

Módulo 3

Estruturas de Dados



Lição 5

Árvores

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

1. Objetivos

Árvores podem ser ordenadas, orientadas ou livres. A alocação de ponteiros pode ser usada para representar árvores e estas podem ser representadas seqüencialmente usando a representação aritmética da árvore. Zero ou mais árvores separadas fazem juntas uma floresta e esta ordenada pode ser convertida em uma árvore binária única e vice-versa usando correspondência natural.

Ao final desta lição, o estudante será capaz de:

- Discutir os conceitos básicos e definições de árvores
- Identificar os tipos de árvores: ordenadas, orientadas e árvores livres
- Usar a representação de árvores com ponteiros
- Explanar os conceitos básicos e definições sobre florestas
- Converter uma floresta na sua representação de árvore binária e vice-versa usando a correspondência natural
- Percorrer uma floresta usando o processo pré-ordem, pós-ordem, por nível e por família
- Criar representações de árvores usando a alocação seqüencial
- Utilizar a representação aritmética de árvores
- Utilizar árvores em uma aplicação: O problema da equivalência

2. Definições e Conceitos Relacionados

2.1. Árvores Ordenadas

Uma árvore ordenada é um conjunto finito, chamado T , de um ou mais *nodes* de modo que há um *node* especialmente chamado de raiz, e os demais *nodes* raiz são particionados em $n \geq 0$ conjuntos disjuntos (sem elementos em comum) T_1, T_2, \dots, T_n , onde cada um desses conjuntos é por sua vez uma árvore ordenada. Em uma árvore ordenada, a ordem de cada *node* na árvore é importante.

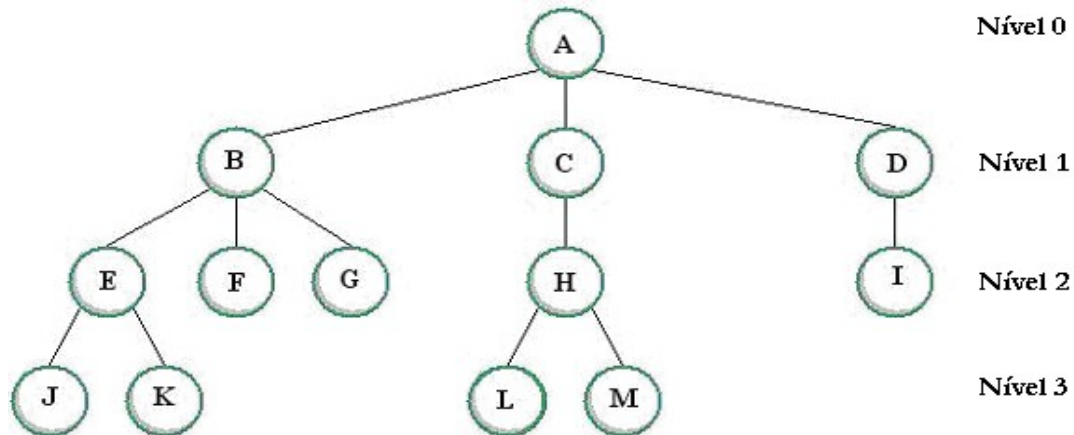


Figura 1: Uma árvore ordenada

O **grau** de uma árvore é definido como o grau do(s) *node(s)* com o maior grau. Por essa razão, a árvore acima tem o grau 3.

2.2. Árvore Orientada

Uma árvore orientada é uma árvore em que a ordem de cada subárvore de cada *node* da árvore é secundário.

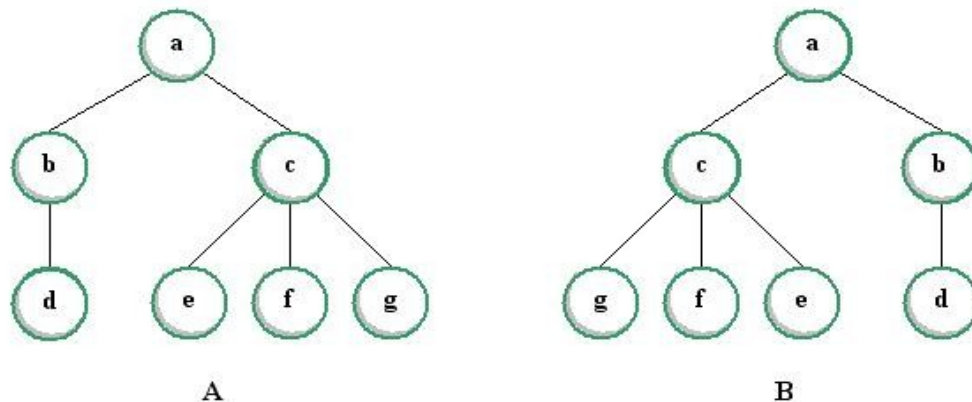


Figura 2: Uma Árvore Orientada

No exemplo acima, as duas árvores são duas árvores orientadas diferentes, mas são a mesma árvore.

2.3. Árvore Livre

Uma árvore livre não tem um *node* designado como raiz e a orientação de um *node* para outro é sem importância.

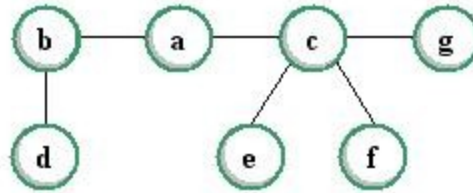


Figura 3: Uma Árvore Livre

2.4. Progressão de Árvores

Quando em uma árvore livre é designado um *node* raiz, ela torna-se uma árvore orientada. Quando a ordem dos *nodes* é definida em uma árvore orientada, ela torna-se uma árvore ordenada. O seguinte exemplo demonstra essa progressão.

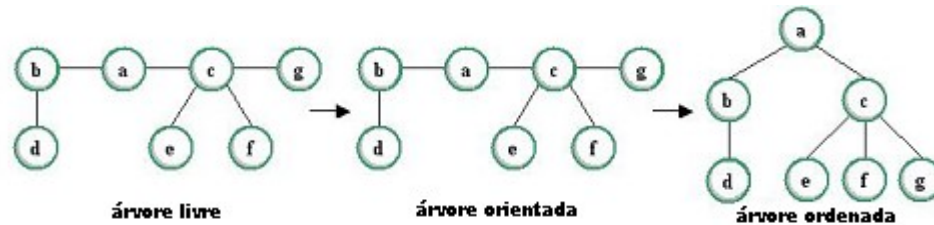


Figura 4: Progressão de Árvores

3. Representação Ligada de Árvores

Alocação ligada pode ser usada para representar árvores. A figura abaixo mostra a estrutura dos *nodes* usados nessa representação.



Figura 5: Estrutura de node de uma Árvore

Na estrutura de *nodes*, k é o grau da árvore. $SON_1, SON_2, \dots, SON_k$ são ponteiros para os possíveis k filhos de um *node*.

Aqui temos algumas propriedades de uma árvore com n *nodes* e com grau k :

- O número de campos ponteiros é igual a $n*k$
- O número de ponteiros não vazios é igual a $n-1$ (Número de ramos)
- O número de ponteiros vazios é igual a $n*k - (n-1)$, ou seja, $n(k-1) + 1$

A representação ligada é a maneira mais natural de representar uma árvore. Porém, devido às propriedades acima, uma árvore com grau 3 terá 67% de ponteiros nulos, enquanto que em uma árvore com grau 10, o espaço vazio será de 90%. Uma perda considerável de espaço é introduzida por essa abordagem. Caso a utilização de espaço seja um assunto importante, podemos optar por usar a estrutura alternativa.



Figura 6: Node de Árvore Binária

Com essa estrutura, LEFT aponta para o filho à esquerda do *node* enquanto RIGHT aponta para o próximo irmão mais novo.

4. Florestas

Quando zero ou mais árvores disjuntas são associadas, são conhecidas como florestas. A seguir um exemplo de floresta:

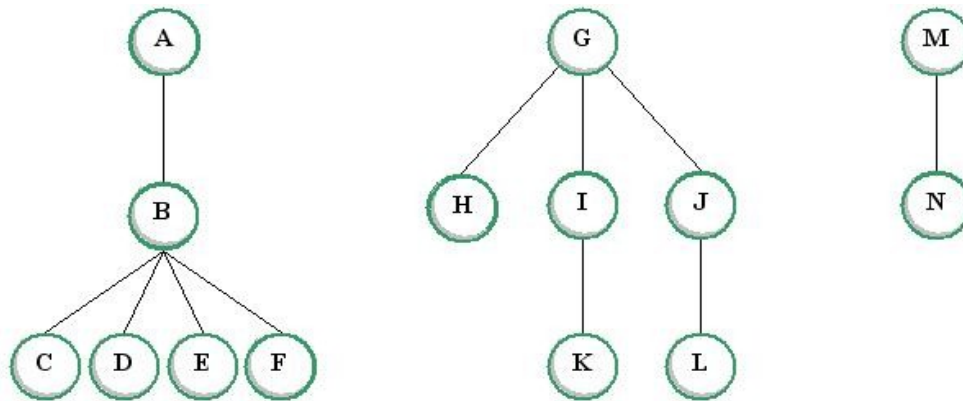


Figura 7: Floresta F

Se as árvores que compreendem a floresta são árvores ordenadas e se a sua ordem na floresta é essencial, ela é conhecida como **floresta ordenada**.

4.1. Correspondência Natural: Árvore Binária, representação de Floresta

Uma floresta ordenada, digamos F, pode ser convertida em uma árvore binária única, digamos B(F), e vice-versa, usando um processo bem definido conhecido como *correspondência natural*. Formalmente:

Seja $F = (T_1, T_2, \dots, T_n)$ uma floresta ordenada de árvores ordenadas. A árvore binária B(F) correspondente a F é obtida da seguinte maneira:

- Se $n = 0$, então B(F) é vazia.
- Se $n > 0$, então a raiz de B(F) é a raiz de T_1 ; a subárvore esquerda de B(F) é B($T_{11}, T_{12}, \dots, T_{1m}$), onde $T_{11}, T_{12}, \dots, T_{1m}$ são subárvores da raiz de T_1 ; e a subárvore direita de B(F) é B(T_2, T_3, \dots, T_n).

A correspondência natural pode ser implementada usando uma abordagem não-recursiva:

- Ligar os filhos de cada família da esquerda para direita. (Nota: as raízes da árvore na floresta são irmãs, filhas de um pai desconhecido.)
- Remover ligações de um pai para todos os seus filhos exceto o filho mais velho (ou mais à esquerda).
- Inclinar a figura resultante em 45 graus.

O exemplo a seguir ilustra a transformação de uma floresta em sua árvore binária equivalente:

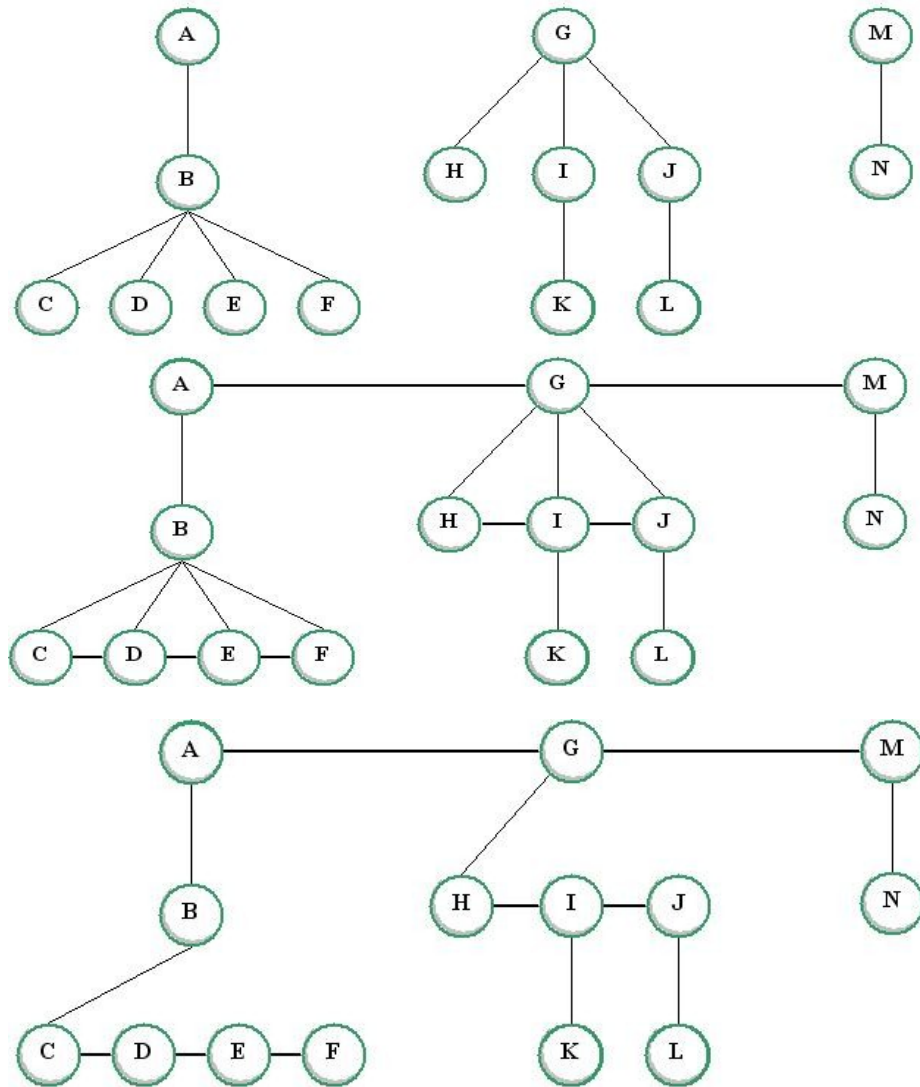


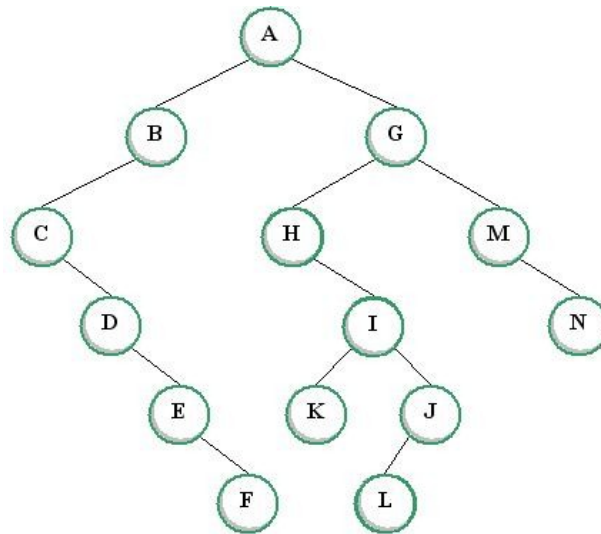
Figura 8: Exemplo de Correspondência Natural

Em correspondência natural, a *raiz* é trocada pela raiz da primeira árvore, a subárvore esquerda é trocada pelas subárvores da primeira árvore e a subárvore direita é trocada pelas árvores remanescentes.

4.2. Atravessando a Floresta

Como em árvores binárias, florestas podem ser atravessadas (percorridas). Contudo, uma vez que o conceito de *node* intermediário não esteja definido, uma floresta somente pode ser atravessada em pré-ordem e pós-ordem.

Se a floresta estiver vazia, o atravessar é considerado **executado**; senão:



- Atravessar em Pré-Ordem
 - Visitar a raiz da primeira árvore
 - Atravessar as subárvores da primeira árvore em pré-ordem
 - Atravessar as árvores restantes em pré-ordem
- Atravessar em Pós-Ordem
 - Atravessar as subárvores da primeira árvore em pós-ordem
 - Visitar a raiz da primeira árvore
 - Atravessar as árvores remanescentes em pós-ordem

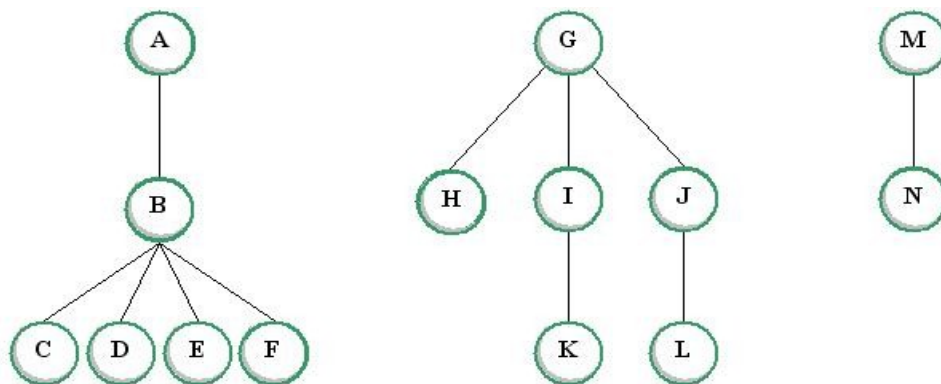


Figura 9: Floresta F

Floresta pré-ordem : A B C D E F G H I K J L M N

Floresta pós-ordem : C D E F B A H K I L J G N M

A árvore binária equivalente da floresta resultará na seguinte listagem para pré-ordem em-ordem e pós-ordem

B(F) pré-ordem : A B C D E F G H I K J L M N

B(F) em-ordem : C D E F B A H K I L J G N M

B(F) pós-ordem : F E D C B K L J I H N M G A

Observe que a floresta pós-ordem produz o mesmo resultado que na floresta em-ordem B(F). Não é coincidência.

4.3. Representação Seqüencial de Florestas

Florestas podem ser implementadas usando representação seqüencial. Considere a floresta F e sua árvore binária correspondente. Os exemplos a seguir mostram a árvore usando representações seqüenciais de pré-ordem, família-ordem e nível-ordem.

4.3.1. Representação Seqüencial em Pré-ordem

Nesta representação seqüencial, os elementos são listados nas suas seqüências pré-ordem e dois *arrays* adicionais são mantidos – RLINK e LTAG. **RLINK** é um ponteiro de um irmão mais velho para um irmão mais novo na floresta, ou de um pai para o seu filho da direita na sua representação de árvore binária. **LTAG** indica se um *node* é terminal (*node* folha) e o símbolo ')' é usado como indicativo.

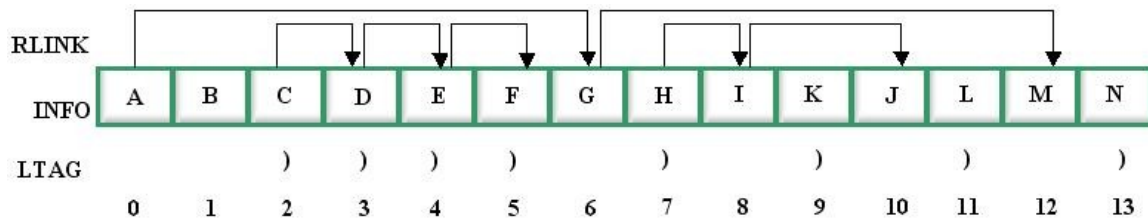


Figura 10: Representação seqüencial pré-ordem da floresta F

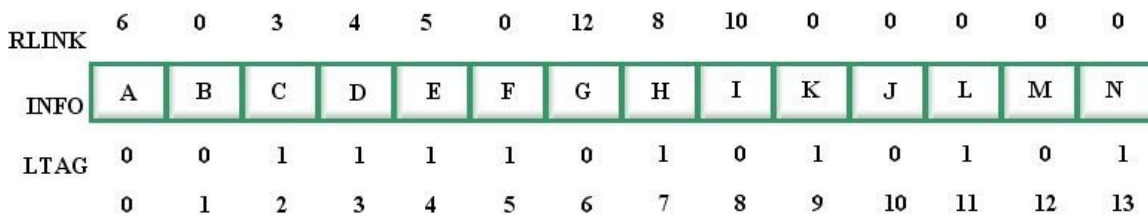


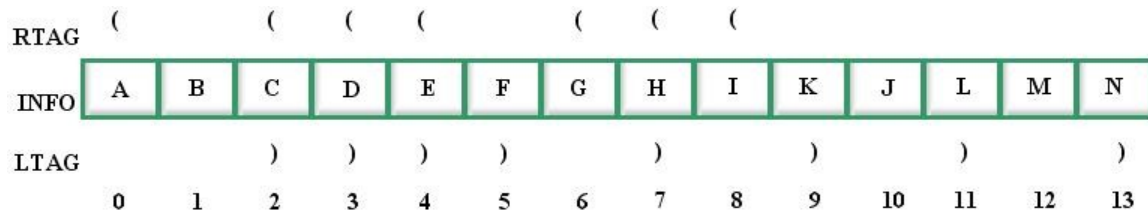
Figura 11: Representação interna real

RLINK contém o *node* apontado pelo *node* atual e LTAG tem um valor de 1 para cada ')' na representação.

Uma vez que o *node* final sempre precede imediatamente um *node* apontado por uma seta, exceto o último *node* na seqüência, o uso do dado pode ser diminuído pela

- (1) Eliminação de LTAG; ou
- (2) Substituição de RLINK com RTAG que simplesmente identifica os *nodes* onde uma seta procede

Usando a segunda opção, será necessária a utilização de uma *stack* para estabelecer a relação entre os *nodes*, já que as setas têm estrutura "último a entrar, primeiro a sair", e usando-a levará a seguinte representação:



E isto é representado internamente como:

RTAG	1	0	1	1	1	0	1	1	1	0	0	0	0	0
INFO	A	B	C	D	E	F	G	H	I	K	J	L	M	N
LTAG	0	0	1	1	1	1	0	1	0	1	0	1	0	1
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Tendo valores de bit para RTAG e LTAG, a última opção mostra claramente o menor espaço requerido para armazenamento. Porém, acarreta mais processamento na recuperação da floresta.

4.3.2. Representação Seqüencial de Percurso por Família (Family-Order)

Nesta representação sequencial, a listagem por família de elementos é usada. Atravessar por família, a primeira família a ser listada consiste de *nodes* raízes de todas as árvores na floresta e subsequente, as famílias são listadas com base em **primeiro-a-entrar primeiro-a-sair (FIFO)**. Esta representação faz uso de LLINK e RTAG. **LLINK** é um ponteiro para o filho mais à esquerda de um *node* ou o filho à esquerda numa representação de árvore binária. **RTAG** identifica o irmão mais novo na linhagem ou o último membro da família.

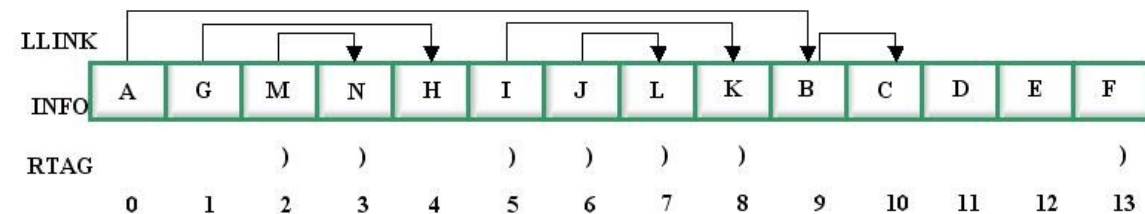


Figura 12: Representação seqüencial família-ordem da floresta F

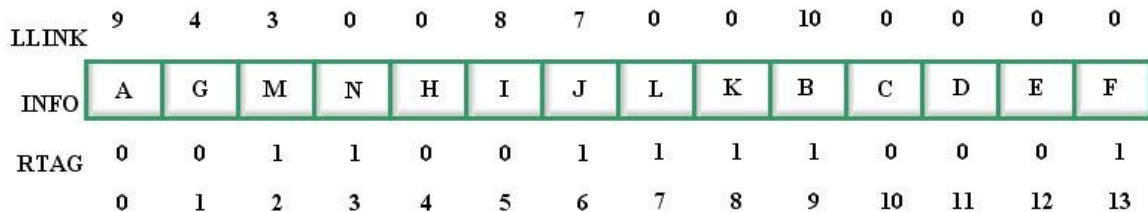
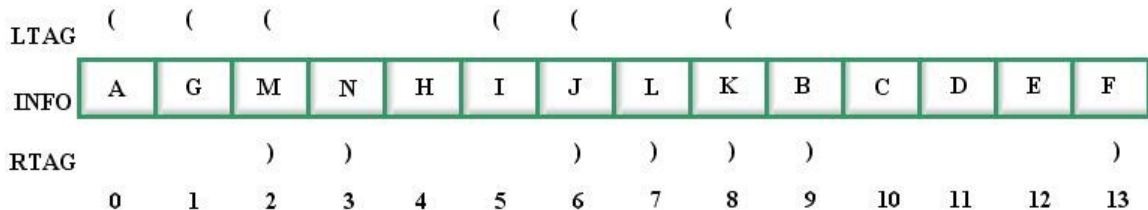


Figura 13: Representação interna real

Assim como na representação seqüencial pré-ordem, uma vez que um valor RTAG sempre precede imediatamente uma seta, exceto para o último *node* da seqüência, uma estrutura alternativa é a substituição de LLINK com LTAG, que é determinado se uma seta provier dele:



LTAG	1	1	1	0	0	1	1	0	1	0	0	0	0	0
INFO	A	G	M	N	H	I	J	L	K	B	C	D	E	F
RTAG	0	0	1	1	0	0	1	1	1	1	0	0	0	1
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

Figura 14: Representação interna real

4.3.3. Representação Sequencial de Percurso por Nível (Level-Order)

A terceira opção de representação sequencial é o percurso por nível. Nesta representação, a floresta é atravessada por nível, por exemplo, de-cima-para-baixo, da-esquerda-para-direita, para obter a listagem de elementos por nível. Assim como no percurso por família, irmãos (os quais constituem uma família) são listados consecutivamente. LLINK e RTAG são usados na representação. **LLINK** é um ponteiro para o filho mais velho na floresta ou o filho da esquerda na sua representação em árvore binária. **RTAG** identifica o irmão mais novo na descendência (ou o último membro da família). Usando a representação sequencial de percurso por nível, temos o seguinte para a floresta F:

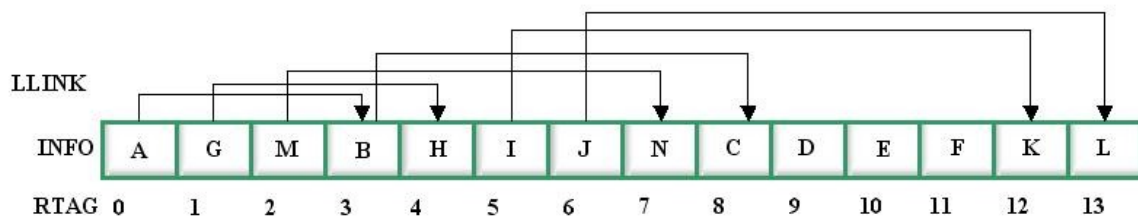


Figura 15: Representação sequencial de percurso por nível

Observe que ao contrário da pré-ordem ou do percurso por família, as setas se cruzam nesta representação. Todavia, é possível ser observado que a primeira cruz a começar é também a primeira a finalizar. Tendo a estrutura FIFO (*first-in, first-out*), uma *queue* poderia ser utilizada para estabelecer o relacionamento entre os *nodes*. Conseqüentemente, assim como nos métodos anteriores, poderia ser representada como:

LTAG	1	1	1	1	0	1	1	0	0	0	0	0	0	0
INFO	A	G	M	B	H	I	J	N	C	D	E	F	K	L
RTAG	0	0	1	1	0	0	1	1	0	0	0	1	1	1
	0	1	2	3	4	5	6	7	8	9	10	11	12	13

4.3.4. Convertendo Representação Sequencial para Representação por Link

Em termos de espaço, a representação sequencial é ideal para florestas. Todavia, uma vez que a representação por link é mais natural para florestas, existem instâncias em que teremos que utilizar esta última. A classe a seguir implementa um método para conversão de representação sequencial para a representação por link:

```
public class SeqForest{
    int RTAG[];
    int INFO[];
    int LTAG[];
    int n = 0;
```

```

public SeqForest(int size) {
    n = size;
    RTAG = new int[n];
    INFO = new int[n];
    LTAG = new int[n];
}
public SeqForest(int[] rtag, int[] info, int[] ltag) {
    n = rtag.length;
    RTAG = rtag;
    INFO = info;
    LTAG = rtag;
}
public BinaryTree convert() {
    BTNode alpha = new BTNode(null);
    BinaryTree t = new BinaryTree(alpha);
    LinkedStack stack = new LinkedStack();
    BTNode sigma;
    BTNode beta;

    // Gera o resto de uma árvore binária
    for (int i=0; i<n-1; i++) {
        // alpha.setInfo(INFO[i]);
        beta = new BTNode(null);
        if (RTAG[i] == 1)
            stack.push(alpha);
        else
            alpha.setRight(null);

        if (LTAG[i] == 1){
            alpha.setLeft(null);
            sigma = (BTNode) stack.pop();
            sigma.setRight(beta);
        } else
            alpha.setLeft(beta);

        alpha.setInfo(INFO[i]);
        alpha = beta;
    }
    // Preenche os campos do node mais a direita
    alpha.setInfo(INFO[n-1]);
    return t;
}
public static void main(String args[]) {
    int[] RTAG = {1,0,1,1,1,0,1,1,1,0,0,0,0};
    int[] INFO = {'A','B','C','D','E','F','G','H','I','K','J','L','M','O'};
    int[] LTAG = {0,0,1,1,1,1,0,1,0,1,0,1,0,1};
    SeqForest f = new SeqForest(RTAG, INFO, LTAG);
    BinaryTree b = f.convert();
    b.preorder();
}
}

```

5. Representações de Árvores Aritméticas

As árvores podem ser representadas seqüencialmente utilizando uma representação aritmética. A árvore pode ser armazenada seqüencialmente baseada em sua pré-ordem, pós-ordem e ordem de níveis de seus *nodes*. O grau ou o peso da árvore pode ser armazenado com sua informação. O **grau**, como definido anteriormente, se refere ao número de filhos que um *node* possui, enquanto que o **peso** se refere ao número de descendentes de um *node*.

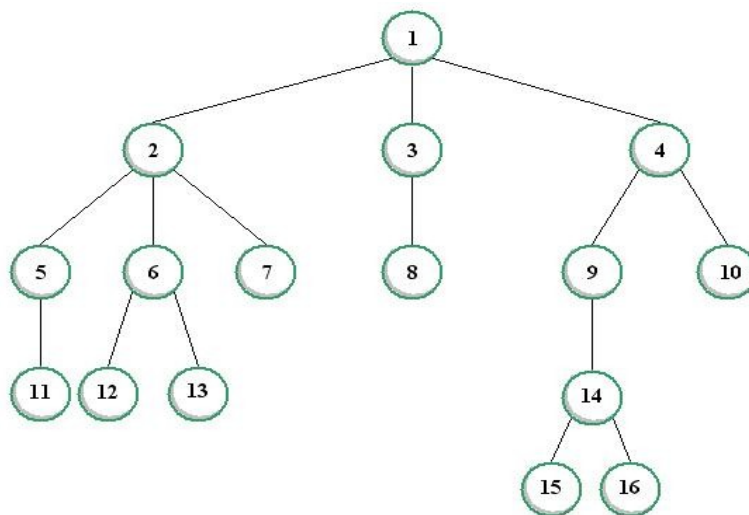


Figura 5.1 árvore T ordenada

A seguir são representadas as diversas maneiras de se representar a árvore acima.

Sequência de Pré-ordem com Grau

INFO	1	2	5	11	6	12	13	7	3	8	4	9	14	15	16	10
GRAU	3	3	1	0	2	0	0	0	1	0	2	1	2	0	0	0

Sequência de Pré-ordem com Peso

INFO	1	2	5	11	6	12	13	7	3	8	4	9	14	15	16	10
PESO	15	6	1	0	2	0	0	0	1	0	5	3	2	0	0	0

Sequência de Pós-ordem com Grau

INFO	11	5	12	13	6	7	2	8	3	15	16	14	9	10	4	1
GRAU	0	1	0	0	2	0	3	0	1	0	0	2	1	0	2	3

Sequência de Pós-ordem com Peso

INFO	11	5	12	13	6	7	2	8	3	15	16	14	9	10	4	1
PESO	0	1	0	0	2	0	6	0	1	0	0	2	3	0	5	15

Sequência de ordem de níveis com Grau

INFO	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
GRAU	3	3	1	2	1	2	0	0	1	0	0	0	0	2	0	0

Sequência de ordem de níveis com Peso

INFO	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
PESO	15	6	1	5	1	2	0	0	3	0	0	0	0	2	0	0

5.1. Aplicação: Árvores e o problema da Equivalência

O problema de equivalência é uma outra aplicação que faz uso de uma árvore internamente representada como uma árvore aritmética.

Uma relação de equivalência é uma relação entre os elementos de uma coleção de objetos S que satisfaçam as 3 propriedades para qualquer objeto x , y e z (não necessariamente distintos) em S :

- (a) Transitividade: se $x \equiv y$ e $y \equiv z$ então $x \equiv z$
- (b) Simetria: se $x \equiv y$ então $y \equiv x$
- (c) Reflexividade: $x \equiv x$

Exemplos de relações de equivalência são as relações "é igual a" ($=$) e a "similaridade" entre as árvores binárias.

5.1.1. O problema da Equivalência

Dados quaisquer pares de relações de equivalência na forma de $i \equiv j$ para qualquer i, j em S , determine se K é equivalente a i , para qualquer K, i pertence a S , com base nos pares dados. Para solucionar o problema utilizaremos o seguinte teorema:

Uma relação de equivalência particiona seu conjunto S em classes disjuntas, chamadas classes de equivalência, tal que dois elementos são equivalentes se e somente se eles pertencerem a mesma classe de equivalência.

Por exemplo, considere o conjunto $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13\}$. Suponha que as relações de equivalência definidas no conjunto são: $1 \equiv 10, 9 \equiv 12, 9 \equiv 3, 6 \equiv 10, 10 \equiv 12, 2 \equiv 5, 7 \equiv 8, 4 \equiv 11, 2 \equiv 13$ e $1 \equiv 9$. Agora, a pergunta, $10 \equiv 2$ é verdadeiro? $6 \equiv 12$ é verdadeiro? Para responder a essas perguntas precisamos criar as classes de equivalência.

Entrada	Classes de equivalência	Modificações
$1 \equiv 10$	$C1 = \{1, 10\}$	Criar uma nova classe($C1$) que contenha 1 e 10
$9 \equiv 12$	$C2 = \{9, 12\}$	Criar uma nova classe($C2$) que contenha 9 e 12
$9 \equiv 3$	$C2 = \{9, 12, 3\}$	Adicionar 3 a $C2$
$6 \equiv 10$	$C1 = \{1, 10, 6\}$	Adicionar 6 a $C1$
$10 \equiv 12$	$C2 = \{1, 10, 6, 9, 12, 3\}$	Juntar $C1$ e $C2$ dentro de $C2$, descartar $C1$
$2 \equiv 5$	$C3 = \{2, 5\}$	Criar uma nova classe($C3$) que contenha 2 e 5
$7 \equiv 8$	$C4 = \{7, 8\}$	Criar uma nova classe($C4$) que contenha 7 e 8
$4 \equiv 11$	$C5 = \{4, 11\}$	Criar uma nova classe($C5$) que contenha 4 e 11
$6 \equiv 13$	$C2 = \{1, 10, 6, 9, 12, 3, 13\}$	Adicionar 13 a $C2$
$1 \equiv 9$		Sem mudanças

Desde que 13 não tenha equivalências, as classes finais são:

$C2 = \{1, 10, 6, 9, 12, 3, 13\}$
 $C3 = \{2, 5\}$
 $C4 = \{7, 8\}$
 $C5 = \{4, 11\}$

E $10 \equiv 2$? Já que se encontram em classes diferentes, não são equivalentes.

E $6 \equiv 12$? Já que se ambos pertencem a **C2**, são equivalentes.

5.1.2. Implementação no Computador

Para implementar a solução para o problema da equivalência, é preciso um modo de representar as classes de equivalência. Precisa-se também de um modo de unir as classes de equivalência (a operação **union**) e determinar se 2 objetos pertencem a uma mesma classe de equivalência ou não (a operação **find**).

Para solucionar a primeira preocupação, as árvores podem ser usadas para representar as classes de equivalência, i.e., uma árvore representa a classe. Neste caso, configurando uma árvore, chamemos t_1 , como uma sub-árvore de uma outra árvore, chamemos t_2 , pode-se implementar a união de classes equivalentes. Também é possível informar se dois objetos pertencem a uma mesma classe ou não respondendo a uma simples questão, "os objetos possuem a mesma origem na árvore?"

Considere o seguinte:

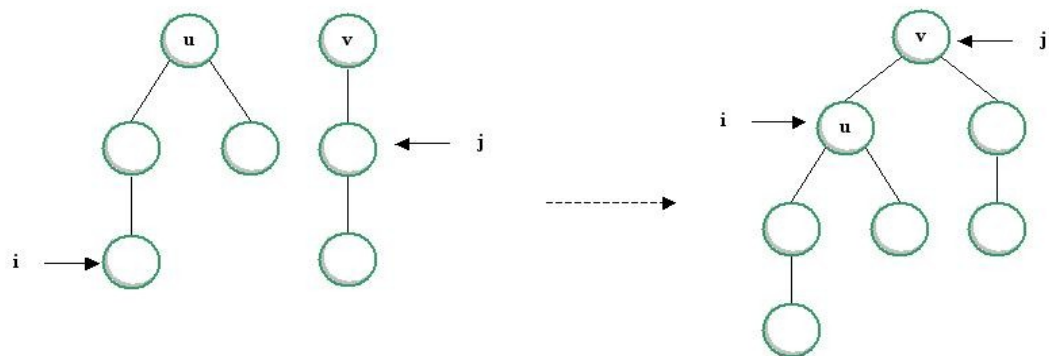


Figura 16: União de classes Equivalentes

São dadas as relações de equivalência $i \equiv j$ onde i e j são raízes. Para unir as duas classes, dizemos que a raiz de i é um novo *child* (tronco) da raiz de j . Este é o processo de união, e o código que o implementa é:

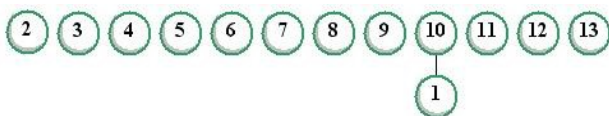
```
while (FATHER[i] > 0)
    i = FATHER[i];
while (FATHER[j] > 0)
    j = FATHER[j];
if (i != j)
    FATHER[j] = i;
```

Os exemplos seguintes mostram a ilustração gráfica do problema de equivalência descrito:

Entrada

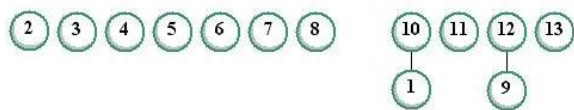
Floresta

1 \equiv 10



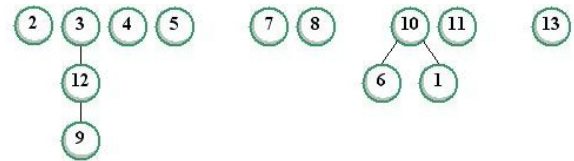
1	2	3	4	5	6	7	8	9	10	11	12	13
10	0	0	0	0	0	0	0	0	0	0	0	0

9 \equiv 12



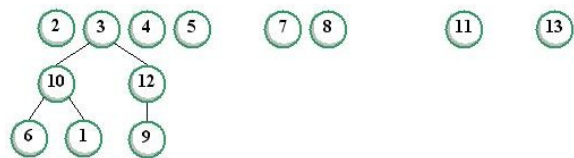
1	2	3	4	5	6	7	8	9	10	11	12	13
10	0	0	0	0	0	0	0	12	0	0	0	0

$9 \equiv 3$



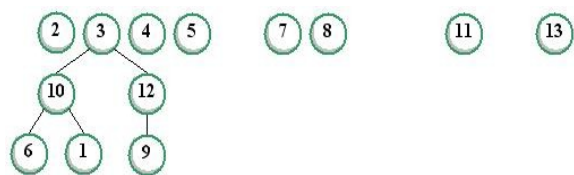
1	2	3	4	5	6	7	8	9	10	11	12	13
10	0	0	0	0	0	0	0	12	0	0	3	0

$6 \equiv 10$



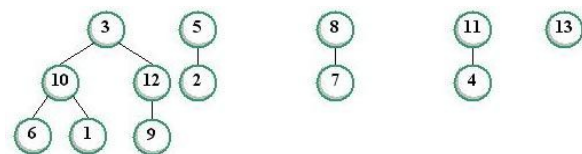
1	2	3	4	5	6	7	8	9	10	11	12	13
10	0	0	0	0	10	0	0	12	0	0	3	0

$10 \equiv 12$



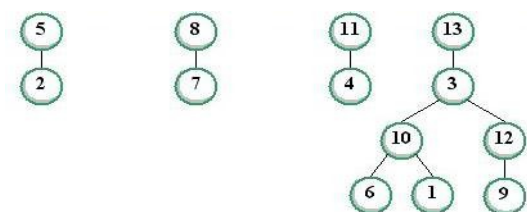
1	2	3	4	5	6	7	8	9	10	11	12	13
10	0	0	0	0	10	0	0	12	3	0	3	0

$2 \equiv 5$



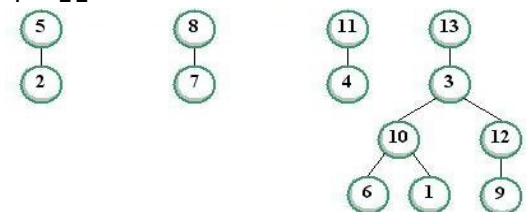
1	2	3	4	5	6	7	8	9	10	11	12	13
10	5	0	0	0	10	0	0	12	3	0	3	0

$7 \equiv 8$



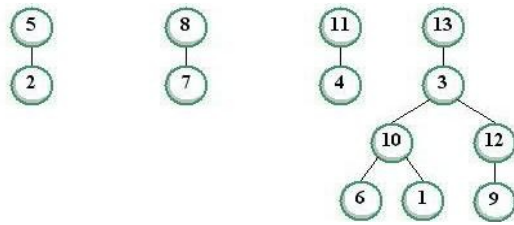
1	2	3	4	5	6	7	8	9	10	11	12	13
10	5	0	0	0	10	8	0	12	3	0	3	0

$4 \equiv 11$



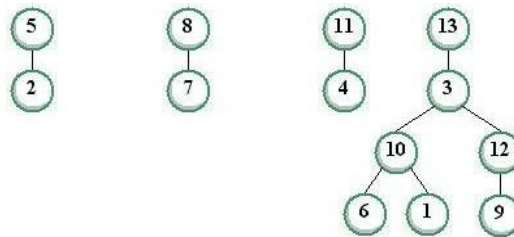
1	2	3	4	5	6	7	8	9	10	11	12	13
10	5	0	11	0	10	8	0	12	3	0	3	0

6 ≡ 13



1	2	3	4	5	6	7	8	9	10	11	12	13
10	5	13	11	0	10	8	0	12	3	0	3	0

1 ≡ 9



1	2	3	4	5	6	7	8	9	10	11	12	13
10	5	13	11	0	10	8	0	12	3	0	3	0

Figura 5.4 Exemplo de equivalência

A seguir a implementação do algoritmo para resolver o problema da equivalência:

```
class Equivalence {
    int[] FATHER;
    int n;

    public Equivalence() {
    }

    public Equivalence(int size) {
        n = size+1; // +1 desde FATHER[0] que não será usado
        FATHER = new int[n];
    }

    public void setSize(int size) {
        FATHER = new int[size+1];
    }

    // Gera a equivalência de classes baseada na equivalência dos pares j,k
    public void setEquivalence(int a[], int b[]) {
        int j, k;
        for (int i=0; i<a.length; i++) {
            // Obtêm a equivalência do par j,k
            j = a[i];
            k = b[i];

            // Pega a raiz de j e k
            while (FATHER[j] > 0)
                j = FATHER[j];
            while (FATHER[k] > 0)
                k = FATHER[k];

            // Se não é equivalente, combina as duas árvores
            if (j != k)
                FATHER[j] = k;
        }
    }
}
```

```

}
/* Aceita dois elementos j e k.
   Retorna verdadeiro se equivalente, senão retorna falso */
public boolean test(int j, int k) {
    // Obtém as raízes de j e k
    while (FATHER[j] > 0)
        j = FATHER[j];
    while (FATHER[k] > 0)
        k = FATHER[k];

    // Se possuírem a mesma raiz, são equivalentes
    return (j == k);
}

public static void main(String args[]){
    int[] j = {1, 9, 9, 6, 10, 2, 7, 4};
    int[] k = {10, 12, 3, 10, 12, 5, 8, 11};
    int n = 13;
    Equivalence eq = new Equivalence(n);
    eq.setEquivalence(j, k);
    System.out.println(eq.test(10, 2));
    System.out.println(eq.test(6, 12));
}
}

```

5.1.3. Degeneração e o Enfraquecimento da Regra para União

O problema com o algoritmo anterior similar ao problema resolvido pelo **enfraquecimento**, i.e., criando uma árvore que possua o maior número de níveis em profundidade possível, feito isso é criado um prejuízo de performance, i.e., tempo de complexidade para $O(n)$. Para esta ilustração, considere o conjunto $S = \{ 1, 2, 3, \dots, n \}$ e a relação de equivalência $1 \equiv 2, 1 \equiv 3, 1 \equiv 4, \dots, 1 \equiv n$. A seguinte figura mostra como a árvore é montada:

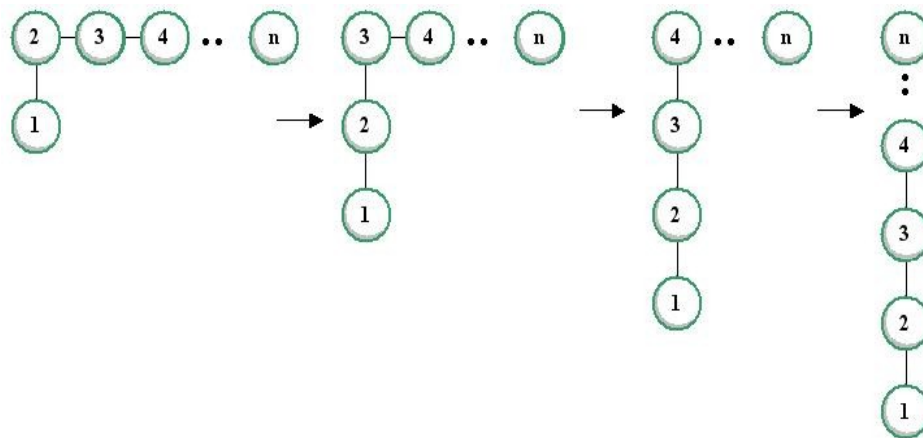


Figura 17: pior caso Floresta (Árvore) em um Problema Equivalente.

Agora, considere a seguinte árvore:

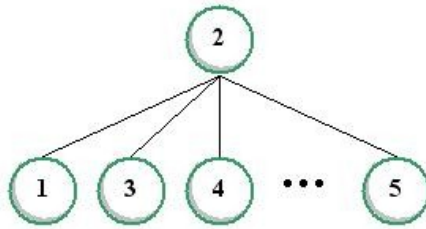


Figura 18: Melhor-caso Floresta (Árvore) em um Problema Equivalente

Em termos de classificações equivalentes, as duas árvores simbolizam a mesma classe. Contudo a segunda árvore possui apenas um ramo a ser atravessado de qualquer *node* à raiz enquanto que na primeira árvore possui ***n-1*** ramos.

Exemplificaremos com a seguinte classe:

```

public class Equivalence2 {
    private int[] FATHER;
    private int n;

    public Equivalence2() {
    }
    public Equivalence2(int size) {
        n = size+1;
        FATHER = new int[n];
    }
    public void setEquivalence(int a[], int b[]) {
        int j, k;
        for (int i=0; i<FATHER.length; i++) FATHER[i] = -1;
        for (int i=0; i<a.length; i++) {
            j = a[i];
            k = b[i];
            j = find(j);
            k = find(k);
            if (j != k) union(j, k);
        }
    }
}
  
```

Para resolver este problema, utilizaremos uma técnica conhecida como a **balanceamento por união**. Definida a seguir:

primeiro o *node i* e o *node j* são raízes. Se o número de *nodes* da árvore cuja raiz *i* for maior que o número de *nodes* com raiz *j*, faz-se o *node i* pai do *node j*; senão, faz o *node j* pai do *node i*.

No algoritmo, Um array *COUNT* pode ser usado para contar o número de *nodes* de cada árvore da floresta. Porém, se o *node* é raiz de classes equivalentes, eles não entram no array PAI não tem importância e a raiz não possui pai. Para entender a vantagem sobre isto, podemos usar esta abertura como array PAI no lugar de usar outro. Para diferenciar entre *contadores* e *rótulos* no PAI, um sinal de menos é adicionado aos contadores. O seguinte método adicionado na classe *Sequence2*, implementa o balanceamento por união:

```

// Implementa o balanceamento por união
public void union(int i, int j) {
    int count = FATHER[i] + FATHER[j];
    if (Math.abs(FATHER[i]) > Math.abs(FATHER[j])) {
  
```

```

    FATHER[j] = i;
    FATHER[i] = count;
  } else {
    FATHER[i] = j;
    FATHER[j] = count;
  }
}

```

A operação de UNIÃO possui tempo de complexidade $O(1)$. Se o balanceamento por união não for aplicado, a ordem é $O(n)$ pesquisa-união na operação de inserção, no pior caso, $O(n^2)$. Fora isso, se aplicado, o tempo de tempo de complexidade é $O(n \log_2 n)$.

5.1.4. Árvores Pior Caso

Outra observação em usar árvores em problemas semelhantes é mostrado para o pior caso até quando a criação *balanceamento por união* é aplicada. Considere a seguinte ilustração:

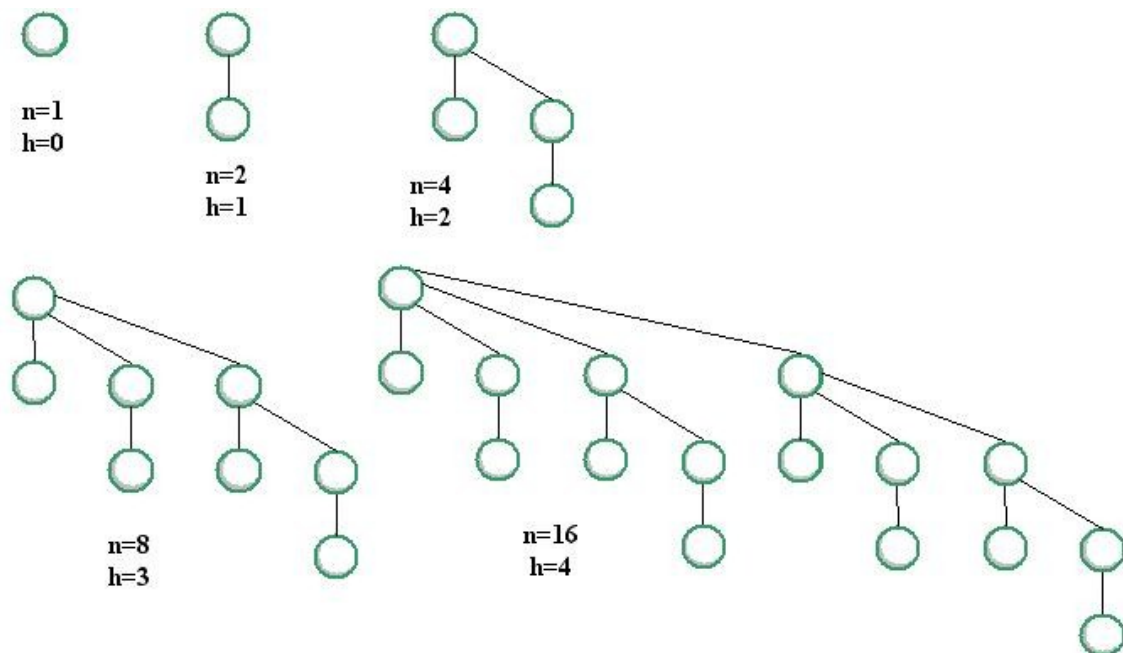


Figura 19: Árvores pior caso

A figura mostra como a árvore pode crescer logaritmicamente apesar do balanceamento por união ser aplicado. Isto é, o pior caso das árvores com n nodes é $\log_2 n$. podemos prevenir, uma árvore possui $n-1$ nodes filhos apenas um node pode ser usado para representar a classe de equivalência. Neste caso, a profundidade para a árvore de pior caso pode ser reduzido por aplicar outra técnica, e esta é a **desmontar por ordem de pesquisa**. Com este, o caminho pode ser feito buscando o caminho percorrido da raiz ao node **p**. Isto é, em um processo de busca, Se o atual caminho descoberto não for o ótimo, ele é "desmontada" para conseguir o ótimo. A ilustração anterior mostra isso, considere a figura seguinte:

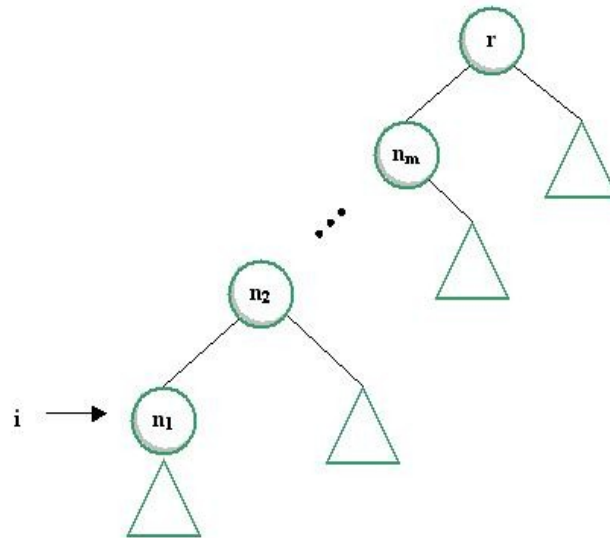


Figura 20: Desmontar por Ordem de Pesquisa 1

Em um relacionamento $i \equiv j$ para qualquer j , requer pelo menos m passos, i.e., executar o $i = \text{PAI}(i)$, para receber a raiz.

Na desmontar por ordem de pesquisa, descobrir n_1, n_2, \dots, n_m quais *nodes* estão no caminho entre o *node* n_1 ao *node* raiz r . para desmontar, faremos r o pai de n_p , $1 \leq p < m$:

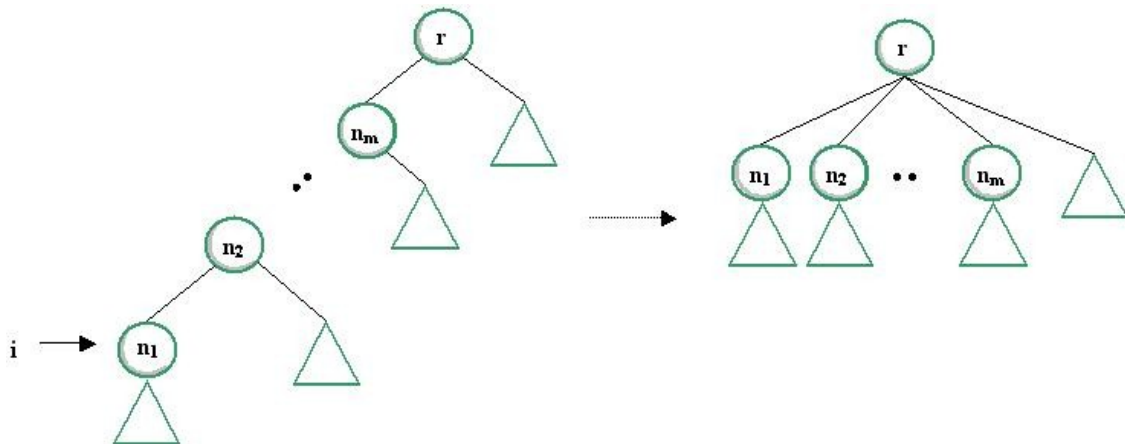


Figura 21: Desmontar por Ordem de Pesquisa 2

O seguinte método adicionado na classe *Sequence2*, implementa este processo:

```
// Implementa Desmontar por Ordem de Pesquisa. Retorna a raiz de i
public int find(int i) {
    int j, k, l;
    k = i;

    // Procura raiz
    while (FATHER[k] > 0)
        k = FATHER[k];

    // Resumir caminho do node i
    j = i;
    while (j != k){
```

```

        l = FATHER[j];
        FATHER[j] = k;
        j = l;
    }
    return k;
}

```

A operação **find** é proporcional ao tamanho do caminho do *node i* para a raiz.

5.1.5. Solução final para o Problema Equivalente

Os métodos a seguir implementados na classe *Sequence2*, finalizam a solução para o problema equivalente:

```

// Gerar equivalentes classes baseada na equivalência do par j,k
public void setEquivalence(int a[], int b[]){
    int j, k;
    for (int i=0; i < FATHER.length; i++)
        FATHER[i] = -1;
    for (int i=0; i < a.length; i++) {
        // Retorna a equivalência entre o par j,k
        j = a[i];
        k = b[i];

        // Retorna as raízes de j e k
        j = find(j);
        k = find(k);

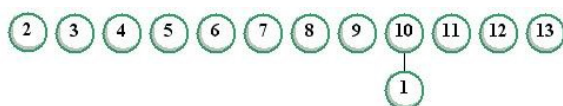
        // Se não são equivalentes, junte as duas árvores
        if (j != k)
            union(j, k);
    }
}

/* Aceitar dois elementos j e k.
   Retorna verdadeiro se forem equivalentes, senão retorna falso*/
public boolean test(int j, int k) {
    // retorna raízes para j e k
    j = find(j);
    k = find(k);

    // Se possuírem a mesma raiz, são equivalentes
    return (j == k);
}

```

A seguir é mostrado o estado de classes equivalentes após esta solução final de equivalência o problema é resolvido:

Entrada	Floresta	PAI																										
1 ≡ 10		<table><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td></tr><tr><td>10</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	12	13	10	0	0	0	0	0	0	0	0	0	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13																
10	0	0	0	0	0	0	0	0	0	0	0	0																

Entrada	Floresta	PAI																										
9 ≡ 12		<table><tr><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th></tr><tr><td>10</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>12</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	12	13	10	0	0	0	0	0	0	0	12	0	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13																
10	0	0	0	0	0	0	0	12	0	0	0	0																
9 ≡ 3, balanceando count(12) > count(3)		<table><tr><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th></tr><tr><td>10</td><td>0</td><td>12</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>12</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	12	13	10	0	12	0	0	0	0	0	12	0	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13																
10	0	12	0	0	0	0	0	12	0	0	0	0																
6 ≡ 10		<table><tr><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th></tr><tr><td>10</td><td>0</td><td>12</td><td>0</td><td>0</td><td>10</td><td>0</td><td>0</td><td>12</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	12	13	10	0	12	0	0	10	0	0	12	0	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13																
10	0	12	0	0	10	0	0	12	0	0	0	0																
10 ≡ 12, count(10) = count(12)		<table><tr><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th></tr><tr><td>12</td><td>0</td><td>12</td><td>0</td><td>0</td><td>12</td><td>0</td><td>0</td><td>12</td><td>12</td><td>0</td><td>0</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	12	13	12	0	12	0	0	12	0	0	12	12	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13																
12	0	12	0	0	12	0	0	12	12	0	0	0																
2 ≡ 5		<table><tr><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th></tr><tr><td>10</td><td>5</td><td>12</td><td>0</td><td>0</td><td>12</td><td>0</td><td>0</td><td>12</td><td>12</td><td>0</td><td>0</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	12	13	10	5	12	0	0	12	0	0	12	12	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13																
10	5	12	0	0	12	0	0	12	12	0	0	0																
7 ≡ 8		<table><tr><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th></tr><tr><td>10</td><td>5</td><td>12</td><td>0</td><td>0</td><td>12</td><td>8</td><td>0</td><td>12</td><td>12</td><td>0</td><td>0</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	12	13	10	5	12	0	0	12	8	0	12	12	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13																
10	5	12	0	0	12	8	0	12	12	0	0	0																
4 ≡ 11		<table><tr><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th><th>11</th><th>12</th><th>13</th></tr><tr><td>12</td><td>5</td><td>12</td><td>11</td><td>0</td><td>12</td><td>8</td><td>0</td><td>12</td><td>12</td><td>0</td><td>0</td><td>0</td></tr></table>	1	2	3	4	5	6	7	8	9	10	11	12	13	12	5	12	11	0	12	8	0	12	12	0	0	0
1	2	3	4	5	6	7	8	9	10	11	12	13																
12	5	12	11	0	12	8	0	12	12	0	0	0																

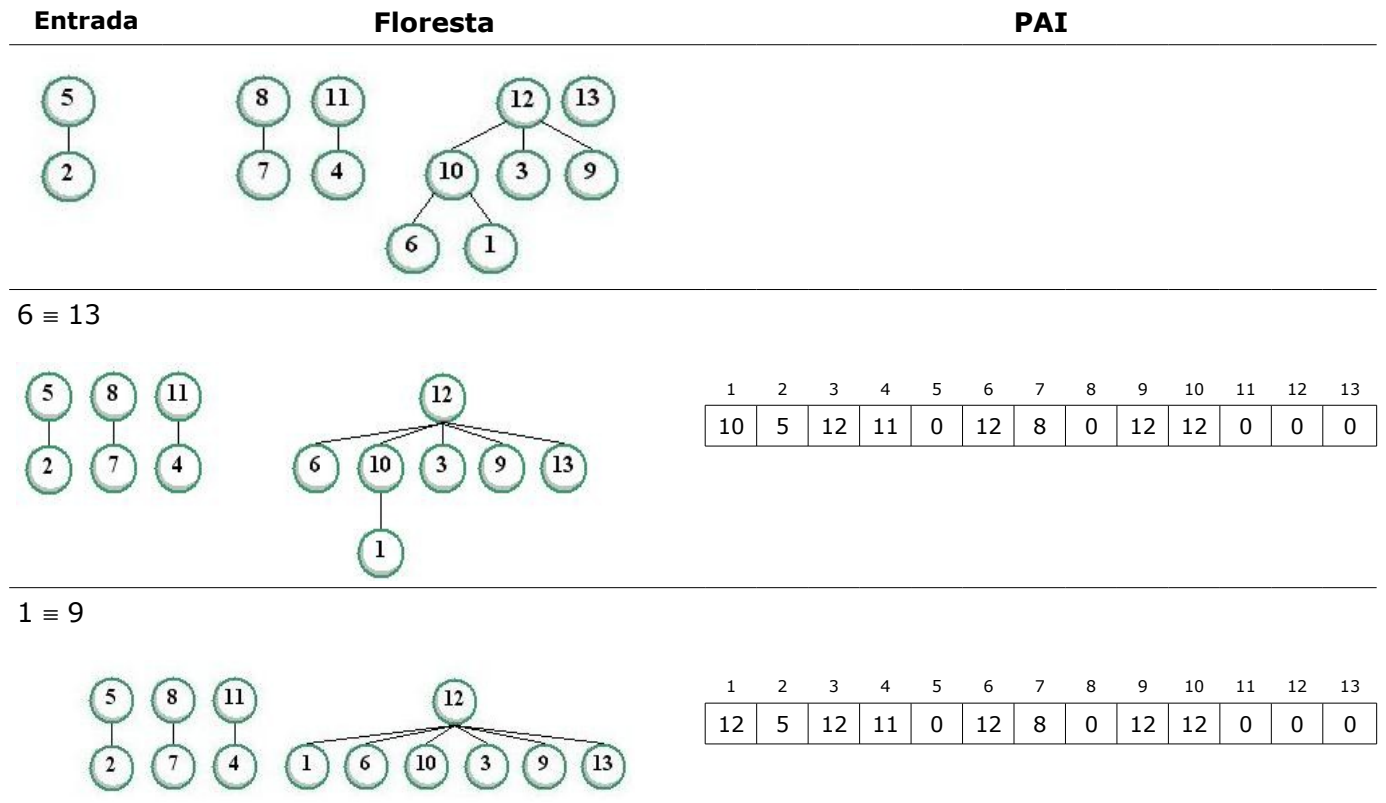


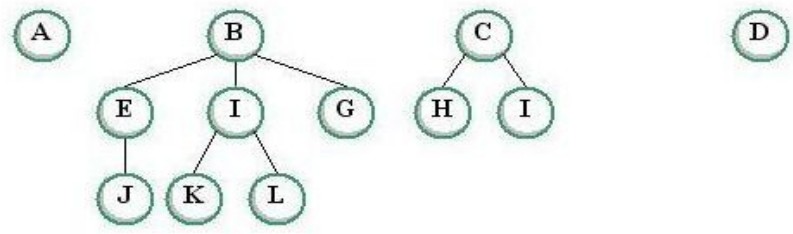
Figura 22: Um Exemplo Usando Balanceamento por União e Desmontar por Ordem de Pesquisa

Podemos utilizar o seguinte método principal para testar esta classe:

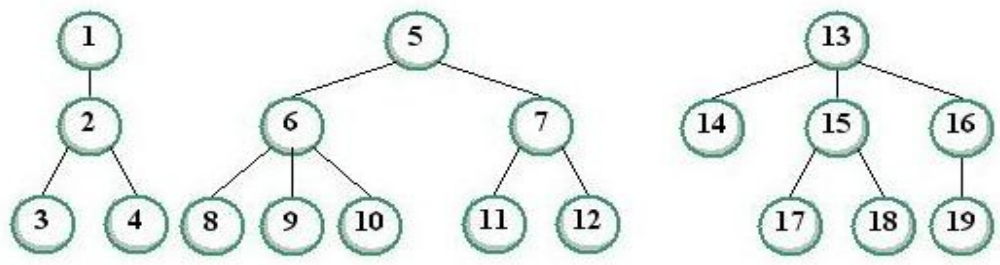
```
public static void main(String args[]){
    int[] j = {1, 9, 9, 6, 10, 2, 7, 4, 6, 1};
    int[] k = {10, 12, 3, 10, 12, 5, 8, 11, 13, 9};
    int n = 13;
    Equivalence2 eq = new Equivalence2(n);
    eq.setEquivalence(j, k);
    System.out.println(eq.test(10, 2));
    System.out.println(eq.test(1, 3));
    System.out.println(eq.test(6, 12));
    for (int i=0; i<n;i++) System.out.println(eq.FATHER[i]);
}
```

6. Exercícios

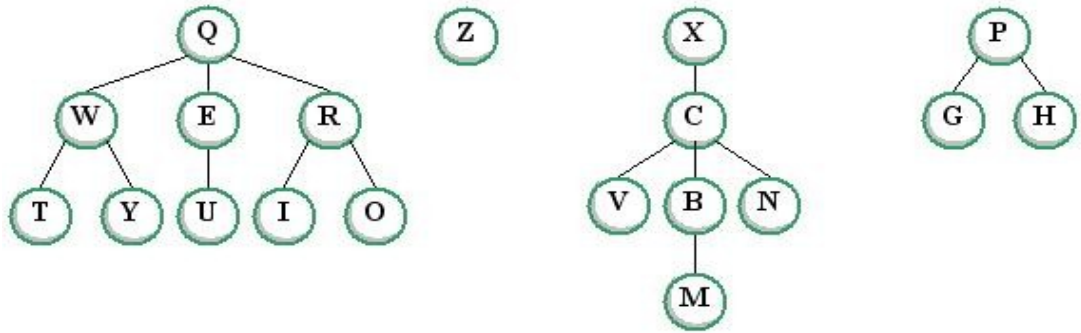
1. Para cada uma das florestas abaixo,



FLORESTA 1



FLORESTA 2



FLORESTA 3

- a) Converta para a árvore binária correspondente.
- b) Dê a representação seqüencial em pré-ordem com pesos.
- c) Dê a representação seqüencial em ordem de família com graus.
- d) Usando representação seqüencial em pré-ordem, mostre a disposição interna usada para armazenar a floresta (com seqüências LTAG e RTAG).

2. Mostre a representação da floresta abaixo usando a representação seqüencial em pré-ordem com pesos:

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
5	2	0	0	1	0	0	2	1	0	4	1	0	0	0

3. Classes de Equivalência

- a) Dado o conjunto $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ e os pares de equivalência: $(1,4)$, $(5,8)$, $(1,9)$, $(5,6)$, $(4,10)$, $(6,9)$, $(3,7)$ e $(3,10)$, construir a classe de equivalência usando florestas. $7 \equiv 6$? $9 \equiv 10$?
- b) Desenhe a floresta correspondente e o array pai resultante das classes equivalentes obtidas dos elementos de $S = \{1,2,3,4,5,6,7,8,9,10,11,12\}$ na base das relações equivalentes $1 \equiv 2$, $3 \equiv 5$, $5 \equiv 7$, $9 \equiv 10$, $11 \equiv 12$, $2 \equiv 5$, $8 \equiv 4$, e $4 \equiv 6$. Use a regra dos pesos para a união.

6.1. Exercícios para Programar

1. Criar a definição da classe de Java de representações seqüenciais em ordem de nível de florestas. Criar também um método que converta a representação seqüencial em ordem de nível em sua representação de ponteiros.

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.