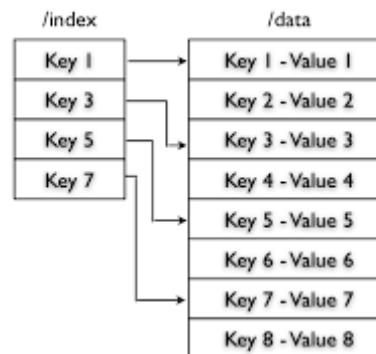




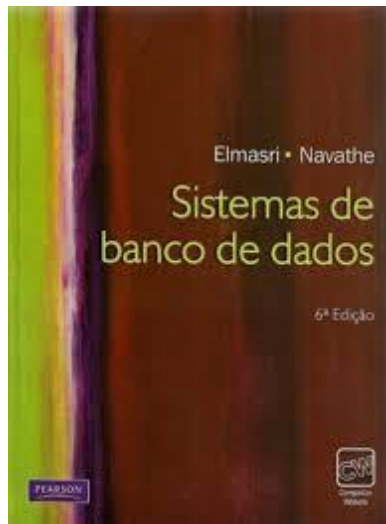
## Unidade 10 – Estruturas de Indexação em Banco de Dados Parte 1



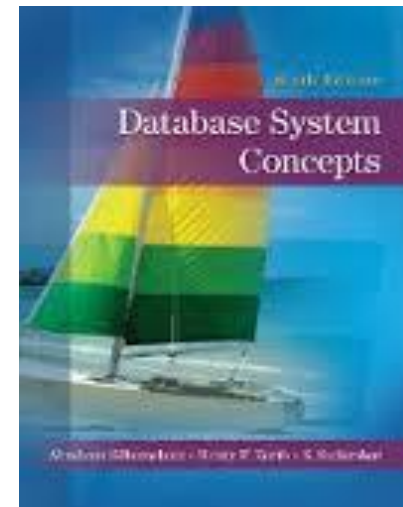
Prof. Aparecido V. de Freitas  
Doutor em Engenharia  
da Computação pela EPU SP



# Bibliografia



Sistemas de Banco de Dados  
Elmasri / Navathe 6ª edição



Sistema de Banco de Dados  
Korth, Silberschatz – Sixth Edition



# 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.





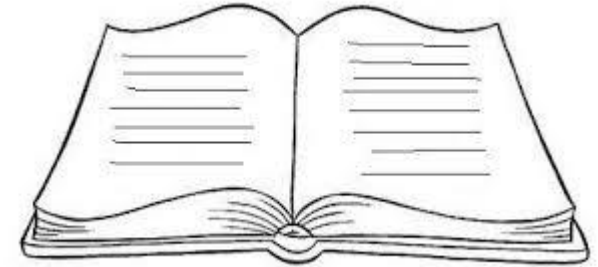
# Índices

- ✦ 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<sup>+</sup>-trees );
- ✦ Índices também podem ser construídos com base em **Hashing**.





# Índices ordenados de único nível

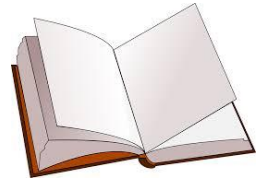


- ✦ Procedimento semelhante ao índice usado em um livro;
- ✦ Por meio da pesquisa no índice de um livro, obtém-se o endereço do termo desejado (número da página) e em seguida procura-se o termo na página citada.
- ✦ O índice costuma armazenar cada valor do campo de índice junto com uma lista de ponteiros 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.





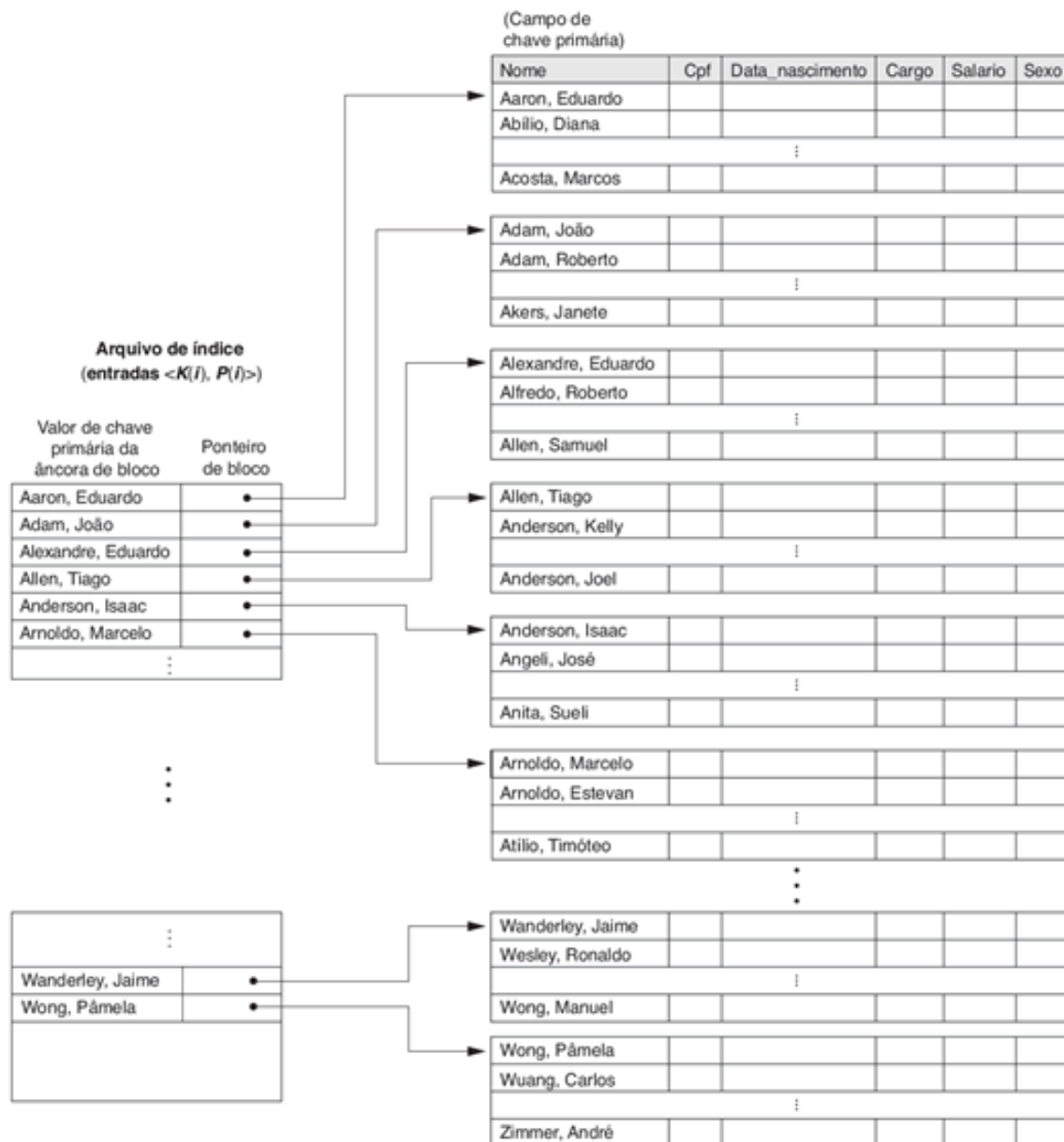
# Índices Primários



- ⊕ 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 esperso** (não denso), pois inclui uma entrada para cada bloco de disco do arquivo.



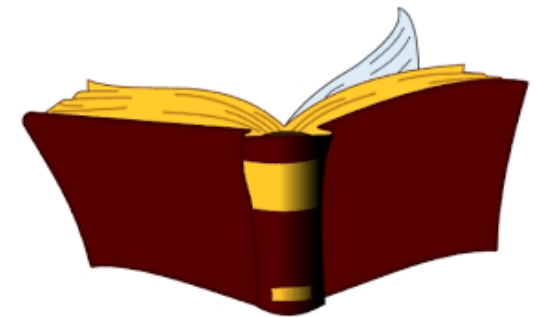
# Índices Primários





# 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  $bfr = \lfloor B/R \rfloor = 1024/100 = 10$  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.





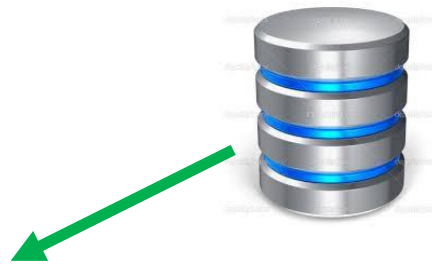
# Exemplo 1 – Arquivo ordenado com índice primário

- ⊕ 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  $R_i = (9+6) = 15$  bytes, de modo que o fator de bloco para o arquivo de índice é:  $bfri = \lfloor B/R_i \rfloor = 1024 / 15 = 68$  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.



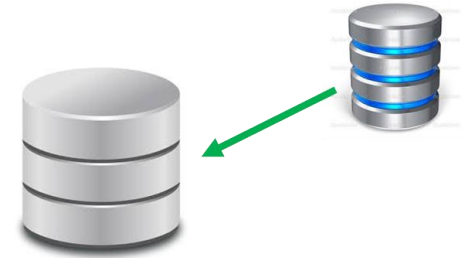
# Manutenção de dados com Índices Primários

- ✦ 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

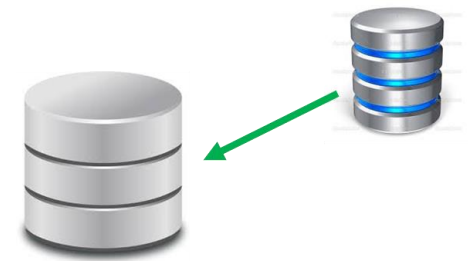


- ⊕ O índice secundário pode ser criado em um campo que é uma **chave candidata** e tem um valor **único** em cada registro, ou em um campo **não chave** com valores **duplicados**.
- ⊕ O arquivo de dados pode ser um arquivo **ordenado**, **desordenado** ou **hashed**.
- ⊕ O índice é novamente um arquivo 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

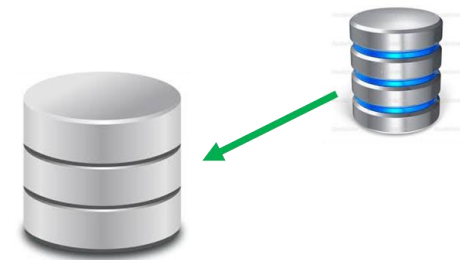
- ⊕ O índice é construído com base em um campo de chave (único) que tem um valor distinto para cada registro.
- ⊕ Tal campo é às vezes chamado de Chave Secundária.
- ⊕ No modelo relacional, isso corresponderia a qualquer atributo de chave **UNIQUE** ou ao atributo de chave primária da tabela.
- ⊕ Nesse 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.

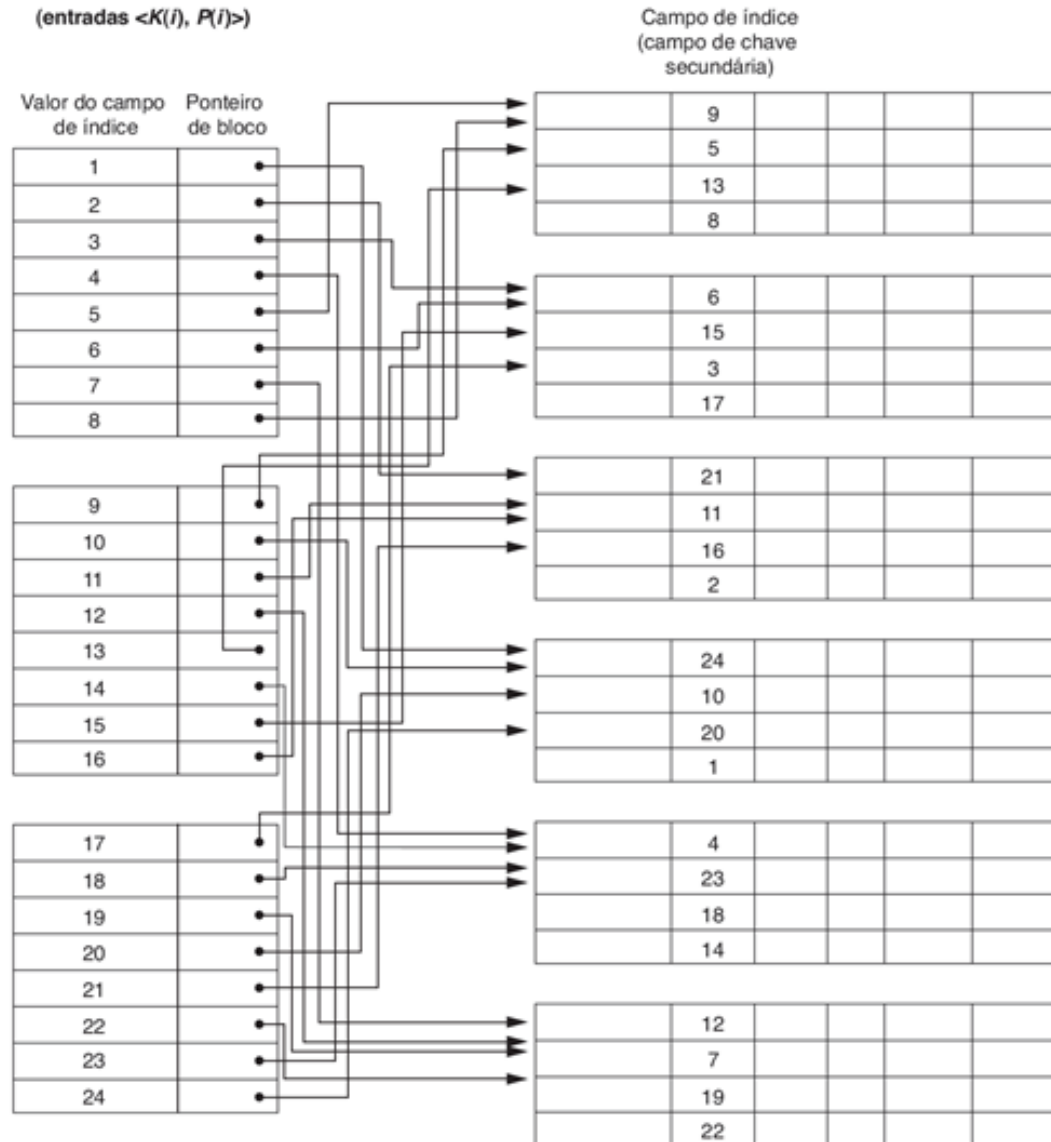




# Índices Secundários com chaves candidatas

**Arquivo de índice**  
(entradas  $\langle K(i), P(i) \rangle$ )

**Arquivo de dados**





## 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  $R_i = (9 + 6) = 15$  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,  $bi = ri / bfri = 30.000/68 = 442$  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.

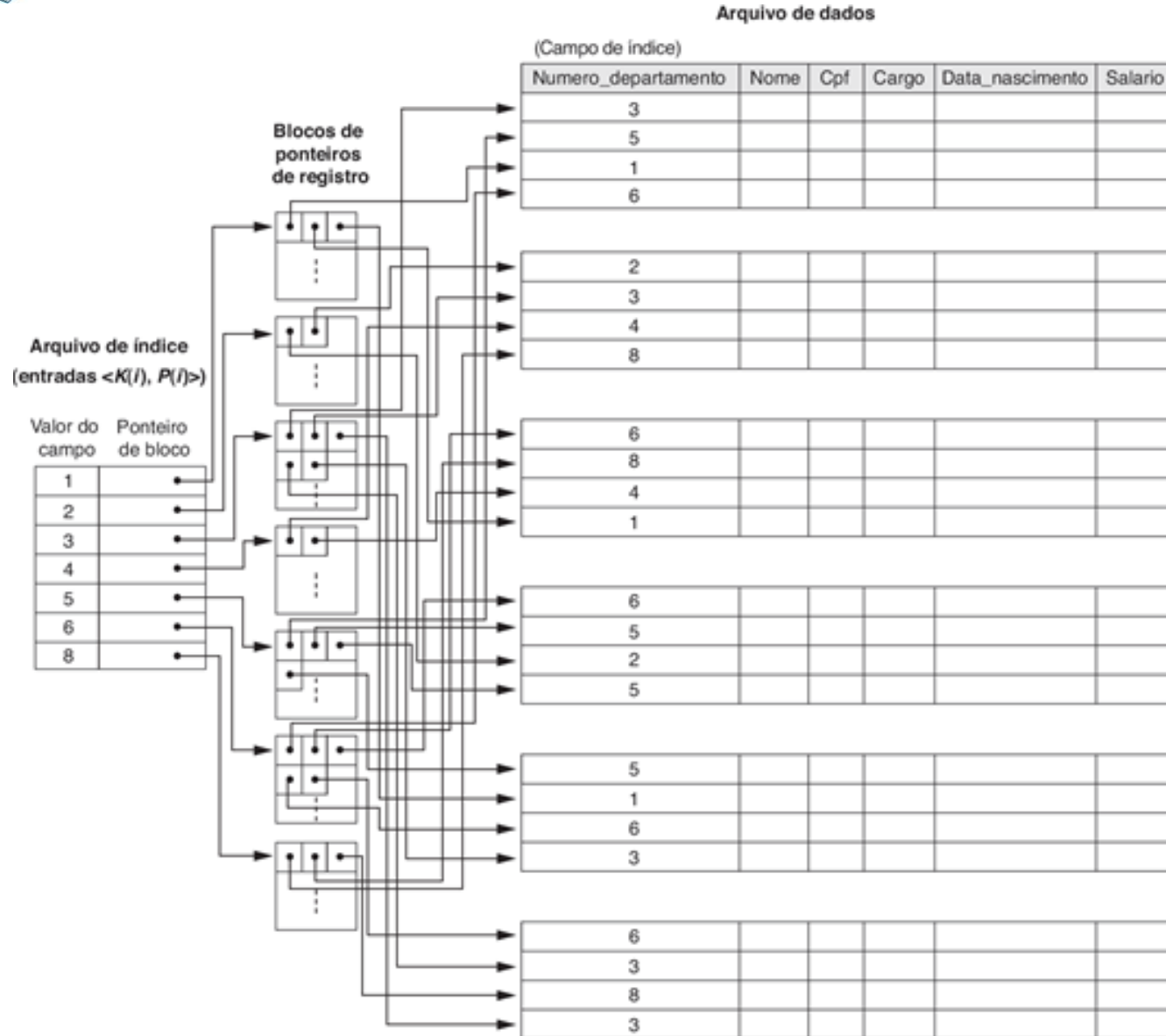


# Índices Secundários em campos não chave

- ⊕ 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





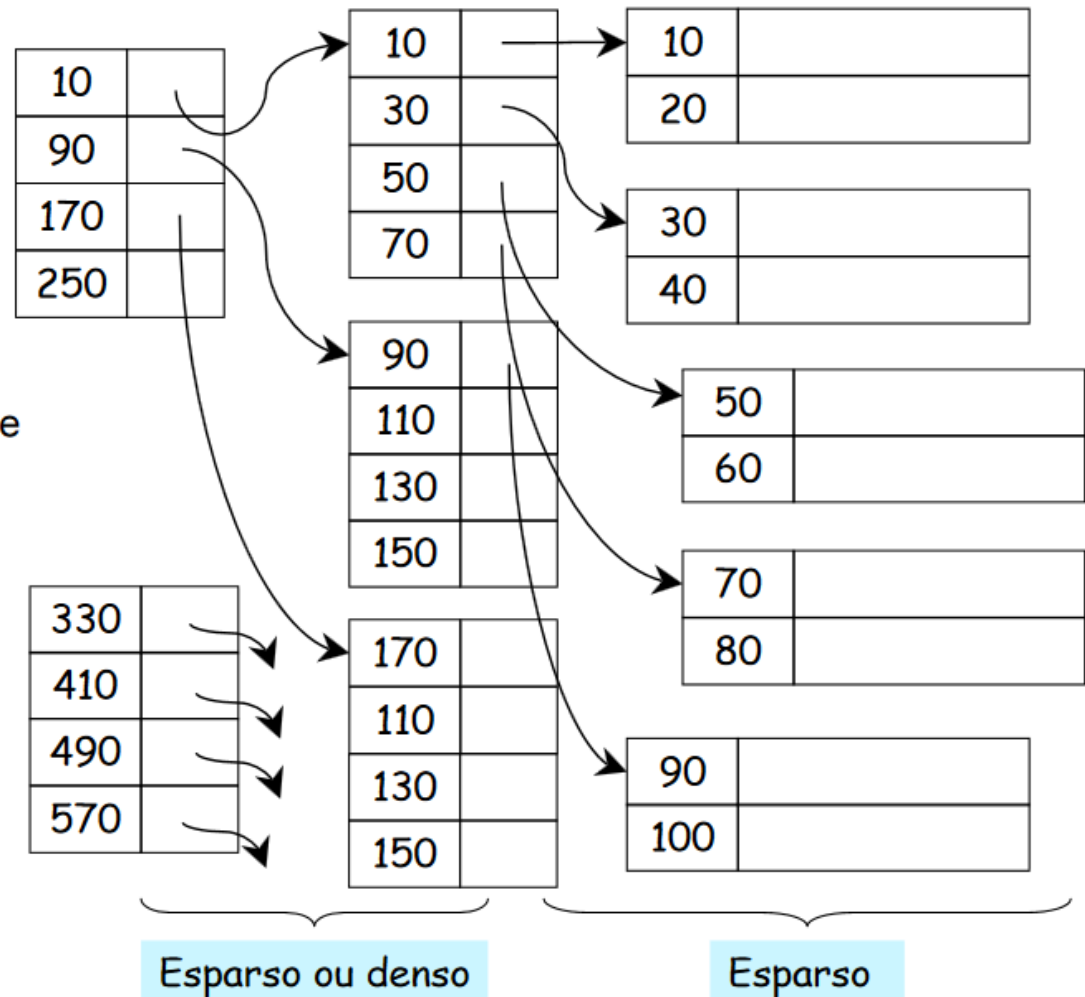
# Índices Multiníveis





# Índices Multiníveis

- **Motivação:** se o arquivo de índices se torna muito grande para ser armazenado em bloco de disco, é interessante indexá-lo em mais de um nível
- **Vantagem:** índice pequeno pode ser mantido em memória e o tempo de busca é mais baixo
- **Desvantagem:** muitos níveis de índices podem aumentar a complexidade do sistema (talvez seja melhor usar a árvore-B)





# Índices Multiníveis

- Um índice multinível é um “Índice de índice”.
- Primeiro nível: arquivo ordenado pela chave de indexação, valores distintos, entradas de tamanho fixo.
  - Demais níveis: índice primário sobre o índice do nível anterior e assim sucessivamente até que no último nível o índice ocupe apenas um bloco.
  - Número de acessos a bloco: um a cada nível de índice, mais um ao bloco do arquivo de dados.



# Índices Multiníveis

- ⊕ Com índices ordenados de um nível, operações de searching podem ser aplicadas por meio de uma pesquisa binária que requer aproximadamente  $\log_2 b_i$  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 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**).

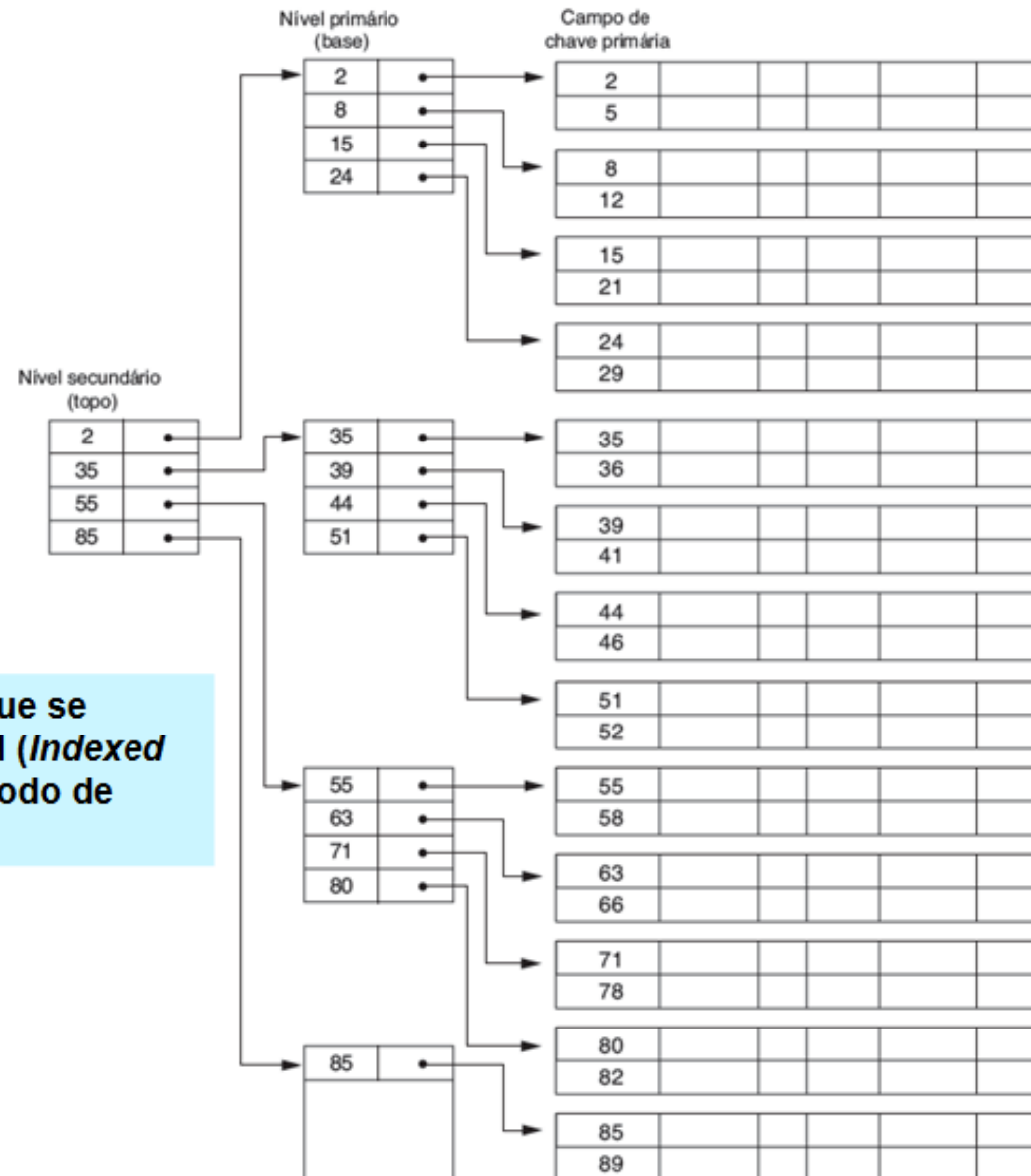




# Índices Multiníveis

Índice de dois níveis

Arquivo de dados



Índice primário de dois níveis que se parece com a organização ISAM (*Indexed Sequential Access Method* – Método de Acesso Sequencial Indexado).



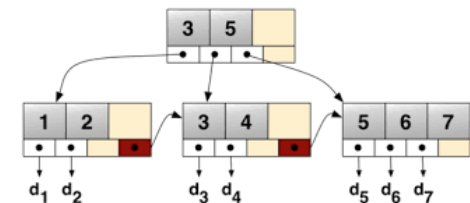
## Exemplo 3 – Índice Multinível

- ⊕ 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  $b2 = b1/fo = 442/68 = 7$  blocos;
- ⊕ O número de blocos de terceiro nível será  $b2/fo = 7/68 = 1$  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 **fisicamente** ordenados.
- ✦ 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<sup>+</sup>-trees**.
- ✦ B-trees e B<sup>+</sup>-trees são casos especiais da famosa estrutura de dados de pesquisa, conhecida por **árvore**.





# Árvores

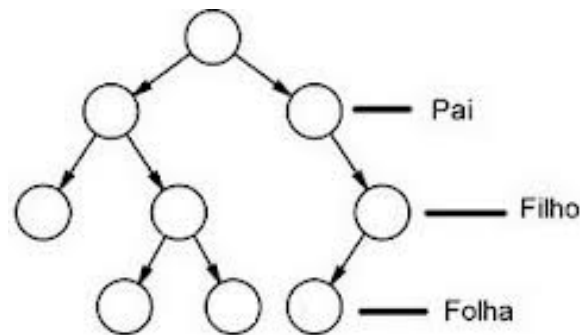
- Árvore é uma estrutura de dados não-linear.
- Tem uma importância muito grande na Computação, pois disponibiliza algoritmos muito mais rápidos que os encontrados nas estruturas lineares.
- Têm diversas aplicações: sistemas de arquivos, interfaces gráficas, banco de dados, etc.
- Os relacionamentos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





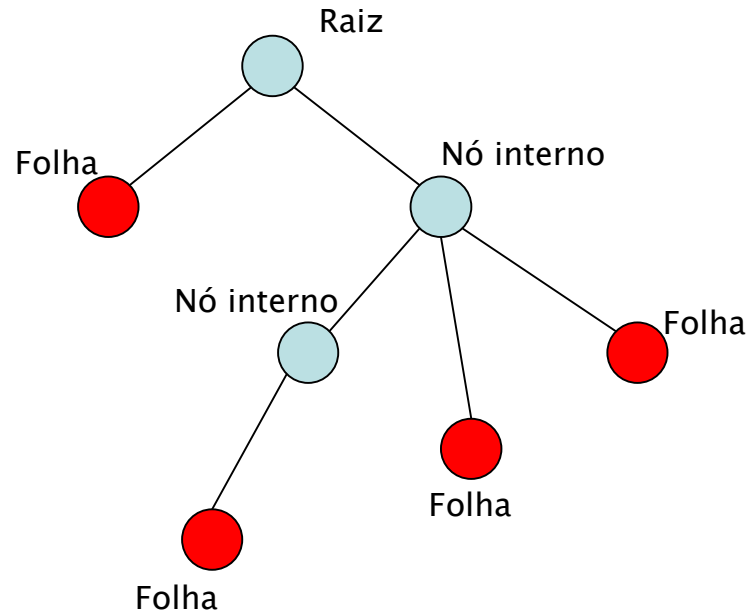
# Definição Formal

- 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

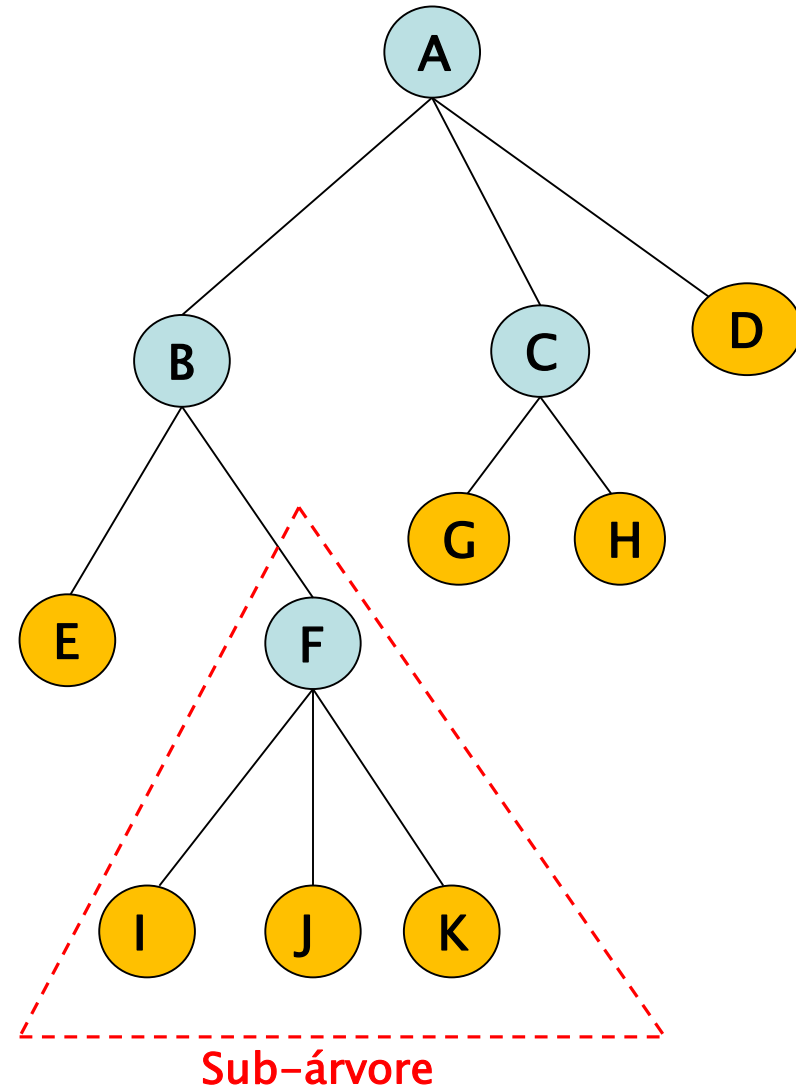
- Dois nós que são filhos do mesmo pai são **irmãos**.
- Um nó **v** é **externo** se não tem filhos.
- Nós externos também são conhecidos por **folhas**.
- Um nó **v** é **interno** se tem um ou mais filhos.





# 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.

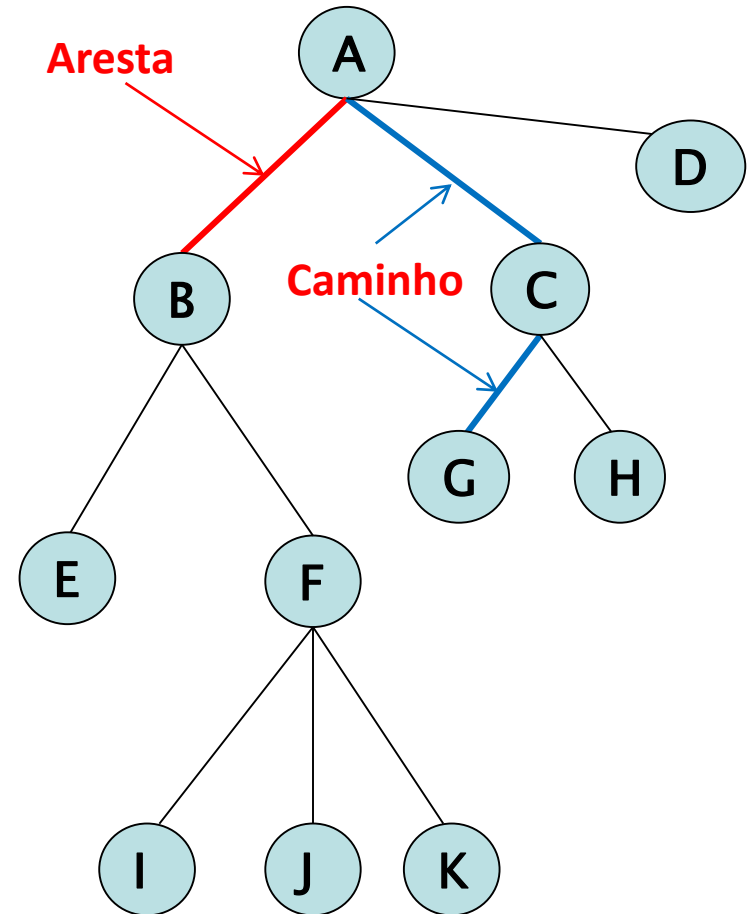






# Definições

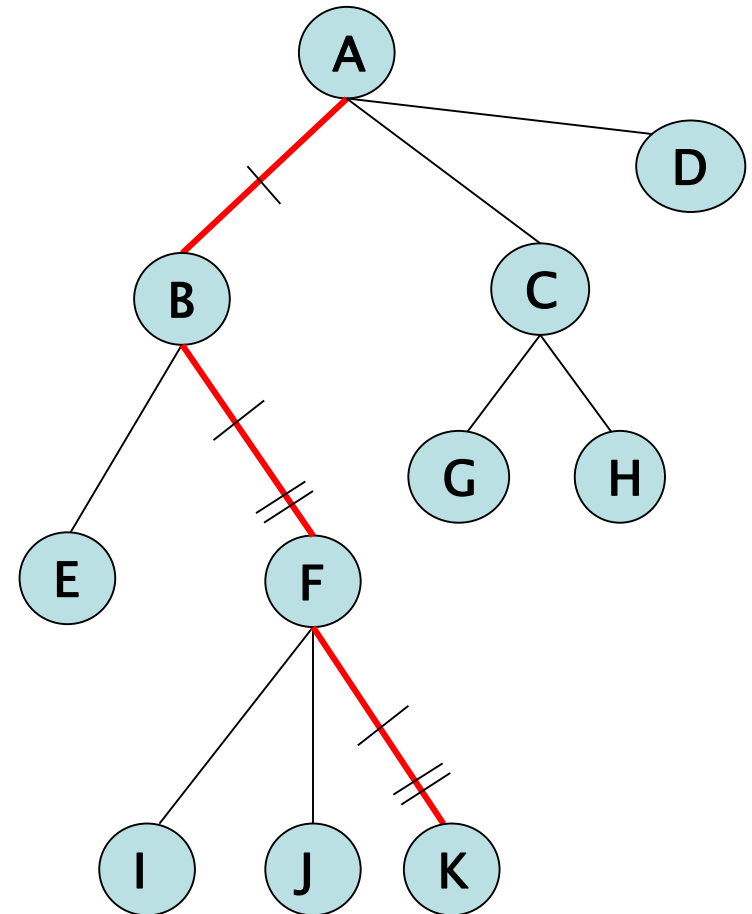
- ⊕ **Aresta:** é um par de nós  $(u,v)$  tal que  $u$  é pai de  $v$ .  $(A,B)$
- ⊕ **Caminho:** é uma sequência de nós tais que quaisquer dois nós consecutivos da sequência sejam arestas.  $((A,C),(C,G))$
- ⊕ **Tamanho de um caminho:** # de arestas em um caminho. ( Tamanho do caminho  $((A,C),(C,G))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





# Definições

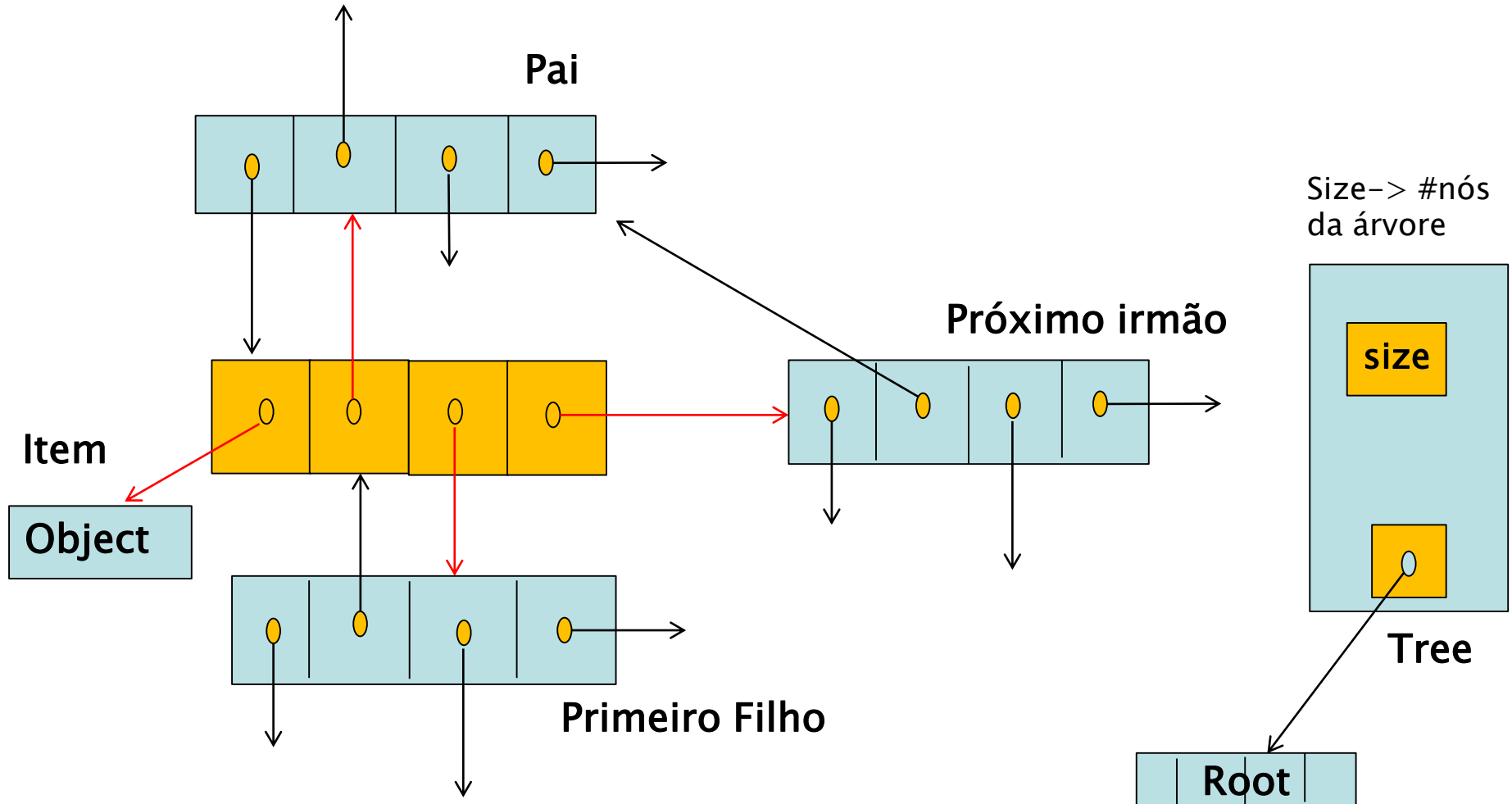
- ⊕ **Profundidade de um nó n:** é Tamanho do caminho da raiz até o nó **n**.  
(Dept (K) = 3)
- ⊕ **Profundidade da raiz: ZERO**
- ⊕ **Altura de um nó:** Tamanho do caminho de **n** até seu mais profundo descendente.  
(Altura(B) = 2) .
- ⊕ **Altura de qualquer folha:** ZERO
- ⊕ **Altura da Árvore = Altura da Raiz**





# Representando Nó de Árvores

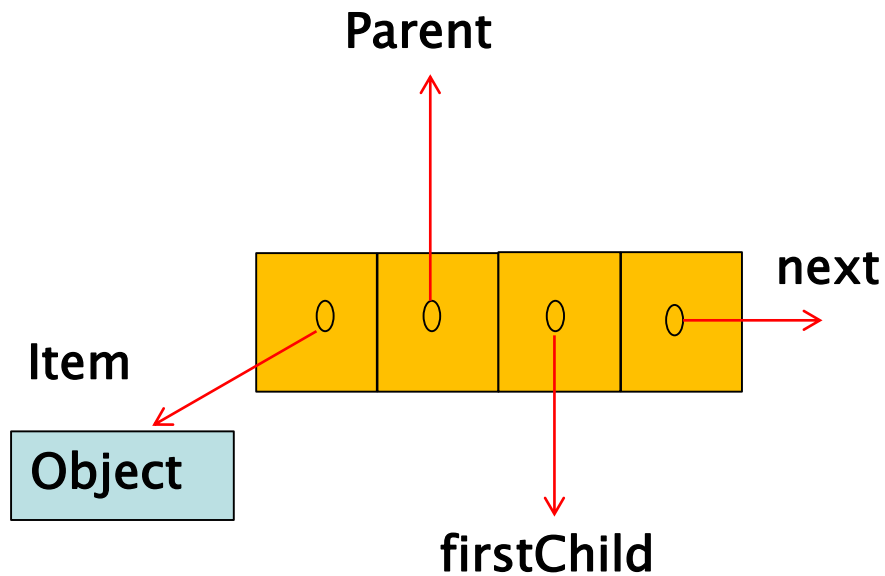
- Cada nó tem quatro referências: item, pai, primeiro filho e próximo irmão.





# 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;
```

```
    .  
    .  
    .
```

```
}
```

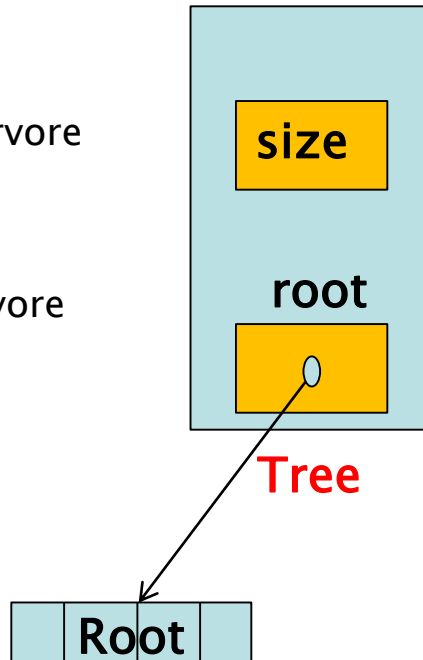


# Representando Árvores

- 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 {
```

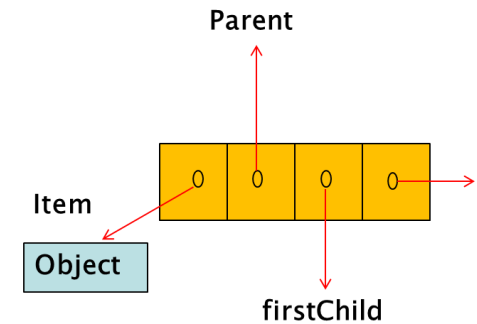
```
    Node_Tree root;  
    int size;
```

```
    .  
    .  
    .
```

```
}
```

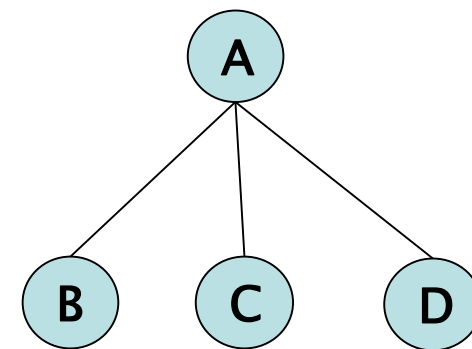
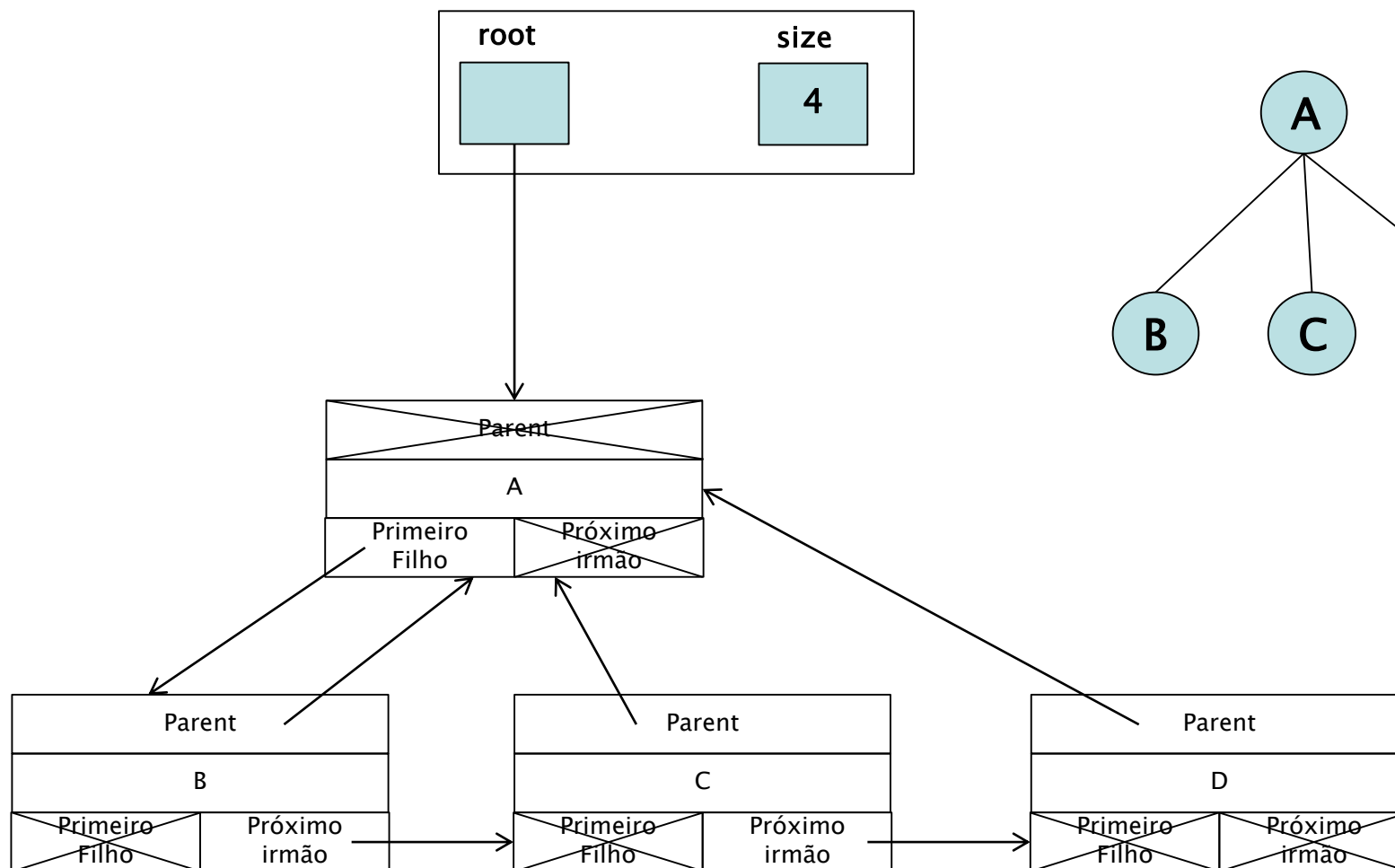


# Representando Nó da árvore



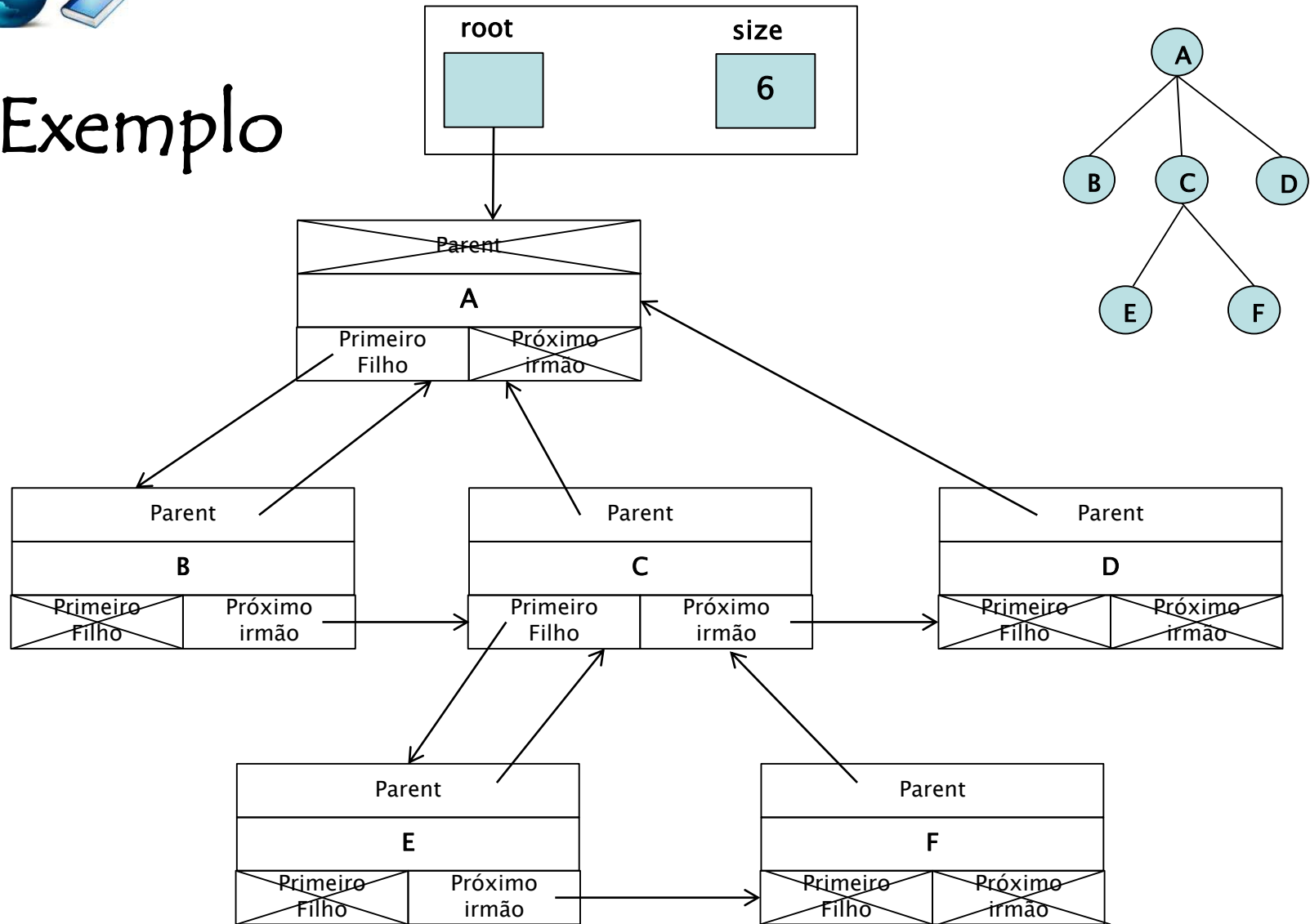


## Exemplo





# Exemplo







# 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**

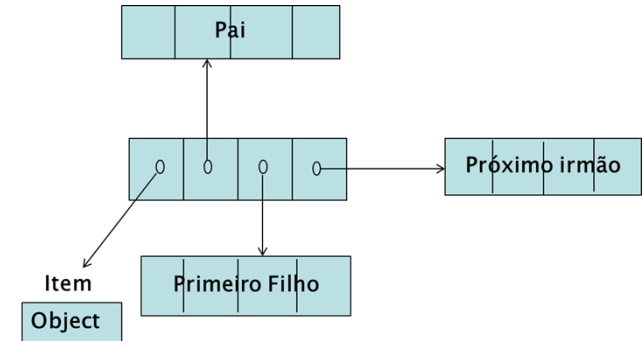
```
package maua;
```

```
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...");  
}
```



**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

```
package maua;
```

```
public class Tree {
```

```
    Node_Tree root;  
    int size;
```

```
    public Tree() {
```

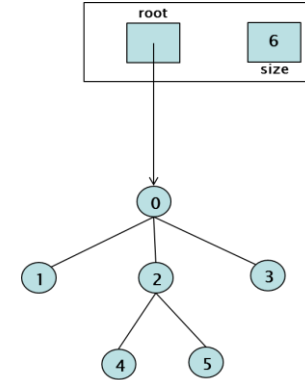
```
        this.root = null;  
        this.size = 0;
```

```
    }
```

```
    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**.



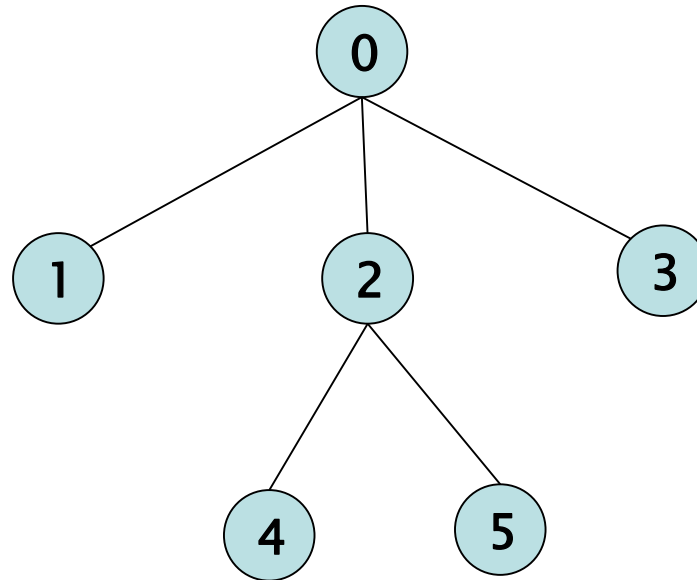
# Percurso – Preorder

- 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

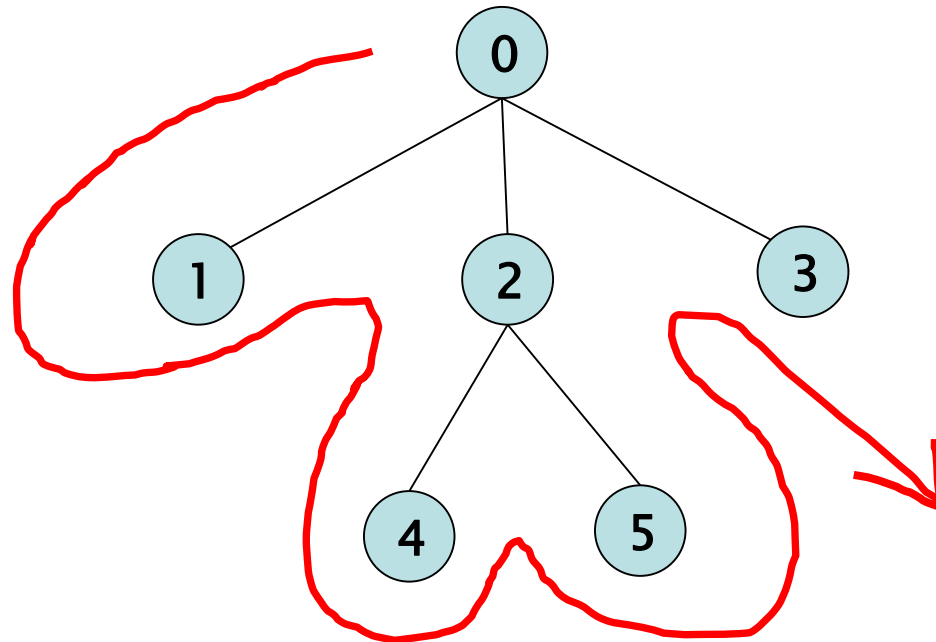


- Imprimir os nós da árvore com o uso da travessia preorder.





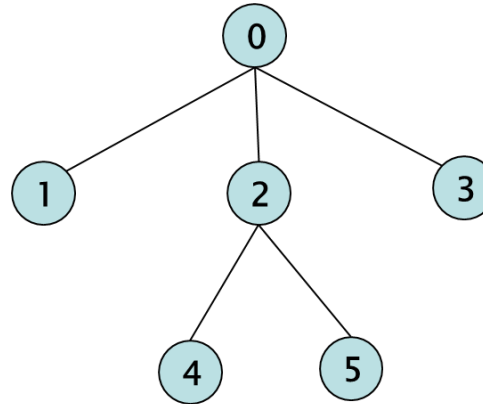
# Percurso – Preorder



- 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



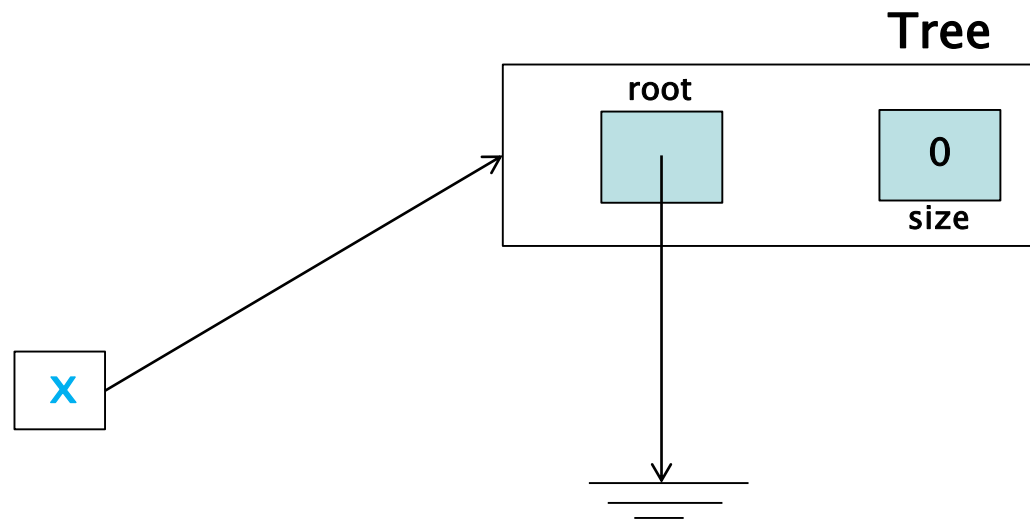
1. Construir a estrutura de dados que corresponde à árvore (estrutura de controle)
2. Criar o nó root e vinculá-lo à árvore
3. Construir os nós que compõem a árvore
4. Estabelecer os relacionamentos hierárquicos entre os nós
5. Aplicar o algoritmo de preorder na raiz da árvore.



# 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();
```

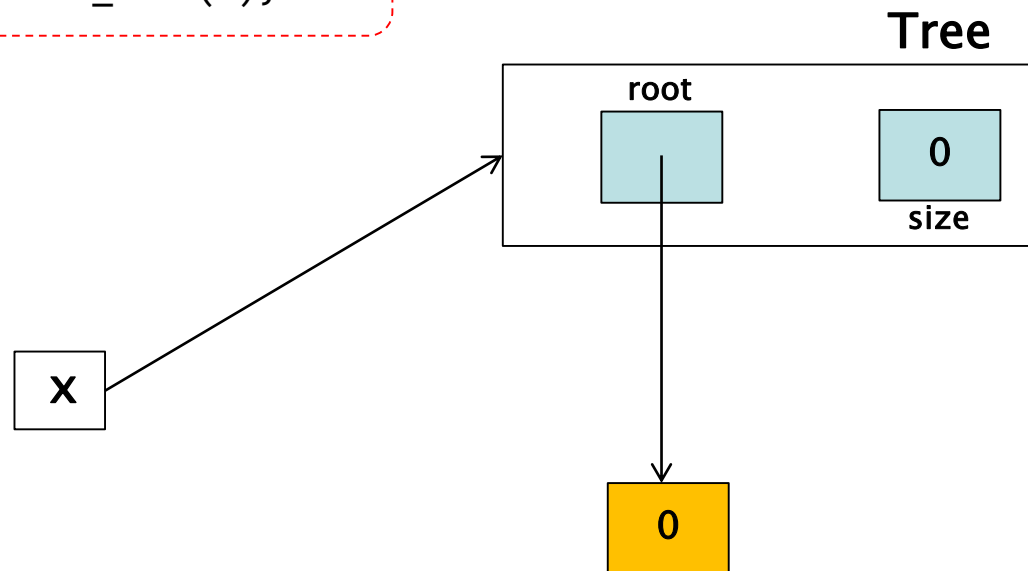




## 2. Criar o nó root e vinculá-lo à árvore

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

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



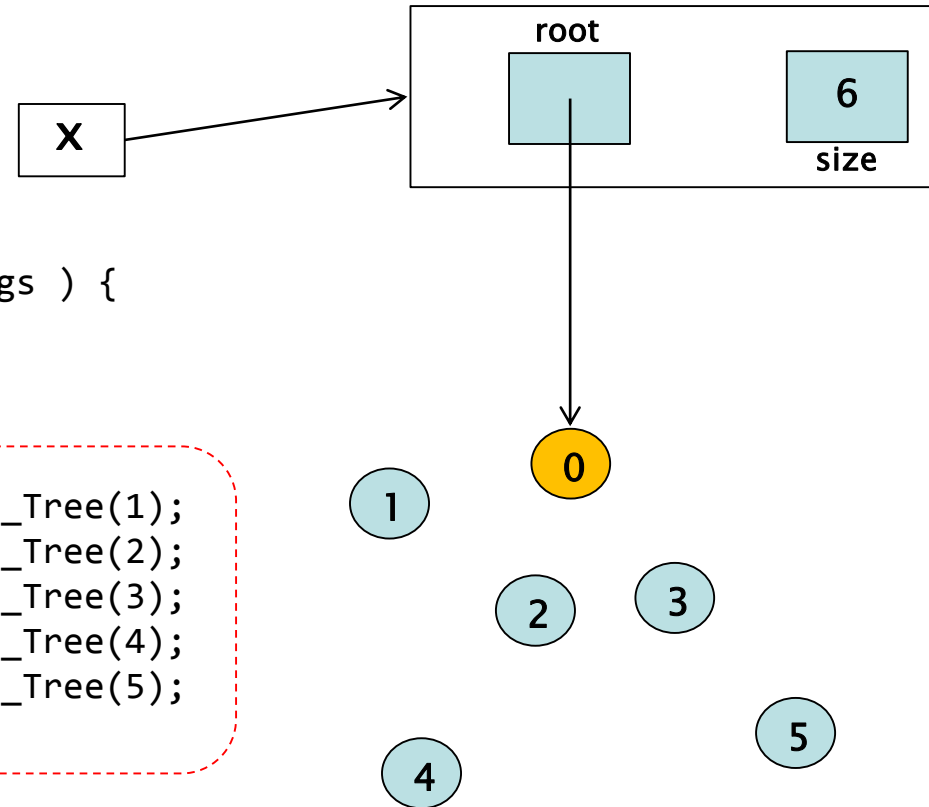


### 3. Construir os nós que compõem a árvore

```
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);
```

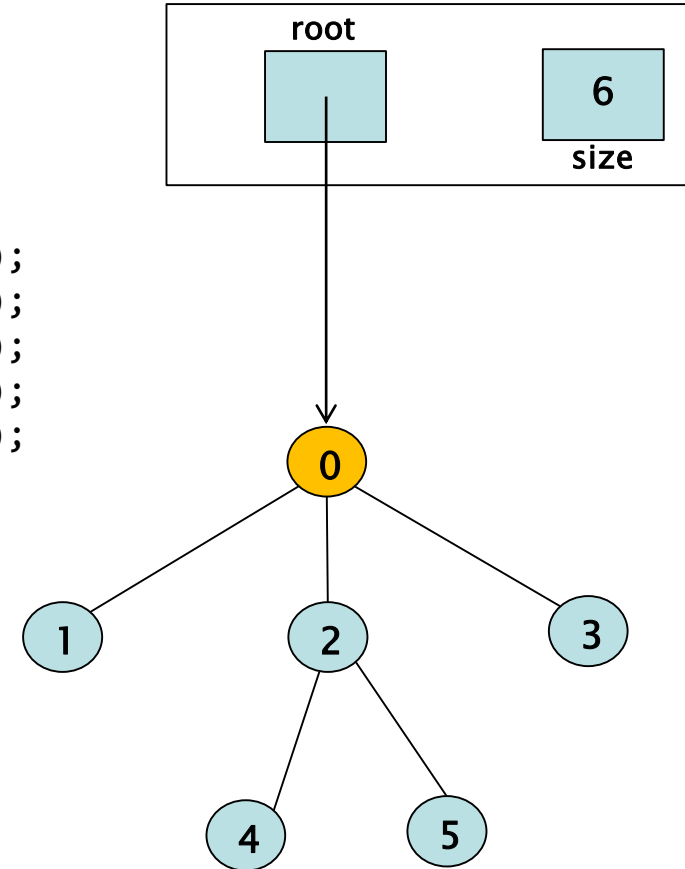
```
        x.root.firstChild = no_1;
        no_1.parent = x.root;
        no_1.Next = no_2;
```

```
        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;
```

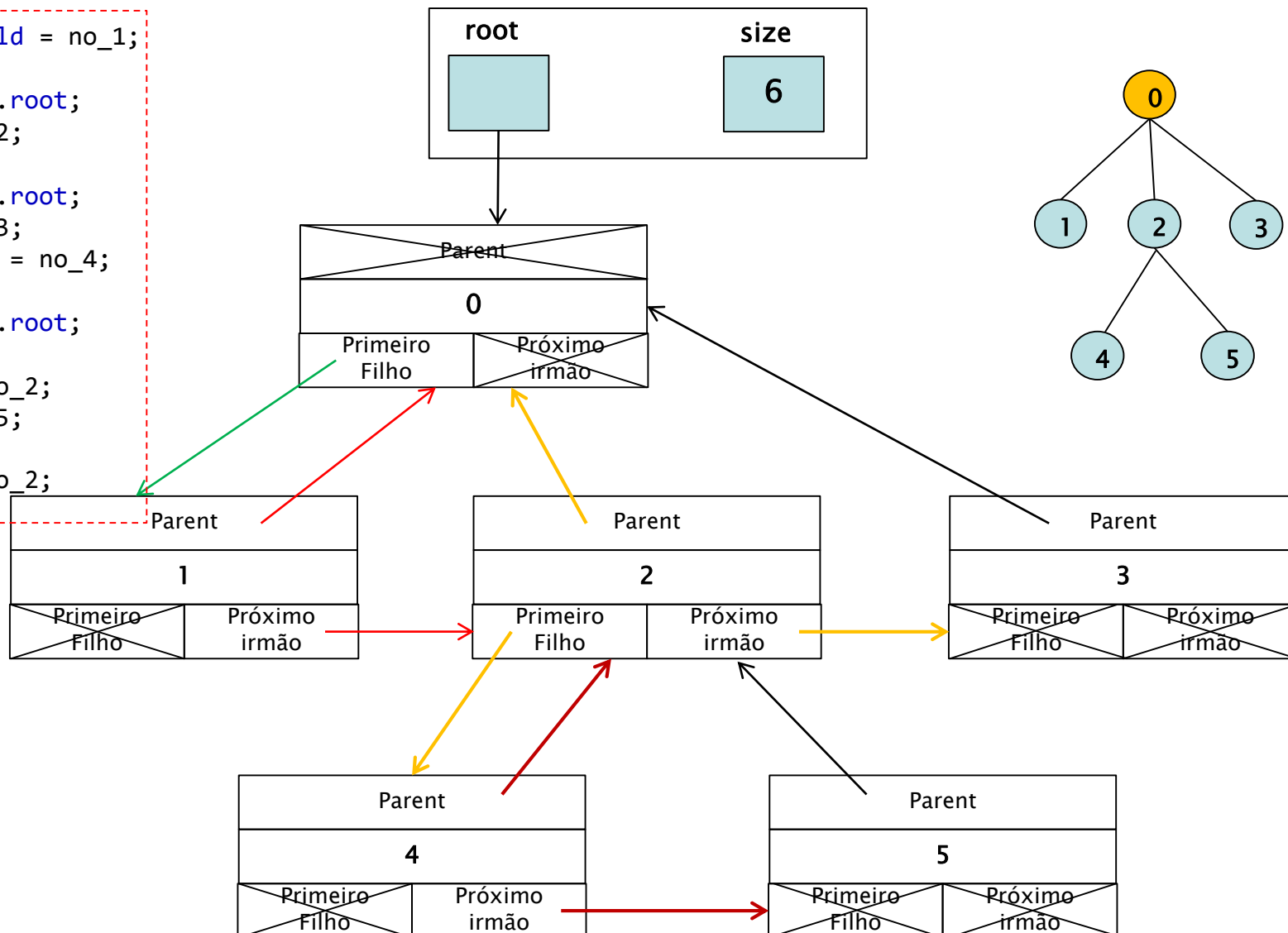
```
no_2.firstChild = no_4;
```

```
no_3.parent = x.root;
```

```
no_4.parent = no_2;
```

```
no_4.Next = no_5;
```

```
no_5.parent = no_2;
```





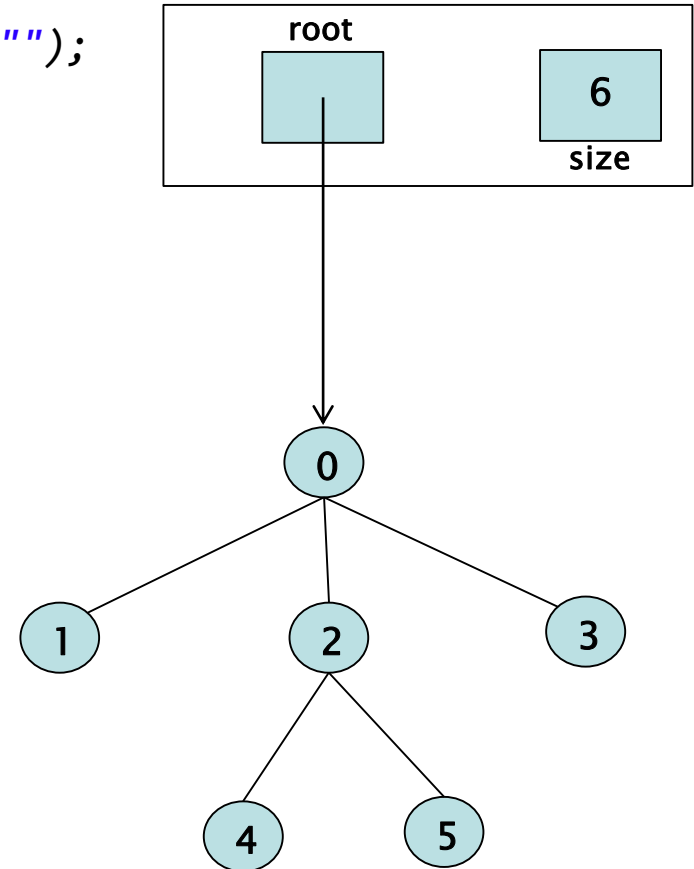
## 5. Aplicar o algoritmo de preorder na raiz da árvore.

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

```
}
```

```
}
```

Resposta do programa:







```
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;

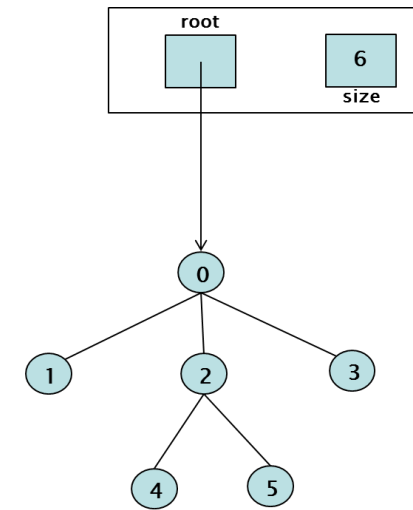
        no_3.parent = x.root;

        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 ("");

    }
}
```



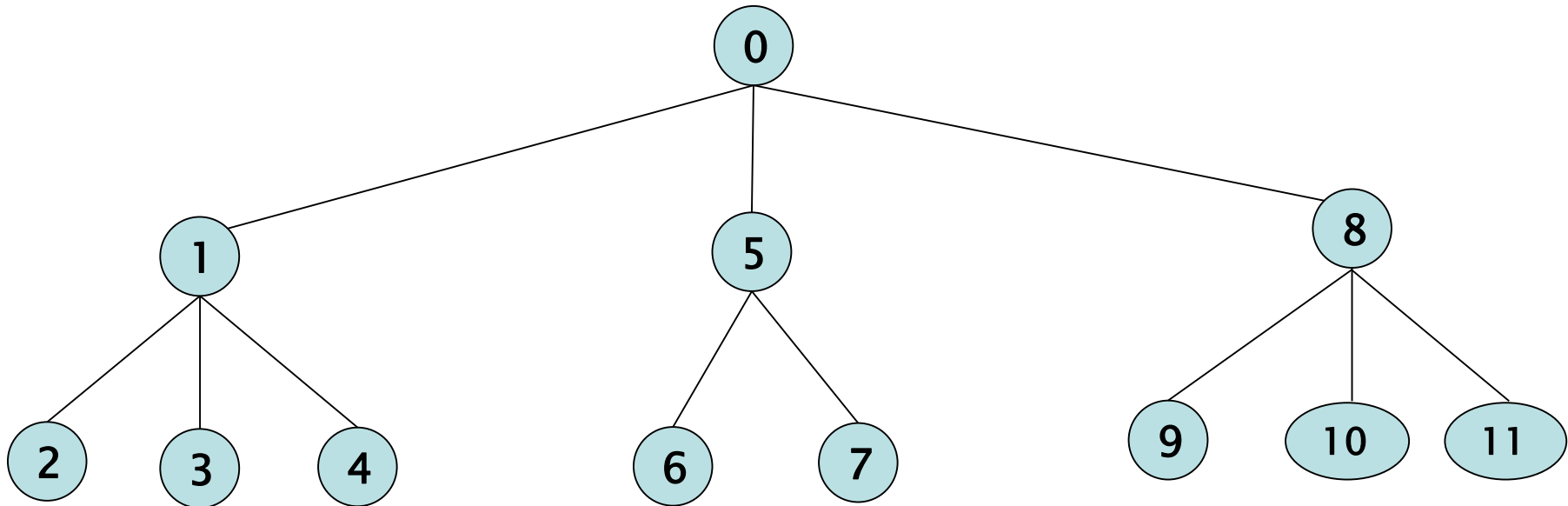
Resposta do programa:



0  
1  
2  
4  
5  
3



# Outro exemplo

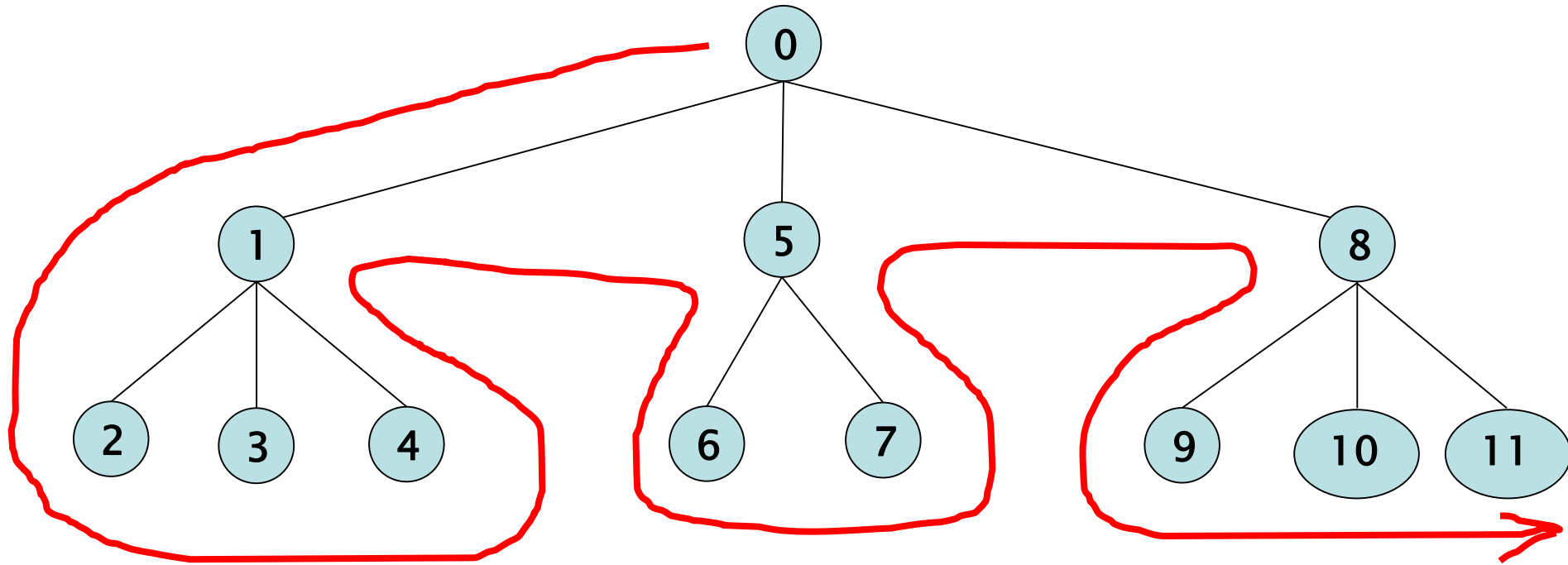


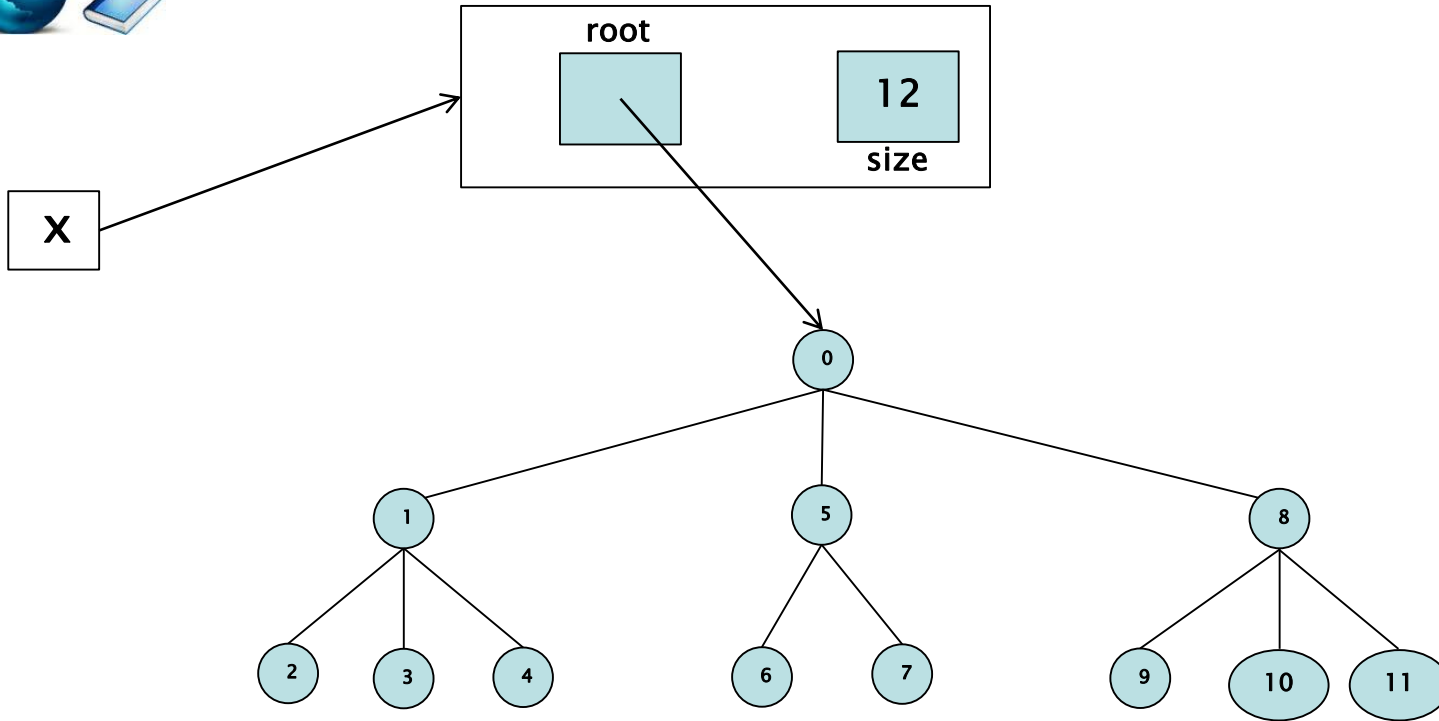
Qual o percurso preordem desta árvore ?





# 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;
```



```
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;
```

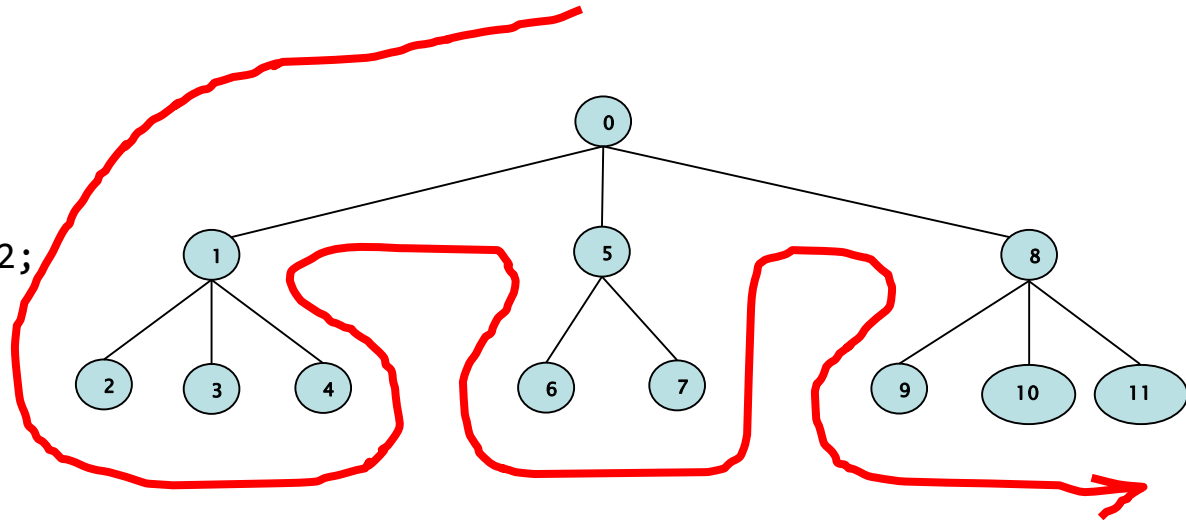
```
no_5.firstChild=no_6;
no_6.Next = no_7;
no_6.parent = no_5;
no_7.parent = no_5;
```

```
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();
```

```
}
```

```
}
```



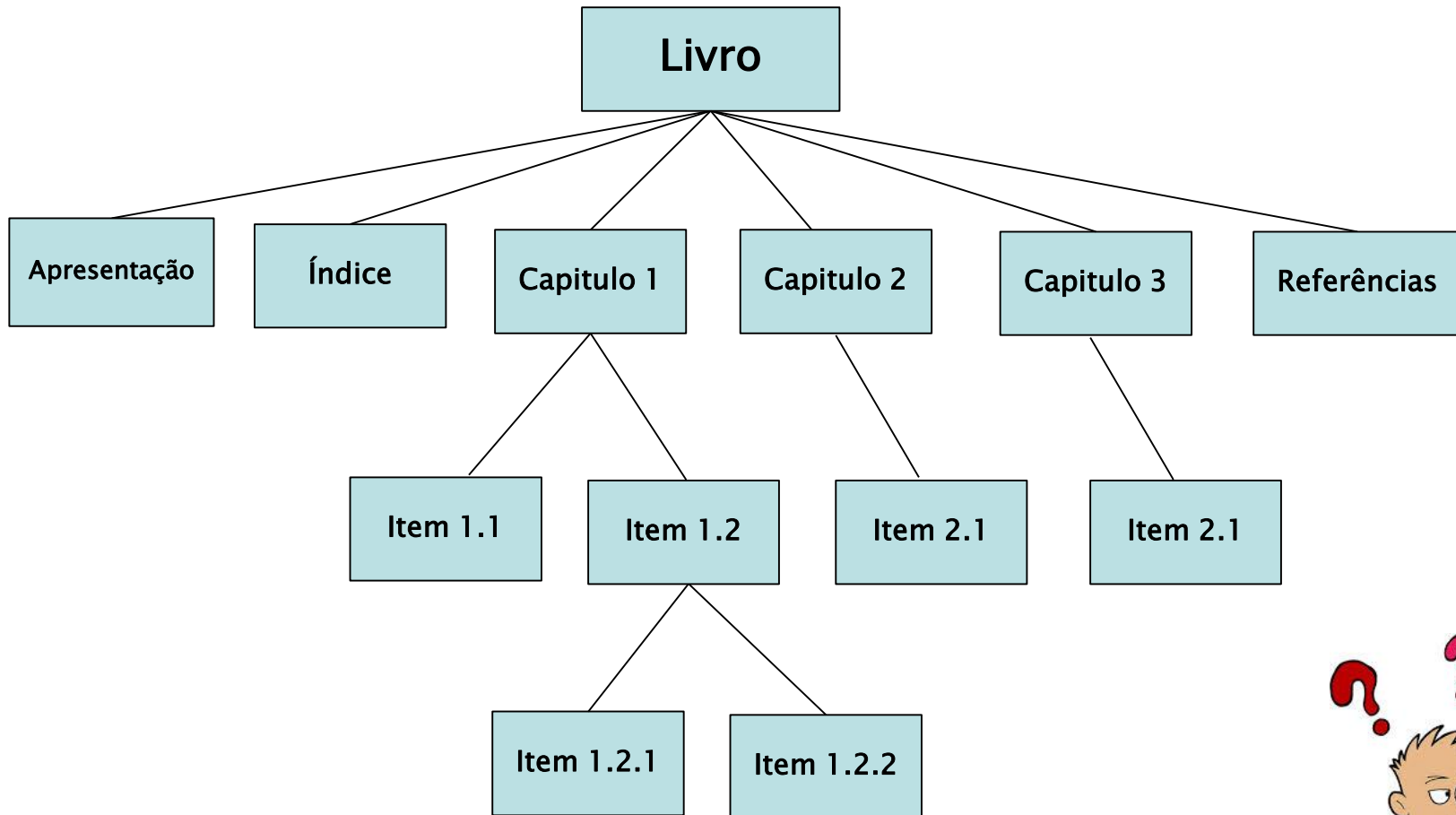
**Resposta: Preorder** →



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



# Exercício



Qual o percurso preordem desta árvore ?





# Percurso – Postorder

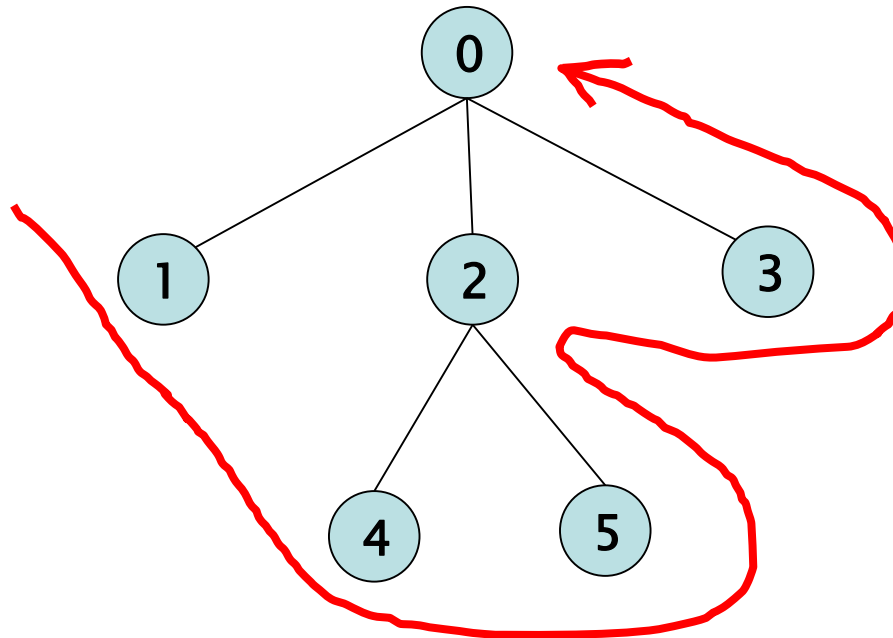
- Este algoritmo pode ser visto como o oposto do percurso preorder, pois as sub-árvores dos filhos são recursivamente atravessadas e em seguida o root é visitado.

```
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.



```
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);
```

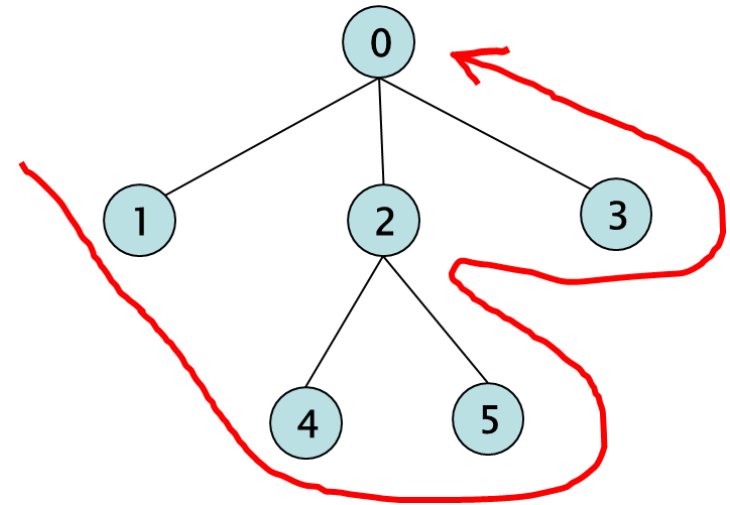
```
x.root.firstChild = no_1;  
no_1.parent = x.root;  
no_1.Next = no_2;
```

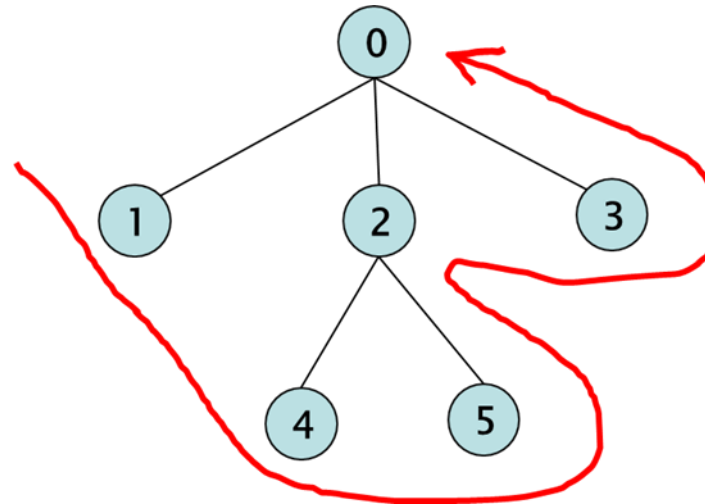
```
no_2.parent = x.root;  
no_2.Next = no_3;
```

```
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;
```



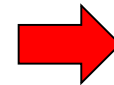


```
x.root.postorder();  
System.out.println ("");
```

```
}
```

```
}
```

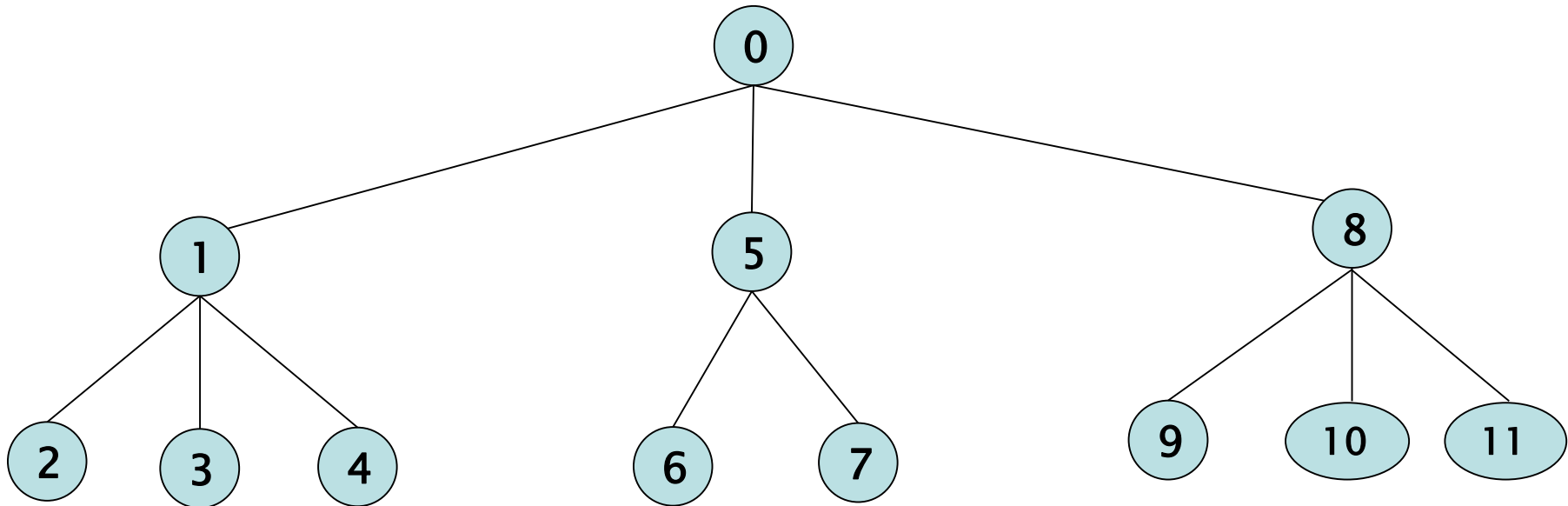
**Resposta: Posorder**



1  
4  
5  
2  
3  
0

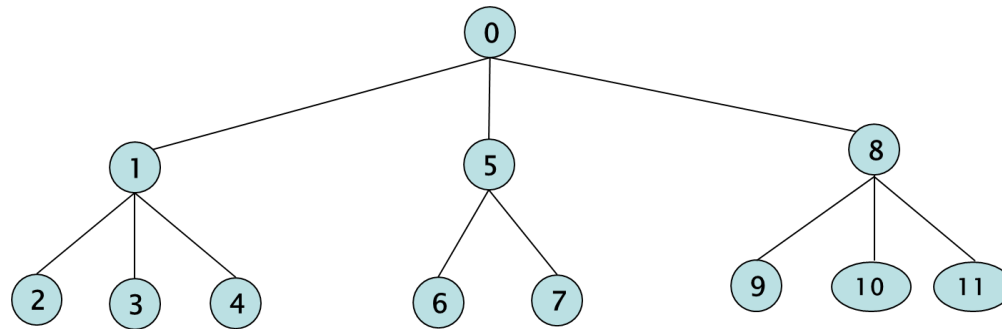


## Outro exemplo



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);
```

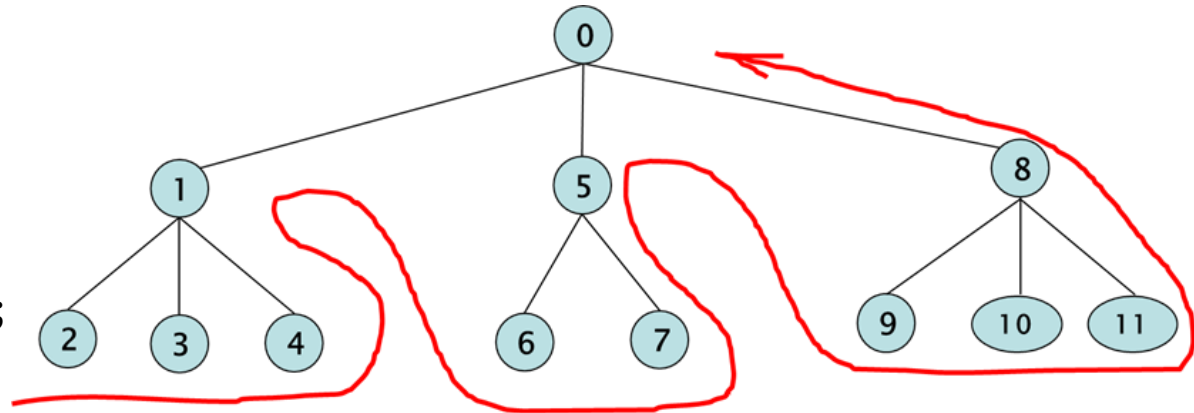
```
        Node_Tree no_1 = new Node_Tree(1);  
        Node_Tree no_2 = new Node_Tree(2);  
        Node_Tree no_3 = new Node_Tree(3);  
        Node_Tree no_4 = new Node_Tree(4);  
        Node_Tree no_5 = new Node_Tree(5);  
        Node_Tree no_6 = new Node_Tree(6);  
        Node_Tree no_7 = new 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;
```

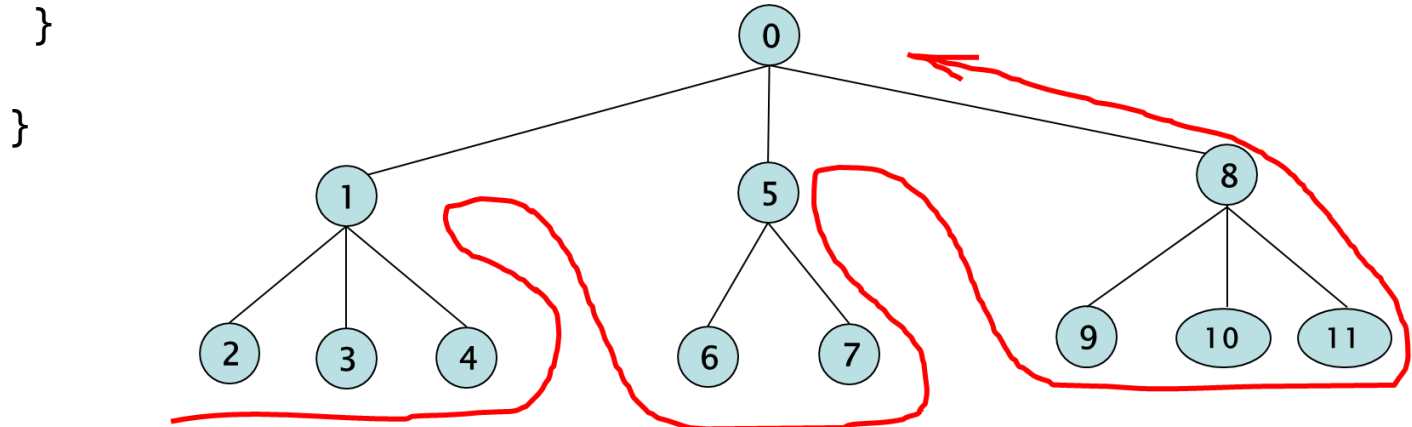
```
no_5.firstChild=no_6;  
no_6.Next = no_7;  
no_6.parent = no_5;  
no_7.parent = no_5;
```





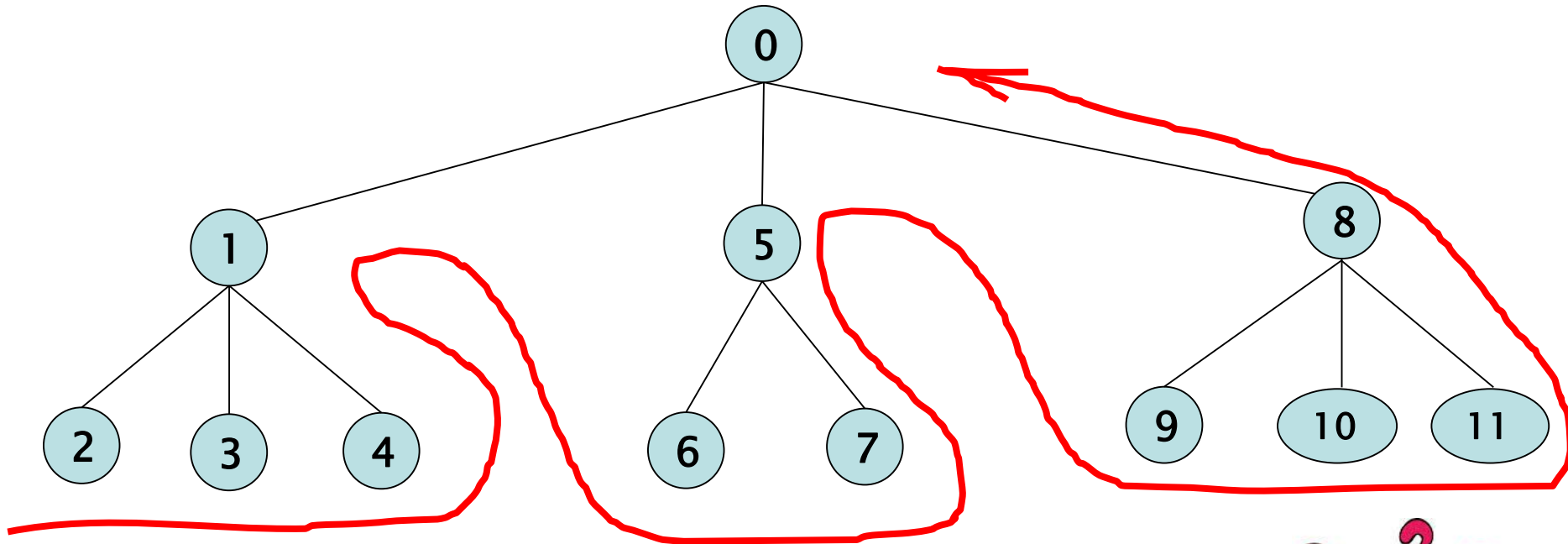
```
no_8.firstChild = no_9;  
no_9.Next = no_10;  
no_10.Next = no_11;  
no_9.parent = no_8;  
no_10.parent = no_8;  
no_11.parent = no_8;
```

```
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



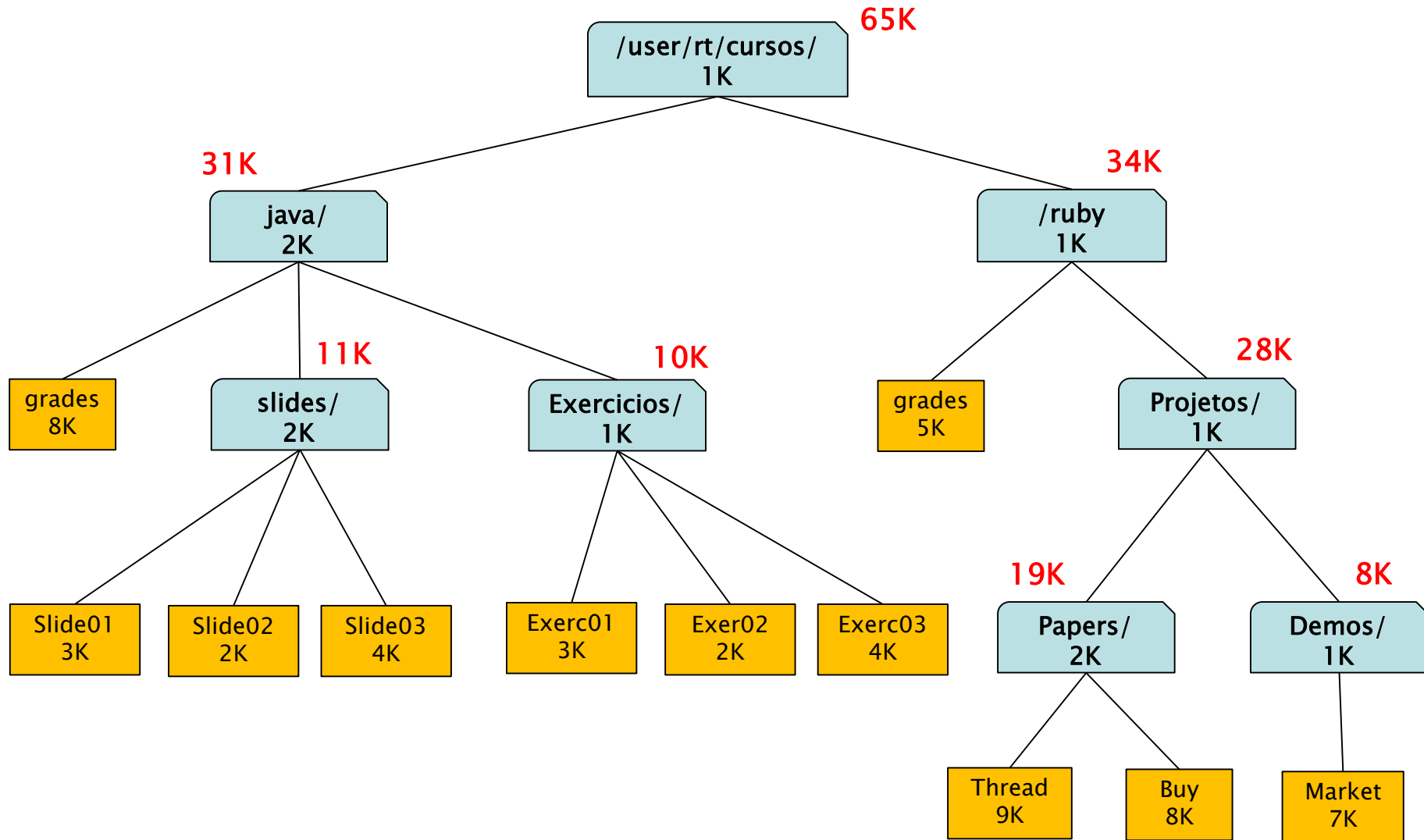




# Aplicação travessia postorder

- 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.

