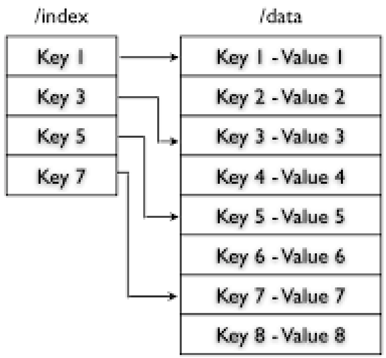
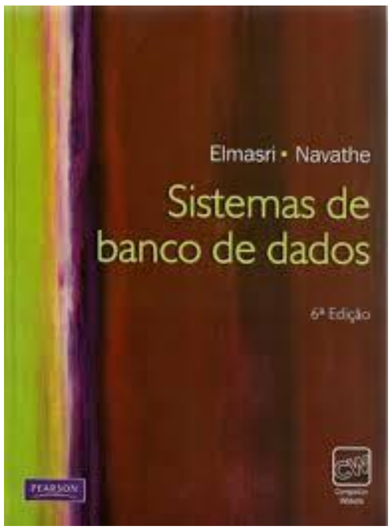
Unidade 10 – Estruturas de Indexação em Banco de Dados

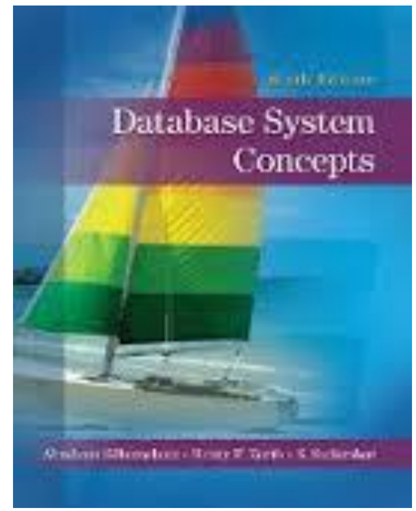


Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP

Bibliografia



Sistemas de Banco de Dados
Elmasri / Navathe 6ª edição



Sistema de Banco de Dados
Korth, Silberschatz – Sixth Editon

Introdução

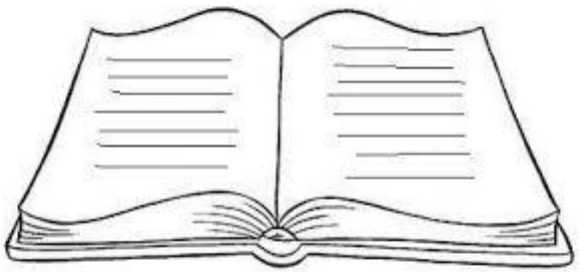
- **Índices** são estruturas de acesso auxiliares, utilizadas para agilizar a recuperação de registros em resposta a certas condições de pesquisa;

- São arquivos **adicionais** que oferecem formas alternativas de acesso aos dados (caminhos de acesso) sem afetar seu posicionamento físico.

Índices

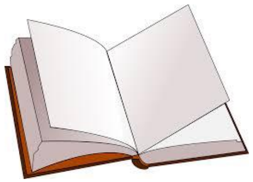
- São construídos com base em qualquer campo do arquivo. Esses campos são chamados de **campos de indexação**;
- Num mesmo arquivo podem-se criar índices em múltiplos campos. Isso significa que um arquivo pode ter vários **índices**;
- Para se encontrar um registro, pesquisa-se o **índice**. O índice retorna o **endereço** do bloco do disco primário onde o registro está localizado;
- Índices usualmente baseiam-se em **arquivos ordenados** (índices de um único nível) e estruturas de dados em **árvores** (índices multi-nível, B^+-trees);
- Índices também podem ser construídos com base em **Hashing**.

Índices ordenados de único nível



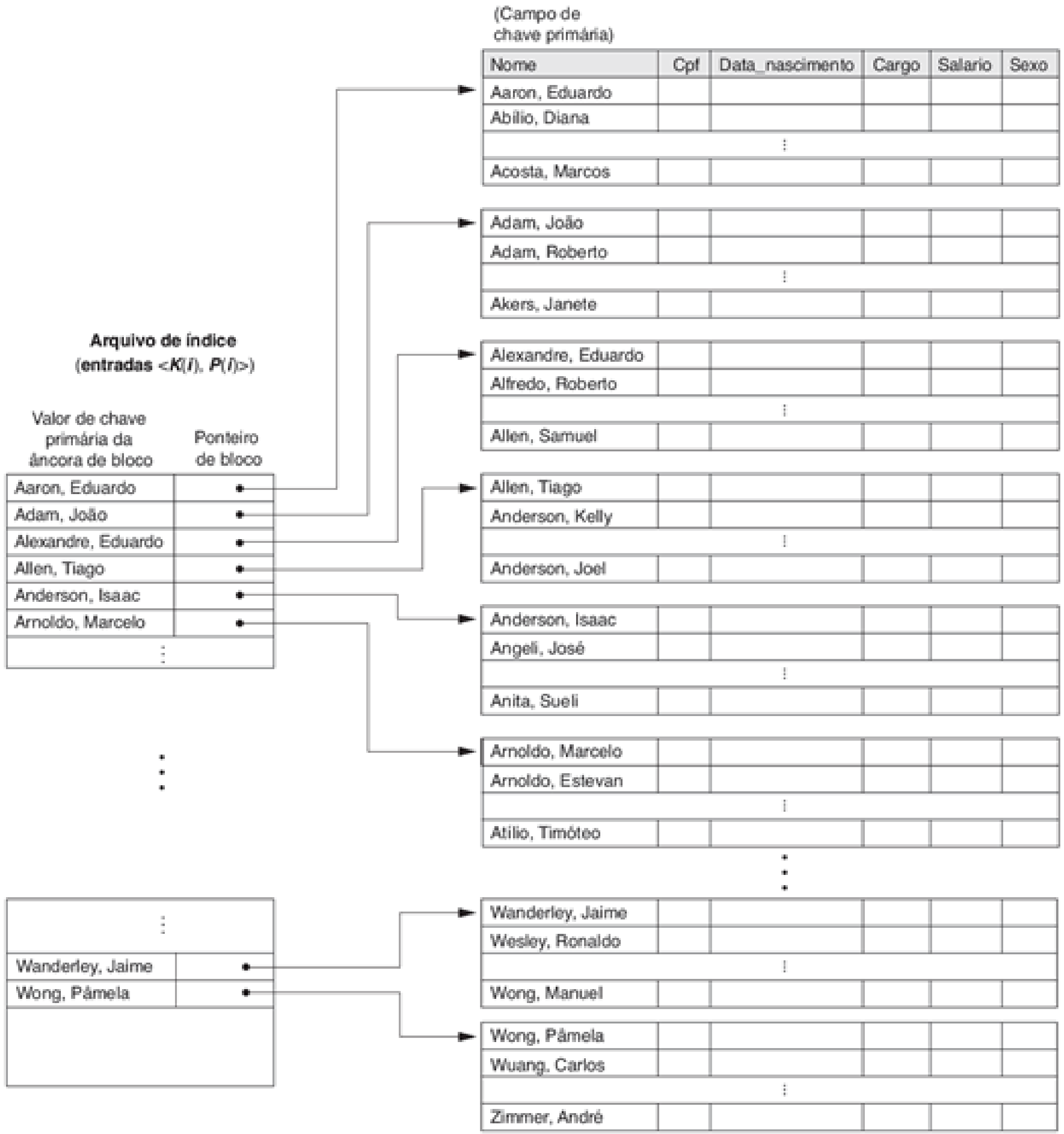
- Procedimento semelhante ao índice usado em um **livro**;
- Por meio da pesquisa no índice de um livro, obtém-se o endereço do termo desejado (número da página) e em seguida procura-se o termo na página citada.
- O índice costuma armazenar cada valor do campo de índice junto com uma **lista de ponteiros** para todos os blocos de disco que contêm registros com esse valor de campo.
- Os valores no índice são ordenados de modo que se possa realizar uma **pesquisa binária**.

Índices Primários



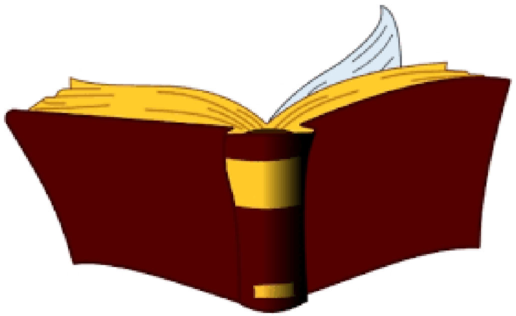
- O campo de índice é o campo chave de um arquivo ordenado de registros;
- Índice primário é um arquivo ordenado cujos registros são de tamanho fixo com dois campos: **campo de índice** e **ponteiro para um bloco de disco** (endereço de bloco).
- Existe **uma** entrada de índice no índice primário para cada **bloco** no arquivo de dados.
- Cada entrada de índice tem o valor do campo de chave primária para o primeiro registro em um bloco e um ponteiro para esse bloco.
- O número total de entradas no índice é igual ao número de blocos de disco no arquivo de dados ordenado. O primeiro registro em cada bloco do arquivo é chamado de REGISTRO DE ÂNCORA do bloco.
- Um índice primário é um **índice esparso** (não denso), pois inclui uma entrada para cada bloco de disco do arquivo.

Índices Primários



Índices Primários

- O arquivo de índice para um índice primário ocupa um espaço muito menor do que o arquivo de dados, por dois motivos:
  - **Primeiro**, existem menos entradas de índice do que registro no arquivo de dados;
  - **Segundo**, cada entrada de índice normalmente é menor em tamanho que um registro de dados, pois tem apenas dois campos: **chave** e **ponteiro** para o bloco de dados.
- Portanto, uma pesquisa binária no arquivo de índice requer menos acessos de bloco do que uma pesquisa binária no arquivo de dados.



Exemplo 0 – Arquivo ordenado sem índice

- Suponha um arquivo **ordenado** com **r** = 30.000 registros armazenados em um disco com tamanho de bloco **B** = 1024 bytes. Os registros de arquivo são de tamanho fixo e não espalhados, com tamanho de registro **R** = 100 bytes.

- O fator de bloco para o registro é **bfr** = |_ B/R _| = 1024/100 = 10 registros por bloco;

- O número de blocos necessários para o arquivo é r/bfr = 30.000 / 10 = 3.000 blocos.

- Uma pesquisa binária no arquivo de dados precisaria de aproximadamente:

  $\log_2 b = \log_2 3.000 =$ **12** acessos de bloco.

Exemplo 1 – Arquivo ordenado com índice primário

- Suponha, no exemplo anterior, que o campo **chave** de ordenação do arquivo seja V = 9 bytes de extensão, um **ponteiro** de bloco seja P = 6 bytes de extensão e se tenha construído um **índice primário** para o arquivo.

- O tamanho de cada entrada de índice é **Ri** = (9+6) = 15 bytes, de modo que o fator de bloco para o arquivo de índice é: **bfri** = |_ B/Ri _| = 1024 / 15 = **68** entradas por bloco.

- O número total de entradas de índice ri é igual ao número de blocos no arquivo de dados, que é 3000, ou seja o índice tem 3000 registros. (o arquivo de dados tem 30.000 registros).

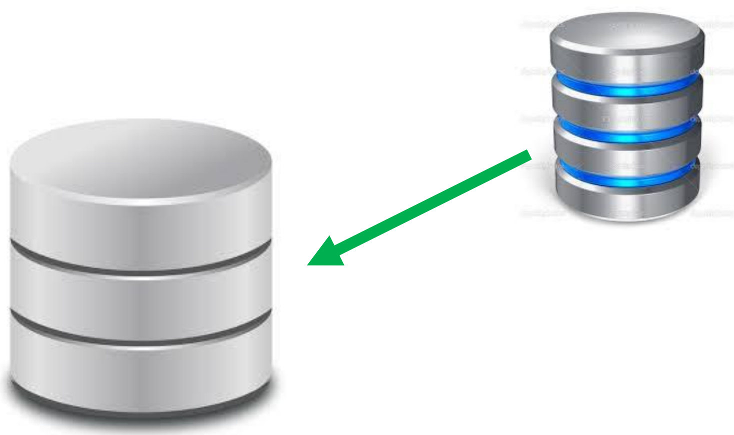
- O número de blocos no arquivo de índice é: **bi** = ri / bfri = 3000 / 68 = **45** blocos.

- Para realizar uma pesquisa binária no arquivo de índice, seriam necessários $\log_2 45$ = **6** acessos de bloco.

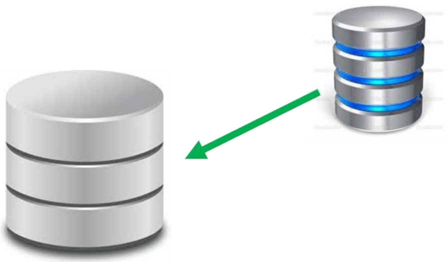
- Para procurar um registro usando o índice, precisa-se de um acesso de bloco adicional ao arquivo de dados, resultando um total de 6 + 1 = **7** acessos de bloco, uma melhoria em relação à pesquisa binária no arquivo de dados, que exigiu **12** acessos a bloco de disco.

Manutenção de dados com Índices Primários

- Com índices primários, aumenta-se o problema de **inserção** e **exclusão** de registros de dados, pois a movimentação de registros pode ocasionar mudança nos registros de âncora de alguns blocos.



Índices Secundários



- O índice secundário pode ser criado em um campo que é uma **chave candidata** e tem um valor **único** em cada registro, ou em um campo **não chave** com valores **duplicados**.
- O arquivo de dados pode ser um arquivo **ordenado**, **desordenado** ou **hashed**.
- O índice é novamente um arquivo ordenado com dois campos:
- O primeiro campo é do mesmo tipo de dado de algum campo não ordenado do arquivo de dados que seja um campo de índice.
- O segundo campo é um ponteiro de bloco ou um ponteiro de registro.
- Muitos índices secundários podem ser criados para o mesmo arquivo de dados – cada um representa um meio adicional de acessar esse arquivo com base em algum campo específico.

Índices Secundários com chaves candidatas

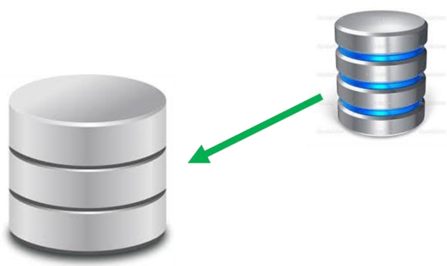
- O índice é construído com base em um campo de chave (único) que tem um valor distinto para cada registro.

- Tal campo é às vezes chamado de Chave Secundária.

- No modelo relacional, isso corresponderia a qualquer atributo de chave **UNIQUE** ou ao atributo de chave primária da tabela.

- Nesse caso, **existe uma entrada de índice para cada registro no arquivo de dados**, que contém o valor do campo para o registro e um ponteiro para o bloco em que o registro está armazenado ou para o próprio registro.

- Assim, tal índice é **denso**.



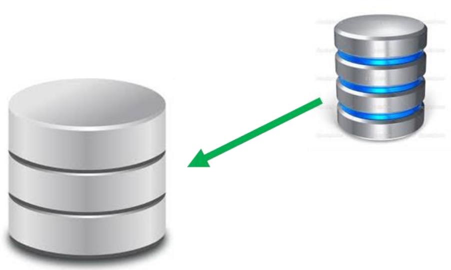
Índices Secundários com chaves candidatas

- As entradas de índice são ordenadas pelo valor de chave K, de modo que se pode realizar uma pesquisa binária.

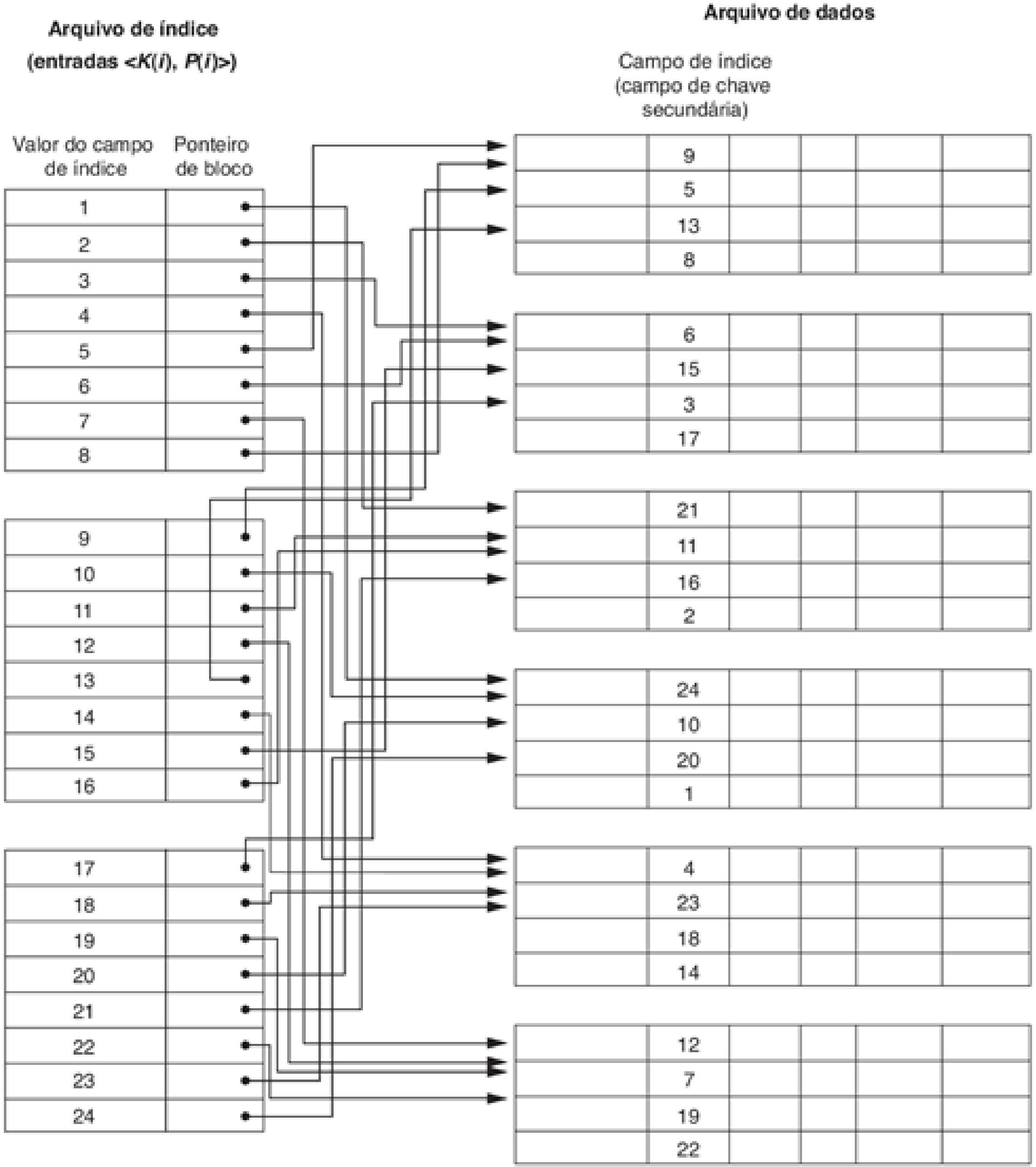
- Como os registros de dados não estão ordenados pelo valor de chave secundária, não se pode utilizar **âncoras de bloco**.

- É por isso que uma entrada de índice é criada para cada registro no arquivo de dados.

- Um índice secundário, em geral, **precisa de mais espaço de armazenamento** e tempo de busca maior que um índice primário, devido ao seu **maior número de entradas**.



Índices Secundários com chaves candidatas



Exemplo 2 – Índice Secundário com Chave Candidata

- Suponha um arquivo **ordenado** com **r** = 30.000 registros armazenados em um disco com tamanho de bloco **B** = 1024 bytes. Os registros de arquivo são de tamanho fixo e não espalhados, com tamanho de registro **R** = 100 bytes. O arquivo tem 3.000 blocos.

- Suponha que queiramos procurar um registro com um valor específico para a chave secundária – um campo de chave não ordenado do arquivo que tem V = 9 bytes de extensão.

- Sem o índice secundário, para fazer uma pesquisa linear no arquivo, seriam necessário – em média – b/2 = 3.000 /2 = 1.500 acessos de bloco na média.

Exemplo 2 – Índice Secundário com Chave Candidata

- Suponha que construíssemos um índice secundário nesse campo de chave não ordenado do arquivo.
- Como no exemplo anterior, um ponteiro de bloco tem P = 6 bytes de extensão, de modo que cada entrada de índice tem Ri = (9 + 6) = 15 bytes, e o fator de bloco para o índice é bfri = 1.024/15 = 68 entradas por bloco.
- Em um índice secundário denso como esse, o número total de entradas de índice ri é igual ao número de registros no arquivo de dados, que é igual a 30.000.
- O número de blocos necessários para o índice é, portanto, bi = ri / bfri = 3.000/68 = 442 blocos.

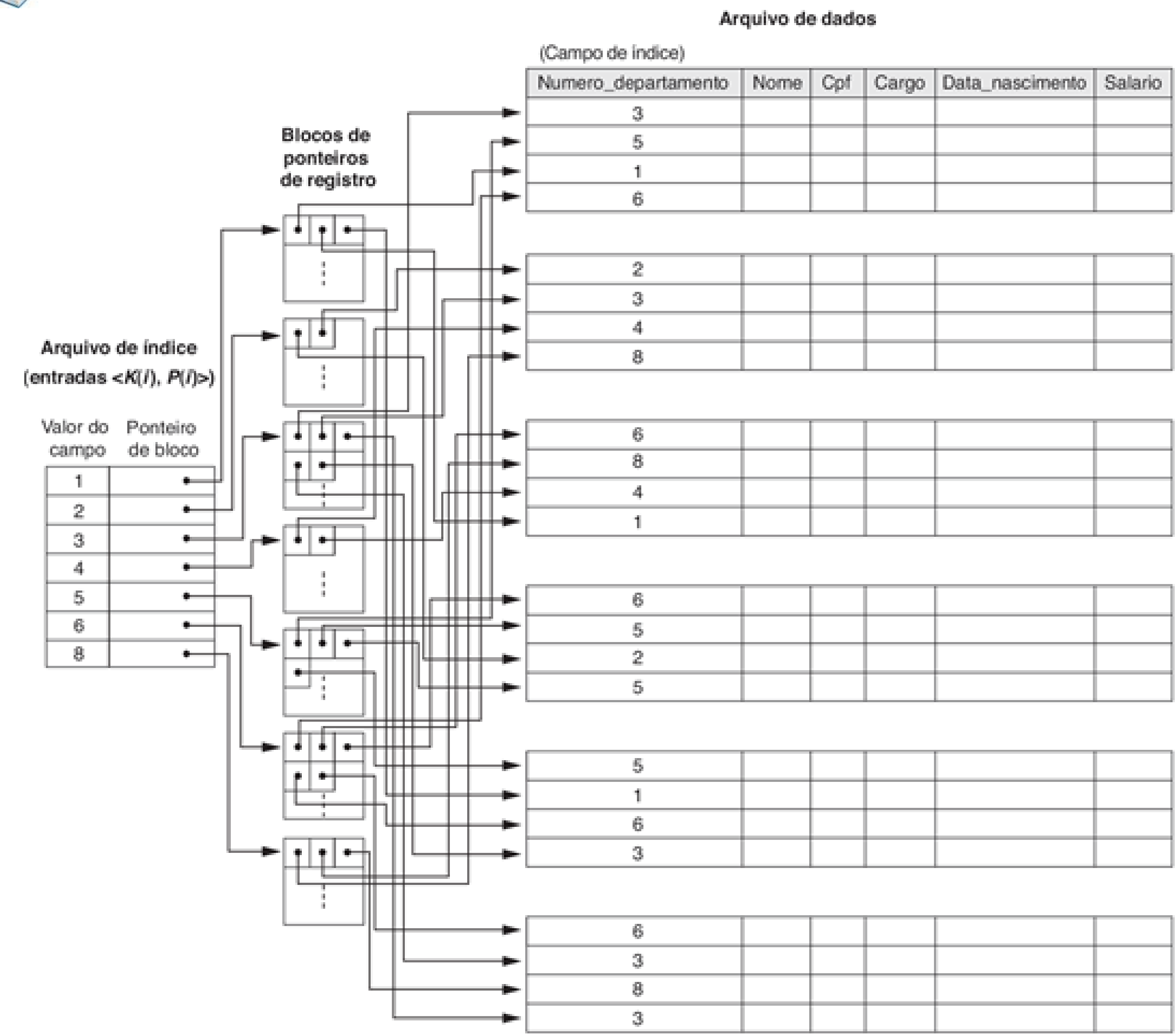
Exemplo 2 – Índice Secundário com Chave Candidata

- Uma pesquisa binária nesse índice secundário precisa de $\log_2 bi = \log_2 442 = 9$ acessos de bloco.
- Para procurar um registro usando o índice, precisamos de um acesso de bloco adicional ao arquivo de dados para um total de 9 + 1 = 10 acessos de bloco - uma grande melhoria em relação aos 1.500 acessos de bloco necessários – em média – para um pesquisa linear.

Índices Secundários em campos não chave

- Pode-se também criar índices secundários em um campo não chave, não ordenado de um arquivo de dados.
- Nesse caso, diversos registros no arquivo de dados podem ter o mesmo valor para o campo de índice.
- **Cria-se um nível de indireção extra** para lidar com os múltiplos ponteiros.

Índices Secundários em campos não chave



Índices Multiníveis

- Com índices ordenados de um nível, operações de searching podem ser aplicadas por meio de uma pesquisa binária que requer aproximadamente log $_2$ bi acessos de bloco para um índice com b_i blocos, pois cada etapa do algoritmo reduz a parte do arquivo de índice que se continua a pesquisa por um fator de **2**.

- Com **índices multinível**, pode-se reduzir a parte do índice que se continua a pesquisar por bfr_i, o fator de bloco para o índice, que é muito maior que 2.

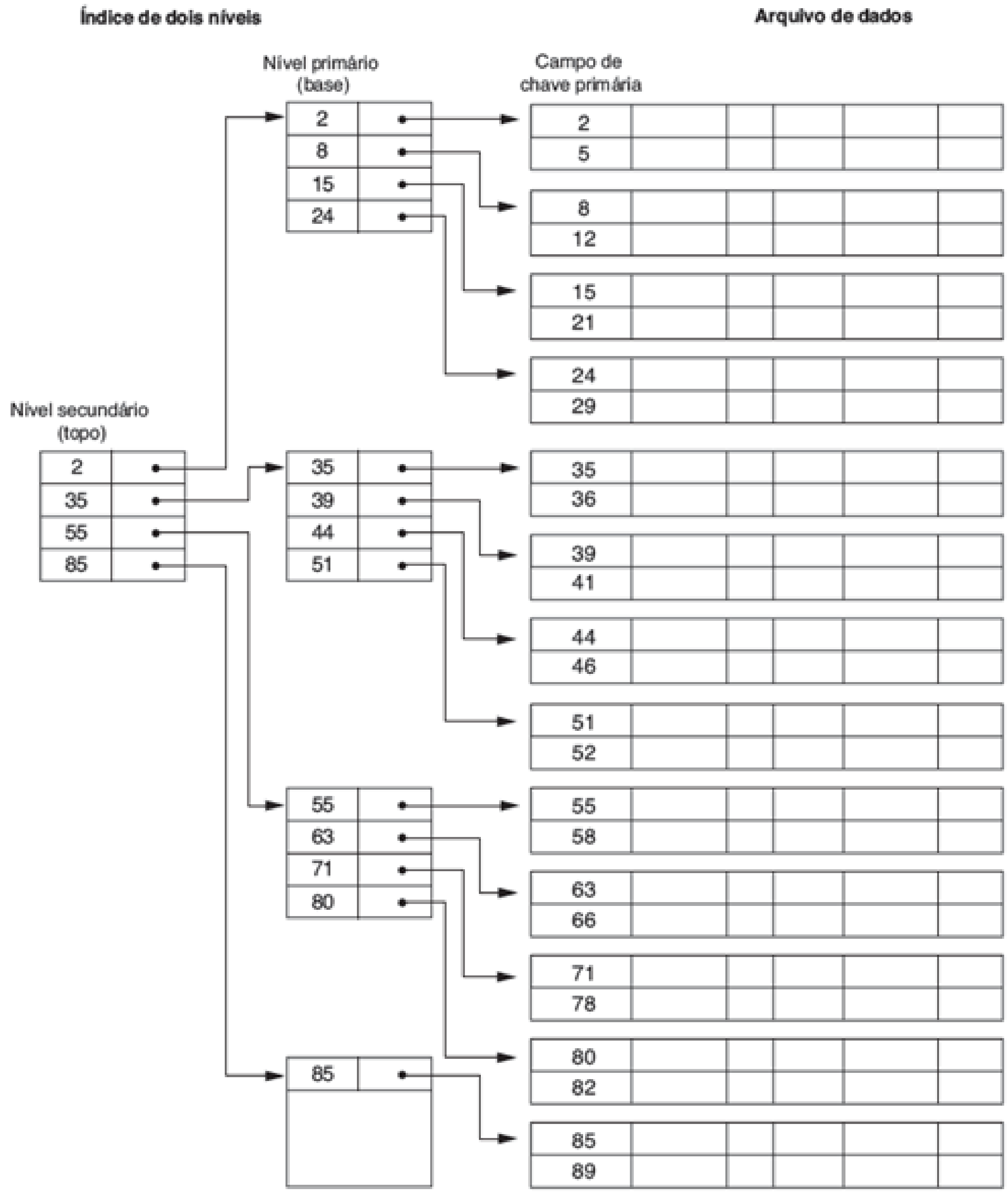
- Logo, com índices multinível, o espaço de pesquisa é reduzido muito mais rapidamente.

- O valor **bfr_i** é chamado **fan-out** do índice multinível e simbolizado por **f_0**.

- Na pesquisa binária, o espaço de pesquisa de registro é dividido em duas metades, enquanto que com índices multinível o dividimos **n** vezes (onde **n** é o **fan-out**).

- A pesquisa em um índice multinível requer aproximadamente ($\log_{f0} b_i$) que é substancialmente menor do que comparado a uma pesquisa binária.

Índices Multiníveis



Exemplo 3 – Índice Multinível

- Suponha que o índice secundário denso do Exemplo 2 seja convertido em um índice multinível;
- O fator de bloco de índice bfri = 68 entradas de índice por bloco, que é também o fan-out fo para o índice multinível;
- O número de blocos do primeiro nível b1 = 442 também já foi calculado;
- O número de blocos de segundo nível será b2 = b1/f0 = 442/68 = 7 blocos;
- O número de blocos de terceiro nível será b2/f0 = 7 /68 = 1 bloco.
- Logo, o terceiro nível é o **nível topo** do índice e t = 3.
- Para acessar um registro qualquer, deve-se acessar um bloco em cada nível mais um bloco do arquivo de dados, de modo que precisaríamos t + 1 = 3 + 1 = **4 acessos de bloco**.
- No exemplo 2, foram necessários **10** acessos a bloco. (redução portanto de **10** p/ **4** acessos de bloco).

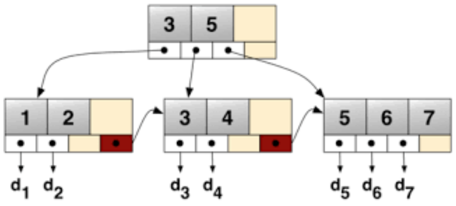
Índice Multinível - Observações

- Com índices multinível, reduz-se o número de blocos acessados quando se pesquisa um registro, dado seu valor de campo de indexação.

- Ainda se enfrenta problemas ao se lidar com inserções e exclusões de índice, pois todos os níveis de índice são arquivos ordenados fisicamente.

- Pode-se adotar um índice multinível chamado índice multinível dinâmico, que deixa algum espaço em cada um de seus blocos para inserir novas entradas e usa algoritmos apropriados de inserção/exclusão para criar e excluir novos blocos de índice quando o arquivo de dados cresce e encolhe. Esses esquemas são geralmente implementados por meio de estruturas de dados chamadas **B-trees** e **B$^+$-trees**.

- B-trees e B+-trees são casos especiais da famosa estrutura de dados de pesquisa, conhecida por **árvore**.

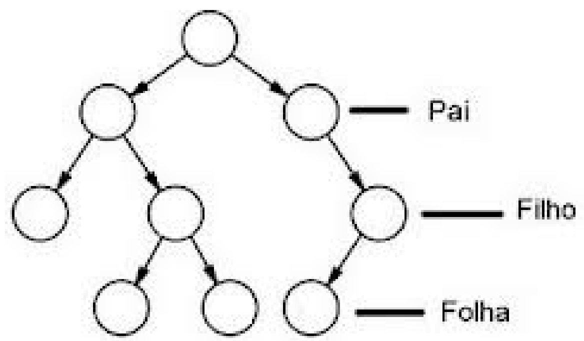


Árvores

- Árvore é uma estrutura de dados não-linear.
- Tem uma importância muito grande na Computação, pois disponibiliza algoritmos muito mais rápidos que os encontrados nas estruturas lineares.
- Têm diversas aplicações: sistemas de arquivos, interfaces gráficas, banco de dados, etc.
- Os relacionamentos encontrados em uma árvore são hierárquicos.
- Exemplo: Árvore Genealógica

Definição

- Árvore é um tipo abstrato de dados onde os dados são estruturados de forma hierárquica.
- Com exceção do topo, cada elemento da árvore tem um elemento pai e zero ou mais elementos filhos.
- Normalmente, o elemento topo é chamado **raiz** da árvore



Definição Formal

- Uma árvore **T** é um conjunto de nós que armazenam elementos em relacionamentos pai-filho com as seguintes propriedades:

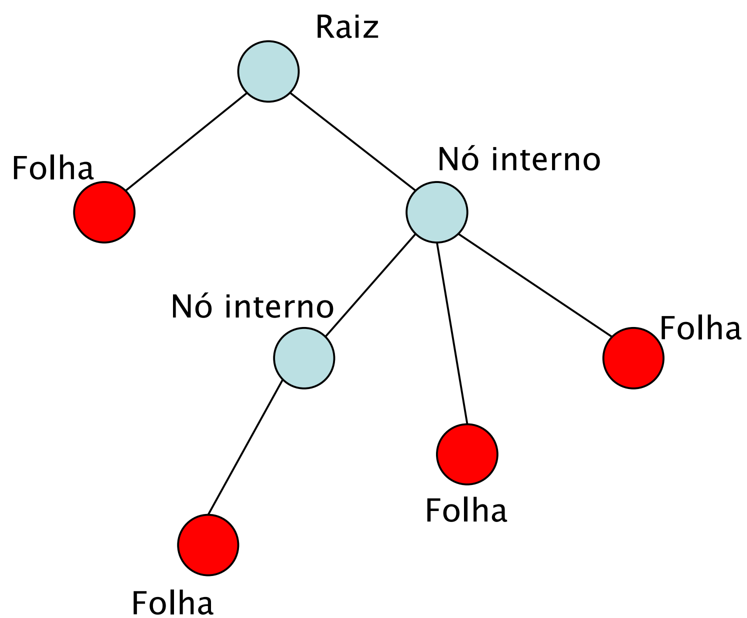
  - Se **T** não é vazia, ela tem um nó especial chamado **raiz** de **T**, que não tem pai.
  - Cada nó v de **T** diferente da raiz tem um único nó pai w;

- Uma árvore pode não ter nós. Quando isso ocorre, dizemos que a árvore **T** é vazia.

- Assim, uma árvore **T** ou é vazia ou consiste de um nó raiz **r** e um conjunto (possivelmente vazio) de árvores cujas raízes são filhas de **r**.

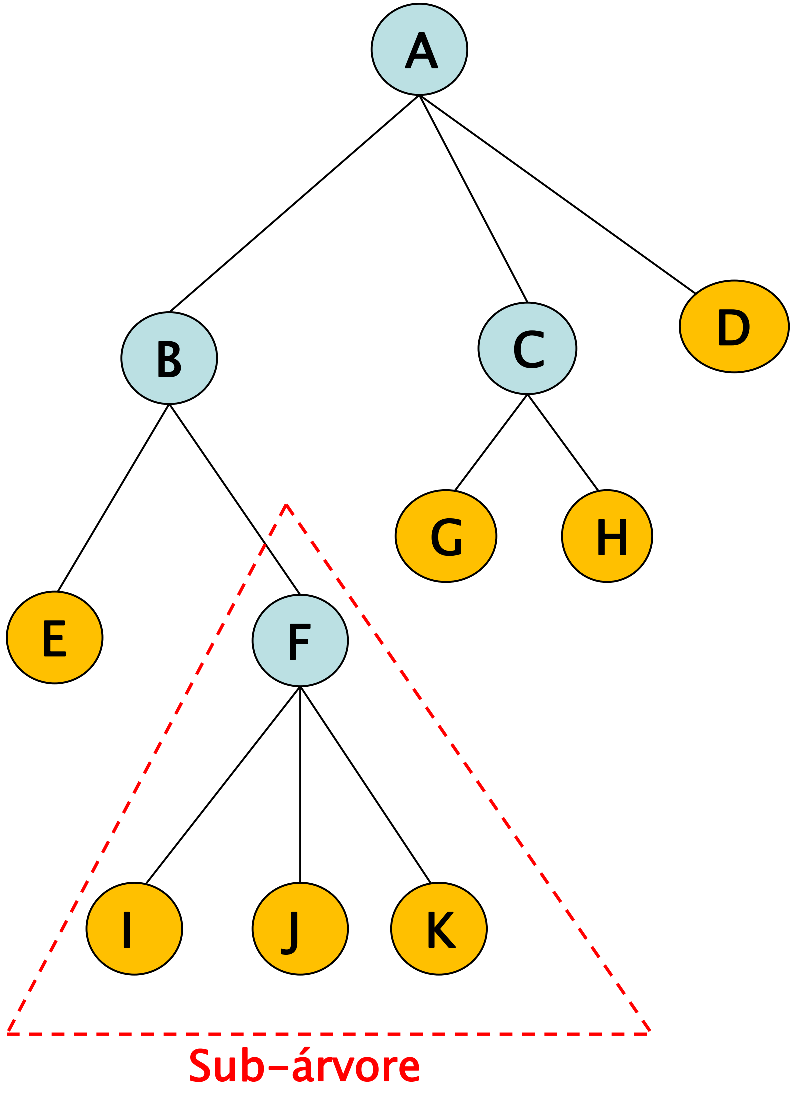
Outros relacionamentos

- Dois nós que são filhos do mesmo pai são **irmãos**.
- Um nó **v** é **externo** se não tem filhos.
- Nós externos também são conhecidos por **folhas**.
- Um nó **v** é **interno** se tem um ou mais filhos.



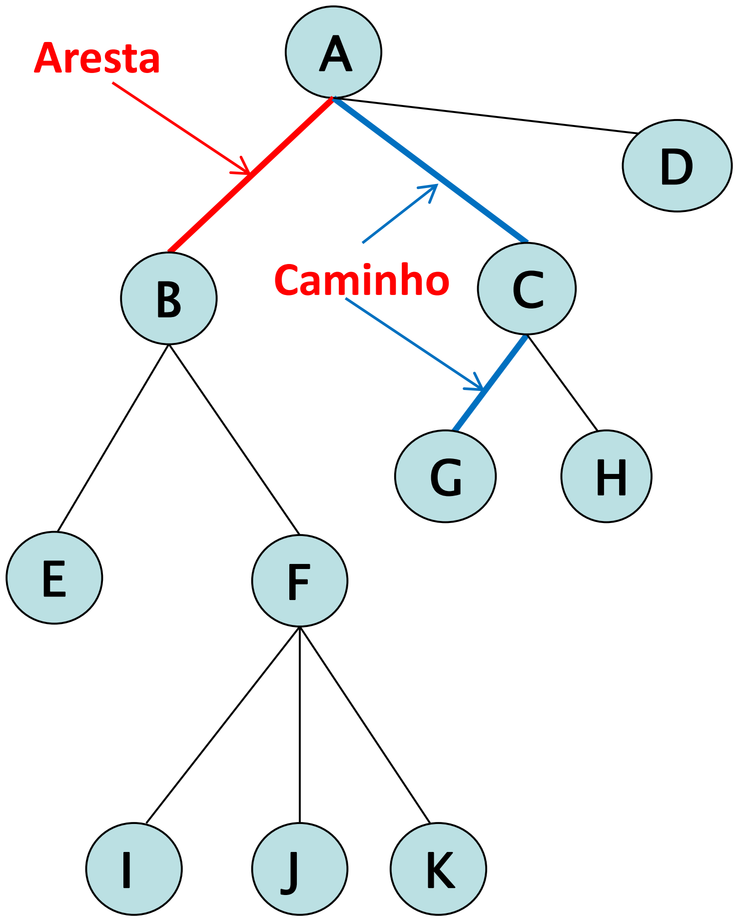
Definições

- **Raiz** (root): Nó sem pai (A)
- **Nó interno**: Nó com pelo menos um filho (A,B,C,F)
- Nó externo ou nó **folha**: nó sem filhos (D,E,G,H,I,J,K)
- **Ancestral** de um nó: pai, avô, bisavô, ...
- **Descendente** de um nó: filho, neto, bisneto, ...
- **Sub-Árvore**: árvore formada por um nó e seus descendentes.



Definições

- **Aresta**: é um par de nós (u,v) tal que u é pai de v. (A,B)
- **Caminho**: é uma sequência de nós tais que quaisquer dois nós consecutivos da sequência sejam arestas. ((A,C),(C,G))
- **Tamanho de um caminho**: # de arestas em um caminho. (Tamanho do caminho ((A,C),(C,G)) = 2).



Definições

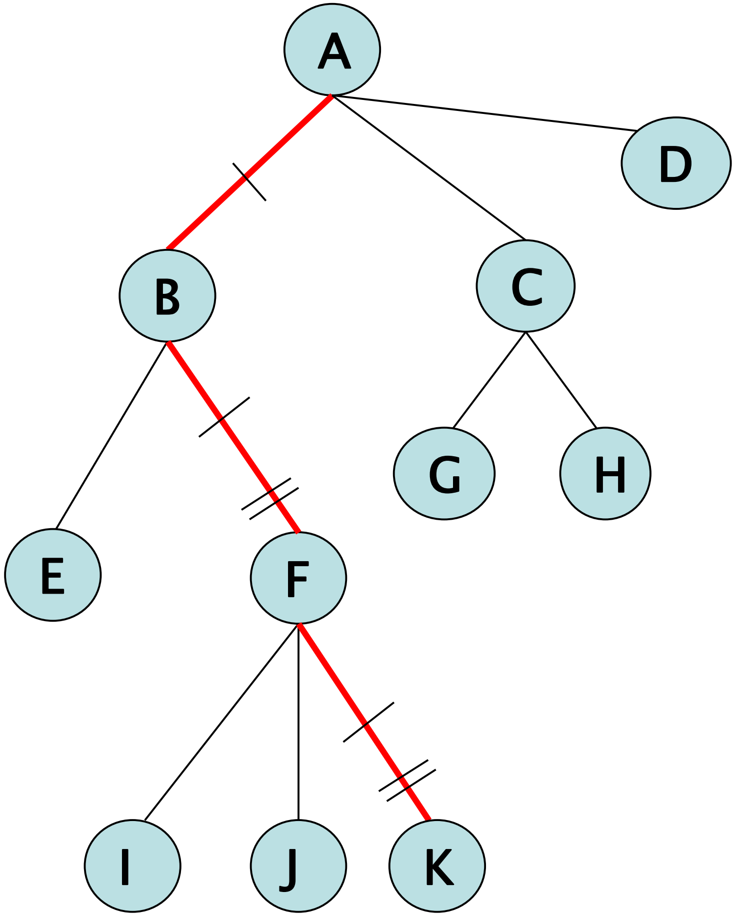
- **Profundidade de um nó n**: é Tamanho do caminho da raiz até o nó **n**.
  (Dept (K) = 3)

- **Profundidade da raiz: ZERO**

- **Altura de um nó**: Tamanho do caminho de **n** até seu mais profundo descendente. (Altura(B) = 2) .

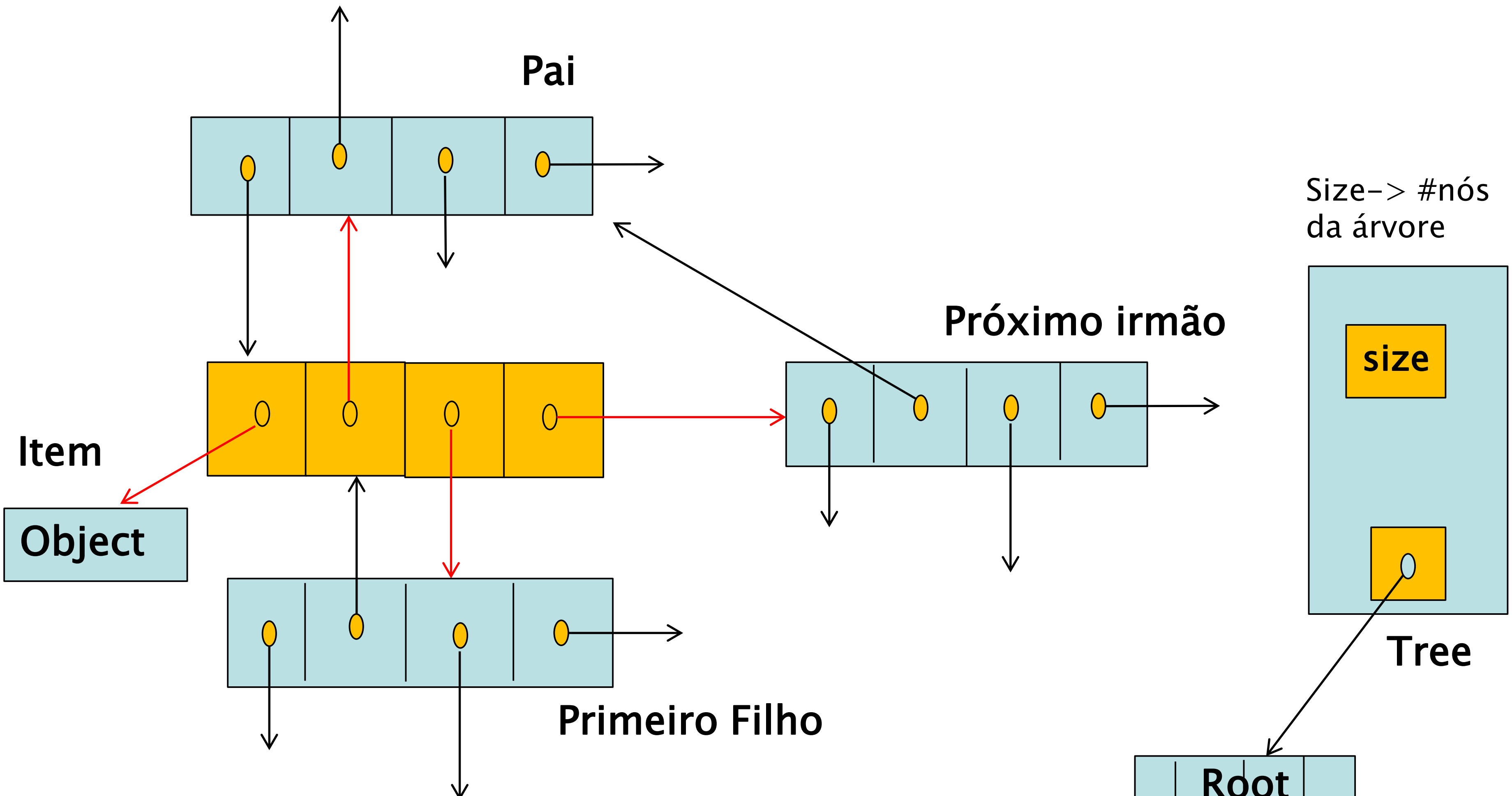
- **Altura de qualquer folha**: ZERO

- **Altura da Árvore** = **Altura da Raiz**



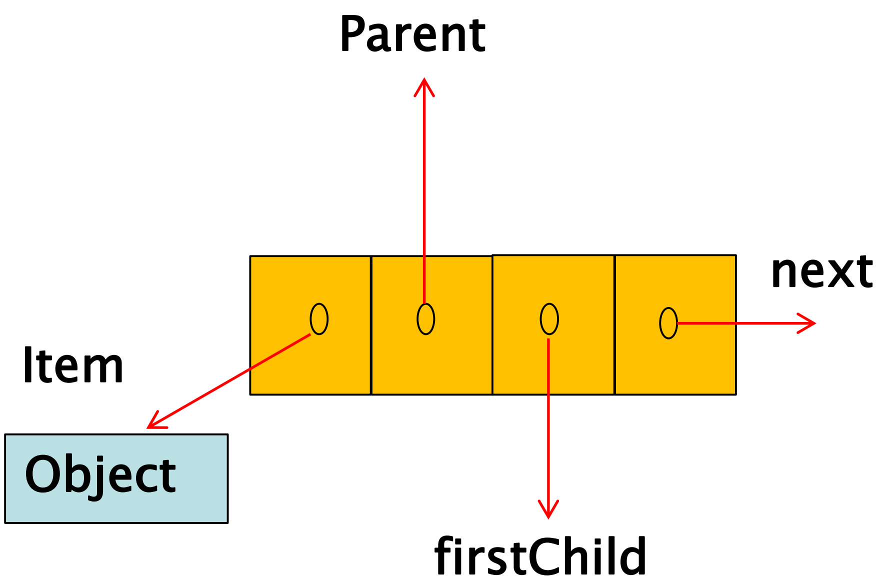
Representando Nó de Árvores

- Cada nó tem quatro referências: item, pai, primeiro filho e próximo irmão.



Representando Nó de Árvores

- Cada nó tem quatro referências: item, pai, primeiro filho e próximo irmão.



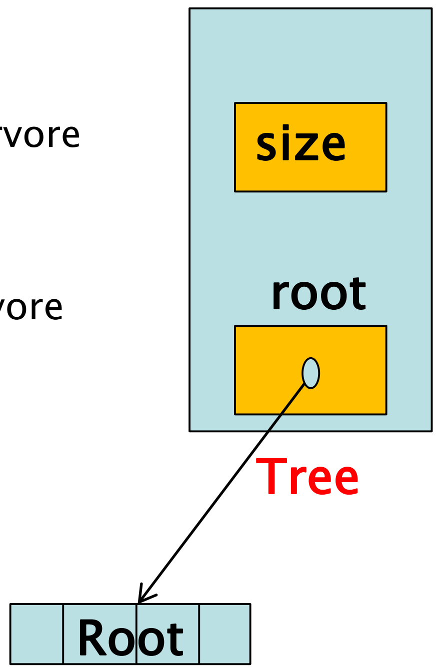

```
Class Node_Tree {

        Object item;
        Node_Tree parent;
        Node_Tree firstChild;
        Node_Tree next;
        .
        .
        .

}
```

Representando Árvores

- O nó de controle possui a referência para o root e o total de nós na árvore.

size -> #nós da árvore

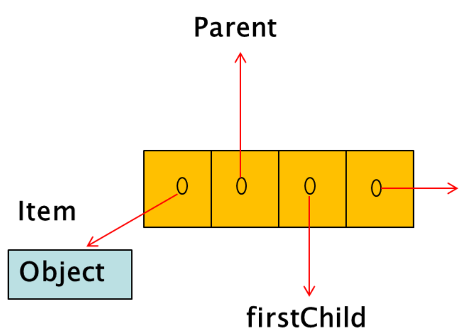
root -> raiz da árvore



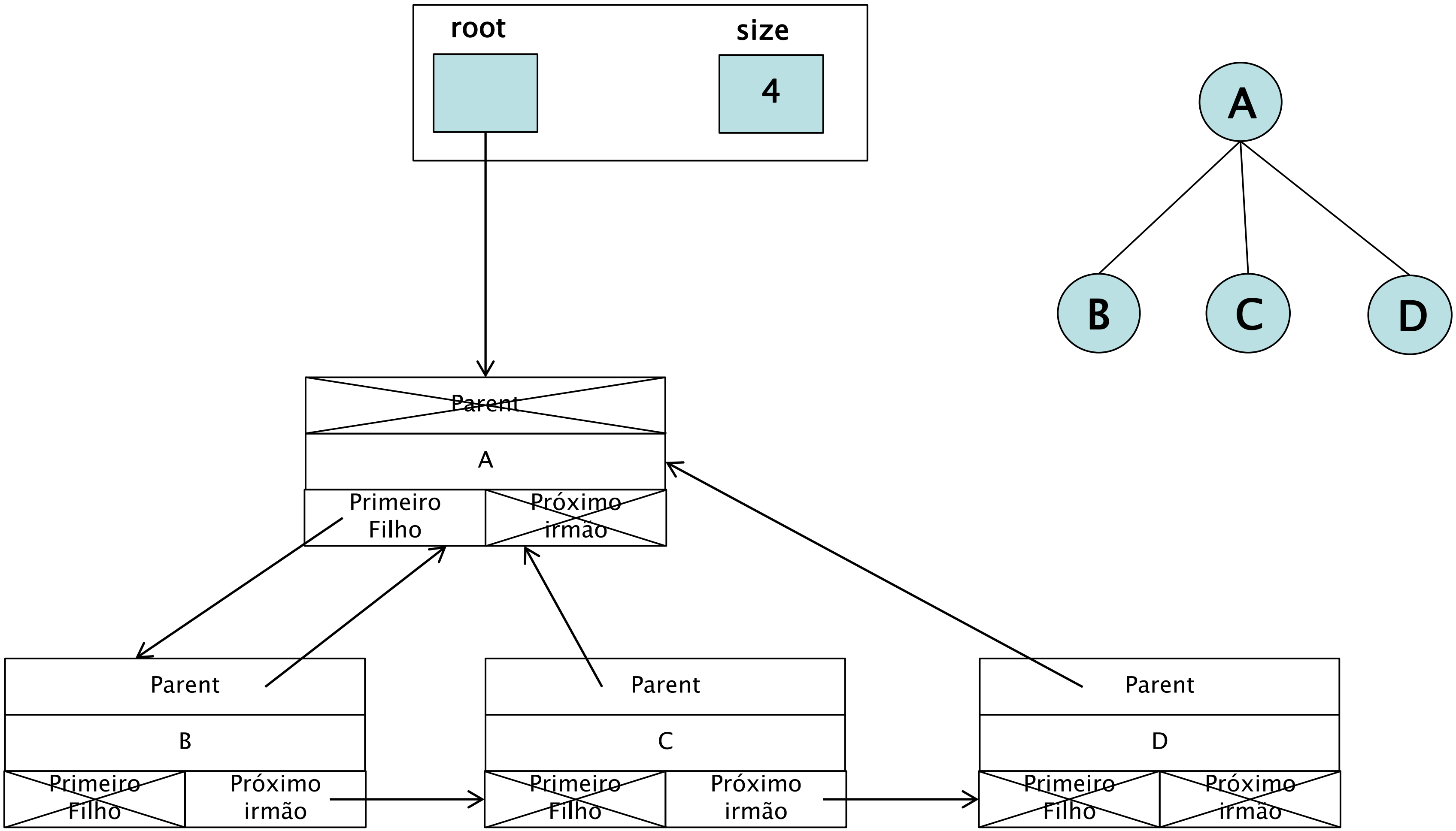

```
Class Tree {

        Node_Tree root;
        int size;
        .
        .
        .

}
```

Representando Nó da árvore

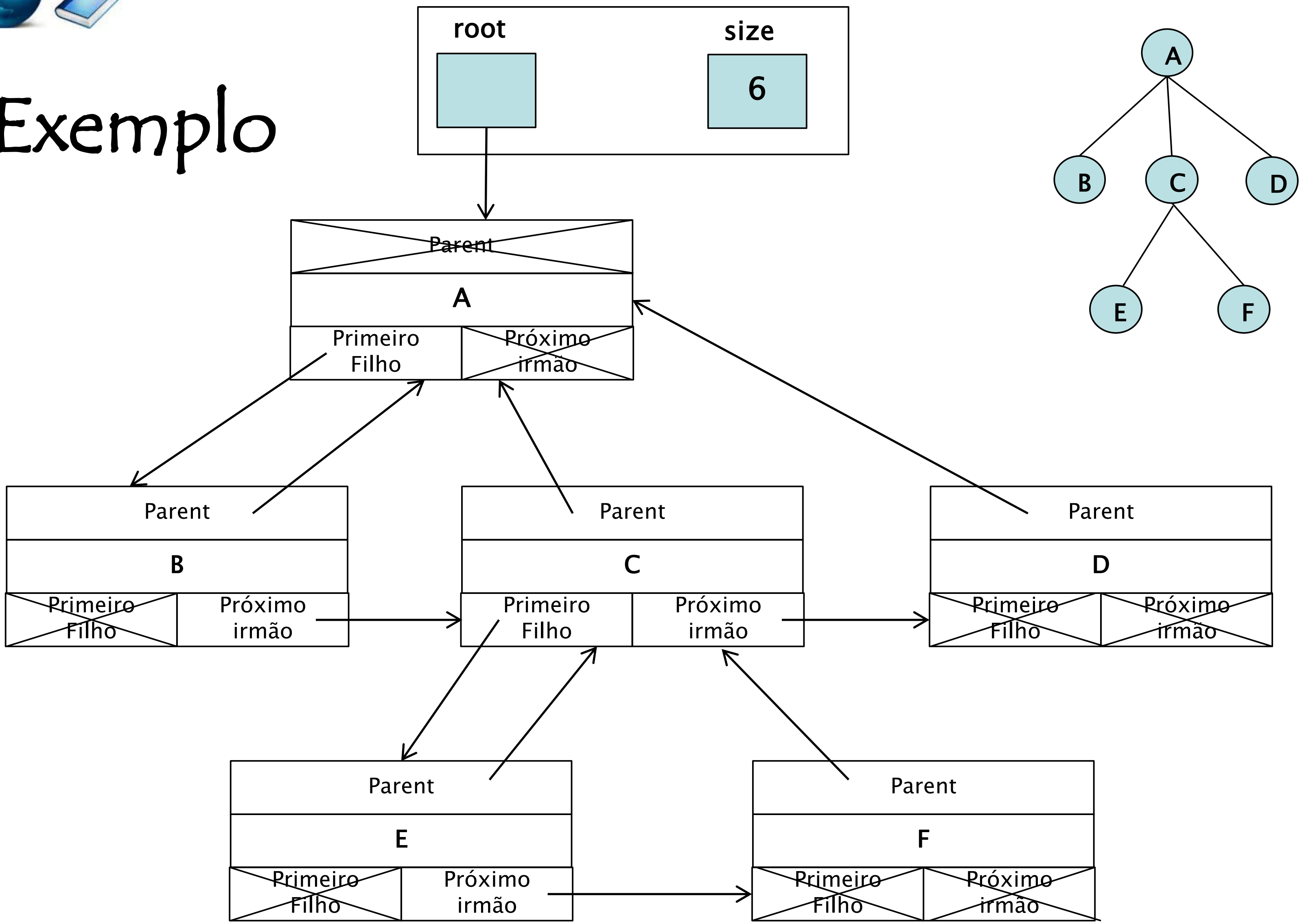
Pai	
Item	
Primeiro Filho	Próximo irmão



Exemplo


root
size
4
A
B
C
D
Parent
A
Primeiro Filho
Próximo irmão
Parent
B
Primeiro Filho
Próximo irmão
Parent
C
Primeiro Filho
Próximo irmão
Parent
D
Primeiro Filho
Próximo irmão

Exemplo

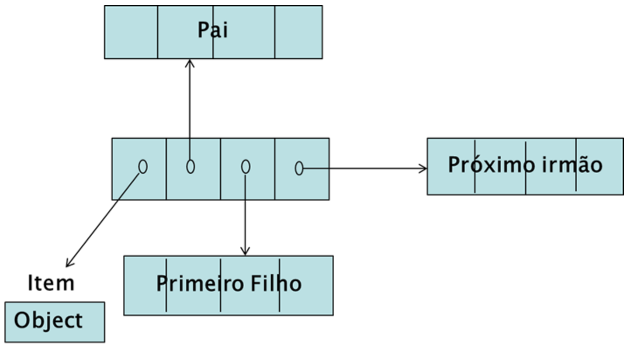

root
size
6
Parent
A
Primeiro Filho
Próximo irmão
Parent
B
Primeiro Filho
Próximo irmão
Parent
C
Primeiro Filho
Próximo irmão
Parent
D
Primeiro Filho
Próximo irmão
Parent
E
Primeiro Filho
Próximo irmão
Parent
F
Primeiro Filho
Próximo irmão
A
B
C
D
E
F

Os tipos abstratos de dados Tree e Node_Tree

```
ret_Root(): retorna o node root da árvore
parent(): retorna o pai do nó
imprime_Parent(): imprime o dado armazenado no pai
children(): retorna lista com os filhos do nó
imprime_Filhos(): Imprime dados dos filhos do nó
isInternal(): testa se nó é node interno
isExternal(): testa se nó é node externo
size(): retorna o número de nodes na árvore
isEmpty(): testa se a árvore é vazia
dept(): retorna o número de ancestrais do node
height(): retorna a altura do node
preorder(): retorna nodes em ordem preorder
postorder(): retorna nodes em ordem postorder
listNodes(): retorna uma coleção dos nodes da árvore
replace(v,e): altera o dado em um determinado node
```

Classe Node_Tree

```java
package maua;

import java.util.Iterator;
import java.util.LinkedList;
import java.util.List;

public class Node_Tree {

        Integer item;
        Node_Tree parent;
        Node_Tree firstChild;
        Node_Tree Next;

        public Node_Tree(Integer item) {
                this.item = item;
                this.parent = null;
                this.firstChild = null;
                this.Next = null;
        }
```



```
parent(v): retorna o pai de v

imprime_Parent(): imprime o dado armazenado no pai
```

```
public Node_Tree parent() {

        if (this.parent == null)
                return null;

        else return (this.parent );

}

public void imprime_Parent() {
        if (this.parent != null)
                System.out.println("Pai:  " + this.parent.item );

        else
                System.out.println("Este nó é root, não tem pai...");
}
```

QualitSys

children(): retorna lista com os filhos do nó

```
public List<Node_Tree> children() {

        List<Node_Tree> lista_children = new LinkedList<Node_Tree>();

        Node_Tree trab;

        if (this.firstChild != null) {
                lista_children.add(this.firstChild);
                trab = this.firstChild ;
                while (trab.Next != null) {
                        lista_children.add(trab.Next );
                        trab = trab.Next ;
                }
                return lista_children;
        }
        else return null;

}
```

imprime_Filhos(): Imprime dados dos filhos do nó

```
public void Imprime_Filhos() {

        List<Node_Tree> lista_children = new LinkedList<Node_Tree>();

        lista_children = this.children();

        if (lista_children != null ) {

                Iterator<Node_Tree> il = lista_children.iterator();

                while (il.hasNext()) {
                        System.out.println(il.next().item);

                }
        }
        else
        System.out.println("Este nó não tem filhos....");
}
```

```
isInternal(): testa se nó é node interno
```

```
public boolean isInternal() {
        if (this.firstChild != null)
                return true;
        else return false;
}
```

dept(): retorna o número de ancestrais do nó

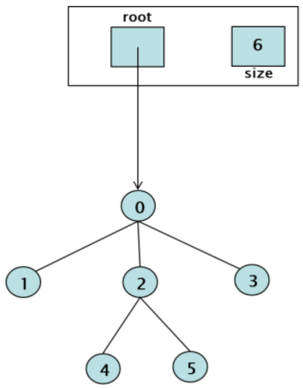
```
public int dept() {
        if (this.parent == null)
                return 0;
        else return ( 1 + this.parent.dept() );

}
```

height(): retorna a altura do nó

```
public int height() {

        if (this.firstChild == null )
                return 0;
        else  {
                int h=0;
                List<Node_Tree> lista_children = this.children();
                Iterator<Node_Tree> il = lista_children.iterator();
                while (il.hasNext())
                        h = Math.max(h, il.next().height());
                return 1 + h;
        }
}
```

Classe Tree

```java
package maua;

public class Tree {

        Node_Tree root;
        int size;

        public Tree() {

                this.root = null;
                this.size = 0;
        }

        public void insert_root(Integer valor) {

                Node_Tree node = new Node_Tree(valor);
                this.root = node;
                this.size = 1;

        }
```



ret_Root(): retorna o node root da árvore.

```
public Node_Tree ret_Root() {
        return (this.root);
}
```

`size(): retorna o número de nós da árvore`

```
public int size() {
        return this.size;
}
```

isEmpty(): testa se a árvore é vazia

```
public boolean isEmpty() {

        if (this.size == 0 )
                return true;
        else return false;
}
```

Travessia de Árvores

- Os métodos vistos até agora permitem que se crie a árvores, seus nós e os relacionamentos (pai/filho) entre os nós criados.
- Travessia de uma árvore significa percorrer todos os nós da mesma.

Atravessando Árvores

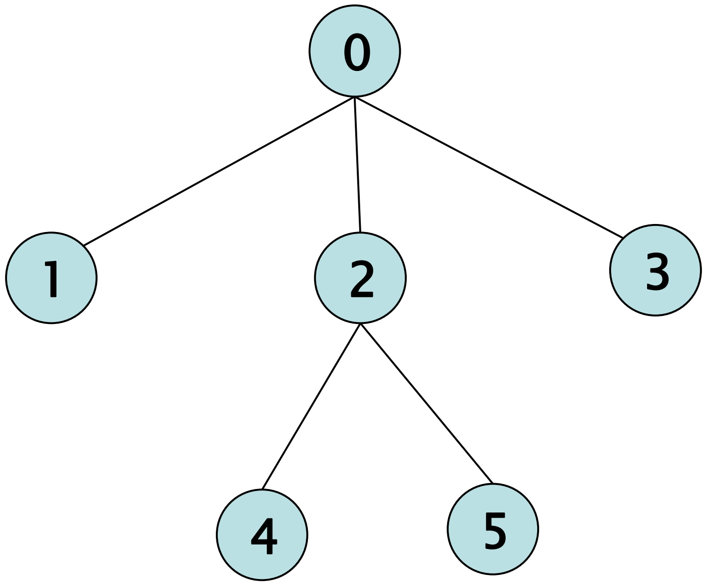
- Atravessar a árvore significa visitar **uma única vez** cada nó da árvore.
- Existem basicamente dois algoritmos de travessia: **preorder** e **postorder**.

Percurso - Preorder

- Na travessia **preorder** de uma árvore T, a raiz de T é visitada em primeiro lugar e em seguida as sub-árvores são visitadas recursivamente.

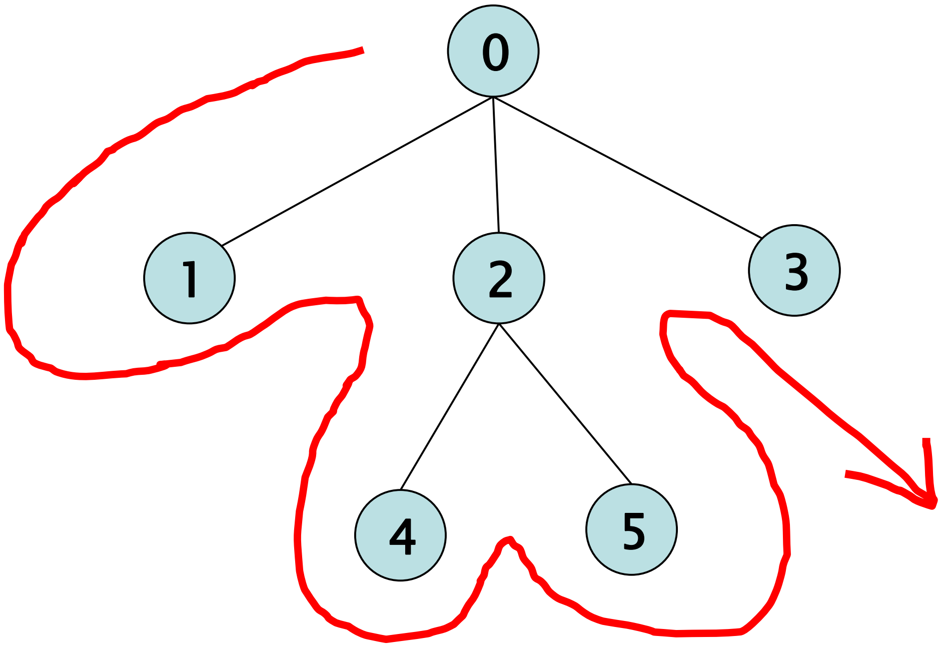
```
public  void preorder() {

        System.out.println(this.item );

        List<Node_Tree> lista_children = this.children();

        if (lista_children != null )
                for (Node_Tree x : lista_children)
                        x.preorder();
}
```

Exercício



- Imprimir os nós da árvore com o uso da travessia preorder.

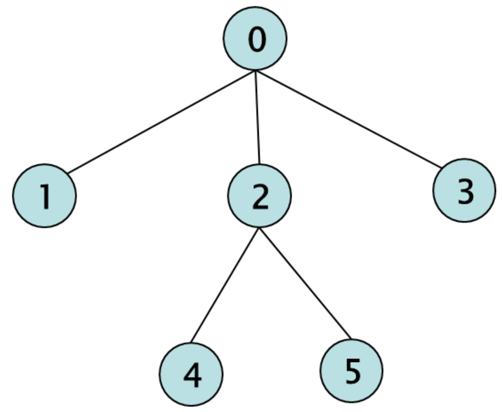
Percurso - Preorder



- Nós são visitados nesta ordem: 0 1 2 4 5 3

- Cada nó é visitado somente uma vez, assim o percurso preorder gasta tempo **O(n)**, onde n é o total de nós da árvore.

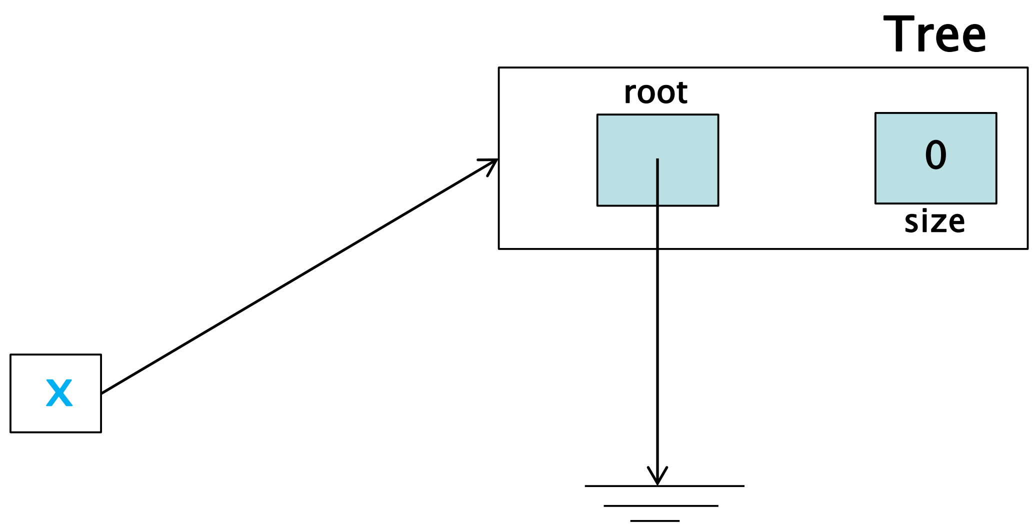
Solução



1. Construir a estrutura de dados que corresponde à árvore (estrutura de controle)
2. Criar o nó root e vinculá-lo à árvore
3. Construir os nós que compõem a árvore
4. Estabelecer os relacionamentos hierárquicos entre os nós
5. Aplicar o algoritmo de preorder na raiz da árvore.

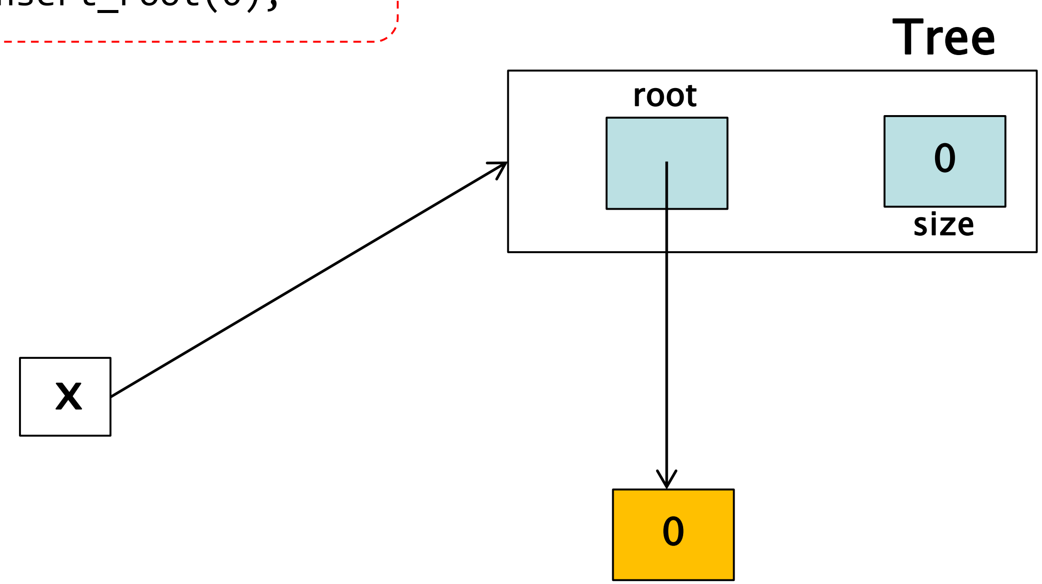
1. Construir a estrutura de dados que corresponde à árvore (estrutura de controle)

```
package maua;
public class Teste_Tree_preorder {
        public static void main(String[] args ) {

                Tree x = new Tree();
```

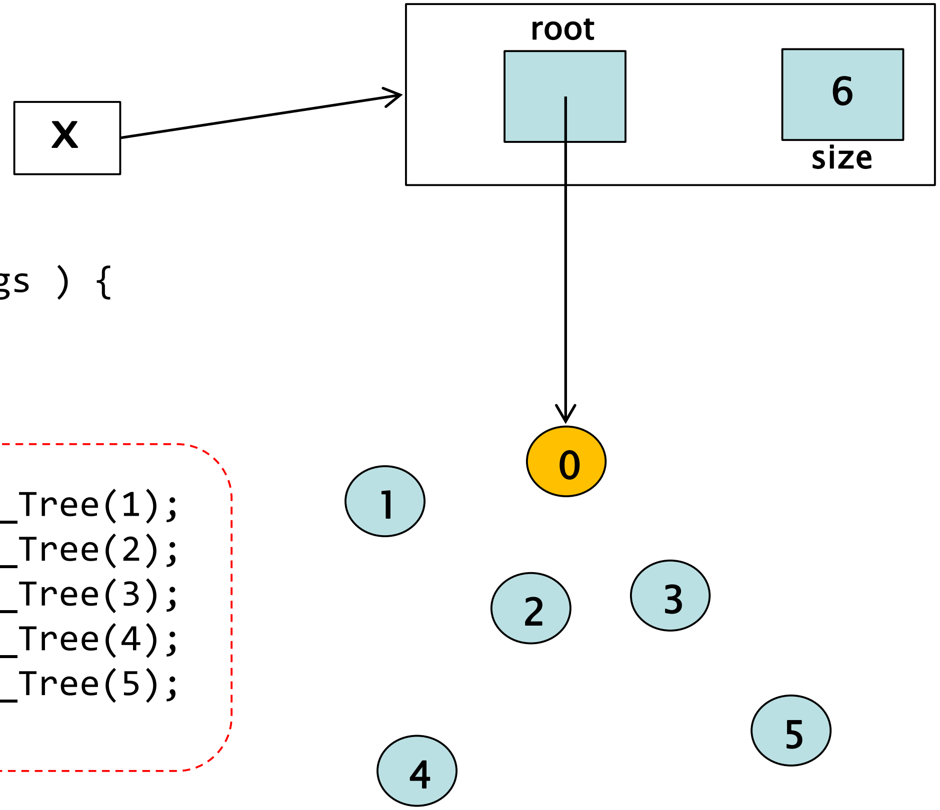


2. Criar o nó root e vinculá-lo à árvore

```
package maua;
public class Teste_Tree {
        public static void main(String[] args ) {
                Tree x = new Tree();
                x.insert_root(0);
```



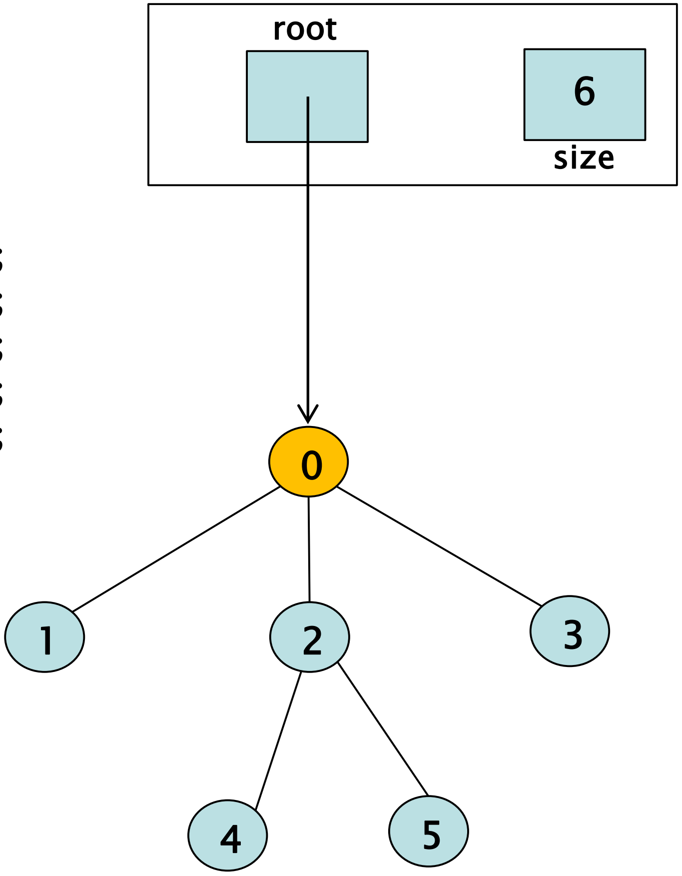
3. Construir os nós que compõem a árvore




```
package maua;
public class Teste_Tree {
        public static void main(String[] args ) {

                Tree x = new Tree();
                x.insert_root(0);

                Node_Tree  no_1 = new Node_Tree(1);
                Node_Tree  no_2 = new Node_Tree(2);
                Node_Tree  no_3 = new Node_Tree(3);
                Node_Tree  no_4 = new Node_Tree(4);
                Node_Tree  no_5 = new Node_Tree(5);
```

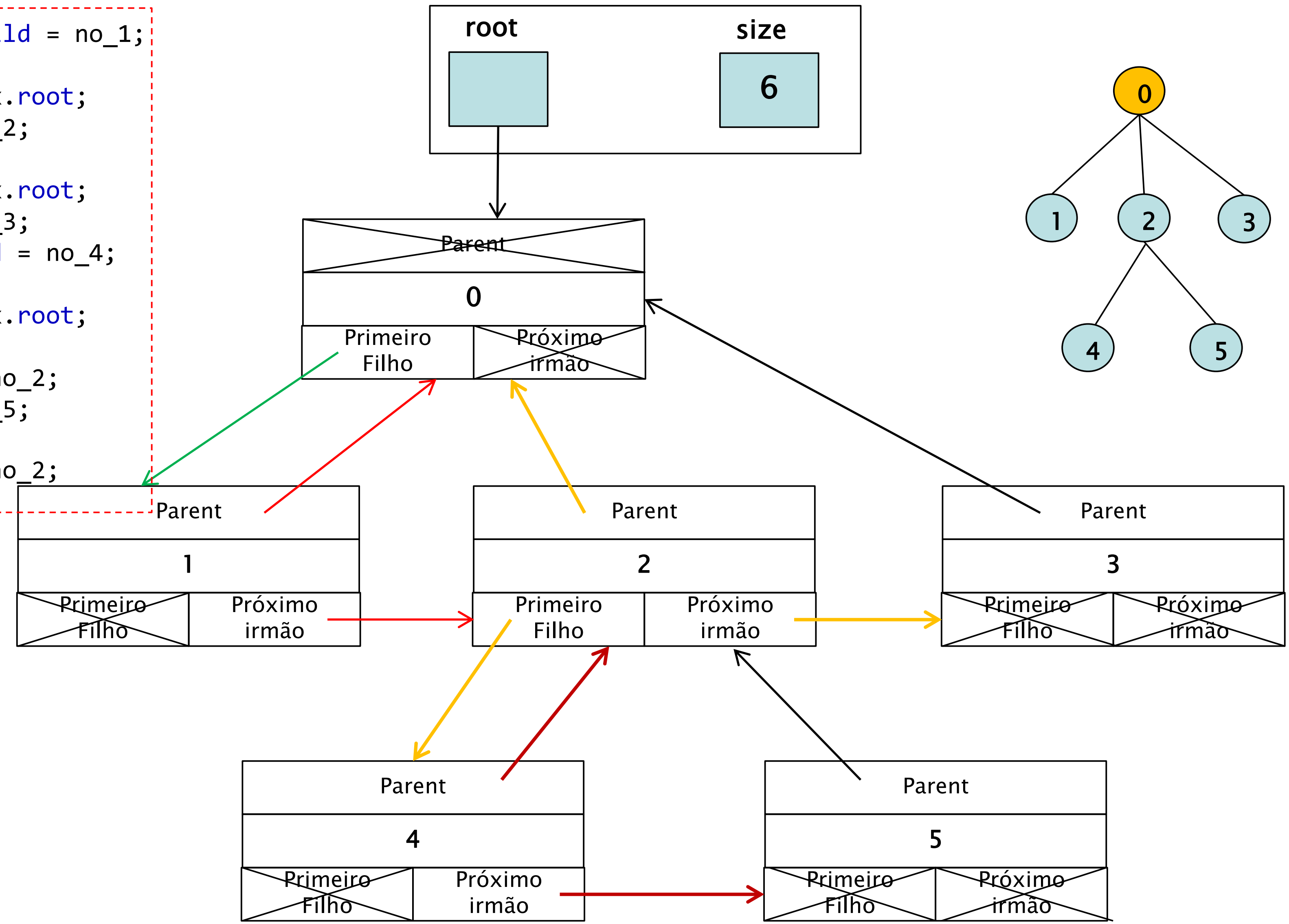
4. Estabelecer os relacionamentos hierárquicos entre os nós

```
package maua;
public class Teste_Tree {
        public static void main(String[] args ) {

                Tree x = new Tree();
                x.insert_root(0);

                Node_Tree  no_1 = new Node_Tree(1);
                Node_Tree  no_2 = new Node_Tree(2);
                Node_Tree  no_3 = new Node_Tree(3);
                Node_Tree  no_4 = new Node_Tree(4);
                Node_Tree  no_5 = new Node_Tree(5);

                x.root.firstChild = no_1;
                no_1.parent = x.root;
                no_1.Next = no_2;

                no_2.parent = x.root;
                no_2.Next = no_3;

                no_3.parent = x.root;

                no_2.firstChild = no_4;
                no_4.parent = no_2;

                no_4.Next = no_5;
                no_5.parent = no_2;
```



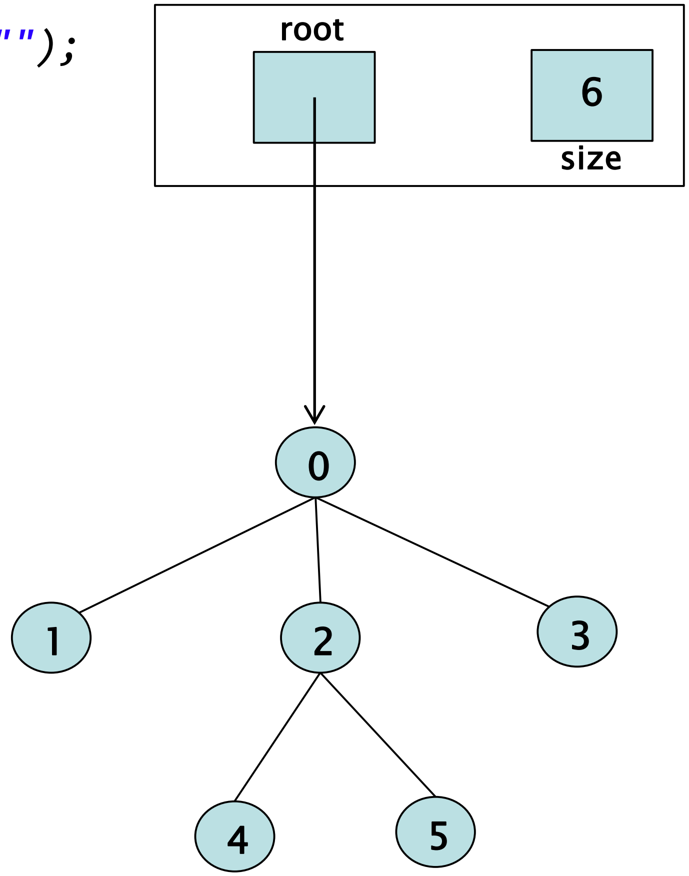
4. Estabelecer os relacionamentos hierárquicos entre os nós

```
x.root.firstChild = no_1;

no_1.parent = x.root;
no_1.Next = no_2;

no_2.parent = x.root;
no_2.Next = no_3;
no_2.firstChild = no_4;

no_3.parent = x.root;

no_4.parent = no_2;
no_4.Next = no_5;

no_5.parent = no_2;
```



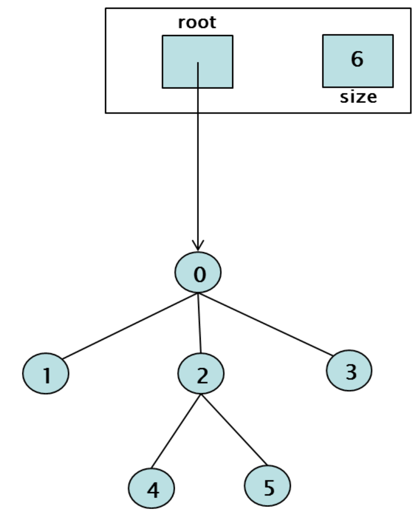
5. Aplicar o algoritmo de preorder na raiz da árvore.

```
x.root.preorder();
System.out.println ("");

    }

}
```



Resposta do programa:

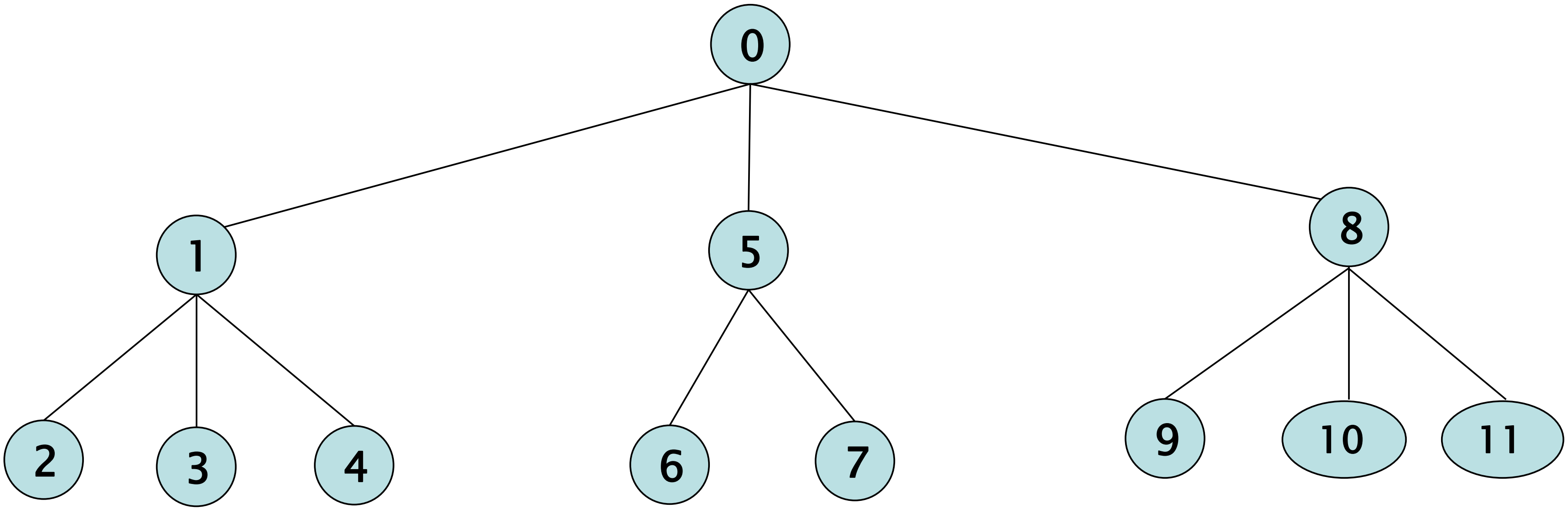
```java
package maua;

public class Teste_Tree_preorder {

        public static void main(String[] args ) {

        Tree x = new Tree();

        x.insert_root(0);

        Node_Tree  no_1 = new Node_Tree(1);
        Node_Tree  no_2 = new Node_Tree(2);
        Node_Tree  no_3 = new Node_Tree(3);
        Node_Tree  no_4 = new Node_Tree(4);
        Node_Tree  no_5 = new Node_Tree(5);

        x.root.firstChild = no_1;

        no_1.parent = x.root;
        no_1.Next = no_2;

        no_2.parent = x.root;
        no_2.Next = no_3;

        no_3.parent = x.root;

        no_2.firstChild = no_4;
        no_4.parent = no_2;

        no_4.Next = no_5;
        no_5.parent = no_2;

        x.root.preorder();
        System.out.println ("");
    }
}
```



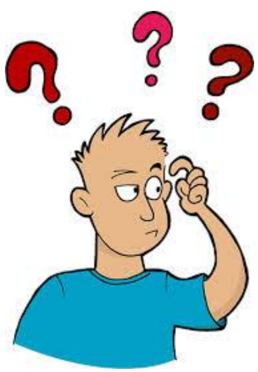
Resposta do programa:

0
1
2
4
5
3

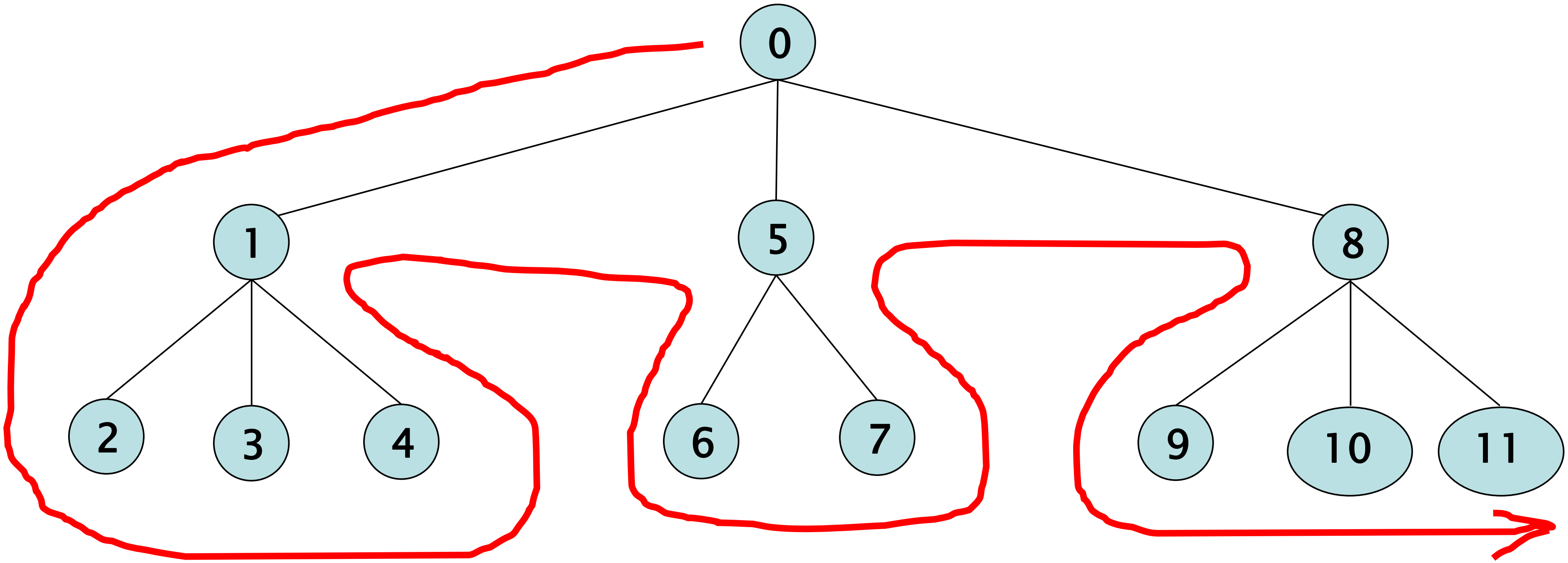
Outro exemplo

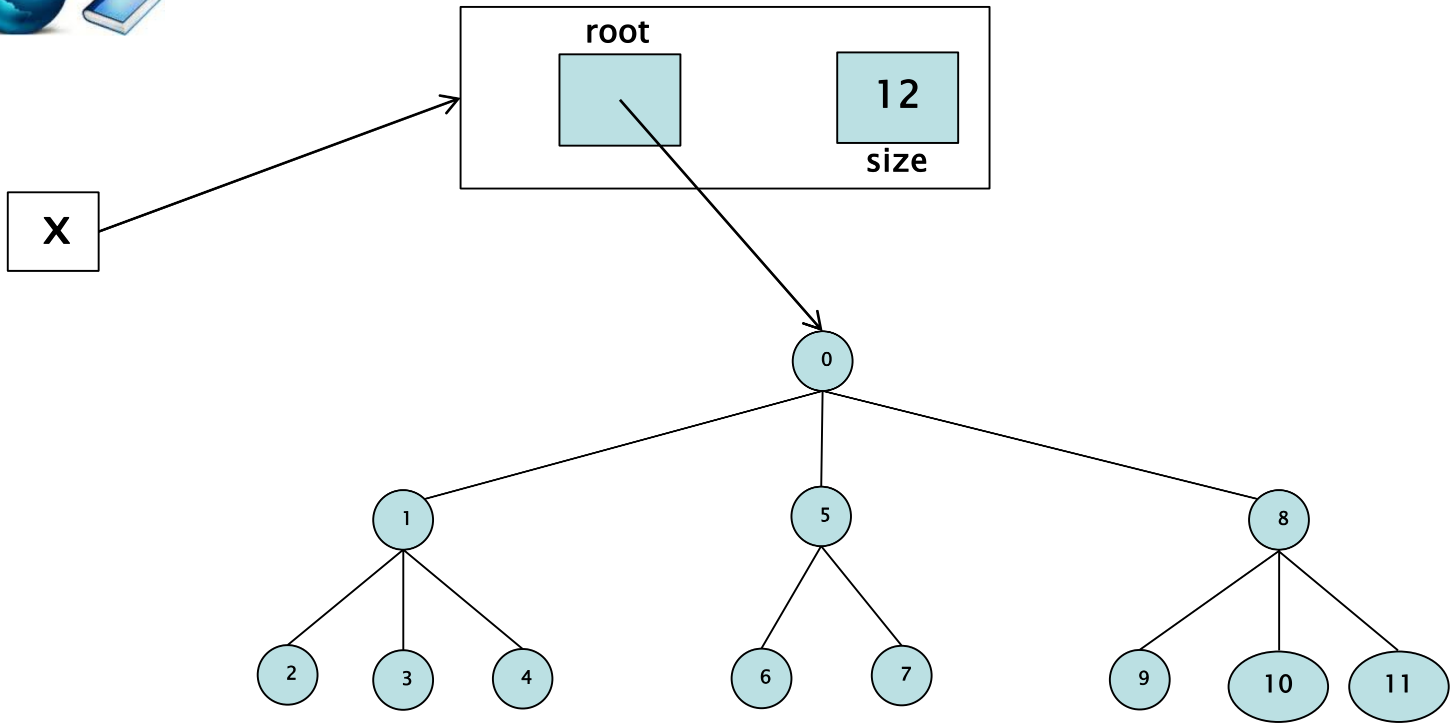


Qual o percurso preordem desta árvore ?

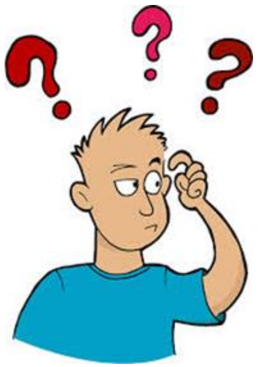


Preorder





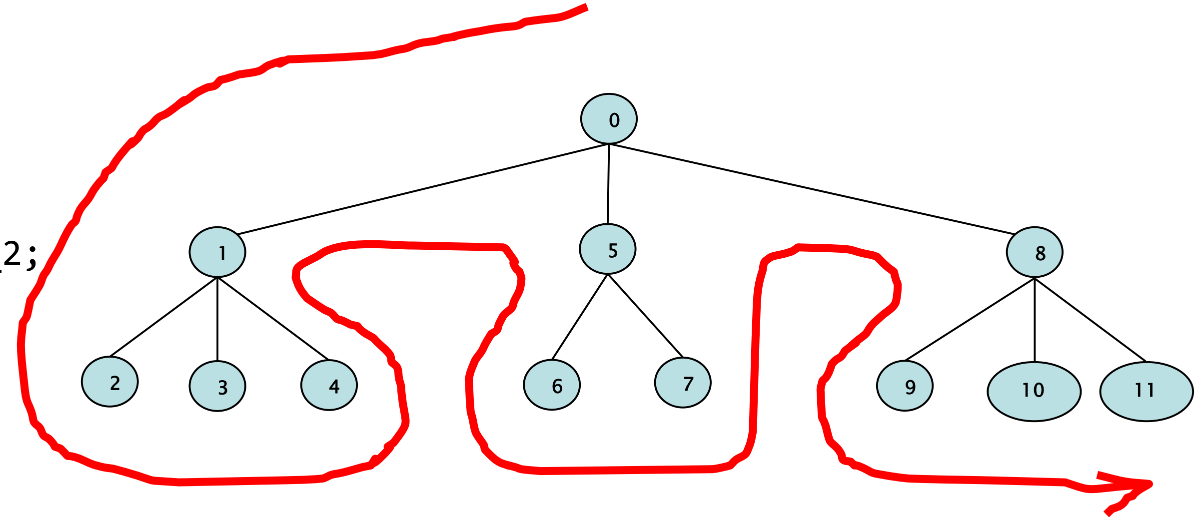
Qual o percurso preordem desta árvore ?



```
package maua;
        public class Teste_Tree {
                public static void main(String[] args ) {

                Tree x = new Tree();
                x.insert_root(0);

                Node_Tree  no_1 = new Node_Tree(1);
                Node_Tree  no_2 = new Node_Tree(2);
                Node_Tree  no_3 = new Node_Tree(3);
                Node_Tree  no_4 = new Node_Tree(4);
                Node_Tree  no_5 = new Node_Tree(5);
                Node_Tree  no_6 = new Node_Tree(6);
                Node_Tree  no_7 = new Node_Tree(7);
                Node_Tree  no_8 = new Node_Tree(8);
                Node_Tree  no_9 = new Node_Tree(9);
                Node_Tree  no_10 = new Node_Tree(10);
                Node_Tree  no_11 = new Node_Tree(11);

                x.root.firstChild = no_1;
```

QualitSys

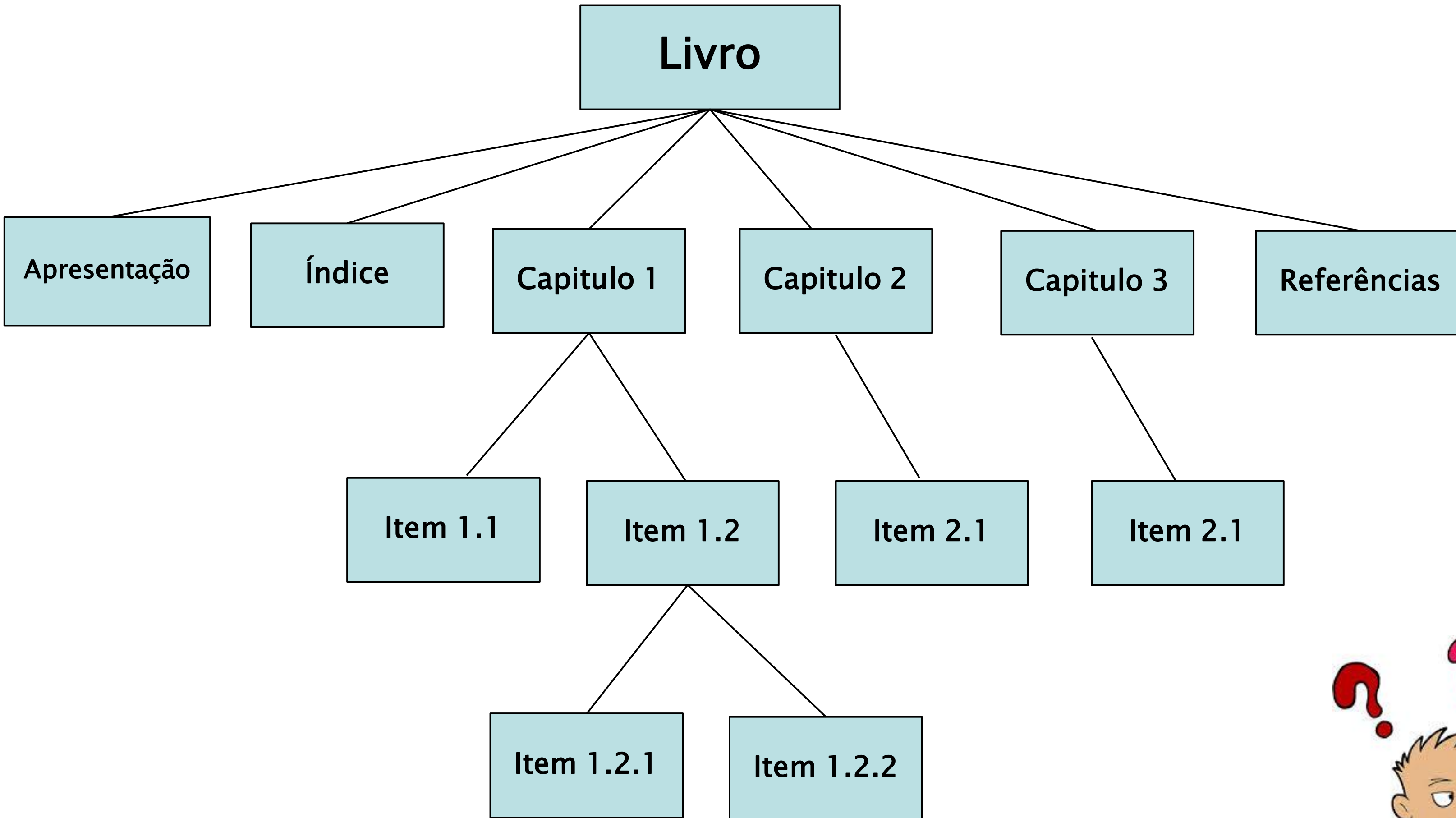
```
no_1.parent = x.root;
no_1.Next = no_5;
no_5.Next = no_8;
no_5.parent = x.root;
no_8.parent = x.root;

no_1.firstChild = no_2;
no_2.Next = no_3;
no_3.Next = no_4;
no_2.parent = no_1;
no_3.parent = no_1;
no_4.parent = no_1;

no_5.firstChild=no_6;
no_6.Next = no_7;
no_6.parent = no_5;
no_7.parent = no_5;

no_8.firstChild = no_9;
no_9.Next = no_10;
no_10.Next = no_11;
no_9.parent = no_8;
no_10.parent = no_8;
no_11.parent = no_8;

x.root.preorder();

}

}
```

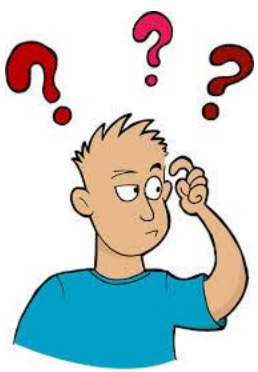


Resposta: Preorder

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

Exercício





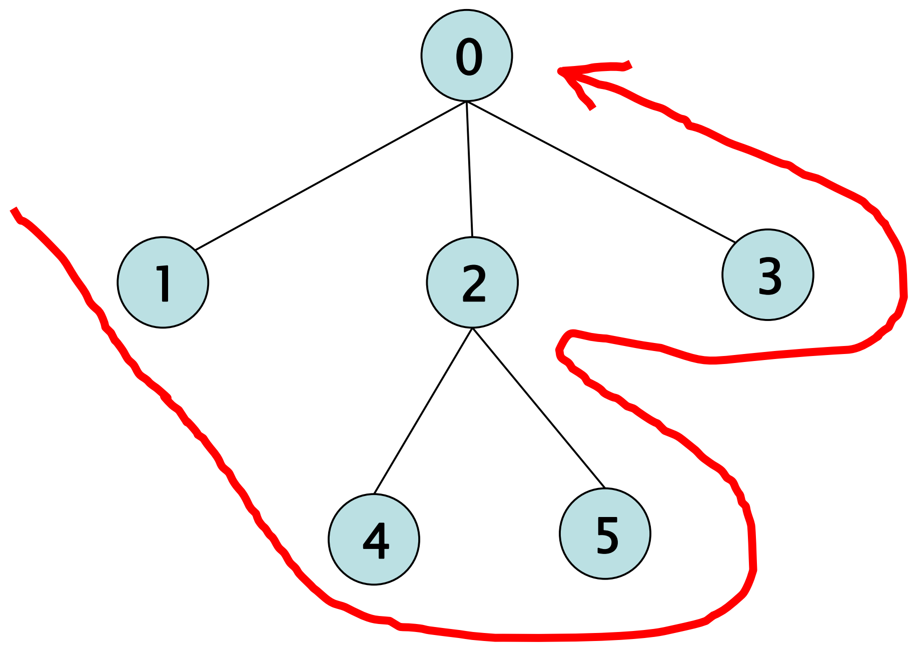
Qual o percurso preordem desta árvore ?

Percurso - Postorder

- Este algoritmo pode ser visto como o **oposto** do percurso **preorder**, pois as sub-árvores dos filhos são recursivamente atravessadas e em seguida o root é visitado.

```
public  void postorder() {

        List<Node_Tree> lista_children = this.children();

        if (lista_children != null )
                for (Node_Tree x : lista_children)
                        x.postorder();

        System.out.println(this.item );



}
```

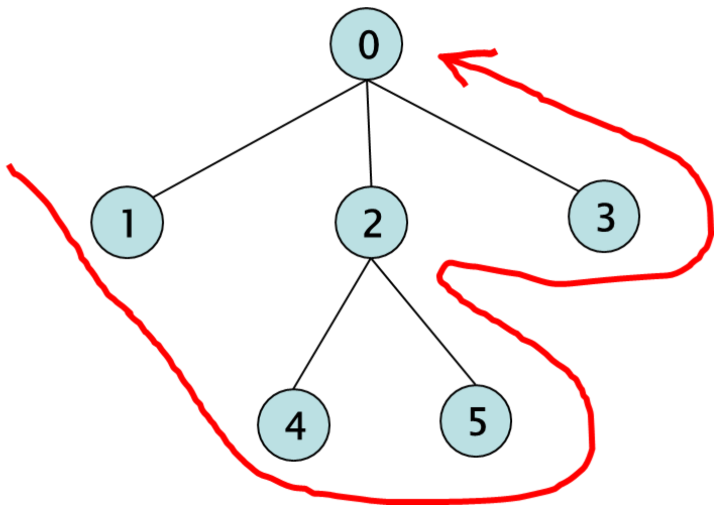
Percurso - Postorder

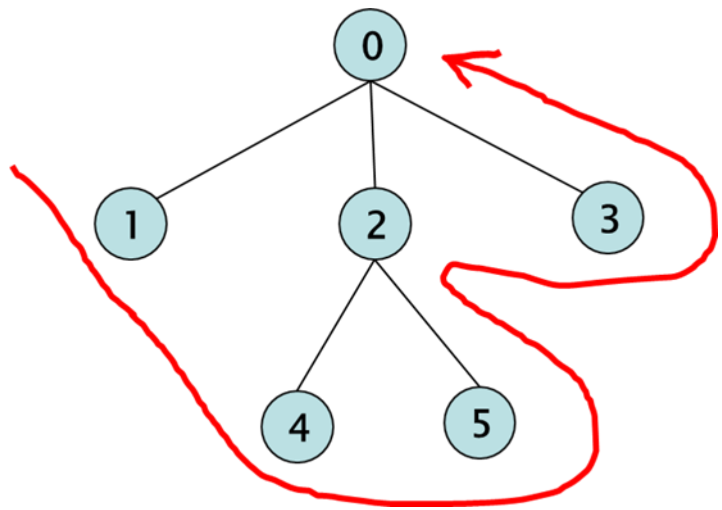


- Nós são visitados nesta ordem: 1 4 5 2 3 0

- Cada nó é visitado somente uma vez, assim o percurso preorder gasta tempo O(n), onde n é o total de nós da árvore.

```java
package uscs;
public class Teste_Tree {
public static void main(String[] args ) {

Tree x = new Tree();
x.insert_root(0);

Node_Tree  no_1 = new Node_Tree(1);
Node_Tree  no_2 = new Node_Tree(2);
Node_Tree  no_3 = new Node_Tree(3);
Node_Tree  no_4 = new Node_Tree(4);
Node_Tree  no_5 = new Node_Tree(5);

x.root.firstChild = no_1;
no_1.parent = x.root;
no_1.Next = no_2;

no_2.parent = x.root;
no_2.Next = no_3;

no_3.parent = x.root;

no_2.firstChild = no_4;
no_4.parent = no_2;

no_4.Next = no_5;
no_5.parent = no_2;
```



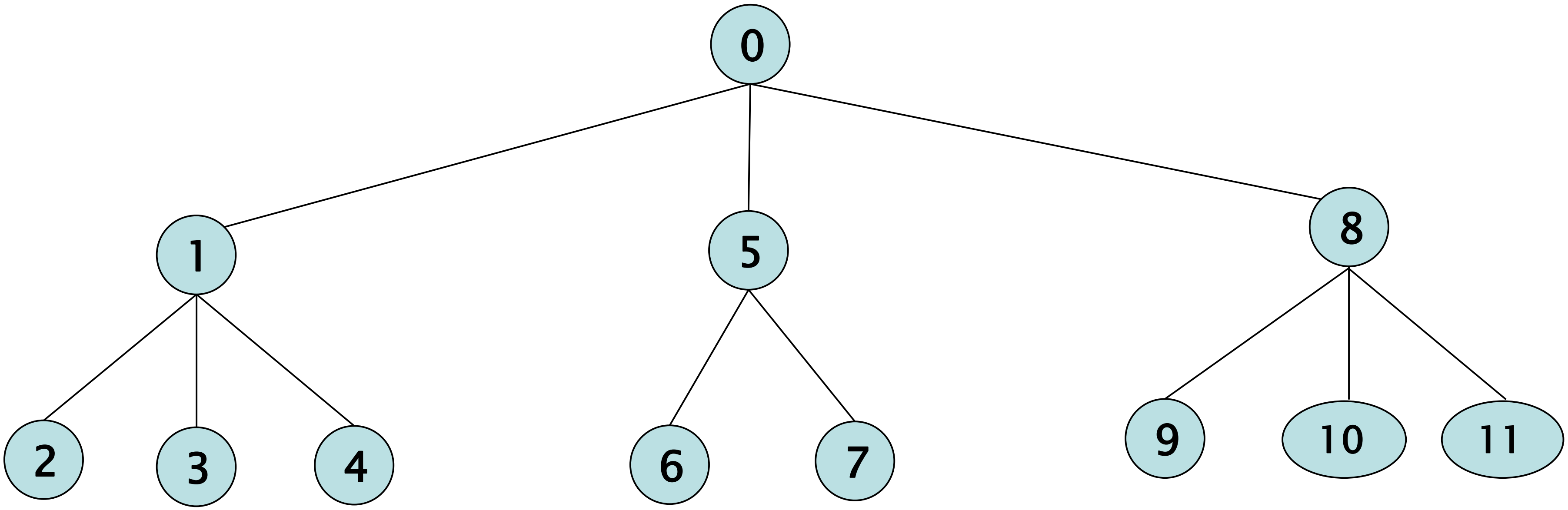



```
        x.root.postorder();
        System.out.println ("");

    }

}
```

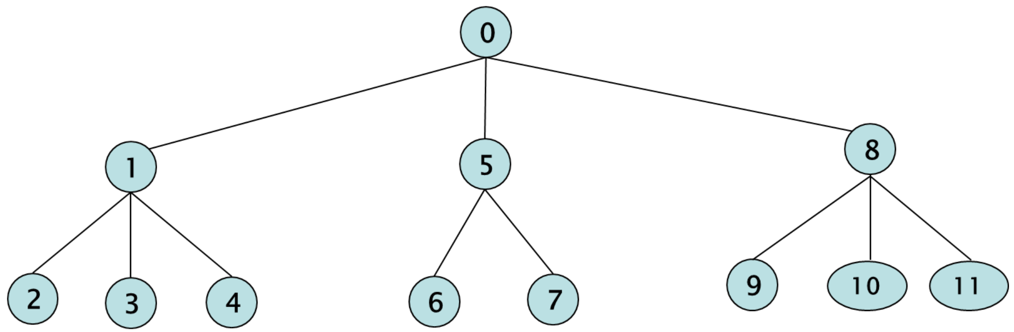
Resposta: Posorder

1
4
5
2
3
0

Outro exemplo

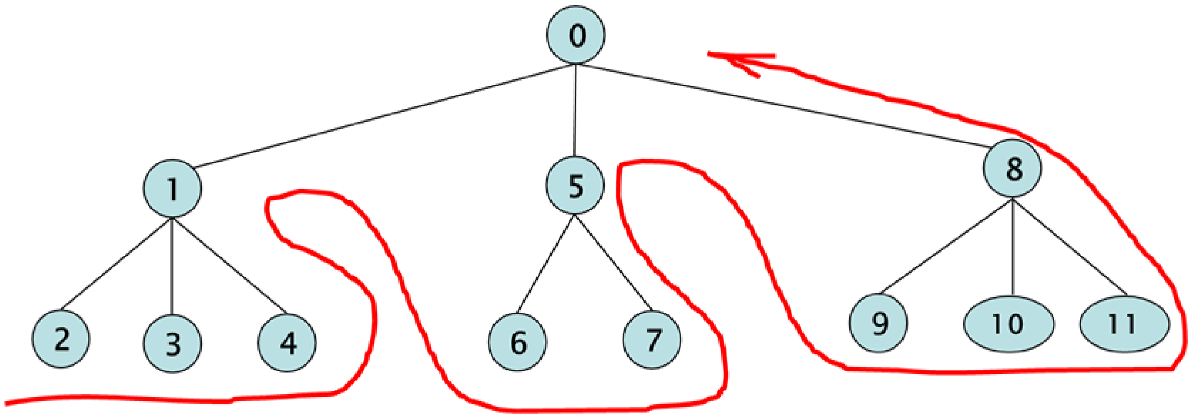


Qual o percurso postordem desta árvore ?

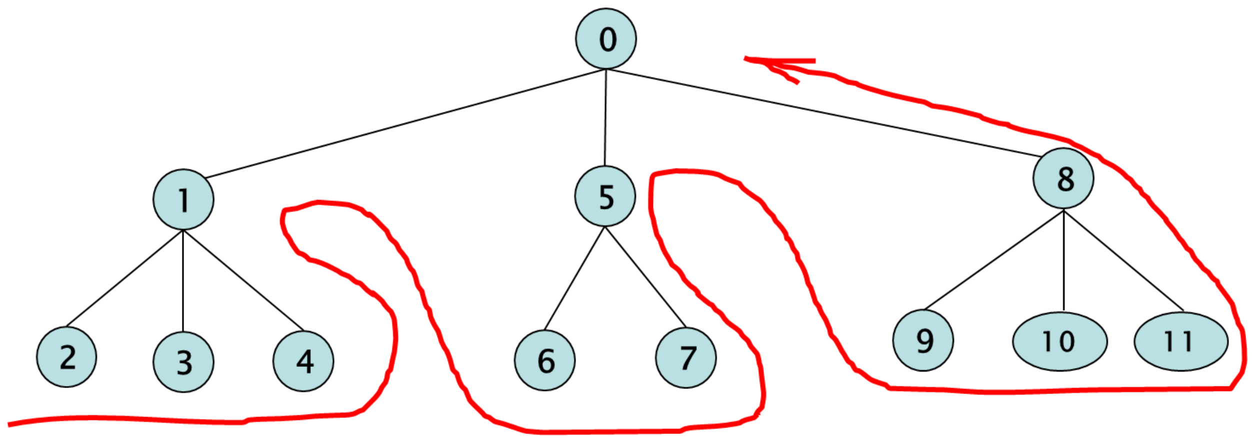



```
package maua;
public class Teste_Tree {
        public static void main(String[] args ) {

                Tree x = new Tree();
                x.insert_root(0);

                Node_Tree  no_1 = new Node_Tree(1);
                Node_Tree  no_2 = new Node_Tree(2);
                Node_Tree  no_3 = new Node_Tree(3);
                Node_Tree  no_4 = new Node_Tree(4);
                Node_Tree  no_5 = new Node_Tree(5);
                Node_Tree  no_6 = new Node_Tree(6);
                Node_Tree  no_7 = new Node_Tree(7);
                Node_Tree  no_8 = new Node_Tree(8);
                Node_Tree  no_9 = new Node_Tree(9);
                Node_Tree  no_10 = new Node_Tree(10);
                Node_Tree  no_11 = new Node_Tree(11);
```

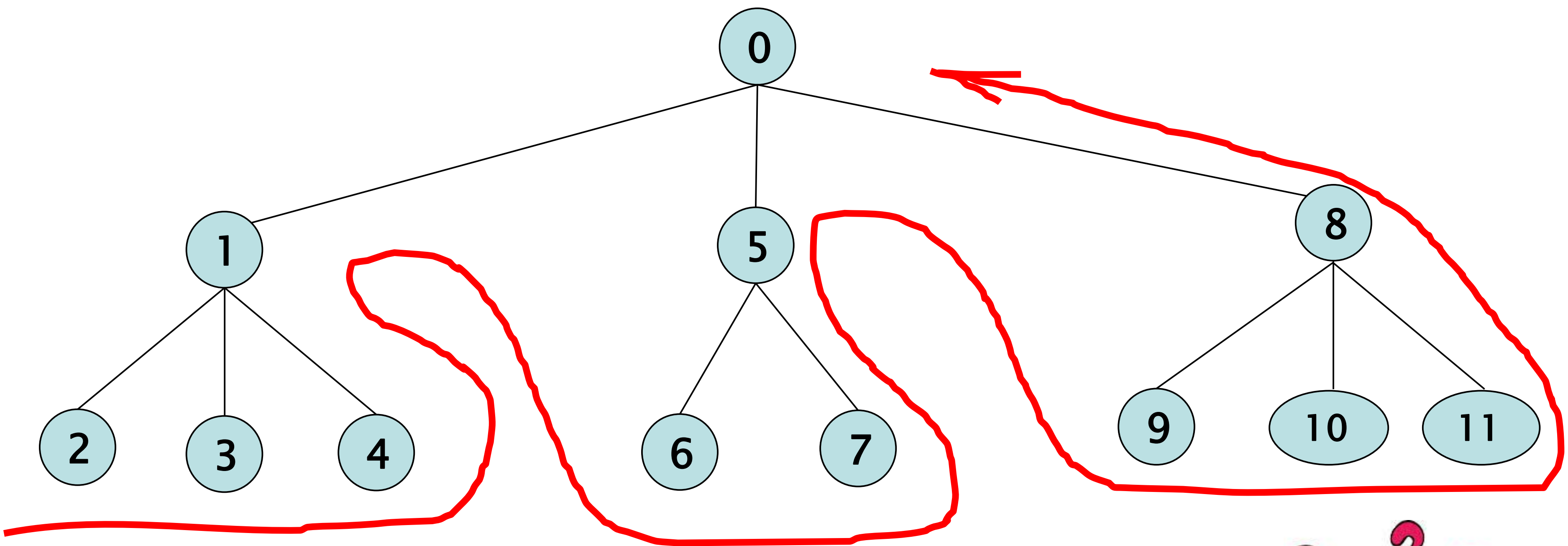
```
x.root.firstChild = no_1;
no_1.parent = x.root;
no_1.Next = no_5;
no_5.Next = no_8;
no_5.parent = x.root;
no_8.parent = x.root;

no_1.firstChild = no_2;
no_2.Next = no_3;
no_3.Next = no_4;
no_2.parent = no_1;
no_3.parent = no_1;
no_4.parent = no_1;

no_5.firstChild=no_6;
no_6.Next = no_7;
no_6.parent = no_5;
no_7.parent = no_5;
```



```
no_8.firstChild = no_9;
no_9.Next = no_10;
no_10.Next = no_11;
no_9.parent = no_8;
no_10.parent = no_8;
no_11.parent = no_8;

x.root.postorder();
System.out.println ("");

 }

}
```



Outro exemplo



Qual o percurso postordem desta árvore ?

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

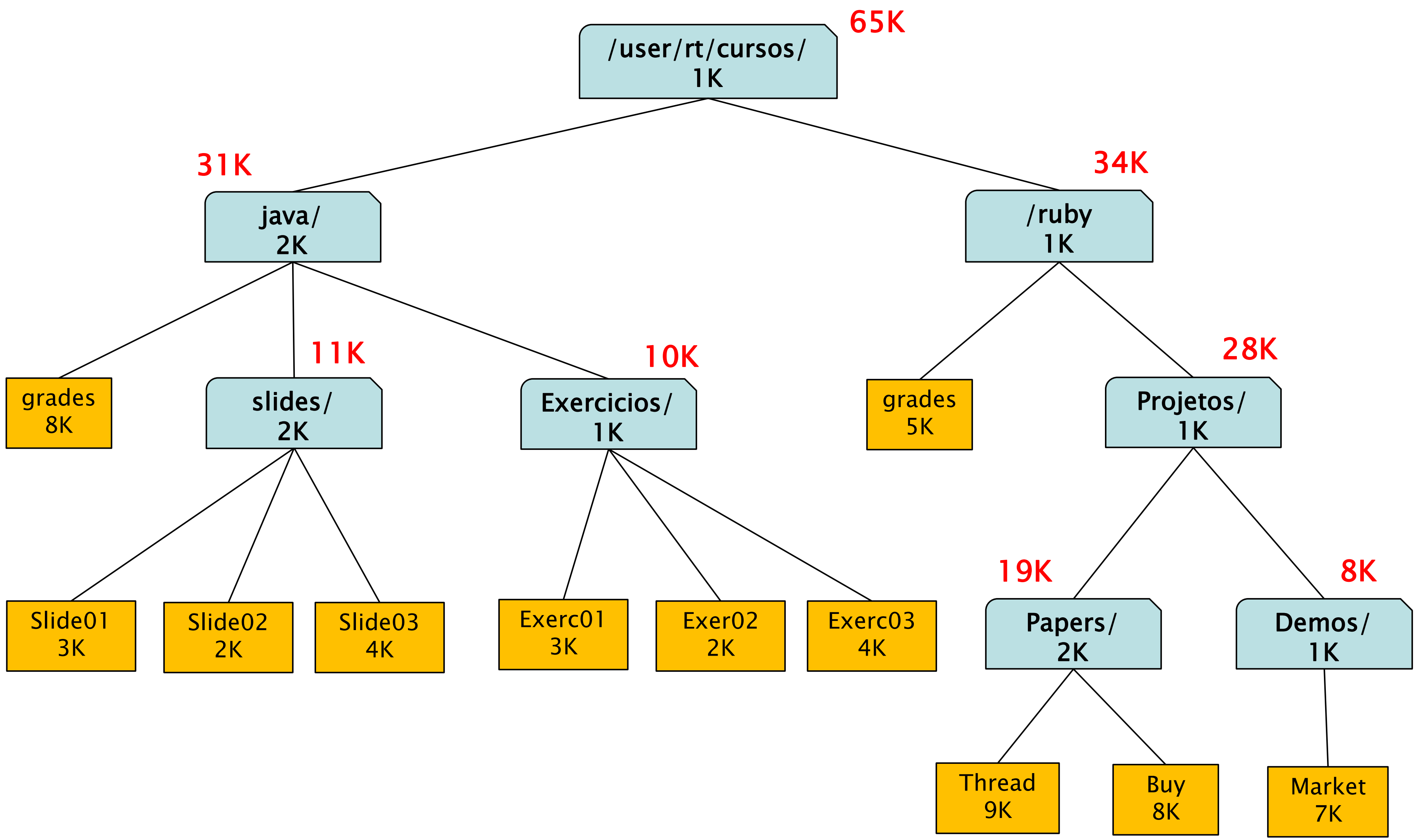
Aplicação travessia postorder

- O método **postorder** é útil para resolver problemas onde desejamos computar alguma propriedade para cada nó **v** da árvore, mas esta computação requer que a mesma computação tenha sido feita previamente para os filhos do nó **v**.

- Para exemplificar o método, considere um sistema de arquivos em árvore, onde nós externos representam arquivos e nós internos diretórios. O problema consiste em computar o espaço em disco usado por um diretório, o qual é recursivamente calculado por:

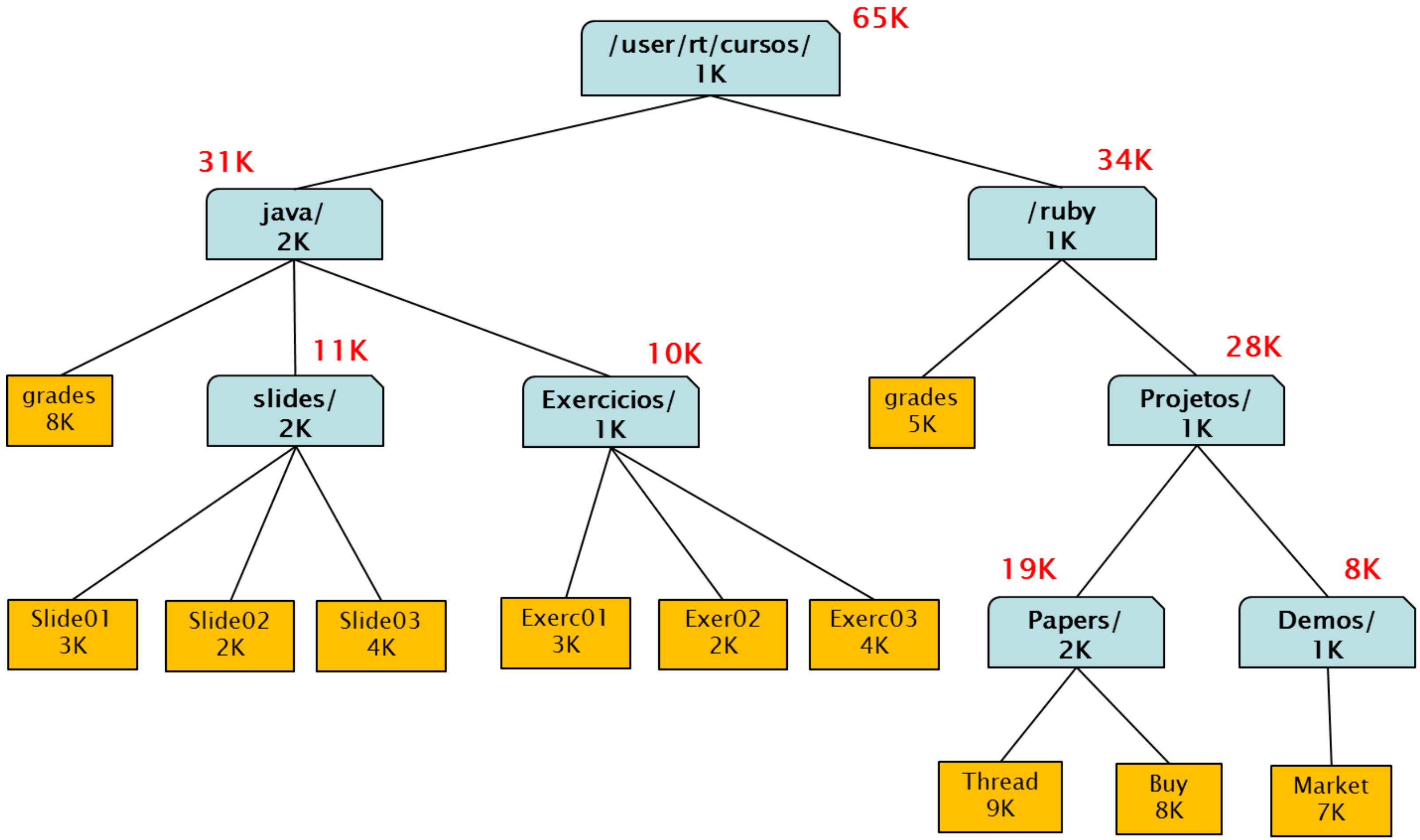
  - o tamanho do próprio diretório
  - o tamanho dos arquivos no diretório
  - o espaço usado pelos diretórios filhos

Aplicação travessia postorder



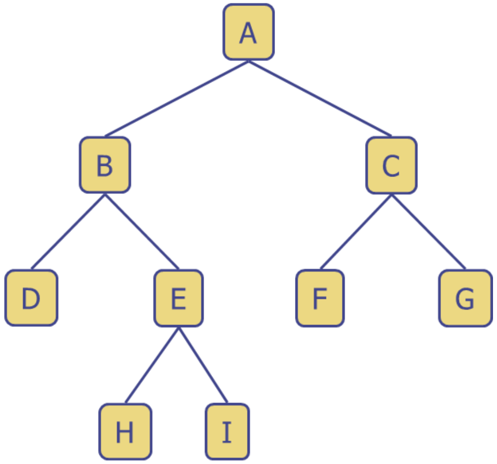
Exercício

- Escrever um código Java para retornar o espaço total de bytes armazenados por um sistema de arquivos.

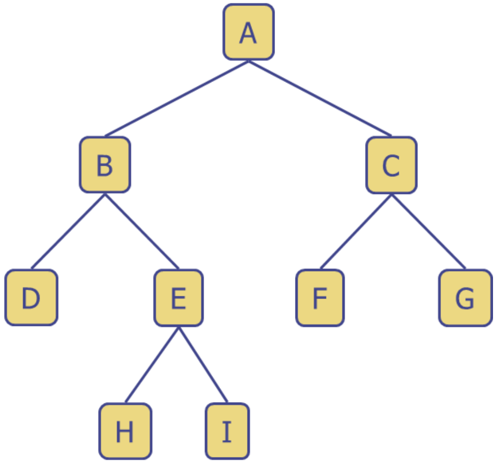


Árvore Binária

- É uma árvore ordenada com as seguintes propriedades:
  - Todo nó tem no máximo 2 filhos.
  - Cada filho é rotulado como sendo **filho a esquerda** ou **filho a direita**.
  - Um filho a esquerda precede o filho a direita na ordenação dos filhos de um nó.
  - Assim, filhos formam um par ordenado.

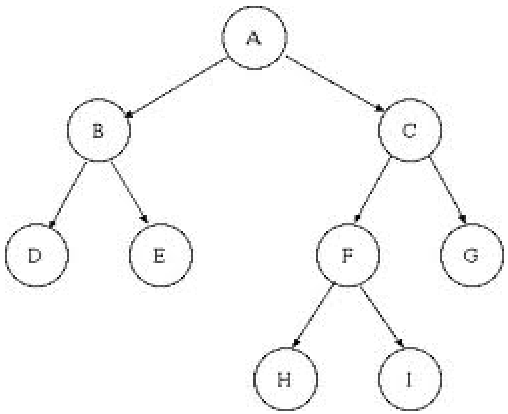


Árvore Binária



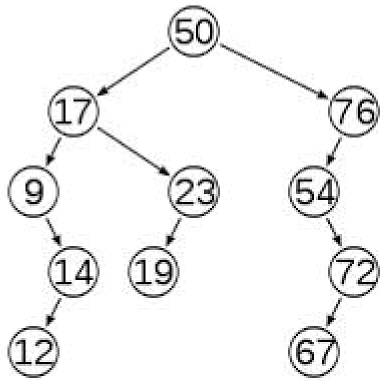
Árvore Binária própria

- Uma árvore binária é **<u>própria</u>** se cada nó tem 0 ou 2 filhos.
- Em uma árvore binária **<u>própria</u>** cada nó interno tem exatamente 2 filhos.



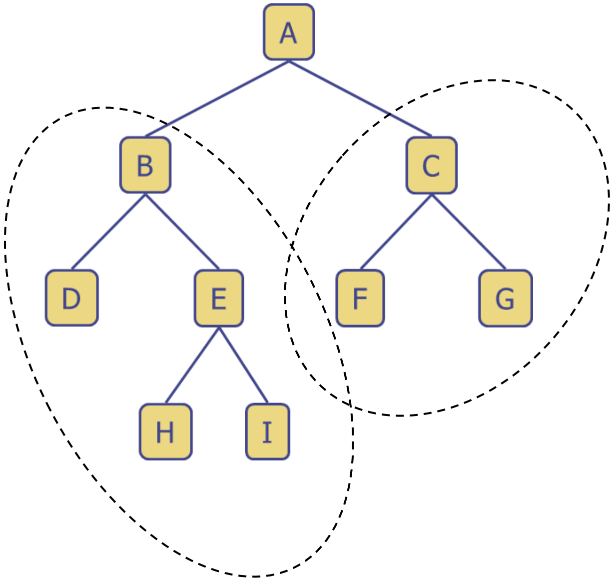
Árvore Binária Imprópria

- Uma árvore é **imprópria** se não for própria, ou seja, a árvore tem pelo menos um nó com apenas um filho.



Definição Recursiva

- Uma **árvore binária** é:
  - Uma árvore que consiste de apenas um nó, ou
  - Uma árvore cuja raiz tem um par ordenado de filhos, onde cada qual é uma árvore binária.



ADT – Árvore Binária

- A árvore binária estende a ADT Árvore, isto é , herda todos os métodos vistos no capítulo anterior (árvores genéricas).

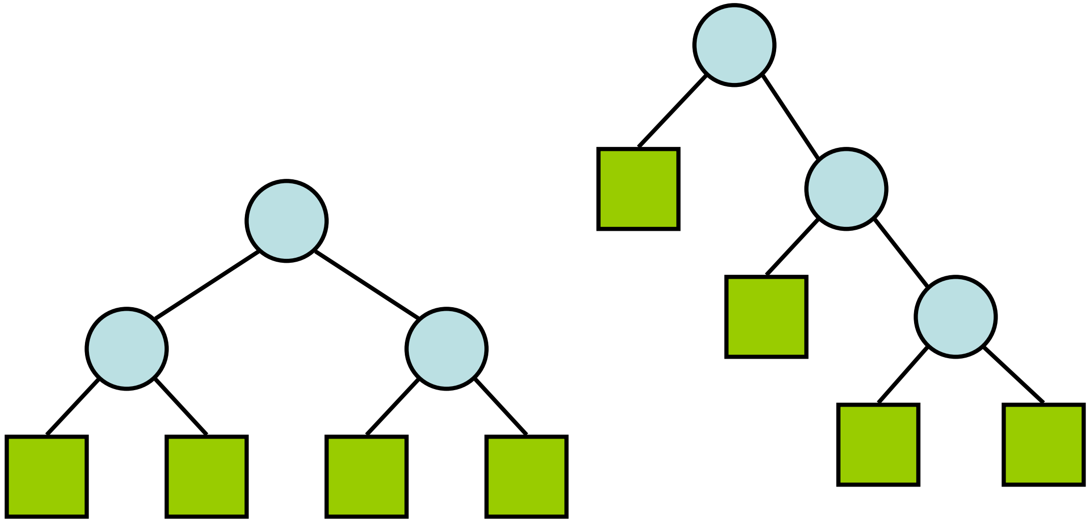
- Adicionalmente, suporta os seguintes métodos:

```
left(): retorna o filho esquerdo de um nó
right(): retorna o filho direito de um nó
hasLeft(): testa se o nó tem filho a esquerda
hasRight(): testa se o nó tem filho a direita
inorder(): percurso inorder
```

Árvore Binária Própria - Propriedades

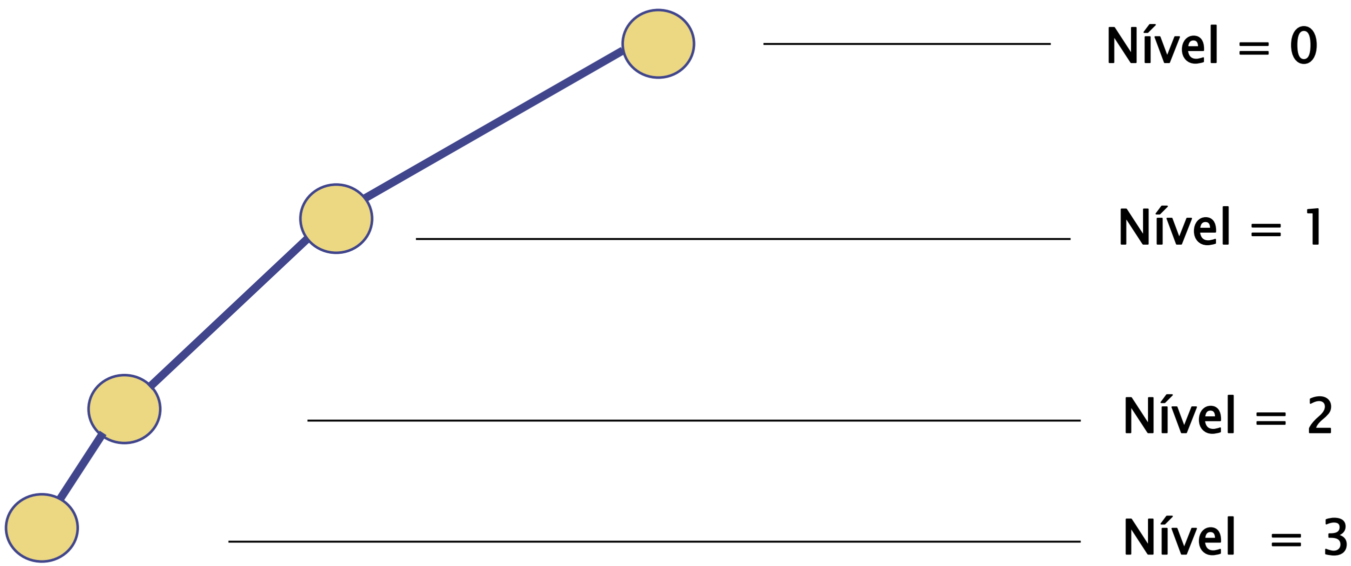
- Notação
  - ***n*** número de nós
  - ***e*** número de nós externos
  - ***i*** número de nós internos
  - ***h*** altura
  - ***b*** número de arestas

- Propriedades
  - $\mathbf{e} = \mathbf{i} + 1$
  - $\mathbf{n} = 2\mathbf{e} - 1$
  - $\mathbf{h} \leq \mathbf{i}$
  - $\mathbf{h} \leq (\mathbf{n} - 1)/2$
  - $\mathbf{e} \leq 2^{\mathbf{h}}$
  - $\mathbf{h} \geq \log_2 \mathbf{e}$
  - $\mathbf{h} \geq \log_2 (\mathbf{n} + 1) - 1$



Número mínimo de nós

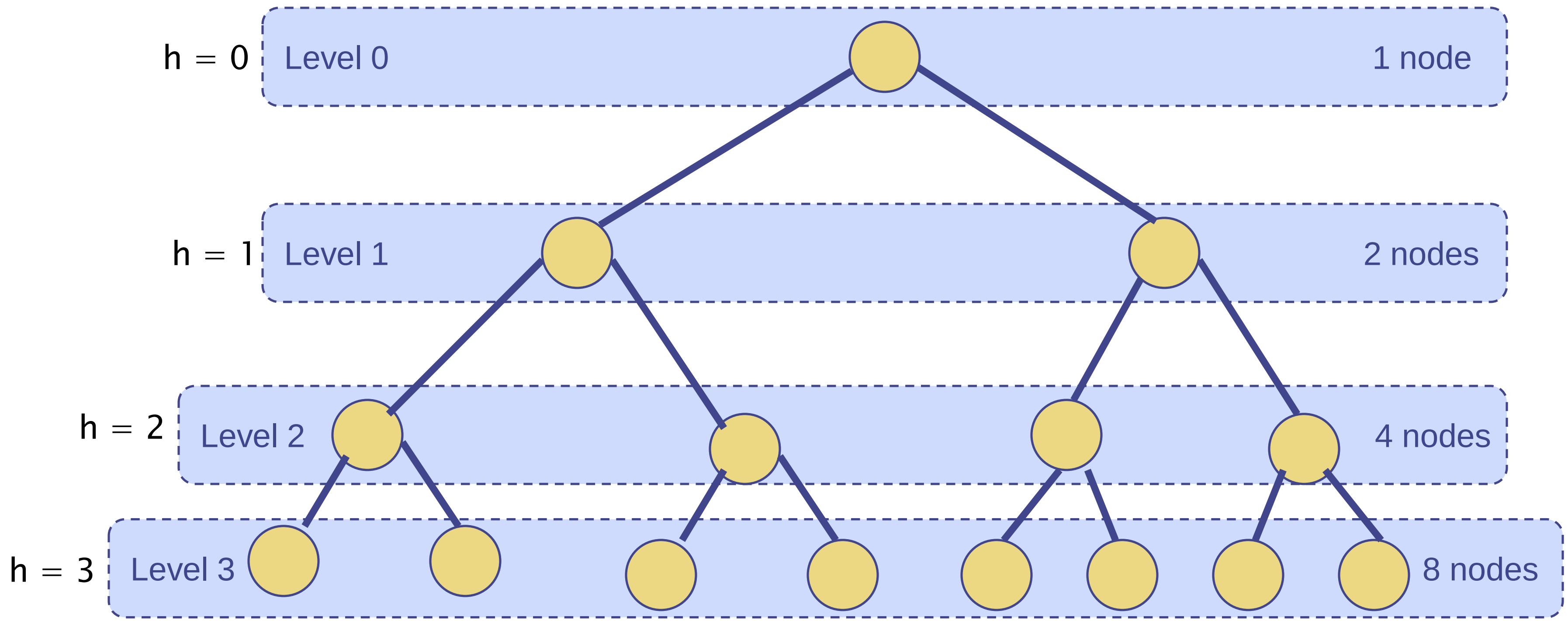
- O número mínimo de nós em uma árvore binária de altura **h**, é **n ≥ h+1**.
- Ao menos um nó em cada um dos níveis d.



Número mínimo de nós é ***h+1***

- **Altura de um nó: Tamanho do caminho de n até seu mais profundo descendente.**

Máximo número de nós

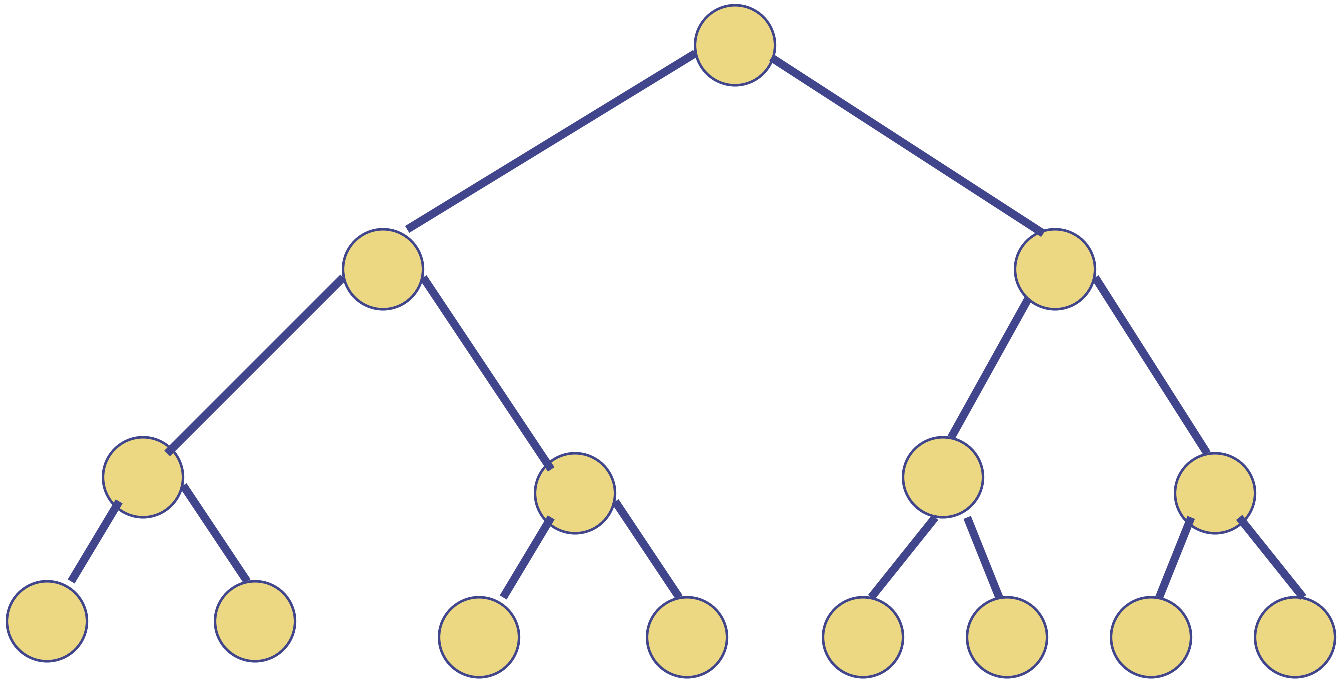


Máximo número de nós $= 1 + 2 + 4 + 8 + \ldots + 2^h$

$$n \leq 2^{h+1} - 1$$

Árvore Binária Completa (Full)

Uma árvore binária completa de altura *h* tem $2^{h+1} - 1$ nós.



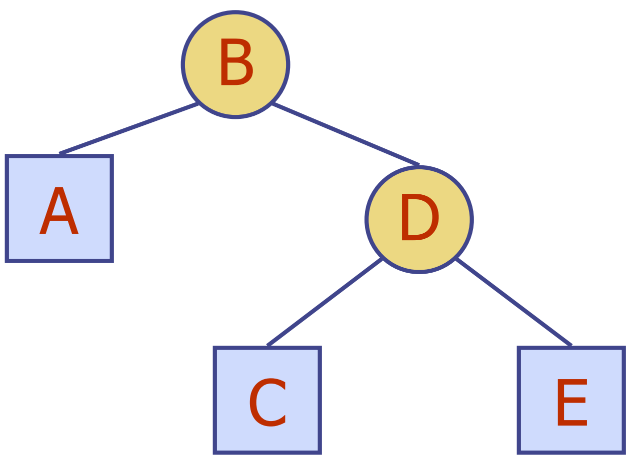
Árvore binária completa de altura 3

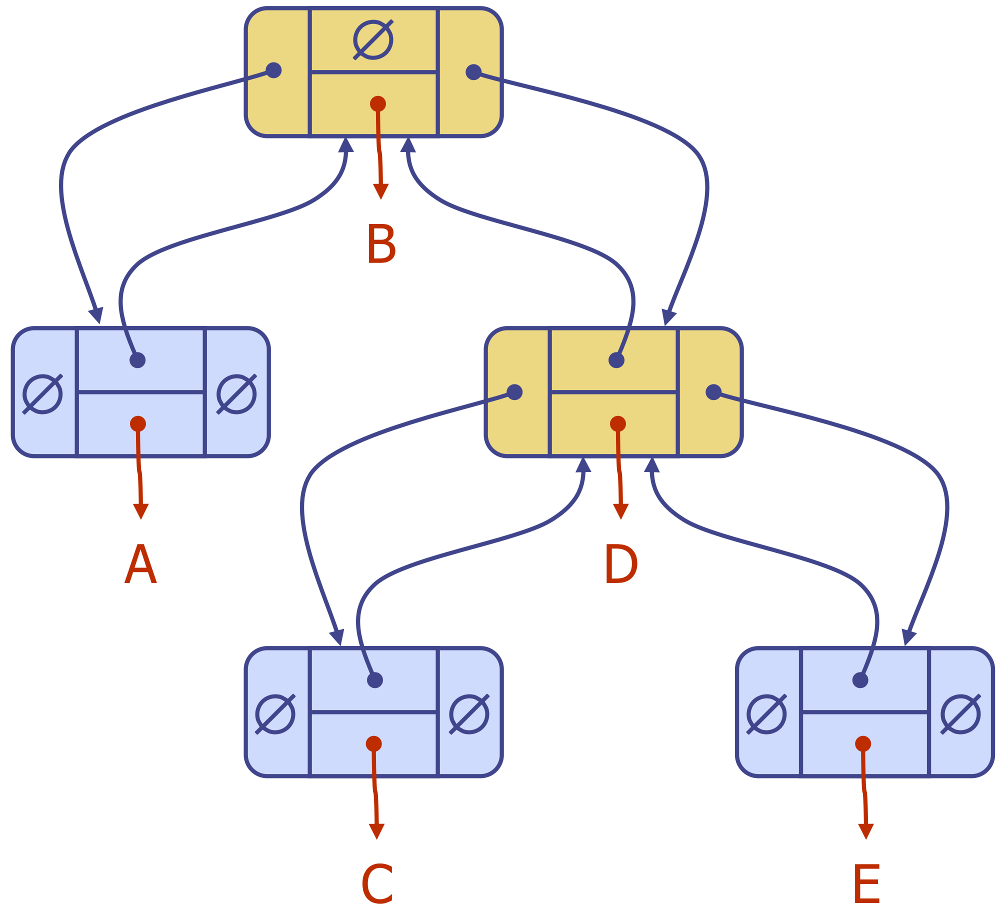
Representação de árvores binárias

1. **Linked Structure**
2. **Array List**

Representação por lista ligada

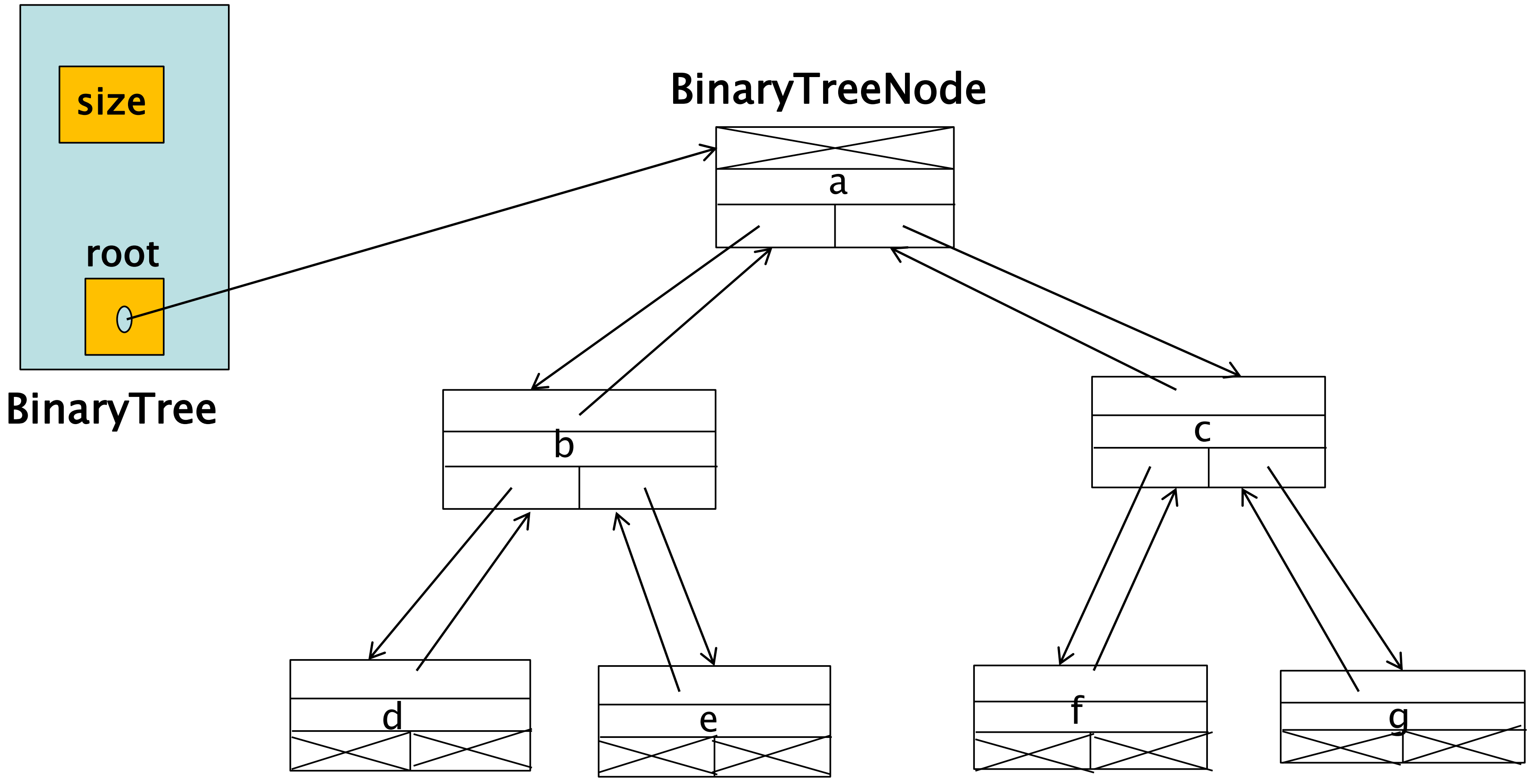
- Um nó é representado por um objeto armazenando:
  - Elemento
  - Nó pai
  - Nó Left child
  - Nó Right child





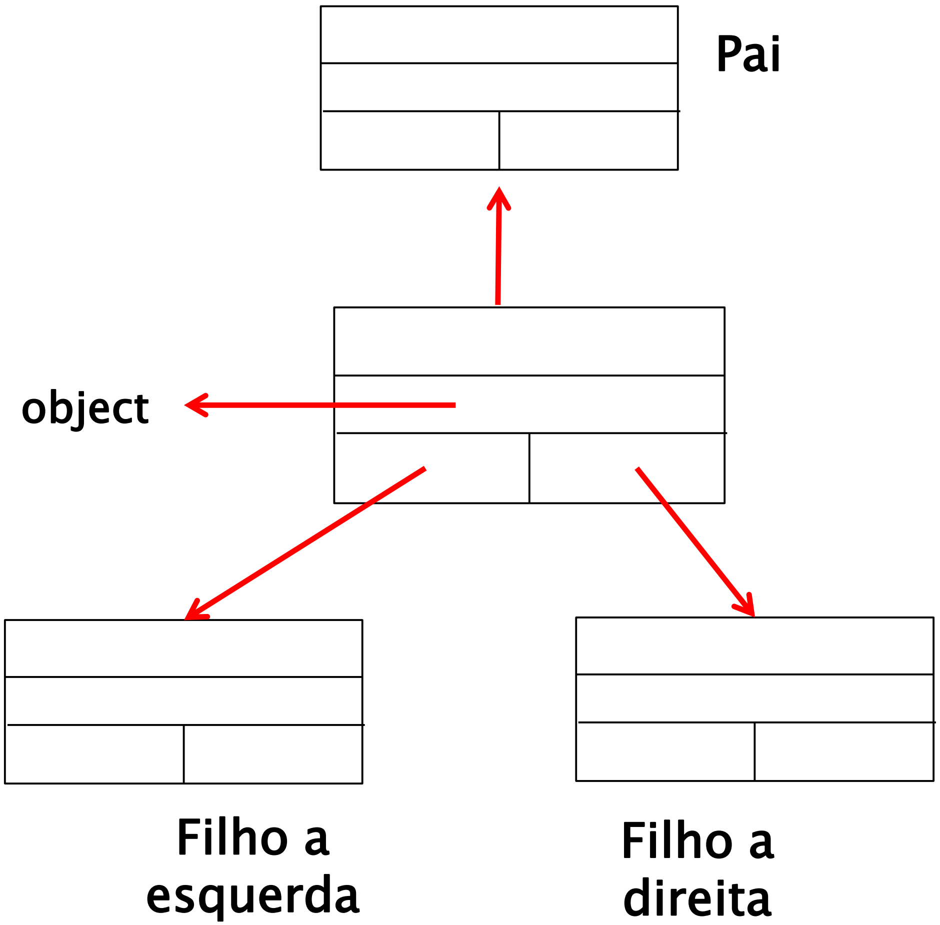
Representação por lista ligada

size -> #nós da árvore



Representando nó da Árvore

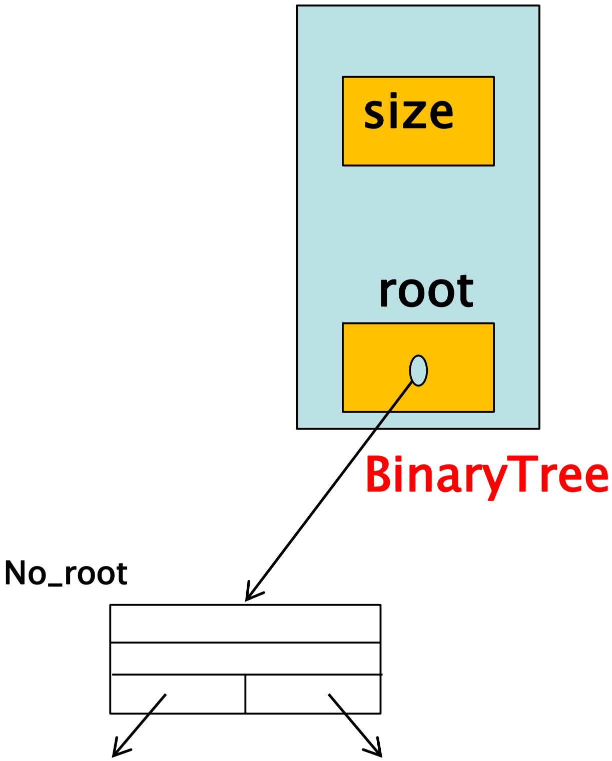
- Cada nó tem quatro referências: item, pai, filho a esquerda e filho a direita.




```
Class BinaryTreeNode {

        Object item;
        BinaryTreeNode parent;
        BinaryTreeNode left;
        BinaryTreeNode right;

}
```

Representando a Árvore

Size-> #nós da árvore

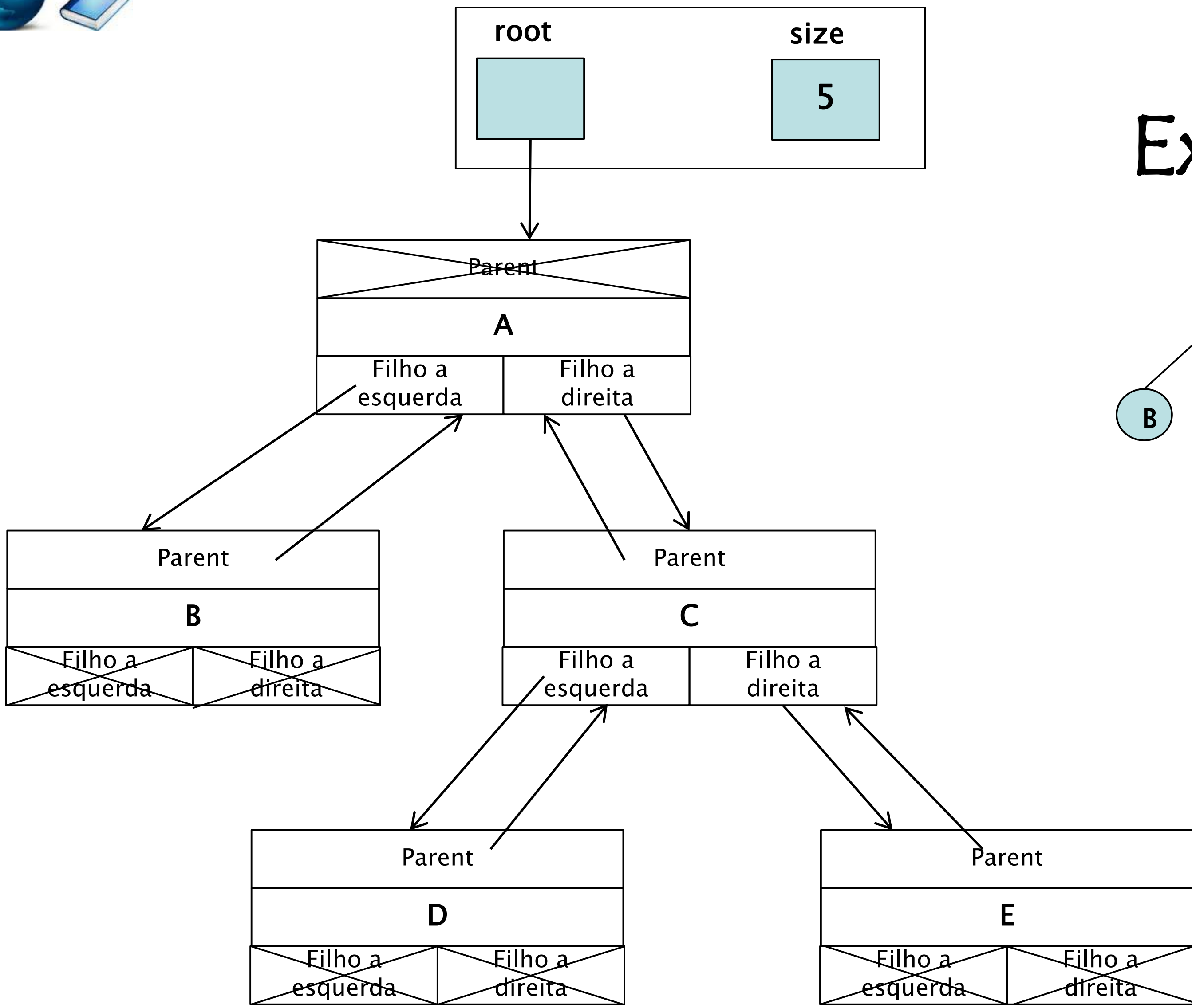


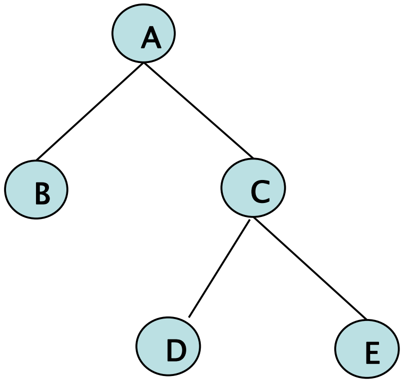

```
Class BinaryTree {

        BinaryTreeNode root;
        int size;

}
```

Representando Nó da árvore

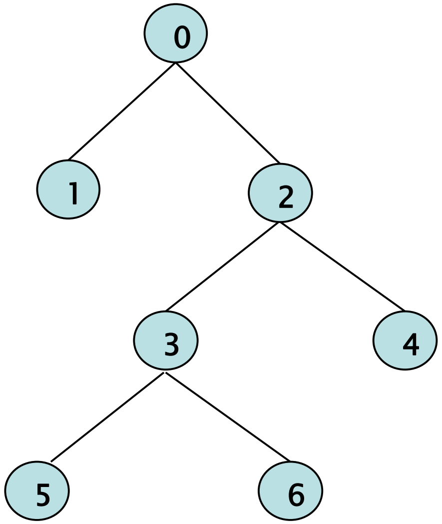
<table>
<tr><td colspan="2">Parent</td></tr>
<tr><td colspan="2">Item</td></tr>
<tr><td>Filho a esquerda</td><td>Filho a direita</td></tr>
</table>

Exemplo


root
size
5
Parent
A
Filho a esquerda
Filho a direita
Parent
B
Filho a esquerda
Filho a direita
Parent
C
Filho a esquerda
Filho a direita
Parent
D
Filho a esquerda
Filho a direita
Parent
E
Filho a esquerda
Filho a direita


A
B
C
D
E

Exemplo



```java
public class BinaryTree {

        BinaryTreeNode root;
        int size;

        public BinaryTree() {

                this.root = null;
                this.size = 0;
        }

        public void insert_root(int valor) {

                BinaryTreeNode node = new BinaryTreeNode(valor);
                this.root = node;
                this.size = 1;

        }
```

```java
    public BinaryTreeNode ret_Root() {
          return (this.root);
    }

    public int size() {
          return this.size;
    }

    public boolean isEmpty() {

          if (this.size == 0 )
                return true;
          else return false;
    }
}
```

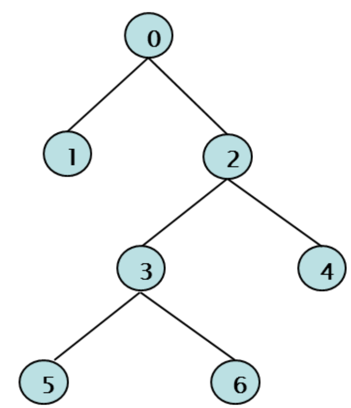
```java
package maua;

	public class BinaryTreeNode {

		int item;
		BinaryTreeNode parent;
		BinaryTreeNode left;
		BinaryTreeNode right;


	public BinaryTreeNode(int item) {

		this.item = item;
		this.parent = null;
		this.left = null;
		this.right = null;

}
```

```
public BinaryTreeNode left() {
        if (this.left == null)
                return null;
        else return this.left ;

}

public boolean  isLeft() {
        if (this.left == null)
                return false;
        else return true ;

}

public BinaryTreeNode right() {
        if (this.right == null)
                return null;
        else return this.right ;

}
```

```java
public boolean  isRight() {
        if (this.right == null)
                return false;
        else return true ;

}

public void binaryPreorder() {

        System.out.println(this.item);
        if (this.isLeft())
                this.left.binaryPreorder();
        if (this.isRight())
                this.right.binaryPreorder();
}
```

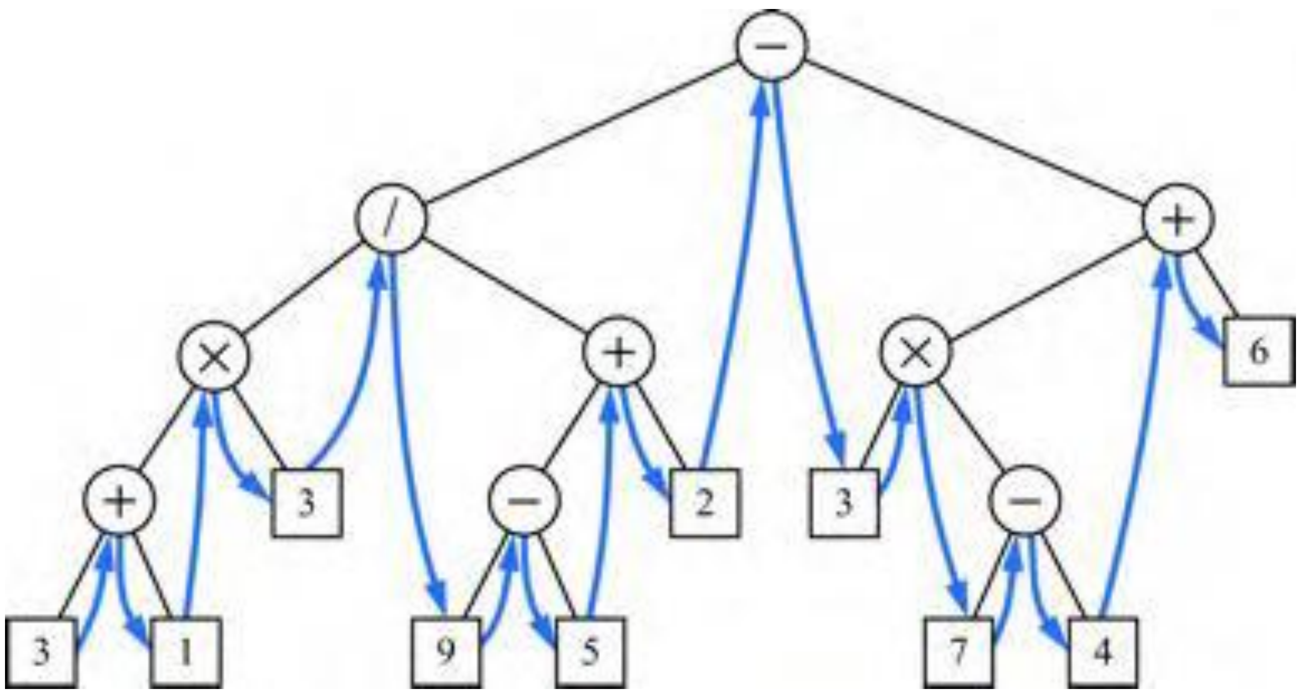
```
    public void binaryPostorder() {


            if (this.isLeft())
                    this.left.binaryPostorder();
            if (this.isRight())
                    this.right.binaryPostorder();
            System.out.println(this.item);

    }

    public void binaryInorder() {


            if (this.isLeft())
                    this.left.binaryInorder();
            System.out.println(this.item);
            if (this.isRight())
                    this.right.binaryInorder();
    }

}
```

```
package maua;

        public class Teste_BinaryTreeNode {

        public static void main(String[] args ) {

                BinaryTree x = new BinaryTree();
                x.insert_root(0);

                BinaryTreeNode no_1 = new BinaryTreeNode(1) ;
                BinaryTreeNode no_2 = new BinaryTreeNode(2) ;
                BinaryTreeNode no_3 = new BinaryTreeNode(3) ;
                BinaryTreeNode no_4 = new BinaryTreeNode(4) ;
                BinaryTreeNode no_5 = new BinaryTreeNode(5) ;
                BinaryTreeNode no_6 = new BinaryTreeNode(6) ;

                x.root.left = no_1;
                x.root.right = no_2;
                no_2.left = no_3;
                no_2.right = no_4;
                no_3.left = no_5;
                no_3.right = no_6;

                x.root.binaryPreorder();
                x.root.binaryPostorder();
                x.root.binaryInorder();

        }

}
```

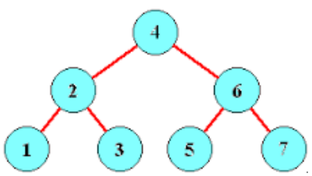


Travessia Inorder

- Representa um método de travessia adicional válido para árvores binárias.
- Nesta travessia, visitamos um nó entre as chamadas recursivas das subárvores esquerda e direita.

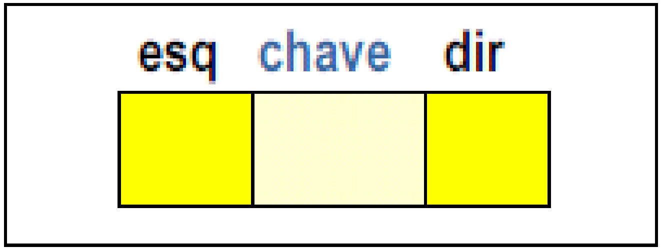


Árvore Binária de Pesquisa



- Também conhecida por:
  - Árvore Binária de Busca
  - Árvore Binária Ordenada
  - Search Tree (em inglês)

- Apresentam uma relação de ordem entre os nós.
- A ordem é definida por um campo **chave** (key).
- **Não permite chaves duplicadas**.



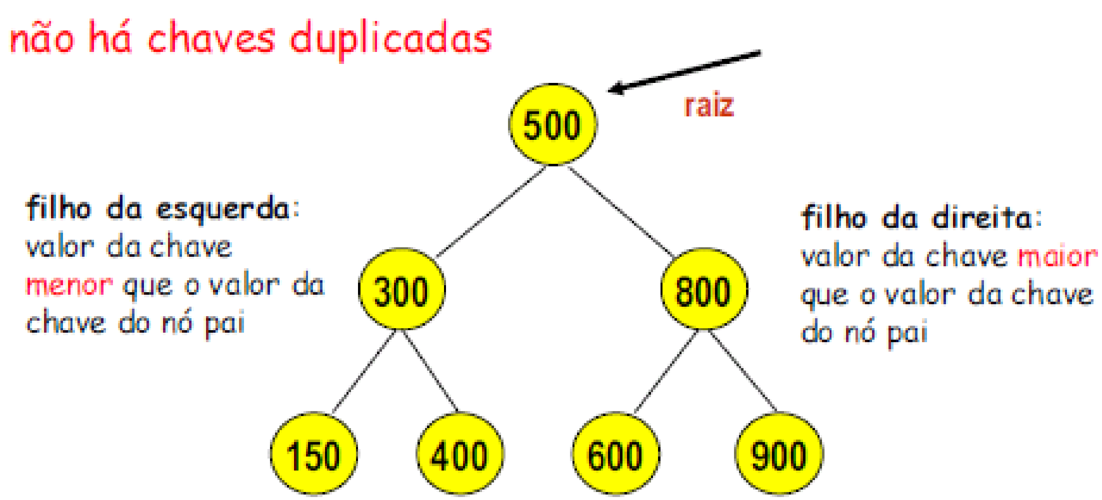
Árvore Binária de pesquisa

- **Definição de Niklaus Wirth:**

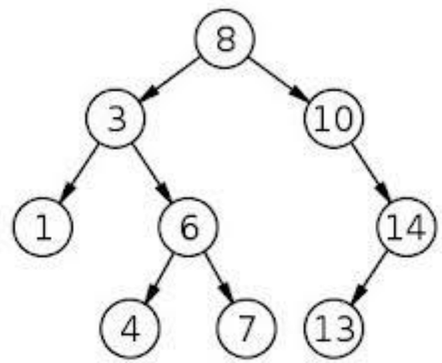
Árvore que se encontra organizada de tal forma que, para cada nó t_i , todas as chaves da sub-árvore:

à esquerda de t_i são **menores** que t_i e

à direita de t_i são **maiores** que t_i.



Inserção em Árvores Binárias de Pesquisa

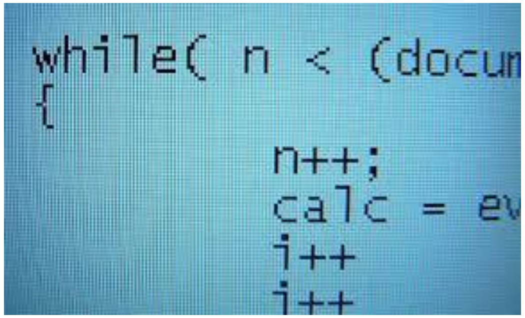


Carga da Árvore Binária de pesquisa

```
int[] valores = { 17,49,14,23,27,15,2,1,34,10,12 } ;
```

```
String[] nomes = {
"Paulo","Ana","José","Rui","Paula","Bia","Selma","Carlos","Silvia","Teo","Saul" } ;
```

A partir das listas acima, implementar a árvore binária de busca.



Inserção em uma árvore de busca binária

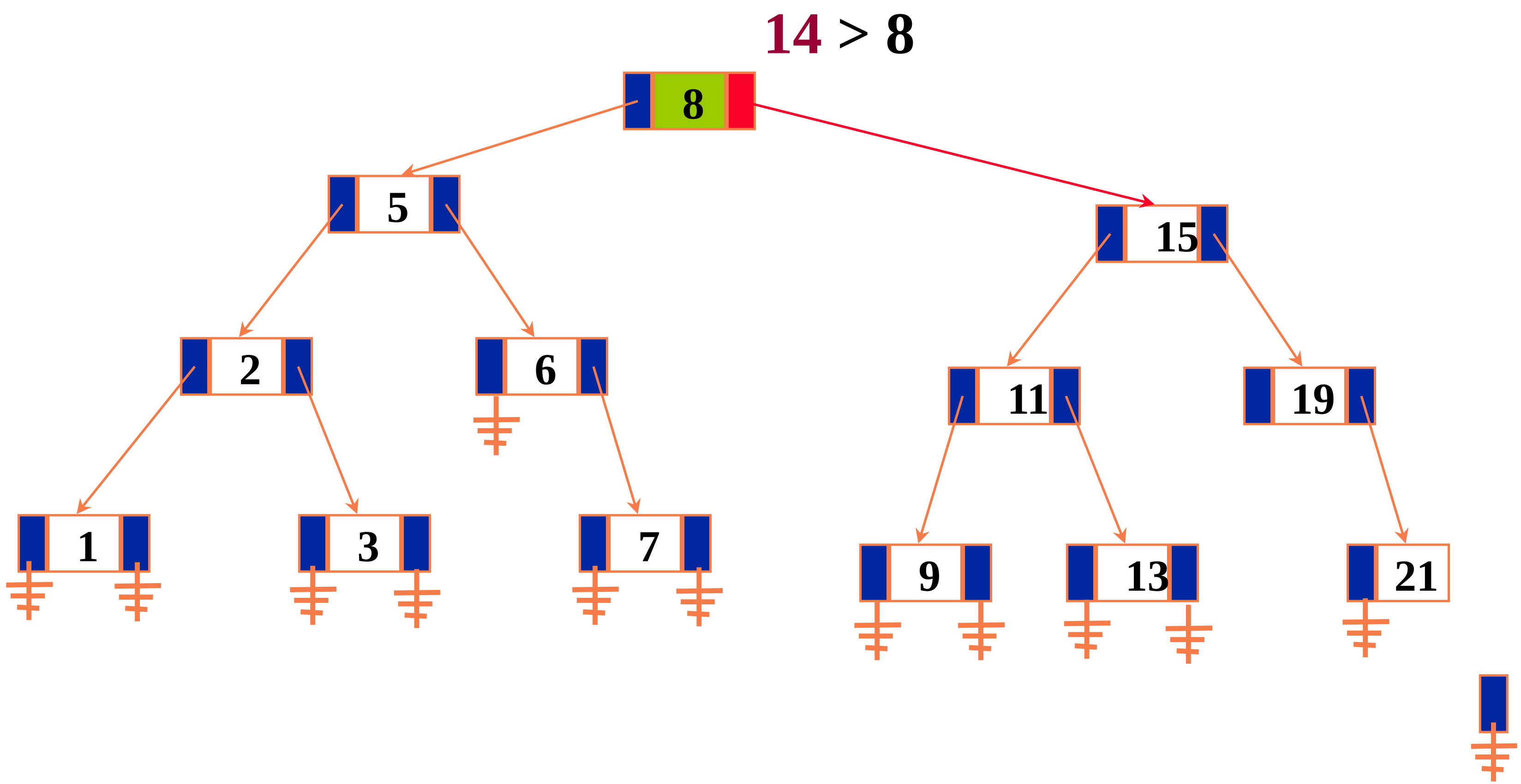
Lembrando que ...

- A sub-árvore da **direita** de um nó deve possuir chaves **maiores** que a chave do pai.
- A sub-árvore da **esquerda** de um nó deve possuir chaves **menores** que a chave do pai.

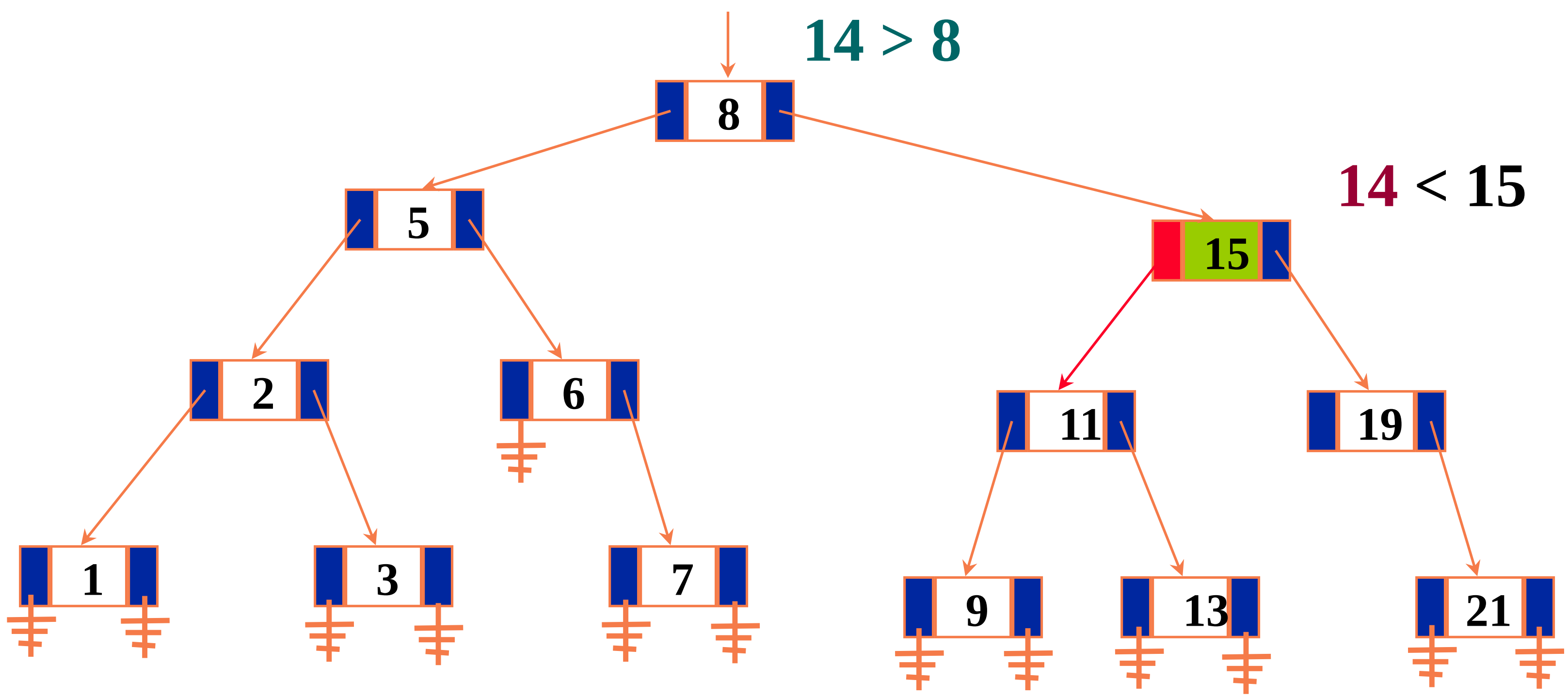
Princípio Básico

- Percorrer a árvore até encontrar um nó sem filho, de acordo com os critérios acima.

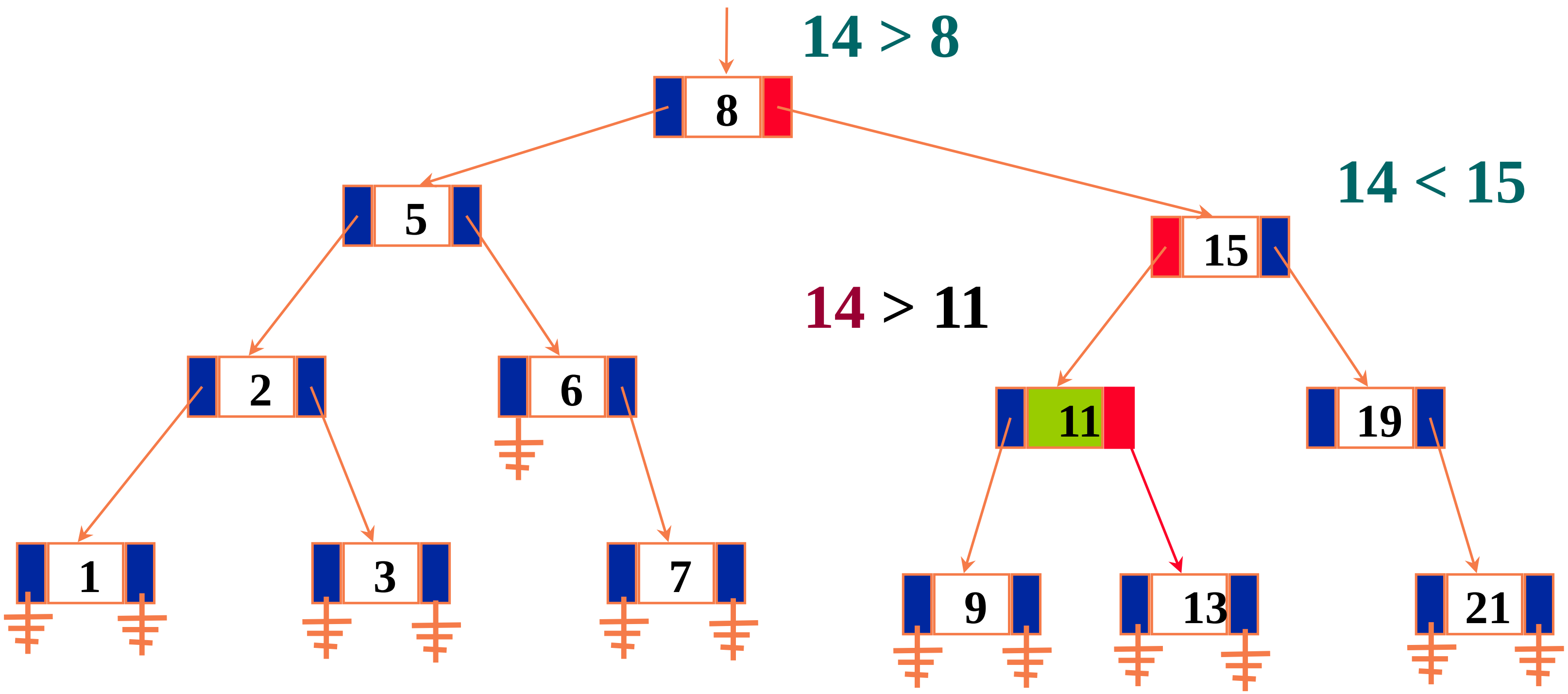
Exemplo – Inserção de elemento com chave 14



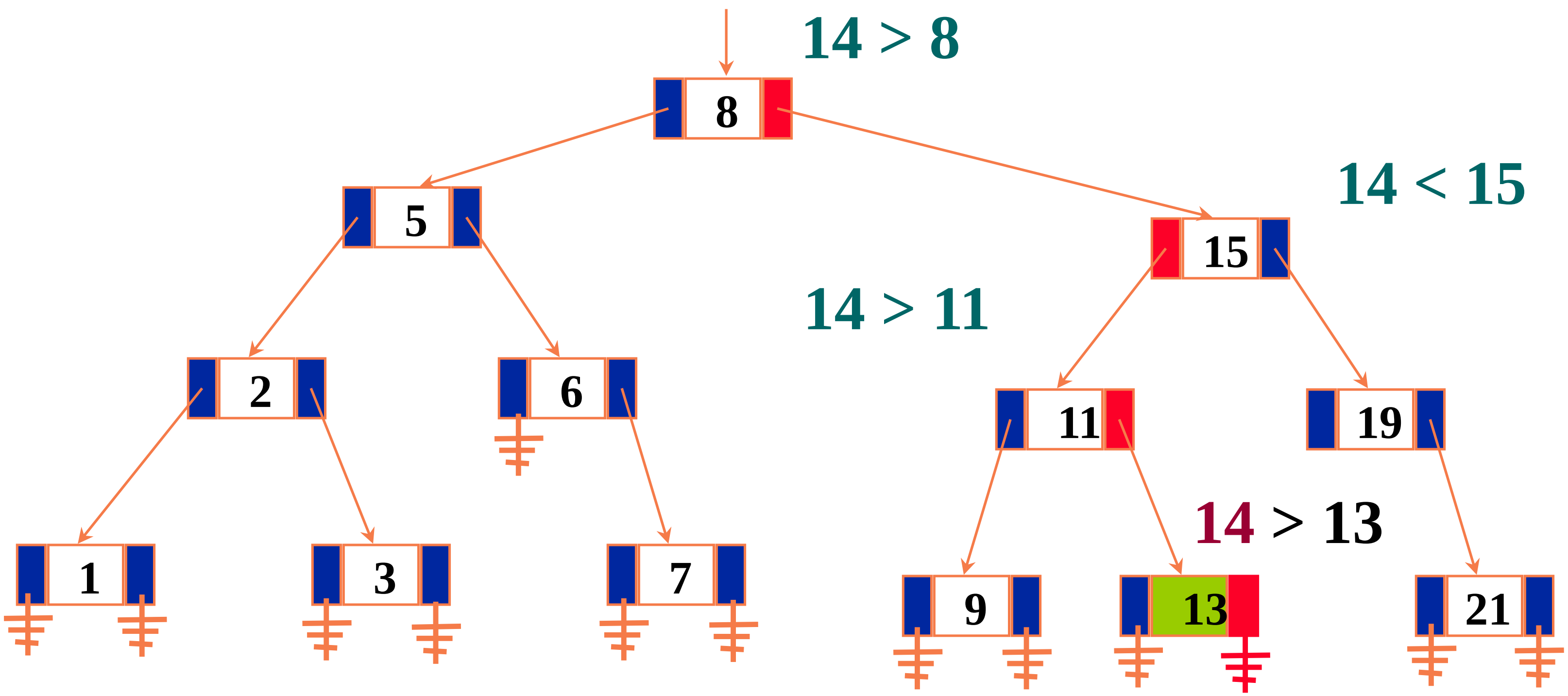
Exemplo – Inserção de elemento com chave 14



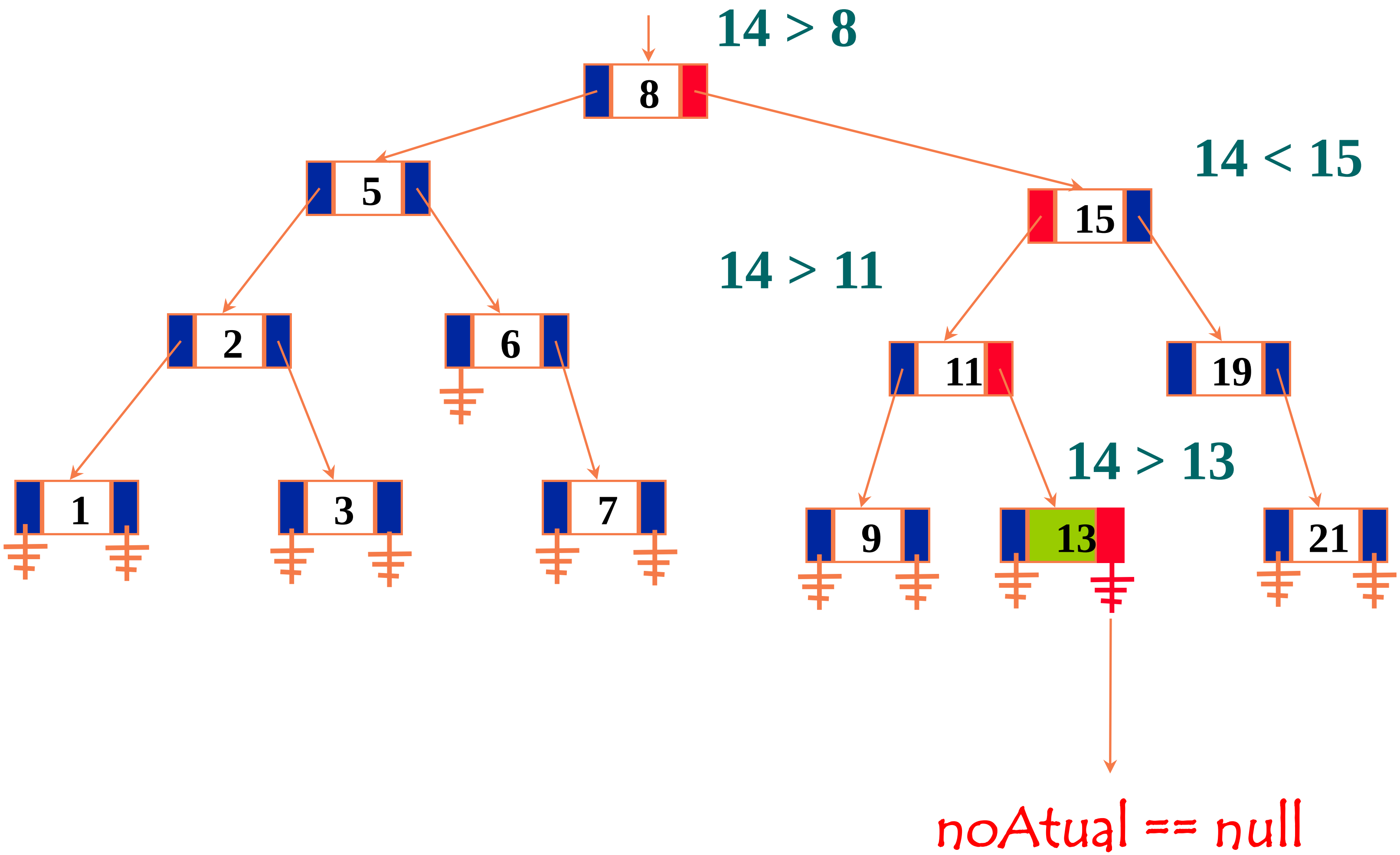
Exemplo – Inserção de elemento com chave 14



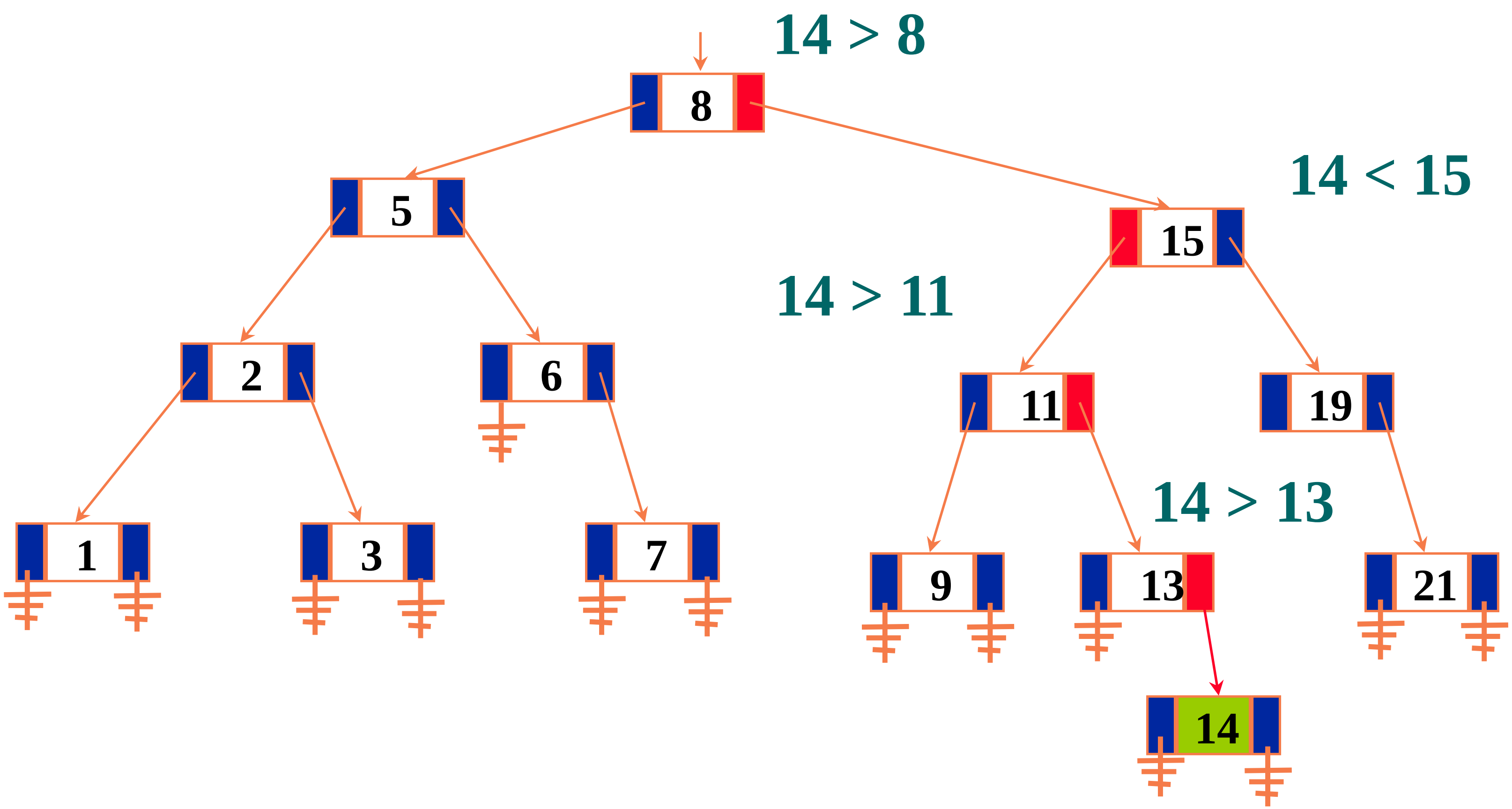
Exemplo – Inserção de elemento com chave 14



Exemplo – Inserção de elemento com chave 14



Exemplo – Inserção de elemento com chave 14



Implementação da função addnode()

```
public void addNode(int chave, String nome) {

 SearchTreeNode newNode = new SearchTreeNode(chave,nome);
 if (root == null)
    this.insert_root(newNode);
 else {
        SearchTreeNode  NodeTrab = this.root;
        NodeTrab = this.root;
        while (true) {
            if (chave < NodeTrab.key) {
                if (NodeTrab.left  == null) {
                     NodeTrab.left = newNode;
                     newNode.parent = NodeTrab;
                     newNode.nome = nome;
                     return;
                }
                else NodeTrab = NodeTrab.left;

            }
            else {
                 if (NodeTrab.right == null) {
                     NodeTrab.right = newNode;
                     newNode.parent = NodeTrab;
                     newNode.nome = nome;
                     return;
                 }
                     else NodeTrab = NodeTrab.right;
                 }
        }
 }
}
```

Busca em árvores de pesquisa

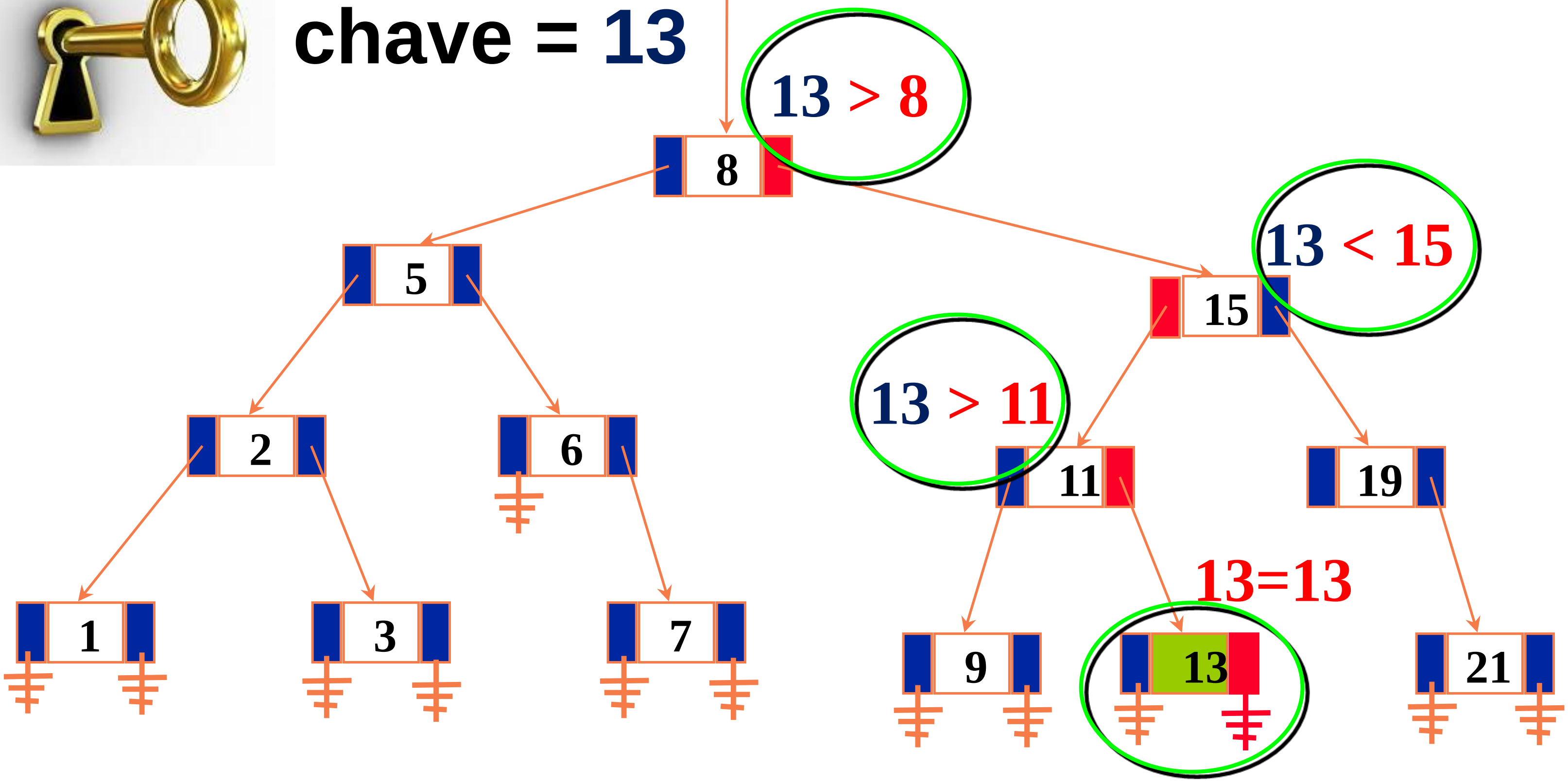
- ✓ Em uma árvore binária é possível encontrar qualquer chave existente **X** atravessando-se árvore:
  - ✓ sempre **à esquerda** se **X** for **menor** que a chave do nó visitado e
  - ✓ sempre **à direita** toda vez que for **maior**.
  - ✓ A escolha da direção de busca só depende de **X** e da chave que o nó atual possui.

- ✓ A busca de um elemento em uma árvore balanceada com **n** elementos toma tempo médio menor que $\log_2(\mathbf{n})$, tendo a busca então $O(\log_2 \mathbf{n})$.

n	log2 (n)
1	0,00
10	3,32
13	3,70
20	4,32
50	5,64
100	6,64
200	7,64
500	8,97

Exemplo – Busca da chave 13

chave = 13



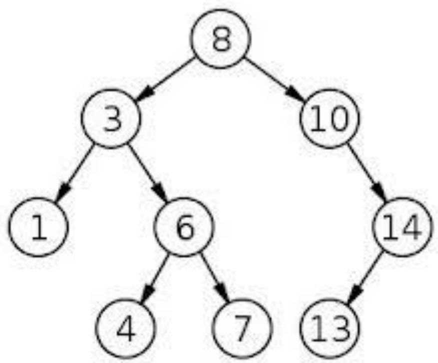
Algoritmo iterativo de search em árvore de busca

```
Node buscaChave (int chave) {
    Node noAtual = raiz; // inicia pela raiz

    while (noAtual != null && noAtual.item != chave) {

        if (chave < noAtual.key)
            noAtual=noAtual.left ; // caminha p/esquerda
        else
            noAtual=noAtual.right;  // caminha p/direita

    }
    return noAtual;
}
```

Implementação – Busca

```
public SearchTreeNode Search_key(int key) {

        SearchTreeNode nodeTrab = this.root; // inicia pela raiz

        while (nodeTrab != null && nodeTrab.key != key) {

                if (key < nodeTrab.key)
                        nodeTrab = nodeTrab.left;
                else
                        nodeTrab = nodeTrab.right;
                }
        return nodeTrab;
}
```

Eliminação em Árvores Binárias de Busca



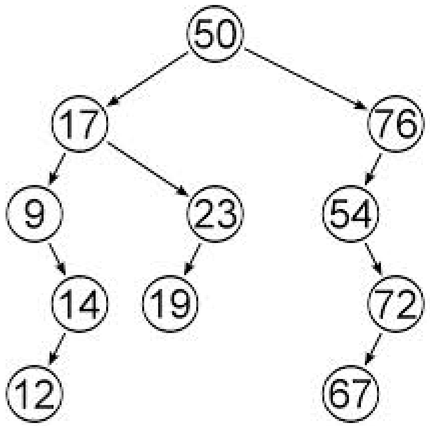
Eliminação em Árvores Binárias de Busca

- ✓ A eliminação é mais complexa do que a inserção.

- ✓ A razão básica é que a característica organizacional da árvore não deve ser alterada.

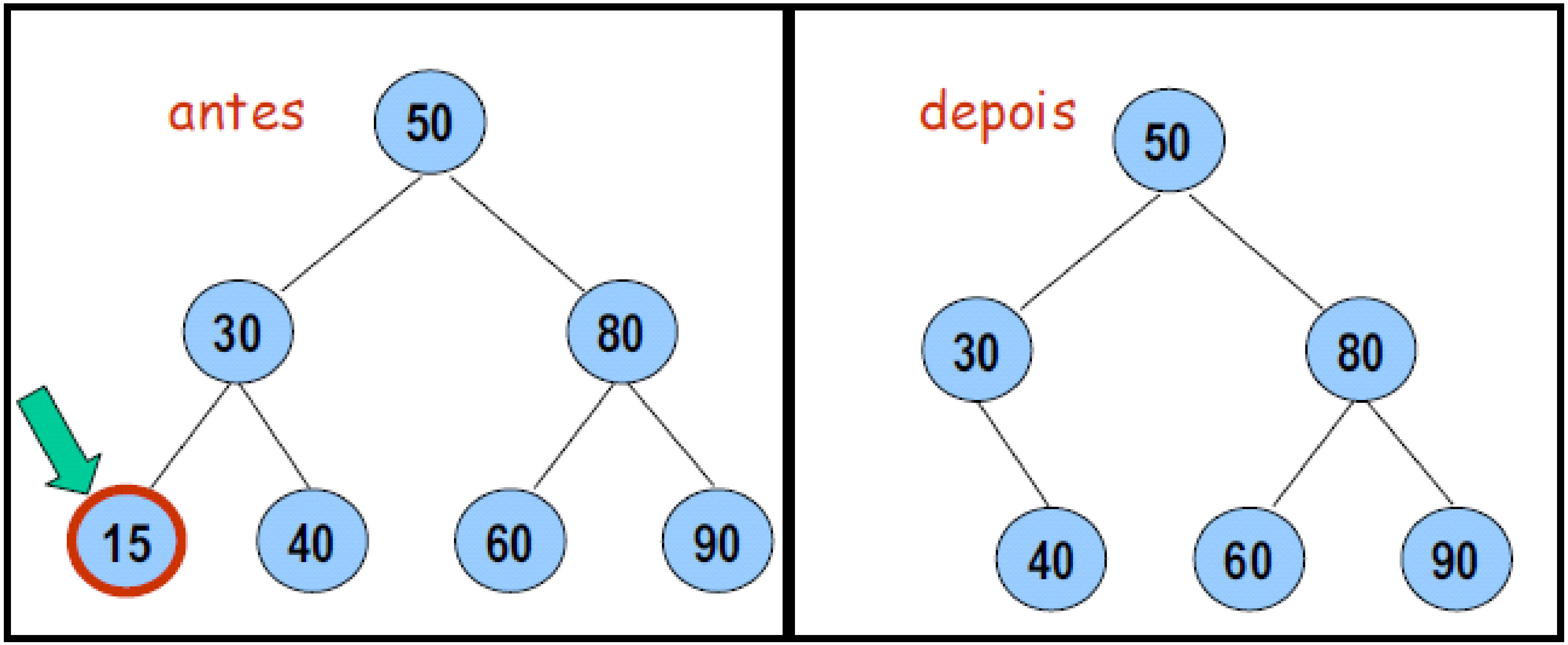
  - A sub-árvore **direita** de um nó deve possuir chaves **maiores** que a do pai.
  - A sub-árvore **esquerda** de um nó deve possuir chaves **menores** que a do pai.

- ✓ Para garantir isto, o algoritmo deve "**remanejar**" os nós.



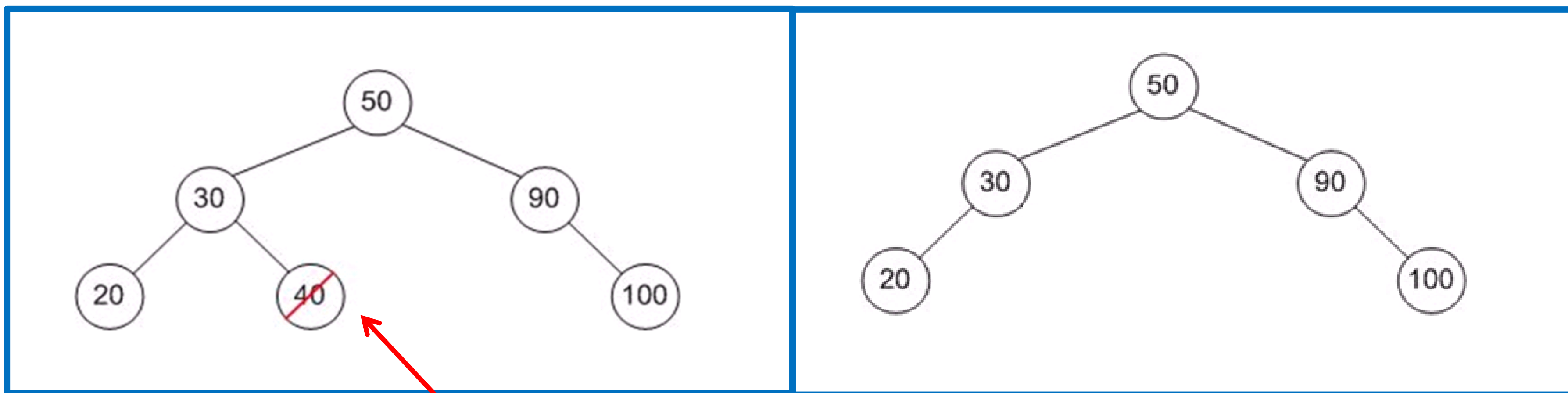
Caso 1 – Remoção de nó folha

Caso mais simples, basta retirá-lo da árvore



Eliminação da chave 15

Caso 1 – Outro exemplo

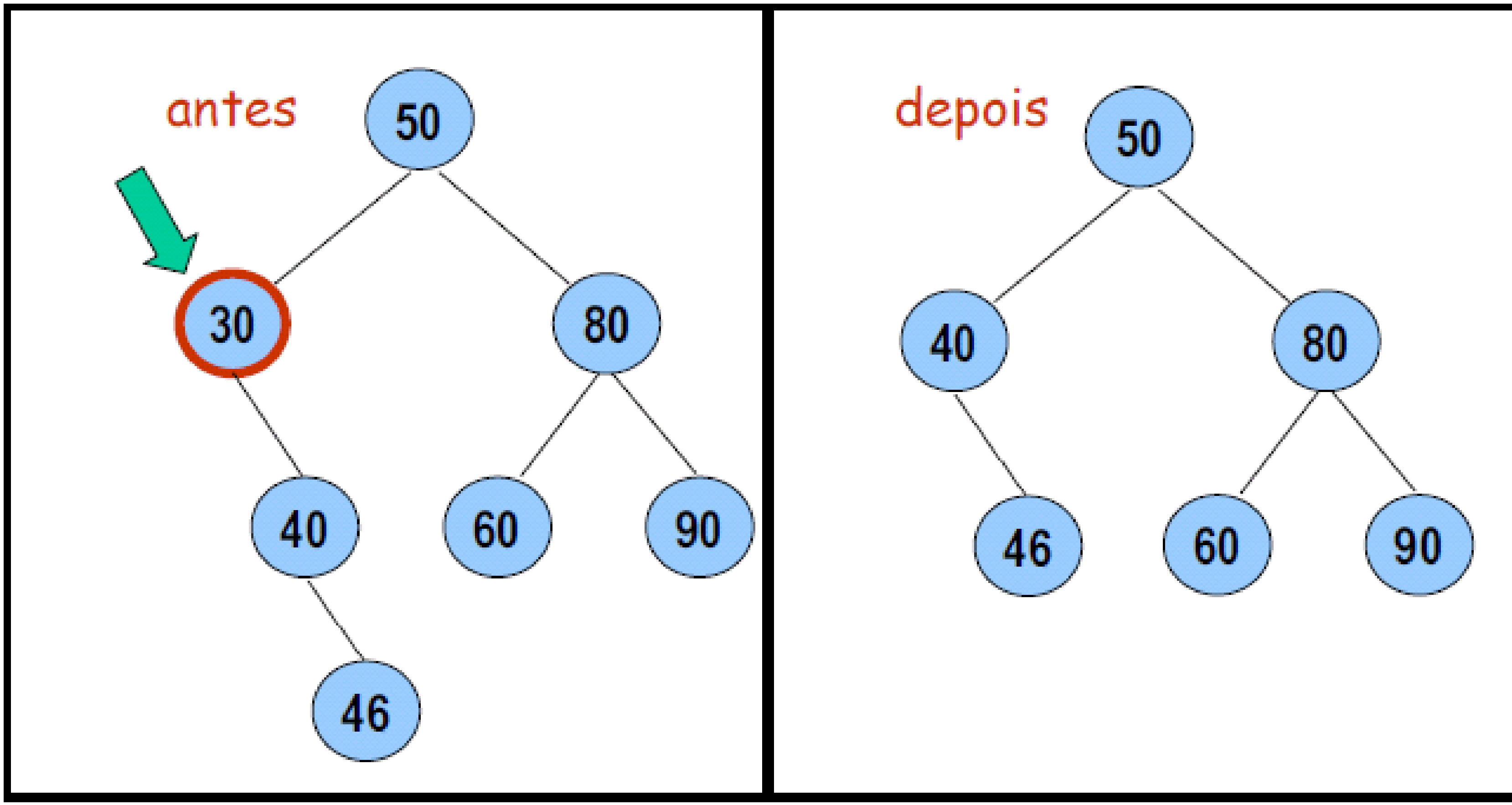


Eliminação da chave 40

Eliminação de nó que possui uma sub-árvore filha

- ✓ Se o nó a ser removido possuir somente uma sub-árvore filha:
  - *Move-se essa sub-árvore toda para cima.*
  - *Se o nó a ser excluído é filho esquerdo de seu pai, o seu filho será o novo filho esquerdo deste e vice versa.*

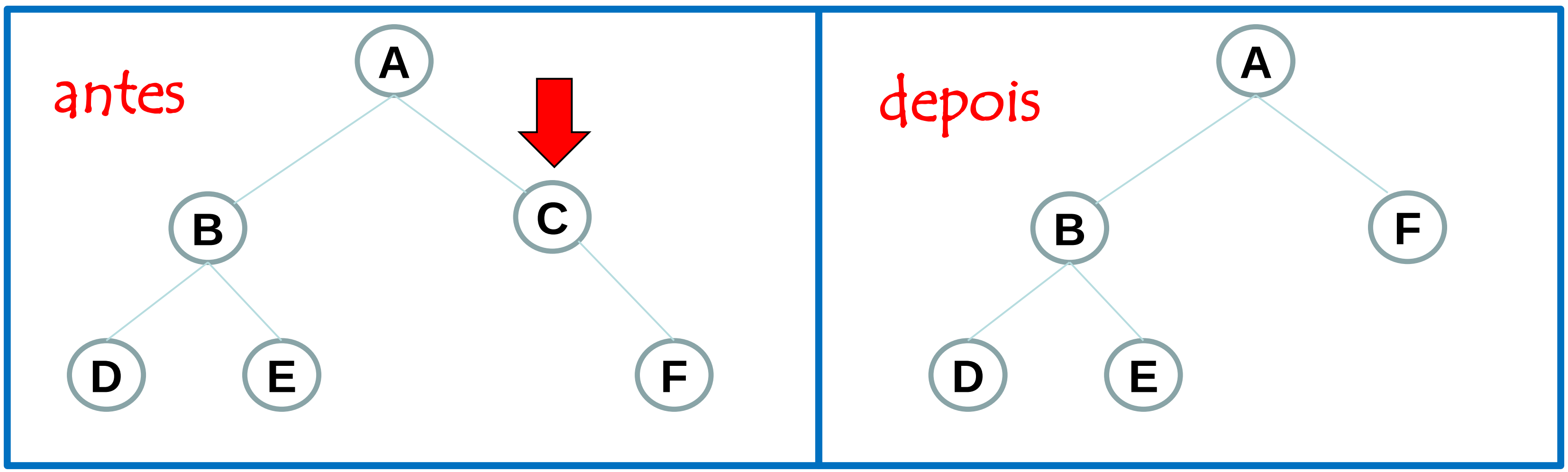
Caso 2 – O nó tem somente uma sub-árvore



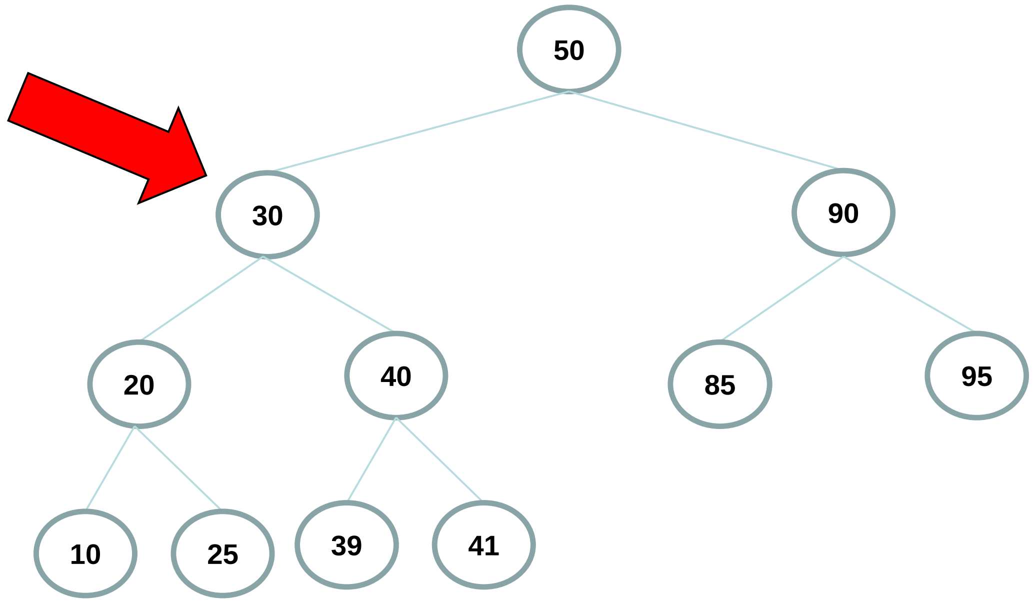
Eliminação da chave 30

O ponteiro do pai aponta para o filho deste

Caso 2 – Outro Exemplo



Eliminação de nó que possui duas sub-árvores filhas

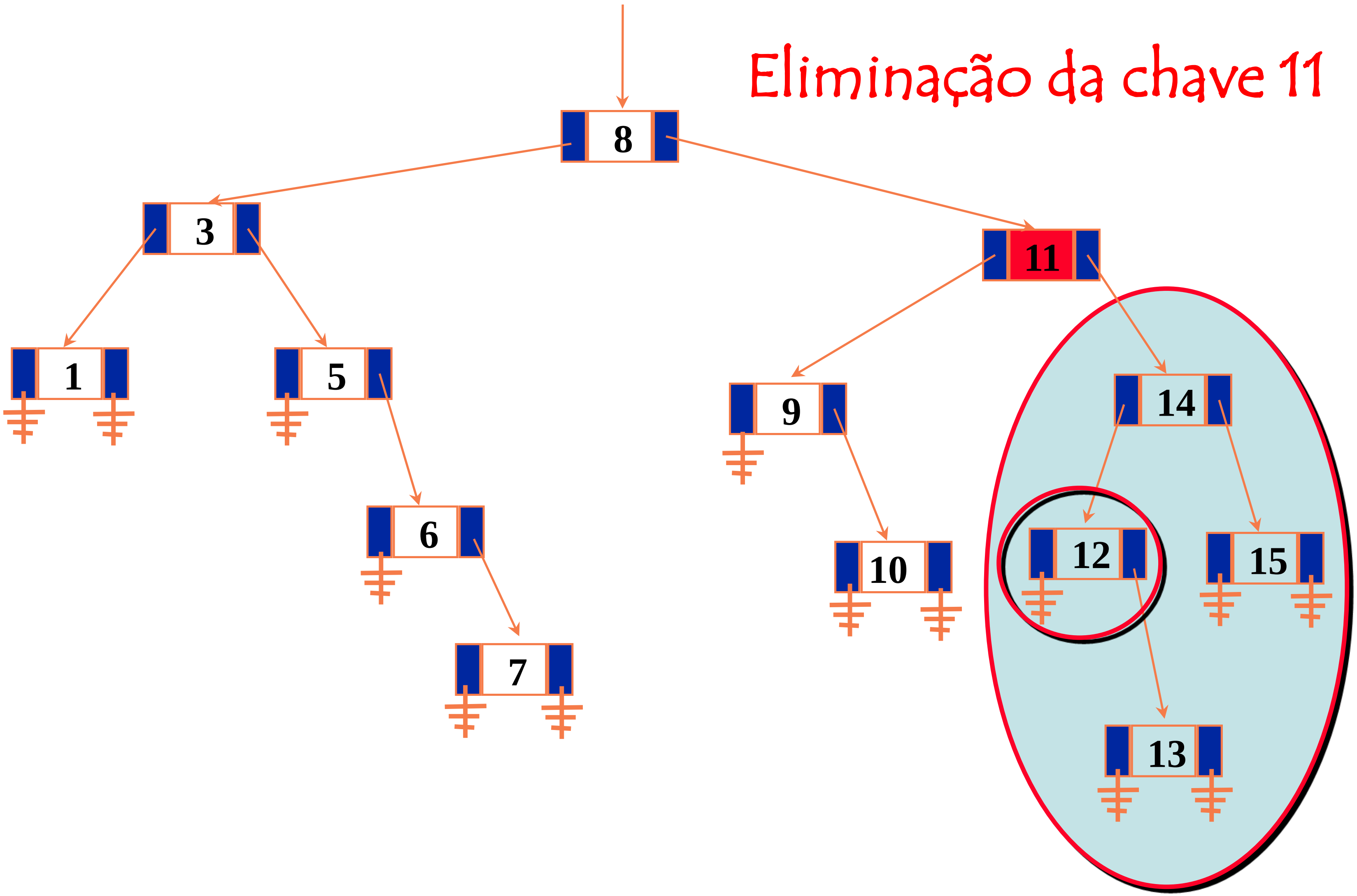


<u>A estratégia geral (Mark Allen Weiss)</u> é:

Substituir a chave retirada pela menor chave da sub-árvore direita

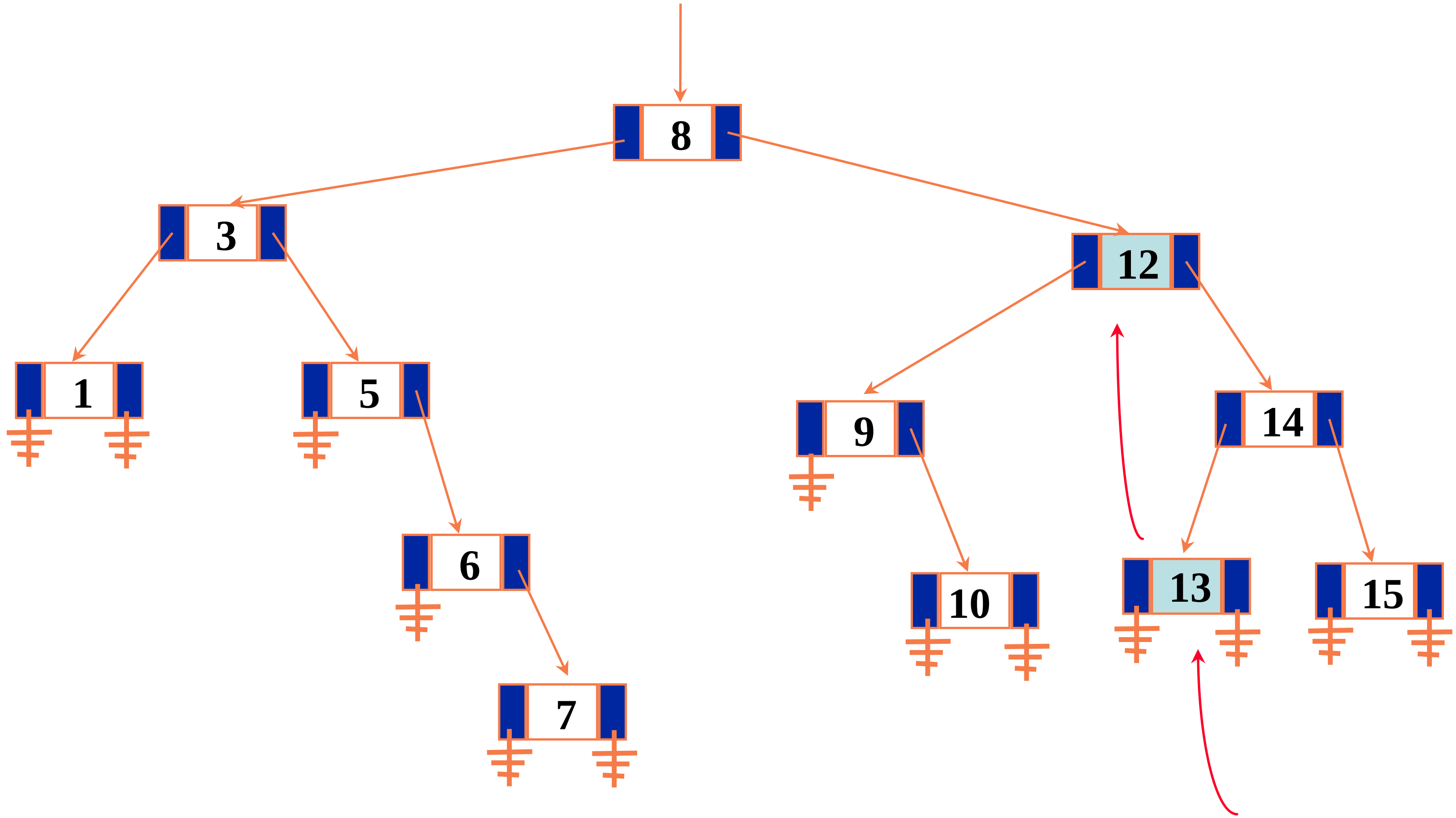
Caso 3 – Nó tem duas sub-árvores

Eliminação da chave 11



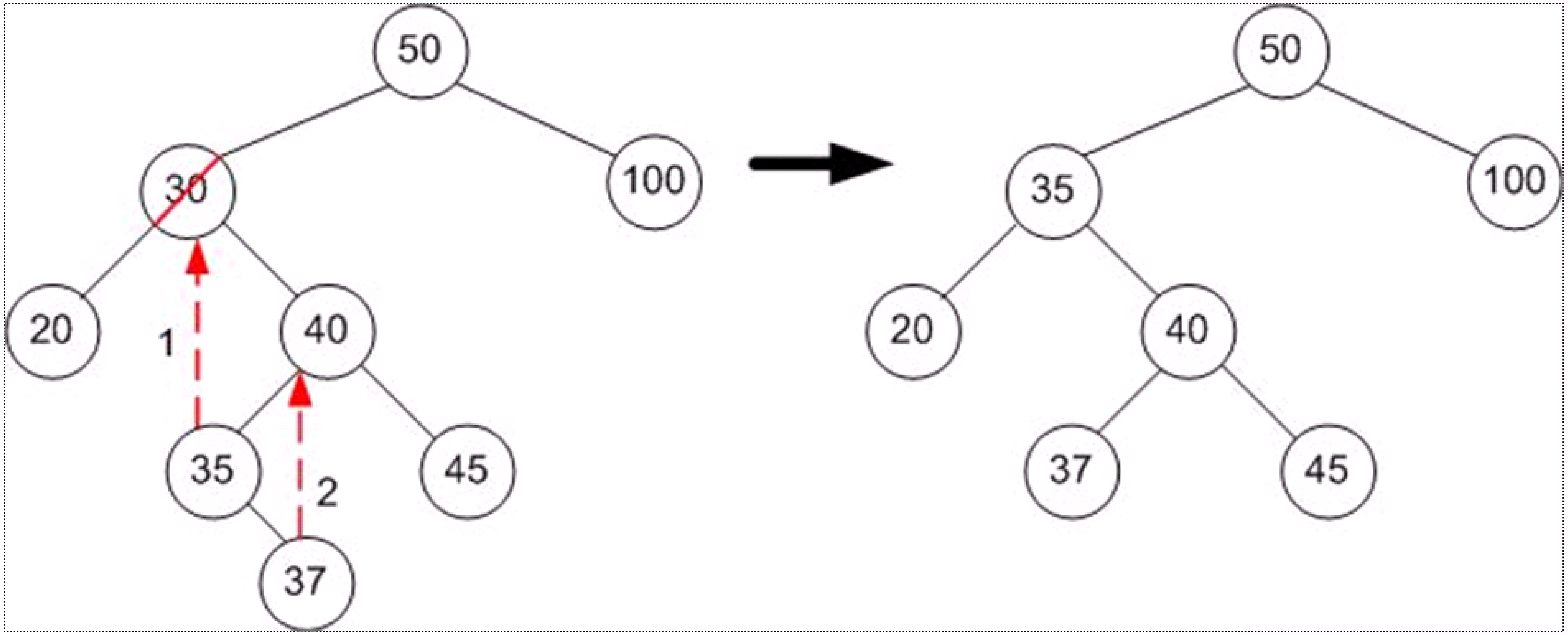
Substituir a chave retirada pela menor chave da sub-árvore direita

Caso 3 – Nó tem duas sub-árvores



Eliminação da chave 11

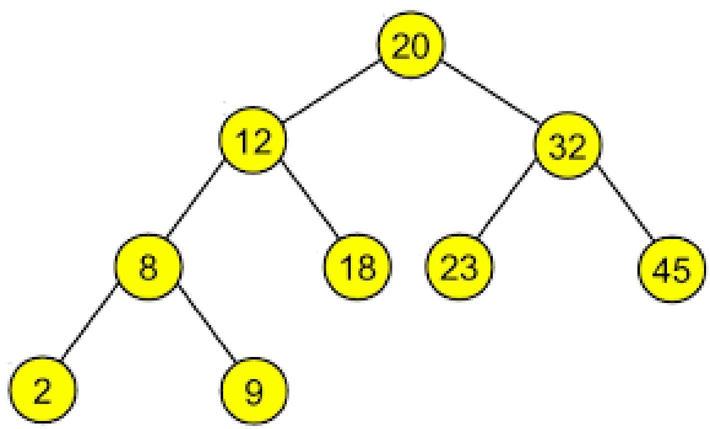
Caso 3 – Outro exemplo



Eliminação da chave 30

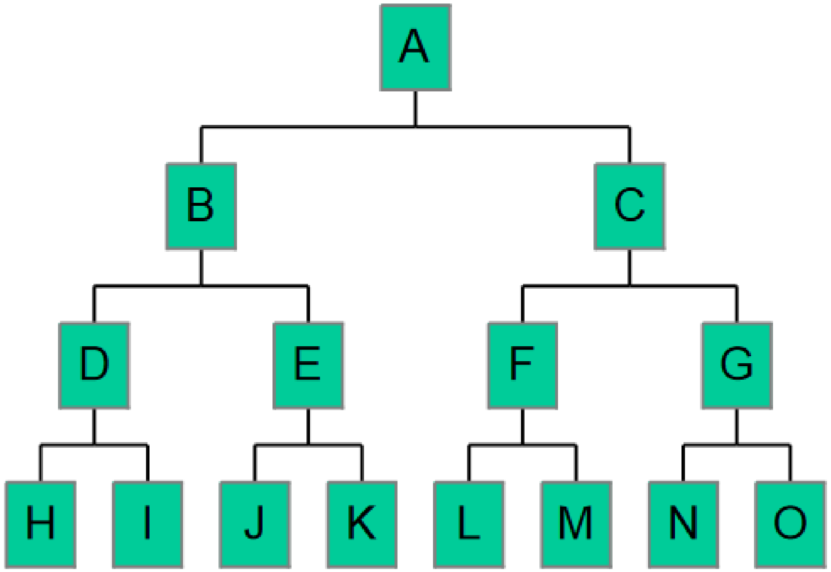
Árvore AVL

- ✓ É uma árvore de busca binária autobalanceada;
- ✓ Em tal árvore, as **alturas** das suas sub-árvores a partir de cada nó diferem de no **máximo 1 unidade**;
- ✓ O nome **AVL** vem de seus criadores (Adelson Velsky e Landis)



Buscas com árvores

- **Para grandes volumes de dados**, o emprego de árvores binárias de pesquisa trás o inconveniente de apresentarem **grande altura**.



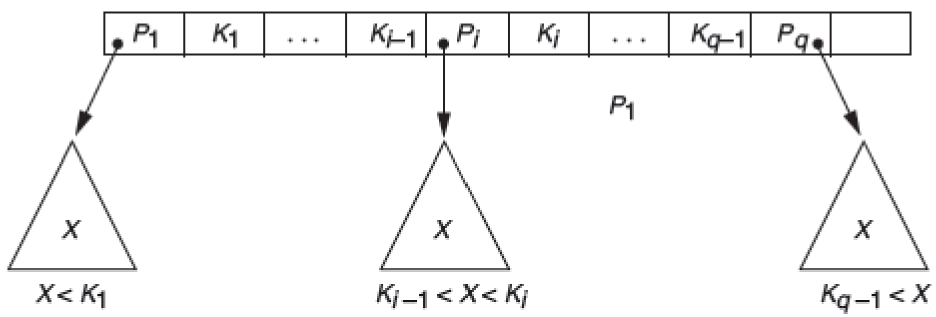
- Indexação em grandes volumes de dados tem maior eficiência com o emprego de **Árvores n-árias de pesquisa.**

Árvores n-ária de Pesquisa

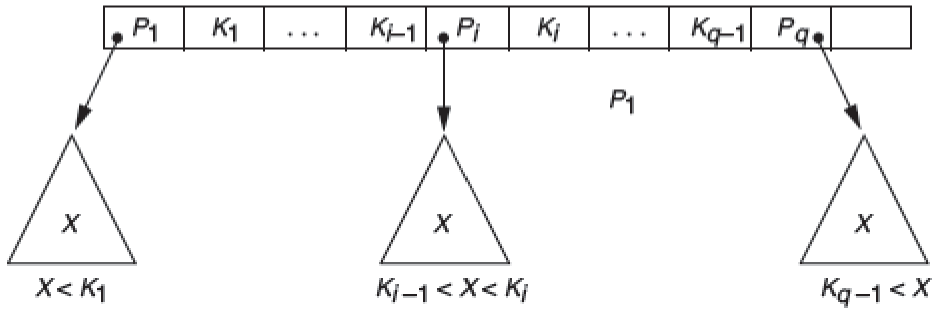
- Uma árvore n-ária de pesquisa de ordem **p** é uma árvore tal que cada nó contém **no máximo p-1** valores de pesquisa e **p** ponteiros na ordem:

P_1, K_1, P_2, K_2, . . . , P_{q-1}, K_{q-1}, P_q , onde q <= p.

- Cada **P_i** é um ponteiro para um nó filho (ou **null**), e cada **K_i** é um valor de pesquisa de algum conjunto ordenado de valores.



Árvores n-ária de Pesquisa



- Em cada nó , $K_1 < K_2 < \; . . . < K_{q-1}$

- Para todos os valores X na subárvore apontada por Pi, temos:

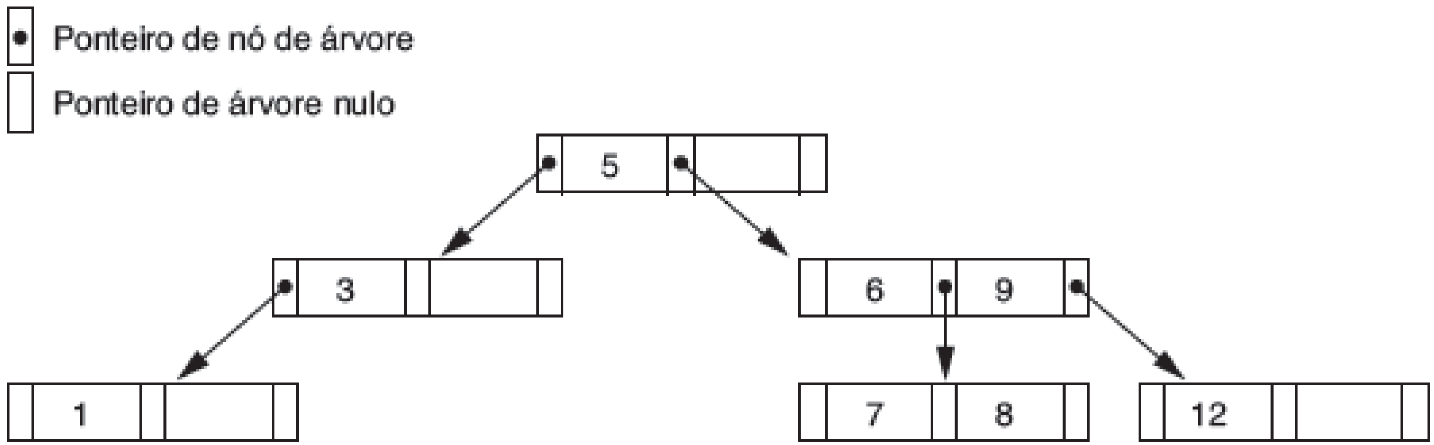
$K_{i-1} < X < K_i$ para $1 < i < q$;

$X < K_i$ para $i = 1$; e

$K_{i-1} < X$ para $i = q$.

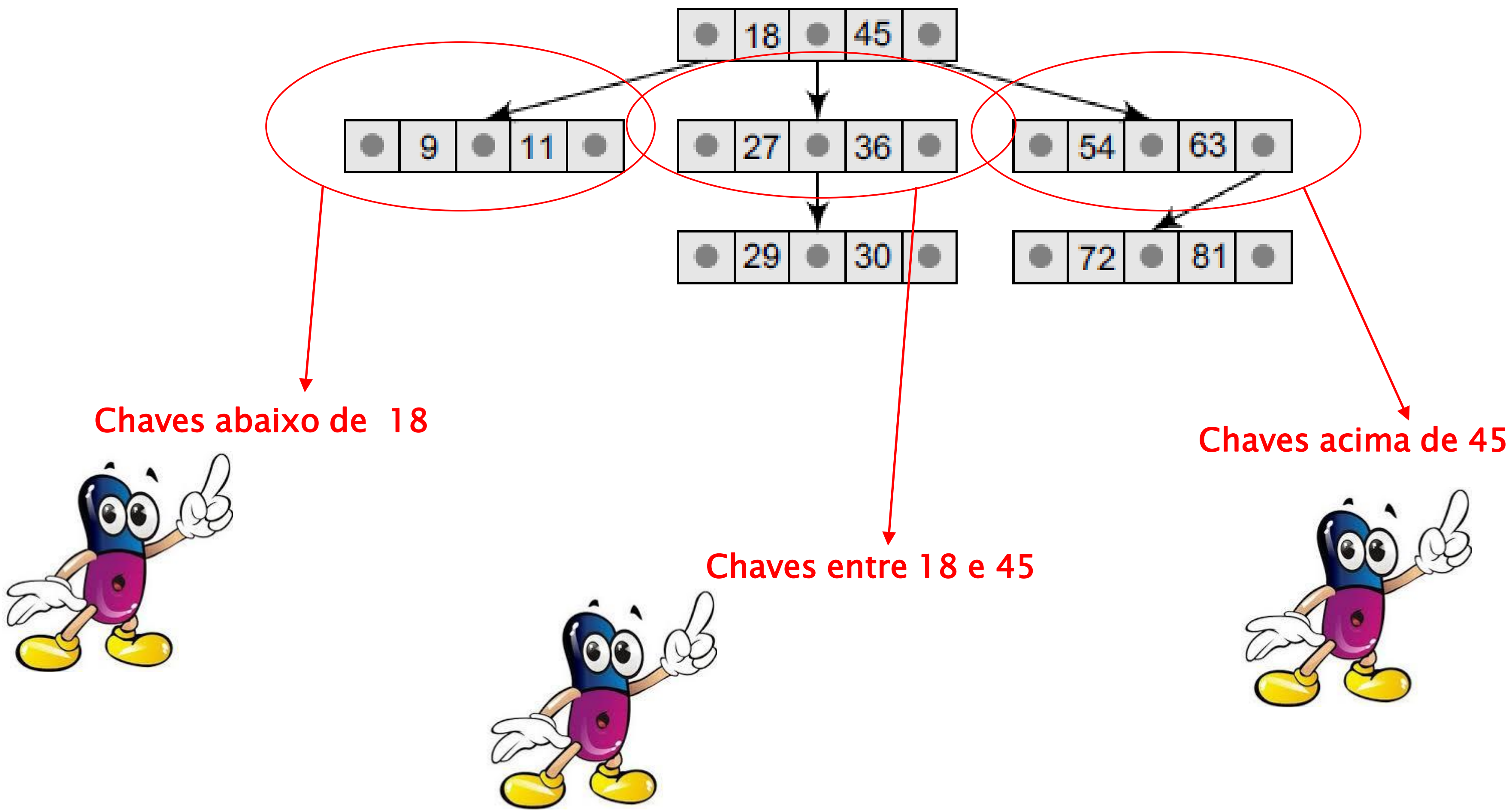
Obs. Sempre que se procura uma um valor X, deve-se seguir o ponteiro P_i apropriado, de acordo com as fórmulas acima.

Árvores n-ária de Pesquisa - Exemplo



Uma árvore de pesquisa de ordem $p = 3$.

Árvores n-ária de Pesquisa - Exemplo



Árvores n-ária de Pesquisa

- Uma **árvore n-ária** de pesquisa pode ser usada como um mecanismo para procurar um registro armazenado em disco.

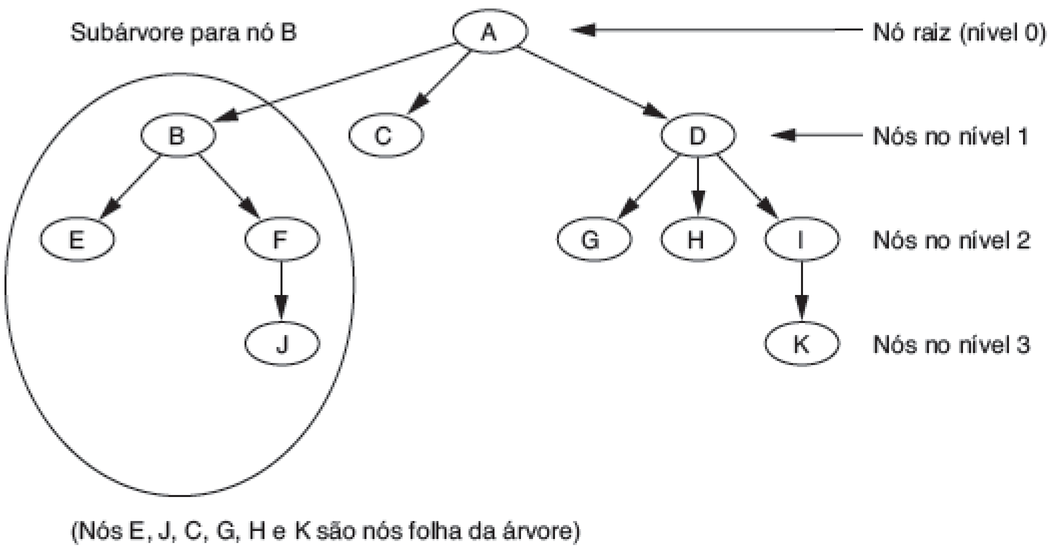
- Cada valor de chave na árvore é **associado** a um ponteiro para o registro no arquivo de dados que tem esse valor.

- Quando um registro é inserido no arquivo, deve-se **atualizar** a árvore de pesquisa com os correspondentes ponteiros para o registro.

- Em geral, os algoritmos de inserção e deleção de registros não garantem que a árvore de pesquisa seja **balanceada**.

Árvores n-ária de Pesquisa Balanceada

- Uma **árvore n-ária** de pesquisa é balanceada quando os nós folha estão no mesmo nível;



A árvore acima não é balanceada, pois há nós folha nos níveis 1, 2 e 3.

Importância do Balanceamento da árvore

- Garantir que os nós sejam igualmente distribuídos, de modo a **minimizar** a profundidade;
- Tornar a velocidade de pesquisa **uniforme**, de modo que o tempo médio para a busca de qualquer chave aleatória seja aproximadamente o mesmo.

Problemas com a Árvore n-ária de pesquisa

- **Inclusões** e **exclusões** de registros podem causar muita reestruturação da árvore;
- Em caso de muitas exclusões, nós podem ficar quase vazios, desperdiçando espaço de armazenamento e aumentando o número de níveis.
- Árvores **B-tree** podem resolver esses problemas.

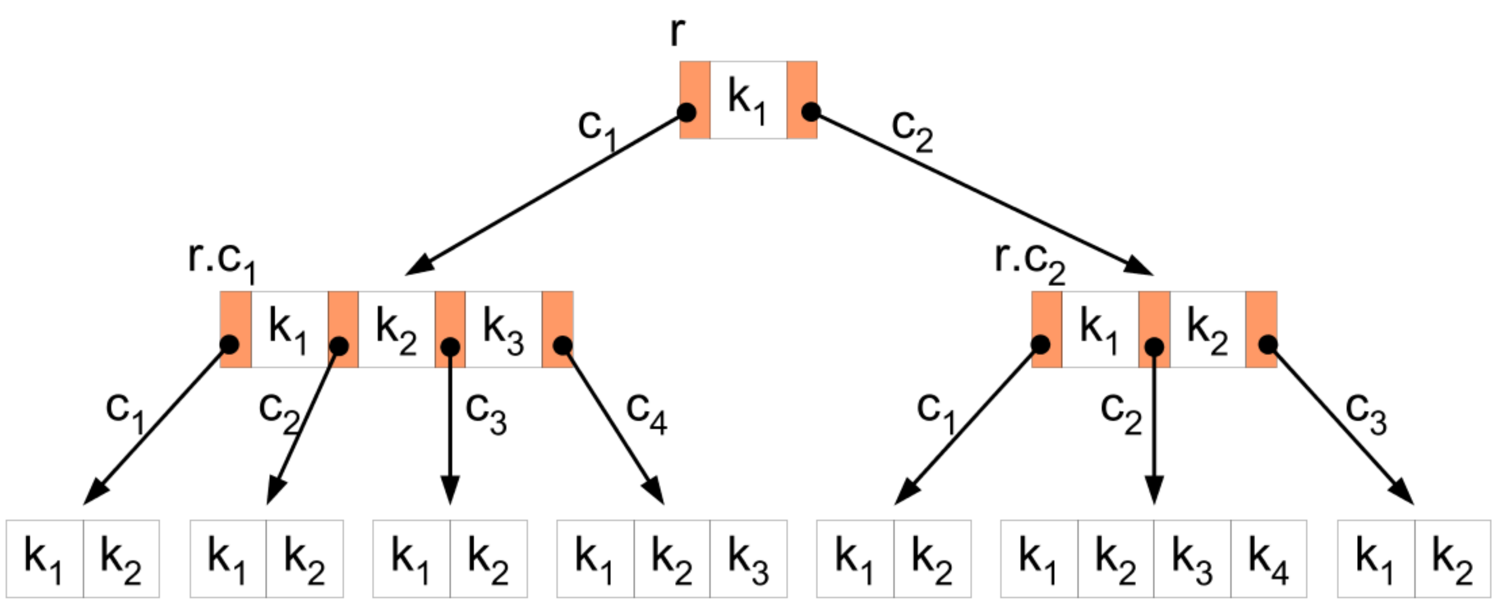
Árvore B-tree

- Estrutura de dados projetada para memória secundária;
- Permite a inserção, remoção e busca de chaves com complexidade logarítmica;
- **Por essa razão, é largamente empregada em sistemas de bancos de dados**;
- Inventada por Rudolf Bayer e Edward Meyers McCreight em 1971;
- Especula-se que o B venha da palavra balanceada;
- **Correspondem à uma generalização das árvores de busca binária**, pois cada nó de uma árvore binária armazena uma única chave, enquanto que B-trees armazenam um maior número de chaves em cada nó.

Árvore B-tree – Visão Geral

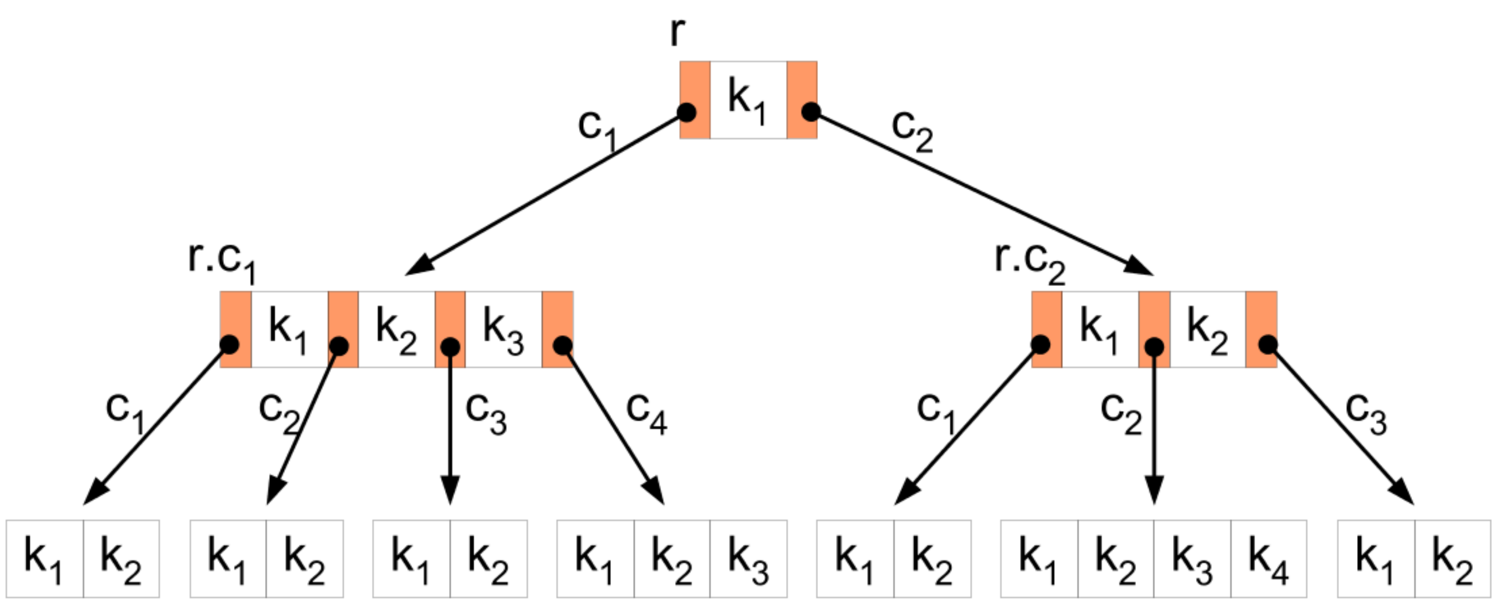
- São organizadas por nós com um conjunto de chaves;

- As chaves em um nó são mantidas em ordem crescente, de modo que para cada chave há dois endereços para nós filhos, sendo que, o **endereço à esquerda é para um nó com chaves menores** e o **da direita para com um conjunto de chaves maiores**.

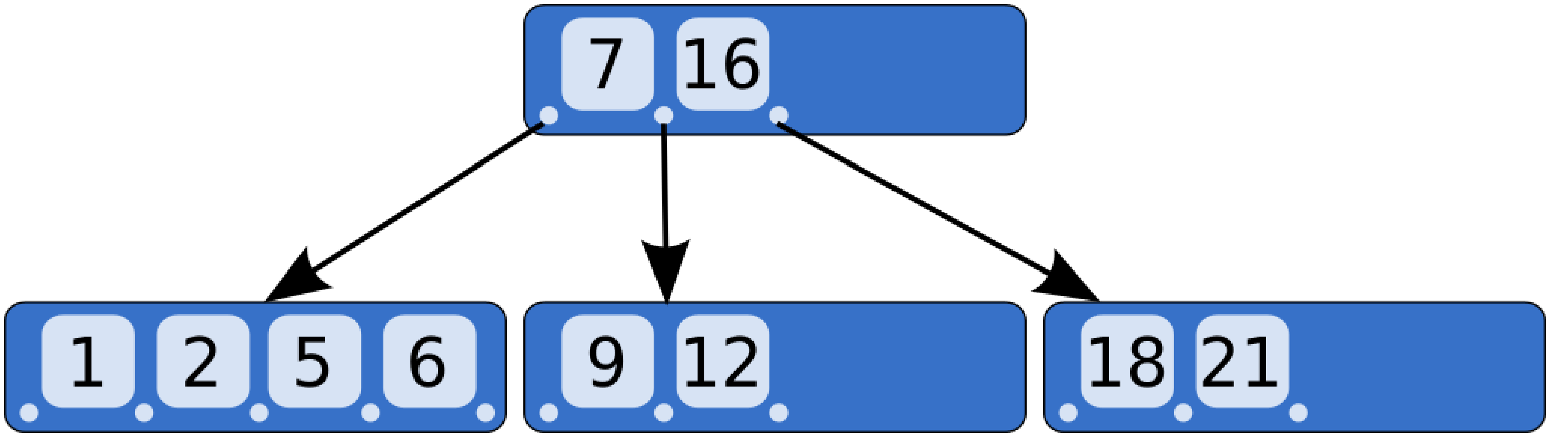


Árvore B-tree – Visão Geral

- São organizadas por nós com um conjunto de chaves;
- As chaves em um nó são mantidas em ordem crescente, de modo que para cada chave há dois endereços para nós filhos, sendo que, o **endereço à esquerda é para um nó com chaves menores** e o **da direita para com um conjunto de chaves maiores**.



Árvore B-tree – Visão Geral

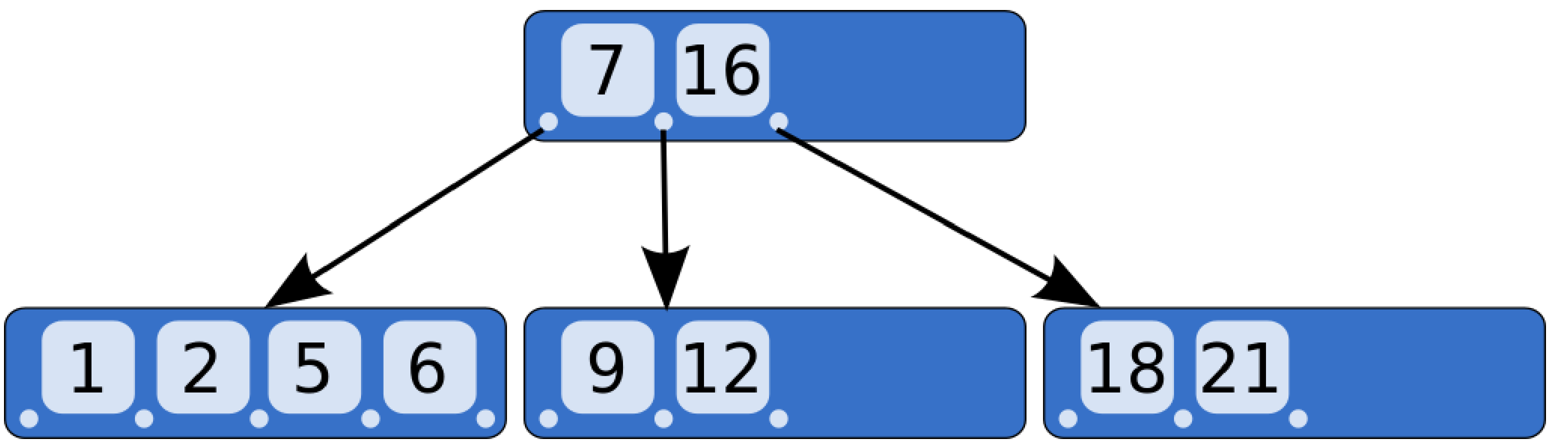


Árvore B-tree – Definição

- Árvore B-tree é uma árvore n-ária de pesquisa com restrições adicionais que garantem que a árvore sempre esteja balanceada e que o espaço desperdiçado pela exclusão de chaves, se houver, nunca se torne excessivo;
- Em um nó de uma árvore B-tree de ordem p, **cada nó tem no máximo p ponteiros de árvore**;
- Cada nó, exceto os nós raiz e folha, tem pelo menos **p/2** ponteiros de árvore (limite superior);
- O nó raiz tem pelo menos dois ponteiros de árvore, a menos que seja o único nó da árvore;
- Um nó com **p** ponteiros de árvore, tem **p-1** valores de campo de chave de pesquisa;
- Todos os nós folha estão no mesmo nível;
- Os nós folha têm a mesma estrutura dos nós internos, exceto que todos os seus ponteiros de árvore são **null**.

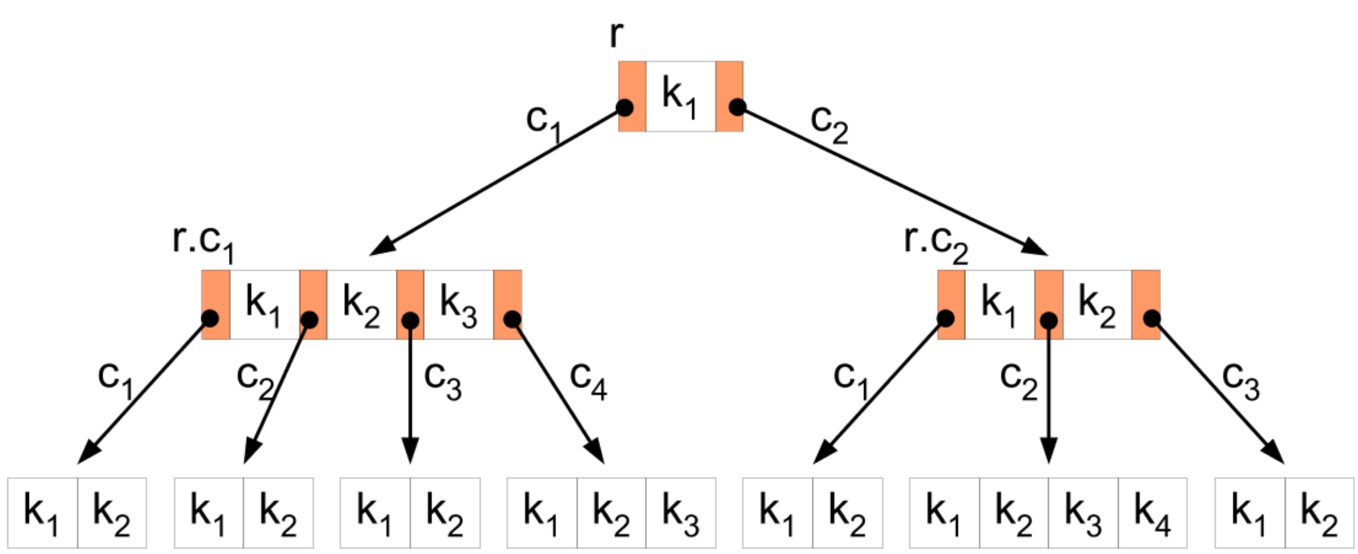
Árvore B-tree – Nó raiz

- O nó raiz de uma B-tree possu o limite de p-1 chaves armazenadas, mas não apresenta um número mínimo de chaves;
- Ou seja, um nó raiz pode ter um número inferior a p/2 -1 chaves .
- Na figura abaixo, o nó raiz possui as chaves 7 e 16



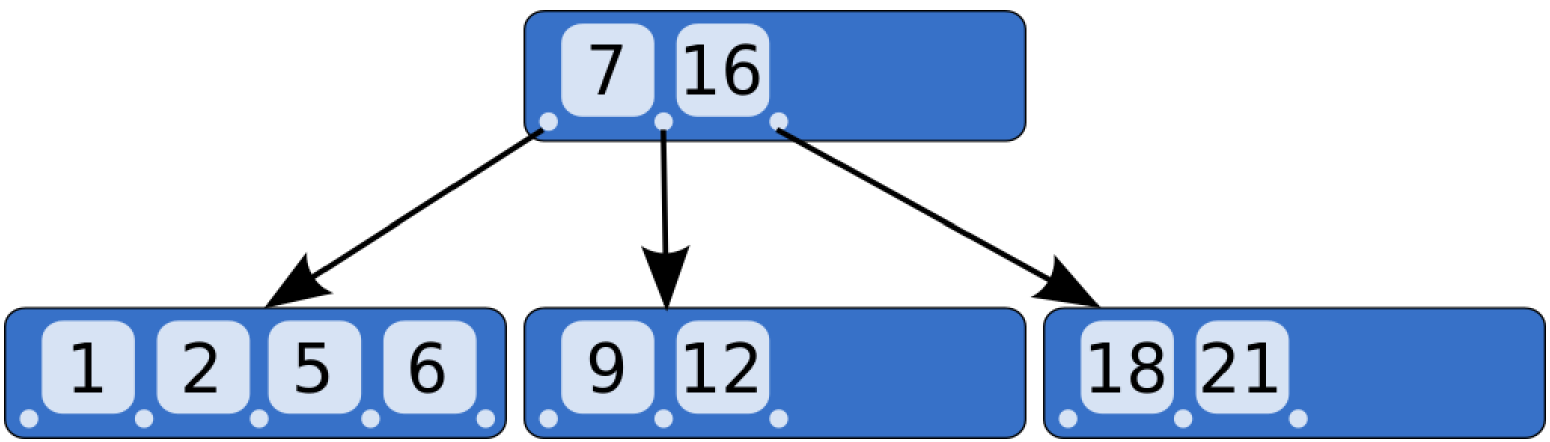
Árvore B-tree – Nó interno

- Nós internos são nós que não são folhas nem raiz;
- Devem ter o número mínimo de p/2 -1 chaves;
- Devem ter o número máximo de p -1 chaves.



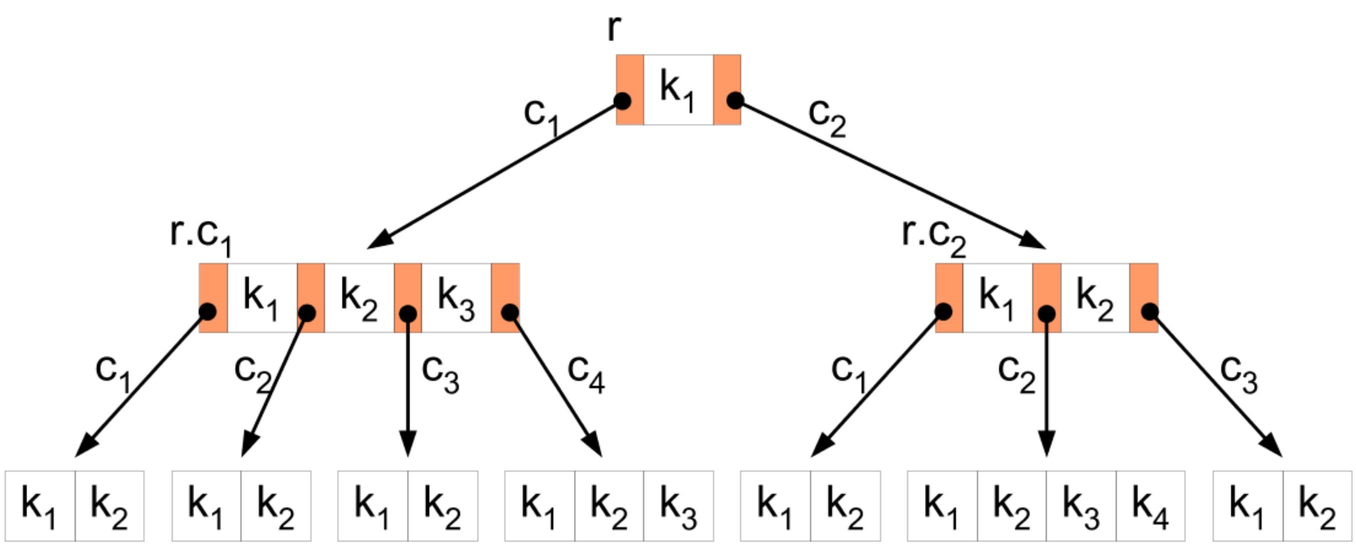
Árvore B-tree – Nó folha

- São nós que possuem a mesma restrição de máximo e mínimo de chaves dos nós internos, mas estes não possuem apontadores para nós filhos.
- Na figura abaixo, são todos os nós exceto o raiz.



Árvore B-tree – Operações de busca

- A busca de uma chave K em uma B-tree é muito parecida com uma busca em árvore binária;
- Porém, agora à cada nó carregado na memória há vários ponteiros para o nó seguinte da busca e não apenas dois;



Árvore B-tree – Operações de inserção

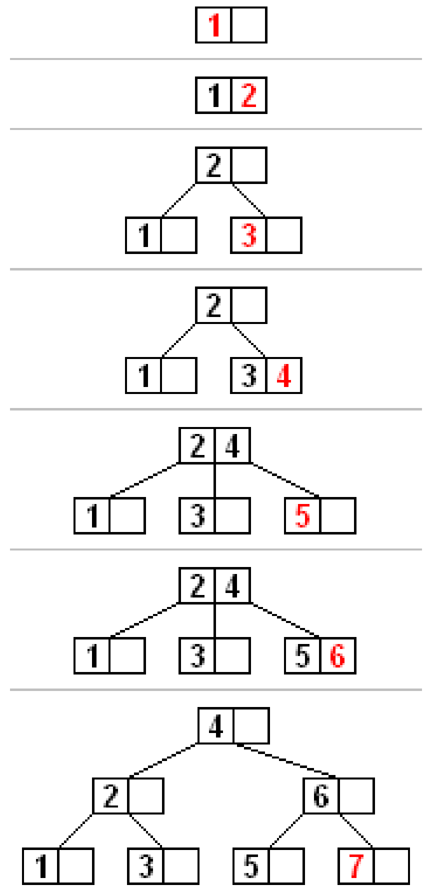
- Uma B-tree começa com um único nó raiz (que também é um nó folha) no nível 0 (zero);

- Criado o nó raiz, a inserção das próximas chaves seguem o mesmo procedimento: busca-se a posição correta da chave em um nó folha e insere-se a chave, garantindo a ordenação destas.

- Podem ocorrer duas situações:

  1. Nó folha está com número de chaves menor que o permitido (p-1). Nesse caso, apenas insere-se a chave de forma ordenado no nó.
  2. Nó já tem o número máximo de chaves permitido (p-1). Nesse caso, ocorre um SPLIT (Divisão do nó) para manter-se o balanceamento da árvore.
  3. Primeiramente, escolhe-se um valor intermediário na sequência ordenada de chaves, incluindo-se a nova chave que ordenada com as chaves do nó estariam no meio da sequência.
  4. Cria-se um novo nó e os valores maiores do que a chave intermediária são armazenados no novo nó e os menores continuam no nó anterior (SPLIT).
  5. A chave intermediária escolhida deverá ser inserida no nó pai, o qual também poderá sofrer overflow. Esta série de overflows pode se propagar para toda a B-tree, o que garante o seu balanceamento na inserção de chaves.

Árvore B-tree – Operações de inserção



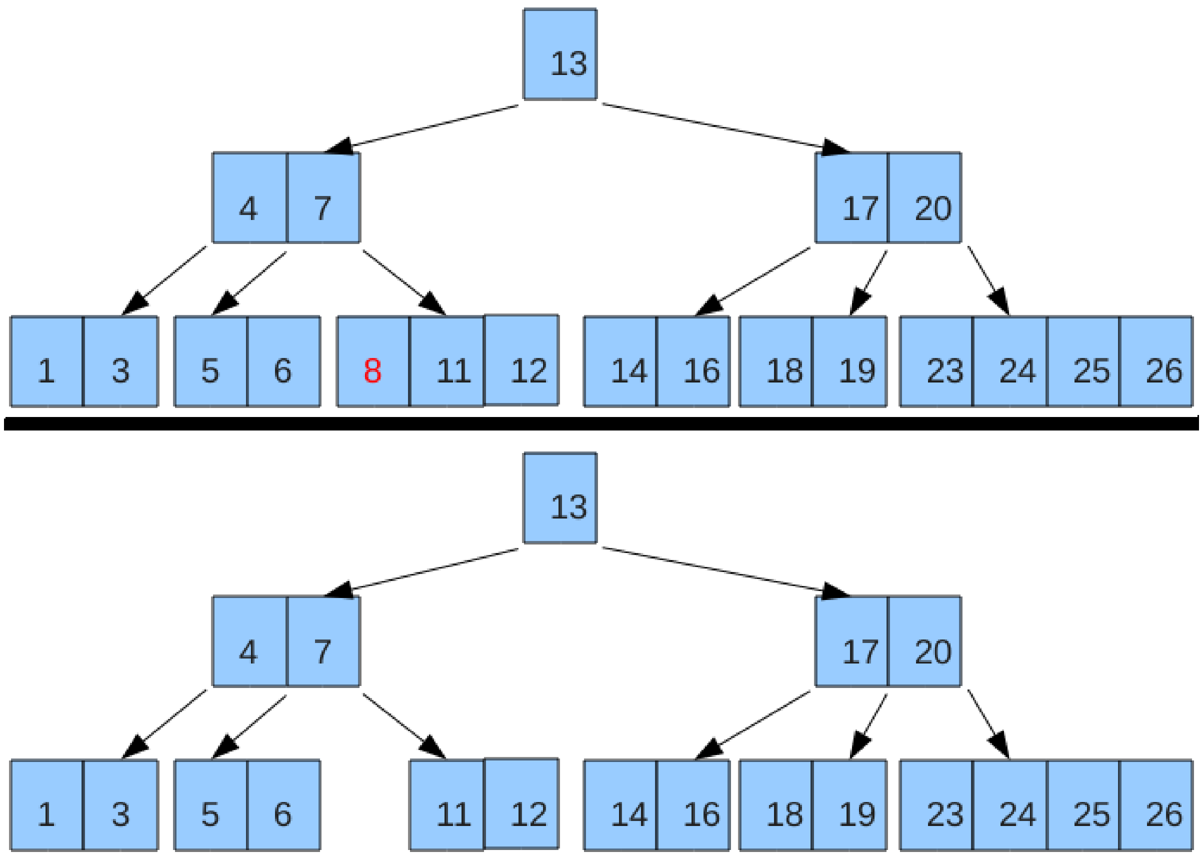
Inserção em B-tree da sequência de 1 a 7. Os nós dessa árvore possuem no máximo três ponteiros.

Árvore B-tree – Operações de Remoção

- O algoritmo de remoção deve garantir que todas as propriedades da árvore sejam mantidas, uma vez que uma chave pode ser eliminada de qualquer nó e não apenas das folhas;
- Nessa operação podem ocorrem underflows, ou seja, quando há um número abaixo do mínimo permitido (p/2 -1) chaves em um nó.
- Na remoção de chaves há vários casos a se analisar.

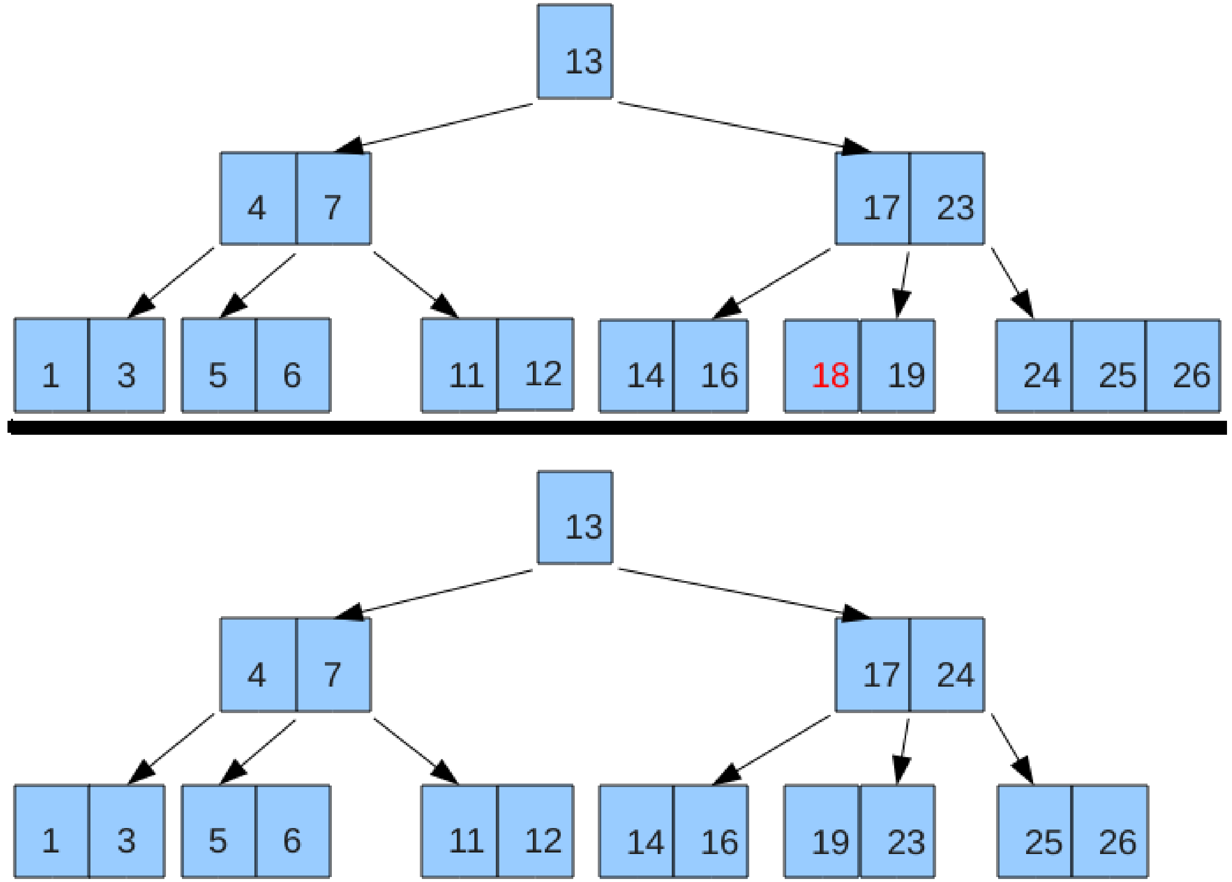
Árvore B-tree – Remoção – Exemplo 1

- A remoção da chave **8** NÃO causa o underflow do nó folha em que ela está, portanto ela é simplesmente deletada e as outras chaves são reorganizadas mantendo a sua ordenação.



Árvore B-tree – Remoção – Exemplo 2

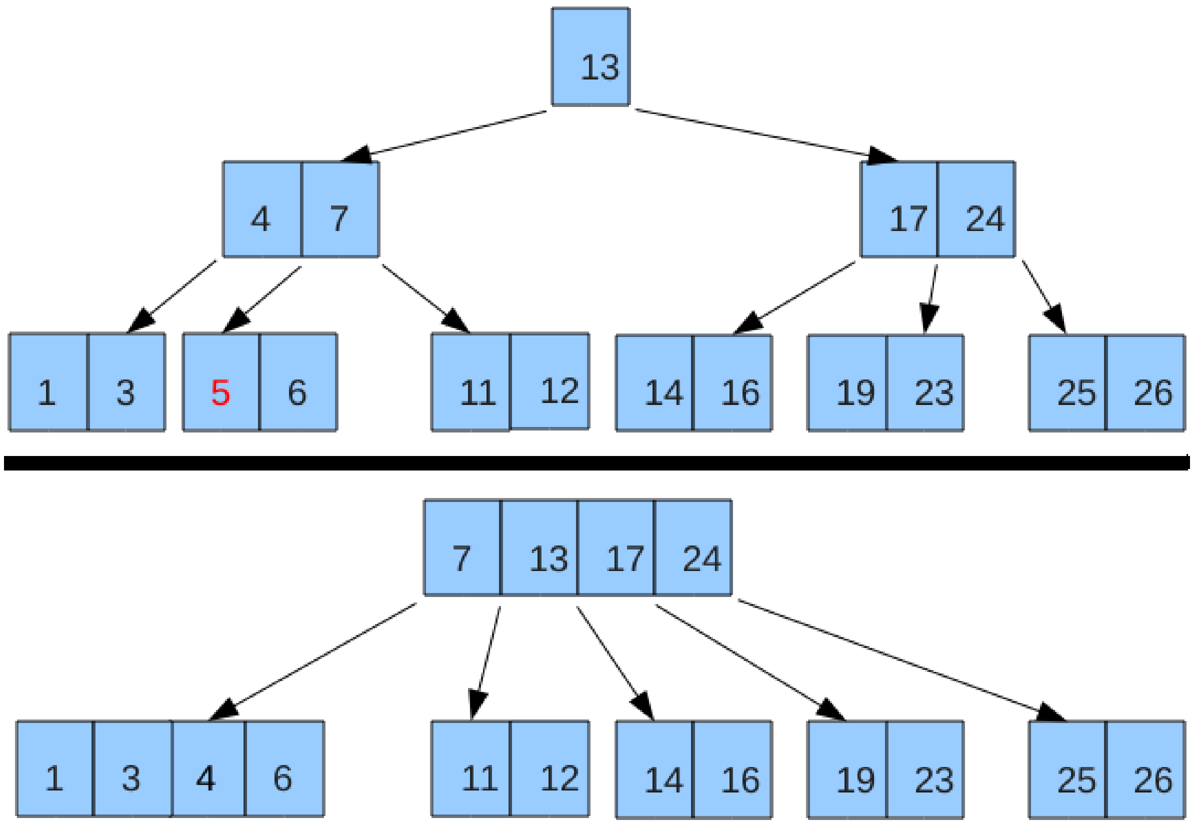
- Na remoção da chave **18** , o nó que contém essa chave possui um nó irmão à direita com um número superior ao mínimo de chaves (nó com chaves 24, 25 e 26) e, portanto, estas podem ser redistribuídas entre os nós, de modo que nenhum deles tenha um número inferior ao mínimo permitido.



QualitSys

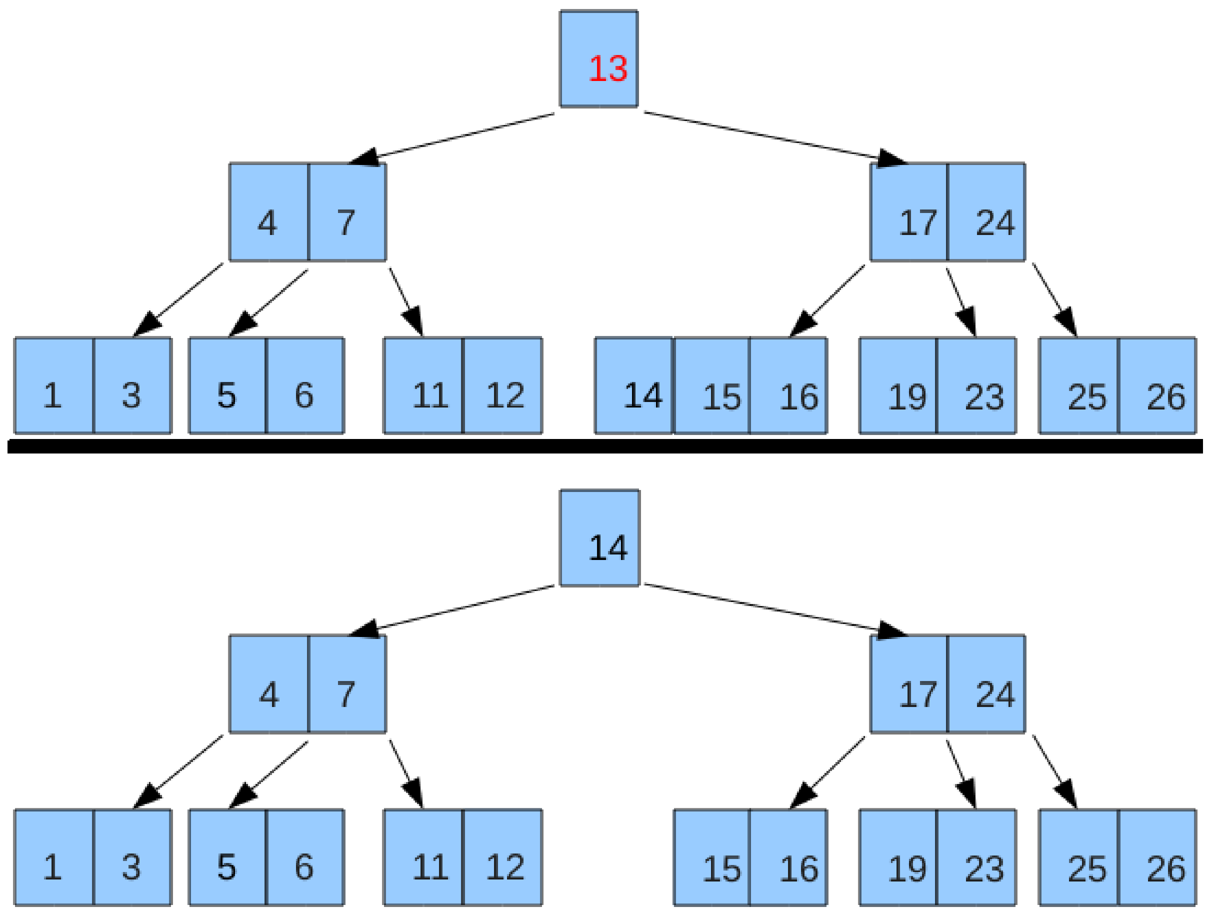
Árvore B-tree – Remoção – Exemplo 3

- Na remoção da chave **5**, como não foi possível utilizar a técnica de redistribuição, pois os nós irmãos possuem o número mínimo de chaves, então foi necessário concatenar o conteúdo do nó que continha a chave 5 com seu nó irmão à esquerda e a chave separadora pai (underflow) e é necessário diminuir a altura da árvore de maneira que o conteúdo do nó pai e seu irmão, juntamente com a raiz, sejam concatenados para formar um único nó.



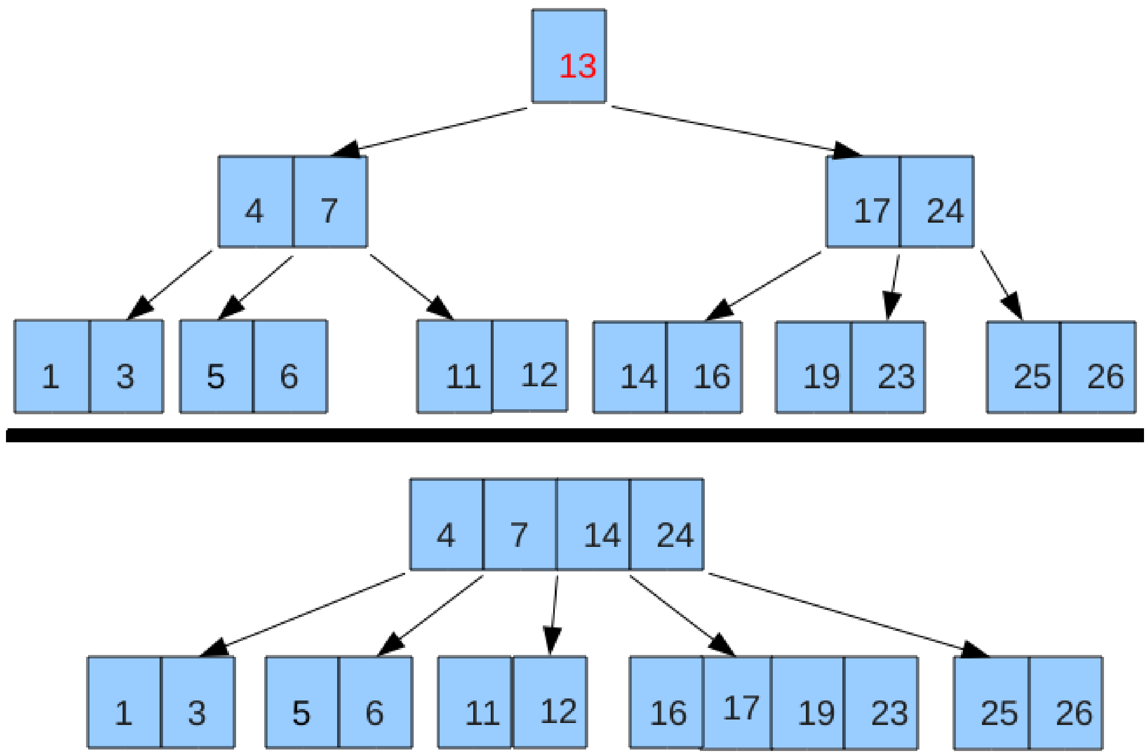
Árvore B-tree – Remoção – Exemplo 4

- A remoção da chave **13**, foi realizada com a substituição do 13 pelo menor número da subárvore à direita de 13 que era o 14. Essa troca não causou underflow do nó em que estava o 14 e, portanto, não gerou grandes alterações na árvore.



Árvore B-tree – Remoção – Exemplo 5

- A remoção da chave **13**, foi realizada de forma semelhante ao exemplo anterior. Mas, nesse caso, ocorre underflow do nó que contém a menor chave da subárvore à direita de 13. Com isso, como não é possível a redistribuição, concatena-se o conteúdo desse nó com seu irmão à direita, o que gera também underflow do nó pai. O underflow do nó pai também é resolvido com a concatenação com seu irmão e a raiz, resultando uma diminuição da altura da árvore.



Árvore B^+-tree

- A maioria das implementações de um índice multinível dinâmico utiliza uma variação da estrutura de dados B-tree chamada B^+-tree.
- Em uma B-tree, cada valor de campo de pesquisa aparece uma vez em algum nível da árvore, junto com um ponteiro de dados.
- Em uma B^+-tree, os ponteiros de dados são armazenados apenas nos nós folha da árvore;
- Logo, em uma B^+-tree, a estrutura dos nós folha difere da estrutura dos nós internos.
- Em uma B^+-tree, os nós folha têm uma entrada para cada valor do campo de pesquisa, junto com um ponteiro para o bloco que contém esse registro.