



# Unidade 11 – Árvores



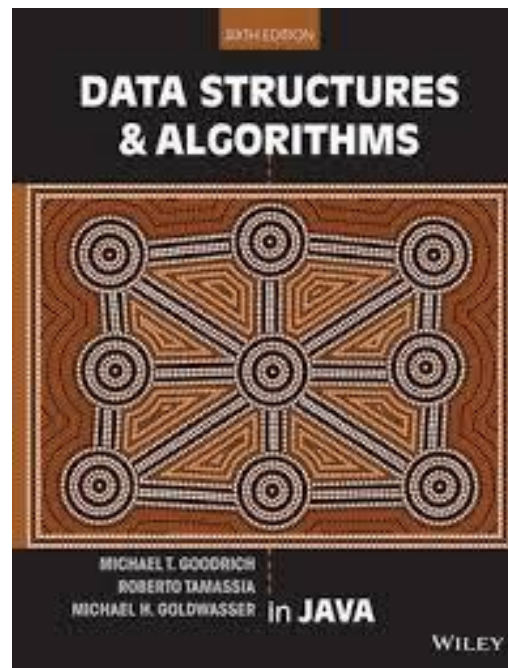
# Bibliografia

- Data Structures and Algorithms in Java – Fourth Edition – Roberto Tamassia – Michael T. Goodrich – John Wiley & Sons, Inc
- Head First Java, 2nd Edition by Kathy Sierra and Bert Bates
- Estrutura de Dados e Algoritmos – Bruno R. Preiss, Editora Campus, 2001
- Estrutura de Dados e Algoritmos em Java – Robert Lafore, Editora Ciência Moderna, 2005
- Algoritmos e Estrutura de Dados – Niklaus Wirth – Editora Prentice Hall do Brasil, 1989
- Estrutura de Dados e Algoritmos em C++, Adam Drozdek – Thompson
- Introdução à Estrutura de Dados, Celes, Cerqueira, Rangel – Elsevier
- FREITAS, Aparecido V. de – 2022 – Estruturas de Dados: Notas de Aula.
- CALVETTI, Robson - 2015 – Estruturas de Dados: Notas de Aula.



## Leitura Recomendada para a Unidade 4

- ⊕ Data Structures and Algorithms in Java (\*), Roberto **Tamassia** and Michael T. **Goodrich**, Sixty Edition – **2014** , Seção **8.1**



(\*) Em português, Estrutura de Dados e Algoritmos em Java



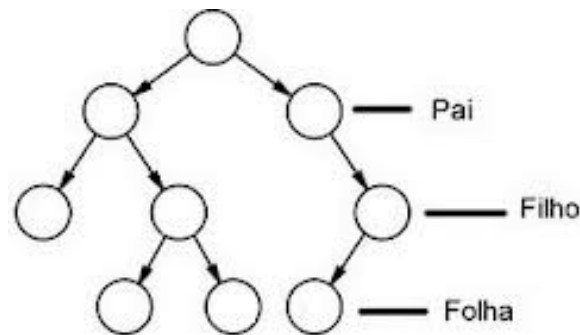
# Árvores

- **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 (**ROOT**).



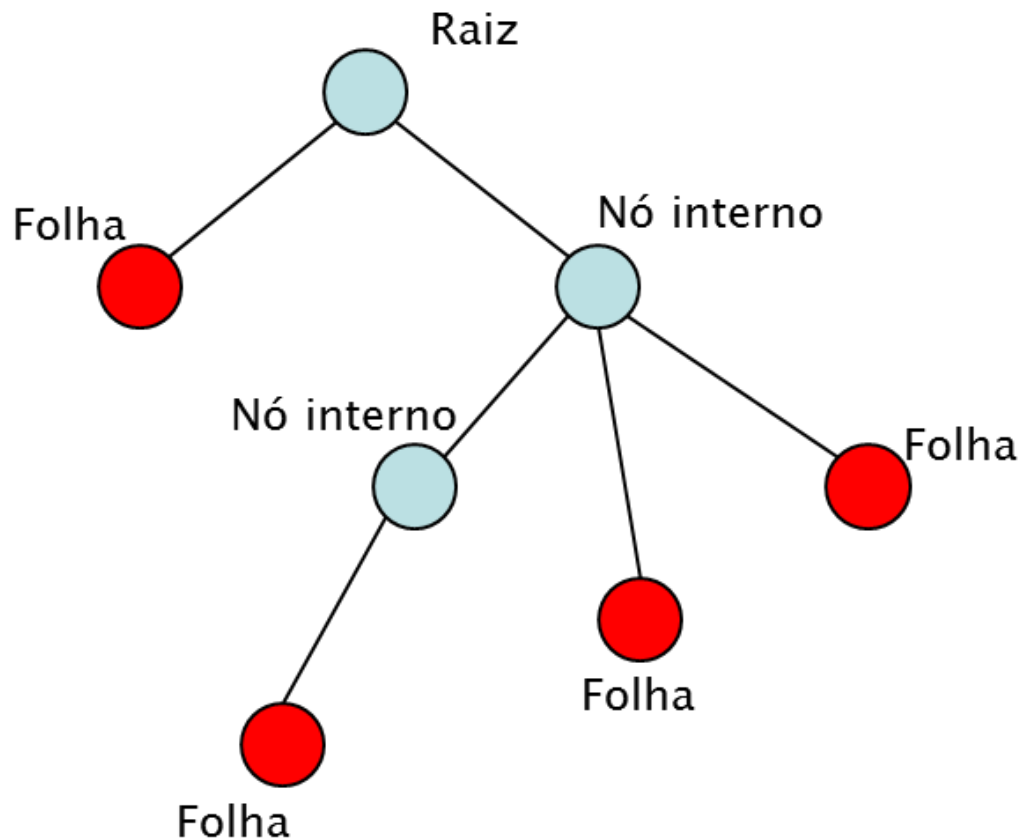
# Definição Formal

- 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**.



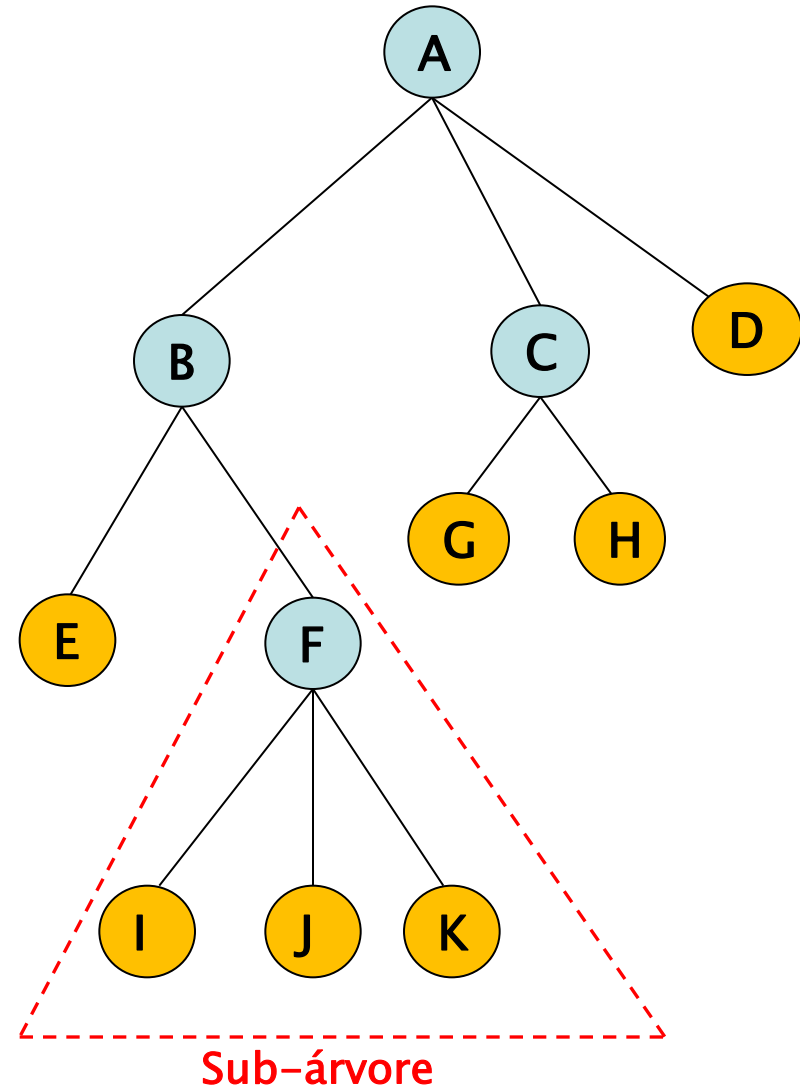
# Outros relacionamentos

- 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

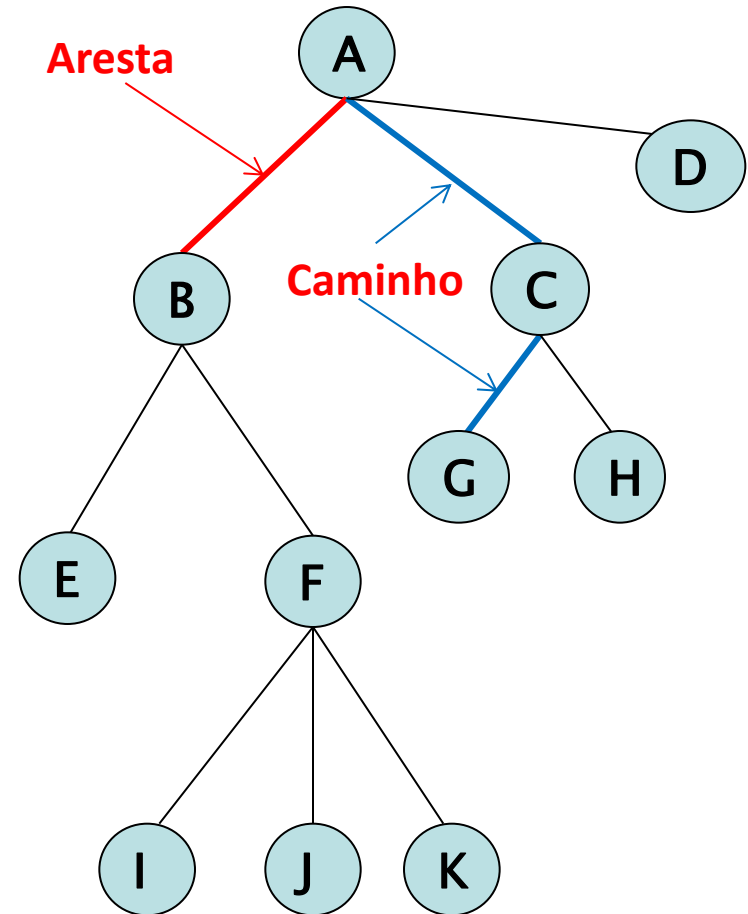
- ⊕ **Raiz** (root): Nó sem pai (A)
- ⊕ **Nó interno**: Nó com pelo menos um filho (A,B,C,F)
- ⊕ Nó externo ou nó **folha**: nó sem filhos (D,E,G,H,I,J,K)
- ⊕ **Ancestral** de um nó: pai, avô, bisavô, ...
- ⊕ **Descendente** de um nó: filho, neto, bisneto, ...
- ⊕ **Sub-Árvore**: árvore formada por um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





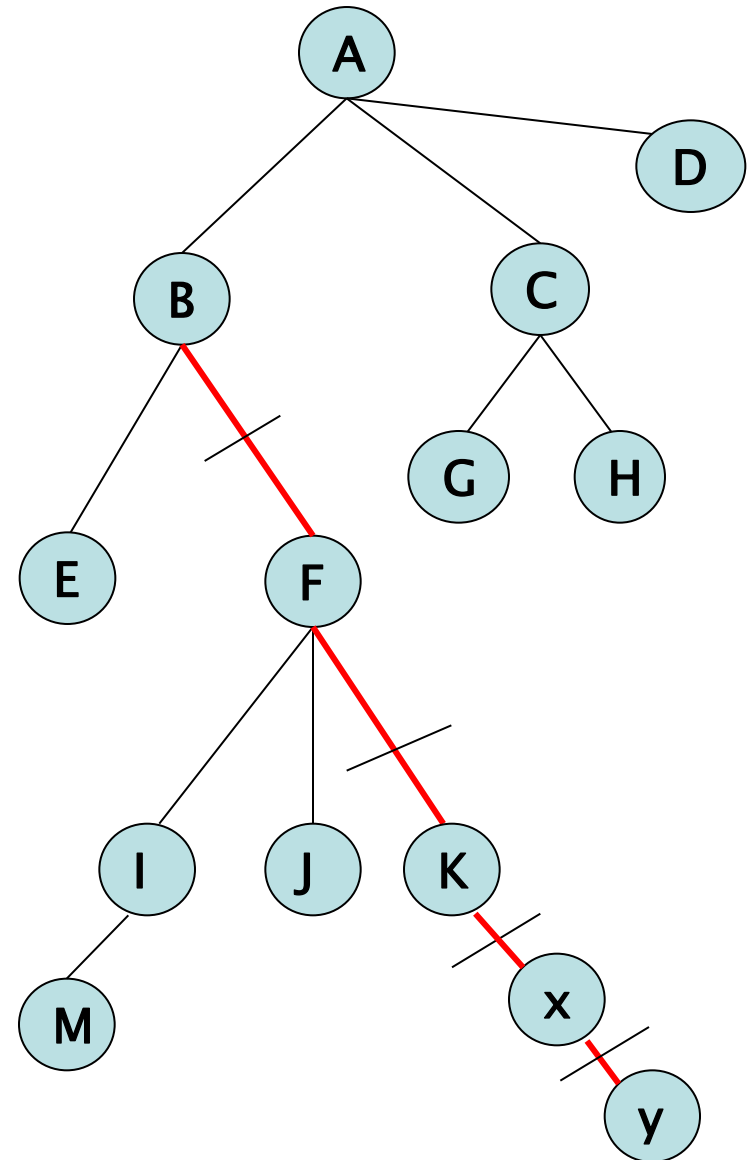
# Definições

- ⊕ **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  $((A,C),(C,G)) = 2$  ).



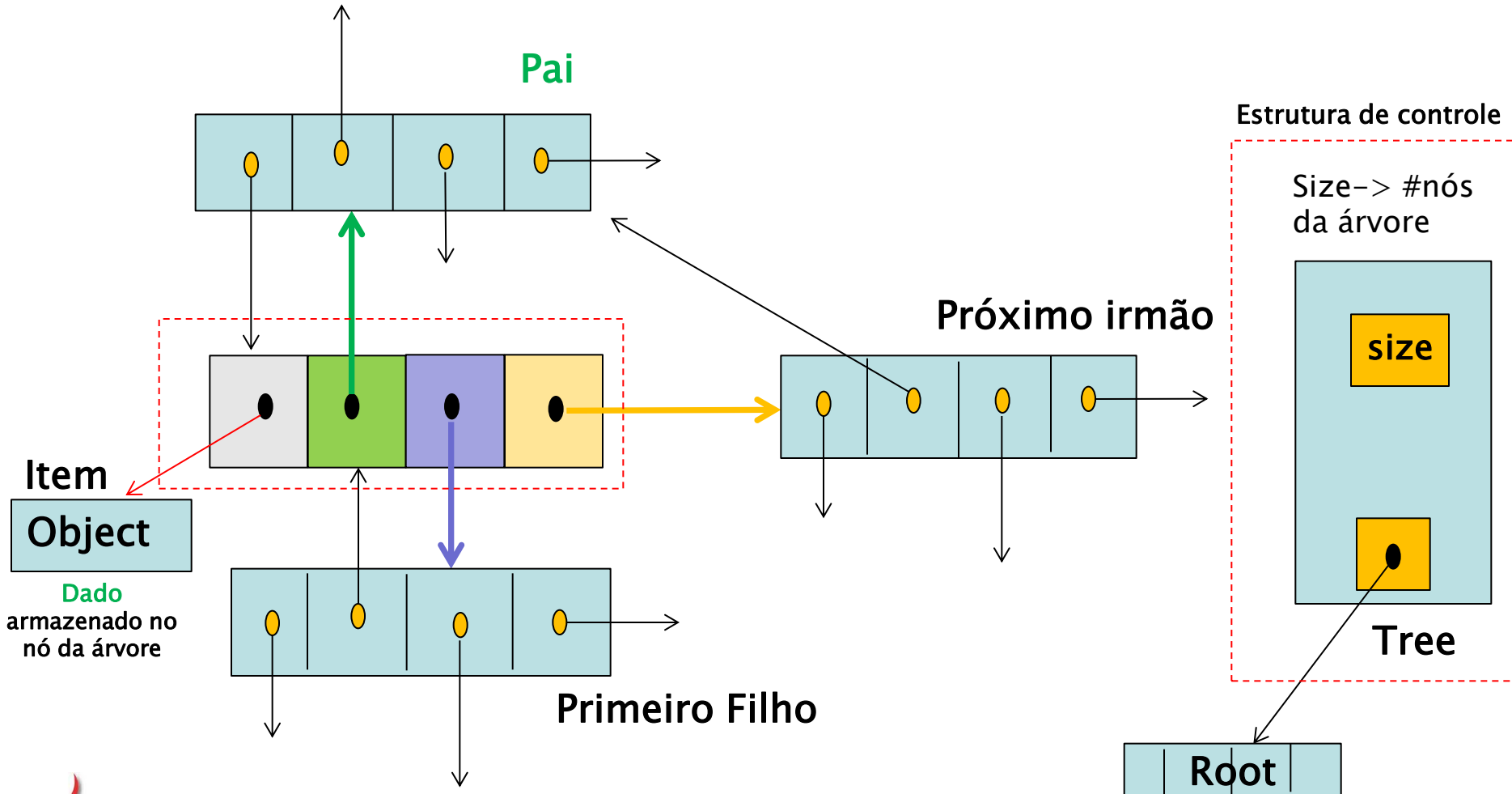
# Definições

- ⊕ **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 o nó de nível mais baixo. (**Altura(B) = 4**) .
- ⊕ **Altura de qualquer folha:** ZERO
- ⊕ **Altura da Árvore = Altura da Raiz**



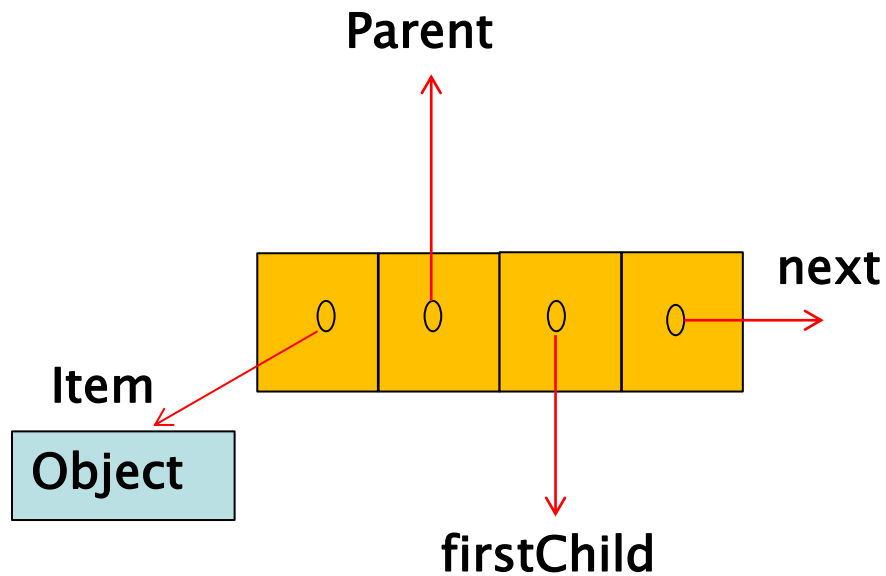
# Representando Nó de Árvores

- 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;  
  
    .  
    .  
    .  
  
}
```

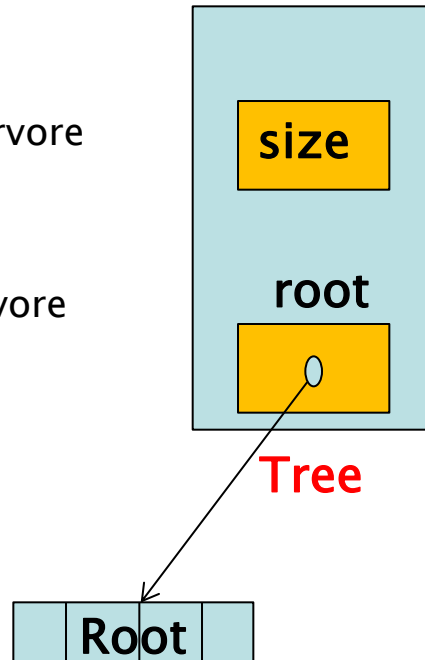


# Representando Árvores

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

**size** -> #nós da árvore

**root** -> raiz da árvore



```
Class Tree {
```

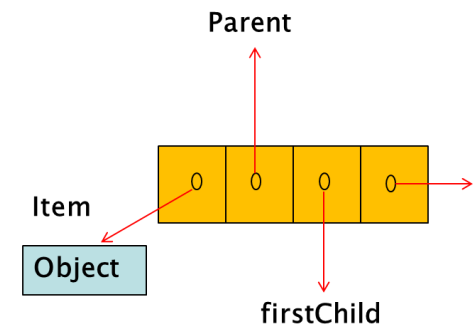
```
    Node_Tree root;  
    int size;
```

```
    .  
    .  
    .
```

```
}
```

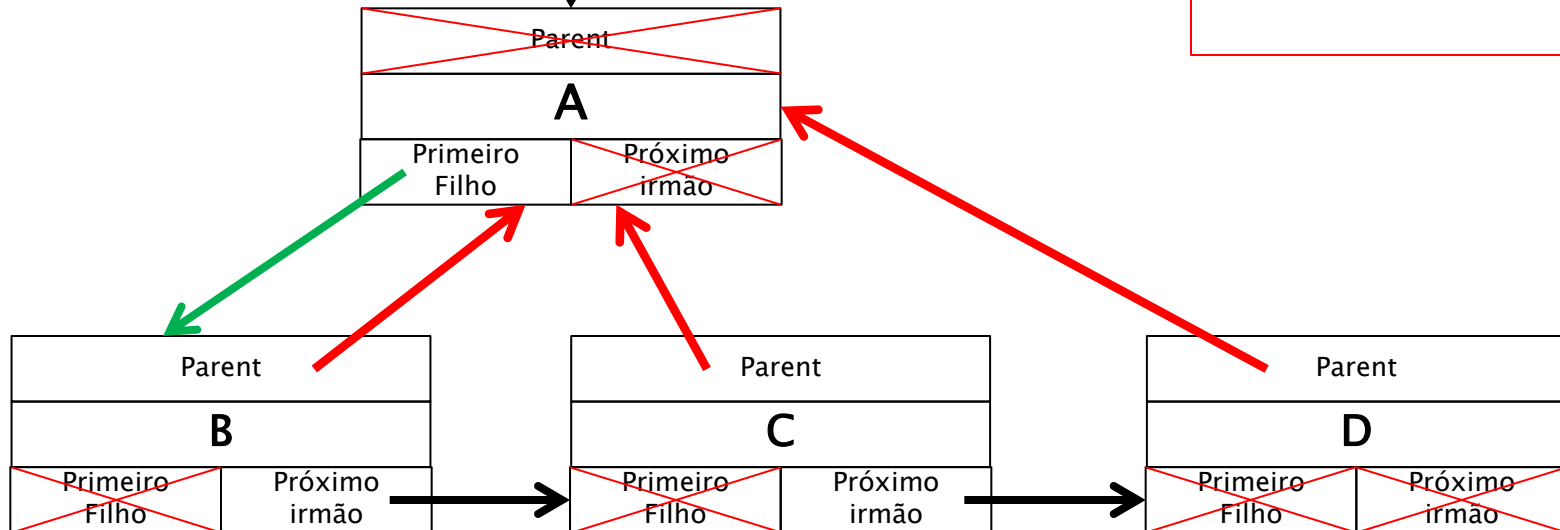
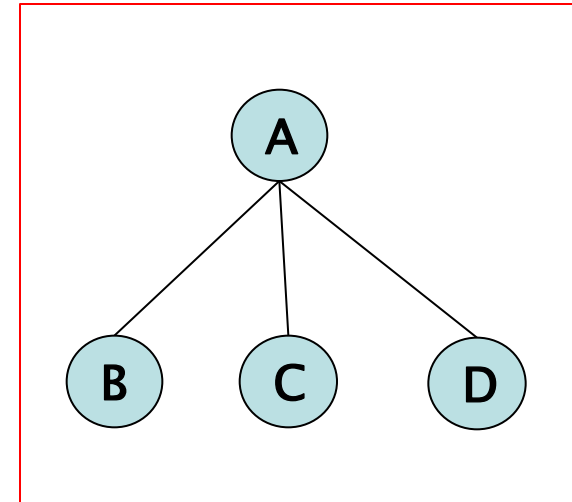
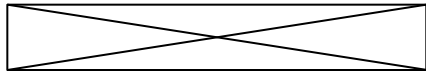


# Representando Nó da árvore

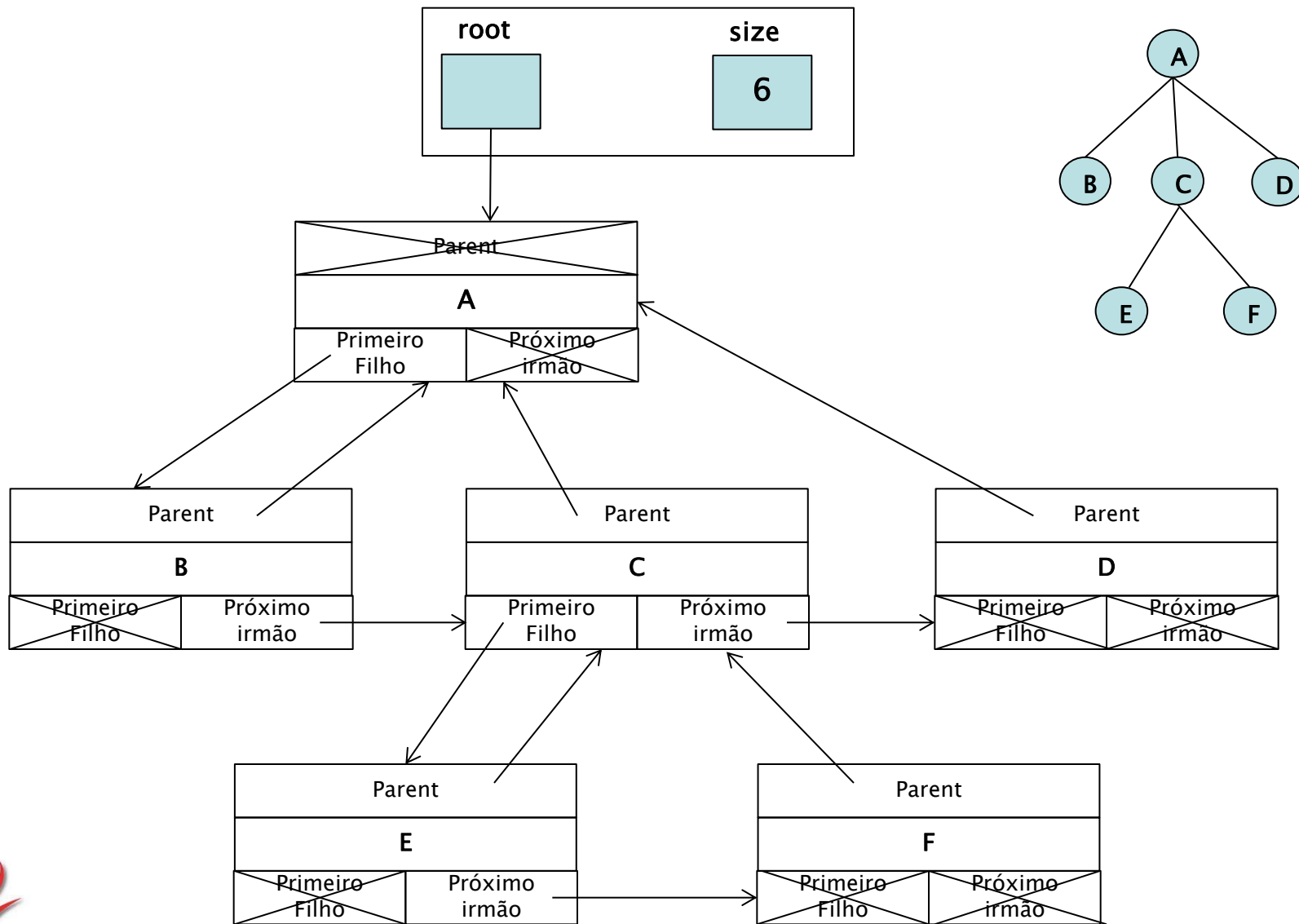


# Exemplo

null



# 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



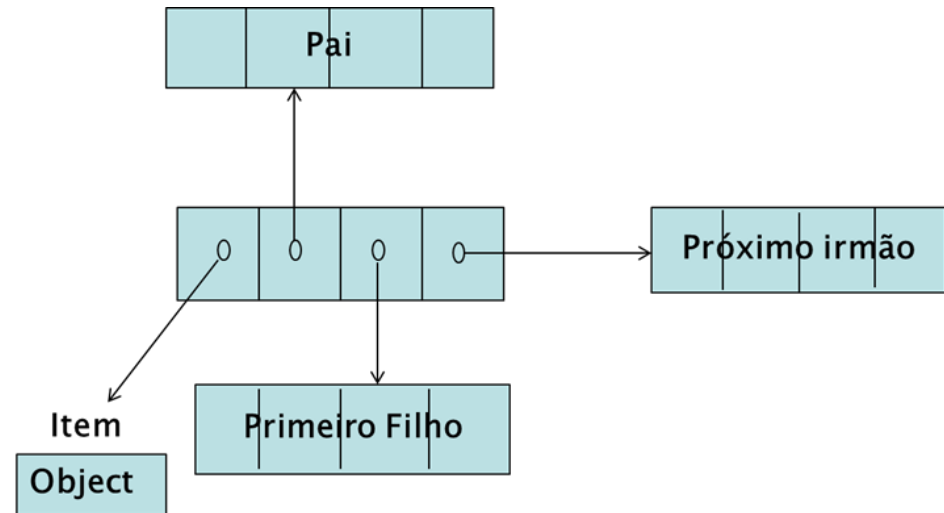
# Classe **Node\_Tree**

```
package uscs;
```

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



**imprime\_Filhos()**: Imprime dados dos filhos do nó

```
public void imprimeFilhos() {  
  
    if (this.firstChild == null)  
        System.out.println("Node nao tem filhos....");  
    else {  
  
        Node_Tree trab = this.firstChild;  
  
        while (trab != null ) {  
  
            System.out.println(trab.item);  
            trab = trab.next;  
        }  
    }  
}
```



**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;

    int h=0;

    Node_Tree trab = this.firstChild;

    while (trab.next != null ) {
        h = Math.max(h , trab.next.height());
        trab = trab.next;
    }

    return 1 + h;

}
```



# Classe Tree

```
package uscs;
```

```
public class Tree {
```

```
    Node_Tree root;
```

```
    int size;
```

```
    public Tree() {
```

```
        this.root = null;
```

```
        this.size = 0;
```

```
    }
```

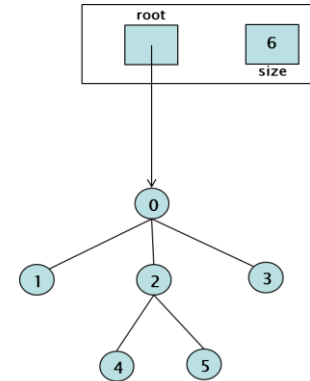
```
    public void insert_root(Integer valor) {
```

```
        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

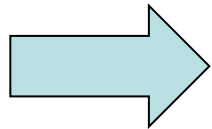
- Atravessar a árvore significa visitar uma única vez cada nó da árvore.
- Existem basicamente dois algoritmos de travessia: **preorder** e **postorder**.



# 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 );
```

```
    Node_Tree trab = this.firstChild;
```

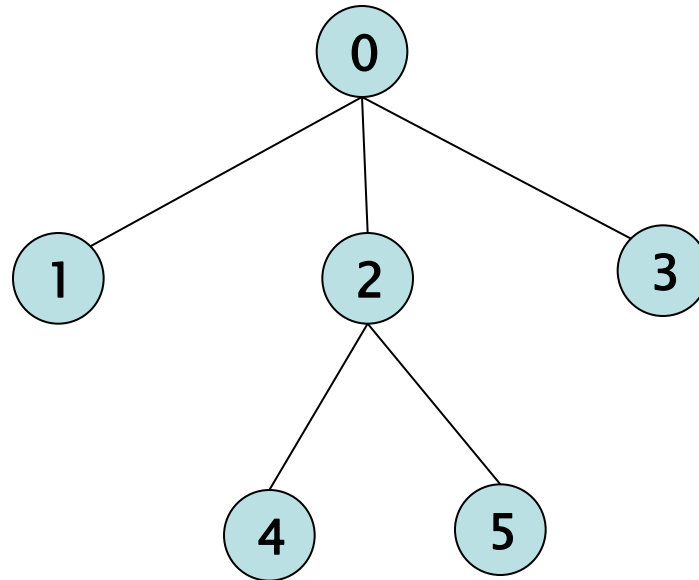
```
    while (trab != null ) {  
        trab.preorder();  
        trab = trab.next;
```

```
    }
```

```
}
```



# Exercício

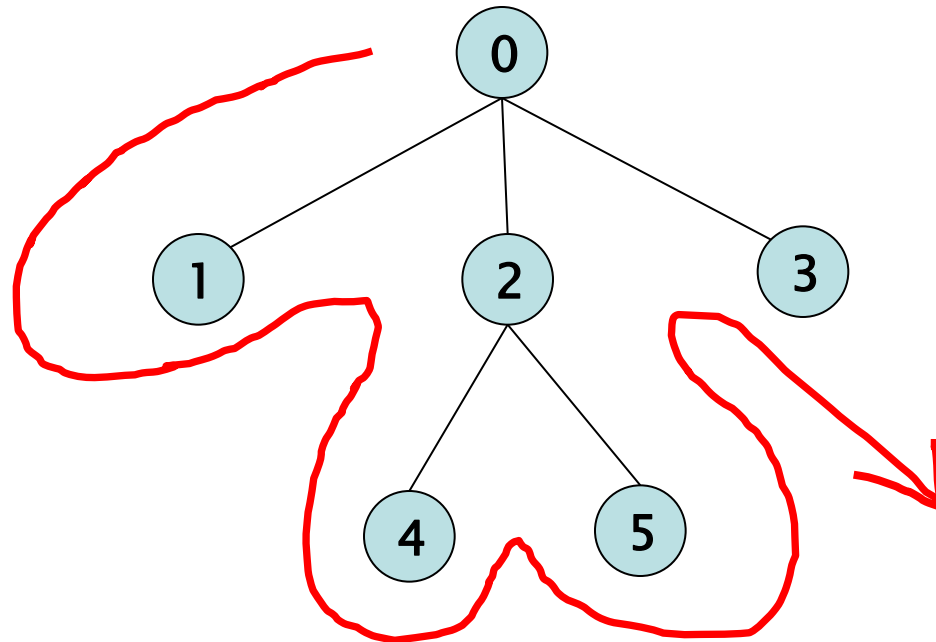


- Imprimir os nós da árvore com o uso da travessia preorder.

0, 1, 2, 4, 5, 3



# Percurso – Preorder

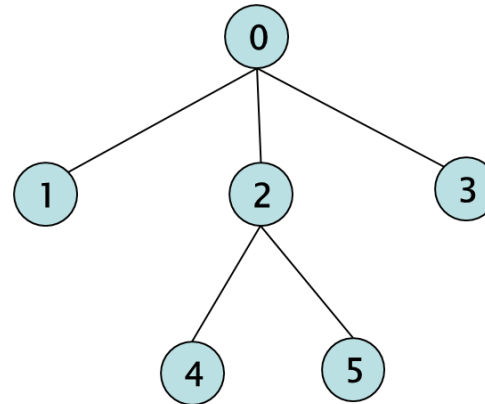


- Nós são visitados nesta ordem: 0 1 2 4 5 3
- Cada nó é visitado somente uma vez, assim o percurso preorder gasta tempo  **$O(n)$** , onde  $n$  é o total de nós da árvore.





# Solução



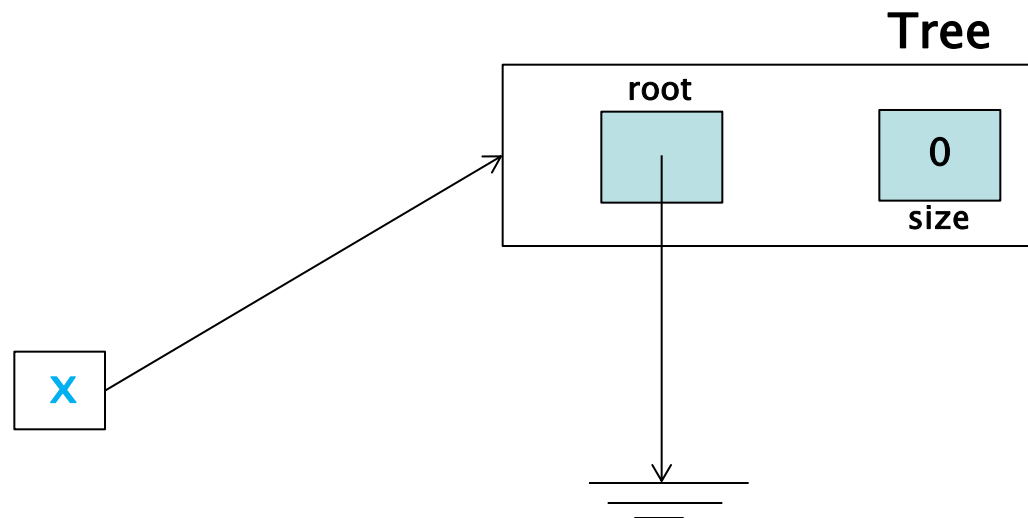
1. Construir a estrutura de dados que corresponde à árvore (estrutura de controle)
2. Criar o nó root e vinculá-lo à árvore
3. Construir os nós que compõem a árvore
4. Estabelecer os relacionamentos hierárquicos entre os nós
5. Aplicar o algoritmo de preorder na raiz da árvore.



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

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

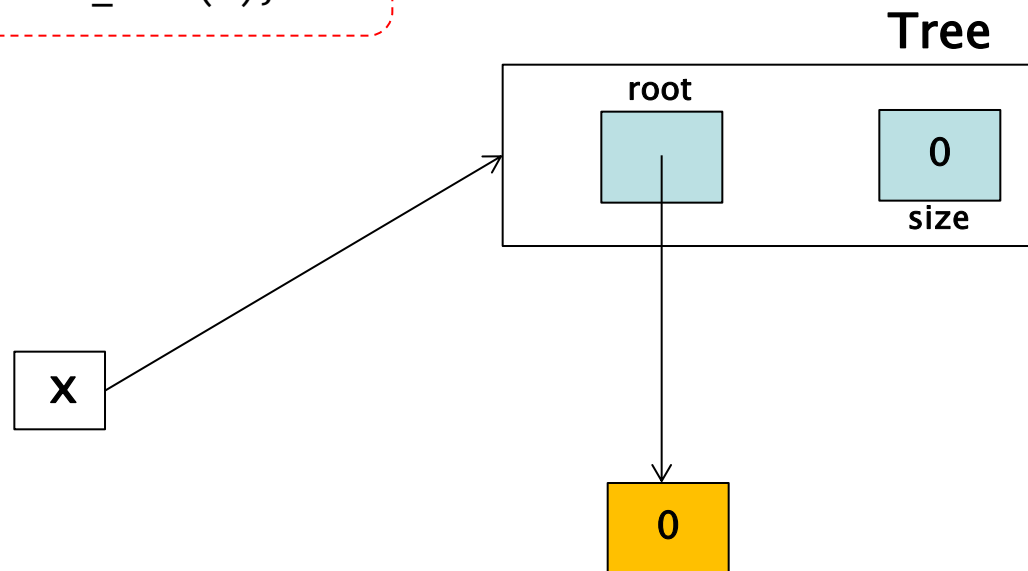
```
        Tree x = new Tree();  
    }
```



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

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

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

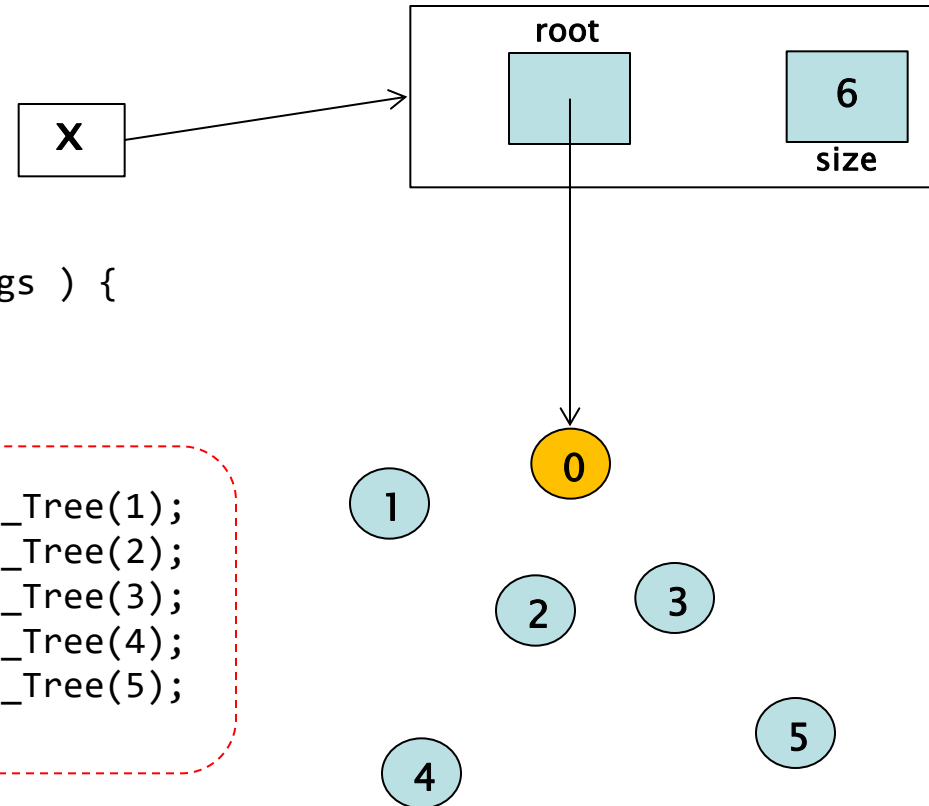


### 3. Construir os nós que compõem a árvore

```
package 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);
```



## 4. Estabelecer os relacionamentos hierárquicos entre os nós

```
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);
```

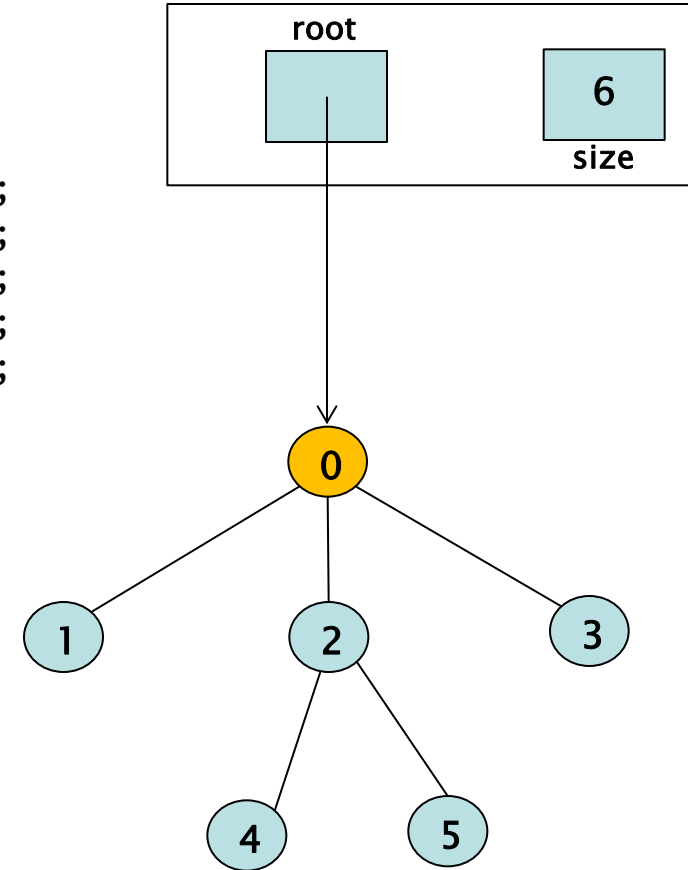
```
        x.root.firstChild = no_1;
        no_1.parent = x.root;
        no_1.next = no_2;
```

```
        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;
```



## 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;
```

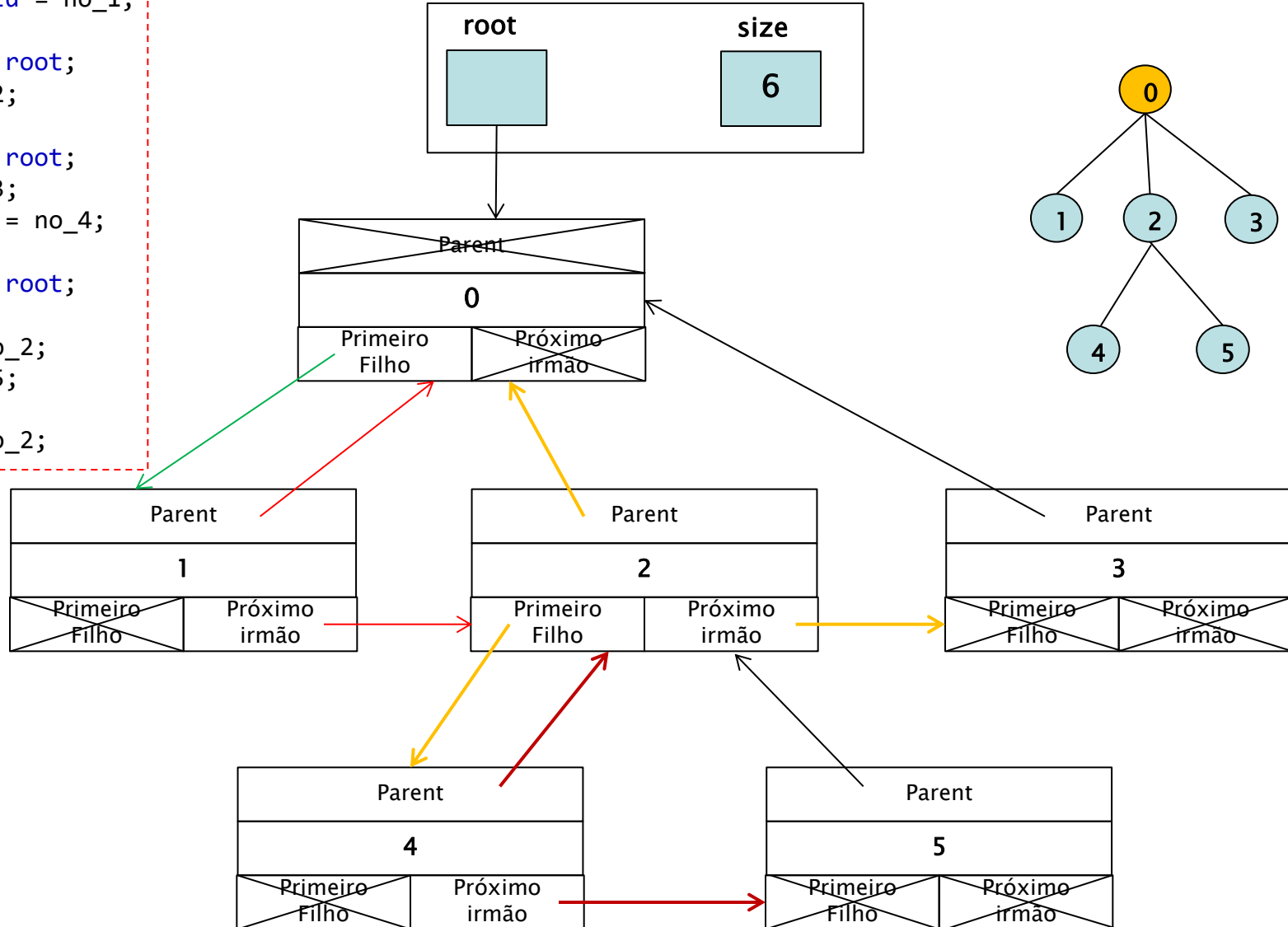
```
no_2.firstChild = no_4;
```

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

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

```
no_4.next = no_5;
```

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



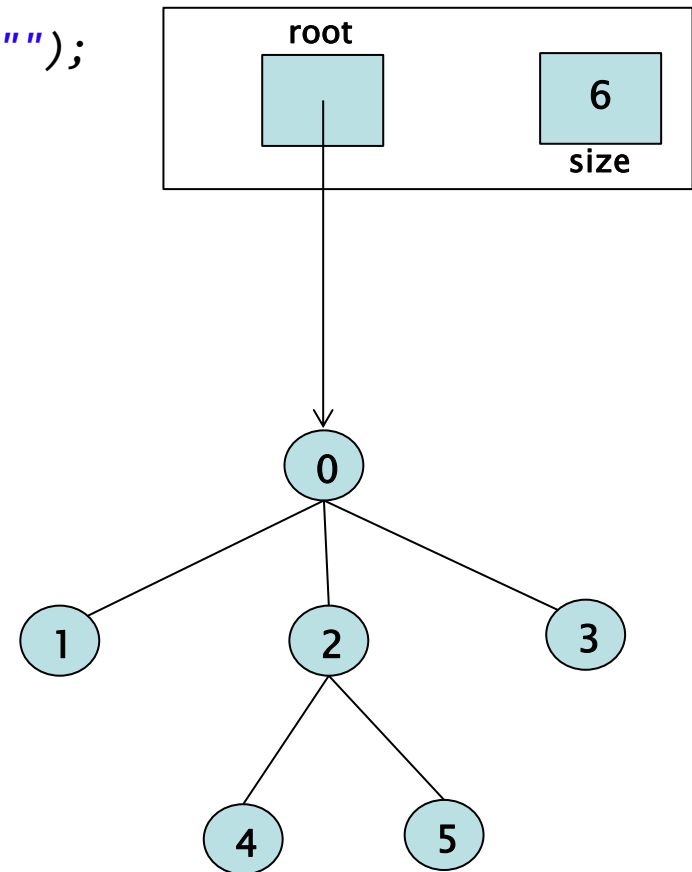
## 5. Aplicar o algoritmo de preorder na raiz da árvore.

```

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

```

Resposta do programa:



```
package 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;
```

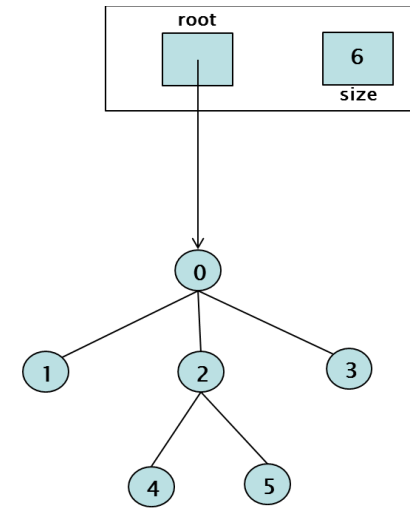
```
        no_5.parent = no_2;
```

```
        x.root.preorder();
```

```
        System.out.println ("");
```

```
    }
```

```
}
```



## Resposta do programa:

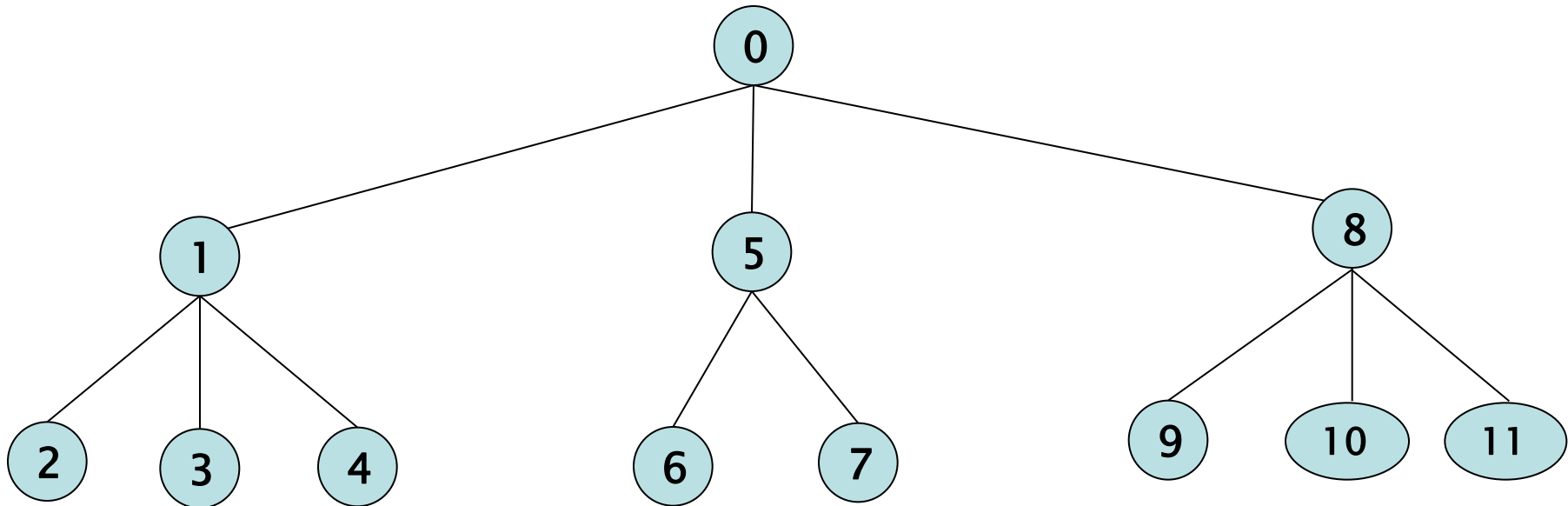


0  
1  
2  
4  
5  
3

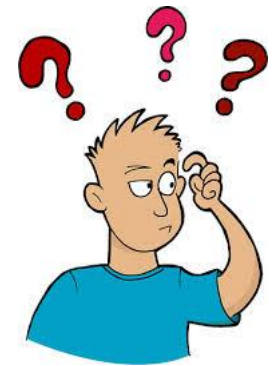




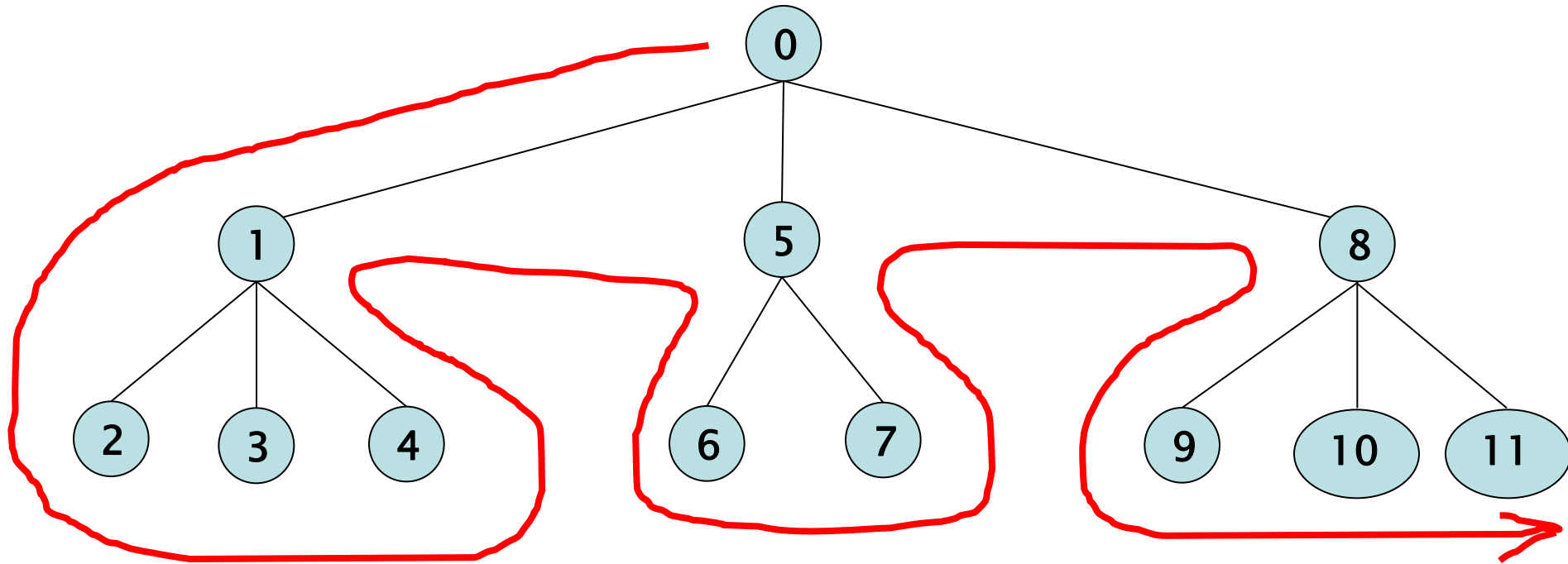
# Outro exemplo

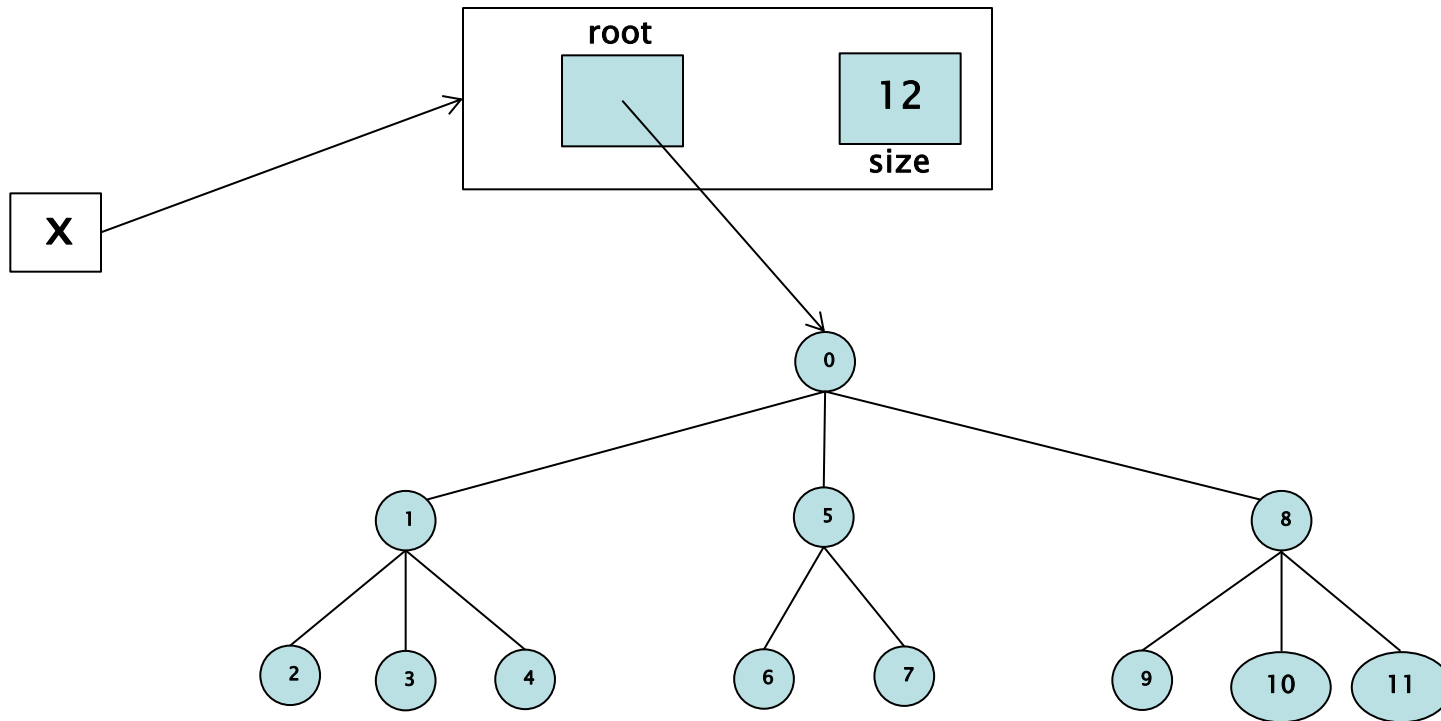


Qual o percurso preordem desta árvore ?



# Preorder





Qual o percurso preordem desta árvore ?



```
package 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);
            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;
```

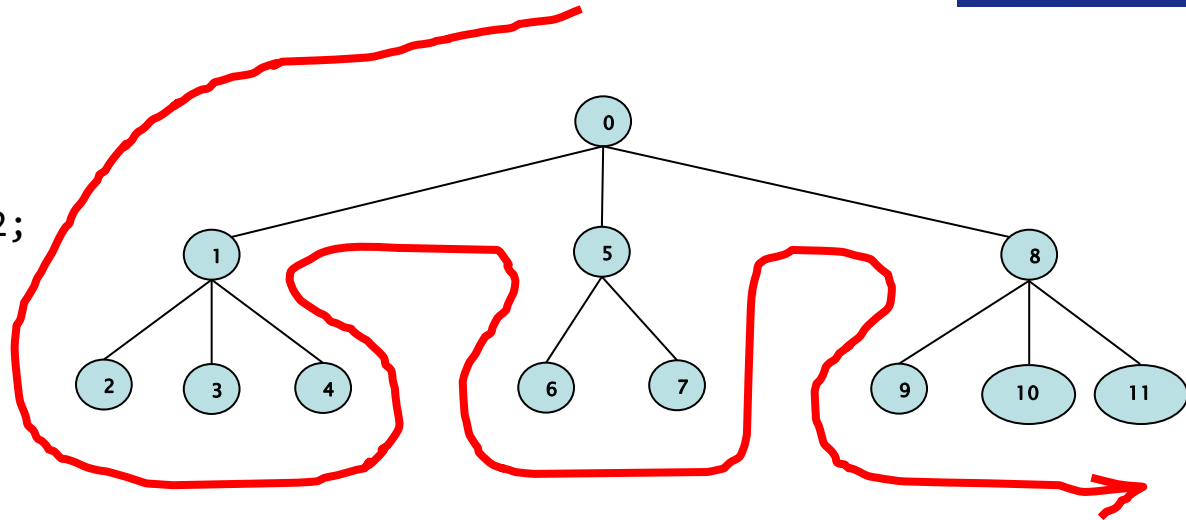
```
no_5.firstChild=no_6;  
no_6.next = no_7;  
no_6.parent = no_5;  
no_7.parent = no_5;
```

```
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();
```

```
}
```

```
}
```



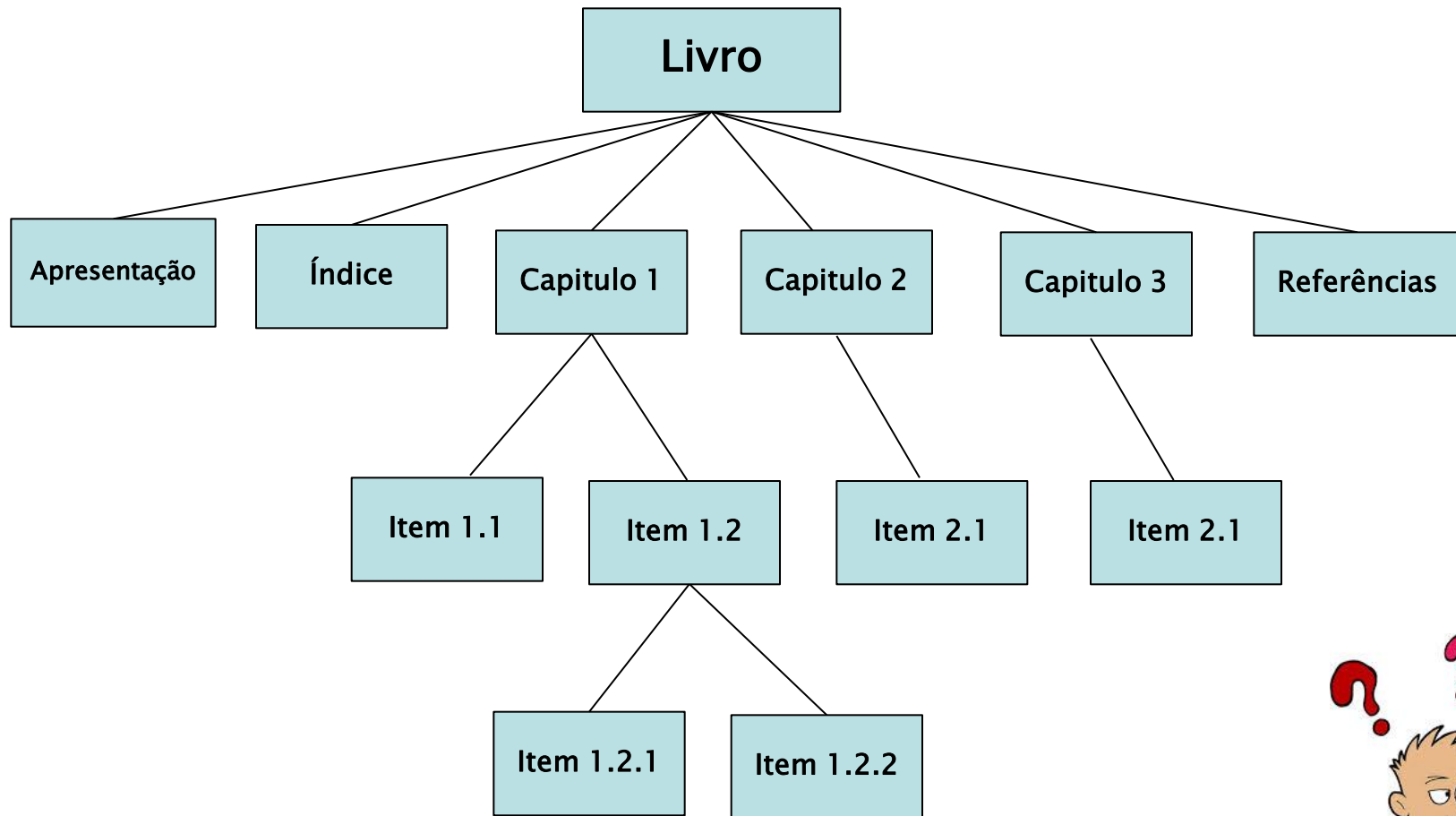
**Resposta: Preorder** →



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



# Exercício



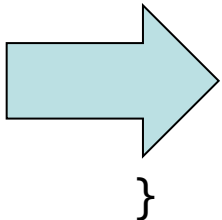
Qual o percurso preordem desta árvore ?



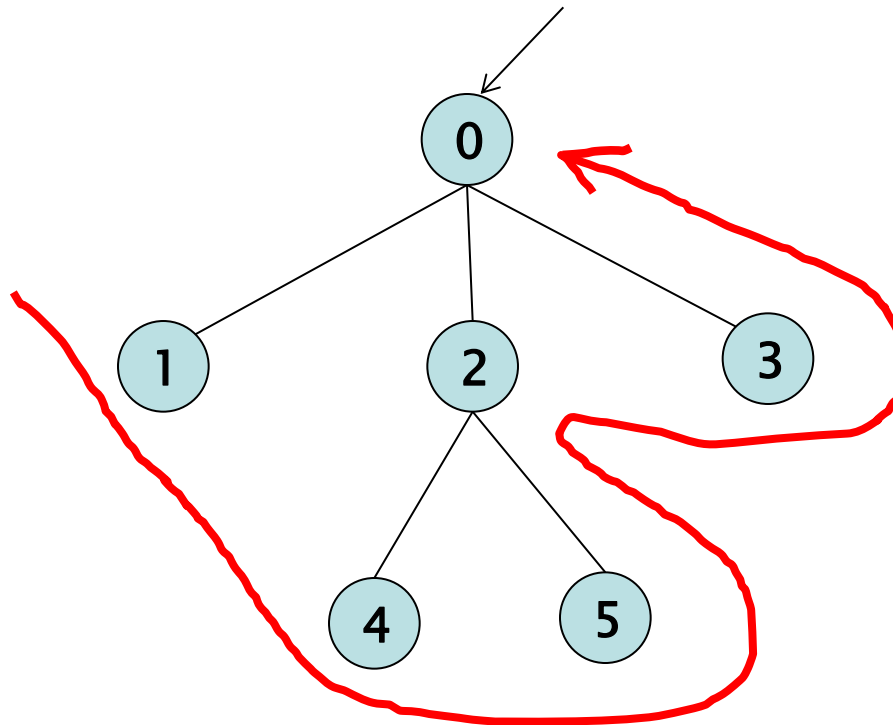
# Percurso – Posorder

- 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 posorder() {  
  
    Node_Tree trab = this.firstChild;  
  
    while (trab != null ) {  
        trab.postorder();  
        trab = trab.next;  
    }  
  
    System.out.println(this.item );  
}
```



# Percurso – Posorder



- Nós são visitados nesta ordem: **145230**
- 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);
```

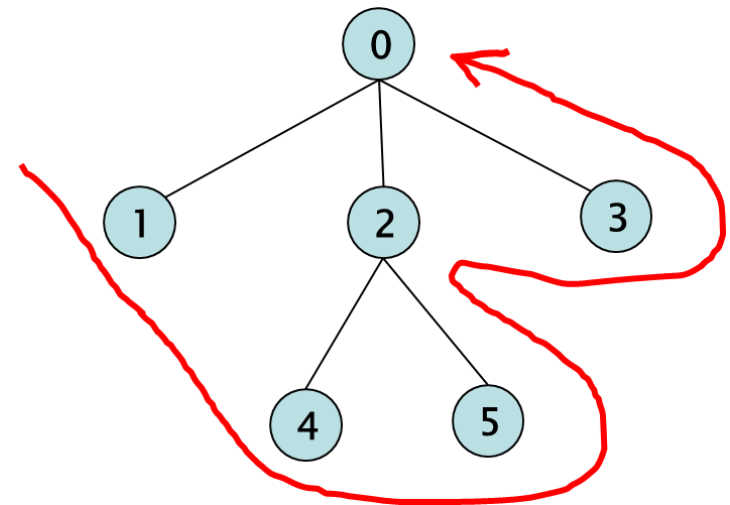
```
x.root.firstChild = no_1;
no_1.parent = x.root;
no_1.next = no_2;
```

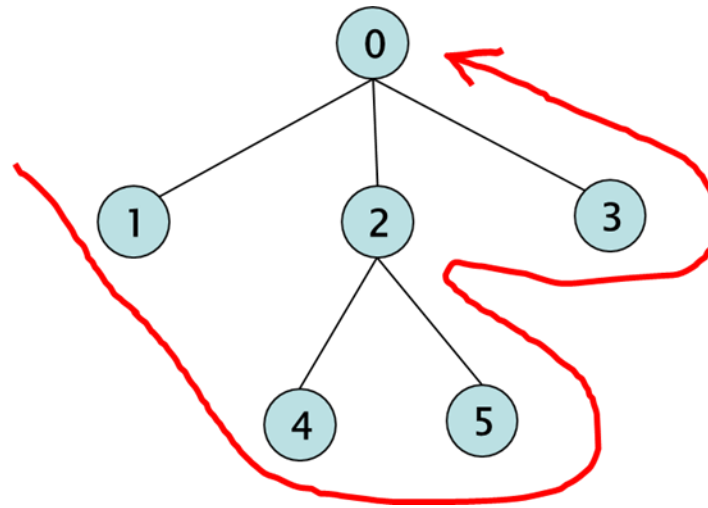
```
no_2.parent = x.root;
no_2.next = no_3;
```

```
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;
```



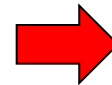


```
x.root.postorder();  
System.out.println ("");
```

```
}
```

```
}
```

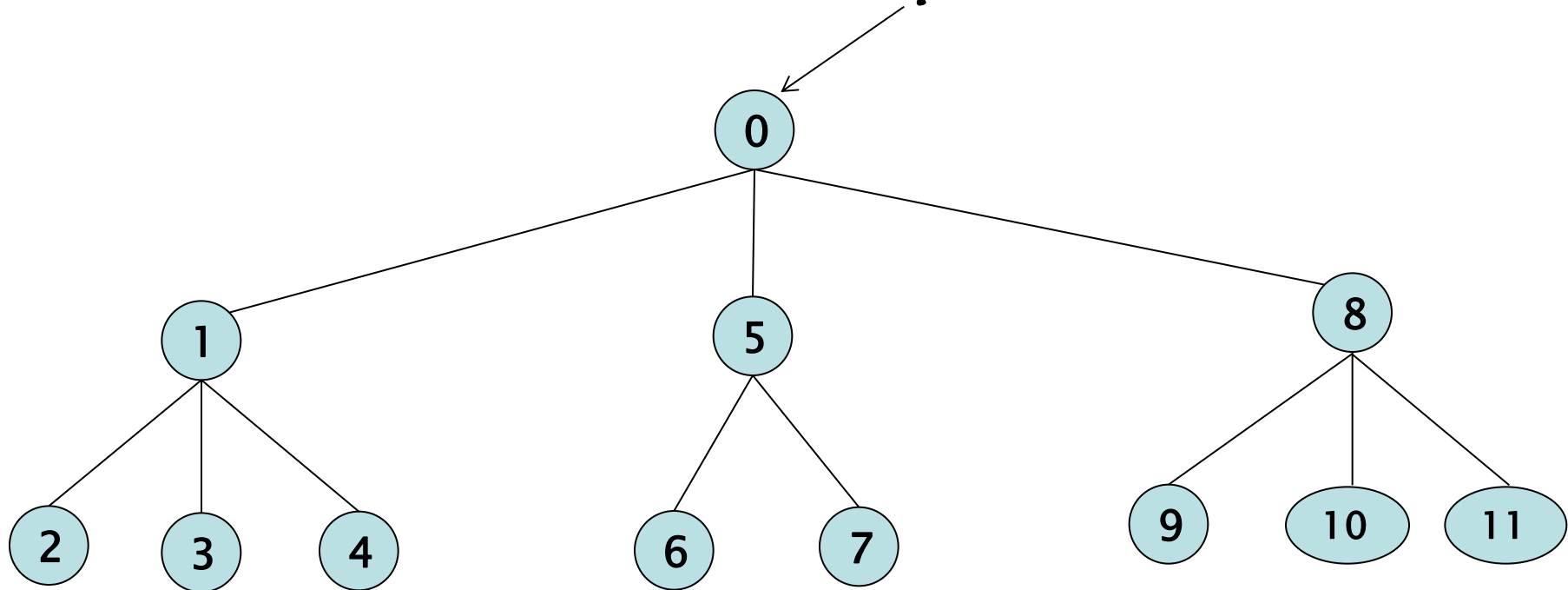
**Resposta: Posorder**



1  
4  
5  
2  
3  
0

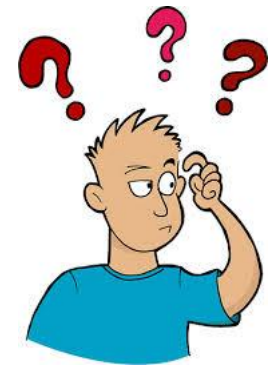


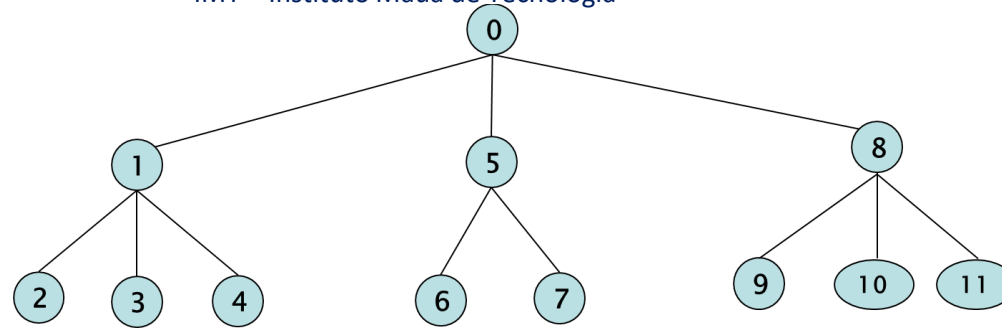
# Outro exemplo



Qual o percurso posordem desta árvore ?

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





```

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);
        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);

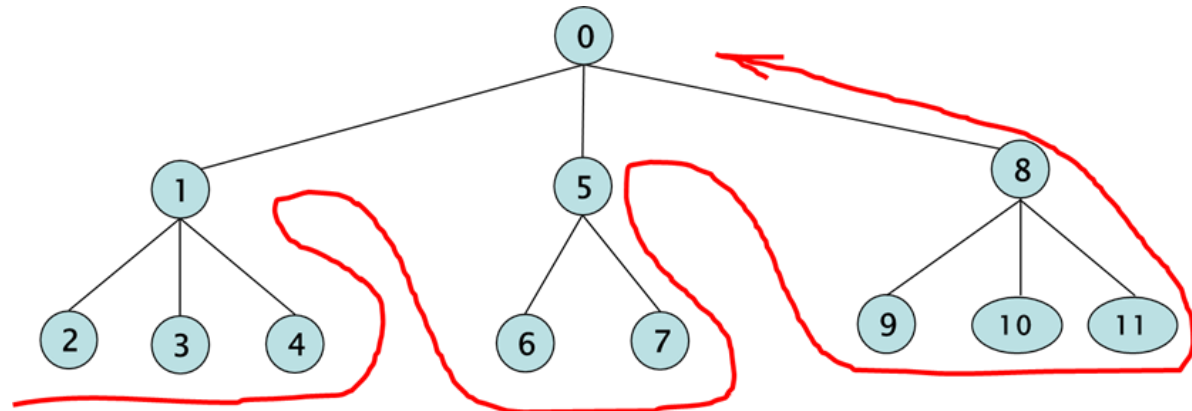
```



```
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;
```

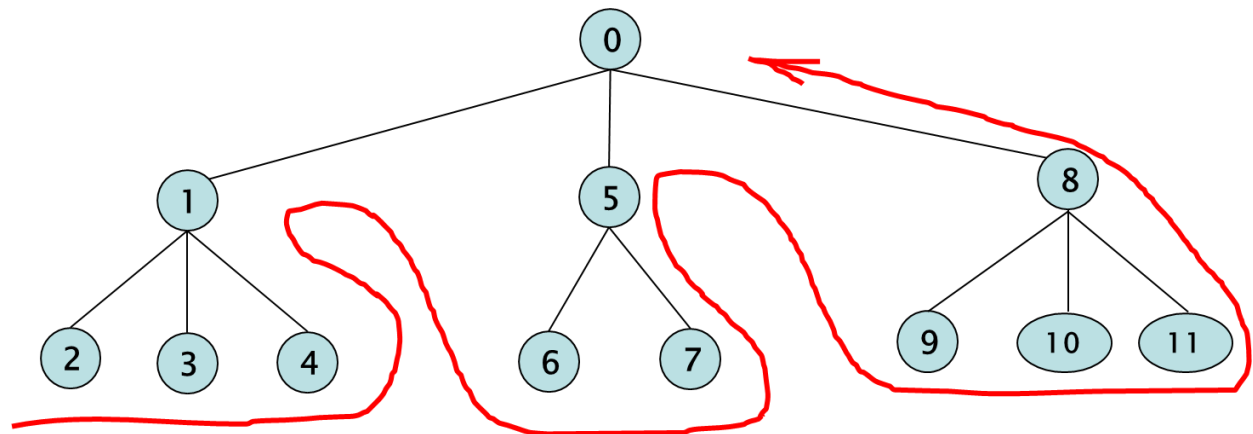


```
no_8.firstChild = no_9;  
no_9.next = no_10;  
no_10.next = no_11;  
no_9.parent = no_8;  
no_10.parent = no_8;  
no_11.parent = no_8;
```

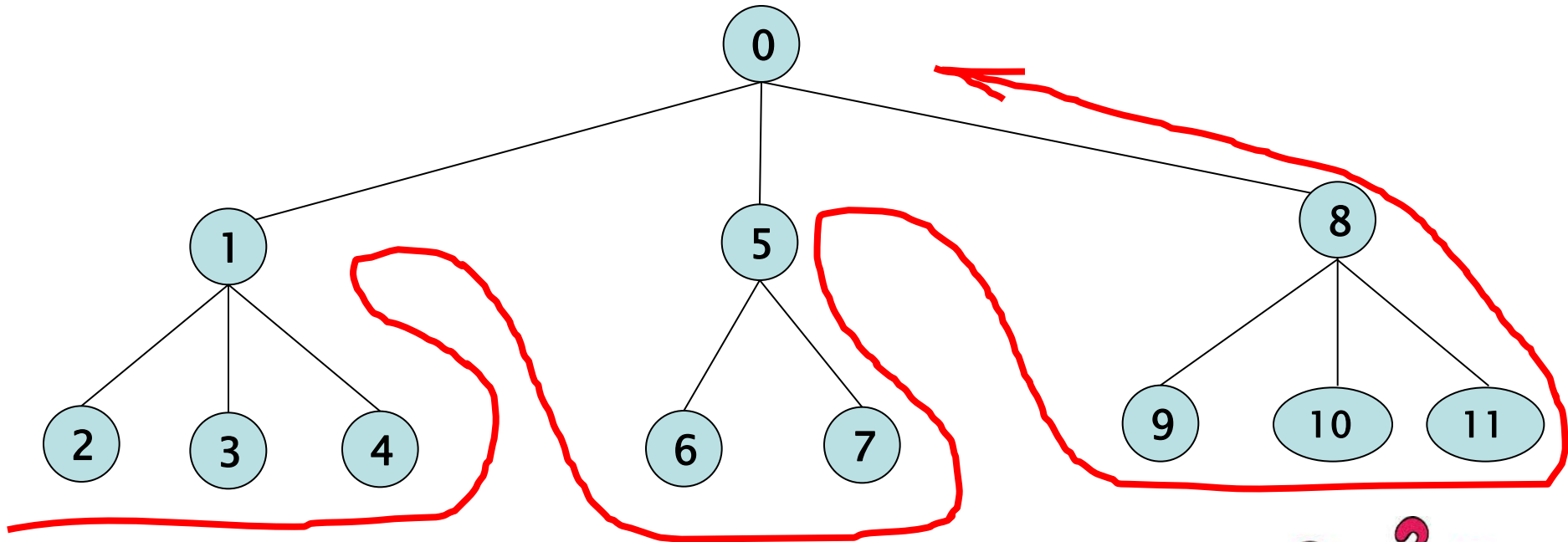
```
x.root.postorder();  
System.out.println ("");
```

```
}
```

```
}
```



# Outro exemplo



Qual o percurso **postordem** desta árvore ?

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



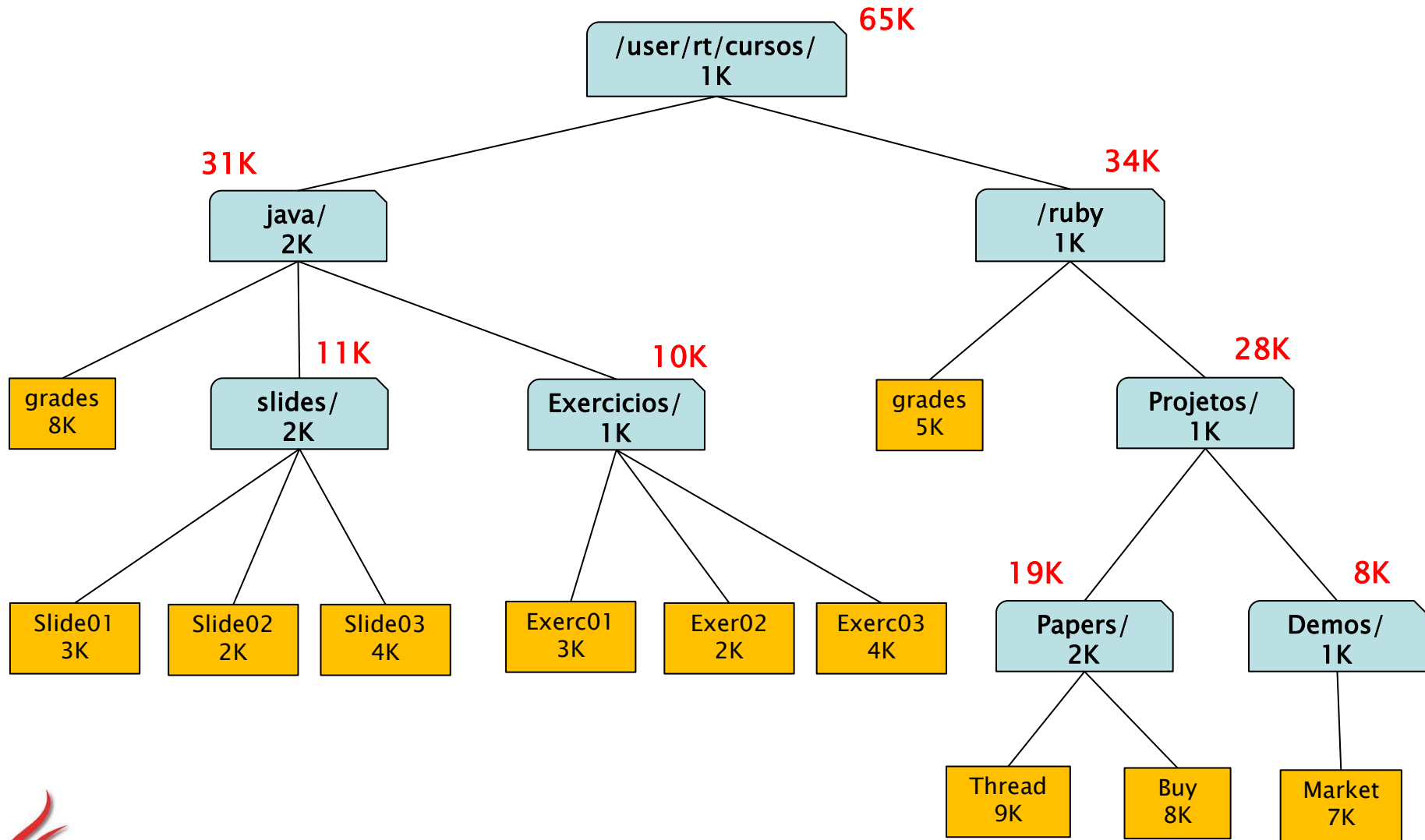
# Aplicação travessia postorder

- 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



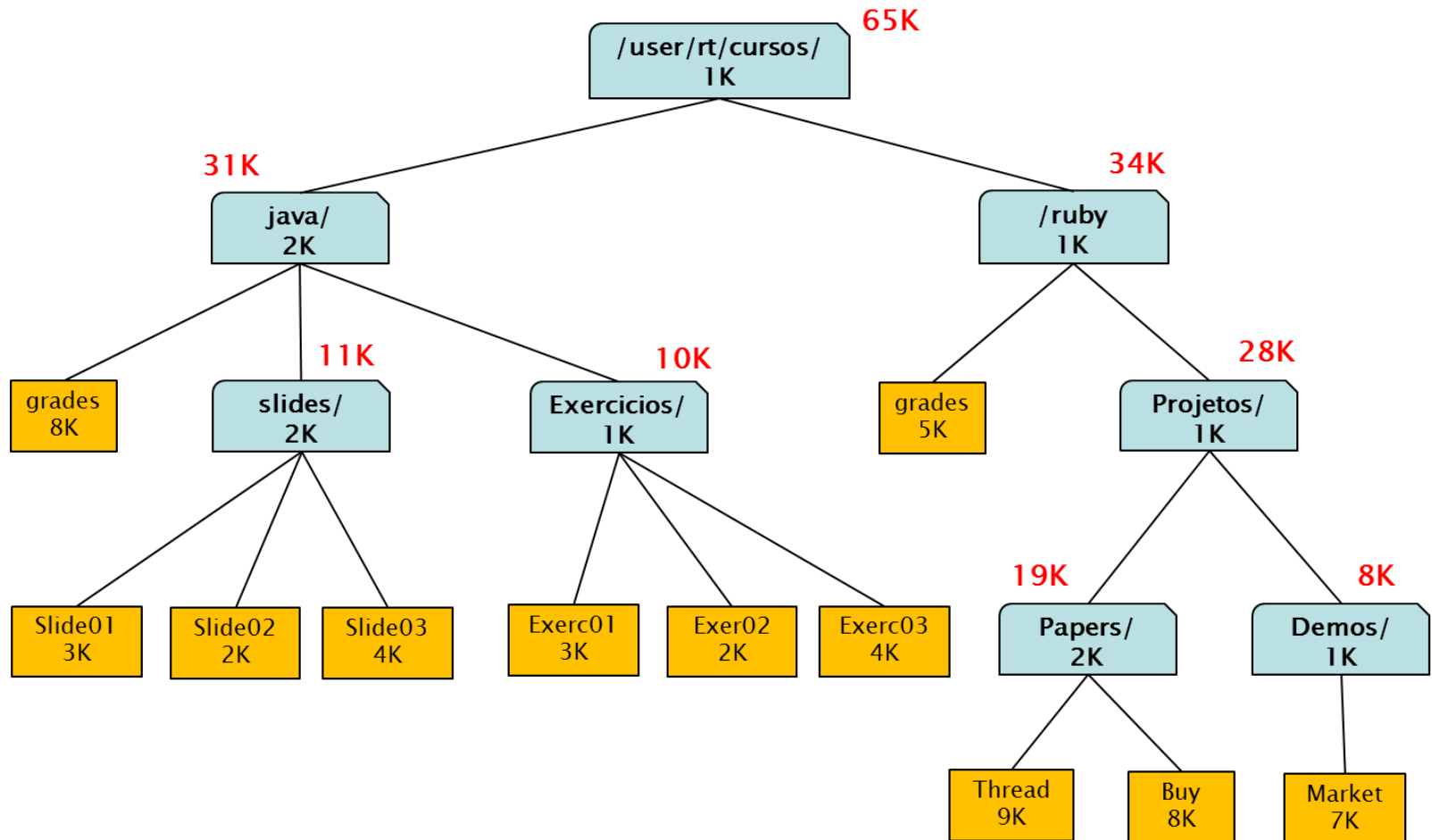


# Aplicação travessia postorder



# Exercício

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



FIM

