Módulo 3

Estruturas de Dados

Lição 4

Árvores Binárias

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 árvore binária é um tipo de dado abstrato que é estruturalmente hierárquico. É uma coleção de *nodes* que pode estar vazia ou pode consistir de uma raiz e duas árvores binárias distintas chamadas de sub-árvores à esquerda e à direita. É semelhante a uma árvore no sentido de que existe o conceito de uma raiz, galhos e folhas. Entretanto, diferem na orientação já que a raiz de uma árvore binária está no topo como primeiro elemento, ao contrário do que ocorre com uma árvore real na qual a raiz localiza-se no final da árvore como último elemento.

Árvores Binárias são mais utilizadas em pesquisa, classificação, localização eficiente em cadeias de caracteres, listas de prioridades, tabelas de decisão e tabelas de símbolos.

Ao final desta lição, o estudante será capaz de:

- Explicar os conceitos básicos e definições relacionadas a árvores binárias
- Identificar as propriedades de uma árvore binária
- Enumerar os diferentes tipos de árvores binárias
- Discutir como as árvores binárias são representadas na memória dos computadores
- Percorrer árvores binárias usando três algoritmos de varredura: pré-ordem, em ordem, pós-ordem
- Discutir aplicações da varredura em árvores binárias
- Usar *heaps* e o algoritmo *heapsort* para classificar um conjunto de elementos

2. Definições e Conceitos Relacionados

Uma árvore binária **T** tem um *node* especial, chamado **r**, que é o *node* **raiz**. Cada *node* **v** de **T**, que é diferente de **r**, possui um *node* **pai p**. O *node* **v** é chamado de ***child*** (ou **filho**) do *node* **p**. Um *node* pode ter no mínimo zero (0) e no máximo dois (2) children que são classificados como ***child* esquerdo** ou ***child* direito**. As **sub-árvores** de T cuja raiz é **v** são consideradas filhas de **v**. É uma **sub-árvore à esquerda** se for o child esquerdo do *node* **v** ou uma **sub-árvore à direita** se estiver ligada ao *child* direito do *node* **v**. O **grau** de um *node* é o número de sub-árvores não-nulas deste *node*. Se um *node* tiver grau zero, ele é classificado como **folha** ou um *node* **terminal.**

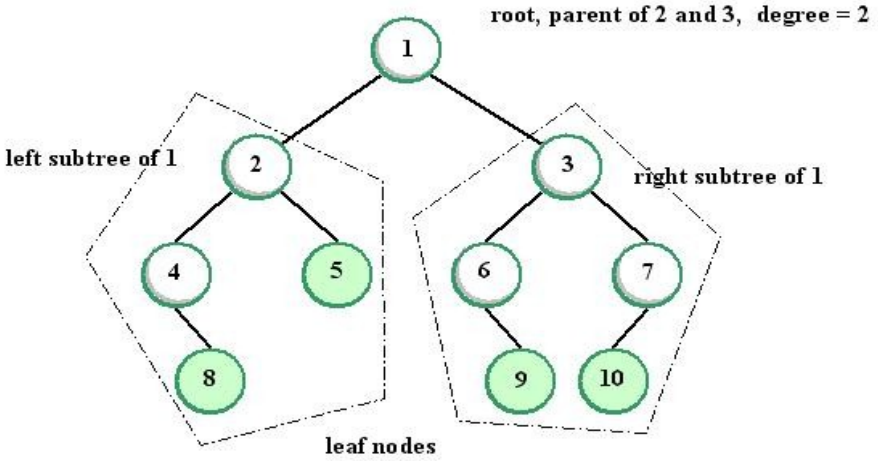


Figura 1: Uma Árvore Binária

O **nível** de um *node* se refere à distância do *node* à raiz. Portanto, a raiz da árvore tem nível 0, as suas sub-árvores têm nível 1 e assim por diante. A **altura** ou **profundidade** de uma árvore é o nível dos seus *node*s mais inferiores, que também é o tamanho do maior caminho da raiz para qualquer folha. Por exemplo, a árvore binária a seguir possui altura 3:

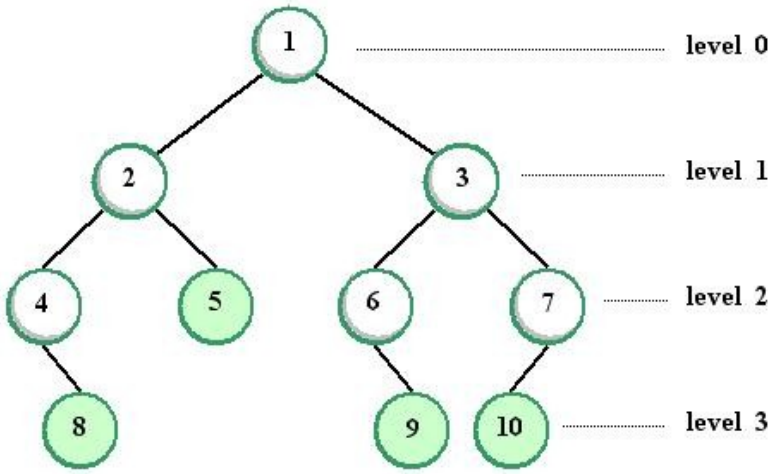


Figura 2: Níveis de uma Árvore Binária

Um *node* é **externo**, se não tiver *children*, caso contrário ele é **interno**. Se dois *node*s tiverem o mesmo pai, são **irmãos**. O **ancestral** de um *node* é ele próprio ou um ancestral de seu pai. Inversamente, o *node* **u** é um **descendente** do *node* **v** se **v** é um ancestral do *node* **u**.

Uma árvore binária pode estar **vazia**. Se a árvore binária tiver zero ou dois children, é classificada

como uma **árvore binária equilibrada ou balanceada**. Deste modo, cada árvore binária equilibrada possui *nodes* internos com dois children ou nenhum.

A figura abaixo mostra os diferentes tipos de árvores binárias: (a) mostra uma árvore binária vazia; (b) mostra uma árvore binária com apenas um *node*, a raiz; (c) e (d) mostram árvores sem children à direita e à esquerda respectivamente; (e) mostra uma árvore binária inclinada à esquerda enquanto (f) mostra uma árvore binária completa.

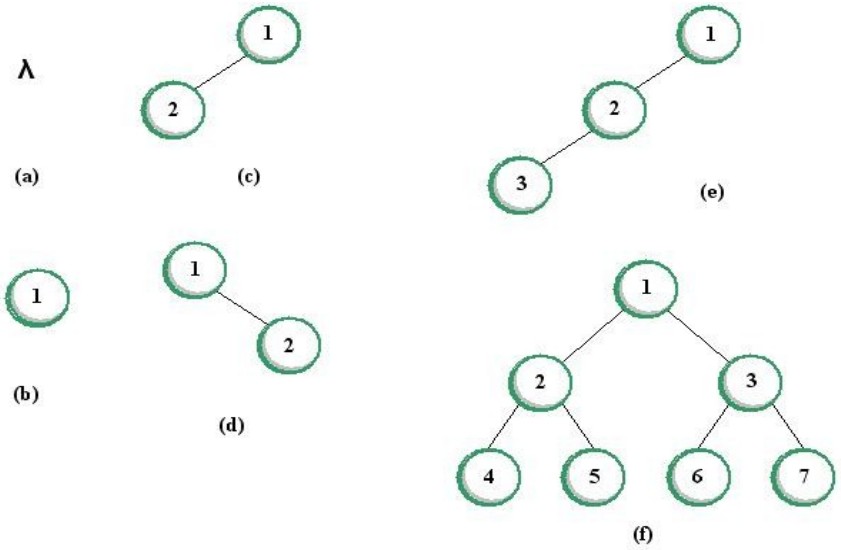


Figura 3: Diferentes Tipos de Árvore Binária

2.1. Propriedades de uma Árvore Binária

Para uma árvore binária (equilibrada ou balanceada) de profundidade k,

- O número máximo de *nodes* no nível i é 2^i , i ≥ 0.
- O número de *nodes* é no mínimo 2k + 1 e no máximo $2^{k+1} - 1$.
- O número de *nodes* externos é no mínimo h+1 e no máximo 2^k.
- O número de *nodes* internos é no mínimo h e no máximo $2^k - 1$.
- Se n_o é o número de *nodes* folhas e n_2 é o número de *nodes* de grau 2 numa árvore binária, então $n_o = n_2 + 1$.

2.2. Tipos de Árvores Binárias

Uma árvore binária pode ser classificada como degenerada, estritamente binária, cheia ou completa.

Uma árvore binária **degenerada à direita (esquerda)** é uma árvore em que os *nodes* não têm sub-árvores à esquerda (direita). Dado um número de *nodes*, uma árvore binária degenerada à esquerda ou à direita tem profundidade máxima.

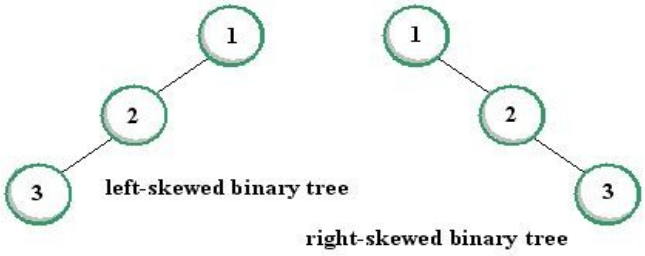


Figura 4: Árvores Binárias Degeneradas à Esquerda e à Direita

Uma **árvore estritamente binária** é uma árvore em que todos os *nodes* têm duas sub-árvores ou nenhuma sub-árvore.

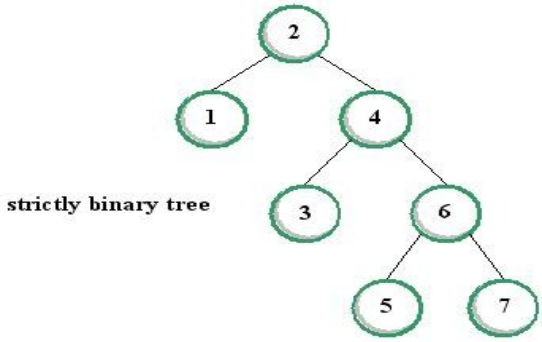


Figura 5: Árvore Estritamente Binária

Uma **árvore binária cheia** é uma árvore estritamente binária em que todos os *nodes* terminais estão no nível mais baixo. Dada uma profundidade, esta árvore tem o número máximo de *nodes*.

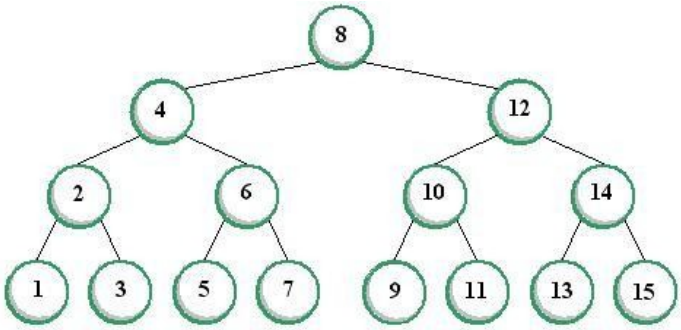


Figura 6: Árvore Binária Cheia

Uma **árvore binária completa** é uma árvore que resulta quando zero ou mais *nodes* são deletados de uma árvore binária cheia em ordem reversa de nível, isto é, da direita para a esquerda e de baixo para cima.

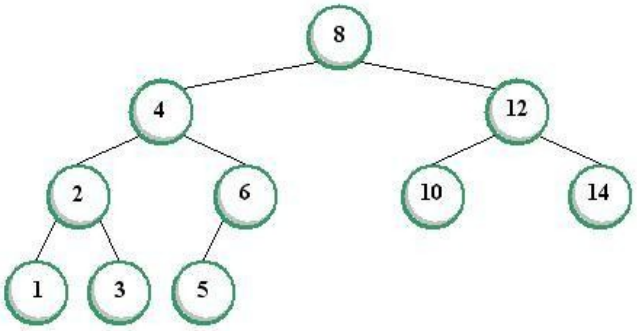


Figura 7: Árvore Binária Completa

3. Representação das Árvores Binárias

A maneira mais 'natural' para se representar uma árvore binária na memória do computador é utilizando a representação por *links*. A seguinte figura mostra a estrutura do *node* de uma árvore binária utilizando esta representação:

Figura 8: Nodes de Árvores Binárias

A seguinte classe Java implementa a representação acima:

```
public class BTNode {
  private Object info;
  private BTNode left, right;

  public BTNode(Object info) {
    this.setInfo(info);
  }
  public BTNode(Object info, BTNode left, BTNode right) {
    this.setInfo(info);
    this.setLeft(left);
    this.setRight(right);
  }
  public void setLeft(BTNode left) {
    this.left = left;
  }
  public BTNode getLeft() {
    return left;
  }
  public void setRight(BTNode right) {
    this.right = right;
  }
  public BTNode getRight() {
    return right;
  }
  public Object getInfo() {
    return info;
  }
  public void setInfo(Object info) {
    this.info = info;
  }
}
```

O exemplo abaixo mostra a representação com *links* de uma árvore binária:

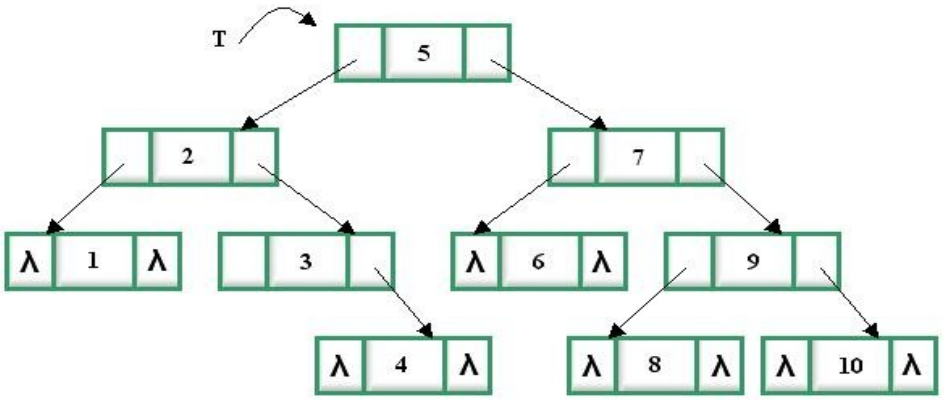


Figura 9: Representação com Links de uma Árvore Binária

Em Java, a seguinte classe define uma árvore binária:

```
public class BinaryTree {
  private BTNode root;

  public BinaryTree(BTNode node) {
    this.root = node;
  }
  public BinaryTree(BTNode node, BTNode left, BTNode right) {
    this.root = node;
    this.root.setLeft(left);
    this.root.setRight(right);
  }
}
```

4. Percorrendo Árvores Binárias

Pesquisas em árvores binárias geralmente envolvem uma **busca** ou **varredura**. **Busca** é um procedimento que percorre os *nodes* de uma árvore binária de maneira linear de modo que cada *node* é visitado apenas uma única vez. **Visitar** pode ser definido como a realização de computações locais no *node*.

Há três maneiras de se percorrer uma árvore: pré-ordem, em ordem e pós-ordem. Os prefixos (pré, em e pós) referem-se à ordem em que a raiz de cada sub-árvore é visitada.

4.1. Busca Pré-Ordem

Na busca pré-ordem de uma árvore binária, a raiz é primeiro *node* a ser visitado. Depois os children são percorridos recursivamente da mesma maneira. Este algoritmo é útil nas aplicações que requerem a listagem de elementos onde os pais sempre devem aparecer antes de seus children.

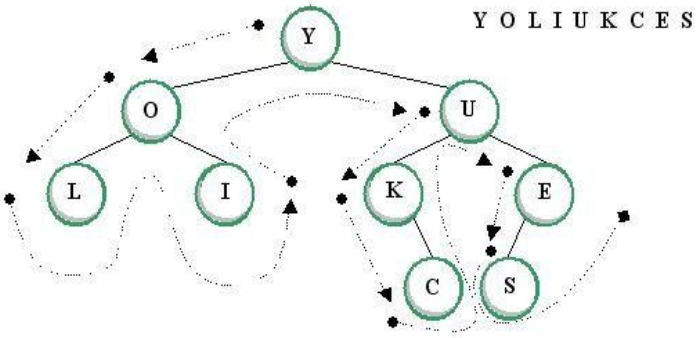


Figura 10: Percorrimento Pré-ordem

Método:

Se a árvore binária estiver vazia, não faça nada (busca finalizada).
Caso contrário:
Visite a raiz.
Percorra a sub-árvore esquerda em pré-ordem.
Percorra a sub-árvore direita em pré-ordem.

Em Java, adicione o seguinte método à classe *BinaryTree*:

```
// Listagem pré-ordem dos elementos da árvore
public void preorder() {
  if (root != null) {
    System.out.println(root.getInfo().toString());
    new BinaryTree(root.getLeft()).preorder();
    new BinaryTree(root.getRight()).preorder();
  }
}
```

4.2. Busca em Ordem

A busca em ordem de uma árvore binária pode ser definida, informalmente, como a pesquisa "da esquerda para a direita" de uma árvore binária. Isto é, a sub-árvore da esquerda é percorrida recursivamente em ordem, seguida por uma visita ao seu *node* pai, e finalizando com a pesquisa recursiva também em ordem da sub-árvore direita.

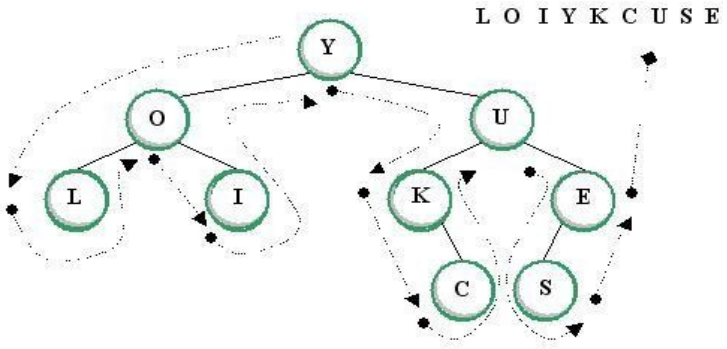


Figura 11: Percorrimento em Ordem

Método:

Se a árvore binária estiver vazia, não faça nada (busca finalizada).
Caso contrário:
Percorra a sub-árvore esquerda em ordem.
Visite a raiz.
Percorra a sub-árvore direita em ordem.

Em Java, adicione o seguinte método à classe *BinaryTree*:

```
// Listagem em ordem dos elementos da árvore
public void inorder() {
  if (root != null) {
    new BinaryTree(root.getLeft()).inorder();
    System.out.println(root.getInfo().toString());
    new BinaryTree(root.getRight()).inorder();
  }
}
```

4.3. Busca Pós-ordem

A busca pós-ordem é o contrário da pré-ordem, isto é, recursivamente percorrem-se primeiro os children e depois os pais.

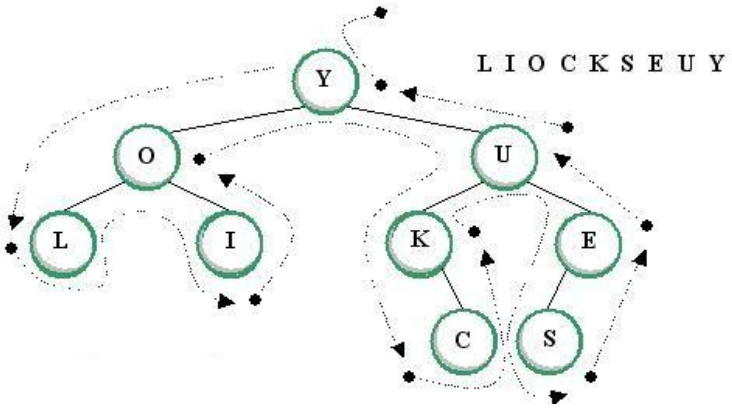


Figura 12: Percorrimento Pós-ordem

Método:

Se a árvore binária estiver vazia, não faça nada (percorrimento finalizado).
Caso contrário:

Percorra a sub-árvore esquerda em pós-ordem.
Percorra a sub-árvore direita em pós-ordem.
Visite a raiz.

Em Java, adicione o seguinte método à classe *BinaryTree*:

```
// Listagem pós-ordem dos elementos da árvore
public void postorder() {
  if (root != null) {
    new BinaryTree(root.getLeft()).postorder();
    new BinaryTree(root.getRight()).postorder();
    System.out.println(root.getInfo().toString());
  }
}
```

A seguir temos alguns exemplos:

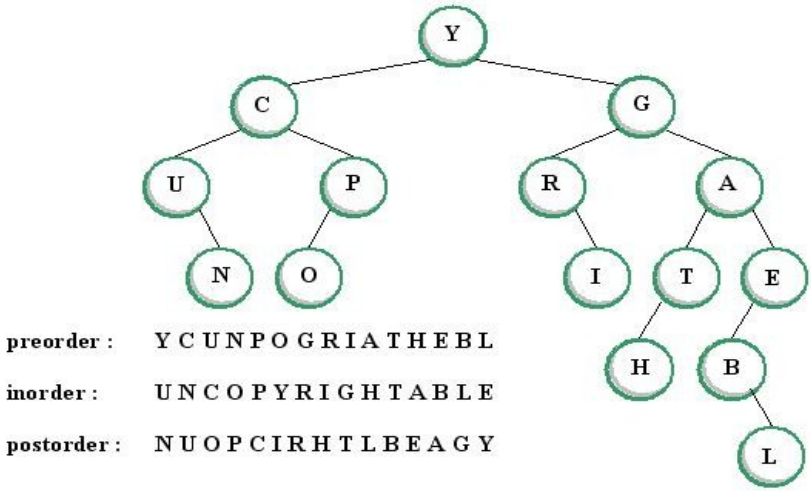


Figura 13: Exemplo 1

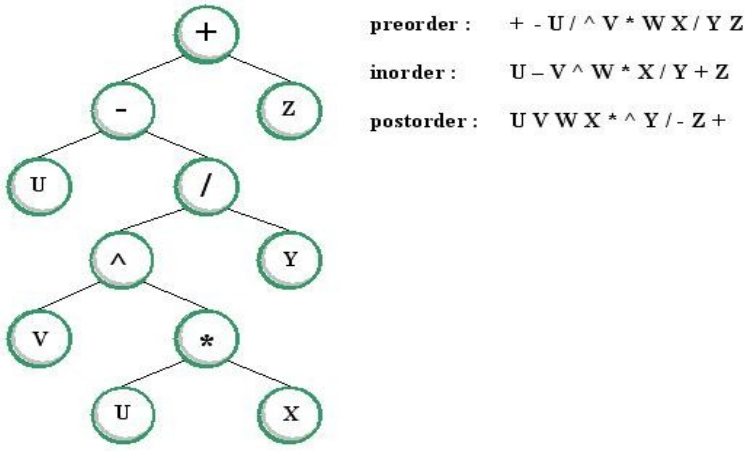


Figura 14: Exemplo 2

5. Aplicações de Busca em Árvores Binárias

A busca em árvores binárias tem várias aplicações. Nesta seção, duas aplicações serão abordadas: a duplicação de uma árvore binária e a verificação da equivalência entre duas árvores binárias.

5.1. Duplicando uma Árvore Binária

Existem momentos em que a duplicação de uma árvore binária é necessária para um processamento. Para isso, pode ser utilizado o seguinte algoritmo para duplicar uma árvore binária existente:

1. Percorra a sub-árvore esquerda do *node* α em pós-ordem e faça uma cópia dela
2. Percorra a sub-árvore da direita do *node* α em pós-ordem e faça uma cópia dela
3. Faça uma cópia do *node* α e anexe as cópias de suas sub-árvores da esquerda e da direita

O seguinte método da classe *BinaryTree* implementa este algoritmo:

```
public BTNode copy() {
  BTNode newRoot;
  BTNode newLeft;
  BTNode newRight;
  if (root != null) {
    newLeft = new BinaryTree(root.getLeft()).copy();
    newRight = new BinaryTree(root.getRight()).copy();
    newRoot = new BTNode(root.getInfo(), newLeft, newRight);
    return newRoot;
  }
  return null;
}
```

5.2. Equivalência entre Duas Árvores Binárias

Duas árvores binárias são equivalentes se uma é cópia exata da outra. O seguinte algoritmo verifica a equivalência entre duas árvores binárias:

1. Verifique se o *node* α e o *node* β contêm os mesmos dados
2. Percorra as sub-árvores da esquerda, *nodes* α e β em pré-ordem e verifique se são equivalentes
3. Percorra as sub-árvores da direita, *nodes* α e β em pré-ordem e verifique se são equivalentes

O seguinte método da classe *BinaryTree* implementa este algoritmo:

```
public boolean equivalent(BinaryTree t2) {
  boolean answer = false;
  if ((root == null) && (t2.root == null))
    answer = true;
  else {
    answer = (root.getInfo().equals(t2.root.getInfo()));
    if (answer) answer = new BinaryTree(root.getLeft()).equivalent(
      new BinaryTree(t2.root.getLeft()));
    if (answer) answer = new BinaryTree(root.getRight()).equivalent(
      new BinaryTree(t2.root.getRight()));
  }
  return answer;
}
```

É possível testar esta classe implementando o seguinte método main:

```
public static void main(String args[]) {
  BinaryTree bt1 = new BinaryTree(
    new BTNode("Root1"), new BTNode("Left1"), new BTNode("Right1"));
  BinaryTree bt2 = new BinaryTree(
    new BTNode("Root2"), new BTNode("Left2"), new BTNode("Right2"));
  BinaryTree bt3 = new BinaryTree(
    new BTNode("Root1"), new BTNode("Left1"), new BTNode("Right1"));

  System.out.println("Preorder(bt1): ");
  bt1.preorder();
  System.out.println("Inorder(bt1): ");
  bt1.inorder();
  System.out.println("Postorder(bt1): ");
  bt1.postorder();

  BinaryTree bt4 = new BinaryTree(bt1.copy());
  System.out.println("Preorder (bt4): ");
  bt4.preorder();

  System.out.println(bt1.equivalent(bt2));
  System.out.println(bt1.equivalent(bt3));
}
```

E na execução desta classe, como resultado teremos:

```
Preorder(bt1):
Root1
Left1
Right1
Inorder(bt1):
Left1
Root1
Right1
Postorder(bt1):
Left1
Right1
Root1
Preorder (bt4):
Root1
Left1
Right1
false
true
```

6. Aplicação de Árvore Binária: *Heaps* e o Algoritmo *Heapsort*

Um ***heap*** é definido como uma árvore binária completa que tem elementos armazenados em seus *nodes* e satisfaz a *propriedade ordem-heap*. Uma árvore binária completa, como definida na lição anterior, resulta quando zero ou mais *nodes* são excluídos de uma árvore binária completa em ordem inversa de nível. Conseqüentemente, suas folhas ficam no máximo em dois níveis adjacentes e as folhas do nível mais baixo ficam na posição mais à esquerda da árvore binária completa.

A **propriedade *ordem-heap*** define que para todo *node* **u** exceto a raiz, a chave armazenada em **u** é menor ou igual à chave armazenada em seu respectivo *node* pai. Então, a raiz sempre contém o valor máximo.

Nesta aplicação, os elementos armazenados em um *heap* satisfazem a ***ordem total***. Uma **ordem total** é uma relação entre os elementos de um conjunto de objetos, nomeado S, que satisfazem as propriedades de quaisquer objetos x, y e z em S:

- Transitividade: se x < y e y < z então x < z.
- Tricotomia: para quaisquer dois objetos x e y em S, exatamente uma destas relações é verdadeira: x > y, x = y ou x < y.

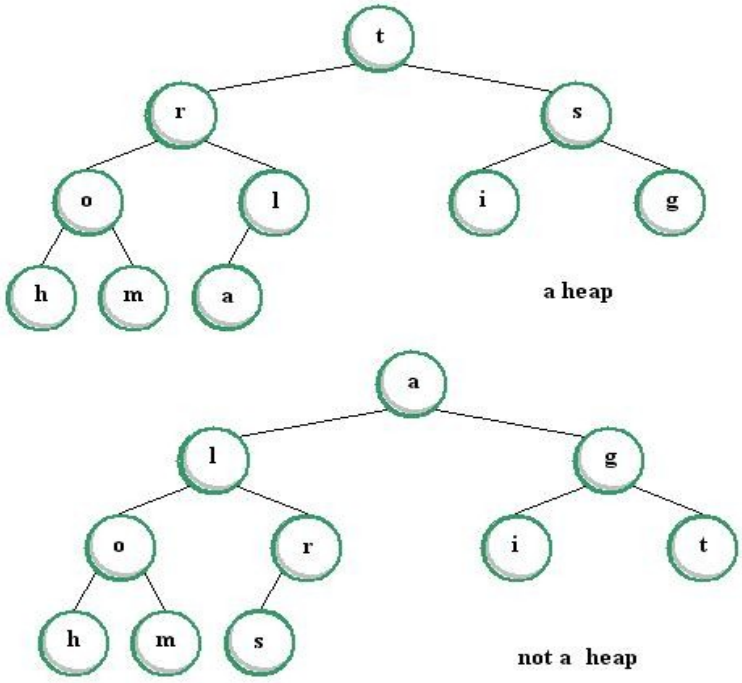


*Figura 15: Duas representações dos elementos **a l g o r i t h m s***

6.1. Shift-Up

Uma árvore binária completa pode ser convertida em uma *stack* por meio da aplicação de um processo chamado *shift-up*. Neste processo, chaves maiores "sobem" a árvore para satisfazer a *propriedade de ordenamento de stacks*. Este é um processo que se dá de baixo para cima e da direita para a esquerda, durante o qual as sub-árvores menores de uma árvore binária completa são convertidas em *stacks*, seguido pela conversão das sub-árvores que as contêm, e assim por diante, até que toda a árvore binária seja convertida em uma *stack*.

Observe que quando uma sub-árvore com raiz em algum *node*, por exemplo α, é convertida em uma *stack*, as sub-árvores da esquerda e da direita, vinculadas à mesma, já são *stack*s. Este tipo de sub-árvore é denominada quase-*stack*. Quando uma quase-*stack* é convertida em uma *stack*, pode ocorrer de uma de suas sub-árvores deixar de ser uma *stack* (ex. Pode se tornar uma quase-*stack*). A mesma, porém, pode ser convertida em uma *stack* e o processo continua com sub-árvores pequenas e outras menores ainda perdendo e reconquistando a propriedade de *stack*, com chaves MAIORES migradas para o topo.

Para testarmos os exemplos, criamos uma nova classe denominada *SequentialHeap* e a iniciamos com o seguinte código:

```
public class SequentialHeap {
  int key[];

  public SequentialHeap(){
    key = new int[10];
  }
  public SequentialHeap(int size){
    key = new int[size];
  }
  public SequentialHeap(int k[]){
    key = k;
  }
}
```

6.2. Representação Seqüencial de uma Árvore Binária Completa

Uma árvore binária completa pode ser representada seqüencialmente em um vetor unidimensional de tamanho n em ordem de nível. Se os *nodes* de uma árvore são numerados, como ilustrado abaixo,

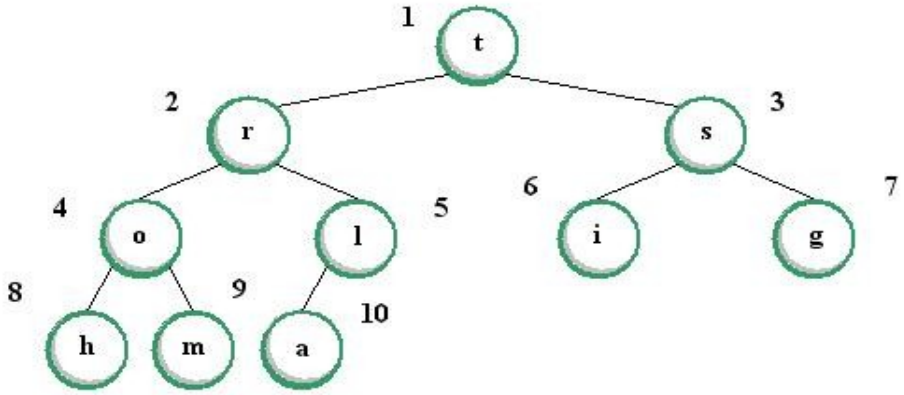


Figura 16: Uma árvore binária completa

pode então ser representada seqüencialmente por meio de um vetor de nome CHAVE, conforme demonstrado abaixo:

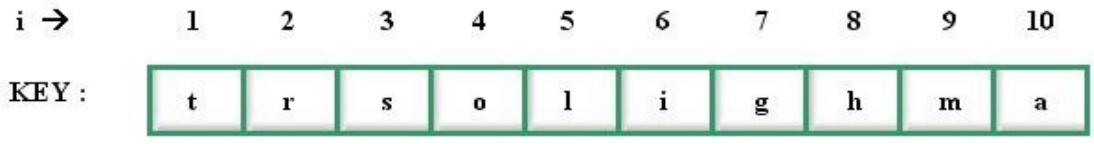


Figura 17: Representação Seqüencial de uma Árvore Binária Completa

A representação seqüencial de uma árvore binária completa possibilita a localização dos children (se houver), do pai (se existir) e do pai de um *node* em tempo constante, por meio da utilização da seguinte fórmula:

- se 2i ≤ n, o *child* à esquerda do *node* i é 2i; senão, o *node* i não possui nenhum child à esquerda
- se 2i + 1 ≤ n, o *child* à direita do *node* i é 2i + 1; senão, o *node* i não possui nenhum child à direita
- se 1 < i ≤ n, o pai do *node* i é ⌊ (i)/2 ⌋

O método abaixo, inserido na classe *SequentialHeap*, implementa o processo de *shift-up* em uma árvore binária completa representada seqüencialmente:

```
// Converte uma árvore binária com n nodes e raiz em uma stack
private void shiftUp (int i, int n) {
  int k = key[i];  // mantém a chave na raiz da stack
  int child = 2 * i; // child à esquerda
  while (child <= n) {
    // se child à direita é maior, aponta-o para o da direita
    if (child < n && key[child+1]> key[child])
      child++ ;
    // se a propriedade de stack não for satisfeita
    if (key[child] > k) {
      key[i] = key[child] ; // Move child para cima
      i = child;
      child = 2 * i; //Considera child da esquerda novamente
    } else
        break;
  }
  key[i] = k ; // aqui começa a raiz
}
```

Para converter uma árvore binária em uma quase-*stack*:

```
// Converte chave em quase-stack
for (int i=n/2; i>1; i--){
  // o primeiro node que tem children é n/2
  shiftUp(i, n);
}
```

6.3. O Algoritmo Heapsort

Heapsort é um algoritmo elegante de ordenação que foi desenvolvido em 1964 por *R. W. Floyd* e *J. W. J. Williams*. O *heap* é a base deste algoritmo de ordenação elegante:

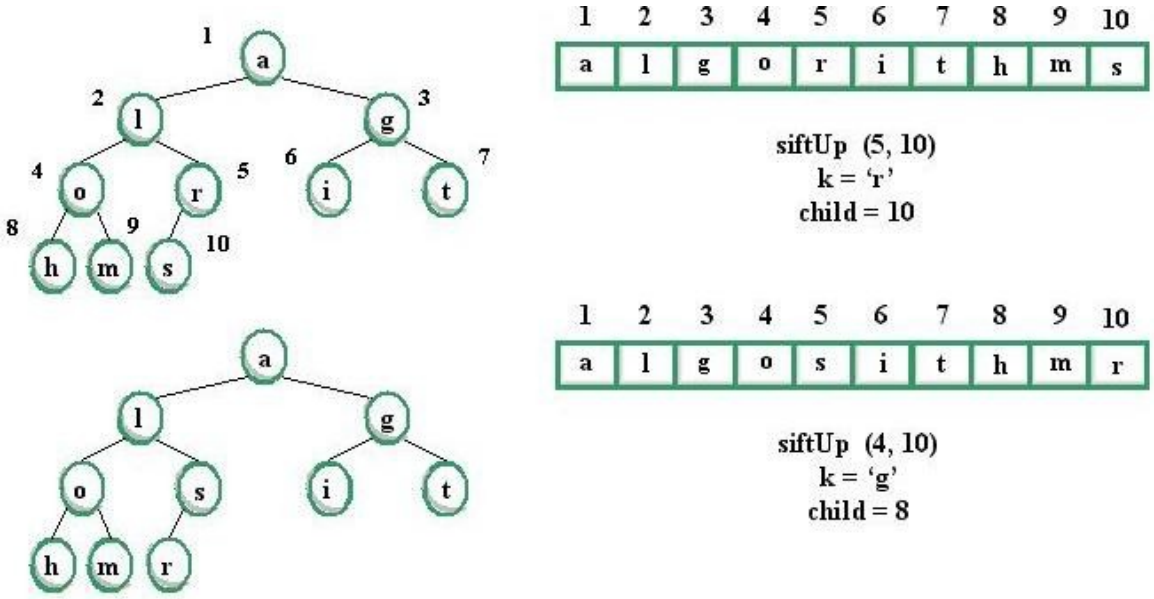
1. Atribua as chaves a serem ordenadas aos *nodes* de uma árvore binária completa
2. Converta esta árvore binária em um *heap* aplicando o método *shift-up* aos seus *nodes* em ordem reversa de nível
3. Repita o seguinte até que o *heap* esteja vazio:
   (a) Remova a chave na raiz do heap (o menor valor no *heap*) e coloque-o na saída
   (b) Extraía do *heap* o *node*-folha mais à direita no nível mais baixo, obtenha sua chave e armazene-a na raiz do *heap*
   (c) Aplique *shift-up* à raiz para converter a árvore binária em um *heap* novamente

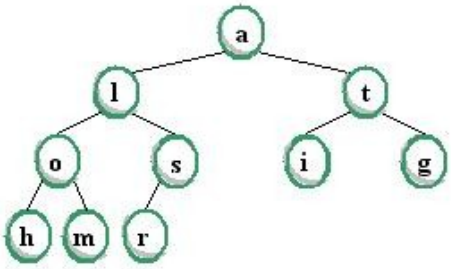
O método a seguir, inserido na classe *SequentialHeap*, implementa o algoritmo *heapsort* em Java:

```
public void sort() {
  int n = key.length-1;
  // converte a chave para almost-heap
  for (int i=n/2; i>1; i--) {
    // primeiro node como child é n/2
    shiftUp(i, n);
  }
  // Muda o corrente tamanho da chave[1] com a chave[i]
  for (int i=n; i>1; i--) {
    shiftUp(1, i);
    int temp = key[i];
    key[i] = key[1];
    key[1] = temp;
  }
}
```

O exemplo seguinte mostra a execução do *heapSort* com as chaves de entrada:

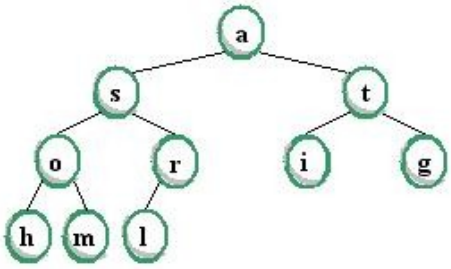
a l g o r i t h m s





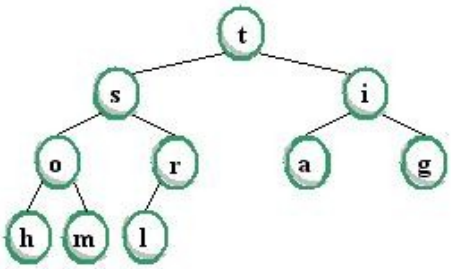
1	2	3	4	5	6	7	8	9	10
a	l	t	o	s	i	g	h	m	r

siftUp (3, 10)
k = 'g'
child = 6



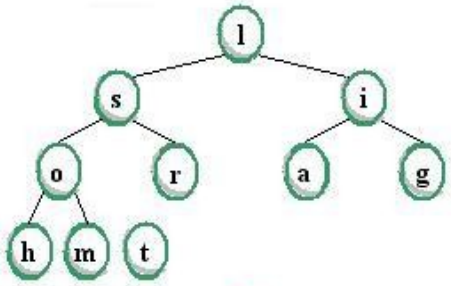
1	2	3	4	5	6	7	8	9	10
a	s	t	o	r	i	g	h	m	l

siftUp (2, 10)
k = 'l'
child = (4) 5



1	2	3	4	5	6	7	8	9	10
t	s	i	o	r	a	g	h	m	l

siftUp (1, 10)
k = 'a'
child = (2) 3



1	2	3	4	5	6	7	8	9	10
l	s	i	o	r	a	g	h	m	t

heap → | ← sorted

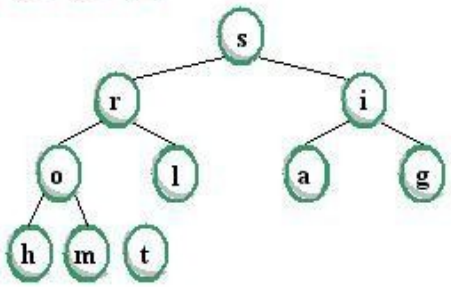
i = 10

key[1] ⇔ key [10]

1	2	3	4	5	6	7	8	9	10
s	r	i	o	l	a	g	h	m	t

heap → | ← sorted

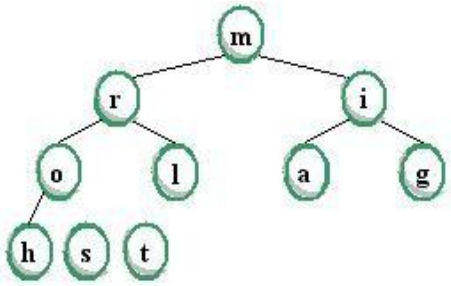
siftUp (1, 9)



1	2	3	4	5	6	7	8	9	10
m	r	i	o	l	a	g	h	s	t

heap → | ← sorted

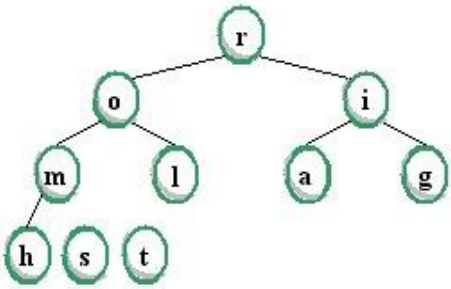
key[1] ⇔ key [9]



1	2	3	4	5	6	7	8	9	10
r	o	i	m	l	a	g	h	s	t

heap → | ← sorted

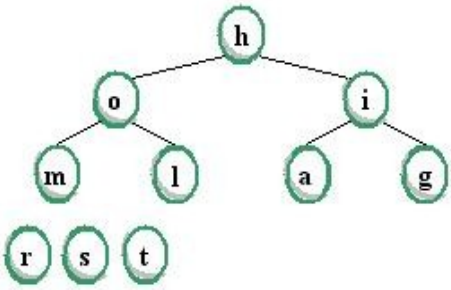
siftUp (1,8)

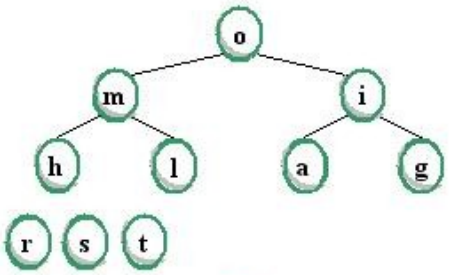


1	2	3	4	5	6	7	8	9	10
h	o	i	m	l	a	g	r	s	t

heap → | ← sorted

key[1] ⇔ key [8]

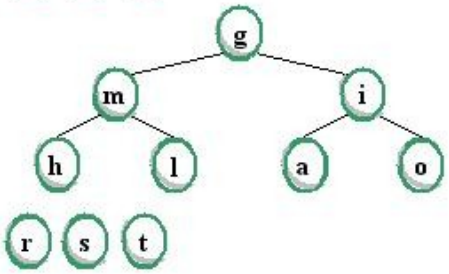




1	2	3	4	5	6	7	8	9	10
o	m	i	h	l	a	g	r	s	t

heap → | ← sorted

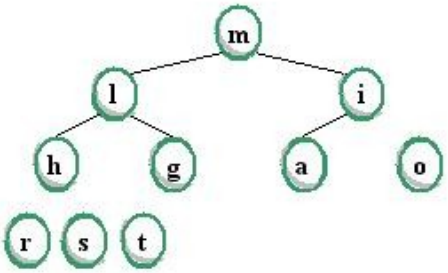
siftUp (1, 7)



1	2	3	4	5	6	7	8	9	10
g	m	i	h	l	a	o	r	s	t

heap → | ← sorted

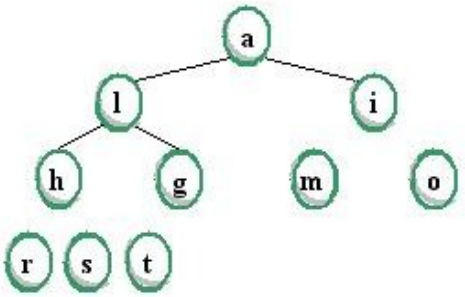
key[1] ⇔key[7]



1	2	3	4	5	6	7	8	9	10
m	l	i	h	g	a	o	r	s	t

heap → | ← sorted

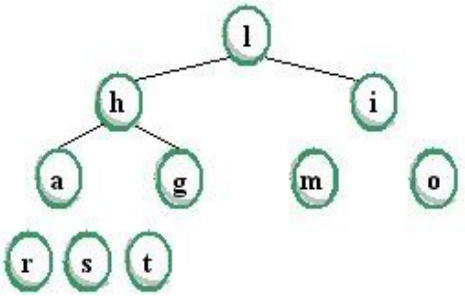
siftUp(1, 6)



1	2	3	4	5	6	7	8	9	10
a	l	i	h	g	m	o	r	s	t

heap → | ← sorted

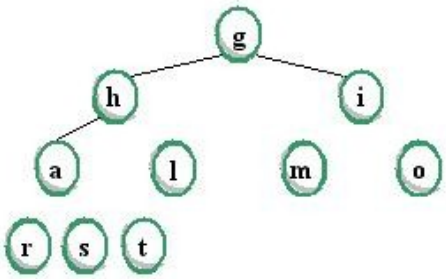
key[1] ⇔ key[6]



1	2	3	4	5	6	7	8	9	10
l	h	i	a	g	m	o	r	s	t

heap → | ← sorted

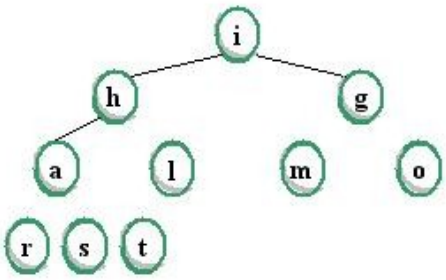
siftUp(1, 5)



1	2	3	4	5	6	7	8	9	10
g	h	i	a	l	m	o	r	s	t

heap → | ← sorted

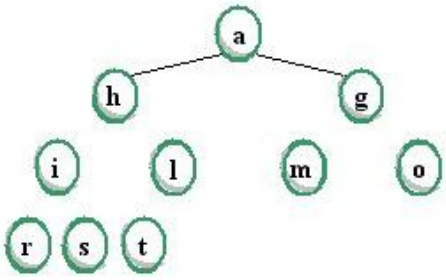
key[1] ⇔ key[5]



1	2	3	4	5	6	7	8	9	10
i	h	g	a	l	m	o	r	s	t

heap → | ← sorted

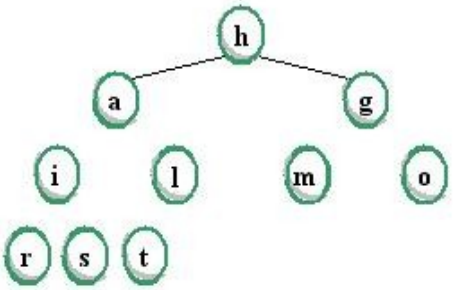
siftUp (1, 4)



1	2	3	4	5	6	7	8	9	10
a	h	g	i	l	m	o	r	s	t

heap → | ← sorted

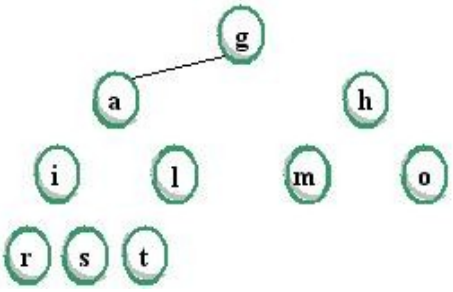
key[1] ⇔ key[4]



1	2	3	4	5	6	7	8	9	10
h	a	g	i	l	m	o	r	s	t

heap → | ← sorted

siftUp (1, 3)



1	2	3	4	5	6	7	8	9	10
g	a	h	i	l	m	o	r	s	t

heap → | ← sorted

key[1] ⇔ key[3]

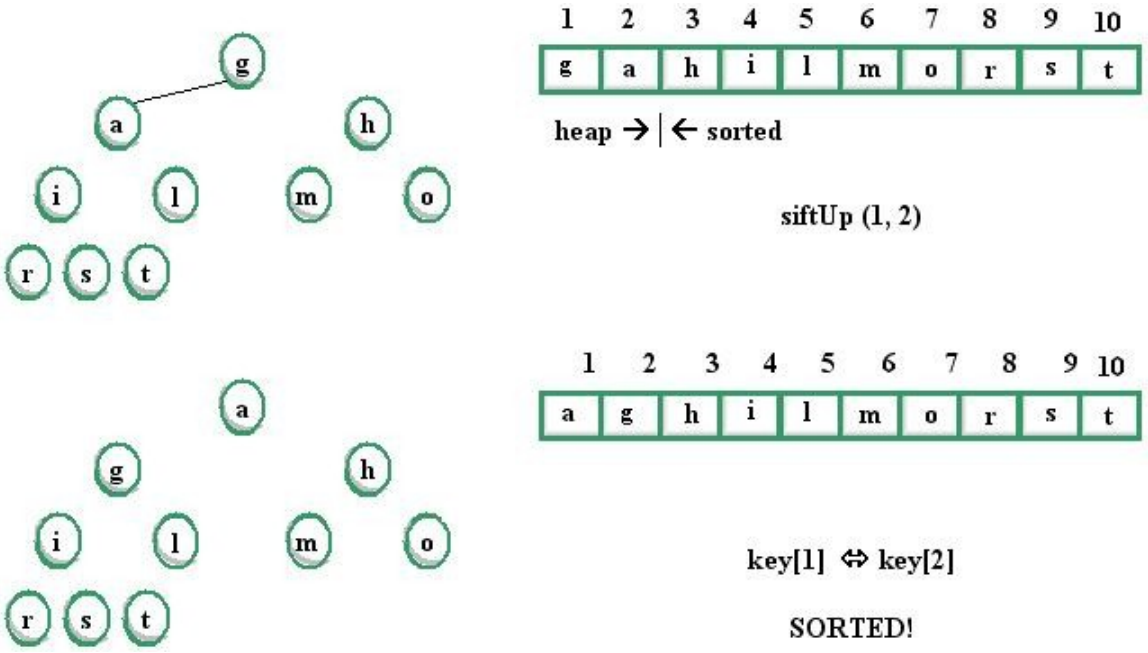


Figura 18: Exemplo de Heapsort

Poderíamos testar esta classe com a implementação do seguinte método principal:

```
public static void main(String args[]){
  int keys[] = {'a','l','g','o','r','i','t','h','m','s'};
  SequentialHeap heap = new SequentialHeap(keys);
  System.out.print("Before sorting: ");
  for (int i=0; i < keys.length;i++)
    System.out.print((char) keys[i] + " ");
  System.out.println();
  heap.sort();
  System.out.print("Before sorting: ");
  for (int i=0; i < keys.length;i++)
    System.out.print((char) keys[i] + " ");
  System.out.println();
}
```

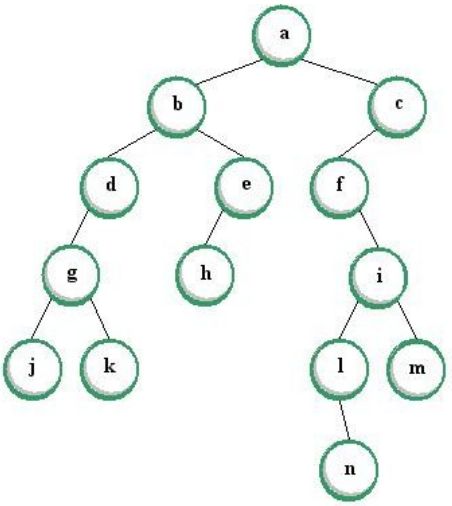
E iremos obter o seguinte resultado:

```
Before sorting: a l g o r i t h m s
Before sorting: a g h i l m o r s t
```

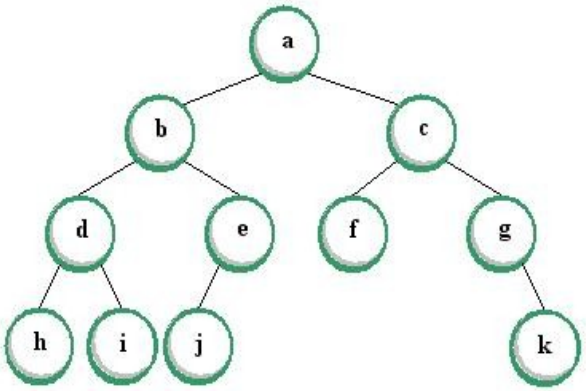
7. Exercícios

1. *Varredura*. Percorra a seguinte árvore em pré-ordem, em ordem e pós-ordem.

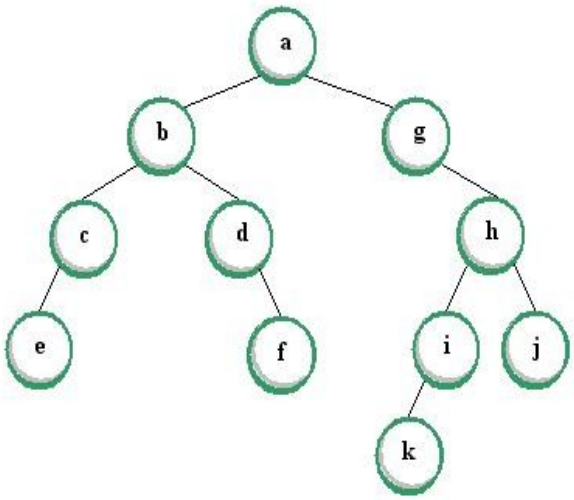
a)



b)



c)



2. *Heapsort.* Organize os seguintes elementos em ordem crescente. Mostre o estado da árvore em cada passo.

   a) C G A H F E D J B I
   b) 1 6 3 4 9 7 5 8 2 12 10 14

7.1. Exercícios para Programar

1. *Heaps* podem ser usadas para avaliação de expressões. Um método alternativo é a utilização de árvores binárias. Os usuários vêem as expressões na forma pré-fixada, mas a forma pós-fixada é a mais adequada para que computadores avaliem as expressões. Neste exercício de programação, crie um algoritmo que faça a conversão da expressão pré-fixada para sua forma pós-fixada, utilizando árvore binária. Cinco operações binárias são apresentadas e estão listadas aqui de acordo com a precedência:

Operação	Descrição
^	Exponenciação (maior precedência)
* /	Multiplicação e Divisão
+ -	Adição e Subtração

Na sua implementação, considere a precedência e prioridade dos operadores.

2. Modifique o algoritmo *heapsort* para retornar as chaves em ordem decrescente, ao invés de crescente, deslocando para cima as chaves menores ao invés das chaves maiores.

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.