



Unidade 5 – Análise de Algoritmos com Estruturas de Dados Hierárquicas – Parte 1



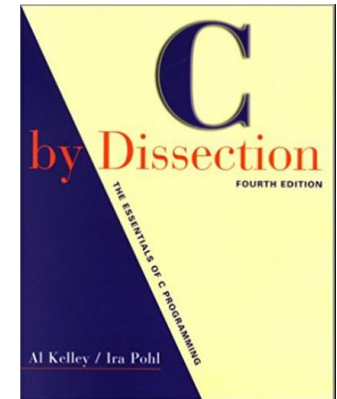
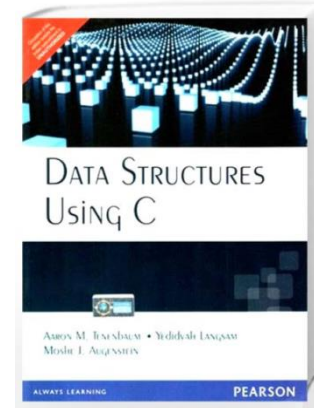
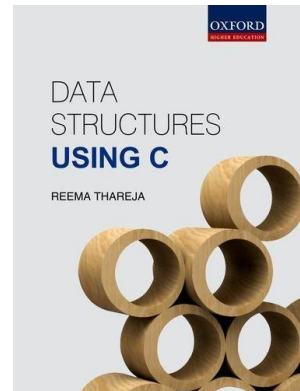
Prof. Aparecido V. de Freitas
Doutor em Engenharia
da Computação pela EPUSP
aparecidovfreitas@gmail.com





Bibliografia

- Algoritmos – Teoria e Prática – Cormen – Segunda Edição – Editora Campus, 2002
- Data Structures using C – Oxford University Press – 2014
- Data Structures Using C – A. Tenenbaum, M. Augensem, Y. Langsam, Pearson 1995
- C By Dissection – Kelley, Pohh – Third Edition – Addison Wesley





Árvores

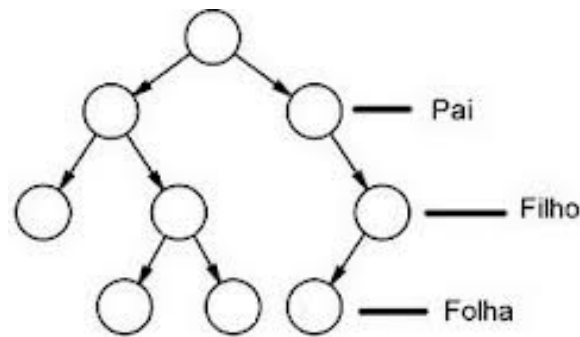
- Árvore é uma estrutura de dados não-linear.
- Tem uma importância muito grande na Computação, pois disponibiliza algoritmos muito mais rápidos que os encontrados nas estruturas lineares.
- Têm diversas aplicações: sistemas de arquivos, interfaces gráficas, banco de dados, etc.
- Os relacionamentos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

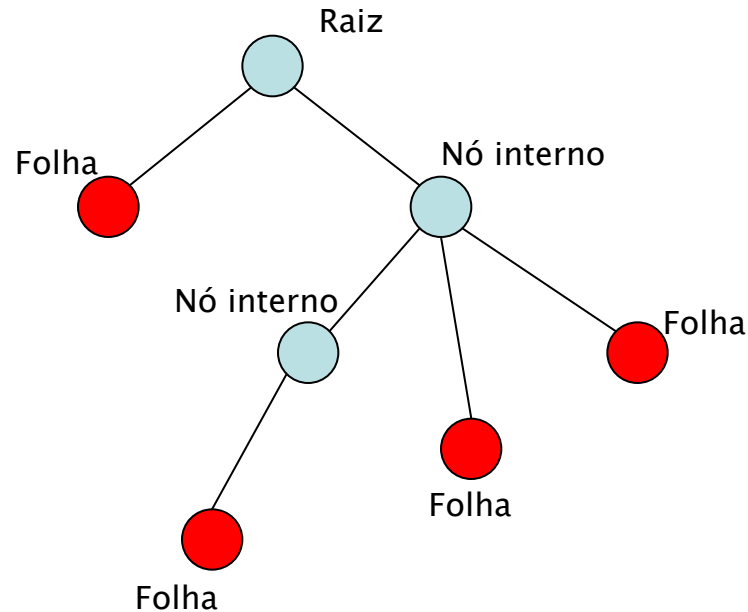
- Uma árvore **T** é um conjunto de nós que armazenam elementos em relacionamentos pai-filho com as seguintes propriedades:
 - Se **T** não é vazia, ela tem um nó especial chamado **raiz** de **T**, que não tem pai.
 - Cada nó v de **T** diferente da raiz tem um único nó pai w ;
- Uma árvore pode não 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**.





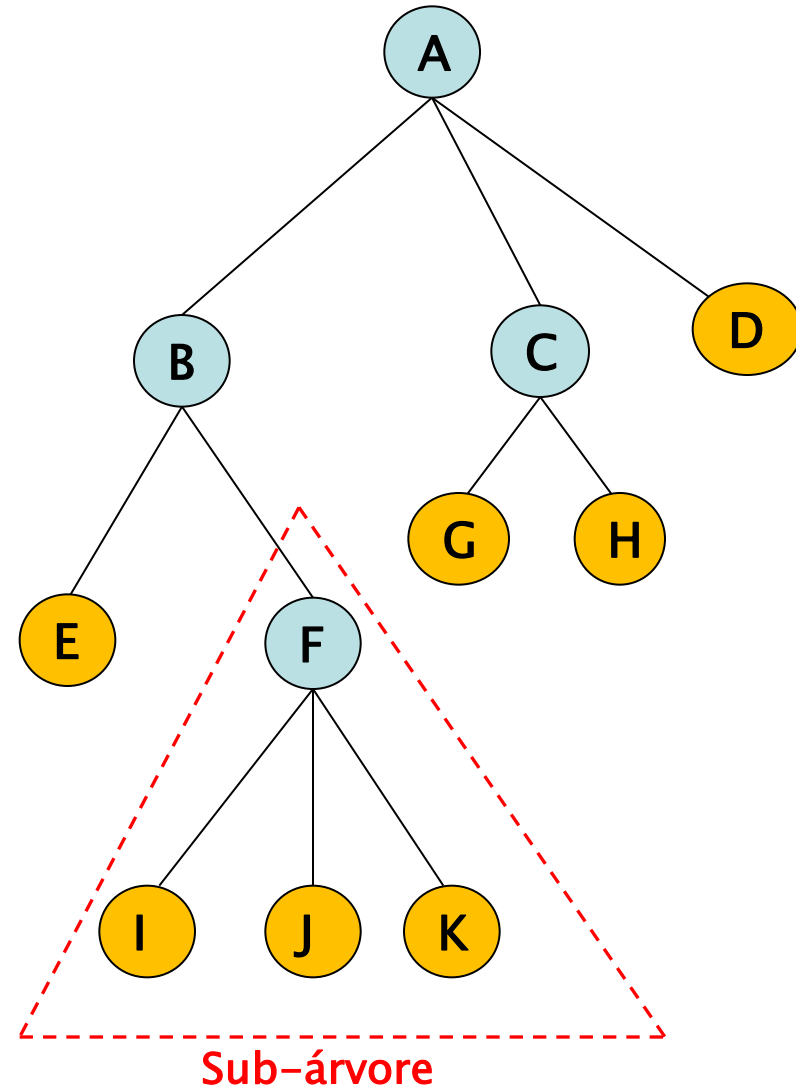
Outros relacionamentos

- 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.



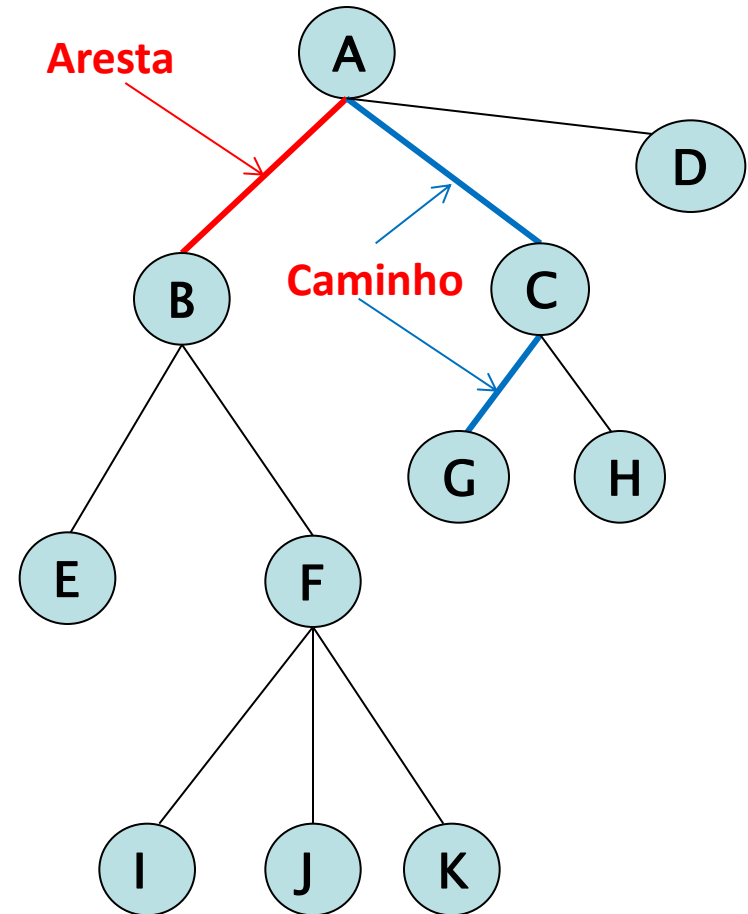
Definições

- ⊕ **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.



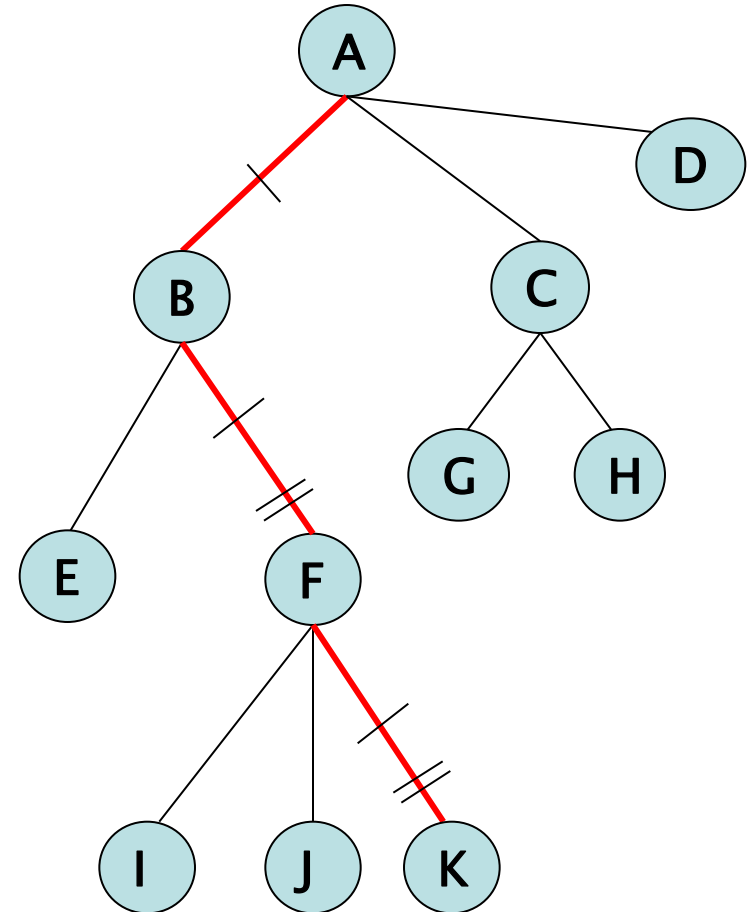
Definições

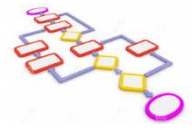
- ⊕ **Aresta:** é um par de nós (u,v) tal que u é pai de v . (A,B)
- ⊕ **Caminho:** é uma sequência de nós tais que quaisquer dois nós consecutivos da sequência sejam arestas. $((A,C),(C,G))$
- ⊕ **Tamanho de um caminho:** # de arestas em um caminho. (Tamanho do caminho $((A,C),(C,G)) = 2$).



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**





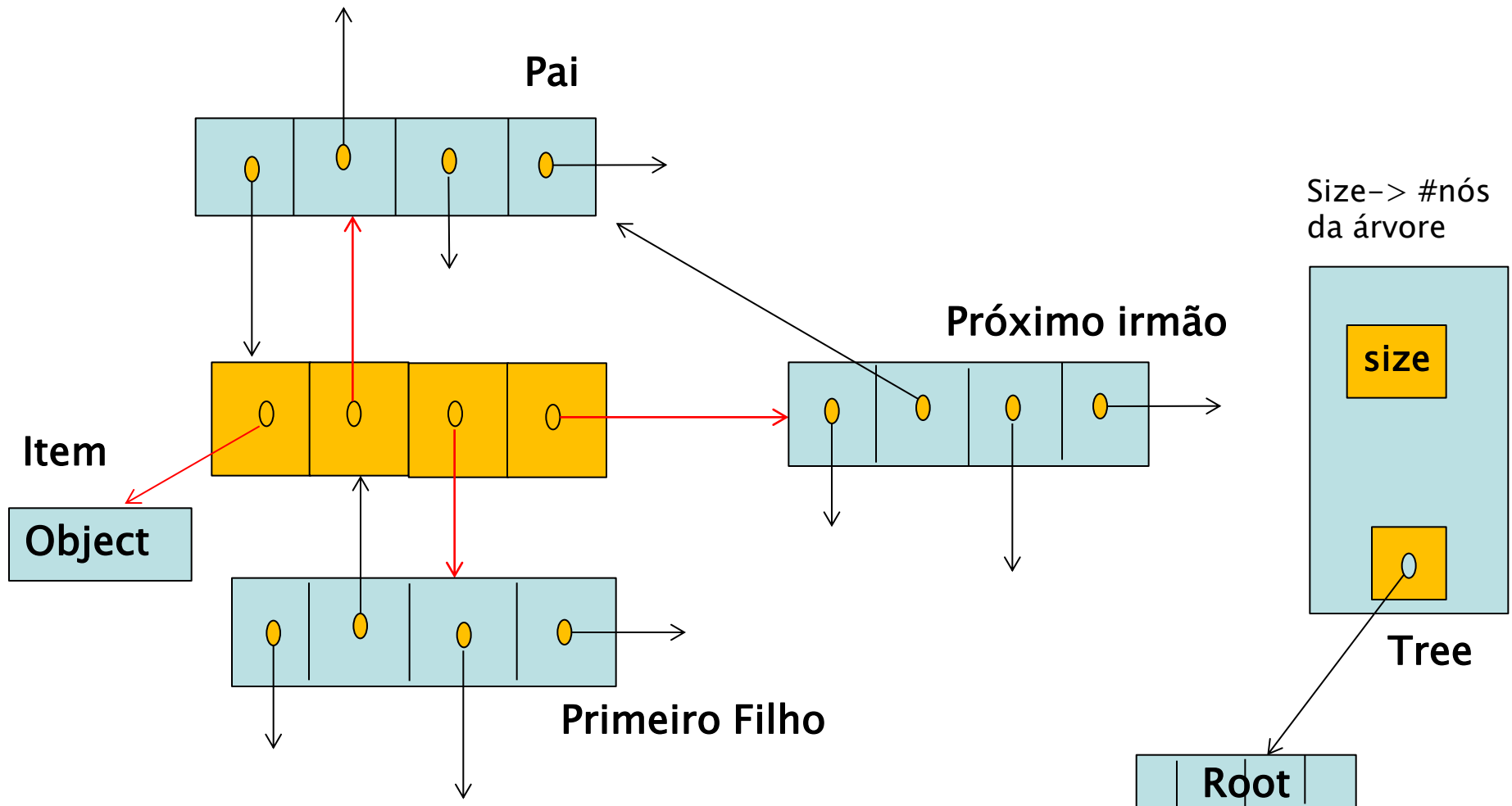
Tipos de Árvores

- ⊕ Árvores Gerais (Genéricas)
- ⊕ Árvores Binárias
- ⊕ Árvores Binárias de Pesquisa
- ⊕ Árvores de Expressão



Representando Nó de Árvores Genéricas

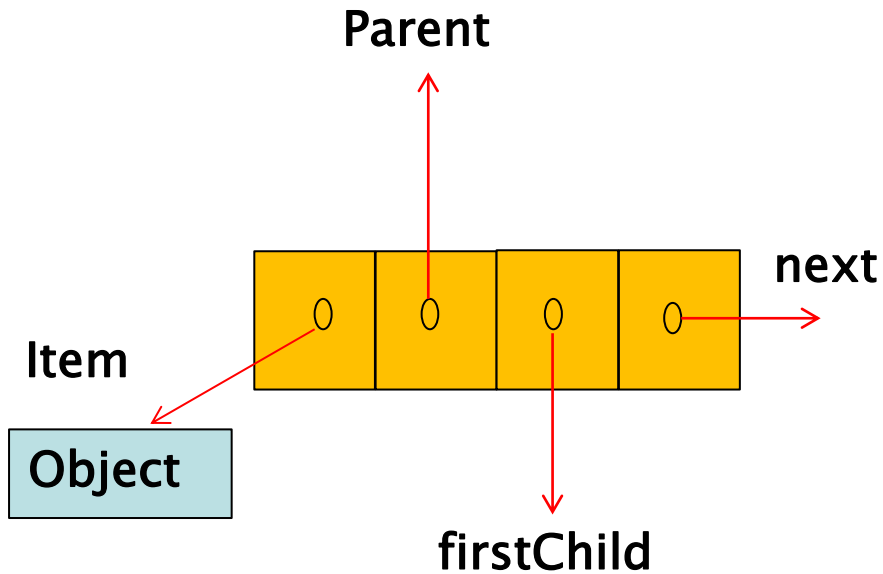
- Cada nó tem quatro referências: item, pai, primeiro filho e próximo irmão.





Representando Nó de Árvores

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



```
struct node {  
  
    int data;  
  
    struct node * parent;  
  
    struct node * firstchild;  
  
    struct node * next;  
  
}
```



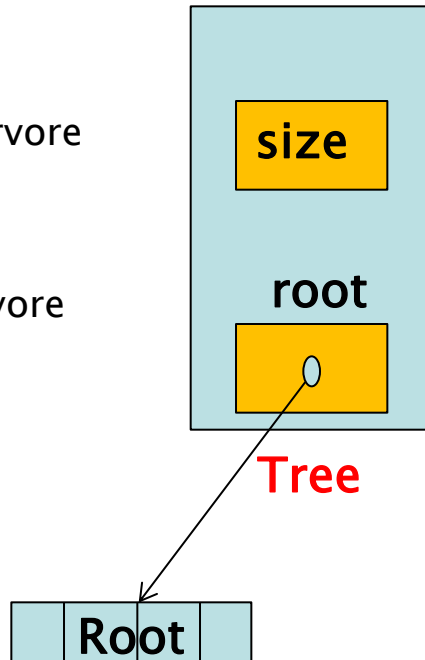


Representando Árvores

- 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

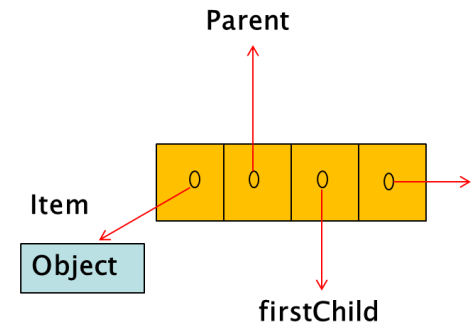


```
struct Tree {  
    int size;  
    struct node * root;  
}
```

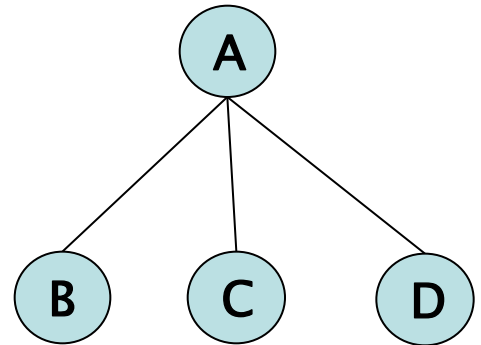
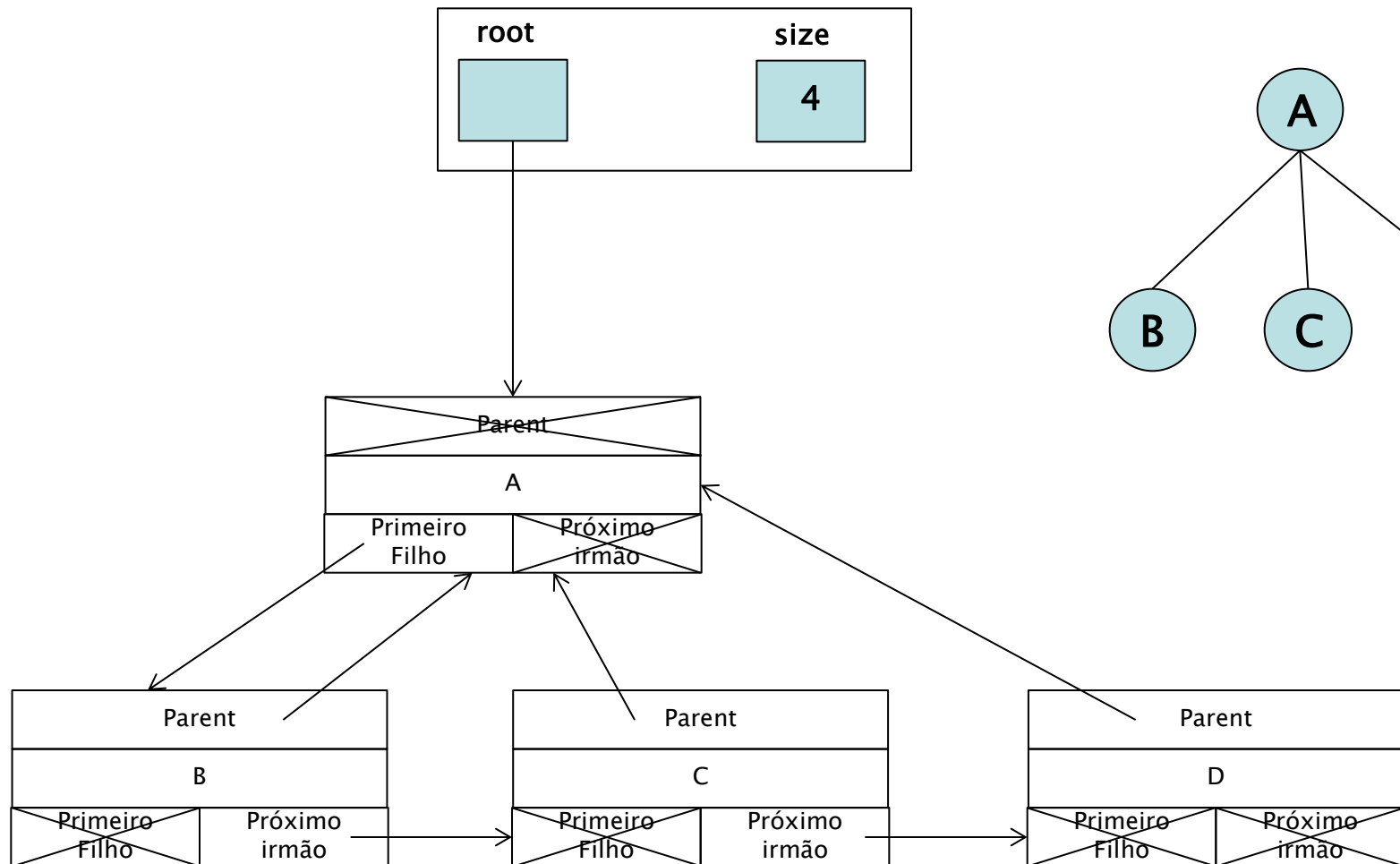




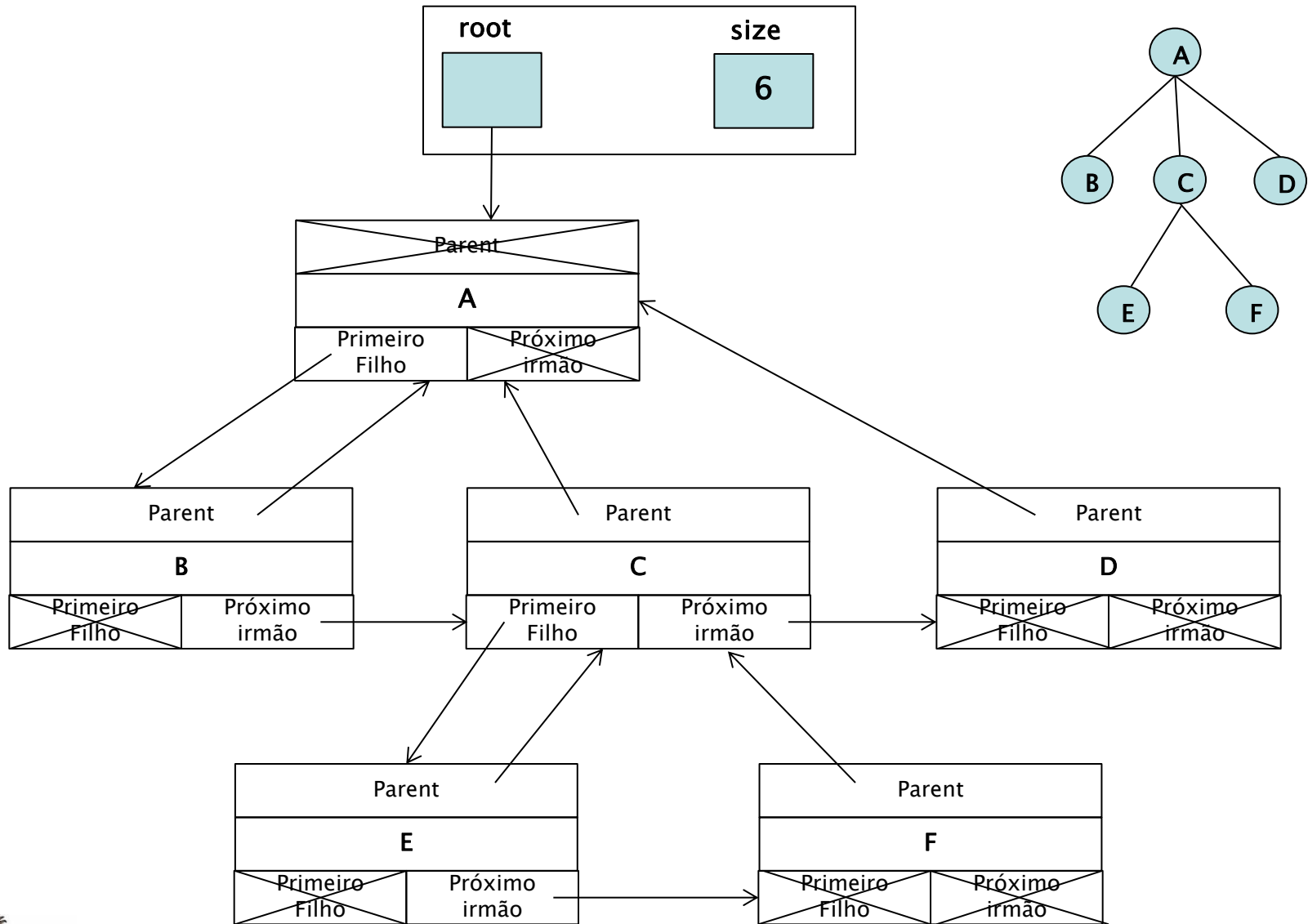
Representando Nó da árvore



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



Node_Tree

```
#include <stdio.h>
#include <stdlib.h>

struct node * node_parent(struct node * );

struct node {
    int data;

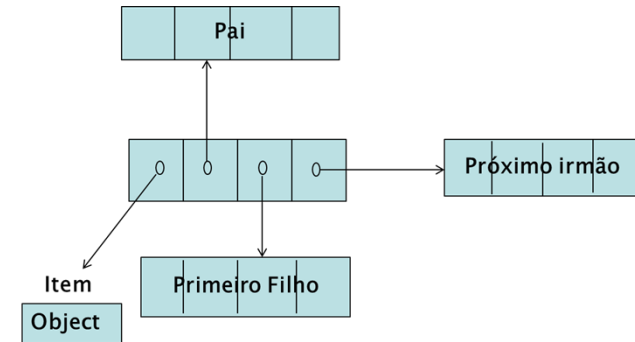
    struct node * parent;

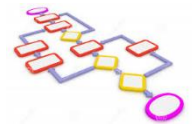
    struct node * firstchild;

    struct node * next;
};

struct tree {
    int size;

    struct node * root;
};
```





node_parent(v): retorna o pai de v

imprimeParent(): imprime o dado armazenado no pai

```
struct node * node_parent (struct node * ponteiro ) {  
  
    if (ponteiro == NULL)  
        return NULL;  
  
    return  ponteiro->parent;  
}
```

```
void imprimeParent ( struct node * ponteiro) {  
  
    if (ponteiro -> parent == NULL)  
        printf ("\nNode root. Este node nao tem pai.....");  
    else  
        printf ("\nDado armazenado no node parent:  %d " , ponteiro -> parent -> data);  
}
```





imprimeFilhos(): imprime os filhos do nó

```
void imprimeFilhos(struct node * ponteiro) {  
  
    if (ponteiro -> firstchild == NULL )  
        printf (" Este node nao tem filhos.....");  
    else {  
  
        struct node * trab = ponteiro -> firstchild;  
  
        while ( trab != NULL ) {  
  
            printf ("\t %d", trab -> data );  
            trab = trab -> next;  
        }  
    }  
}
```





isInternal(): testa se nó é node interno

```
enum boolean isInternal( struct node * ponteiro) {  
  
    if (ponteiro -> firstchild != NULL ) {  
        printf ("\nsim, o node eh interno....");  
        return false;  
    }  
    printf ("\no node nao eh interno....");  
    return true;  
}
```





dept(): retorna o número de ancestrais do nó

```
int dept(struct node * ponteiro) {  
  
    if (ponteiro -> parent == NULL )  
        return 0;  
    return ( 1 + dept( ponteiro -> parent));  
}
```





height(): retorna a altura do nó

```
int height(struct node * ponteiro) {  
  
    if (ponteiro -> firstchild == NULL)  
        return 0;  
  
    int h = 0;  
  
    struct node * trab = ponteiro -> firstchild;  
  
    while (trab -> next != NULL) {  
  
        h = maximo(h, height(trab -> next));  
        trab = trab -> next;  
    }  
  
    return (h + 1) ;  
  
}
```



Estrutura Tree

```
#include <stdio.h>
#include <stdlib.h>
```

```
struct node * node_parent(struct node * );
```

```
struct node {
    int data;

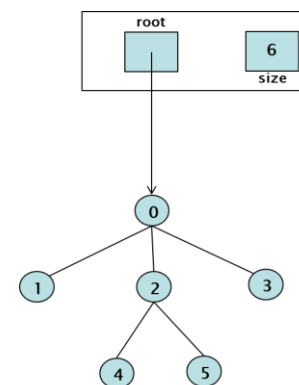
    struct node * parent;

    struct node * firstchild;

    struct node * next;
};

struct tree {
    int size;

    struct node * root;
};
```





ret_Root(): retorna o node root da árvore.

```
struct node * ret_Root () {  
  
    return root;  
  
}
```





sizeTree(): retorna o número de nós da árvore

```
int sizeTree() {  
  
    return size;  
  
}
```





isEmpty(): testa se a árvore é vazia

```
enum boolean isEmpty() {  
  
    if (size == 0)  
        return true;  
    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

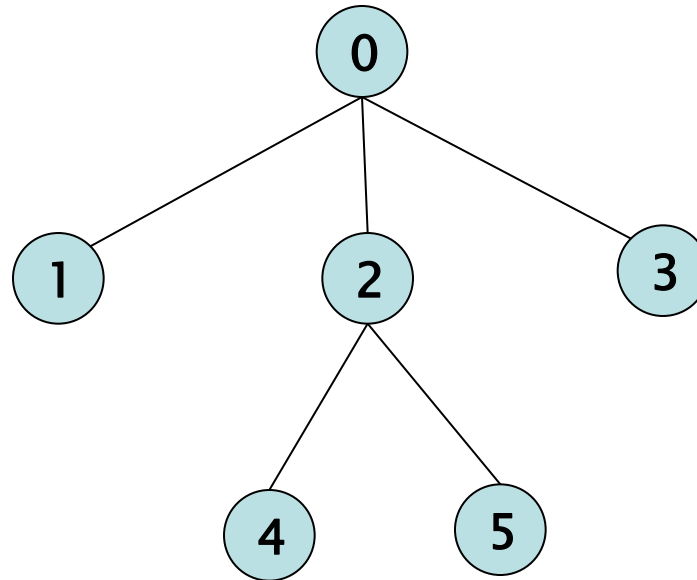
- Na travessia **preorder** de uma árvore T, a raiz de T é visitada em primeiro lugar e em seguida as sub-árvores são visitadas recursivamente.

```
void preorder(struct node * ponteiro ) {  
  
    printf ("\t %d" , ponteiro -> data );  
  
    struct node * trab = ponteiro -> firstchild;  
  
    while (trab != NULL) {  
  
        preorder (trab);  
        trab = trab ->next;  
    }  
}
```





Exercício

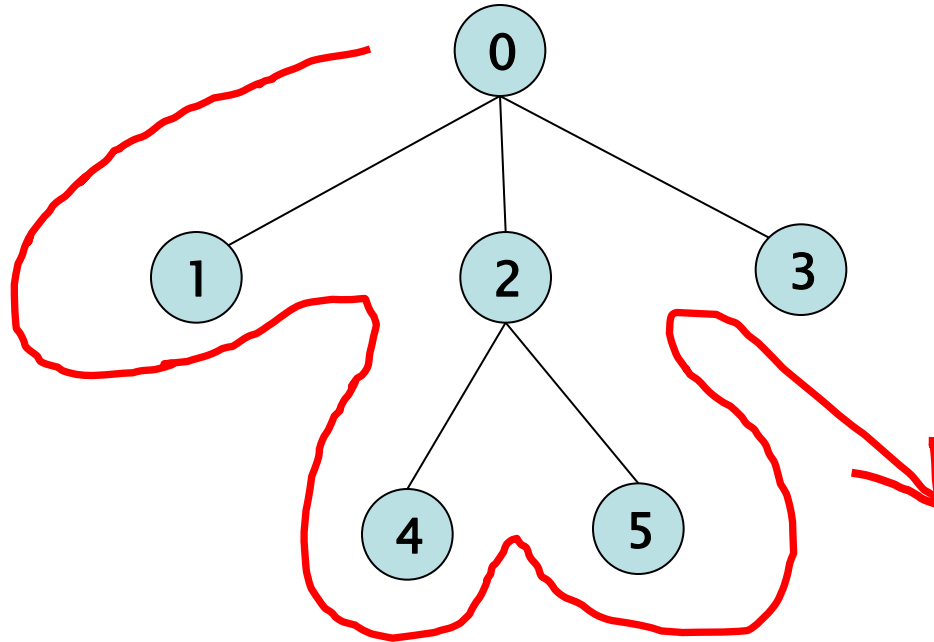


- Imprimir os nós da árvore com o uso da travessia preorder.





Percurso – Preorder

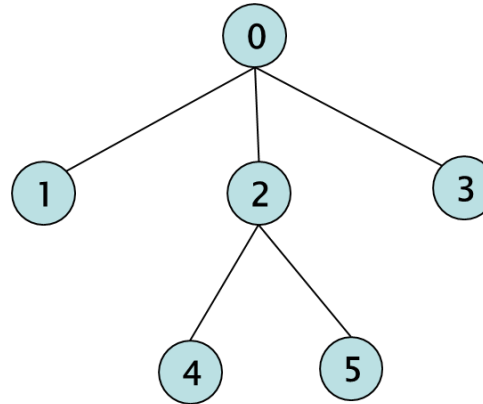


- Nós são visitados nesta ordem: 0 1 2 4 5 3
- Cada nó é visitado somente uma vez, assim o percurso preorder gasta tempo **$O(n)$** , onde n é o total de nós da árvore.





Solução



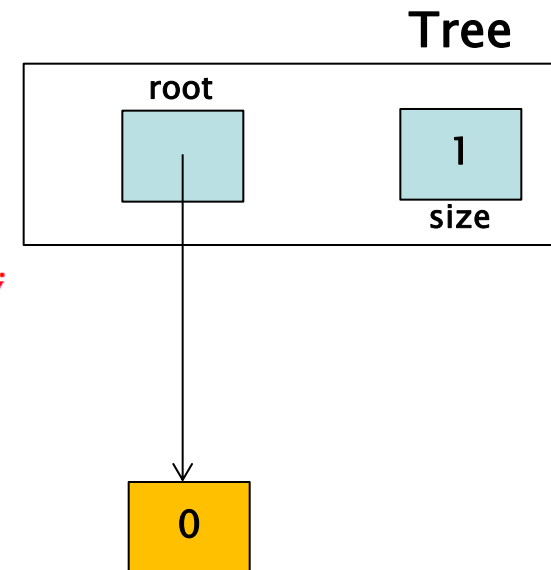
1. Construir a estrutura de dados que corresponde à árvore (estrutura de controle)
2. Criar o nó root e vinculá-lo à árvore
3. Construir os nós que compõem a árvore
4. Estabelecer os relacionamentos hierárquicos entre os nós
5. Aplicar o algoritmo de preorder na raiz da árvore.





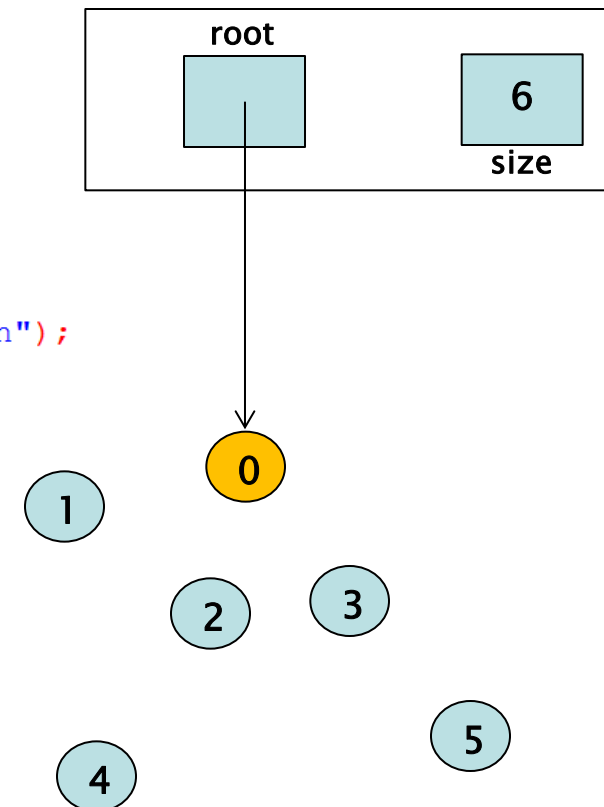
Criação do nó root

```
void insert_root( int valor) {  
  
    struct node * novo_node = cria_node(valor);  
  
    root = novo_node;  
  
    size = 1;  
}  
  
struct node * cria_node (int valor) {  
  
    struct node * new_node ;  
  
    new_node = (struct node *) malloc (sizeof (struct node));  
  
    new_node -> firstchild = NULL;  
    new_node -> next = NULL;  
    new_node -> parent= NULL;  
    new_node -> data = valor;  
  
    return new_node;  
}
```



Construção dos nós que compõem a árvore

```
int main() {  
  
    printf(" **** Implementacao de arvores genericas....\n");  
    insert_root(0);  
  
    struct node * no_1 = cria_node(1);  
    struct node * no_2 = cria_node(2);  
    struct node * no_3 = cria_node(3);  
    struct node * no_4 = cria_node(4);  
    struct node * no_5 = cria_node(5);  
}
```



Relacionamentos hierárquicos entre os nós

```
root -> firstchild = no_1;
```

```
no_1 -> parent = root;
```

```
no_1 -> next = no_2;
```

```
no_2 -> parent = root;
```

```
no_2 -> next = no_3;
```

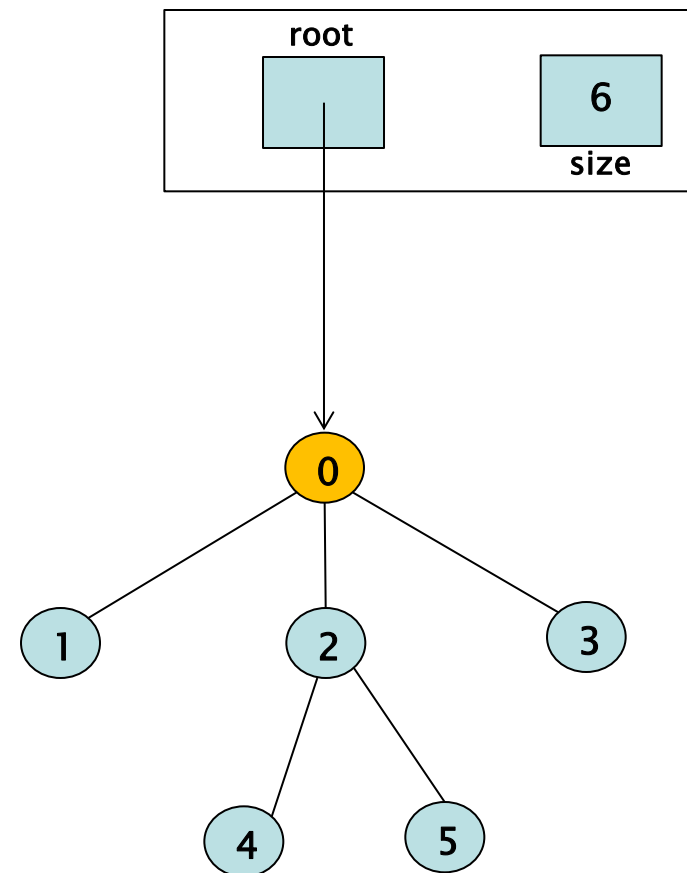
```
no_3 -> parent = root;
```

```
no_2 -> firstchild = no_4;
```

```
no_4 -> parent = no_2;
```

```
no_4 -> next = no_5;
```

```
no_5 -> parent = no_2;
```



Relacionamentos hierárquicos entre os nós

```
root -> firstchild = no_1;
```

```
no_1 -> parent = root;
```

```
no_1 -> next = no_2;
```

```
no_2 -> parent = root;
```

```
no_2 -> next = no_3;
```

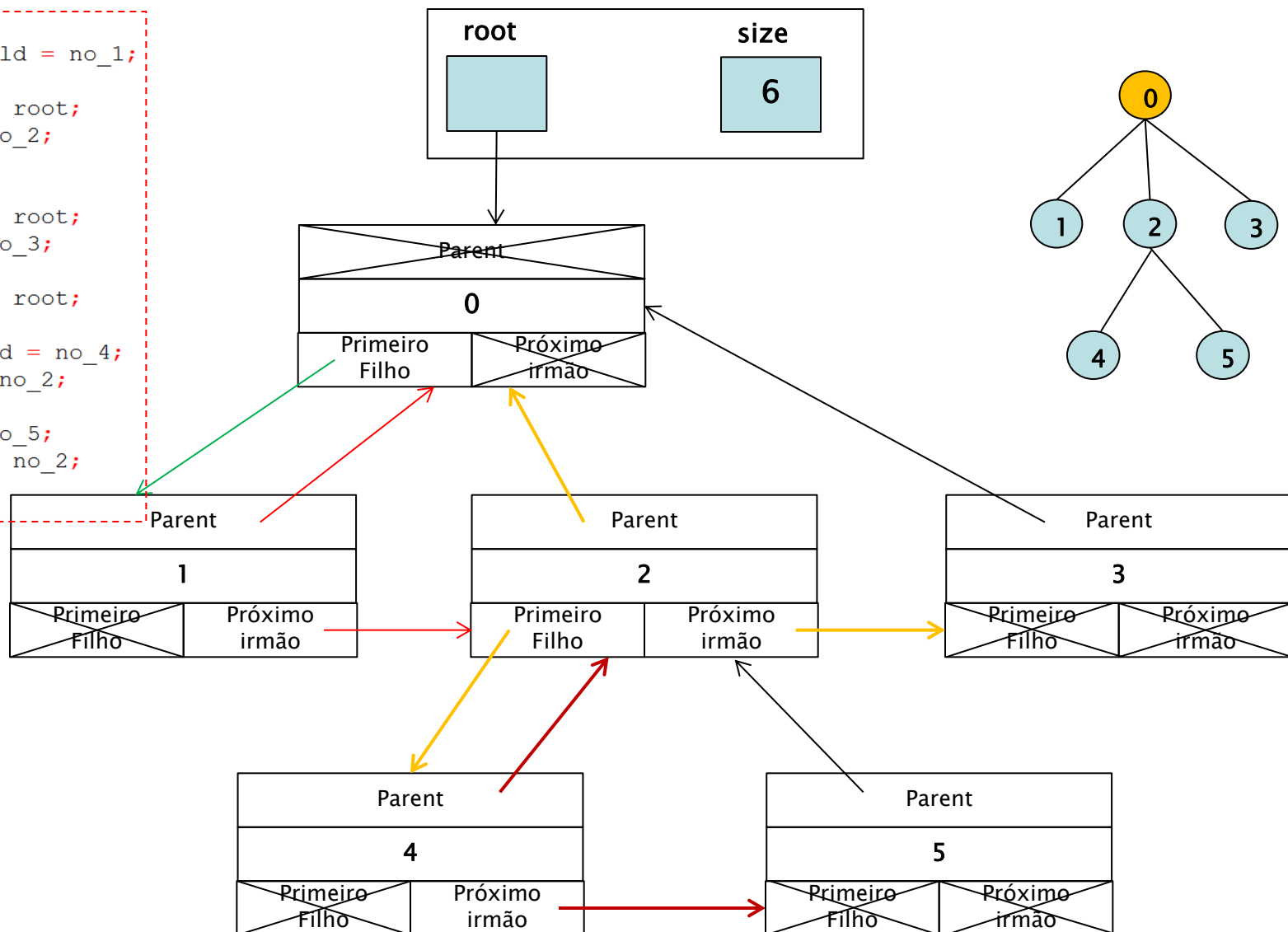
```
no_3 -> parent = root;
```

```
no_2 -> firstchild = no_4;
```

```
no_4 -> parent = no_2;
```

```
no_4 -> next = no_5;
```

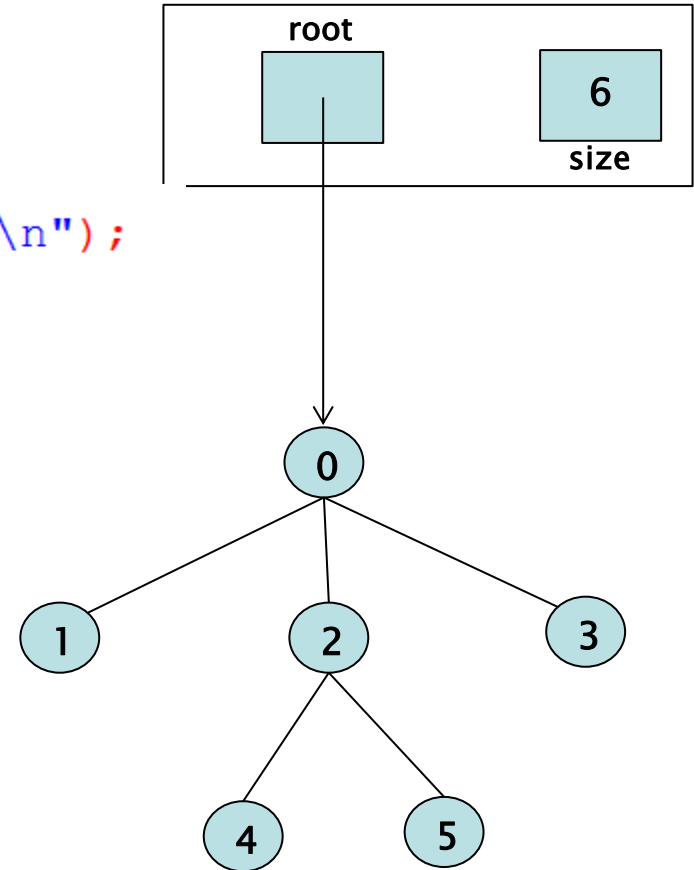
```
no_5 -> parent = no_2;
```





Algoritmo de preorder na raiz da árvore.

```
printf ("\n ***** preorder *****\n\n");  
preorder(root);
```



```
int main() {
```

```
    printf(" **** Implementacao de arvores genericas...\n");
    insert_root(0);
```

```
    struct node * no_1 = cria_node(1);
    struct node * no_2 = cria_node(2);
    struct node * no_3 = cria_node(3);
    struct node * no_4 = cria_node(4);
    struct node * no_5 = cria_node(5);
```

```
    root -> firstchild = no_1;
```

```
    no_1 -> parent = root;
    no_1 -> next = no_2;
```

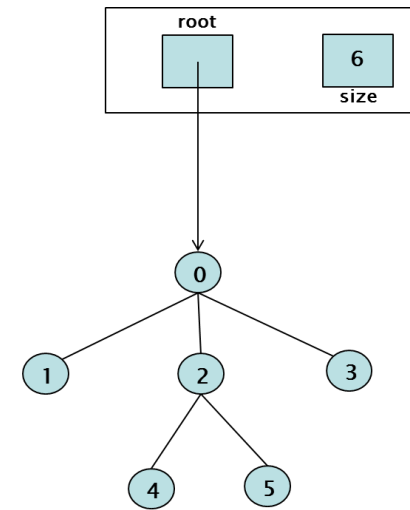
```
    no_2 -> parent = root;
    no_2 -> next = no_3;
```

```
    no_3 -> parent = root;
```

```
    no_2 -> firstchild = no_4;
    no_4 -> parent = no_2;
```

```
    no_4 -> next = no_5;
    no_5 -> parent = no_2;
```

```
    printf ("\n ***** preorder *****\n\n");
    preorder(root);
```



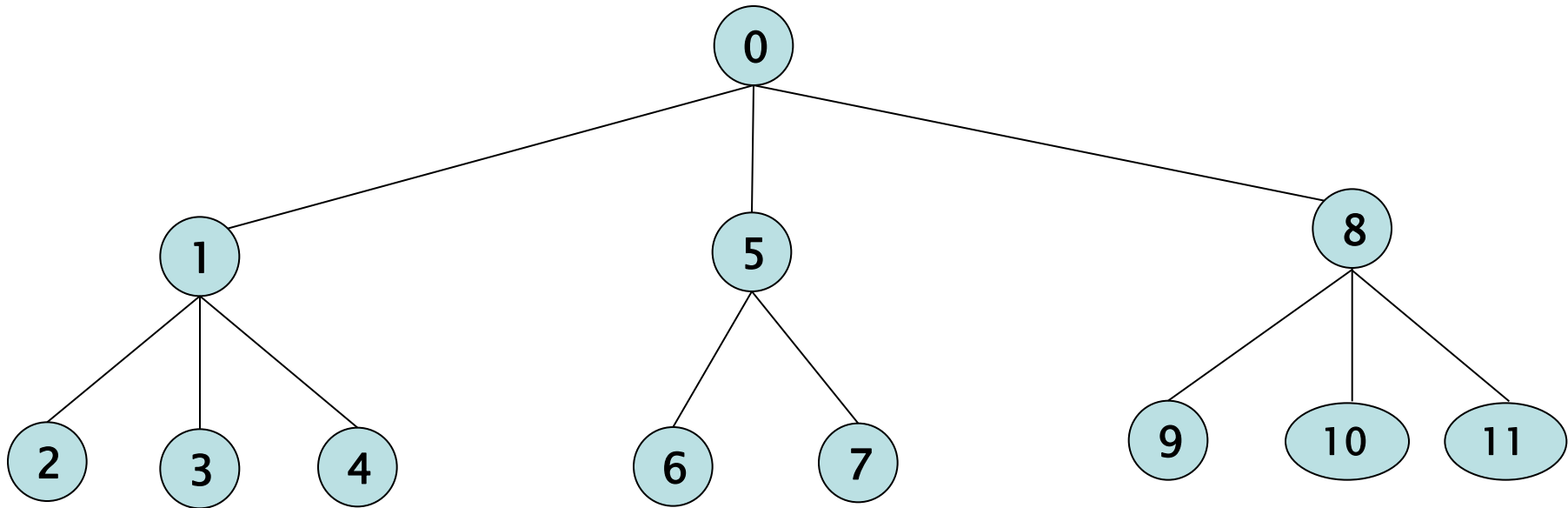
Resposta do programa:



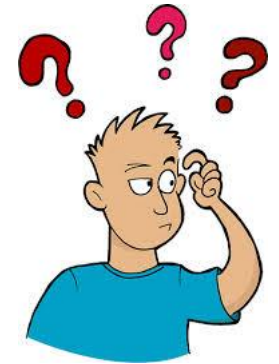
0
1
2
4
5
3



Outro exemplo

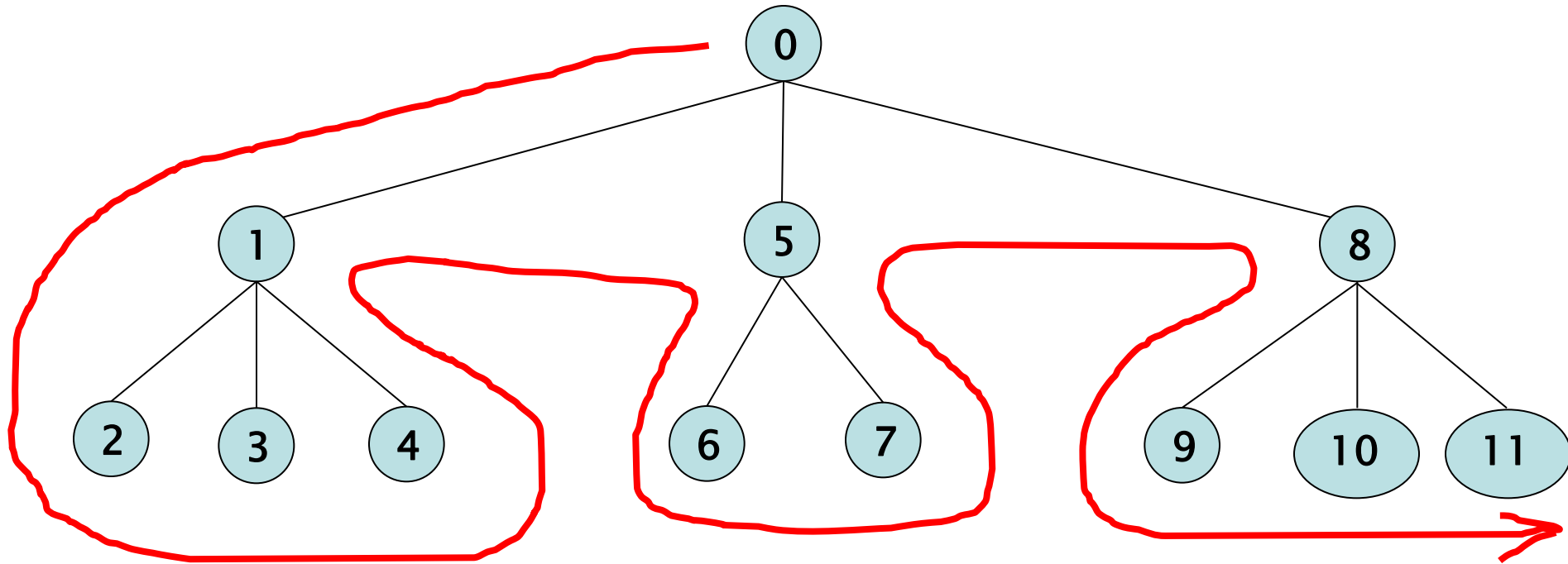


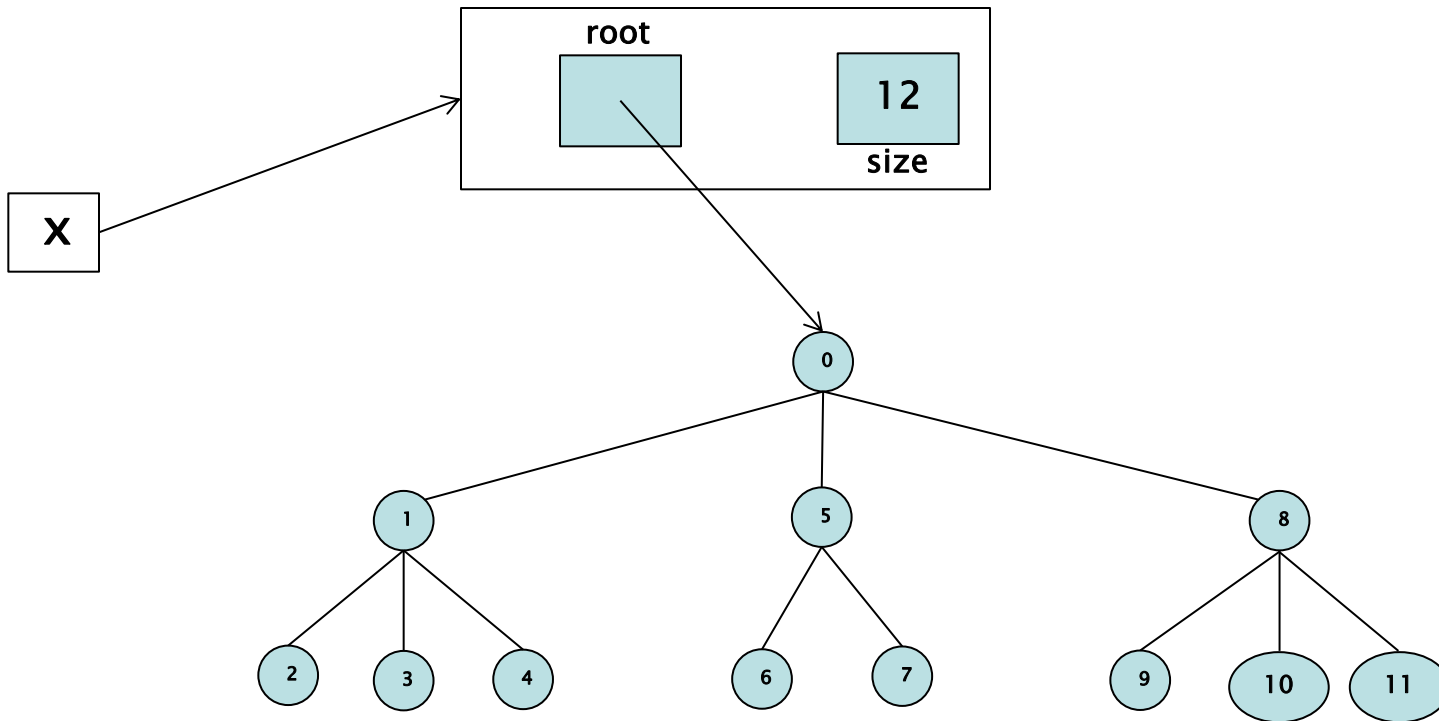
Qual o percurso preordem desta árvore ?





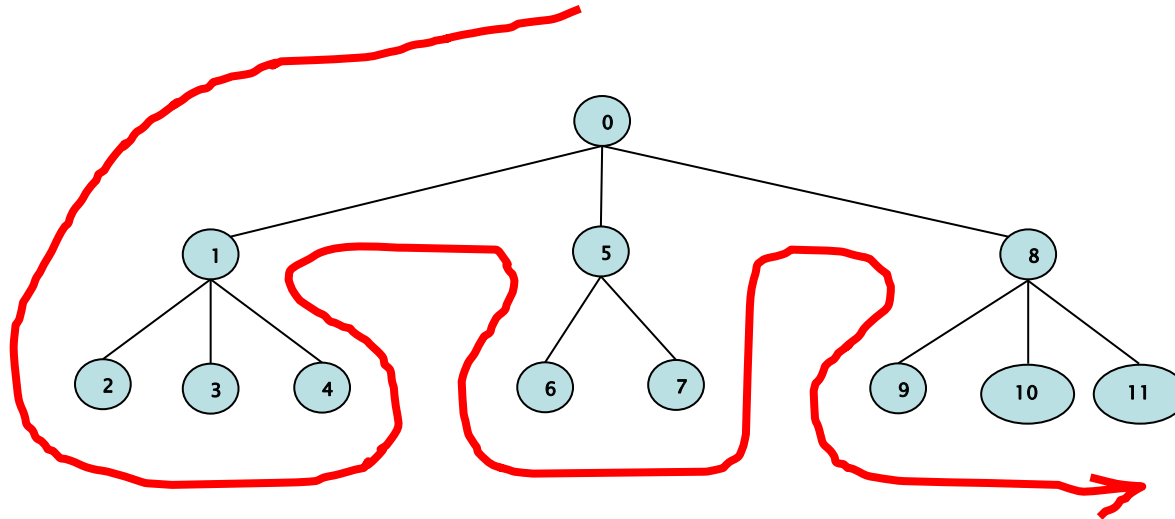
Preorder



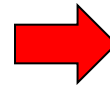


Qual o percurso preordem desta árvore ?





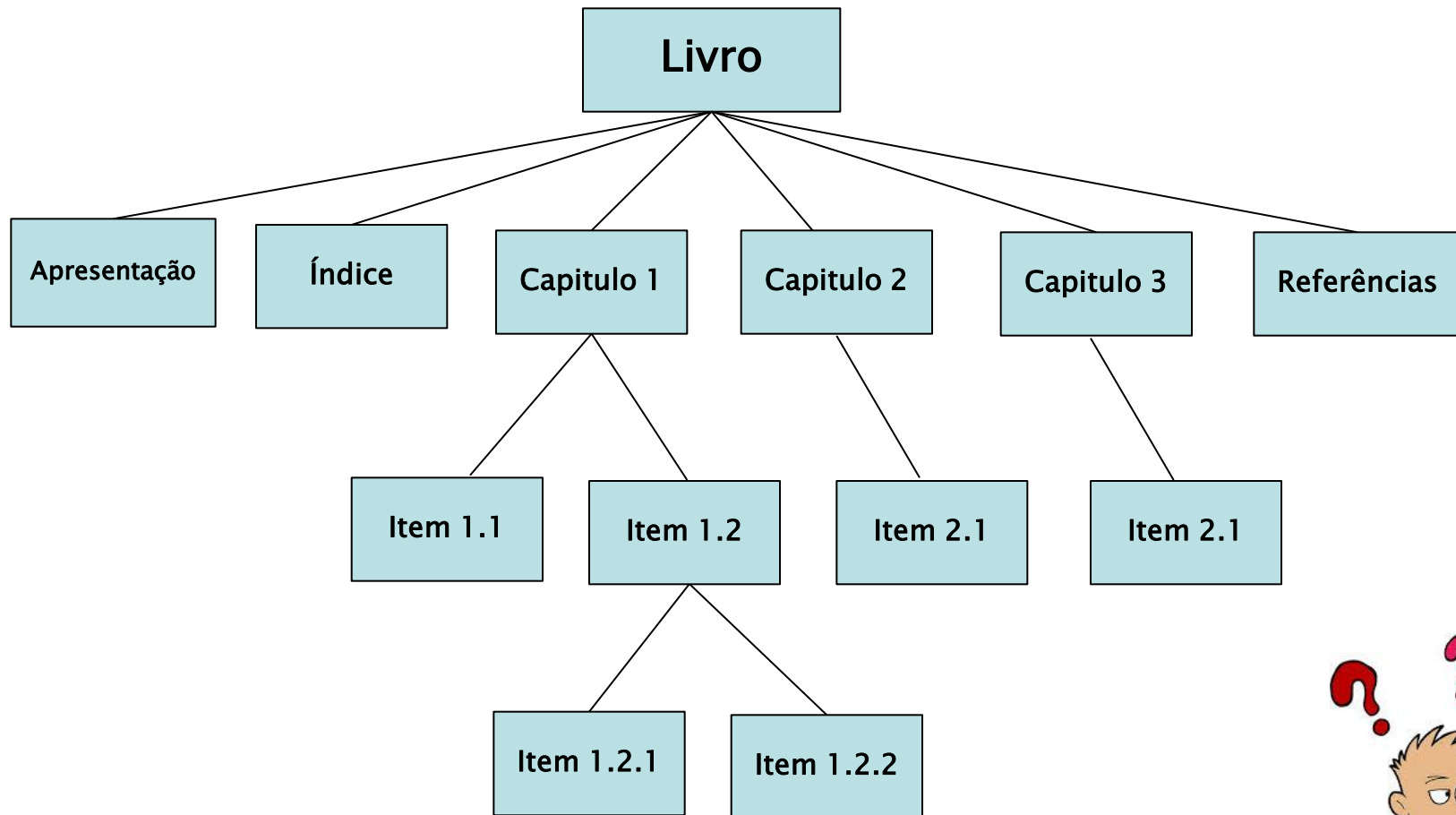
Resposta: Preorder



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



Exercício



Qual o percurso preordem desta árvore ?



Percurso – Postorder

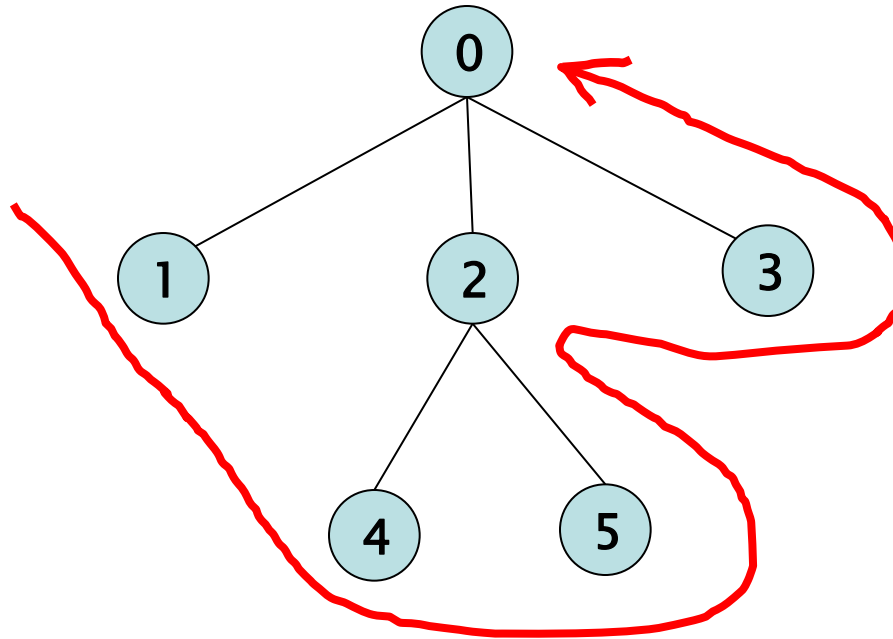
- Este algoritmo pode ser visto como o oposto do percurso preorder, pois as sub-árvores dos filhos são recursivamente atravessadas e em seguida o root é visitado.

```
void posorder(struct node * ponteiro ) {  
  
    struct node * trab = ponteiro -> firstchild;  
  
    while (trab != NULL) {  
  
        posorder (trab);  
        trab = trab ->next;  
    }  
  
    printf ("\t %d" , ponteiro -> data );  
  
}
```



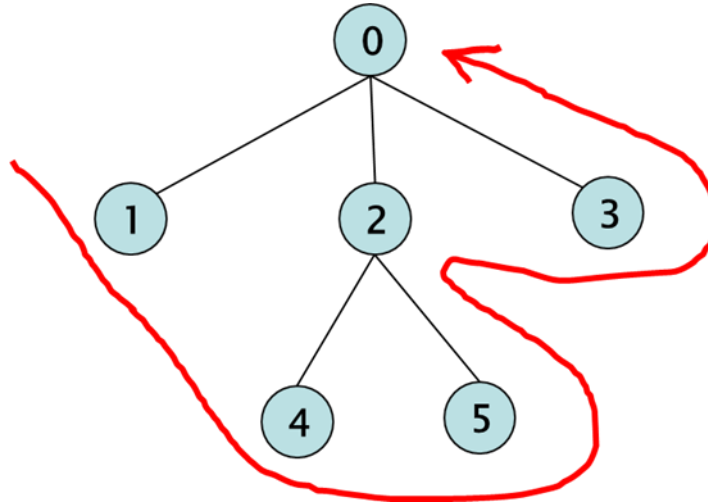


Percurso – Postorder

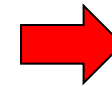


- Nós são visitados nesta ordem: 1 4 5 2 3 0
- Cada nó é visitado somente uma vez, assim o percurso preorder gasta tempo $O(n)$, onde n é o total de nós da árvore.





Resposta: Posorder

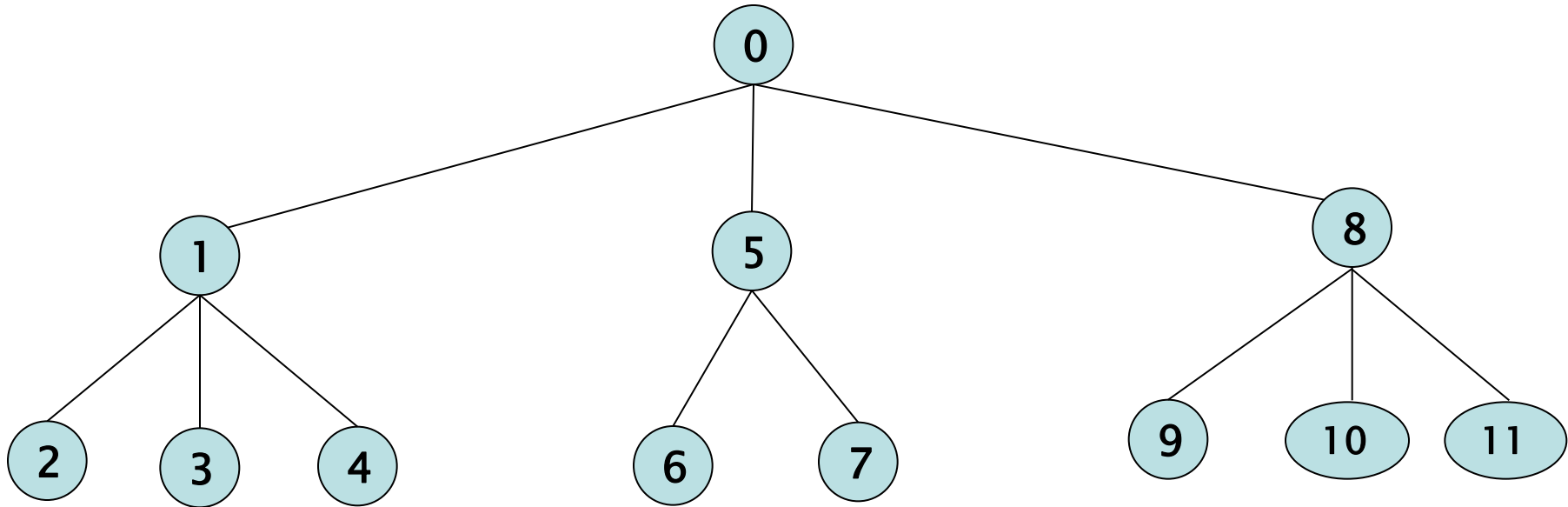


1
4
5
2
3
0

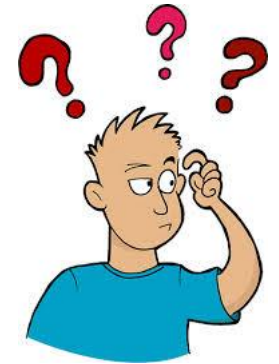




Outro exemplo

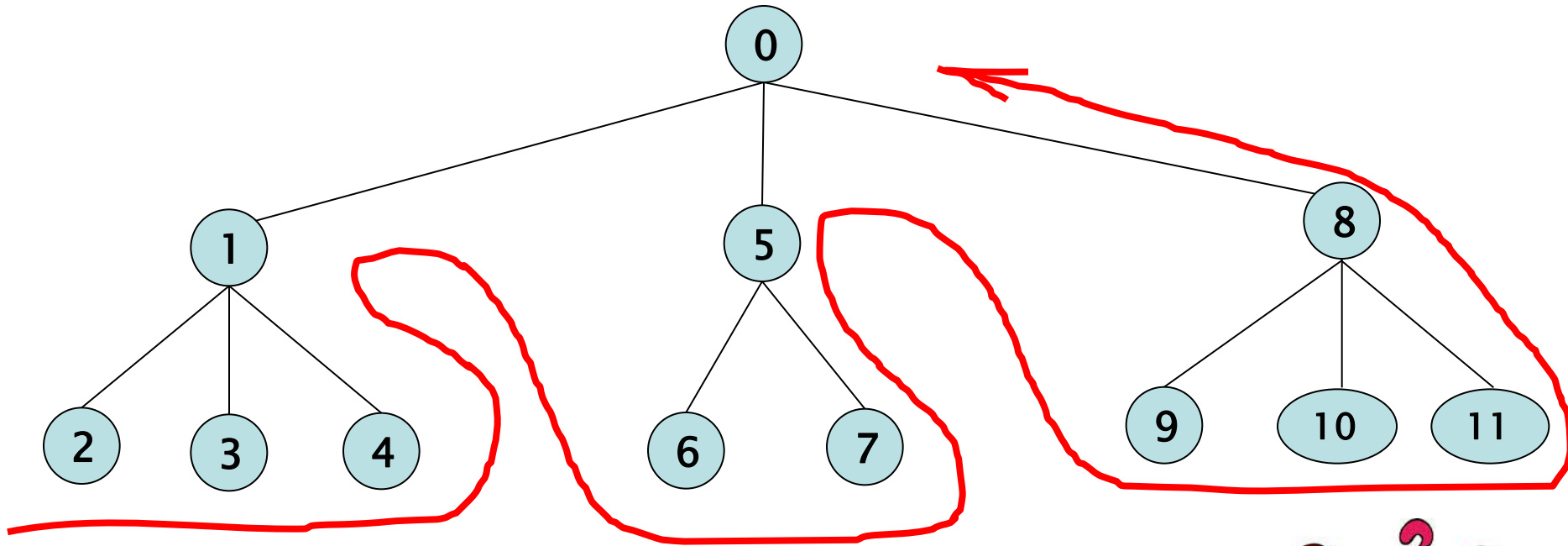


Qual o percurso postordem desta árvore ?



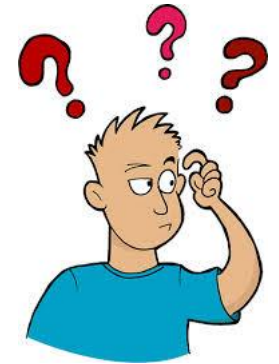


Outro exemplo



Qual o percurso **postordem** desta árvore ?

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

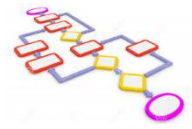




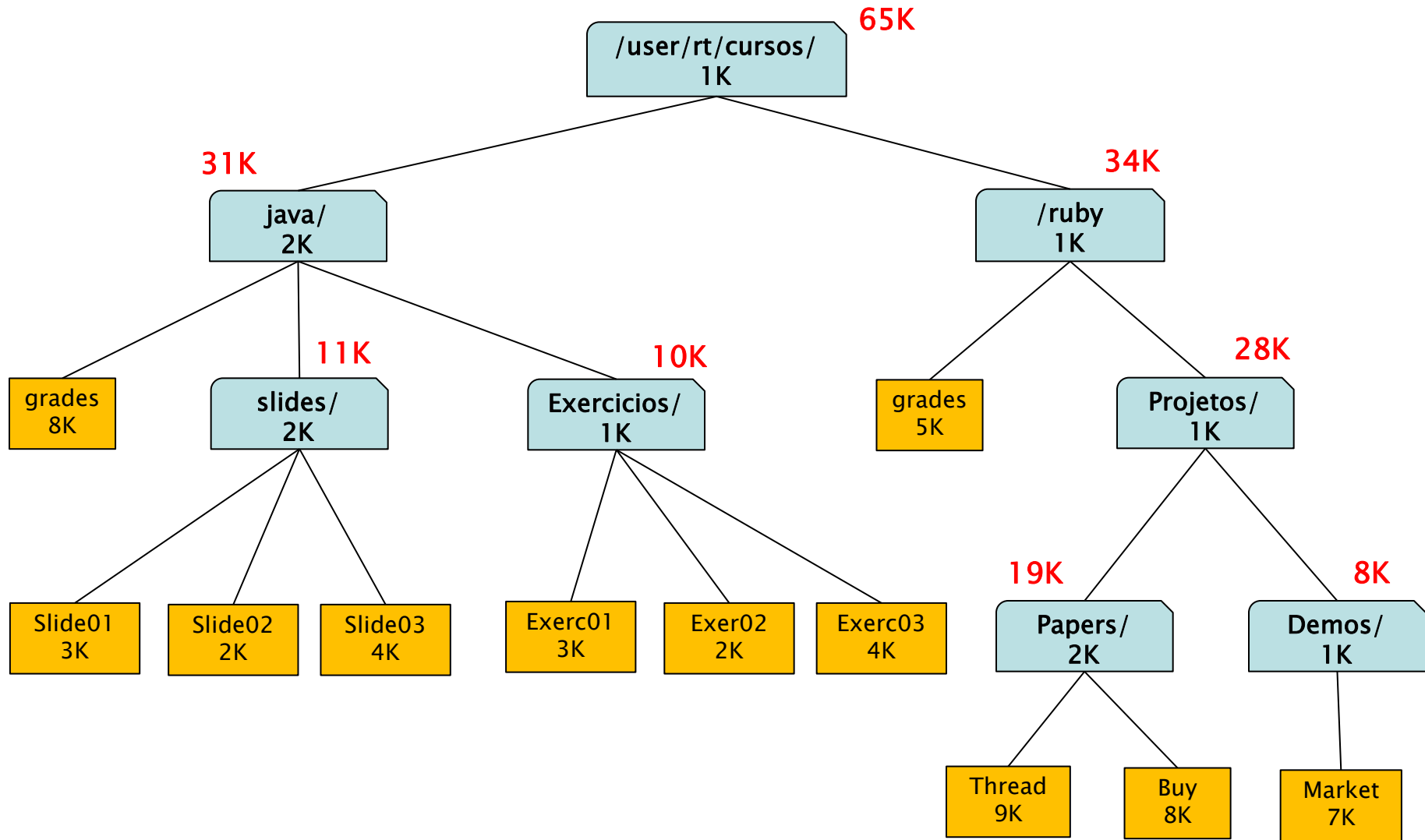
Aplicação travessia postorder

- 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 C para retornar o espaço total de bytes armazenados por um sistema de arquivos.

