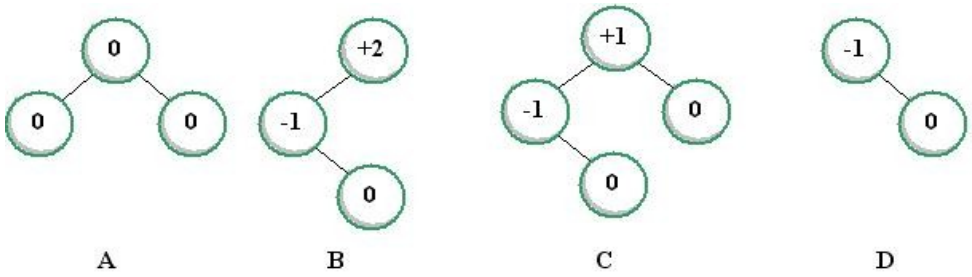

Figura 8. Árvores de Pesquisa Binária com Fatores de Balanceamento

As árvores binárias **A**, **C** e **D** todas têm *nodes* com balanceamentos na faixa [-1, 1], i.e., -1, 0 e +1. Portanto, são árvores AVL. A árvore **B** tem um node com balanceamento +2, e está além da faixa, não sendo uma árvore AVL.

Além de ter uma complexidade temporal $O(\log_2 n)$ para Pesquisas, as seguintes operações terão a mesma complexidade se a árvore AVL for usada:

- Encontrar o nº item, dado n
- Inserir um item em um lugar específico
- Excluir um item específico

3.1.1. Balanceamento da Árvore

As seguintes árvores são exemplos de árvores AVL:

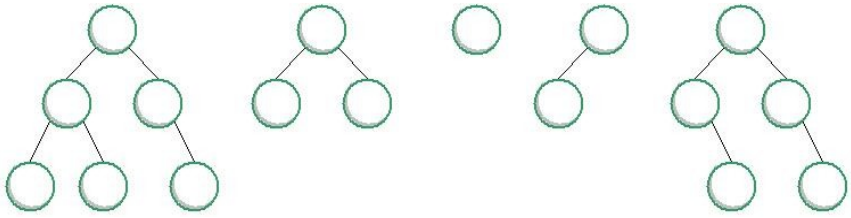

Figura 9. Árvores AVL

As seguintes árvores são exemplos de árvores não-AVL:

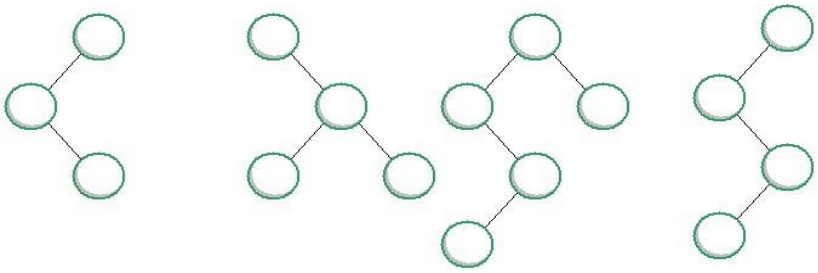

Figura 10. Árvores não-AVL

Para manter o balanceamento de uma árvore AVL, as rotações devem ocorrer durante a inserção e a exclusão. As seguintes rotações são usadas:

- Rotação simples à direita (*Simple right rotation* - RR) – Usada quando o novo item **C** esta na árvore à esquerda do *node* filho **B** do ancestral mais próximo **A** com fator de balanceamento +2

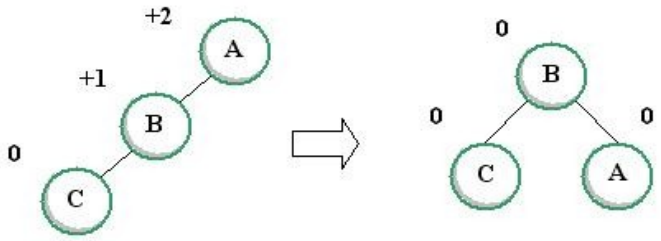

Figura 11. Rotação Simples à Direita

- Rotação simples à esquerda (*Simple left rotation* - LR) – Utilizada quando o novo item **C** está na subárvores à direita do *node* filho **B** do ancestral mais próximo **A** com fator de balanceamento -2.

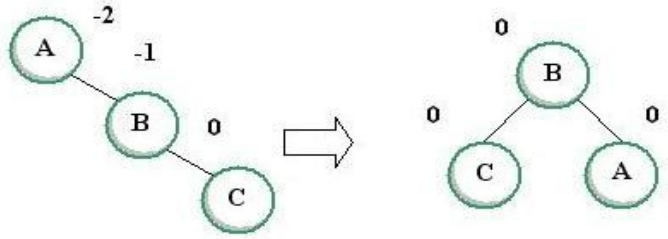

Figura 12. Rotação simples para esquerda

- Rotação esquerda direita (*Left right rotation* - LRR) – utilizado quando o novo item **C** está na subárvore da direita do filho a esquerda do ancestral mais próximo **A** com fator de balanceamento +2.

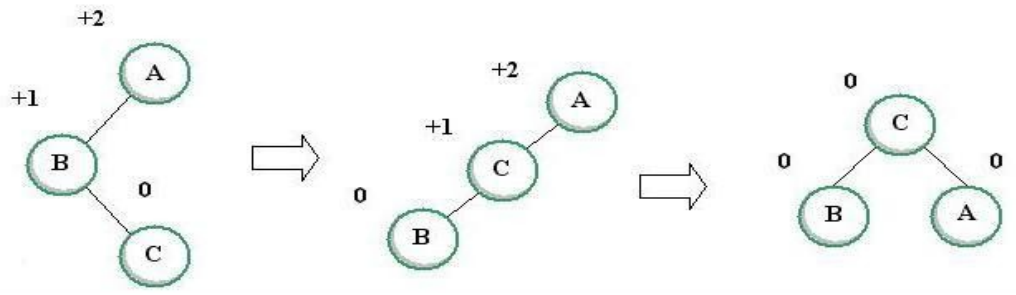

Figura 13. Rotação Esquerda Direita

- Rotação direita esquerda (*Right left rotation* - RLR) – usado quando o novo item **C** estiver na subárvore à esquerda do filho direita **B** do antecessor mais próximo **A**, com fator de balanceamento -2.

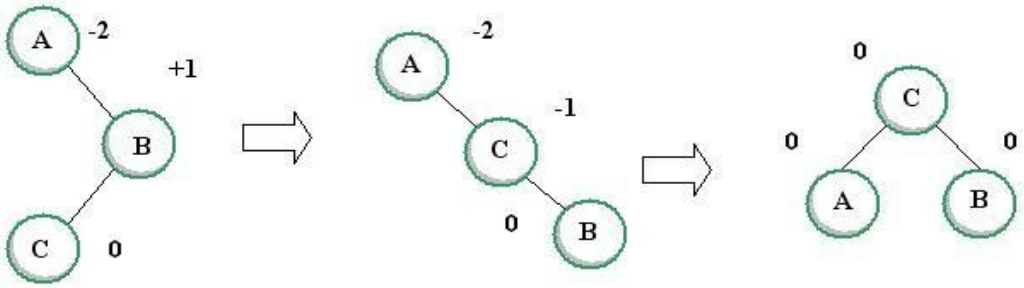

Figura 14. Rotação Direita Esquerda

A **inserção** em uma árvore de AVL e o mesmo que na **BST**. Entretanto, o equilíbrio do resultado tem que ser verificado dentro da árvore de AVL. Para introduzir um novo valor:

1. Uma folha do *node* é introduzido na árvore com equilíbrio 0;
2. Começando pelo *node* novo, uma mensagem de altura da subárvore que contém o *node* novo incrementa por 1 é passada acima da árvore seguindo o trajeto até a raiz. Se a mensagem for recebida por um *node* de sua subárvore à esquerda, 1 é adicionado a seu equilíbrio, caso contrário -1 é adicionado. Se o equilíbrio resultante for +2 ou -2, a rotação tem que ser executada como descrita.

Por exemplo, introduzir os seguintes elementos em uma árvore de AVL:

4	10	9	3	11	8	2	1	5	7	6

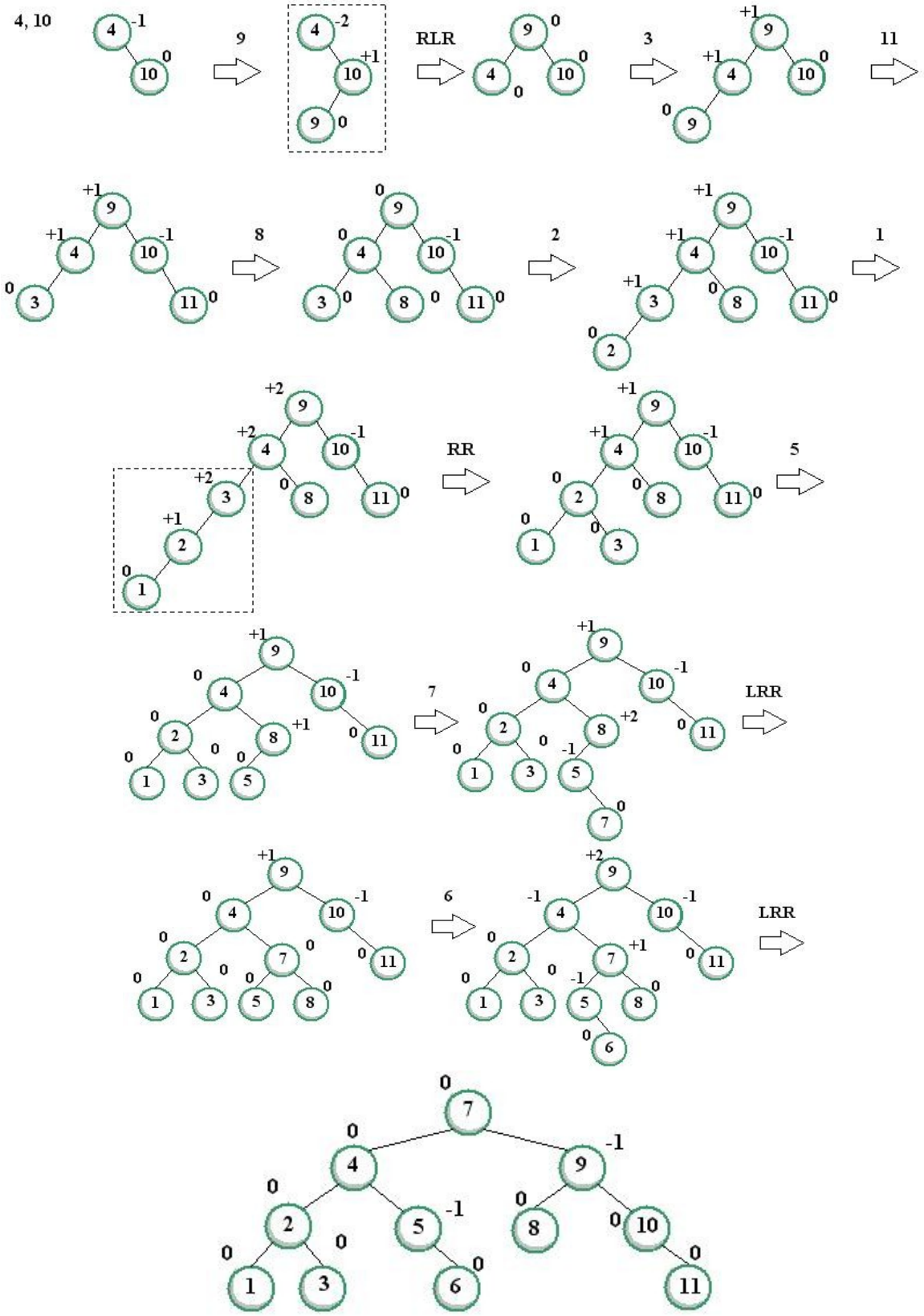

Figura 15. Inserção na árvore AVL

4. Exercícios

1. Insira as seguintes chaves em uma árvore AVL, baseada na ordem informada:

   a) 1 6 7 4 2 5 8 9 0 3

   b) A R F G E Y B X C S T I O P L V

4.1. Exercícios para programar

1. Estenda a classe BST definindo a construção de uma árvore AVL, i.e.,

```
class AVL extends BST{
}
```

Realize um override nos métodos *insert* e *delete* para implementar as rotações para a árvore AVL para manter o balanceamento.

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.