

# Árvores

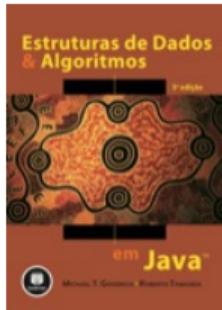
Roland Teodorowitsch

Algoritmos e Estruturas de Dados I - Escola Politécnica - PUCRS

16 de novembro de 2023

# Introdução

# Leitura(s) Recomendada(s)



Capítulo 7, Seções 7.1, 7.2, 7.3, 10.1, 10.2 e 10.5

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de dados e algoritmos em Java**. Tradução: Bernardo Copstein. 5. ed. Porto Alegre: Bookman, 2013. xxii, 713 p. E-book. ISBN 9788582600191. Tradução de: Data Structures and Algorithms in Java, 5th Edition. Disponível em: <<https://integrada.minhabiblioteca.com.br/#/books/9788582600191/>>. Acesso em: 01 ago. 2023.

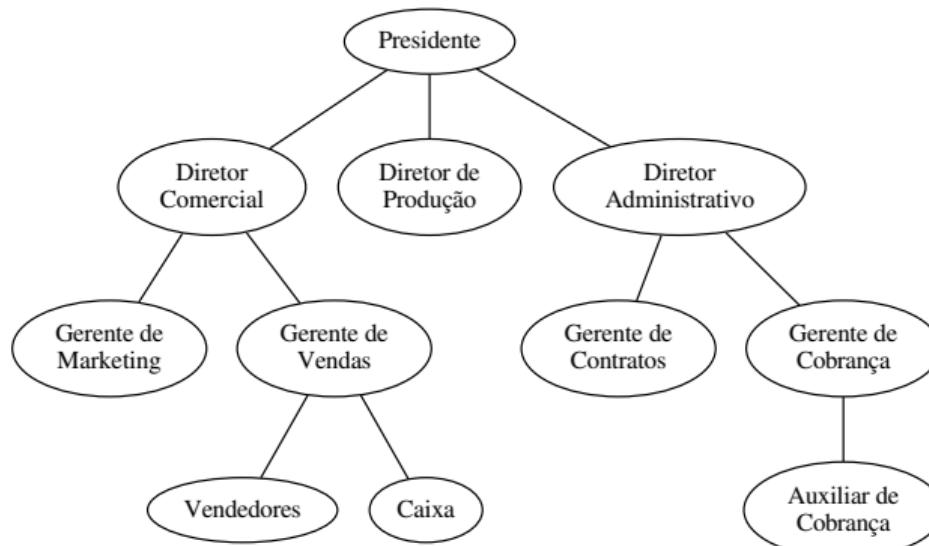
# Conceitos e Terminologia

# Árvores

- Estruturas de dados não lineares
- Permitem a implementação de vários algoritmos mais rápidos do que no uso de estruturas de dados lineares como as listas
- Fornecem uma forma natural de organizar os dados
  - Sistemas de arquivos
  - Bancos de dados
  - Sites da Web

# Árvore

- Tipo abstrato de dados que armazena elementos de maneira **hierárquica**
- Normalmente, são desenhadas colocando-se os elementos dentro de elipses ou retângulos e conectando-os com linhas retas

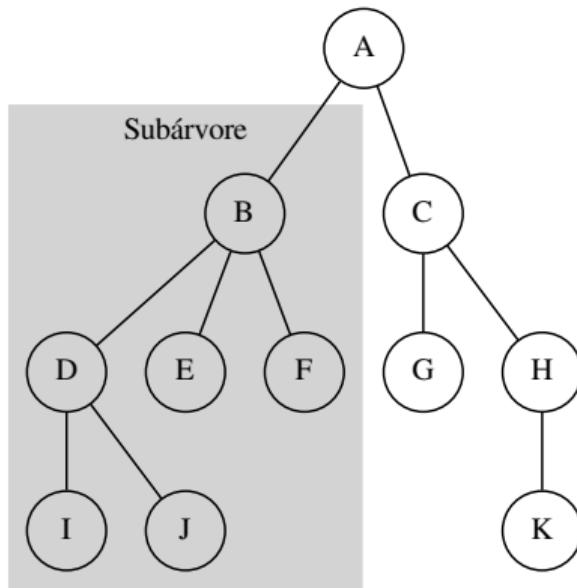


# Definição

- Formalmente uma árvore  $T$  é definida como um conjunto finito de um ou mais **nodos**, com a seguinte propriedade:
  - Se  $T$  não é vazia, existe um nodo denominado **raiz** da árvore
  - Os demais nodos formam  $m > 0$  conjuntos disjuntos  $S_1, S_2, \dots, S_m$ , sendo cada um destes conjuntos uma árvore
  - As árvores  $S_i$  ( $1 \leq i \leq m$ ) são chamadas de **subárvores**
- Pela definição
  - Cada nodo da árvore é a raiz de uma subárvore

# Definição

- Portanto, uma árvore pode ser representada da seguinte forma:



- A é a raiz da árvore

# Relacionamentos entre nodos

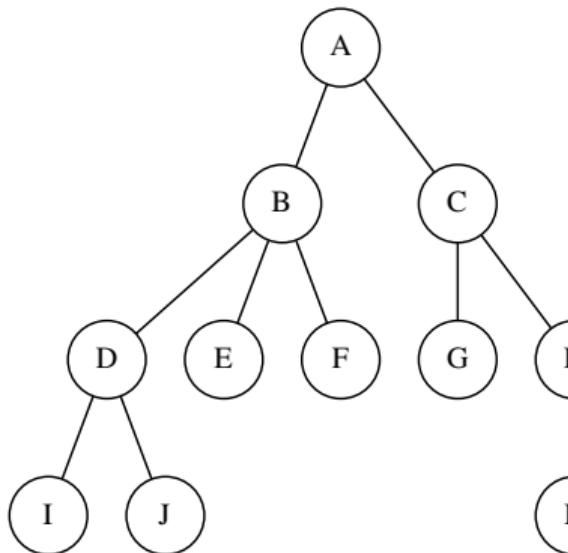
- Outra propriedade de uma árvore **T**:
  - Cada nodo **v** de **T** diferente da raiz tem um único nodo **pai**, **w**
  - Todo nodo com **pai** **w** é **filho** de **w**
- Pela definição
  - Uma árvore pode ser vazia, isto é, não possui nodos
  - Esta convenção permite que se defina uma árvore recursivamente
    - Uma árvore **T** ou está vazia, ou consiste de um nodo **r**, chamado de raiz de **T**, e um conjunto de árvores cujas raízes são filhas de **r**

# Relacionamentos entre nodos

- Outros relacionamentos entre nodos
  - Dois nodos que são filhos de um mesmo pai são **irmãos**
  - Um nodo **v** é **externo** se **v** não tem filhos
  - Um nodo **v** é **interno** se tem um ou mais filhos
- Nodo **interno** também é conhecido como **galho**
- Nodo **externo** também é conhecido como **folha**

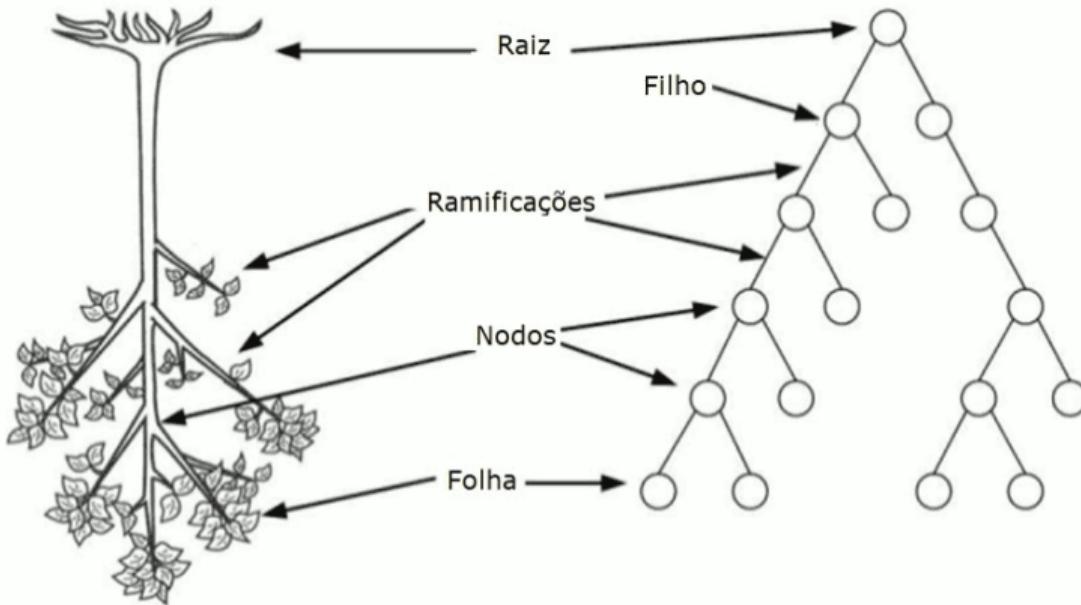
## Relacionamentos entre nodos

- A raiz de uma árvore é chamada de **pai** de suas subárvore
- As raízes das subárvore de um nodo são chamadas de **irmãos**, que, por sua vez, são **filhos** de seu nodo pai



- A é pai de B e C
- D, E e F são irmãos
- I é filho de J

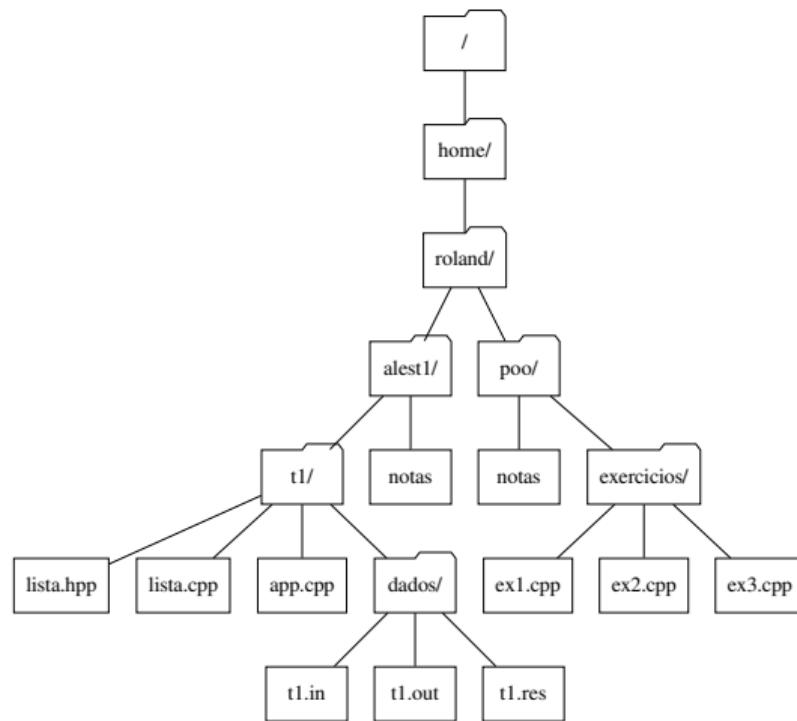
# “Árvore Invertida”



Fonte: [https://di.ubi.pt/~cbarrico/Disciplinas/AlgoritmosEstruturasDadosLEI/Downloads/Teorica\\_ConceitosGeraisArvores.pdf](https://di.ubi.pt/~cbarrico/Disciplinas/AlgoritmosEstruturasDadosLEI/Downloads/Teorica_ConceitosGeraisArvores.pdf)

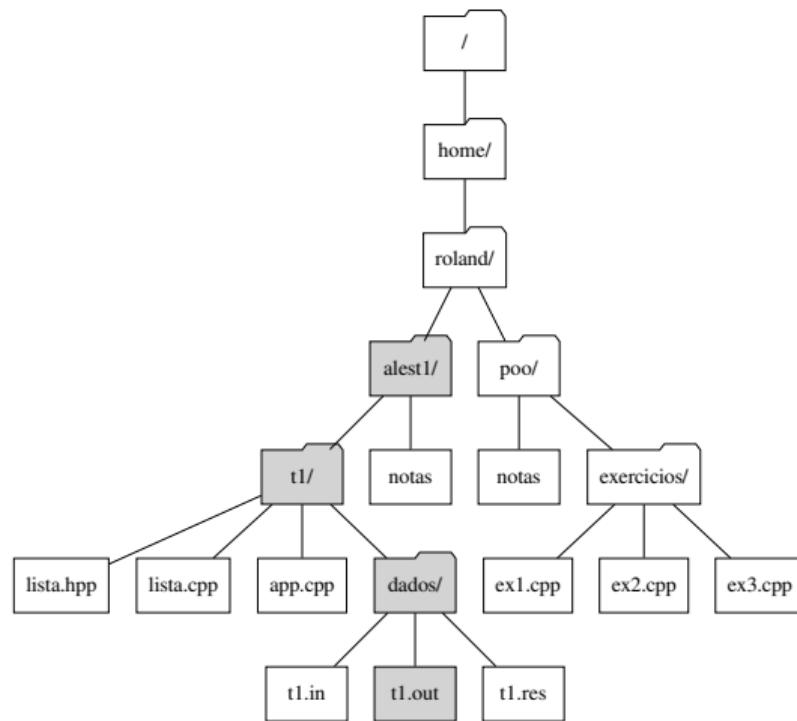
# Exemplo

- Organização hierárquica dos arquivos nos sistemas operacionais é uma árvore
- Nodos internos, neste caso, são associados a diretórios, e nodos externos a arquivos
- No Linux o diretório raiz é “/”



# Arestas e Caminhos em Árvores

- Uma aresta de uma árvore  $T$  é um par de nodos  $(u, v)$  tal que  $u$  é pai de  $v$ , ou vice-versa
- Um caminho de  $T$  é uma sequência de nodos tais que quaisquer dois nodos consecutivos da sequência formam uma aresta
- Exemplo: alest1/, t1/, dados/ e t1.out formam um caminho



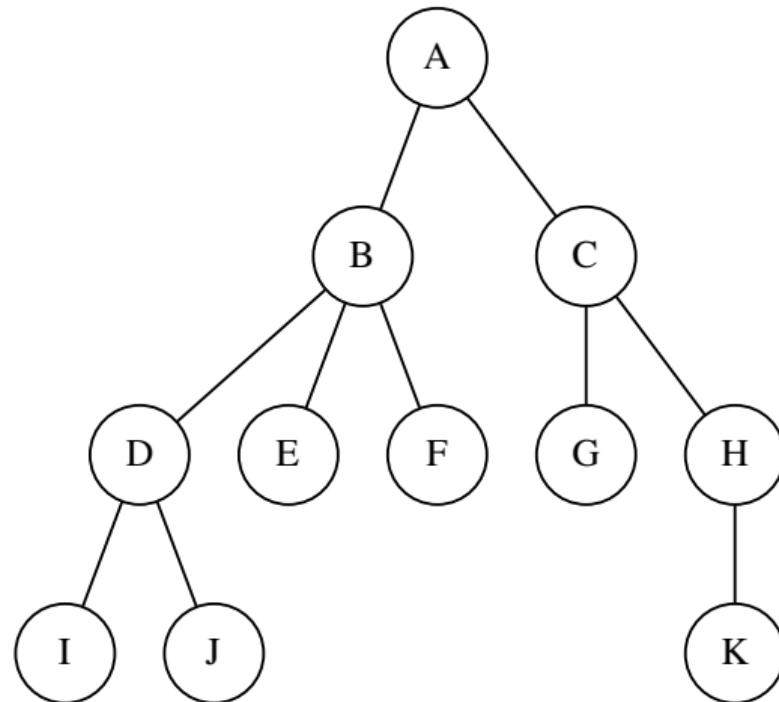
# Grau, Nível e Altura

- Grau
  - É o número de subárvore de um nodo
  - Quando o grau é zero, ou seja, o nodo não possui filhos, ele é **folha**
- Nível de um nodo
  - É o número de linhas que liga o nodo à raiz, sabendo que a raiz é o nível zero
- Altura
  - É definida como sendo o nível mais alto da árvore

# Grau, Nível e Altura

- Grau:

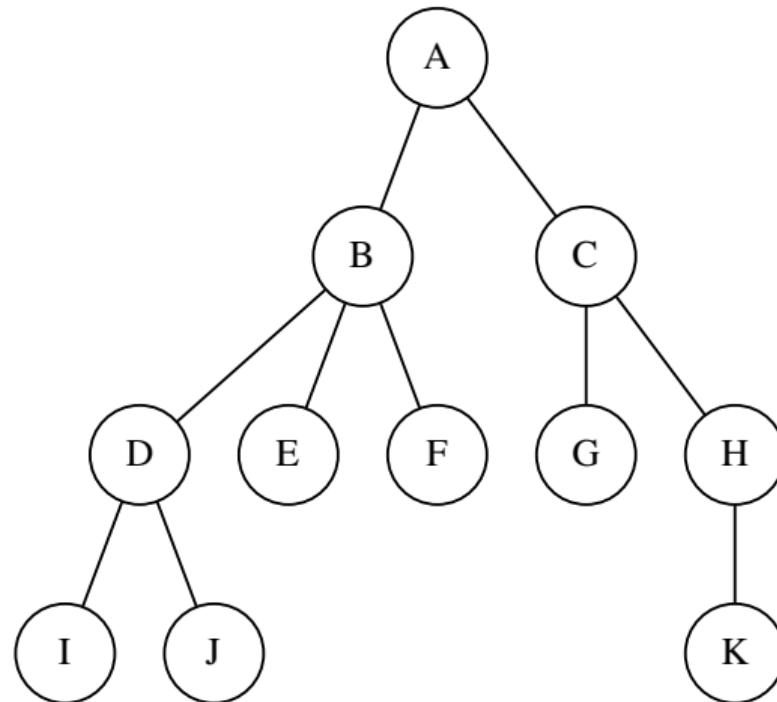
- Nodo A:
- Nodo B:
- Nodo C:
- Nodo D:
- Nodo E:
- Nodo F:
- Nodo G:
- Nodo H:
- Nodo I:
- Nodo J:
- Nodo K:



# Grau, Nível e Altura

- Grau:

- Nodo A: 2
- Nodo B: 3
- Nodo C: 2
- Nodo D: 2
- Nodo E: 0
- Nodo F: 0
- Nodo G: 0
- Nodo H: 1
- Nodo I: 0
- Nodo J: 0
- Nodo K: 0



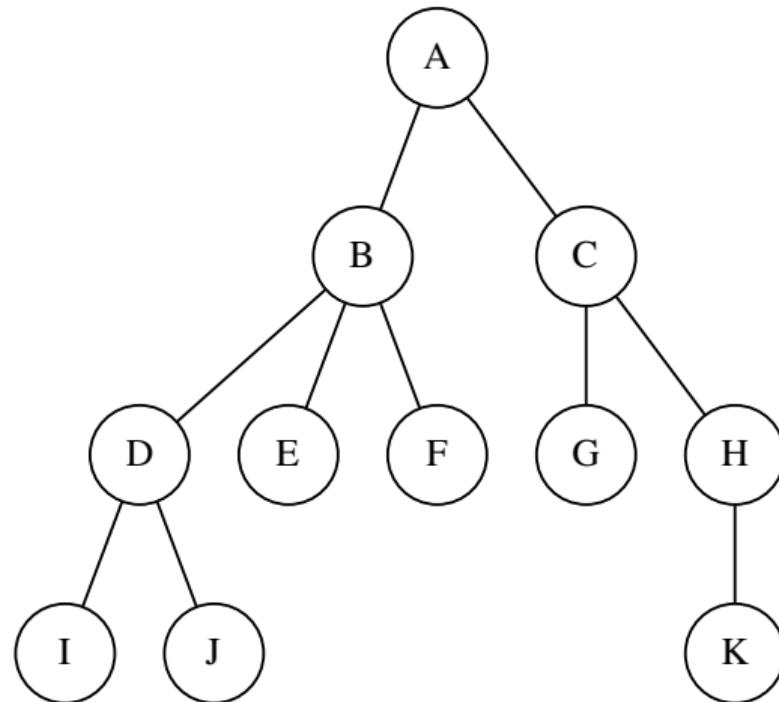
# Grau, Nível e Altura

- Nível:

- Nodo A:
- Nodo B:
- Nodo C:
- Nodo D:
- Nodo E:
- Nodo F:
- Nodo G:
- Nodo H:
- Nodo I:
- Nodo J:
- Nodo K:

- Altura:

- Árvore:



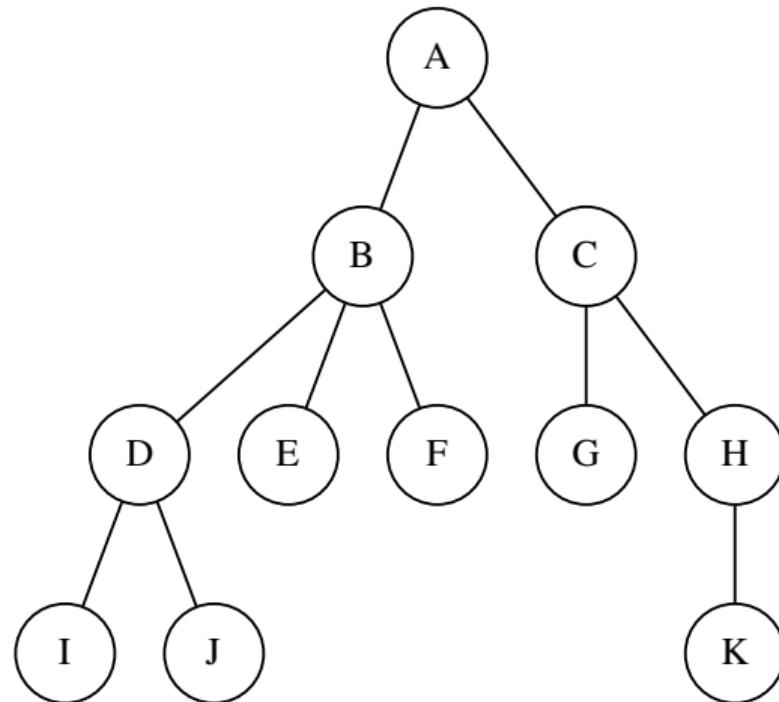
# Grau, Nível e Altura

- Nível:

- Nodo A: 0
- Nodo B: 1
- Nodo C: 1
- Nodo D: 2
- Nodo E: 2
- Nodo F: 2
- Nodo G: 2
- Nodo H: 2
- Nodo I: 3
- Nodo J: 3
- Nodo K: 3

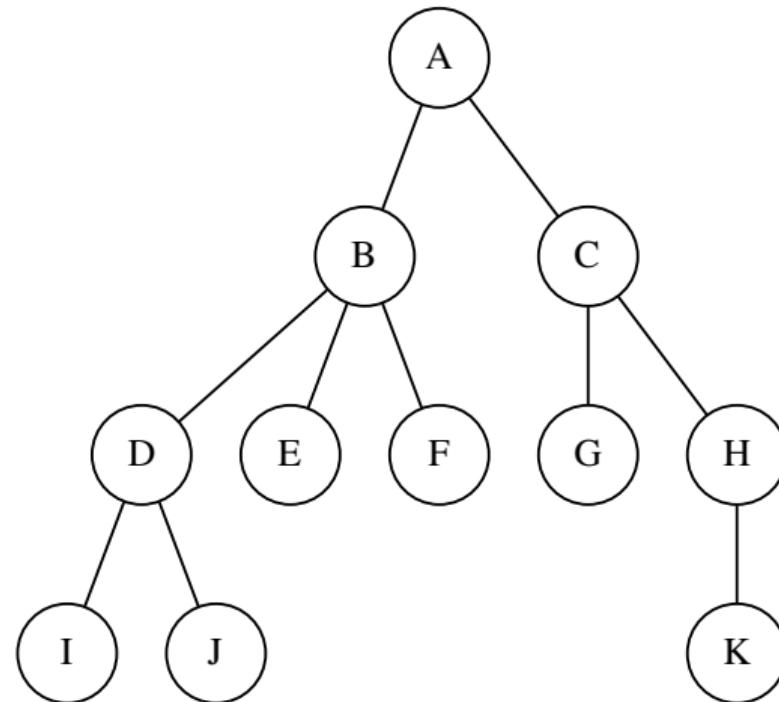
- Altura:

- Árvore: 3



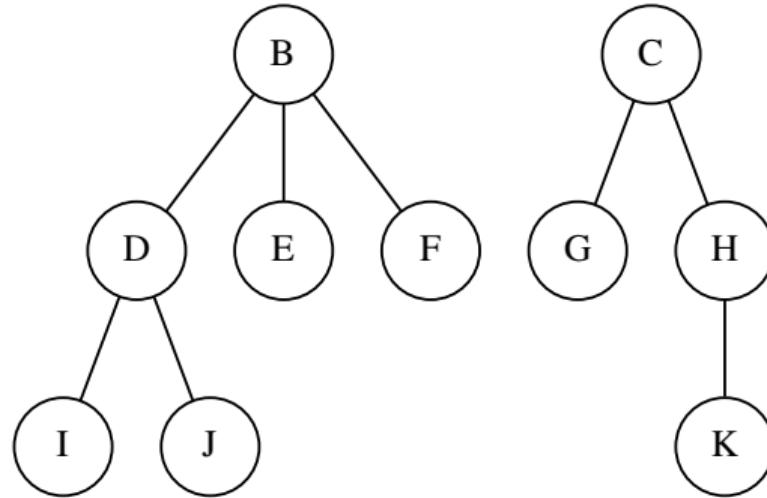
# Grau, Nível e Altura

- Raiz: A
- Nodos internos: B, C, D, H
- Folhas: I, J, E, F, G, K



# Floresta

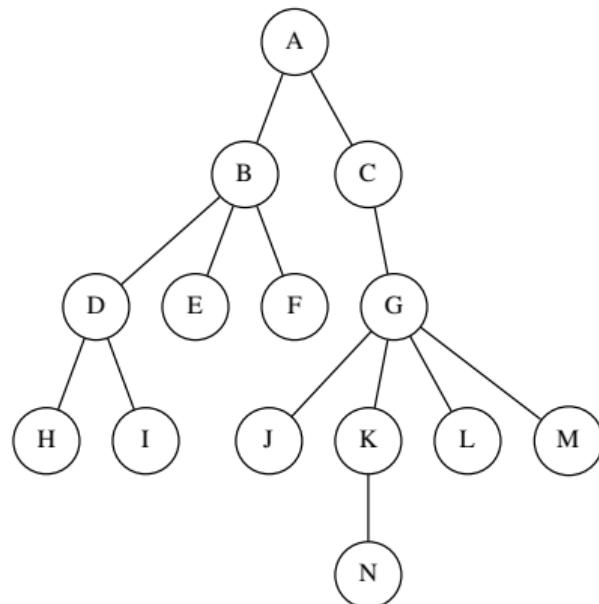
- Conjunto de uma ou mais árvores disjuntas
- Se eliminarmos o nodo raiz de uma árvore, obtém-se uma floresta
- Os filhos da raiz original irão se transformar nas raízes das novas árvores



# Exercícios

# Exercício 1

① Analise a árvore abaixo.

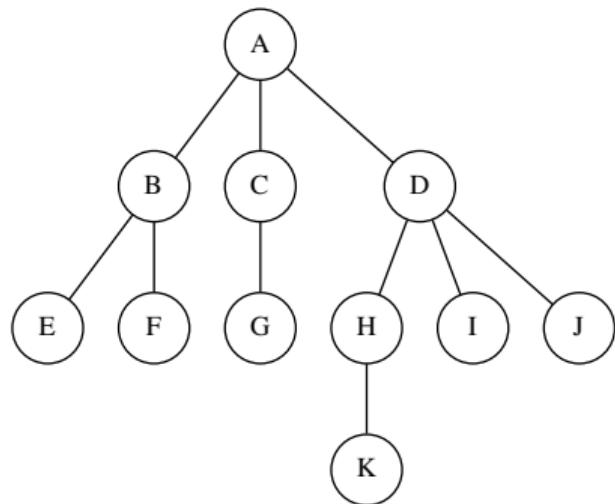


E responda:

- Qual é a altura da árvore?
- Quais são as folhas da árvore?
- Quais são os nodos irmãos?
- Os nodos D e G são pais de que nodos?
- Qual é o grau do nodo B?
- Qual é o grau do nodo G?
- Quais são os níveis dos nodos B, G, H, L e N?

## Exercício 2

② Analise a árvore abaixo.



E responda:

- Qual é a altura da árvore?
- Quais são as folhas da árvore?
- Quais são os nodos irmãos?
- Quais são os nodos internos?
- Qual é o grau do nodo C?
- Qual é o grau do nodo D?
- Quais são os níveis dos nodos C, H e K?

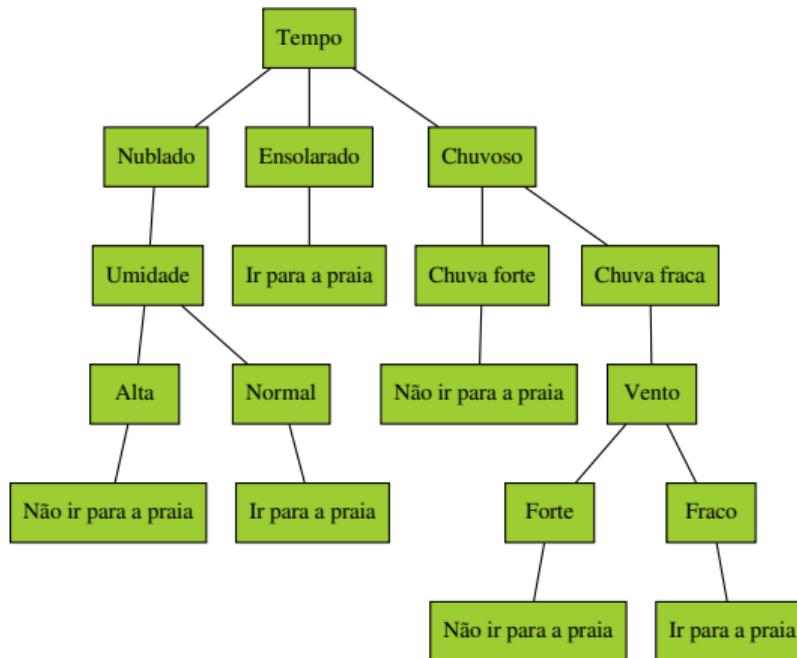
# Aplicações

# Aplicações

- Árvores são estruturas de dados adequadas para representar diversos tipos de informações
- Várias aplicações que podem utilizar árvores

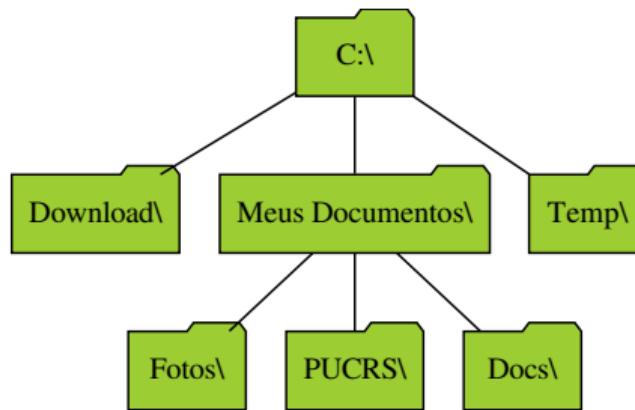
# Árvore de decisão

- Exemplo: Ir ou não ir para praia?



# Representação de estruturas/informações hierárquicas

- Árvore genealógica
- Organização de livros e documentos
- Organização hierárquica de cargos de uma empresa
- Sistemas de arquivos



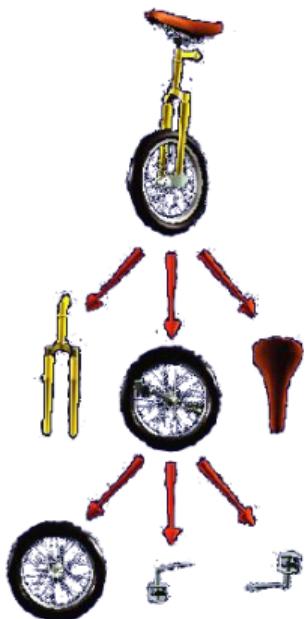
# Representação de estruturas/informações hierárquicas

- Arquivos HTML e XML

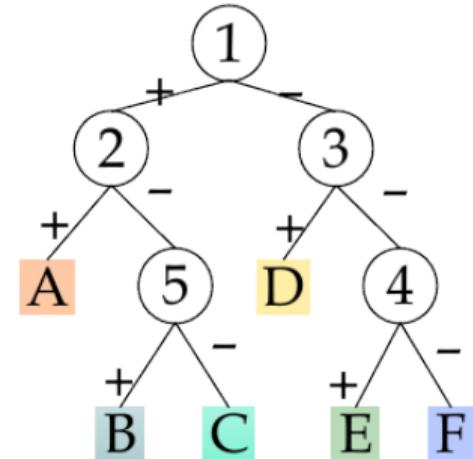
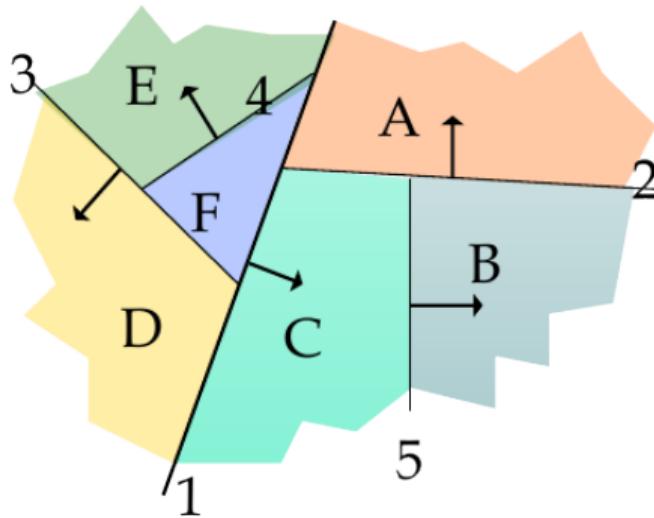
```
<?xml version="1.0"?>
<Company>
  <Employee>
    <FirstName>Maria</FirstName>
    <LastName>Silva</LastName>
    <PhoneNumber>(51)98986767</PhoneNumber>
    <Email>maria.silva@gmail.com</Email>
    <Address>
      <Street>Rua dos Andradas 1586/202</Street>
      <City>Porto Alegre</City>
      <State>RS</State>
      <Zip>90021-212</Zip>
    </Address>
  </Employee>
</Company>
```

# Computação Gráfica

- Grafo de cena

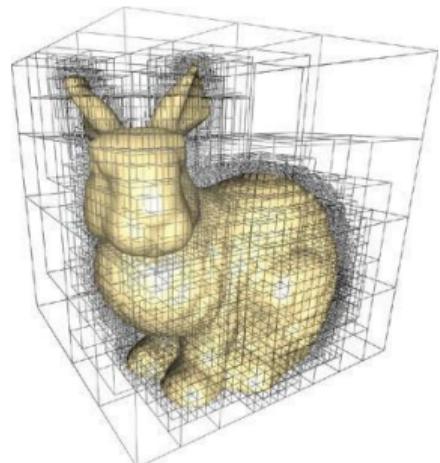
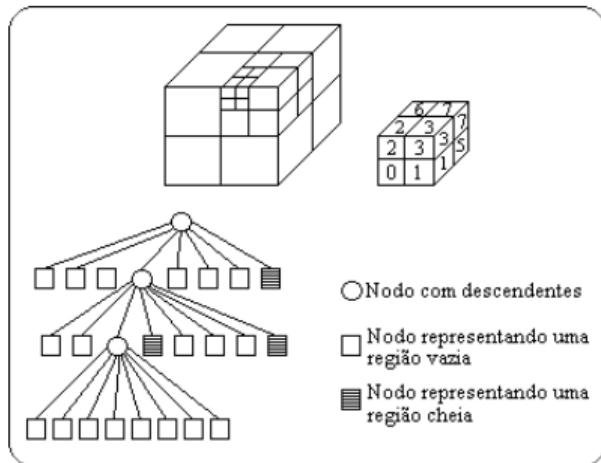
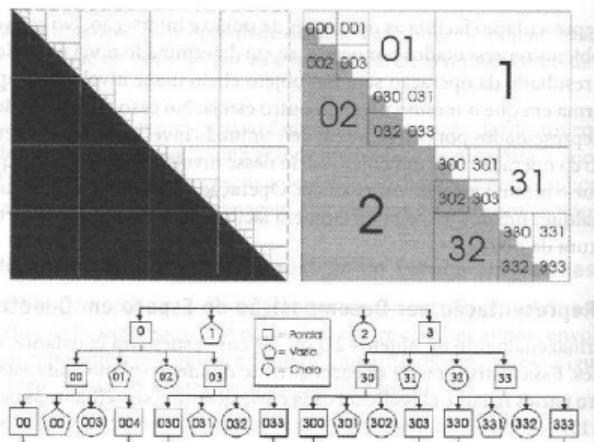


- Algoritmos para remoção de superfícies escondidas (árvore *Binary Space-Partitioning*)



# Computação Gráfica

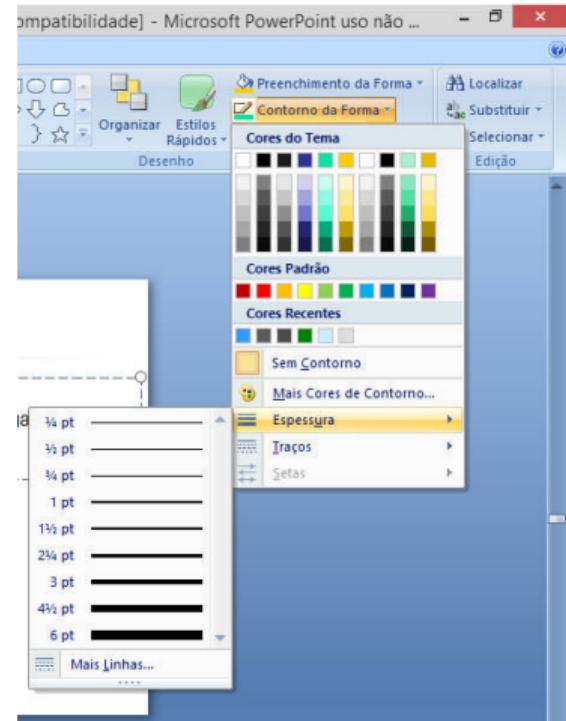
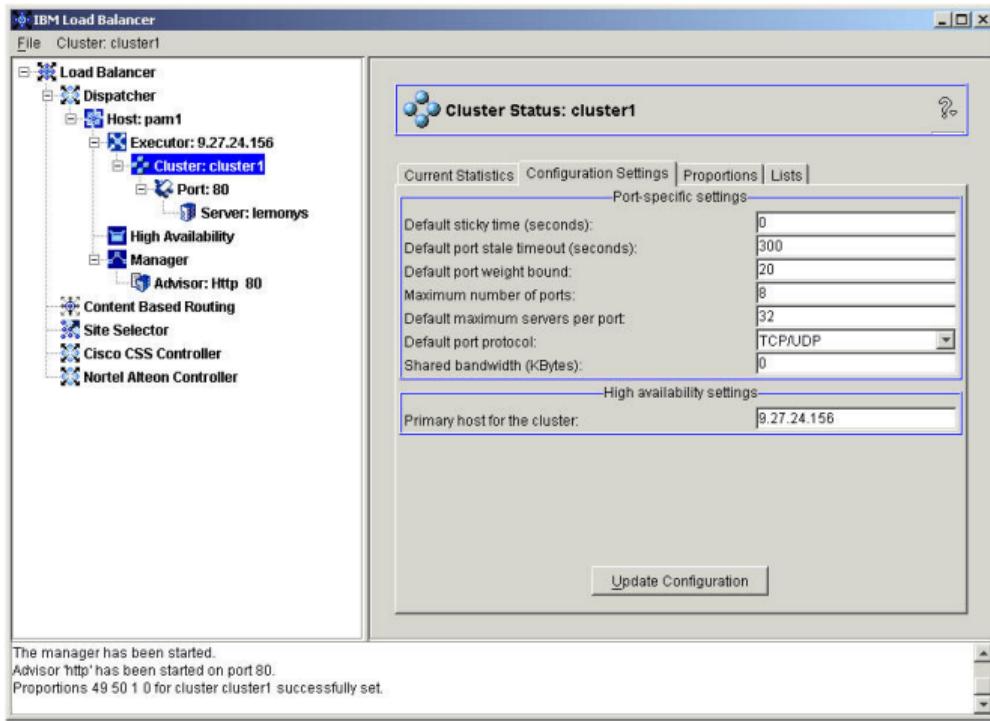
- Representação de objetos (Quadtree e Octree), etc.



# Eliminatórias de Campeonatos

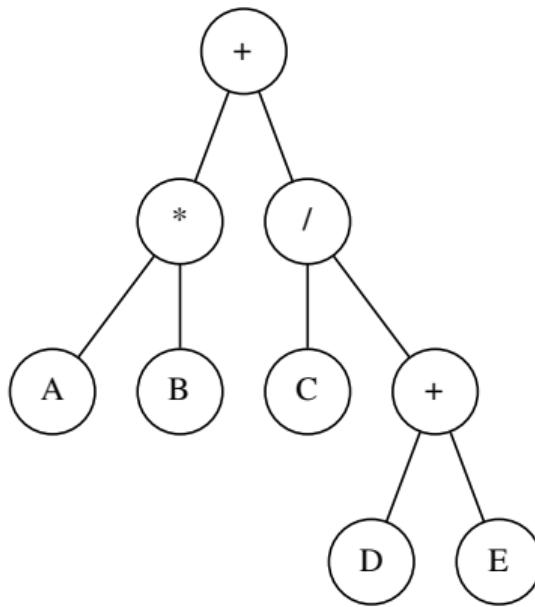


# Interfaces Gráficas com o Usuário

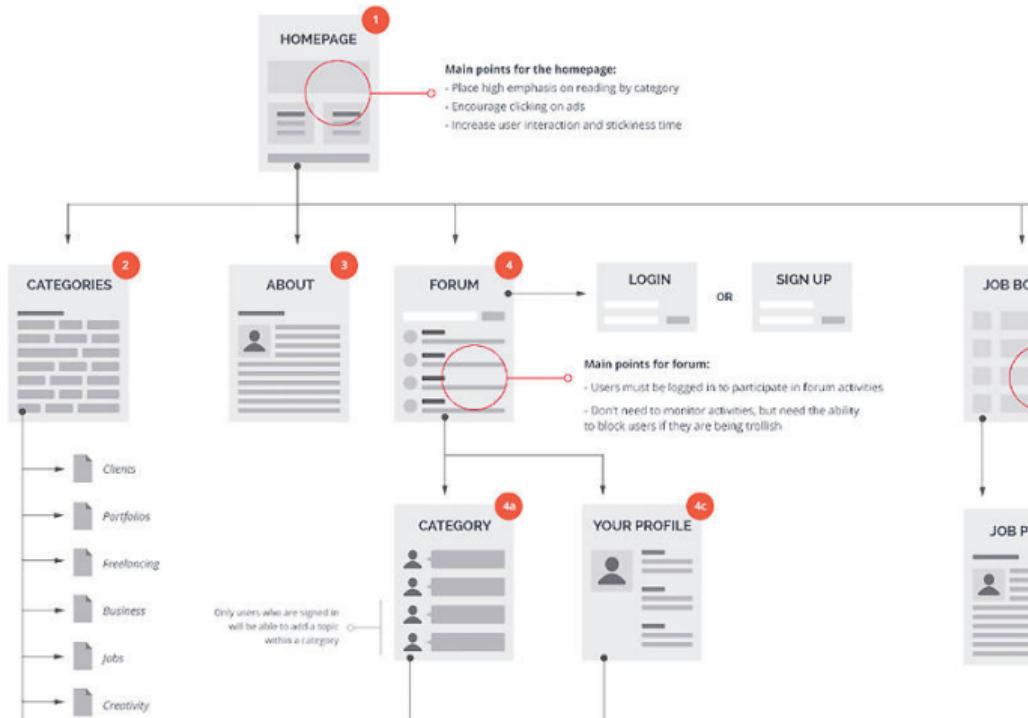


# Expressões Aritméticas

- Pode-se usar árvores para representar e avaliar expressões aritméticas
- Exemplo:  $A * B + C / ( D + E )$



# Organização das Páginas de um Site



Fonte: <https://dribbble.com/shots/1198252-Sitemap-For-Student-Guide>

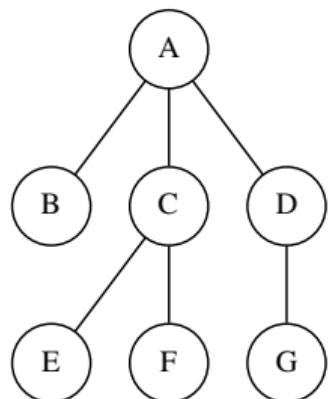
# Representação na Memória

# Representação na Memória

- Da mesma forma que as estruturas de dados lineares, podemos alocar as árvores de duas maneiras
  - Por contiguidade
  - Por encadeamento

# Representação por Contiguidade

- A árvore é armazenada em um arranjo
- Cada posição por arranjo pode, por exemplo, conter, além da informação do nodo, referências aos nodos filhos



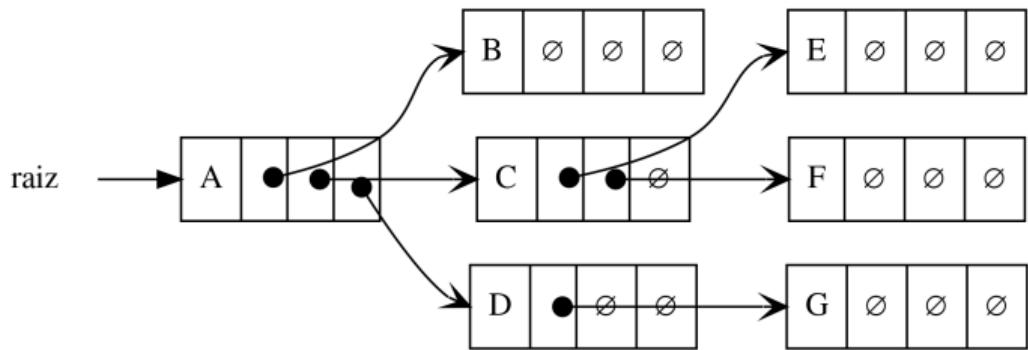
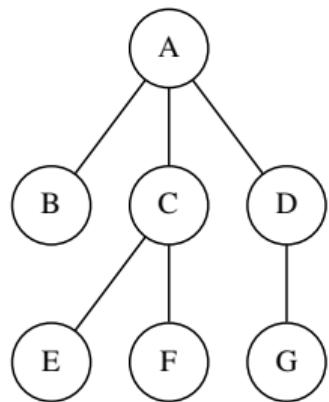
índice	conteúdo	filho1	filho2	filho3
0	A	1	2	3
1	B	-1	-1	-1
2	C	4	5	-1
3	D	6	-1	-1
4	E	-1	-1	-1
5	F	-1	-1	-1
6	G	-1	-1	-1

# Representação por Contiguidade

- Vantagem:
  - Forma de armazenar árvores em arquivos
- Desvantagem:
  - Quantidade de processamento para inserção, remoção ou mesmo localização de um nodo

# Representação por Encadeamento

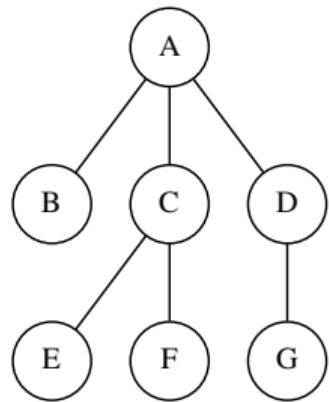
- Forma **mais usual**
- Facilita as operações de inserção, remoção e pesquisa na árvore
- Cada nodo conterá, além da informação, as referências das suas subárvores



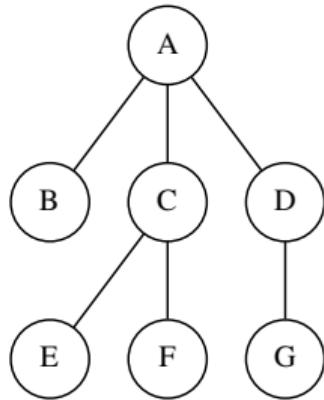
# Representação por Encadeamento

- No caso de árvores “genéricas”, cada nodo pode ter uma quantidade de subárvores diferentes
- Torna-se necessário:
  - Limitar, ou seja, determinar o número máximo de subárvores que cada nodo deve conter
  - Ou ter uma lista de subárvores
- Isso é necessário, pois os nodos de uma mesma árvore são todos do mesmo tipo

# Exemplo



# Exemplo



```

#include <iostream>

using namespace std;

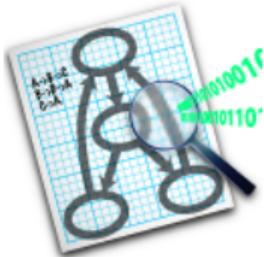
struct Node {
    char info; Node *child1, *child2, *child3;
    Node(char i, Node *c1 = nullptr, Node *c2 = nullptr, Node *c3 = nullptr) {
        info = i; child1 = c1; child2 = c2; child3 = c3;
        cout << "+Node(" << info << ") criado..." << endl;
    }
    ~Node() { cout << "-Node(" << info << ") destruido..." << endl; }
};

int main() {
    Node *b      = new Node('B'),           *e = new Node('E'),
          *f      = new Node('F'),           *g = new Node('G');      // Cria folhas
    Node *c      = new Node('C', e, f),     *d = new Node('D', g);  // Cria intermediarios
    Node *root = new Node('A', b, c, d);    // Cria raiz

    delete b;  delete e;  delete f;  delete g; // Desaloca folhas
    delete c;  delete d;                // Desaloca intermediarios
    delete root;                      // Desaloca raiz

    return 0;
}
  
```

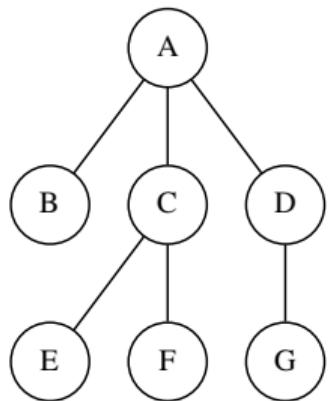
# GraphViz



- Graphviz (abreviação de *Graph Visualization Software*) é uma ferramenta de código-fonte aberto para desenhar grafos (formados por nodos e arestas) especificados a partir de uma linguagem de escrita chamada DOT (que usa a extensão “gv”))
- GraphViz também provê bibliotecas para que aplicações possam usar suas facilidades
- É um *software* livre licenciado através da Licença Pública Eclipse
- Página: <https://graphviz.org/>
- Há várias opções para gerar imagens a partir dos arquivos no formato DOT
  - <https://dreampuf.github.io/GraphvizOnline/>

# GraphViz

## Grafo:



## Versão 1:

```
graph "Árvore A" {
    node [shape=circle]
    A --- { B C D }
    C --- { E F }
    D --- G
}
```

## Versão 2:

```
graph "Árvore A (Versão 2)" {
    node [shape=circle]
    A --- B
    A --- C
    A --- D
    C --- E
    C --- F
    D --- G
}
```

# Exercícios

## Exercício 3

3 Considere a árvore e o programa que cria esta árvore correspondente, usando estruturas encadeadas em C++, ambos apresentados na próxima página. Observe que o nodo declarado (`struct Node`) armazena um caractere (`char`) e suporta até 3 subárvore (ou seja, cada nodo pode ter 3 nodos filhos). Para este programa, implemente as seguintes funções:

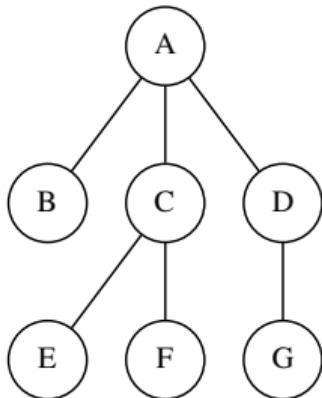
- `void clean(Node *root)`: que recebe o endereço de um nodo e faz a desalocação de todos os nodos da árvore a partir do nodo recebido (`root`) – esta função deve ser implementada de forma recursiva;
- `string strGraphViz(Node *root)`: que recebe o endereço de um nodo e gera uma cadeia de caracteres (`string`) com representação da árvore no formato DOT, usado pelo GraphViz – esta função deve: imprimir a parte inicial do formato DOT; chamar um outro método recursivo (por exemplo, `string strNode(Node *node)`) para gerar a lista de arestas entre os nodos; imprimir a parte final do formato DOT.

Rode o seu programa, por exemplo, com:

```
./exercicio03 > exercicio03.gv
```

Use o site <https://dreampuf.github.io/GraphvizOnline/> para verificar se o arquivo `exercicio03.gv` foi corretamente gerado.

# Exercício 3



```

#include <iostream>
#include <sstream>

using namespace std;

struct Node {
    char info; Node *child1, *child2, *child3;
    Node(char i, Node *c1 = nullptr, Node *c2 = nullptr, Node *c3 = nullptr) {
        info = i; child1 = c1; child2 = c2; child3 = c3;
        cerr << "+_Node("<< info << ")_criado..." << endl;
    }
    ~Node() { cerr << "-_Node("<< info << ")_destruido..." << endl; }
};

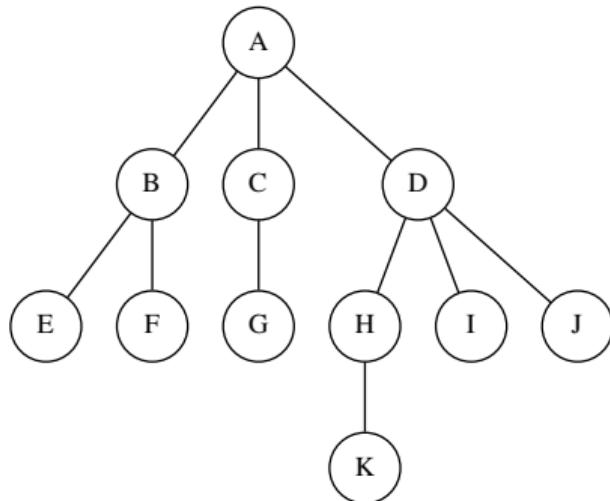
string strGraphViz(Node *root) {
    stringstream ss; /* IMPLEMENTE AQUI */ return ss.str();
}

void clean(Node *subtree) { /* IMPLEMENTE AQUI */ }

int main() {
    Node *b      = new Node('B'),           *e = new Node('E'),
          *f      = new Node('F'),           *g = new Node('G');      // Folhas
    Node *c      = new Node('C', e, f),     *d = new Node('D', g); // Intermediarios
    Node *root = new Node('A', b, c, d);   // Raiz
    cout << strGraphViz(root);
    clean(root);
    return 0;
}
  
```

## Exercício 4

- 4 Considere a árvore abaixo e, usando como modelo o código do exercício 3 (resolvido), implemente um programa em C++ que crie esta árvore em memória, que a imprima no formato DOT e que desaloque corretamente todos os nodos da árvore. Use o site <https://dreampuf.github.io/GraphvizOnline/> para verificar se o arquivo DOT foi corretamente gerado.



# TAD para Árvore

# TAD para Árvore

- Um TAD para árvore:

- Armazena elementos em nodos
- O posicionamento dos nodos satisfaz as relações pai-filho
- Operações consideram as propriedades desta estrutura de dados hierárquica

# TAD para Árvore

- Uma árvore deve disponibilizar métodos de acesso que retornam e aceitam posições:
  - `root()`: retorna a raiz da árvore
  - `parent(v)`: retorna o nodo pai de `v`, ocorrendo um erro se for a raiz
  - `children(v)`: retorna os filhos do nodo `v`

# TAD para Árvore

- Métodos de consulta:

- `isInternal(v)`: testa se um nodo v é interno e retorna true ou false
- `isExternal(v)`: testa se um nodo v é externo e retorna true ou false
- `isRoot(v)`: testa se um nodo v é raiz e retorna true ou false

# TAD para Árvore

- Métodos “genéricos” (não estão necessariamente relacionados com sua estrutura):
  - `size()`: retorna o número de nodos na árvore
  - `isEmpty()`: testa se a árvore tem ou não tem algum nodo
  - `iterator()`: retorna um iterator de todos os elementos armazenados nos nodos da árvore
  - `positions()`: retorna uma coleção com todos os nodos da árvore
  - `replaceElement(v,e)`: retorna o elemento armazenado em v e o substitui por e

## Observação

## Sobre a Solução dos Exercícios 3 e 4

- Os exercícios 3 e 4 usam o método recursivo `clean()` para desalocar árvores

```
void clean(Node *subtree) {
    if (subtree->child1 != nullptr) clean(subtree->child1);
    if (subtree->child2 != nullptr) clean(subtree->child2);
    if (subtree->child3 != nullptr) clean(subtree->child3);
    delete subtree;
}
```

- Considerando-se que `root` é um `Node *` e é a raiz da árvore, a sua desalocação seria então feita usando-se `clean(root);`;
- Uma alternativa mais interessante poderia ser usar o método destrutor para isso:

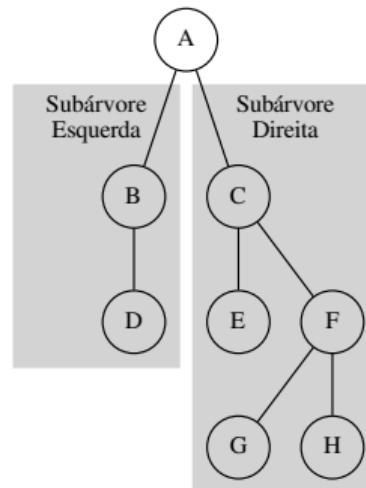
```
~Node() {
    if (child1 != nullptr) delete child1;
    if (child2 != nullptr) delete child2;
    if (child3 != nullptr) delete child3;
    cerr << "->Node(" << info << ") destruido..." << endl;
}
```

- Neste caso a árvore seria desalocada usando-se `delete root;`

# Árvore Binária

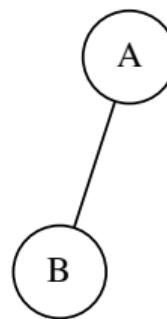
# Árvore Binária

- Uma árvore binária é aquela na qual o grau de cada nodo é menor ou igual a 2
- As subárvore de um nodo são chamadas de
  - Subárvore da esquerda
  - Subárvore da direita

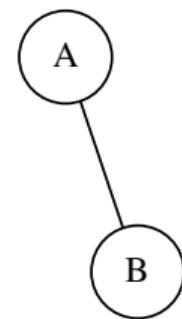


# Árvore Binária

- Se o grau de um nodo é 1, deve-se especificar se a sua subárvore é a da esquerda ou a da direita

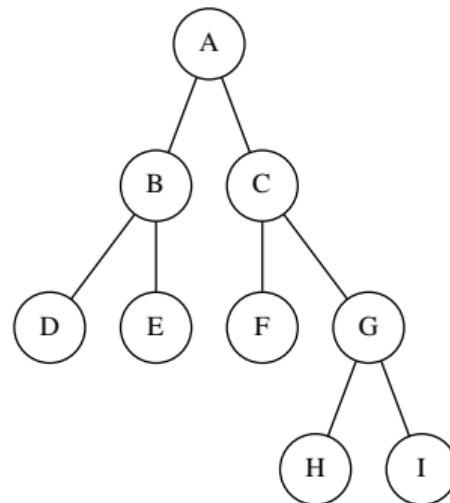


São  
árvores  
DIFERENTES



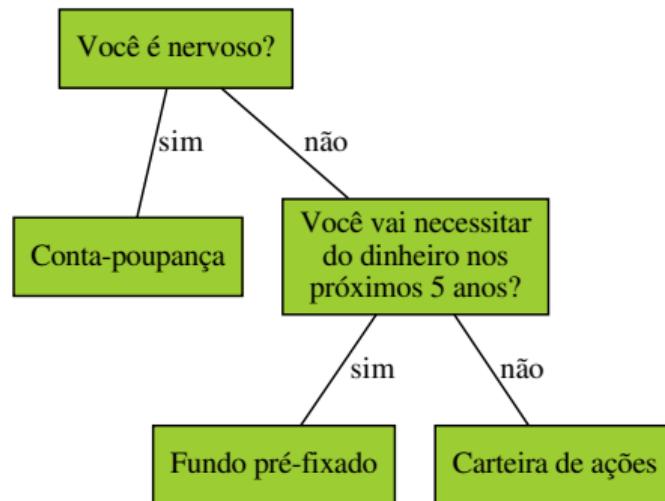
# Árvore Binária Própria

- Uma árvore binária é **própria** se cada um de seus nodos internos tiver dois filhos
- Todos os nodos, com exceção dos nodos-folha, têm exatamente dois filhos



# Árvores de Decisão

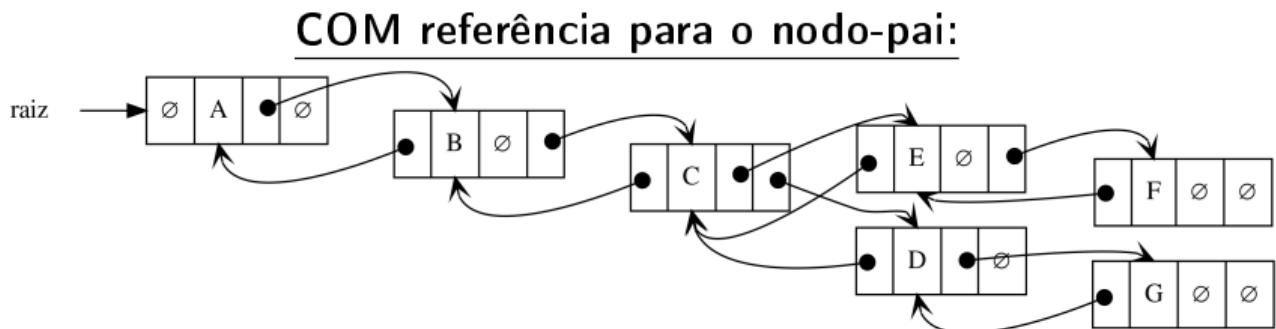
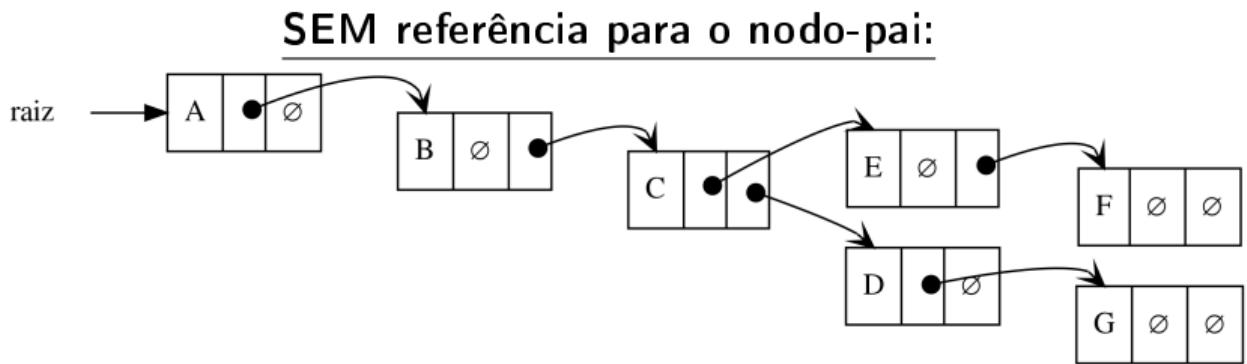
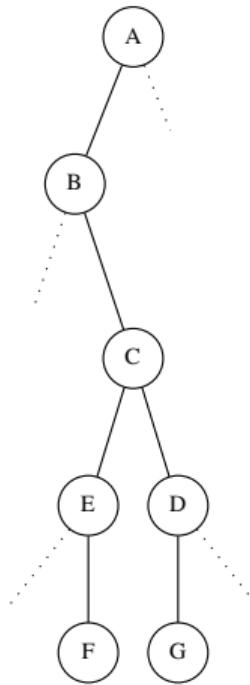
- Classe importante de árvores binárias
- Quando se quer representar as diferentes saídas que podem resultar a partir das respostas a um conjunto de perguntas do tipo sim ou não
- Cada nodo interno é associado com uma questão



# Árvore Binária

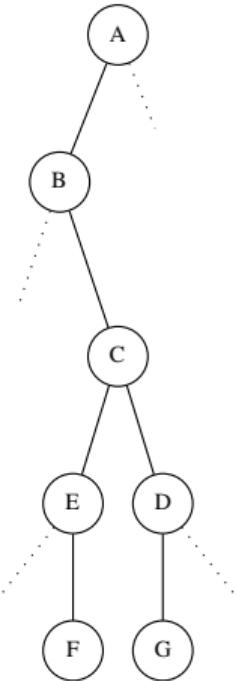
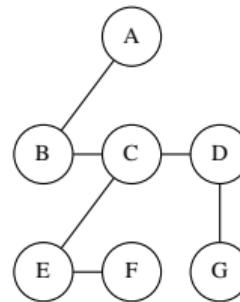
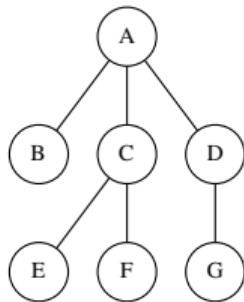
- Os nodos de uma árvore binária terão no mínimo:
  - A informação
  - Referência para o nodo da esquerda
  - Referência para o nodo da direita
- Também se pode usar referência para o nodo pai (o que facilita a “navegação”)
- Árvores binárias são fáceis de implementar, pois cada nodo tem no máximo 2 filhos

# Árvore Binária



# Árvore Binária

- Uma árvore qualquer pode ser transformada em árvore binária
- Passos para a transformação:
  - 1 Ligar os nodos irmãos
  - 2 Desligar a ligação do nodo pai com os filhos, exceto o primeiro filho
- Nodos de mesmo nível (irmãos) são encadeados à direita e nodos com nível maior (filhos) são encadeados à esquerda (iniciando pelo primeiro filho)



# Exercício

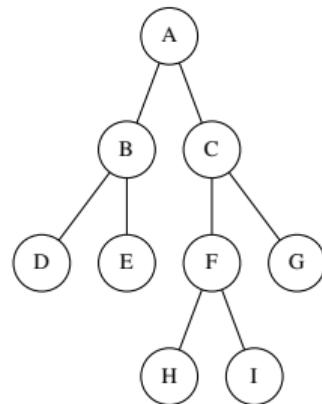
# Exercício 5

- 5 O código apresentado nas duas páginas seguintes (`exercicio05.cpp`), define um nodo (`struct Node`) para uma árvore binária. Este nodo armazena: a informação (`info`, um único caractere), uma referência para o nodo pai (`parent`) e referências para as subárvore da esquerda (`left`) e direita (`right`). Além destas informações, o nodo contém um construtor e um destrutor. Juntamente com a definição do nodo são fornecidas funções para mostrar a árvore no formato GraphViz e uma função `main()`, que cria a árvore ao lado (usando o código fornecido) e realiza alguns testes com as funções que você deverá implementar.

As funções que você deverá implementar são as seguintes:

- `int degree(Node *subtree)`: retorna o número de filhos (grau) de determinado nodo da árvore (parâmetro `subtree`);
- `int depth(Node *subtree)`: retorna o nível de determinado nodo (parâmetro `subtree`) dentro da árvore (o que pode ser feito navegando pela árvore usando as referências para o nodos-pai, e contando o número de nodos até alcançar a raiz);
- `int height(Node *subtree)`: retorna a altura da árvore/subárvore a partir de um nodo específico (parâmetro `subtree`) – deve ser implementado como uma função recursiva;
- `int size(Node *subtree)`: retorna o número de nodos que há na árvore a partir de determinado nodo (parâmetro `subtree`) – deve ser implementado como uma função recursiva;

Observação: os códigos usados neste exercício foram adaptados das soluções dos exercícios 3 e 4. As soluções, por outro lado, precisam ser desenvolvidas.



# Exercício 5 (continuação)

```
#include <iostream>
#include <sstream>

using namespace std;

struct Node {
    char info; Node *parent, *left, *right;
    Node(char i, Node *l = nullptr, Node *r = nullptr) {
        info = i; left = l; right = r; parent = nullptr;
        if (left != nullptr) left->parent = this;
        if (right != nullptr) right->parent = this;
        #ifdef DEBUG
        cerr << "+_Node(" << info << ") criado..." << endl;
        #endif
    }
    ~Node() {
        if (left != nullptr) delete left;
        if (right != nullptr) delete right;
        #ifdef DEBUG
        cerr << "-_Node(" << info << ") destruído..." << endl;
        #endif
    }
};

string strNode(Node *Node) {
    stringstream ss;
    if (Node->left != nullptr) ss << " _ " << Node->info << " _ _ " << Node->left->info << endl << strNode(Node->left);
    if (Node->right != nullptr) ss << " _ " << Node->info << " _ _ " << Node->right->info << endl << strNode(Node->right);
    return ss.str();
}

string strGraphViz(Node *root) {
    stringstream ss;
    ss << "graph TD\nArvoreBinária[" << endl << "Node[shape=circle]" << endl << strNode(root) << "}" << endl;
    return ss.str();
}
```

# Exercício 5 (continuação)

```

int degree(Node *subtree) { return 0; } // SUBSTITUIR/IMPLEMENTAR
int depth(Node *subtree) { return 0; } // SUBSTITUIR/IMPLEMENTAR
int height(Node *subtree) { return 0; } // SUBSTITUIR/IMPLEMENTAR
int size(Node *subtree) { return 0; } // SUBSTITUIR/IMPLEMENTAR

int main() {
    Node *d = new Node('D');
    Node *b = new Node('B', d, new Node('E'));
    Node *f = new Node('F', new Node('H'), new Node('I'));
    Node *c = new Node('C', f, new Node('G'));
    Node *root = new Node('A', b, c);
    cout << strGraphViz(root);
    cout << "degree(root):" << degree(root) << "[2]" << endl;
    cout << "degree(b):" << degree(b) << "[2]" << endl;
    cout << "degree(d):" << degree(d) << "[0]" << endl;
    cout << "depth(root):" << depth(root) << "[0]" << endl;
    cout << "depth(b):" << depth(b) << "[1]" << endl;
    cout << "depth(f):" << depth(f) << "[2]" << endl;
    cout << "size(root):" << size(root) << "[9]" << endl;
    cout << "size(b):" << size(b) << "[3]" << endl;
    cout << "size(c):" << size(c) << "[5]" << endl;
    cout << "height(root):" << height(root) << "[3]" << endl;
    cout << "height(b):" << height(b) << "[1]" << endl;
    cout << "height(c):" << height(c) << "[2]" << endl;
    delete root;
    return 0;
}

```

## Observações

## Sobre a Implementação do Construtor do Exercício 5

- Observe o código que foi usado no construtor da struct Node do exercício 5:

```
Node(char i, Node *l = nullptr, Node *r = nullptr) {
    info = i;    left = l;    right = r;    parent = nullptr;
    if ( left != nullptr ) left->parent = this;
    if ( right != nullptr ) right->parent = this;
}
```

- Quando um nodo é criado, o construtor recebe as subárvores que serão inseridas como seus “filhos”
- Como em cada nodo há uma referência para o nodo pai, este campo precisa ser atualizado nos nodos/subárvores filhos, o que é feito nas duas últimas linhas do construtor
- Para referenciar que os campos pai (parent) dos filhos (left e right) apontarão para o nodo que está sendo inicializado usa-se a autorreferência this

# Sobre a Solução do Exercício 5

- A solução do exercício 5 apresenta uma solução para o método `size()`, a princípio, mais fácil de entender:

```
int size(Node *subtree) {
    if (subtree == nullptr) return 0; // Somente pode ocorrer na primeira chamada,
                                    // ou seja, se a árvore estiver vazia
    int res = 1;
    if (subtree->left != nullptr) res += size(subtree->left);
    if (subtree->right != nullptr) res += size(subtree->right);
    return res;
}
```

- Veja uma solução alternativa e equivalente abaixo:

```
int size(Node *subtree) {
    if (subtree == nullptr) return 0;
    return 1 + size(subtree->left) + size(subtree->right);
}
```

- Analise as diferenças!

# Árvores Genéricas

# Definição

- Formalmente uma árvore  $T$  é definida como um conjunto de **nodos**, que armazenam elementos em relacionamentos **pai-filho** com as seguintes propriedades:
  - Se  $T$  não é vazia, ela tem um nodo especial chamado de **raiz** de  $T$  que não tem pai
  - Cada nodo  $v$  de  $T$  diferente da raiz tem um único nodo **pai**,  $w$ ; todo nodo com pai  $w$  é **filho** de  $w$

# Representações

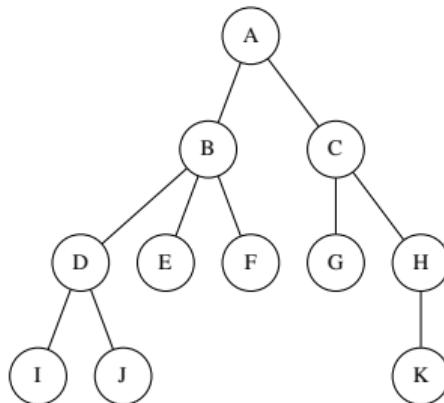
- Textual

```
T={A,{B,{D,{I},{J}},{E},{F}}, {C,{G},{H},{K}}}}
```

- GraphViz

```
graph arvore3 {
    node [shape=circle]
    A -- { B C }
    B -- { D E F }
    D -- { I J }
    C -- { G H }
    H -- { K }
}
```

- Visual/Gráfica



# Operações

- Alocar nodos
- Desalocar nodos e seus descendentes
- Obter a informação de um nodo
- Obter a referência do nodo-pai de um nodo
- Inserir nodos ou subárvores como filhos de outro nodo
- Obter o grau (número de filhos) de um nodo
- Obter a referência para determinado nodo-filho de um nodo
- Remover determinado nodo-filho de um nodo
- Verificar se um nodo é raiz, interno ou externo
- Obter o nível de determinado nodo
- Obter o número de nodos a partir de determinado nodo da árvore
- Determinar a altura de uma árvore
- Verificar se determinada informação aparece a partir de um nodo
- Localizar o nodo que armazena determinada informação
- Percorrer a árvore
- Obter representações da árvore ou de subárvores em formatos específicos
- etc.

# TAD para Árvores Genéricas

- A forma mais usual de implementação consiste no uso de **estruturas encadeadas** (alocação dinâmica)
- Cada nodo contém:
  - A informação
  - Uma referência para o nodo pai
  - Uma lista de referências para os nodos filhos (subárvores)
- Também é comum armazenar para uma árvore, o seu número total de nodos

# Lista de Referências para os Nodos Filhos

- Como um nodo pode ter número variável de filhos, é interessante trabalhar com uma lista dinamicamente expansível
- Para implementar esta lista dinamicamente expansível é possível usar o *container vector* da *Standard Template Library* da linguagem C++
- Exemplo de uso de vector:

```
#include <vector>

// ...

vector<int> v;      // Cria um vetor de inteiros vazio
v.push_back(10);    // Adiciona 10 no vetor -> v = { 10 }
v.push_back(20);    // Adiciona 20 no vetor -> v = { 10, 20 }
v.push_back(30);    // Adiciona 30 no vetor -> v = { 10, 20, 30 }
v.push_back(40);    // Adiciona 40 no vetor -> v = { 10, 20, 30, 40 }
v.erase( v.begin() + 1 ); // Remove o elemento de índice 1 do vetor -> v = { 10, 30, 40 }
v.pop_back();        // Remove o último elemento do vetor -> v = { 10, 30 }
for (int i=0; i<v.size(); ++i) // mostra o vetor
    cout << v[i] << endl;      // v é usado como se fosse um arranjo
```

# TAD para Árvores Genéricas

- É possível declarar um TAD para:
  - A **estrutura de dados**, que terá uma classe interna para o nodo, determinando automaticamente onde cada informação será inserida (chamadas para acrescentar e remover algum dado NÃO especificam onde a informação será inserida; nodos e referências que formam a árvore permanecem encapsulados)
  - O **nodo**, deixando o gerenciamento da árvore a cargo da própria aplicação (é necessário fornecer chamadas para acrescentar nodos-filho em determinado nodo, e também para remover nodos-filhos, etc.)

# Exemplo de TAD para uma Árvores de Inteiros

```
#include <vector>
using namespace std;

#ifndef _INTTREE_HPP
#define _INTTREE_HPP
class IntTree {
private:
    struct Node {
        int info;
        Node *parent;
        vector<Node *> child;
        Node(int i) { info = i; parent = nullptr; }
        ~Node() { for (int i=0; i<child.size(); ++i) delete child[i]; }
    };
    Node *root;
public:
    IntTree(); // Cria a árvore de inteiros vazia (root = nullptr)
    ~IntTree(); // Desaloca os nodos da árvore de inteiros (clear());
    bool add(int node, int info); // Adiciona info como filho de node
    int getRoot(); // Retorna o elemento armazenado na raiz
    void setRoot(int info); // Altera o elemento armazenado na raiz
    int getParent(int info); // Retorna o pai do nodo que contém info
    int removeBranch(int info); // Remove o elemento que contém info e seus filhos
    bool contains(int info); // Retorna true se a árvore contém o elemento info
    bool isInternal(int info); // Retorna true se o elemento info está armazenado em um nodo interno
    bool isExternal(int info); // Retorna true se o elemento info está armazenado em um nodo externo
    bool isRoot(int e); // Retorna true se o elemento info está armazenado na raiz
    bool isEmpty(); // Retorna true se a árvore está vazia (return root==nullptr;)
    int size(); // Retorna o número de elementos armazenados na árvore
    void clear(); // Remove todos os elementos da árvore (delete root;)
    vector<int> preorder(); // Retorna um vetor com todos os elementos da árvore na ordem pré-fixada
    vector<int> postorder(); // Retorna um vetor com todos os elementos da árvore na ordem pos-fixada
    vector<int> levelorder(); // Retorna um vetor com todos os elementos da árvore na ordem em largura
};
#endif
```

# Exemplo de TAD para um Nodo de Árvore de Caracteres

```

#ifndef _NODECHARTREE_HPP
#define _NODECHARTREE_HPP
#include <vector>
#include <string>

using namespace std;

class NodeCharTree {
private:
    char info;
    NodeCharTree *parent;
    vector<NodeCharTree *> childs;
    string strGraphVizNode(NodeCharTree const *node) const; // Imprime as conexões de um nodo com seus filhos em GraphViz
public:
    NodeCharTree(char i);                                // Cria um nodo com o caractere i, e sem pai
    ~NodeCharTree();                                     // Destroi o nodo atual e seus descendentes
    char getInfo() const;                               // Retorna o caractere armazenado no nodo atual
    void setInfo(char i);                               // Altera o caractere armazenado no nodo atual
    NodeCharTree *getParent() const;                   // Retorna a referência para o nodo-pai do nodo atual
    NodeCharTree *getChild(int index) const;           // Retorna a referência para o nodo-filho de índice index ou nullptr
    bool isRoot() const;                               // Retorna true se o nodo for raiz
    bool isInternal() const;                           // Retorna true se o nodo for interno
    bool isExternal() const;                           // Retorna true se o nodo for externo
    int degree() const;                               // Retorna o grau (número de filhos) do nodo atual
    int depth() const;                                // Retorna o nível do nodo atual
    int height() const;                               // Retorna a altura da subárvore/árvore a partir do nodo atual
    int size() const;                                 // Retorna o número de nodos a partir do nodo atual
    void addSubtree(NodeCharTree *subtree);           // Adiciona uma subárvore como filho do nodo atual
    bool removeSubtree(NodeCharTree *subtree);         // Remove a subárvore dentre os filhos do atual (true indica sucesso)
    bool contains(char i) const;                      // Retorna true se o nodo atual ou algum de seus descendentes contiverem i
    NodeCharTree const *find(char i) const;            // Retorna a referência para o nodo, a partir do atual, que contém i ou nullptr
    string strGraphViz() const;                       // Gera uma representação da árvore a partir do nodo atual no formato GraphViz
};

#endif

```

# Criando uma Árvore de Caracteres (Passo 1)

## Código:

```
// Criação de nodos
NodeCharTree *b = new NodeCharTree('B');
NodeCharTree *a = new NodeCharTree('A');
NodeCharTree *d = new NodeCharTree('D');
NodeCharTree *f = new NodeCharTree('F');
NodeCharTree *c = new NodeCharTree('C');
NodeCharTree *e = new NodeCharTree('E');
```



## Memória:

```
b = ref1 --> |info='B'|parent=nullptr|childs={}
a = ref2 --> |info='A'|parent=nullptr|childs={}
d = ref3 --> |info='D'|parent=nullptr|childs={}
f = ref4 --> |info='F'|parent=nullptr|childs={}
c = ref5 --> |info='C'|parent=nullptr|childs={}
e = ref6 --> |info='E'|parent=nullptr|childs{}
```

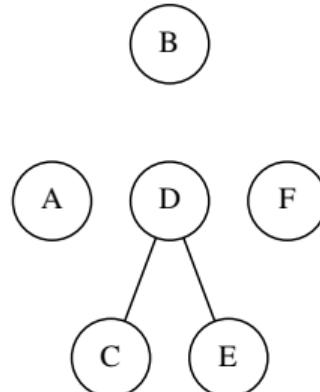
# Criando uma Árvore de Caracteres (Passo 2)

## Código:

```
// Adicionando os filhos de D
d->addSubtree( c );
d->addSubtree( e );
```

## Memória:

```
b = ref1 --> |info='B'|parent=nullptr|childs={}
a = ref2 --> |info='A'|parent=nullptr|childs={}
d = ref3 --> |info='D'|parent=nullptr|childs={ref5,ref6}
f = ref4 --> |info='F'|parent=nullptr|childs={}
c = ref5 --> |info='C'|parent=ref3  |childs={}
e = ref6 --> |info='E'|parent=ref3  |childs={}|
```



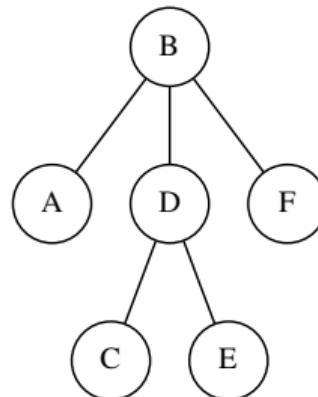
# Criando uma Árvore de Caracteres (Passo Final)

## Código:

```
// Adicionando os filhos de B
b->addSubtree( a );
b->addSubtree( d );
b->addSubtree( f );
```

## Memória:

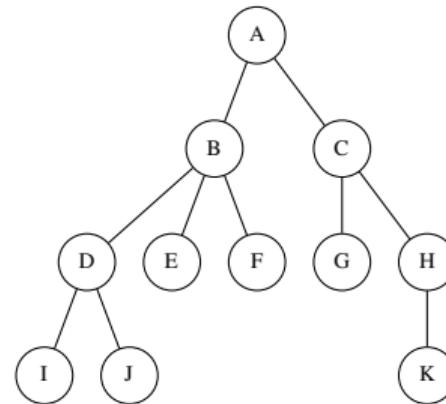
```
b = ref1 --> |info='B'|parent=nullptr|childs={ref2,ref3,ref4}|
a = ref2 --> |info='A'|parent=ref1 |childs={}
d = ref3 --> |info='D'|parent=ref1 |childs={ref5,ref6}|
f = ref4 --> |info='F'|parent=ref1 |childs={}
c = ref5 --> |info='C'|parent=ref3 |childs={}
e = ref6 --> |info='E'|parent=ref3 |childs{}
```



## Exercício

## Exercício 6

- 6 Nas páginas a seguir, há 3 arquivos, respectivamente, com: a definição da classe NodeCharTree (arquivo `exercicio06/NodeCharTree.hpp` e duas aplicações para testar a implementação dessa classe (arquivos `exercicio06/app1.cpp` e `exercicio06/app2.cpp`). Ambas as aplicações criam a árvore abaixo e testam a implementação da classe. Você encontra estes arquivos, juntamente com resultados esperados para a sua execução, no arquivo `src.zip` (disponível no Moodle da disciplina). A descrição do que cada método faz encontra-se nos comentários da definição da classe. **Implemente os métodos da classe no arquivo `exercicio06/NodeCharTree.cpp`.**



# Exercício 6: exercício06/NodeCharTree.hpp

```
#ifndef _NODECHARTREE_HPP
#define _NODECHARTREE_HPP
#include <vector>
#include <string>

using namespace std;

class NodeCharTree {
private:
    char info;
    NodeCharTree *parent;
    vector<NodeCharTree *> childs;
    string strGraphVizNode(NodeCharTree const *node) const; // Imprime as conexões de um nodo com seus filhos em GraphViz
public:
    NodeCharTree(char i);                                // Cria um nodo com o caractere i, e sem pai
    ~NodeCharTree();                                     // Destroi o nodo atual e seus descendentes
    char getInfo() const;                               // Retorna o caractere armazenado no nodo atual
    void setInfo(char i);                               // Altera o caractere armazenado no nodo atual
    NodeCharTree *getParent() const;                   // Retorna a referência para o nodo-pai do nodo atual
    NodeCharTree *getChild(int index) const;           // Retorna a referência para o nodo-filho de índice index ou nullptr
    bool isRoot() const;                               // Retorna true se o nodo for raiz
    bool isInternal() const;                           // Retorna true se o nodo for interno
    bool isExternal() const;                           // Retorna true se o nodo for externo
    int degree() const;                               // Retorna o grau (número de filhos) do nodo atual
    int depth() const;                                // Retorna o nível do nodo atual
    int height() const;                               // Retorna a altura da subárvore/árvore a partir do nodo atual
    int size() const;                                 // Retorna o número de nodos a partir do nodo atual
    void addSubtree(NodeCharTree *subtree);           // Adiciona uma subárvore como filho do nodo atual
    bool removeSubtree(NodeCharTree *subtree);         // Remove a subárvore dentre os filhos do atual (true indica sucesso)
    bool contains(char i) const;                      // Retorna true se o nodo atual ou algum de seus descendentes contiverem i
    NodeCharTree const *find(char i) const;            // Retorna a referência para o nodo, a partir do atual, que contém i ou nullptr
    string strGraphViz() const;                       // Gera uma representação da árvore a partir do nodo atual no formato GraphViz
};

#endif
```

# Exercício 6: exercício06/app1.cpp

```
#include <iostream>
#include "NodeCharTree.hpp"

int main() {
    NodeCharTree *i = new NodeCharTree('I');
    NodeCharTree *j = new NodeCharTree('J');
    NodeCharTree *d = new NodeCharTree('D');
    d->addSubtree(i);
    d->addSubtree(j);
    NodeCharTree *e = new NodeCharTree('E');
    NodeCharTree *f = new NodeCharTree('F');
    NodeCharTree *b = new NodeCharTree('B');
    b->addSubtree(d);
    b->addSubtree(e);
    b->addSubtree(f);
    NodeCharTree *k = new NodeCharTree('K');
    NodeCharTree *h = new NodeCharTree('H');
    h->addSubtree(k);
    NodeCharTree *g = new NodeCharTree('G');
    NodeCharTree *c = new NodeCharTree('C');
    c->addSubtree(g);
    c->addSubtree(h);
    NodeCharTree *root = new NodeCharTree('A');
    root->addSubtree(b);
    root->addSubtree(c);
    cout << root->strGraphViz() << endl;
    delete root;
    return 0;
}
```

## Exercício 6: exercício06/app2.cpp

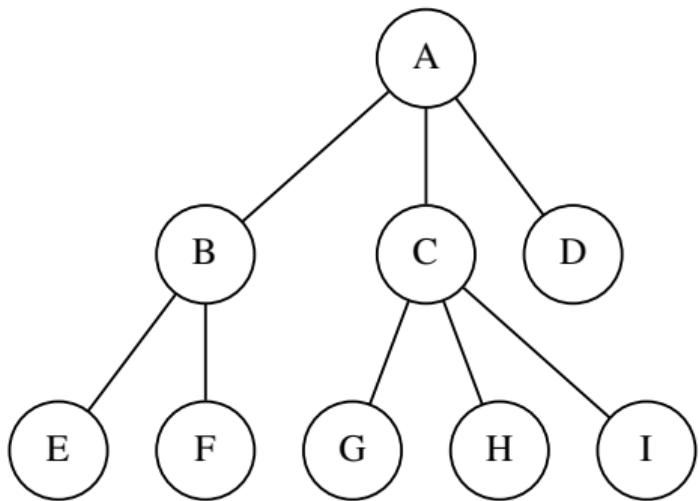
# Caminhamento em Árvores Genéricas

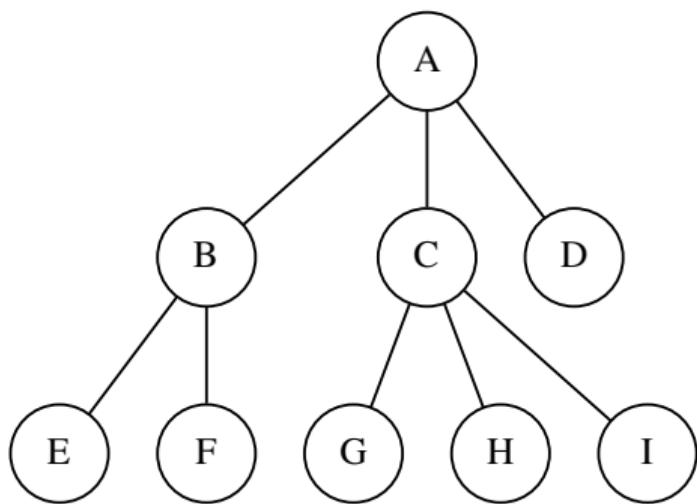
# Caminhamento

- Caminhamento é a forma como vamos percorrer as árvores, ou seja, a ordem em que vamos visitar seus nodos
- Existem vários tipos de caminhamento que permitem varrer a árvore de forma sistemática visitando cada nodo uma única vez
- Para uma árvore genérica é comum o uso de três formas de caminhamento: **pré-ordem**, **pós-ordem** e **em largura**

# Pré-ordem (*pre-order*)

- Do inglês *pre-order traversal*
- O nodo é visitado antes de seus descendentes
- É definido recursivamente como:
  - Visite o nodo
  - Realize um percurso em pré-ordem para cada uma das subárvore do nodo
- Exemplo de aplicação:
  - Impressão de um documento estruturado em capítulos

Pré-ordem (*pre-order*)

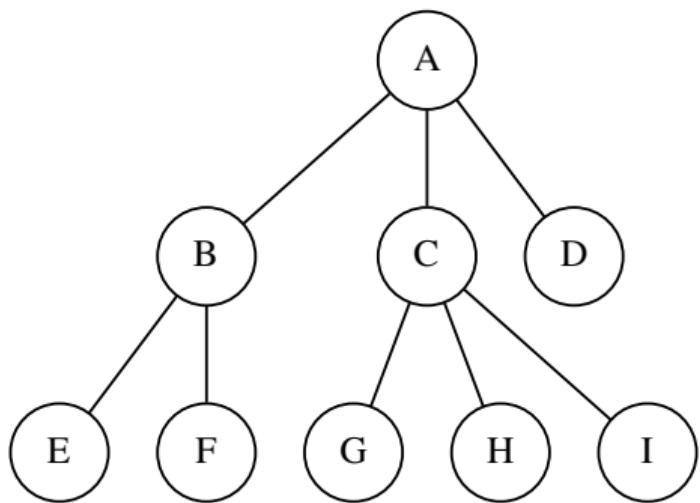
Pré-ordem (*pre-order*)

- 1 A
- 2 B
- 3 E
- 4 F
- 5 C
- 6 G
- 7 H
- 8 I
- 9 D

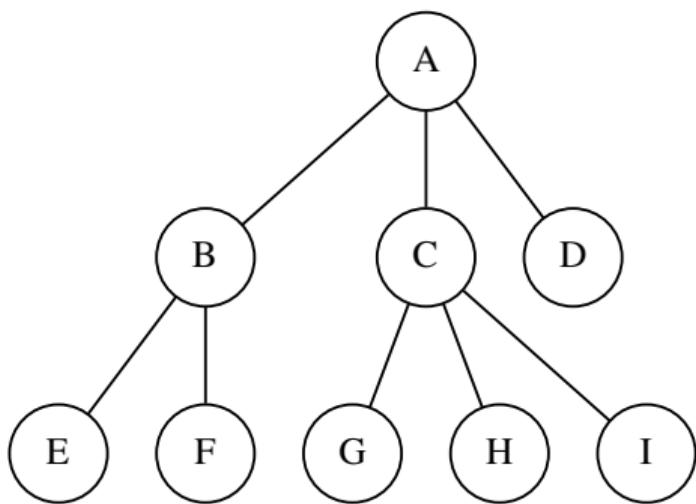
# Pós-ordem (*post-order*)

- Do inglês *post-order traversal*
- O nodo é visitado depois de seus descendentes
- É definido recursivamente como:
  - Realize um percurso em pós-ordem para cada uma das subárvore do nodo
  - Visite o nodo
- Exemplos de aplicação:
  - Calcular o total de espaço em disco ocupado por arquivos em um sistema de diretórios

# Pós-ordem (*post-order*)



# Pós-ordem (*post-order*)

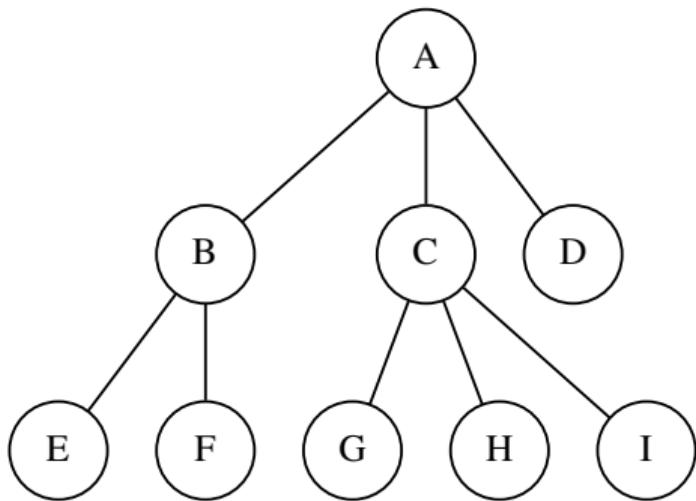


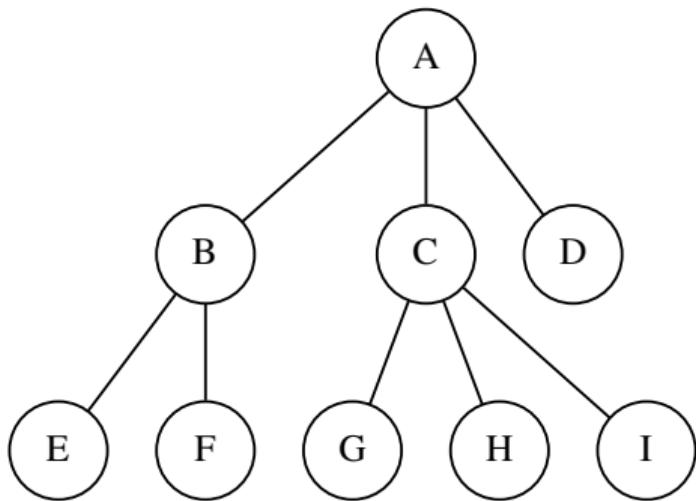
- 1 E
- 2 F
- 3 B
- 4 G
- 5 H
- 6 I
- 7 C
- 8 D
- 9 A

## Em Largura (*level-order*)

- Visita os nodos na ordem dos níveis da árvore, da esquerda para a direita
  - Visite os nodos de nível 0
  - Visite os nodos de nível 1
  - ...
- Algoritmo não-recursivo:
  - Inserir nodo raiz em uma fila
  - Repetir até que a fila esteja vazia
    - Remover o nodo da fila
    - Visitar o nodo atual
    - Inserir na fila, em ordem, cada subárvore não vazia do nodo atual

# Em Largura (*level-order*)



Em Largura (*level-order*)

- 1 A
- 2 B
- 3 C
- 4 D
- 5 E
- 6 F
- 7 G
- 8 H
- 9 I

## Observação

## Templates em C++

- Até aqui foram implementadas **estruturas de dados** diferentes para armazenar **diferentes tipos de informação** (int, char, string, etc.), o que acaba gerando uma grande quantidade de **código repetido** ou no mínimo parecido
- É possível em C++, criar um *template* para determinada estrutura de dados, de forma que a mesma implementação possa ser aproveitada para diferentes tipos
- Assim, em vez de uma classe NodeCharTree, teríamos a seguinte classe genérica:

```
template <typename T>
class NodeTree {
private:
    T info;
    NodeTree *parent;
    vector<NodeTree *> childs;
    // ...
```

- Um nodo poderia então ser criado usando-se, por exemplo:

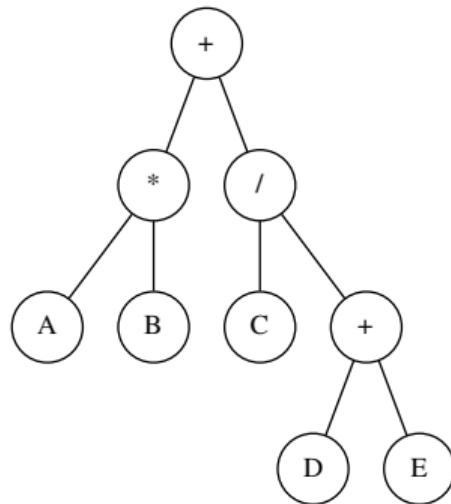
```
NodeTree<char> *a1 = new NodeTree<char>('A');
NodeTree<string> *a2 = new NodeTree<string>("raiz");
```

- O material da disciplina de Programação Orientada a Objetos contém maiores detalhes

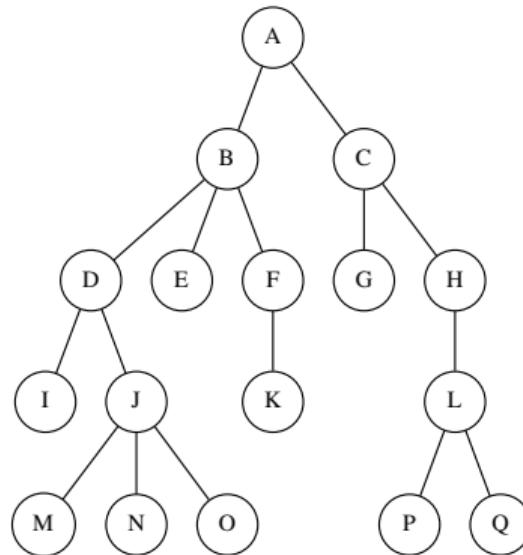
# Exercícios

## Exercício 7

- 7 Para as árvores a seguir, mostrar a ordem que os elementos serão apresentados para cada caminhamento (pré-ordem, pós-ordem e em largura).
- a)



b)

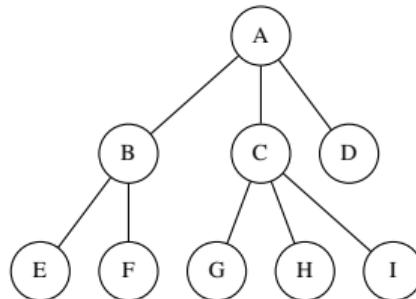


## Exercício 8

- 8 Considerando a classe `NodeCharTree` (resposta do Exercício 6), crie para esta classe três novos métodos que implementam os três tipos de caminhamento vistos até aqui (pré-ordem, pós-ordem e em largura). Estes caminhamentos devem ser implementados em métodos que geram uma cadeia de caracteres com o conteúdo dos nodos, na respectiva ordem do caminhamento, cada nodo em uma linha separada. Em `NodeCharTree.hpp`, acrescente:

```
string preorder() const; // Gera uma cadeia de caracteres (um nodo por linha) em pré-ordem
string postorder() const; // Gera uma cadeia de caracteres (um nodo por linha) em pós-ordem
string levelorder() const; // Gera uma cadeia de caracteres (um nodo por linha) em largura
```

Implemente estes métodos em `NodeCharTree.cpp` e use o programa da próxima página para testar a sua implementação. Este programa cria a árvore abaixo e mostra, para esta árvore, os resultados dos métodos a serem implementados.



## Exercício 8: exercício08/app.cpp

```
#include <iostream>
#include <iomanip>
#include "NodeCharTree.hpp"

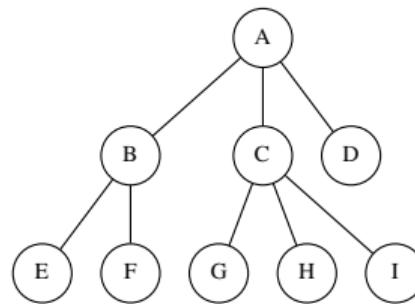
int main() {
    NodeCharTree *b = new NodeCharTree('B');
    b->addSubtree( new NodeCharTree('E') );
    b->addSubtree( new NodeCharTree('F') );
    NodeCharTree *c = new NodeCharTree('C');
    c->addSubtree( new NodeCharTree('G') );
    c->addSubtree( new NodeCharTree('H') );
    c->addSubtree( new NodeCharTree('I') );
    NodeCharTree *root = new NodeCharTree('A');
    root ->addSubtree(b);
    root ->addSubtree(c);
    root ->addSubtree( new NodeCharTree('D') );
    cout << "pre-order: " << (root->preorder() == "A\nB\nE\nF\nC\nG\nH\nI\nD\n" ? "OK" : "ERROR") << endl;
    cout << "post-order: " << (root->postorder() == "E\nF\nB\nG\nH\nI\nC\nD\nA\n" ? "OK" : "ERROR") << endl;
    cout << "level-order: " << (root->levelorder() == "A\nB\nC\nD\nE\nF\nG\nH\nI\n" ? "OK" : "ERROR") << endl;
    delete root;
    return 0;
}
```

## Exercício 9 (DESAFIO)

- 9 Considerando a classe `NodeCharTree` (resposta do Exercício 8), crie uma versão genérica desta classe usando *templates*. A declaração da classe deve ser colocada em um arquivo chamado `NodeTree.hpp`. A **parte inicial** do arquivo `NodeTree.hpp` e a versão do arquivo `app.cpp` que usa esta classe genérica encontram-se nas páginas a seguir.

Lembre-se de que a implementação dos métodos da classe genérica `NodeTree` deve ser colocada no próprio arquivo `NodeTree.hpp`.

Por sua vez, o arquivo `app.cpp` cria a árvore abaixo e mostra os caminhamentos em pré-ordem, pós-ordem e em largura para esta árvore.



# Exercício 9 (DESAFIO): exercício09/NodeTree.hpp (índio)

```

#ifndef _NODETREE_HPP
#define _NODETREE_HPP
#ifndef DEBUG
#include <iostream>
#endif
#include <vector>
#include <sstream>

using namespace std;

template <typename T>
class NodeTree {
private:
    T info;    NodeTree *parent;    vector<NodeTree *> childs;
    string strGraphVizNode(NodeTree const *node) const; // Imprime as conexões de um nodo com seus filhos em GraphViz
public:
    NodeTree(T const &i);                // Cria um nodo com o caractere i, e sem pai
    ~NodeTree();                         // Destroi o nodo atual e seus descendentes
    T getInfo() const;                  // Retorna o caractere armazenado no nodo atual
    void setInfo(T &i);                // Altera o caractere armazenado no nodo atual
    NodeTree *getParent() const;        // Retorna a referência para o nodo-pai do nodo atual
    NodeTree *getChild(int index) const; // Retorna a referência para o nodo-filho de índice index ou nullptr
    bool isRoot() const;                // Retorna true se o nodo for raiz
    bool isInternal() const;            // Retorna true se o nodo for interno
    bool isExternal() const;            // Retorna true se o nodo for externo
    int degree() const;                // Retorna o grau (número de filhos) do nodo atual
    int depth() const;                 // Retorna o nível do nodo atual
    int height() const;                // Retorna a altura da subárvore/árvore a partir do nodo atual
    int size() const;                  // Retorna o número de nodos a partir do nodo atual
    void addSubtree(NodeTree *subtree); // Adiciona uma subárvore como filho do nodo atual
    bool removeSubtree(NodeTree *subtree); // Remove a subárvore dentro os filhos do atual (true indica sucesso)
    bool contains(T &i) const;          // Retorna true se o nodo atual ou algum de seus descendentes contiverem i
    NodeTree const *find(T const &i) const; // Retorna a referência para o nodo, a partir do atual, que contém i ou nullptr
    string strGraphViz() const;         // Gera uma representação da árvore a partir do nodo atual no formato GraphViz
    string preorder() const;            // Gera uma cadeia de caracteres (um nodo por linha) em pré-ordem
    string postorder() const;           // Gera uma cadeia de caracteres (um nodo por linha) em pós-ordem
    string levelorder() const;          // Gera uma cadeia de caracteres (um nodo por linha) em largura
};
```

## Exercício 9 (DESAFIO): exercício09/app.cpp

```
#include <iostream>
#include <iomanip>
#include "NodeTree.hpp"

int main() {
    NodeTree<char> *b = new NodeTree<char>('B');
    b->addSubtree( new NodeTree<char>('E') );
    b->addSubtree( new NodeTree<char>('F') );
    NodeTree<char> *c = new NodeTree<char>('C');
    c->addSubtree( new NodeTree<char>('G') );
    c->addSubtree( new NodeTree<char>('H') );
    c->addSubtree( new NodeTree<char>('I') );
    NodeTree<char> *root = new NodeTree<char>('A');
    root->addSubtree(b);
    root->addSubtree(c);
    root->addSubtree( new NodeTree<char>('D') );
    cout << "pre-order: " << (root->preorder() == "A\nB\nE\nF\nC\nG\nH\nI\nD\n" ? "OK" : "ERROR") << endl;
    cout << "post-order: " << (root->postorder() == "E\nF\nB\nG\nH\nI\nC\nD\nA\n" ? "OK" : "ERROR") << endl;
    cout << "level-order: " << (root->levelorder() == "A\nB\nC\nD\nE\nF\nG\nH\nI\n" ? "OK" : "ERROR") << endl;
    delete root;
    return 0;
}
```

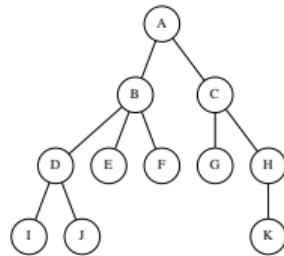
# Exercício 10 (DESAFIO)

- 10 Comandos como o `pstree` do Unix, na versão para GNU/Linux, por exemplo, conseguem imprimir a árvore hierárquica de processos (que é uma árvore genérica) de uma forma visualmente agradável, como mostrado na figura ao lado.

Implemente na classe `NodeCharTree` (resposta do Exercício 8) uma funcionalidade semelhante. Em `NodeCharTree.hpp`, acrescente:

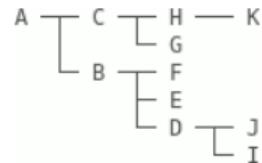
```
string strTree() const;
```

Este método deve gerar, por exemplo, uma cadeia de caracteres equivalente à representação ao lado para a árvore abaixo.



```

systemd
└── ModemManager—2*[{ModemManager}]
   └── NetworkManager—2*[dhclient]
      └── 2*[{NetworkManager}]
   └── accounts-daemon—2*[{accounts-daemon}]
   └── agent—2*[{agent}]
   └── getty
   └── alsactl
   └── avahi-daemon—avahi-daemon
   └── bluetoothd
   └── colord—2*[{colord}]
   └── containerd—14*[{containerd}]
   └── core—16*[{core}]
   └── core
      └── wsatspi
         └── 14*[{core}]
  
```



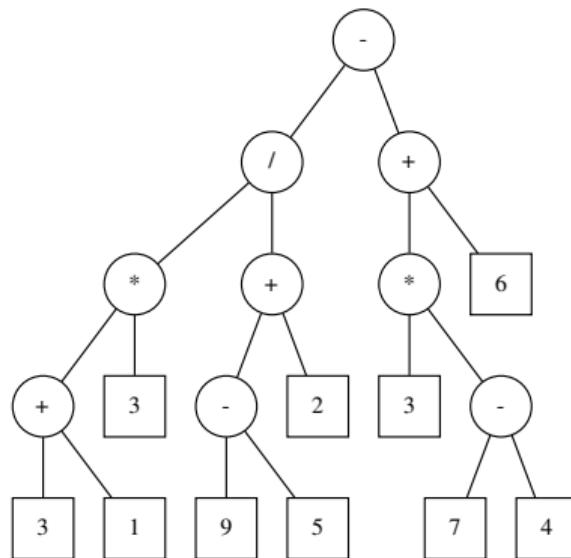
# Árvores Binárias

# Árvores Binárias

- Uma árvore binária é uma árvore ordenada com as seguintes propriedades:
  - Todos os nodos tem no máximo dois filhos
  - Cada nodo filho é rotulado como sendo um **filho da esquerda** ou um **filho da direita**
  - O filho da esquerda precede o filho da direita na ordenação dos filhos de um nodo
- Subárvores:
  - O filho da esquerda de um nodo interno  $v$  é chamado de **subárvore da esquerda**
  - O filho da direita de um nodo interno  $v$  é chamado de **subárvore da direita**
- Árvore binária própria (ou cheia)
  - Cada nodo tem 0 ou 2 filhos
- As estruturas de árvores binárias são muito utilizadas na computação

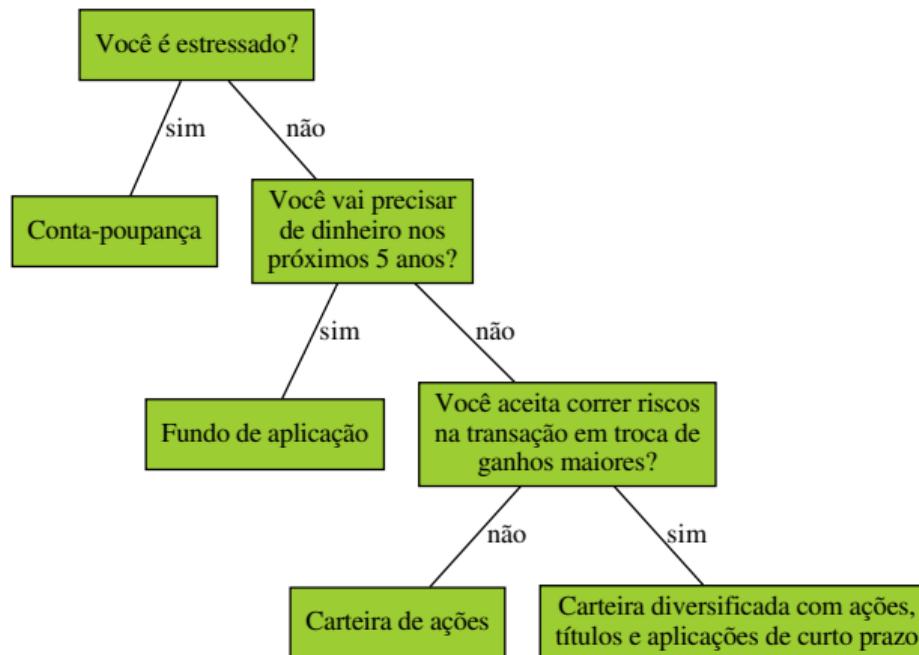
## Exemplo: Árvore para uma Expressão Aritmética

- Nodos externos são associados com variáveis e constantes
- Nodos internos são associados com um operador
- Exemplo:  $((3+1)*3)/((9-5)+2))-(3*(7-4)+6)$



## Exemplo: Árvore de Decisão

- Árvore de decisão que fornece recomendações para um investidor



# TAD para Árvores Binárias

- Forma mais usual de implementação é usando estruturas encadeadas, sendo que cada nó contém:
  - A informação
  - Uma referência para o nodo pai
  - Uma referência para a subárvore da esquerda
  - Uma referência para a subárvore da direita
- Também é comum armazenar para uma árvore, o seu número total de nodos
- Da mesma forma que com árvores genéricas, é possível criar uma TAD para árvores binárias para:
  - A **estrutura de dados**, que terá uma classe interna para o nodo, determinando automaticamente onde cada informação será inserida
  - O **nodo**, deixando o gerenciamento da árvore a cargo da própria aplicação

# TAD para Árvores Binárias

- Para manipular uma árvore implementada como um TAD (ou seja, TAD para a estrutura de dados), pode-se ter métodos para:
  - Criar a árvore
  - Inserir nodos na árvore
  - Pesquisar nodos na árvore
  - Excluir nodos da árvore
  - Determinar a altura da árvore e o nível de um nodo
  - Caminhar em uma árvore, visitando seus nodos
  - ...
- Neste caso, o TAD contém uma classe interna para os nodos e armazena uma referência para a raiz da árvore e o número de elementos já inseridos

# Exemplo de TAD para uma Árvores Binária de Inteiros

```

class IntBTree {
private:
    struct Node {
        int info;
        Node *parent, *left, *right;
        Node(int i) { info = i; parent = left = right = nullptr; }
        ~Node() { delete left; delete right; }
    };
    Node *root;
public:
    IntBTree(); // Cria a árvore de inteiros vazia (root = nullptr)
    ~IntBTree(); // Desaloca os nodos da árvore de inteiros (clear());
    bool addRoot(int info); // Adiciona info como raiz (false se a árvore não está vazia)
    int getRoot(); // Retorna o elemento armazenado na raiz
    void setRoot(int info); // Altera o elemento armazenado na raiz
    bool addLeft(int node, int info); // Adiciona info como filho da esquerda de node (false se node não existir)
    bool addRight(int node, int info); // Adiciona info como filho da direita de node (false se node não existir)
    bool hasLeft(int info); // Retorna true se info possuir subárvore na esquerda
    bool hasRight(int info); // Retorna true se info possuir subárvore na direita
    int getLeft(int info); // Retorna o filho à esquerda de info
    int getRight(int info); // Retorna o filho à direita de info
    int getParent(int info); // Retorna o pai do nodo que contém info
    bool isRoot(int e); // Retorna true se o elemento info está armazenado na raiz
    bool isInternal(int info); // Retorna true se o elemento info está armazenado em um nodo interno
    bool isExternal(int info); // Retorna true se o elemento info está armazenado em um nodo externo
    bool isEmpty(); // Retorna true se a árvore está vazia (return root==nullptr;)
    bool contains(int info); // Retorna true se a árvore contém o elemento info
    int removeBranch(int info); // Remove o elemento que contém info e seus filhos
    int size(); // Retorna o número de elementos armazenados na árvore
    void clear(); // Remove todos os elementos da árvore (delete root;)
    vector<int> preorder(); // Retorna um vetor com todos os elementos da árvore na ordem pré-fixada
    vector<int> postorder(); // Retorna um vetor com todos os elementos da árvore na ordem pos-fixada
    vector<int> inorder(); // Retorna um vetor com todos os elementos da árvore na ordem in-fixada
    vector<int> levelorder(); // Retorna um vetor com todos os elementos da árvore na ordem em largura
};

```

# Exemplo de TAD para um Nodo de Árvore Binária Genérica

```

template <typename T>
class NodeBT {
private:
    T info;      NodeBT *parent, *left, *right;
    string strGraphVizNode(NodeBT const *node) const; // Imprime as conexões de um nodo com seus filhos em GraphViz
public:
    NodeBT(T const &i);                                // Cria um nodo com o caractere i, e sem pai
    ~NodeBT();                                         // Destroi o nodo atual e seus descendentes
    T getInfo() const;                                 // Retorna o caractere armazenado no nodo atual
    void setInfo(T &i);                               // Altera o caractere armazenado no nodo atual
    NodeBT *getParent() const;                         // Retorna a referência para o nodo-pai do nodo atual
    NodeBT *getLeft() const;                           // Retorna a referência para o nodo-filho/subárvore da esquerda
    NodeBT *getRight() const;                          // Retorna a referência para o nodo-filho/subárvore da direita
    void setLeft(NodeBT *subtree);                    // Adiciona uma subárvore como subárvore da esquerda do nodo atual
    void setRight(NodeBT *subtree);                   // Adiciona uma subárvore como subárvore da direita do nodo atual
    void removeLeft();                                // Remove a subárvore da esquerda (desalocando-a, se necessário)
    void removeRight();                               // Remove a subárvore da direita (desalocando-a, se necessário)
    bool isRoot() const;                             // Retorna true se o nodo for raiz
    bool isInternal() const;                          // Retorna true se o nodo for interno
    bool isExternal() const;                          // Retorna true se o nodo for externo
    int degree() const;                             // Retorna o grau (número de filhos) do nodo atual
    int depth() const;                            // Retorna o nível do nodo atual
    int height() const;                           // Retorna a altura da subárvore/árvore a partir do nodo atual
    int size() const;                             // Retorna o número de nodos a partir do nodo atual
    bool contains(T &i) const;                     // Retorna true se o nodo atual ou algum de seus descendentes contiverem i
    NodeBT const *find(T const &i) const;           // Retorna a referência para o nodo, a partir do atual, que contém i ou nullptr
    string strGraphViz() const;                     // Gera uma representação da árvore a partir do nodo atual no formato GraphViz
    string preorder() const;                        // Gera uma cadeia de caracteres (um nodo por linha) em pré-ordem
    string postorder() const;                       // Gera uma cadeia de caracteres (um nodo por linha) em pós-ordem
    string inorder() const;                         // Gera uma cadeia de caracteres (um nodo por linha) em in-ordem
    string levelorder() const;                      // Gera uma cadeia de caracteres (um nodo por linha) em largura
};

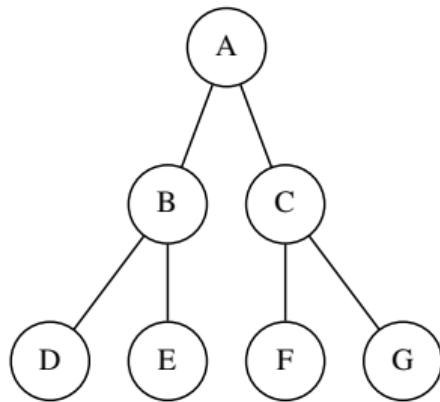
```

# Caminhamento

- Existem essencialmente duas formas diferentes de caminhamento:
  - Percurso em profundidade
    - Percurso pré-fixado ou em pré-ordem
    - Percurso pós-fixado ou em pós-ordem
    - Percurso central ou em ordem central
  - Percurso em largura

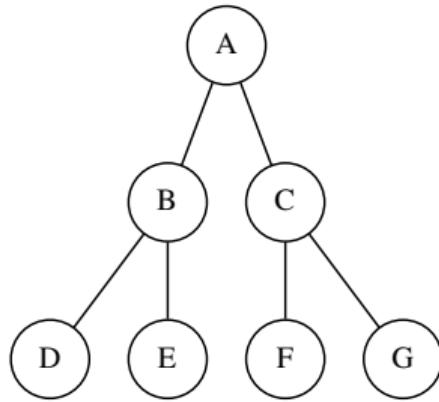
# Caminhamento Pré-fixado

- O nodo é visitado antes de seus descendentes
- Segue a ordem:
  - Visita Raiz
  - Percorre subárvore da esquerda
  - Percorre subárvore da direita



# Caminhamento Pré-fixado

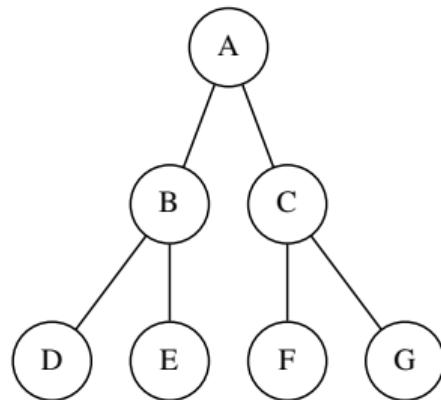
- O nodo é visitado antes de seus descendentes
- Segue a ordem:
  - Visita Raiz
  - Percorre subárvore da esquerda
  - Percorre subárvore da direita



- ① A
- ② B
- ③ D
- ④ E
- ⑤ C
- ⑥ F
- ⑦ G

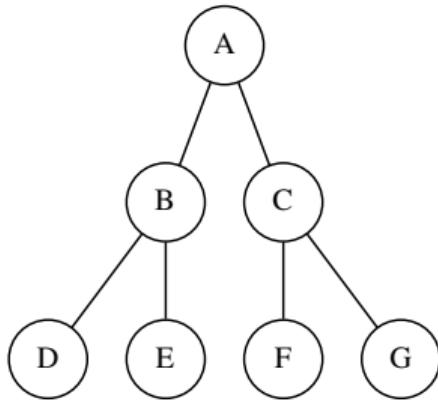
# Caminhamento Pós-fixado

- O nodo é visitado depois de seus descendentes
- Segue a ordem:
  - Percorre subárvore da esquerda
  - Percorre subárvore da direita
  - Visita Raiz



# Caminhamento Pós-fixado

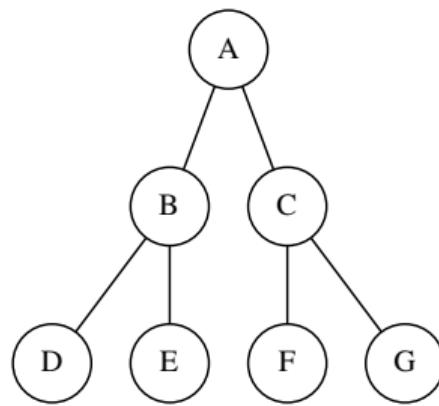
- O nodo é visitado depois de seus descendentes
- Segue a ordem:
  - Percorre subárvore da esquerda
  - Percorre subárvore da direita
  - Visita Raiz



- ① D
- ② E
- ③ B
- ④ F
- ⑤ G
- ⑥ C
- ⑦ A

# Caminhamento Central (*In-order*)

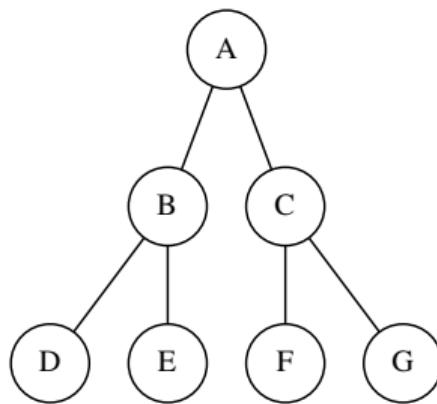
- Segue a ordem:
  - Percorre subárvore da esquerda
  - Visita Raiz
  - Percorre subárvore da direita



# Caminhamento Central (*In-order*)

- Segue a ordem:

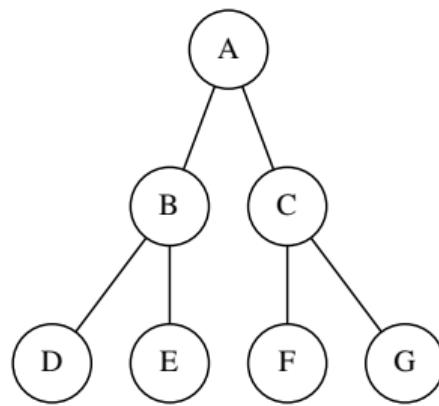
- Percorre subárvore da esquerda
- Visita Raiz
- Percorre subárvore da direita



- 1 D
- 2 B
- 3 E
- 4 A
- 5 C
- 6 F
- 7 G

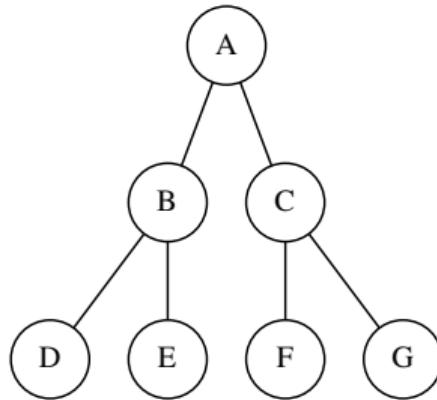
# Caminhamento em Largura

- Visita os nodos na ordem dos níveis da árvore, da esquerda para a direita
  - Visite os nodos de nível 0
  - Visite os nodos de nível 1
  - ...



# Caminhamento em Largura

- Visita os nodos na ordem dos níveis da árvore, da esquerda para a direita
  - Visite os nodos de nível 0
  - Visite os nodos de nível 1
  - ...



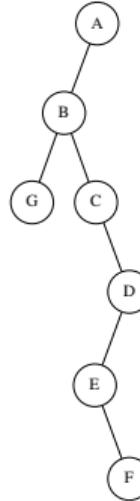
- 1 A
- 2 B
- 3 C
- 4 D
- 5 E
- 6 F
- 7 G

# Exercícios

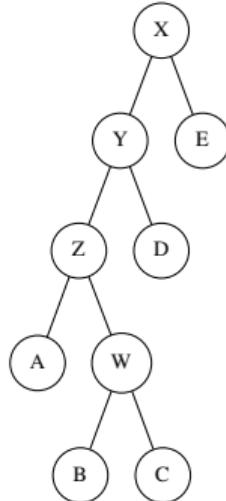
## Exercícios 11 e 12

- 11) Monte a árvore binária para a seguinte seguinte expressão aritmética:  
 $(A+B)-(C*(D-E)+(F/G))$ .
- 12) Apresente os caminhamentos pré-fixado, pós-fixado, central e em largura para as seguintes árvores binárias.

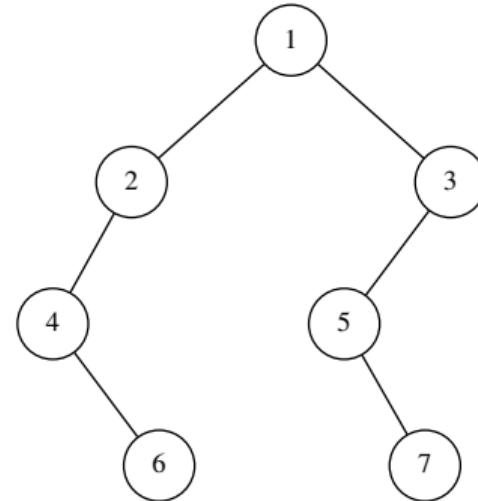
a)



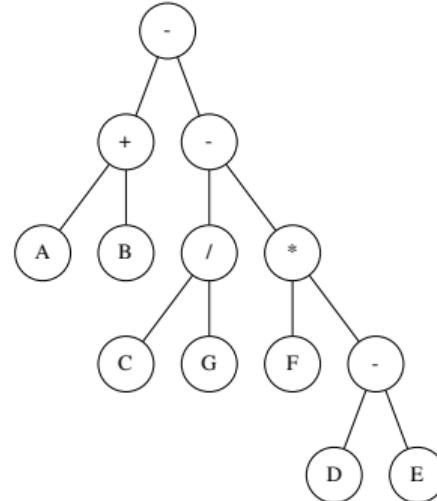
b)



c)

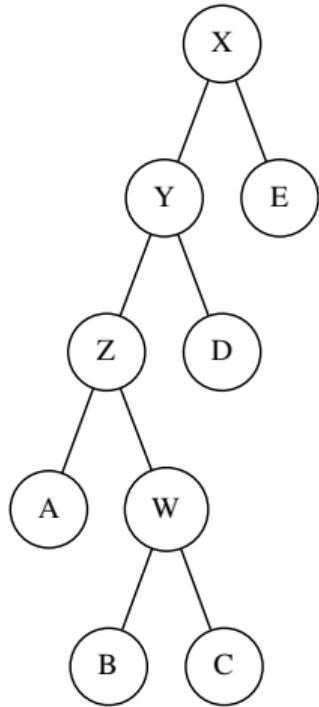


d)



## Exercício 13

- 13 Nas páginas a seguir, há a definição de uma classe genérica para gerenciamento dos nodos de uma árvore binária (`NodeBT`) – cujos métodos devem ser implementados – e uma aplicação para testar a implementação dessa classe (arquivo `exercicio13/app.cpp`). A aplicação cria a árvore ao lado e testa a implementação da classe sobre essa árvore. Você encontra este arquivo, juntamente com o resultado esperado para a sua execução, no arquivo `src.zip` (disponível no Moodle da disciplina). A descrição do que cada método faz encontra-se nos comentários da definição da classe. Por se tratar de um *template*, lembre-se de colocar a definição da classe em um arquivo chamado `NodeBT.hpp` e de **implementar os métodos da classe nesse mesmo arquivo**.



# Trecho do arquivo NodeBT.hpp

```

template <typename T>
class NodeBT {
private:
    T info;      NodeBT *parent, *left, *right;
    string strGraphVizNode(NodeBT const *node) const; // Imprime as conexões de um nodo com seus filhos em GraphViz
public:
    NodeBT(T const &i);                                // Cria um nodo com o caractere i, e sem pai
    ~NodeBT();                                         // Destroi o nodo atual e seus descendentes
    T getInfo() const;                                 // Retorna o caractere armazenado no nodo atual
    void setInfo(T &i);                               // Altera o caractere armazenado no nodo atual
    NodeBT *getParent() const;                         // Retorna a referência para o nodo-pai do nodo atual
    NodeBT *getLeft() const;                           // Retorna a referência para o nodo-filho/subárvore da esquerda
    NodeBT *getRight() const;                          // Retorna a referência para o nodo-filho/subárvore da direita
    void setLeft(NodeBT *subtree);                    // Adiciona uma subárvore como subárvore da esquerda do nodo atual
    void setRight(NodeBT *subtree);                   // Adiciona uma subárvore como subárvore da direita do nodo atual
    void removeLeft();                                // Remove a subárvore da esquerda (desalocando-a, se necessário)
    void removeRight();                               // Remove a subárvore da direita (desalocando-a, se necessário)
    bool isRoot() const;                             // Retorna true se o nodo for raiz
    bool isInternal() const;                         // Retorna true se o nodo for interno
    bool isExternal() const;                          // Retorna true se o nodo for externo
    int degree() const;                            // Retorna o grau (número de filhos) do nodo atual
    int depth() const;                            // Retorna o nível do nodo atual
    int height() const;                           // Retorna a altura da subárvore/árvore a partir do nodo atual
    int size() const;                            // Retorna o número de nodos a partir do nodo atual
    bool contains(T &i) const;                     // Retorna true se o nodo atual ou algum de seus descendentes contiverem i
    NodeBT const *find(T const &i) const;           // Retorna a referência para o nodo, a partir do atual, que contém i ou nullptr
    string strGraphViz() const;                     // Gera uma representação da árvore a partir do nodo atual no formato GraphViz
    string preorder() const;                        // Gera uma cadeia de caracteres (um nodo por linha) em pré-ordem
    string postorder() const;                       // Gera uma cadeia de caracteres (um nodo por linha) em pós-ordem
    string inorder() const;                         // Gera uma cadeia de caracteres (um nodo por linha) em in-ordem
    string levelorder() const;                      // Gera uma cadeia de caracteres (um nodo por linha) em largura
};

```

# Exercício 13: exercício13/app.cpp

```
#include <iostream>
#include <iomanip>
#include "NodeBT.hpp"

void showTree(NodeBT<char> *root) {
    string s = "XYZDAWBC*"; char c = 'C', l = 'L';
    cout << endl << root->strGraphViz() << endl;
    cout << "NPaRiIuEduhguszufCufL" << endl;
    for (int i=0; i<s.length(); ++i) {
        char letter = s[i];
        NodeBT<char> const *node = root->find(letter);
        if (node == nullptr) { cout << letter << "uu*uu*uu*uu*uu*uu*uu*uu*uu*uu*" << endl; continue; }
        NodeBT<char> *parent = node->getParent();
        char parentInfo = (node->getParent() == nullptr)?'*':parent-> getInfo();
        cout << letter << "u" << setw(2) << parentInfo << "u";
        cout << setw(2) << node->isRoot() << "u" << setw(2) << node->isInternal() << "u" << setw(2) << node->isExternal() << "u";
        cout << setw(2) << node->degree() << "u" << setw(2) << node->depth() << "u" << setw(2) << node->height() << "u";
        cout << setw(2) << node->size() << "u" << setw(2) << node->contains(c) << "u" << setw(2) << node->contains(l) << endl;
    }
}

int main() {
    NodeBT<char> *w = new NodeBT<char>('W'); w->setLeft( new NodeBT<char>('B') ); w->setRight( new NodeBT<char>('C') );
    NodeBT<char> *z = new NodeBT<char>('Z'); z->setLeft( new NodeBT<char>('A') ); z->setRight( w );
    NodeBT<char> *y = new NodeBT<char>('Y'); y->setLeft( z );
    NodeBT<char> *root = new NodeBT<char>('X'); root->setLeft( y );
                                            root->setRight( new NodeBT<char>('D') );
    cout << "pre-order:uu" << (root->preorder() == "X\nY\nZ\nA\nW\nB\nC\nD\nE\n" ? "OK" : "ERROR") << endl;
    cout << "post-order:uu" << (root->postorder() == "A\nB\nC\nW\nZ\nD\nY\nE\nX\n" ? "OK" : "ERROR") << endl;
    cout << "in-order:uuu" << (root->inorder() == "A\nZ\nB\nW\nC\nY\nD\nX\nE\n" ? "OK" : "ERROR") << endl;
    cout << "level-order:u" << (root->levelorder() == "X\nY\nE\nZ\nD\nA\nW\nB\nC\n" ? "OK" : "ERROR") << endl;
    showTree( root );
    z->removeRight(); showTree( root );
    root->removeLeft(); showTree( root );
    delete root;
    return 0;
}
```

# Árvores Binárias de Pesquisa (ABP)

# Introdução

- Árvores “comuns” não possuem restrição com relação às posições relativas de seus nodos
- Um item pode aparecer em qualquer posição da árvore
- Algoritmos de busca são ineficientes
  - Pode ser necessário percorrer todos os nodos da árvore.
  - $O(n)$

# Árvores Binárias de Pesquisa (ABP)

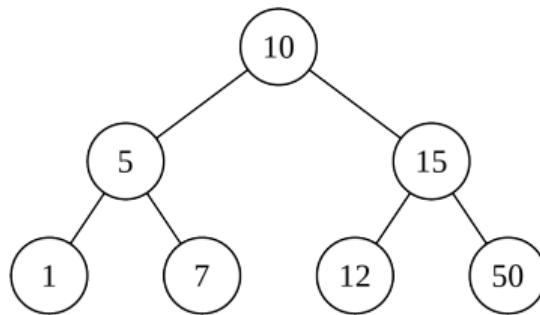
- Classe especial de árvores de busca
- Suportam operações eficientes de busca, inserção e remoção
  - $O(\log n)$ , se balanceada
- Apresentam restrições quanto à posição relativa dos nodos
  - Possuem uma relação de ordem total entre os elementos armazenados (os elementos são usualmente chamados de “chaves”)

# Árvores Binárias de Pesquisa (ABP)

- Seja  $S$  um conjunto cujos elementos possuem uma relação de ordem (exemplo: conjunto de inteiros, lista de nomes, etc.)
- Uma árvore binária de pesquisa para  $S$  é uma árvore binária  $T$  tal que
  - Cada nodo interno  $v$  de  $T$  armazena um elemento de  $S$  denotado  $x(v)$
  - Para cada nodo interno  $v$  de  $T$ , os elementos armazenados na subárvore esquerda de  $v$  são menores ou iguais a  $x(v)$ , e os elementos armazenados na subárvore direita de  $v$  são maiores ou iguais a  $x(v)$

# Árvores Binárias de Pesquisa (ABP)

- Quando a árvore é montada
  - Todos os elementos da subárvore esquerda são menores do que o elemento da raiz
  - Todos os elementos da subárvore direita são maiores do que o elemento da raiz
- Exemplo:



- Quando implementada, todos os métodos respeitam e fazem uso dessa ordenação

# TAD para Árvores Binárias de Pesquisa (ABP)

- A ABP é implementada usando o nodo para árvore binária (NodeBT)
  - A forma mais usual de implementação é através de estruturas encadeadas
  - Cada nodo contém a informação e referências para nodo-pai, subárvore da esquerda e subárvore da direita
- Deve ser possível comparar os elementos de uma árvore
  - Em C++, classes devem ter os operadores relacionais corretamente sobre carregados
  - Em Java, objetos devem implementar a interface Comparable
- Vários métodos são necessários para manipular uma árvore binária de pesquisa
  - Inserir um elemento na árvore
  - Localizar um elemento na árvore
  - Remover um elemento da árvore
  - Caminhar em uma árvore, visitando seus nodos
  - ...

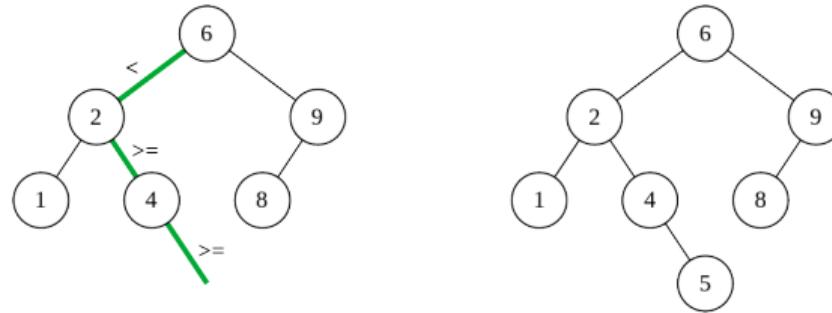
# TAD Genérico para Árvores Binárias de Pesquisa (ABP)

- `bool isEmpty():` retorna true se a árvore está vazia
- `int size():` retorna o número de elementos armazenados na árvore
- `void clear():` remove todos os elementos da árvore
- `T getRoot():` retorna o elemento armazenado na raiz
- `void add(T &e):` insere o elemento e na árvore de forma ordenada
- `bool remove(T &e):` remove o elemento e e reestrutura a árvore se necessário
- `T getLeft(T &e):` retorna o filho à esquerda de e
- `T getRight(T &e):` retorna o filho à direita de e
- `T getParent(T &e):` retorna o pai do elemento e
- `int level(T &e):` retorna o nível do elemento e e - 1 caso o elemento não esteja na árvore
- `int height():` retorna a altura da árvore
- `bool contains(T &e):` retorna true se a árvore contém o elemento e
- `bool isInternal(T &e):` retorna true se o elemento está armazenado em um nodo interno
- `bool isExternal(T &e):` retorna true se o elemento está armazenado em um nodo externo
- `vector<T> preorder():` retorna um vetor com todos os elementos da árvore na ordem pré-fixada
- `vector<T> inorder():` retorna um vetor com todos os elementos da árvore na ordem central
- `vector<T> postorder():` retorna um vetor com todos os elementos da árvore na ordem pos-fixada
- `vector<T> levelorder():` retorna uma lista com todos os elementos da árvore com um caminhamento em largura



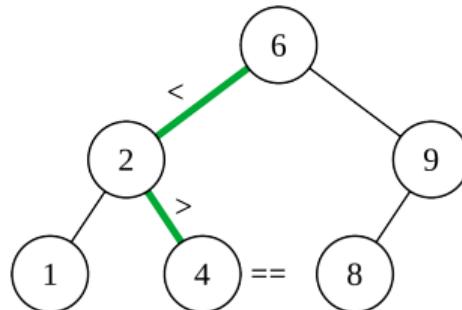
# Inserção em Árvores Binárias de Pesquisa (ABP)

- Se a árvore estiver vazia, então o elemento é a raiz
- Senão
  - Se for menor, inserir na subárvore esquerda
  - Se for maior ou igual, inserir na subárvore direita
- Repetir este processo **recursivamente** até que chegue a uma subárvore vazia
- Neste ponto o novo valor é incluído
- Exemplo: abp.add(5);



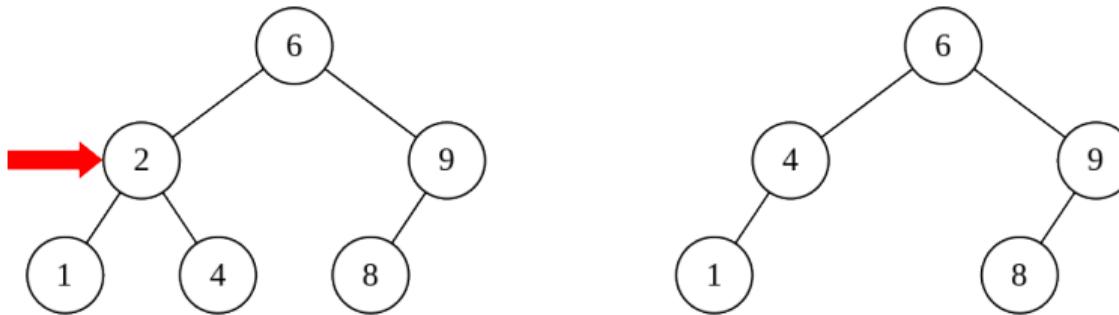
# Busca Recursiva em Árvores Binárias de Pesquisa (ABP)

- Comparar o valor a ser localizado com a raiz
  - Se for igual, então encontrou
  - Se for menor, procura na subárvore esquerda
  - Se for maior, procura na subárvore direita
- Repetir este processo até encontrar ou até chegar a uma subárvore vazia (não encontrou)
- Exemplo: abp.contains(4);



# Remoção em Árvores Binárias de Pesquisa (ABP)

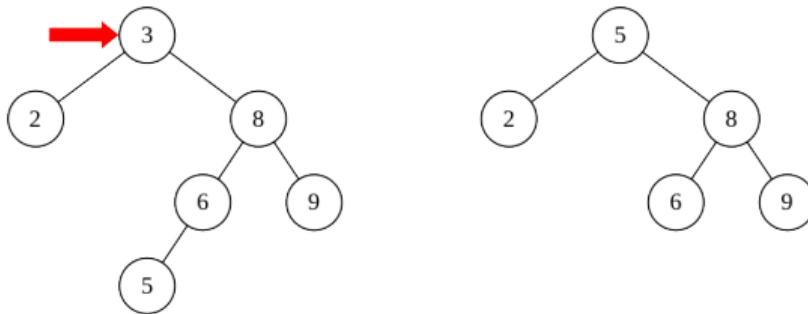
- Trata-se de uma operação mais complexa, pois após a remoção é necessário que a árvore permaneça satisfazendo o critério de ordenação dos dados
- Exemplo: abp.remove(2);



- Algoritmo
  - Localizar o elemento a ser excluído, e, se ele estiver na árvore, guardar quem é o seu pai
  - Verificar se o elemento: é um nodo folha, tem 1 filho ou tem 2 filhos
  - Excluir o nodo e reestruturar a árvore

# Remoção em Árvores Binárias de Pesquisa (ABP)

- Item a ser removido está em uma folha: remover o nodo
- Item a ser removido tem apenas um filho: remover o nodo e o filho ocupa seu lugar
- Item a ser removido tem dois filhos: trocar por um nodo folha e depois remover a folha.  
Há duas possibilidades:
  - Trocar pelo menor item da subárvore direita
  - Trocar pelo maior item da subárvore esquerda
- Exemplo: abp.remove(3);

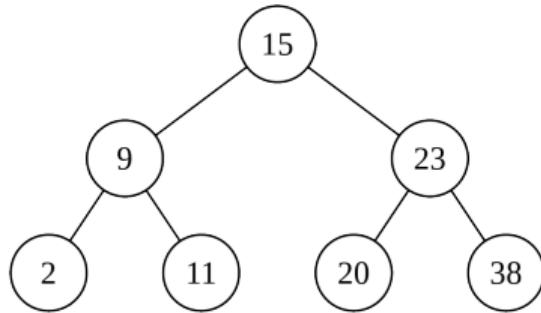


## ABP: Caminhamento

- Os caminhamentos são os mesmos que já foram visto para árvores binárias:
  - Percurso em profundidade
    - Percurso pré-fixado ou em pré-ordem
    - Percurso pós-fixado ou em pós-ordem
    - Percurso central ou em ordem central
  - Percurso em largura

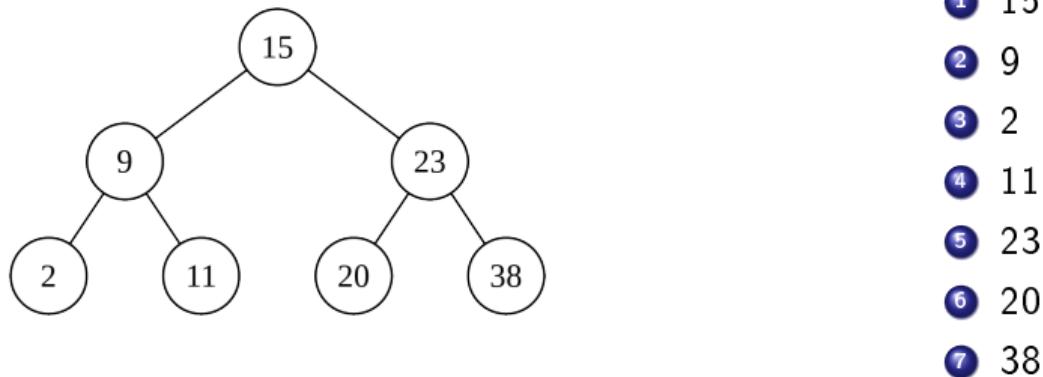
# ABP: Caminhamento Pré-fixado

- O nodo é visitado antes de seus descendentes
- Segue a ordem:
  - Visita Raiz
  - Percorre subárvore da esquerda
  - Percorre subárvore da direita



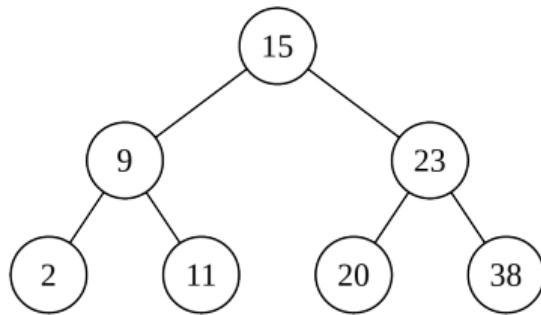
# ABP: Caminhamento Pré-fixado

- O nodo é visitado antes de seus descendentes
- Segue a ordem:
  - Visita Raiz
  - Percorre subárvore da esquerda
  - Percorre subárvore da direita



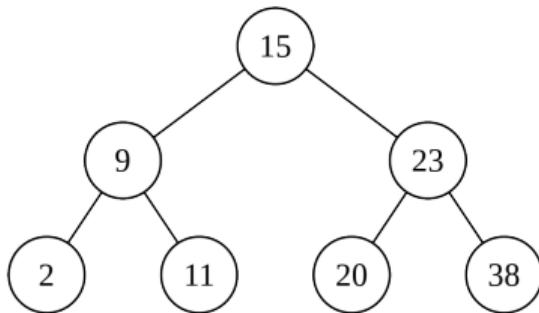
## ABP: Caminhamento Pós-fixado

- O nodo é visitado depois de seus descendentes
- Segue a ordem:
  - Percorre subárvore da esquerda
  - Percorre subárvore da direita
  - Visita Raiz



# ABP: Caminhamento Pós-fixado

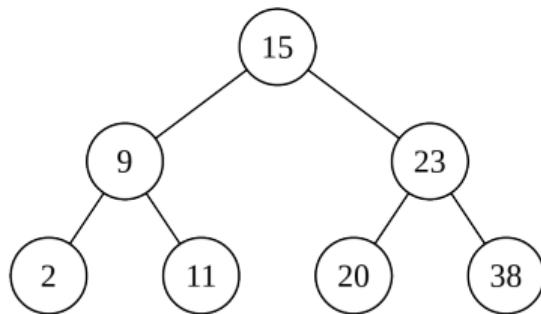
- O nodo é visitado depois de seus descendentes
- Segue a ordem:
  - Percorre subárvore da esquerda
  - Percorre subárvore da direita
  - Visita Raiz



- ① 2
- ② 11
- ③ 9
- ④ 20
- ⑤ 38
- ⑥ 23
- ⑦ 15

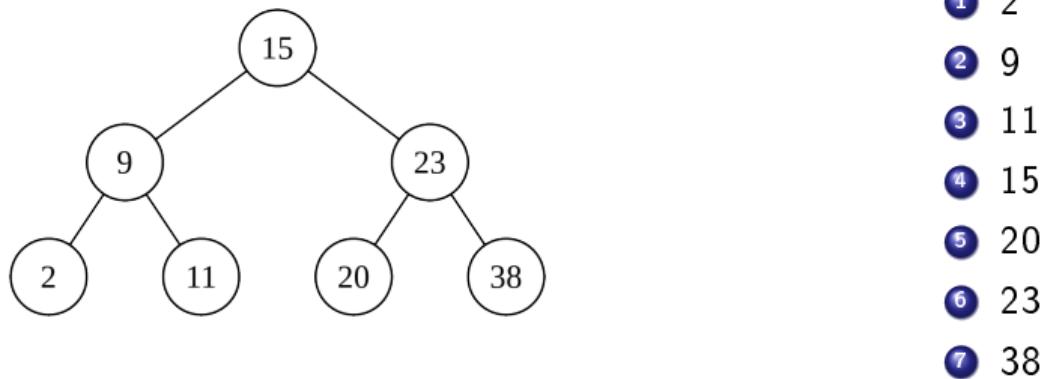
ABP: Caminhamento Central (*In-order*)

- Segue a ordem:
  - Percorre subárvore da esquerda
  - Visita Raiz
  - Percorre subárvore da direita



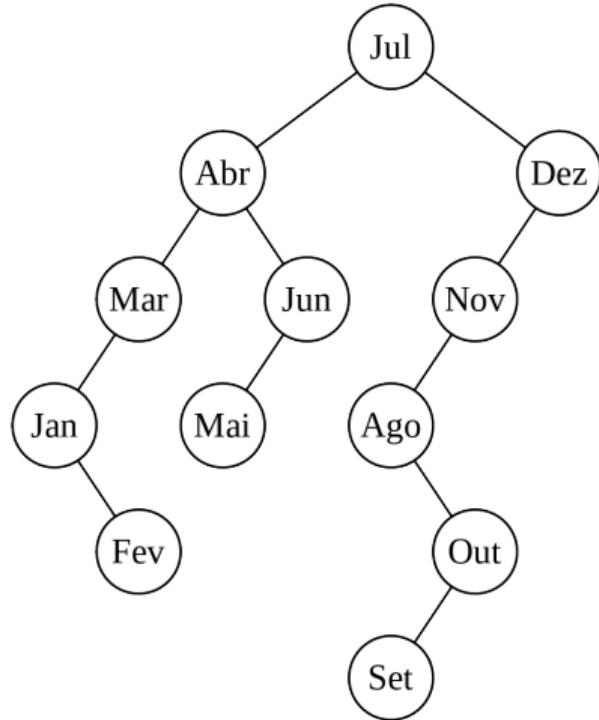
ABP: Caminhamento Central (*In-order*)

- Segue a ordem:
  - Percorre subárvore da esquerda
  - Visita Raiz
  - Percorre subárvore da direita



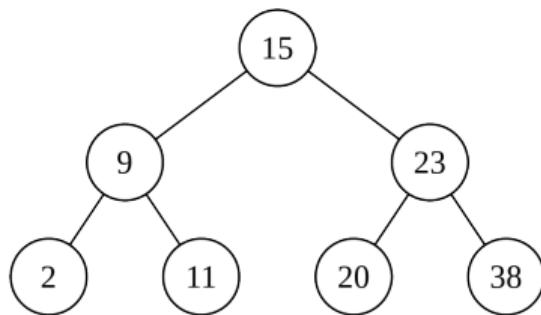
# ABP: Caminhamento Central (*In-order*)

- Em uma ABP, o caminhamento central sempre fornece os dados classificados em ordem crescente
- Exemplo:
  - Considerando: Jan=1, ..., Dez=12
  - **Pré-Fixado:** Jul, Abr, Mar, Jan, Fev, Jun, Mai, Dez, Nov, Ago, Out, Set
  - **Central:** Jan, Fev, Mar, Abr, Mai, Jun, Jul, Ago, Set, Out, Nov, Dez
  - **Pós-Fixado:** Fev, Jan, Mar, Mai, Jun, Abr, Set, Out, Ago, Nov, Dez, Jul



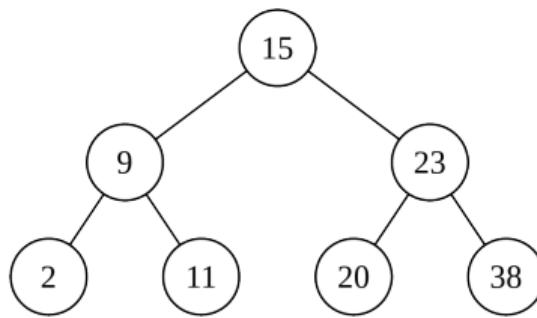
# ABP: Caminhamento em Largura

- Visita os nodos na ordem dos níveis da árvore, da esquerda para a direita
  - Visite os nodos de nível 0
  - Visite os nodos de nível 1
  - ...



# ABP: Caminhamento em Largura

- Visita os nodos na ordem dos níveis da árvore, da esquerda para a direita
  - Visite os nodos de nível 0
  - Visite os nodos de nível 1
  - ...



- 1 15
- 2 9
- 3 23
- 4 2
- 5 11
- 6 20
- 7 38

# Exercícios

## Exercício 14

- 14 Mostre como ficaria uma ABP de inteiros, se fossem incluídos os seguintes elementos (nesta ordem):

8, 2, 10, 6, 5, 15, 7, 3, 20, 11.

Verifique a sua solução usando a seguinte página:

<https://www.cs.usfca.edu/~galles/visualization/BST.html>

## Exercício 15

- 15 Usando a classe genérica para nodo de uma árvore binária, classe NodeBT (desenvolvida no Exercício 13), implemente uma classe, também genérica, chamada BST (de *Binary Search Tree*) de forma que esta classe tenha como variável de instância a referência para o nodo raiz:

```
NodeBT<T> *root;
```

Na interface pública desta classe implemente os seguintes métodos:

- BST(): construtor que simplesmente inicializa root com nullptr
- ~BST(): destrutor que remove todos os nodos da árvore
- string strGraphViz() const: mostra a árvore no formato GraphViz
- void add(const T &e): adiciona o elemento e na posição correta da árvore
- bool contains(const T &e): verifica se a árvore contém o elemento e

O programa da próxima página usa a classe genérica ABP para criar a mesma árvore do Exercício 14.

## Exercício 15: exercício15/app.cpp

```
#include <iostream>
#include "BST.hpp"

int main() {
    int vAdd[] = { 8, 2, 10, 6, 5, 15, 7, 3, 20, 11 };
    int tAdd = sizeof(vAdd) / sizeof(int);
    int vCnt[] = { 1, 2, 0, 3, 4, 5, 21, 6, 22, 7, 23,
                   8, 9, 10, 24, 11, 12, 15, 16, 20, 25 };
    int tCnt = sizeof(vCnt) / sizeof(int);
    BST<int> *bst = new BST<int>();
    for (int i=0; i<tAdd; ++i)
        bst->add( vAdd[i] );
    cout << bst->strGraphViz();
    for (int i=0; i<tCnt; ++i)
        if ( (i%2==0) == bst->contains( vCnt[i] ) ) cout << "ERROR" << endl;
    delete bst;
    return 0;
}
```

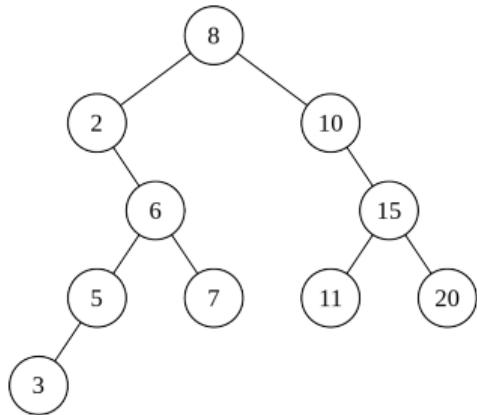
# ABPs Balanceadas

# Árvores Binárias de Pesquisa (ABP)

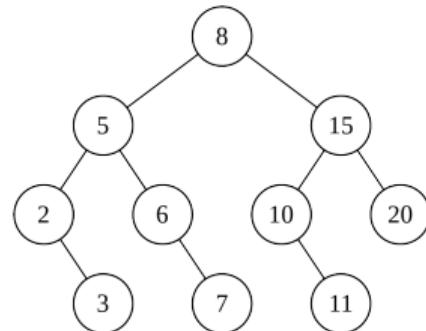
- ABPs apresentam restrições quanto à posição relativa dos nodos
- Suportam operações eficientes de busca e inserção **se estiverem balanceadas**
  - $O(n)$ , se não estiver balanceada
  - $O(\log n)$ , se estiver balanceada

Árvore não balanceada

X



Árvore balanceada



# ABP Balanceada – Introdução

- Usar estruturas de dados e algoritmos **eficientes e simples de implementar** é importante
- Uma ABP balanceada apresenta melhor (desempenho) que uma ABP
  - Muitas vezes, árvores balanceadas podem ter um desempenho muito superior do que listas encadeadas e matrizes
- Critério para determinar se uma árvore não vazia é balanceada:
  - Depende do tipo de árvore
  - Exemplos de ABPs平衡adas:
    - Árvores AVL  
<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>
    - Árvores rubro-negras  
<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>
    - Árvores de Andersson  
[https://en.wikipedia.org/wiki/AA\\_tree](https://en.wikipedia.org/wiki/AA_tree)
    - Splay Tree (nódos são reposicionados de acordo com a frequência de acesso)  
<https://www.cs.usfca.edu/~galles/visualization/SplayTree.html>

# Árvores AVL

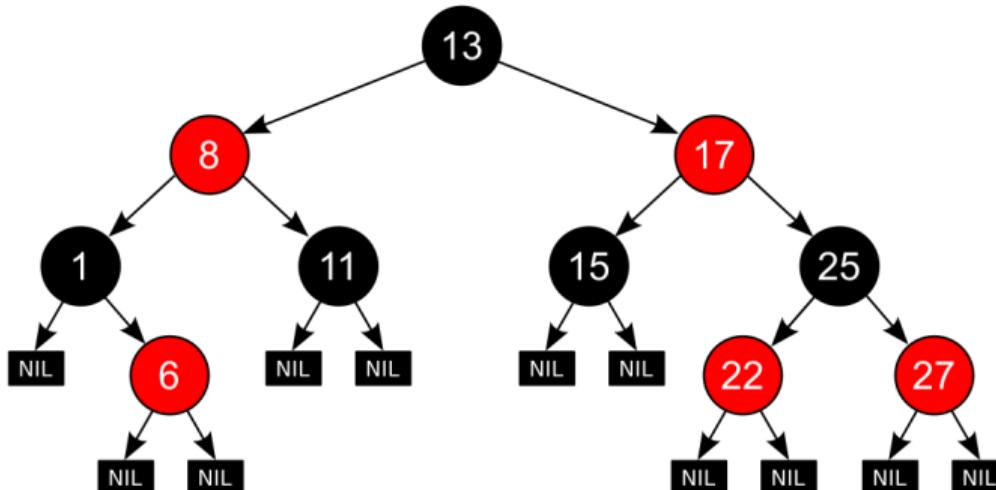
- Nome referente aos autores: G. M. Adelson-Velskii e E. M. Landis
- Operações de busca, inserção e remoção:  $O(\log n)$
- Regra para árvore estar balanceada:
  - Subárvore da esquerda e da direita平衡adas
  - Altura das subárvore devem ter no máximo a diferença de uma unidade
- A cada inserção ou exclusão, é necessário:
  - Atualizar as alturas dos nodos das subárvore envolvidas na operação
  - Calcular o fator de balanceamento da árvore
  - Verificar se a árvore ainda está平衡ada

# Árvores rubro-negras

- Uma árvore rubro-negra (*red-black tree*) é um tipo especial de árvore binária de pesquisa
- Nodos com dados são chamados de internos (ou *non-NIL*)
- Nodos-folha NÃO contém dados e são identificados como *NIL* (`nullptr`, por exemplo)
- Além dos requisitos de uma ABP, uma árvore rubro-negra deve satisfazer aos seguintes requisitos:
  - ① Todo nodo é ou vermelho ou preto
  - ② Todos os nodos *NIL* são considerados pretos
  - ③ Um nodo vermelho NÃO tem filhos vermelhos
  - ④ Todo caminho de determinado nodo para algum de seus nodos descendentes *NIL* passa pelo mesmo número de nodos pretos
  - ⑤ (Conclusão) Se um nodo *N* tem exatamente um filho, ele deve ser um filho vermelho, pois se ele fosse preto, os seus descendentes *NIL* teriam uma profundidade preta diferente do que o filho *NIL* de *N* (o que violaria o requisito 4)

# Árvores rubro-negras

- Para criar árvores rubro-negras, experimente usar  
<https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>
- Exemplo:



Fonte: [https://en.wikipedia.org/wiki/File:Red-black\\_tree\\_example\\_with\\_NIL.svg](https://en.wikipedia.org/wiki/File:Red-black_tree_example_with_NIL.svg)

# Árvores rubro-negras

- Inserções, remoções e buscas possuem pior tempo de execução equivalente a  $O(\log n)$
- Inserções e remoções podem modificar sua estrutura
  - Rebalanceamento local (apenas a parte “afetada” pela inserção ou remoção)
- Propriedades não podem ser violadas
  - Pode ser necessário mudar a estrutura da árvore e as cores de alguns nodos

# Árvores de Andersson

- São ABPs平衡adas baseadas nas árvores rubro-negras
- Propriedades de desempenho e estrutura semelhantes
- Mais simples e fácil de implementar
  - São uma alternativa simples para as tradicionais ABPs平衡adas
  - Performance próxima à das árvores rubro-negras, com menor esforço de programação
- Foram introduzidas por Arne Andersson
  - ANDERSON, Arne. **Balanced Search Trees Made Simple.**  
*Proceedings of Workshop on Algorithms and Data Structures.*  
Springer, Berlin, Heidelberg, 1993. p. 60-71.  
<http://user.it.uu.se/~arnea/ps/simp.pdf>.
- Posteriormente, foi discutida por Mark Allen Weiss
  - WEISS, M. A. **Data Structures & Algorithm Analysis in C++.**

# Árvores Balanceadas AVL

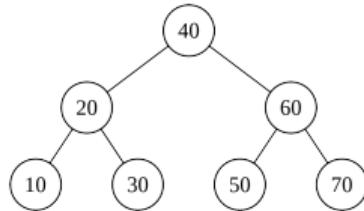
# Árvore Binárias de Pesquisa (ABP)

- Há uma relação de ordem definida pelas chaves (informações armazenadas)
- As operações básicas são: inserção, consulta e remoção
- O balanceamento garante que o tempo de busca seja  $O(\log n)$

# Árvore Binárias de Pesquisa (ABP)

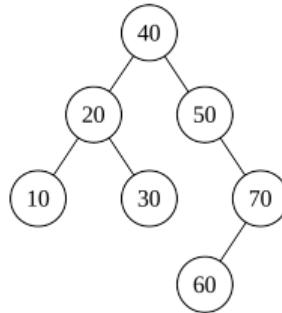
Ordem de inserção:

40, 20, 60, 10, 30, 50, 70



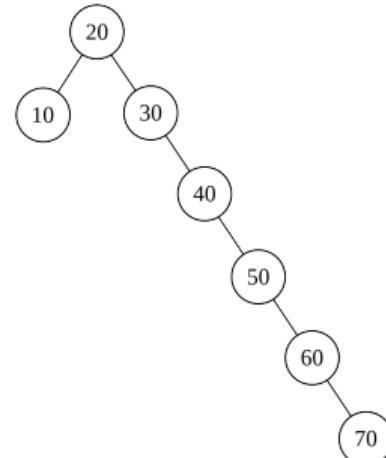
Ordem de inserção:

40, 20, 50, 10, 30, 70, 60



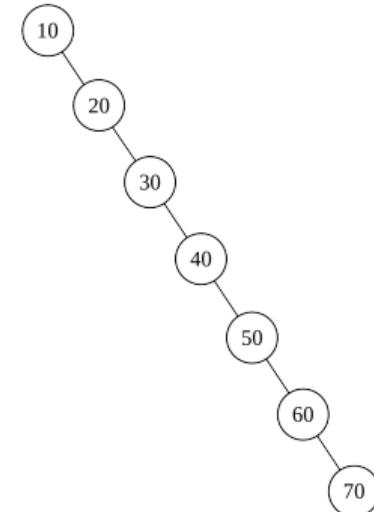
Ordem de inserção:

20, 10, 30, 40, 50, 60, 70



Ordem de inserção:

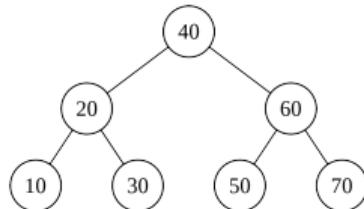
10, 20, 30, 40, 50, 60, 70



# Árvore Binária de Pesquisa (ABP)

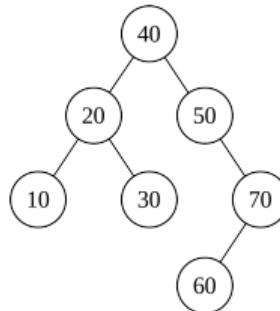
Ordem de inserção:

40, 20, 60, 10, 30, 50, 70



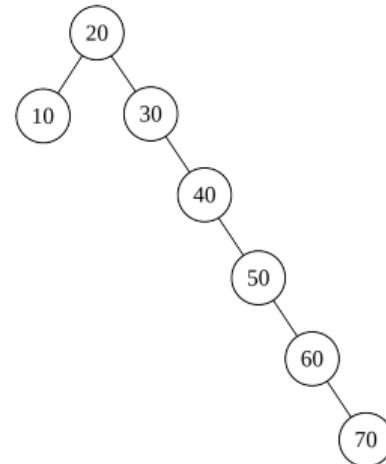
Ordem de inserção:

40, 20, 50, 10, 30, 70, 60



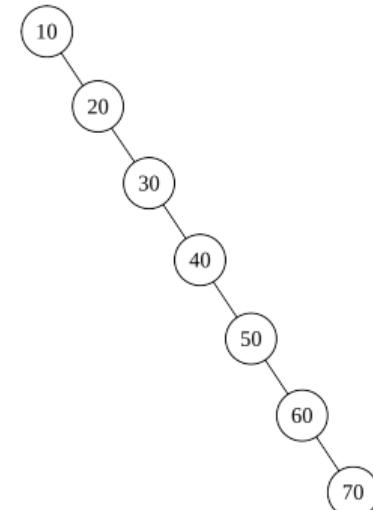
Ordem de inserção:

20, 10, 30, 40, 50, 60, 70



Ordem de inserção:

10, 20, 30, 40, 50, 60, 70



- A ordem de inserção pode causar desbalanceamento, comprometendo o desempenho

# Balanceamento de Árvores

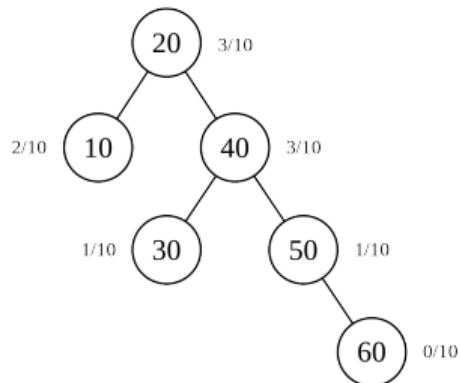
- O balanceamento busca uma distribuição equilibrada dos nodos, tendo como objetivos:
  - Otimizar as operações de consulta
  - Diminuir o número médio de comparações
- Tipos de distribuição
  - Uniforme
    - A árvore é balanceada pela altura (distância entre as alturas dos nodos não deve exceder determinado valor)
    - Exemplo: árvores AVL e rubro-negras
  - Não uniforme
    - Chaves mais solicitadas ficam mais perto da raiz
    - Exemplo: *splay trees*

# Tipos de Distribuição

## Splay Tree

Nodos mais acessados ficam perto da raiz

Ordem de inserção: 20, 50, 10, 60, 40, 30  
 Acessos: 30, 50, 40, 10, 20, 10, 40, 20, 40, 20



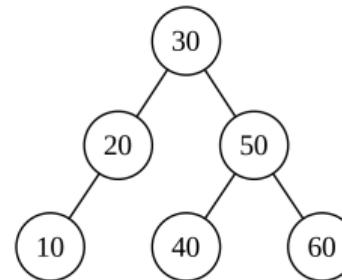
Testar em:

<https://www.cs.usfca.edu/~galles/visualization/SplayTree.html>

## Árvore AVL

Diferenças das alturas das subárvore não excedem um determinado valor

Ordem de inserção: 30, 20, 50, 10, 40, 60



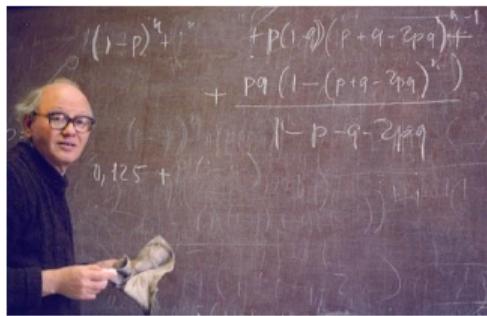
Testar em:

<https://www.cs.usfca.edu/~galles/visualization/AVLtree.html>

# Árvores AVL (ADELSON-VELSKY & LANDIS)

- Foram propostas por Adelson-Velsky e Landis em 1962

ADELSON-VELSKY, G. M.; LANDIS, E. M. (1962). "An algorithm for the organization of information". *Proceedings of the USSR Academy of Sciences*, 146(2): p. 263-266.



Georgy Maximovich Adelson-Velsky (1922-2014)

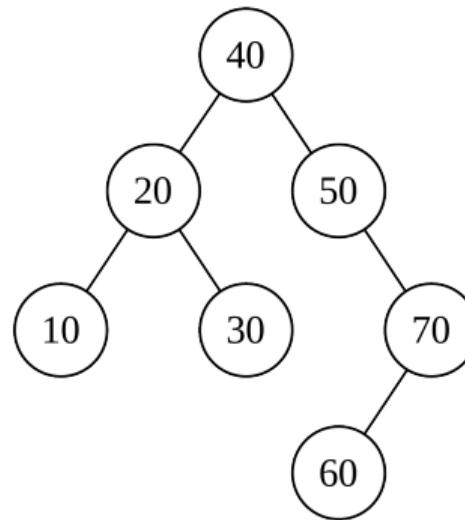
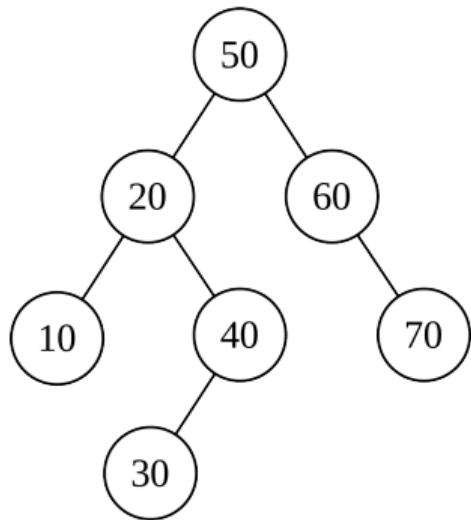


Evgenii Mikhailovich Landis (1921-1997)

- Uma ABP é uma AVL quando, para qualquer um de seus nodos, a diferença entre as alturas de suas subárvore direita e esquerda é no máximo 1

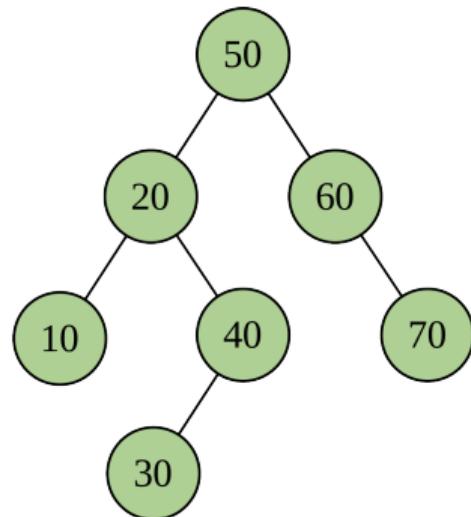
## Árvores AVL

- As seguintes ABP são árvores AVL?

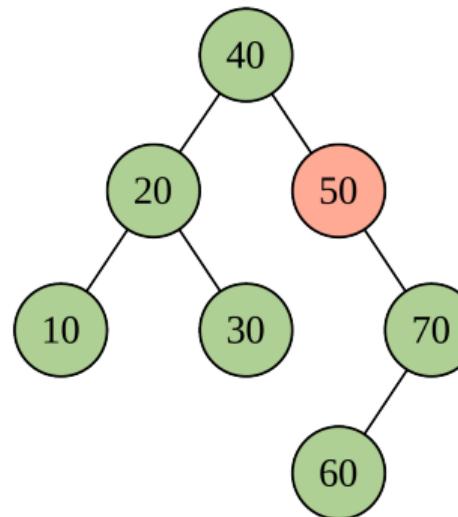


# Árvores AVL

- As seguintes ABP são árvores AVL?



SIM



NÃO

# Fator de Balanceamento

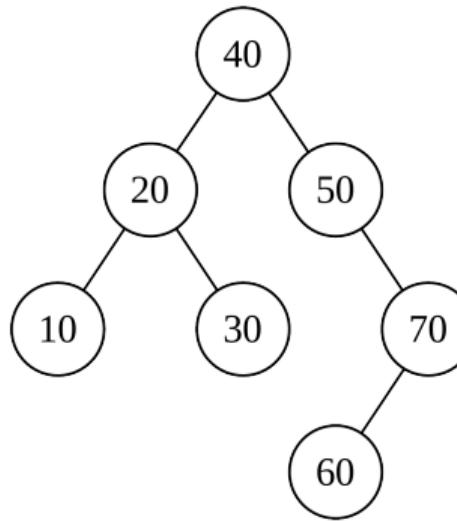
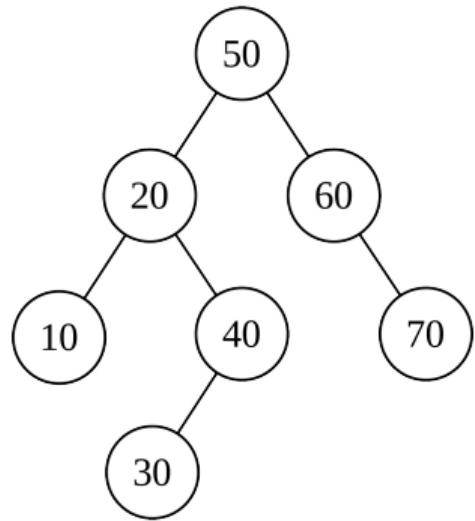
- Fator de Balanceamento (FB): é a diferença entre as alturas das subárvore direita e esquerda de um nodo

$$\text{FB}(\text{nodo}) = \text{altura}(\text{nodo} \rightarrow \text{dir}) - \text{altura}(\text{nodo} \rightarrow \text{esq})$$

- FB precisa ser -1, 0 ou +1 em todos os nodos da árvore para que árvore seja AVL

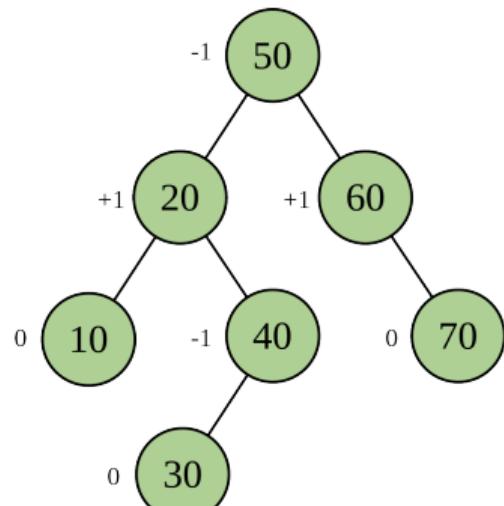
## Fator de Balanceamento: Exemplo

- Determine o fator de balanceamento dos nodos das seguintes árvores e verifique se elas são árvores AVL.

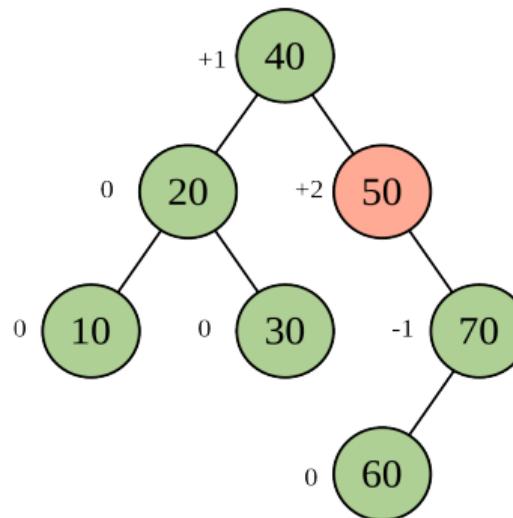


# Fator de Balanceamento: Exemplo

- Determine o fator de balanceamento dos nodos das seguintes árvores e verifique se elas são árvores AVL.



**SIM**



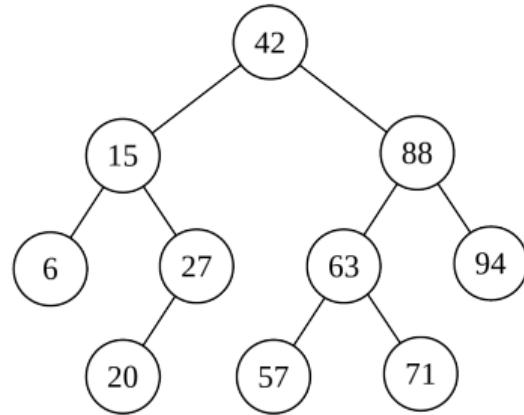
**NÃO**

# Exercício

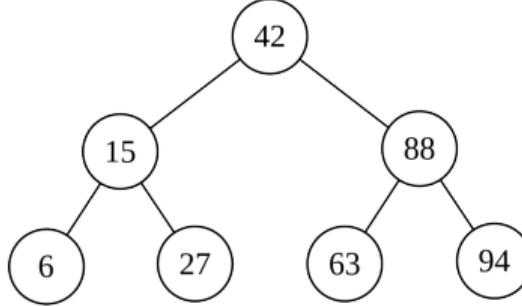
## Exercício 16

- 16) Determine o fator de balanceamento dos nodos das seguintes árvores e verifique se elas são árvores AVL.

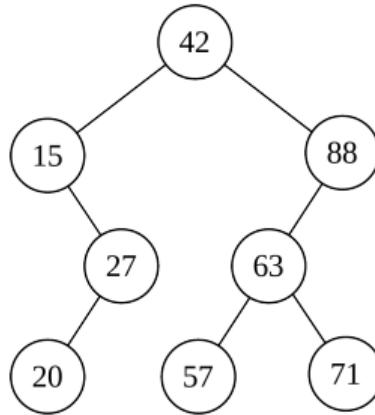
a)



b)



c)



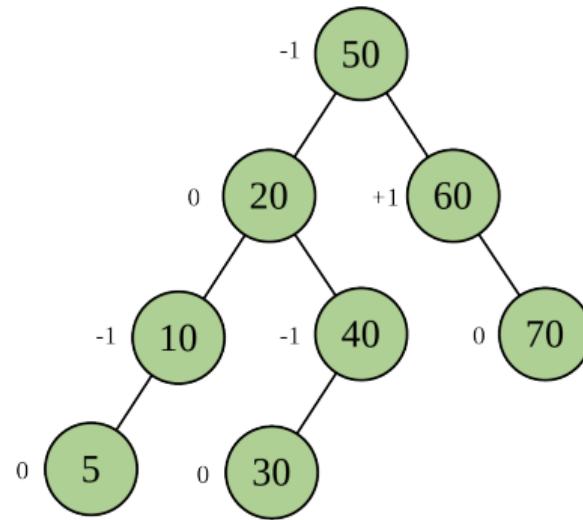
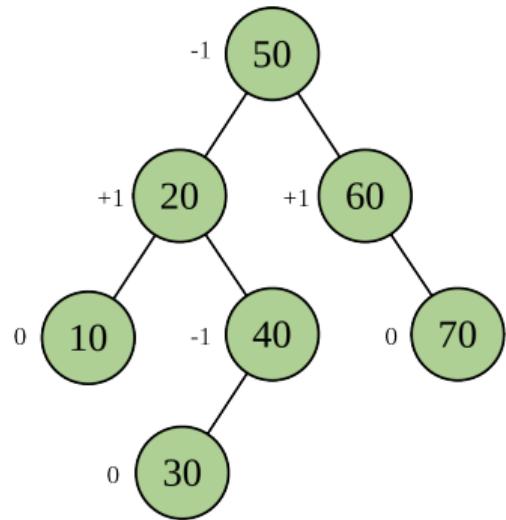
# Operações sobre Árvores Balanceadas AVL

# Operações sobre Árvores Balanceadas AVL

- A pesquisa em árvores AVL é igual à pesquisa em ABPs
- **Inserção e exclusão** devem preservar as propriedades da árvore AVL

## Inserção em Árvores AVL

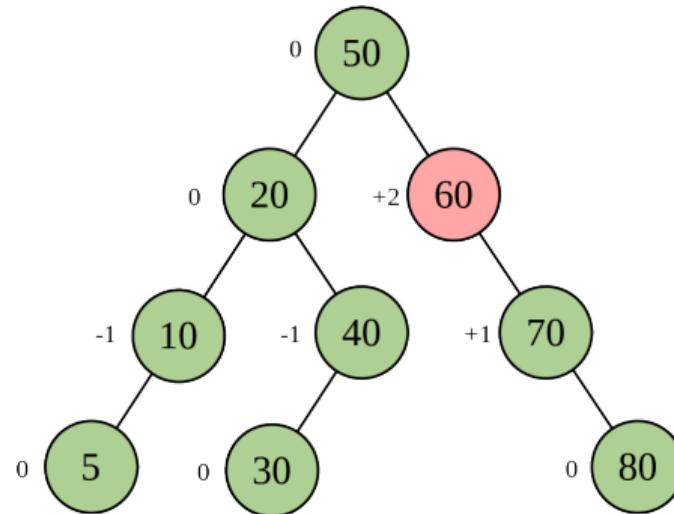
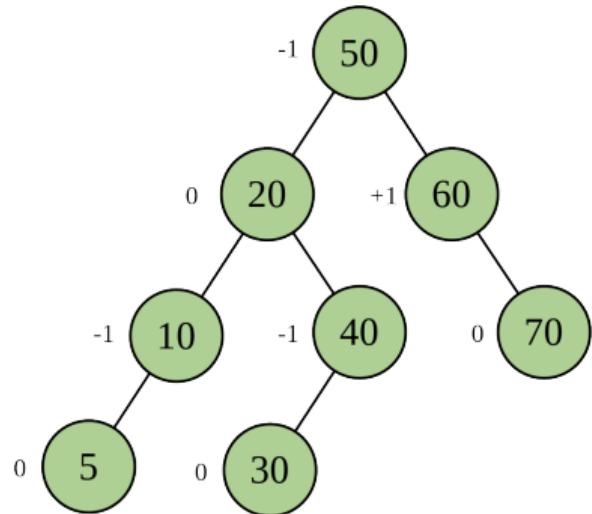
- `avl->add(5);`



- A árvore continua balanceada!

# Inserção em Árvores AVL

- `avl->add(80);`



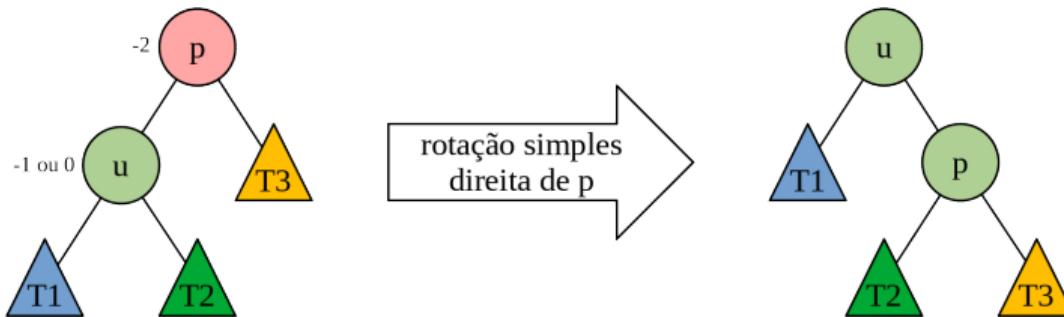
- Agora será necessário REESTRUTURAR A ÁRVORE!

# Balanceamento de Árvores AVL por Rotação

- Quando uma inserção ou exclusão faz com que a árvore perca as propriedades de árvore AVL, deve-se realizar uma operação de reestruturação chamada **rotação**
- A rotação preserva a ordem das chaves, de modo que a árvore resultante é uma ABP válida e é uma árvore AVL válida
- Tipos de rotação
  - Simples: direita ou esquerda
  - Dupla: direita-esquerda ou esquerda-direita
- Nas operações ilustradas a seguir:
  - T1, T2, T3 e T4 são subárvores (vazias ou não)
  - O nó p corresponde ao nó que ficou desbalanceado, ou seja, será a raiz da transformação

# Rotação Simples Direita

- Aplicado em nodos com  $FB = -2$  e filho com  $FB = -1$  ou  $FB = 0$  (sinal igual e negativo)

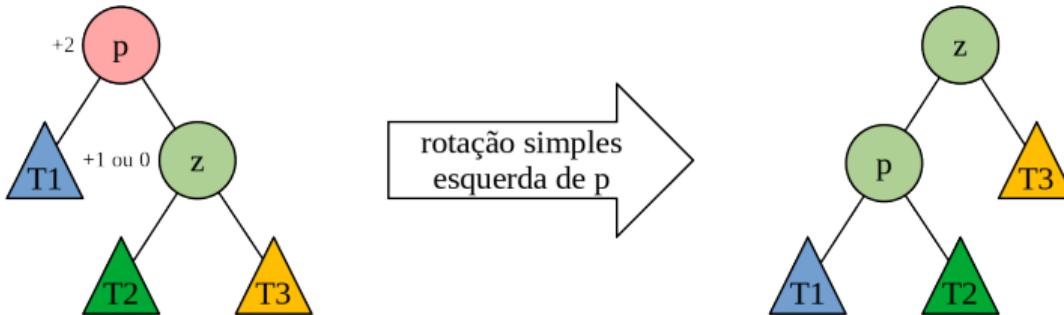


- Implementação (esboço):

```
NodeBT<T> *rotateRight(NodeBT<T> *p) {
    NodeBT<T> u = p->getLeft();
    p->setLeft( u->getRight() );
    u->setRight( p );
    return u; // Ficará no lugar de p
}
```

# Rotação Simples Esquerda

- Aplicado em nodos com  $FB = +2$  e filho com  $FB = +1$  ou  $FB = 0$  (sinal igual e positivo)



- Implementação (esboço):

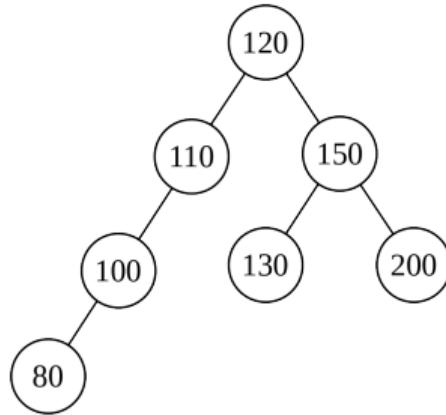
```
NodeBT<T> *rotateLeft(NodeBT<T> *p) {
    NodeBT<T> z = p->getRight();
    p->setRight( z->getLeft() );
    z->setLeft( p );
    return z; // Ficará no lugar de p
}
```

## Exercícios

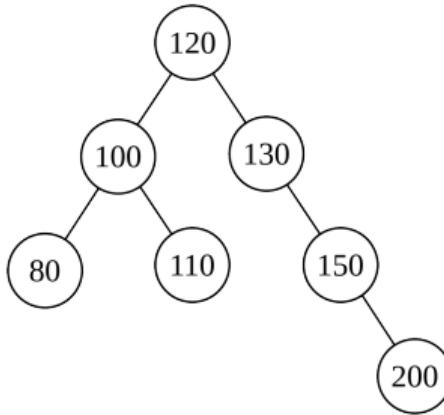
## Exercício 17

- 17) Determine o fator de balanceamento dos nodos das seguintes árvores e aplique a rotação apropriada para transformá-las em árvores AVL.

a)



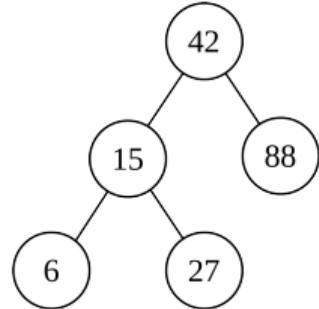
b)



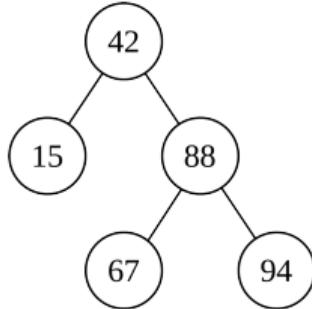
## Exercício 18

- 18 Considere as árvores AVL a seguir, insira os nodos indicados nas respectivas árvores, determine o fator de balanceamento de seus nodos e aplique, se necessário, a rotação apropriada para que elas continuem sendo árvores AVL.

a) `avl18a->add(4);`



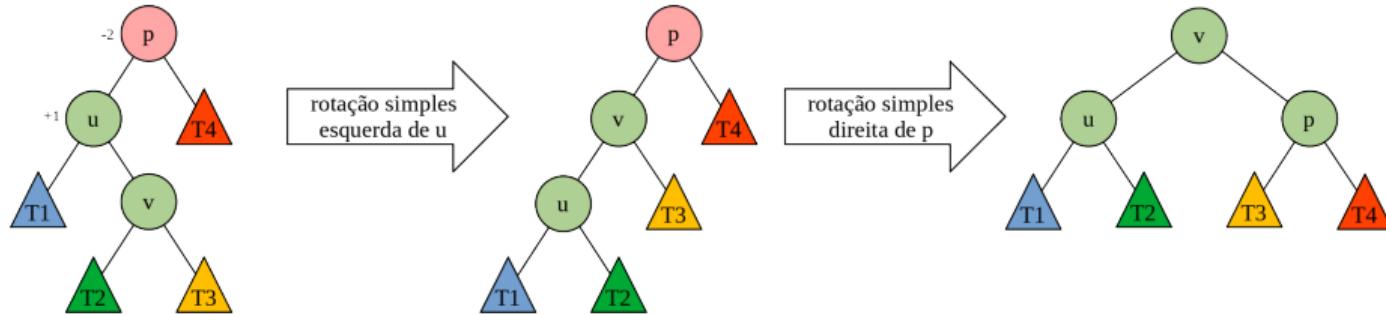
b) `avl18b->add(90);`



## Operações sobre Árvores Balanceadas AVL (cont.)

## Rotação Dupla Esquerda-Direita

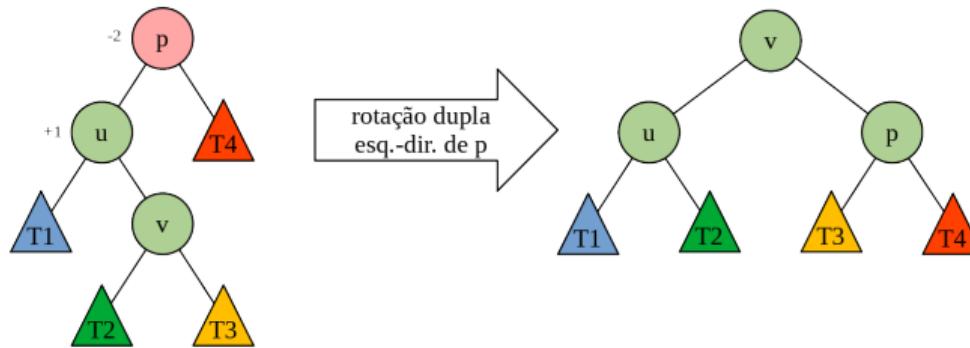
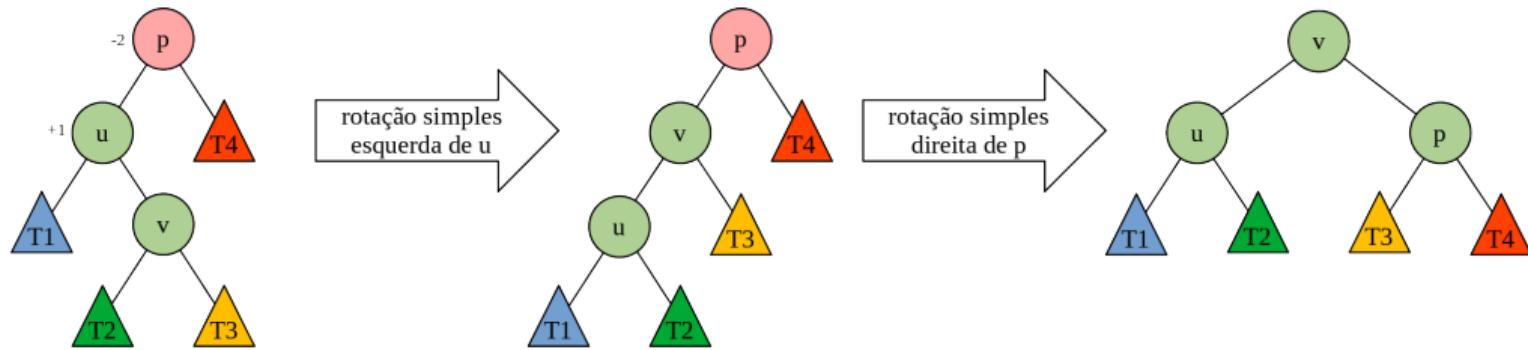
- Aplicado em nodos com  $FB = -2$  e filho com  $FB = +1$  (sinal contrário com pai negativo e filho positivo)
- Aplica-se a rotação simples para a esquerda do filho com  $FB = +1$ , seguida da rotação simples para a direita do pai com  $FB = -2$



- Implementação (esboço):

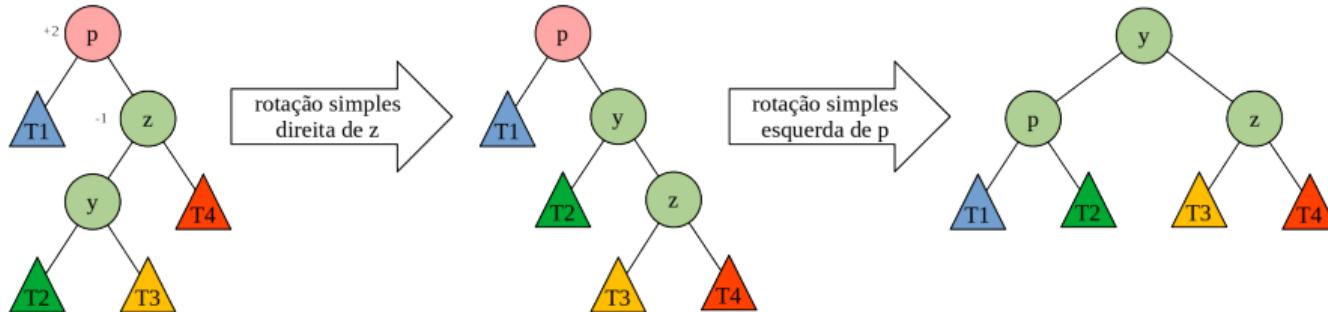
```
NodeBT<T> *rotateLeftRight(NodeBT<T> *p) {
    p->setLeft( rotateLeft( p->getLeft() ) );
    return rotateRight( p ); // Ficará no lugar de p
}
```

# Rotação Dupla Esquerda-Direita



## Rotação Dupla Direita-Esquerda

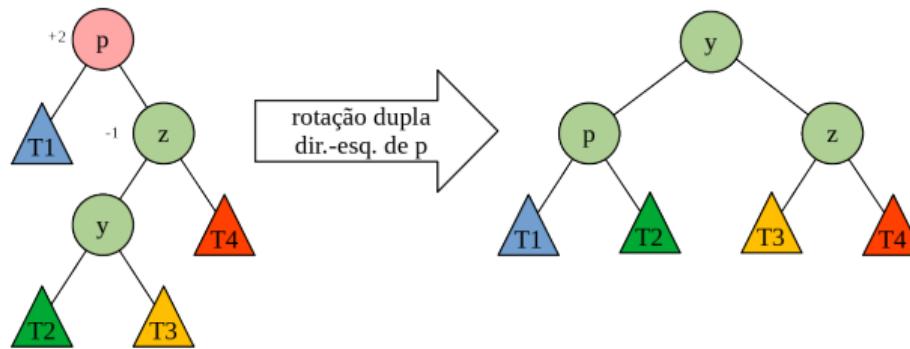
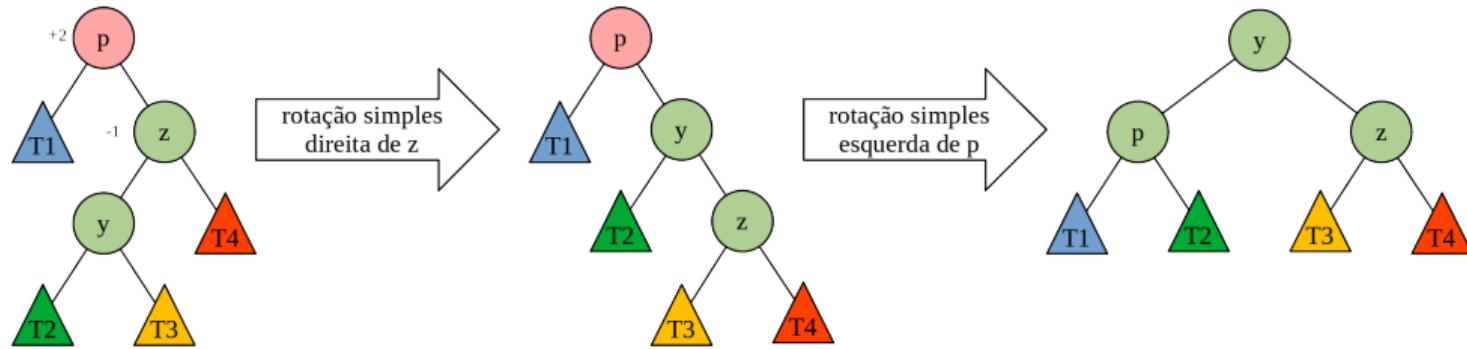
- Aplicado em nodos com  $FB = +2$  e filho com  $FB = -1$  (sinal contrário com pai positivo e filho negativo)
- Aplica-se a rotação simples para a direita do filho com  $FB = -1$ , seguida da rotação simples para a esquerda do pai com  $FB = +2$



- Implementação (esboço):

```
NodeBT<T> *rotateRightLeft(NodeBT<T> *p) {
    p->setRight( rotateRight( p->getRight() ) );
    return rotateLeft( p ); // Ficará no lugar de p
}
```

# Rotação Dupla Direita-Esquerda

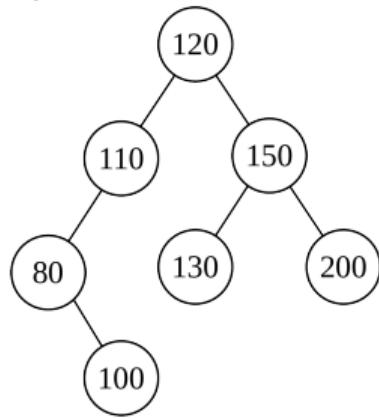


# Exercícios

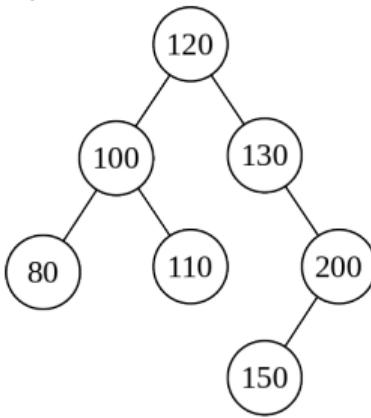
## Exercício 19

- 19) Determine o fator de balanceamento dos nodos das seguintes árvores e aplique a rotação apropriada para transformá-las em árvores AVL.

a)



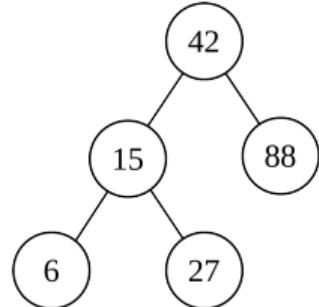
b)



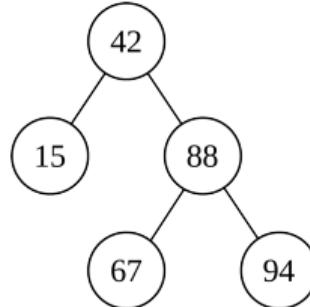
## Exercício 20

- 20 Considere as árvores AVL a seguir, insira os nodos indicados nas respectivas árvores, determine o fator de balanceamento de seus nodos e aplique, se necessário, a rotação apropriada para que elas continuem sendo árvores AVL.

a) `avl20a->add(34);`



b) `avl20b->add(70);`



## Operações sobre Árvores Balanceadas AVL (cont.)

# Inserção de Nodos em Árvores AVL

- Percorrer a árvore verificando se a chave já existe ou não
  - Se a chave existir e a árvore NÃO permitir elementos duplicados, encerrar a tentativa de inserção
  - Caso contrário, localizar a posição correta de inserção do nodo
- Verificar se a inclusão tornará a árvore desbalanceada
  - Em caso negativo, o processo termina
  - Caso contrário, efetuar o balanceamento da árvore
- Descobrir qual a operação de rotação a ser executada
  - Executar a rotação

## Execução das Rotações

- Fator de Balanceamento:  $FB(n) = \text{altura}(n \rightarrow \text{dir}) - \text{altura}(n \rightarrow \text{esq})$
- Se  $|FB| > 1$ , aplicar rotação
  - Se  $FB$  positivo (subárvore da direita é maior): rotações à esquerda
  - Se  $FB$  negativo (subárvore da esquerda é maior): rotações à direita
- Se  $FB$  do nodo e  $FB$  do filho têm o **mesmo sinal**, aplicar **rotação simples**
  - Nodo com  $FB = -2$  e filho com  $FB = -1$  ou  $FB = 0$ :  
rotação do nodo com  $FB = -2$  para direita
  - Nodo com  $FB = +2$  e filho com  $FB = +1$  ou  $FB = 0$ :  
rotação do nodo com  $FB = +2$  para esquerda
- Se  $FB$  do nodo e  $FB$  do filho têm **sinais diferentes**, aplicar **rotação dupla**
  - Nodo com  $FB = -2$  e filho com  $FB = +1$ :  
rotação do nodo com  $FB = +1$  para esquerda, e  
rotação do nodo com  $FB = -2$  para direita
  - Nodo com  $FB = +2$  e filho com  $FB = -1$ :  
rotação do nodo com  $FB = -1$  para direita, e  
rotação do nodo com  $FB = +2$  para esquerda

# Exercício

## Exercício 21

- 21) Inserir nodos com as seguintes chaves em uma árvore AVL, refazendo a árvore quando houver rotação e anotando as rotações realizadas:  
50, 40, 30, 45, 47, 55, 56, 1, 2, 3, 49.

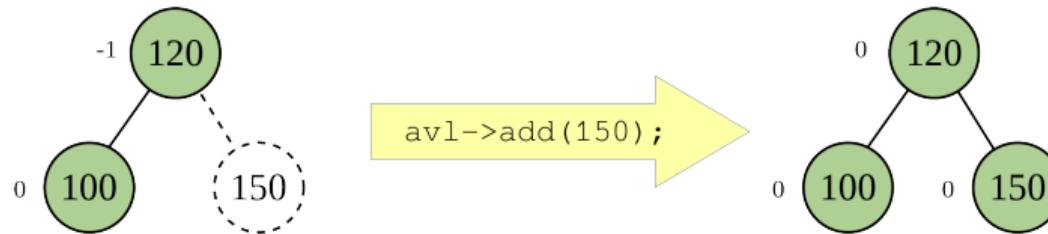
## Operações sobre Árvores Balanceadas AVL (cont.)

# Sobre Inserção em Árvores AVL – PROBLEMAS!

- Como saber se a árvore está balanceada?
  - Verificando se existe um nó “desbalanceado”
- Como saber se um nó está desbalanceado?
  - Determina-se as alturas de suas subárvores e subtrai-se uma da outra
- Procedimento muito lento!
- Como ser mais eficiente?
  - Para cada nó de uma árvore, armazena-se uma variável  $fb$  que registra essa diferença

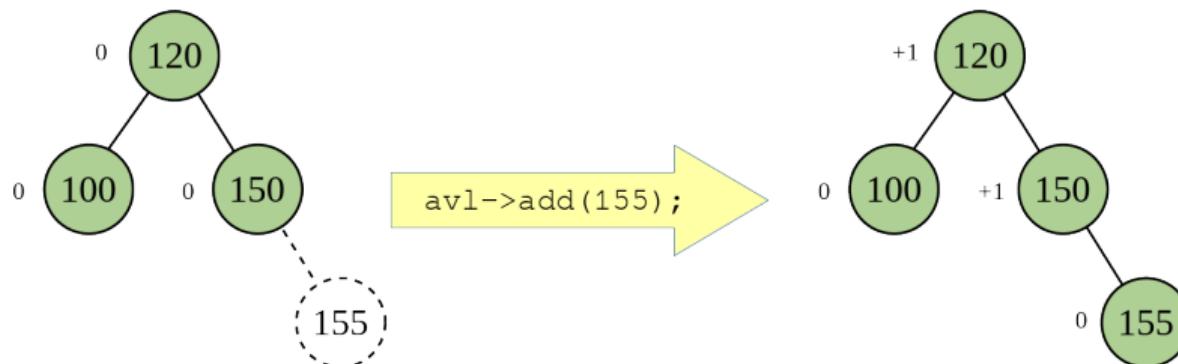
## Manutenção de FB: Inserção à Direita de v

- Se, antes da inclusão,  $FB(v) = -1$ , então  $FB(v)$  se tornará 0
  - Altura da árvore não foi alterada
  - Por consequência, altura dos outros nodos no caminho até a raiz não se altera também



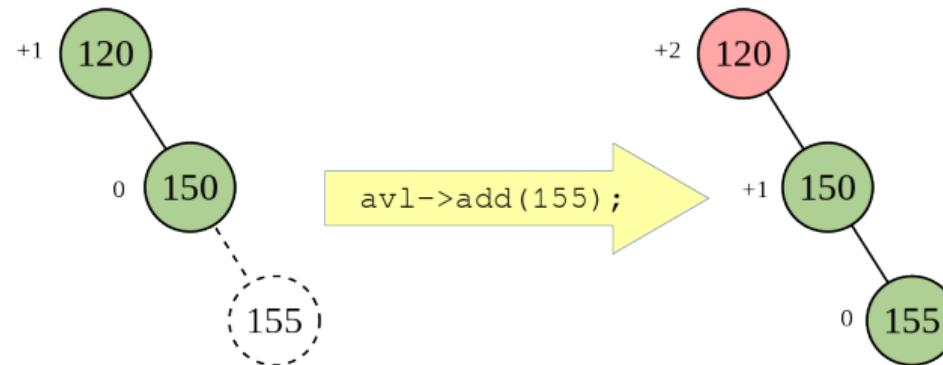
# Manutenção de FB: Inserção à Direita de v

- Se, antes da inclusão,  $FB(v) = 0$ , então  $FB(v)$  se tornará +1
  - Altura da árvore foi modificada
  - Por consequência, altura dos outros nodos no caminho até a raiz pode ter sido alterada também
  - Repetir o processo (recursivamente), com  $v$  substituído por seu pai



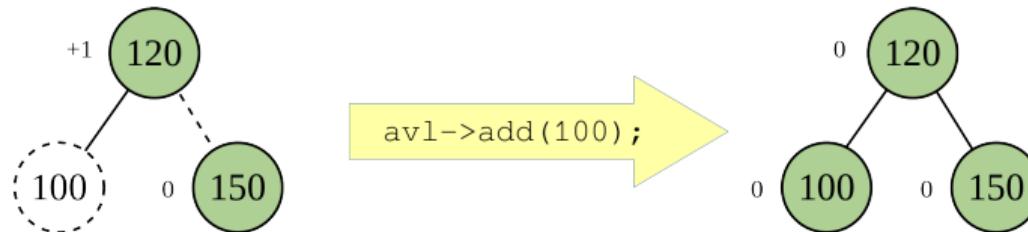
# Manutenção de FB: Inserção à Direita de v

- Se, antes da inclusão,  $FB(v) = +1$ , então  $FB(v)$  se tornará  $+2$ 
  - Esse caso só ocorre por propagação de inserção em nó com  $FB = 0$
  - Altura da árvore foi modificada e o nodo está desbalanceado
  - Rotação correta deve ser empregada
  - Como a árvore será redesenhada, não é necessário verificar os outros nodos



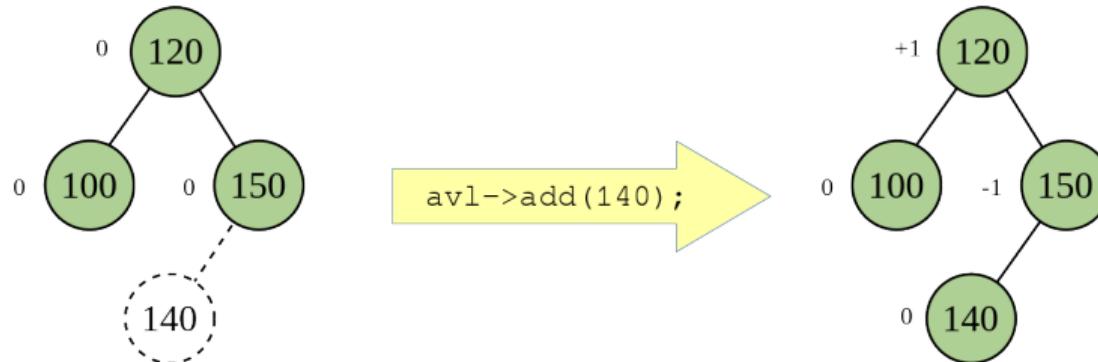
## Manutenção de FB: Inserção à Esquerda de v

- Se, antes da inclusão,  $FB(v) = +1$ , então  $FB(v)$  se tornará 0
  - Altura da árvore não foi alterada
  - Por consequência, altura dos outros nodos no caminho até a raiz não se altera também



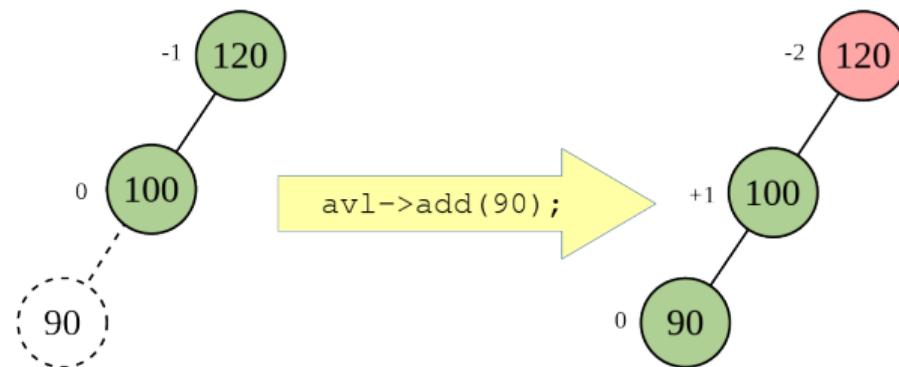
# Manutenção de FB: Inserção à Esquerda de v

- Se, antes da inclusão,  $FB(v) = 0$ , então  $FB(v)$  se tornará  $-1$ 
  - Altura da árvore foi modificada
  - Por consequência, altura dos outros nodos no caminho até a raiz pode ter sido alterada também
  - Repetir o processo (recursivamente), com  $v$  substituído por seu pai



# Manutenção de FB: Inserção à Esquerda de v

- Se, antes da inclusão,  $FB(v) = -1$ , então  $FB(v)$  se tornará  $-2$ 
  - Esse caso só ocorre por propagação de inserção em nodo com  $FB = 0$
  - Altura da árvore foi modificada e o nodo está desbalanceado
  - Rotação correta deve ser empregada
  - Como a árvore será redesenhada, não é necessário verificar os outros nodos

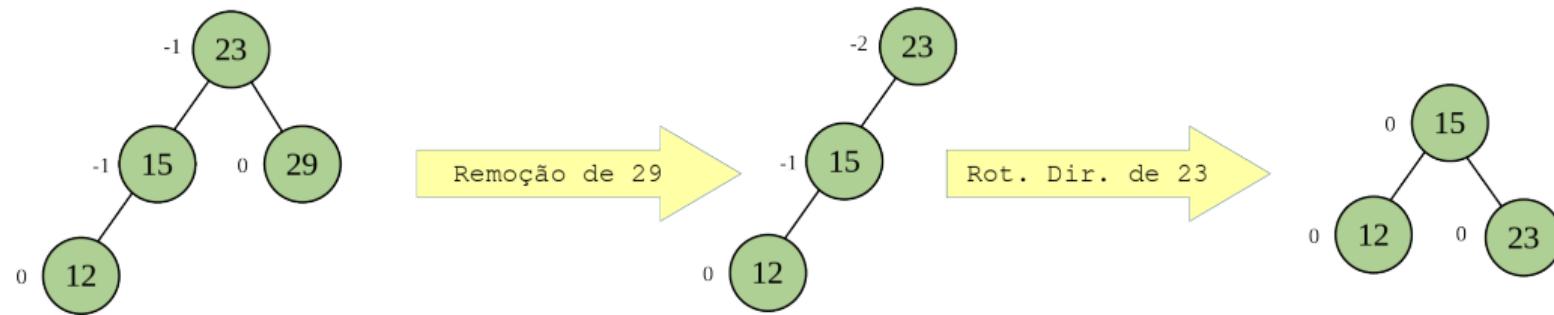


# Remoção de Nós em Árvores AVL

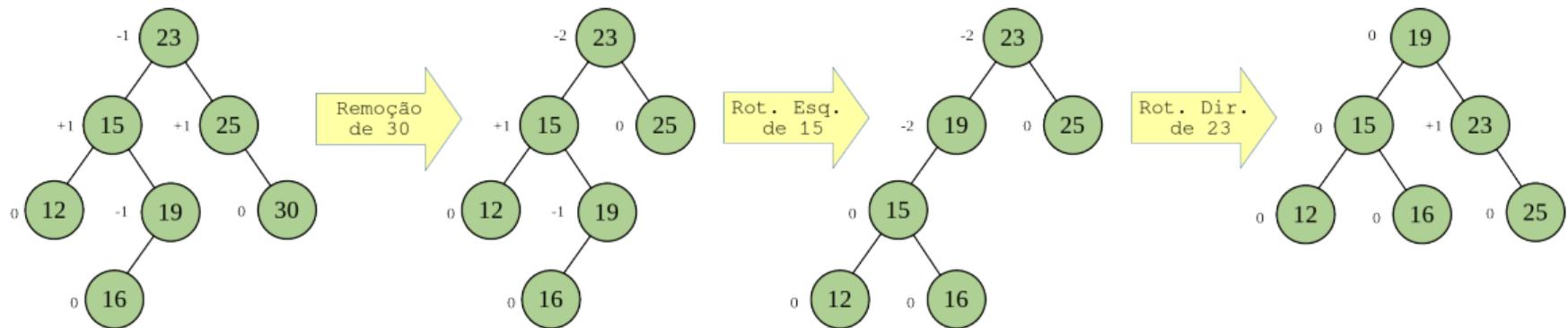
- Parecido com as inclusões

- Realizar a remoção
- Recalcular FB
- Fazer rotações que forem necessárias

## Exemplo 1: Remoção de 29



## Exemplo 1: Remoção de 30



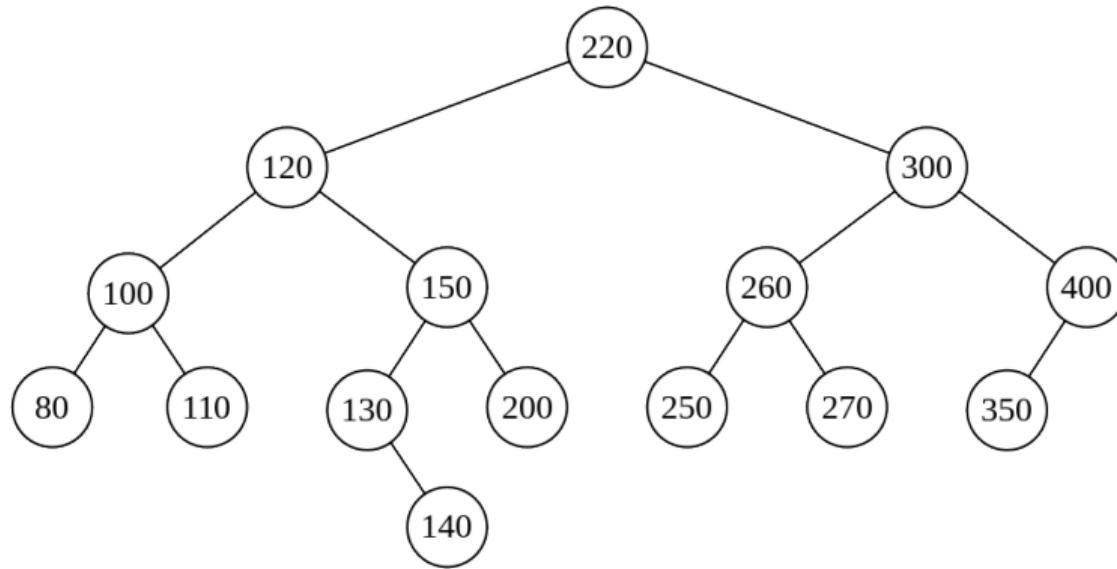
# Remoção de Nodo Intermediário

- Mesmo raciocínio!!
- **Lembrete:** se nodo excluído tem 2 filhos, substituir pelo nodo de maior chave da subárvore esquerda, seguindo o algoritmo de remoção em ABP

# Exercício

## Exercício 22

- 22) Remova os nodos 400, 140, 120, 130, 150, 200, 250 e 350 da árvore AVL abaixo.



# Créditos

# Créditos

- Estas lâminas foram adaptadas de materiais desenvolvidos pelos professores Isabel Harb Manssour e Iaçanã Ianiski Weber.

# Soluções

## Exercício 1

Qual é a altura da árvore?

4

Quais são as folhas da árvore?

H, I, E, F, J, N, L, M

Quais são os nodos irmãos?

B e C; D, E e F; H e I; J, K, L e M

Os nodos D e G são pais de que nodos?

D é pai de H e I; G é pai de J, K, L e M

Qual é o grau do nodo B?

3

Qual é o grau do nodo G?

4

Quais são os níveis dos nodos B, G, H, L e N?

B, nível 1; G, nível 2; H, nível 3; L, nível 3; N, nível 4

## Exercício 2

Qual é a altura da árvore?

3

Quais são as folhas da árvore?

E, F, G, K, I, J

Quais são os nodos irmãos?

B, C e D; E e F; H, I e J

Quais são os nodos internos?

B, C, D e H

Qual é o grau do nodo C?

1

Qual é o grau do nodo D?

3

Quais são os níveis dos nodos C, H e K?

C, nível 1; H, nível 2; K, nível 3

# Exercício 3: exercício03-resp.cpp

```
string strNode(Node *node) {
    stringstream ss;
    if ( node->child1 != nullptr ) ss << "uu" << node->info << "u--u" << node->child1->info << endl << strNode(node->child1);
    if ( node->child2 != nullptr ) ss << "uu" << node->info << "u--u" << node->child2->info << endl << strNode(node->child2);
    if ( node->child3 != nullptr ) ss << "uu" << node->info << "u--u" << node->child3->info << endl << strNode(node->child3);
    return ss.str();
}

string strGraphViz(Node *root) {
    stringstream ss;
    ss << "graph uArvore u{" << endl << "uu" node u[shape=circle]" << endl << strNode(root) << "}" << endl;
    return ss.str();
}

void clean(Node *subtree) {
    if ( subtree->child1 != nullptr ) clean(subtree->child1);
    if ( subtree->child2 != nullptr ) clean(subtree->child2);
    if ( subtree->child3 != nullptr ) clean(subtree->child3);
    delete subtree;
}
```

## Exercício 4: exercício04.cpp

```
#include <iostream>
#include <sstream>

using namespace std;

struct Node {
    char info; Node *child1, *child2, *child3;
    Node(char i, Node *c1 = nullptr, Node *c2 = nullptr, Node *c3 = nullptr) {
        info = i; child1 = c1; child2 = c2; child3 = c3;
        cerr << "+_Node("<< info << ")_criado..." << endl;
    }
    ~Node() { cerr << "-_Node("<< info << ")_destruido..." << endl; }
};

string strNode(Node *node) {
    stringstream ss;
    if ( node->child1 != nullptr ) ss << " _" << node->info << " -- _" << node->child1->info << endl << strNode(node->child1);
    if ( node->child2 != nullptr ) ss << " _" << node->info << " -- _" << node->child2->info << endl << strNode(node->child2);
    if ( node->child3 != nullptr ) ss << " _" << node->info << " -- _" << node->child3->info << endl << strNode(node->child3);
    return ss.str();
}

string strGraphViz(Node *root) {
    stringstream ss;
    ss << "graph _Arvore _{" << endl << " _node _[shape=circle]" << endl << strNode(root) << " }" << endl;
    return ss.str();
}

void clean(Node *subtree) {
    if ( subtree->child1 != nullptr ) clean(subtree->child1);
    if ( subtree->child2 != nullptr ) clean(subtree->child2);
    if ( subtree->child3 != nullptr ) clean(subtree->child3);
    delete subtree;
}
```

## Exercício 4: exercício04.cpp (continuação)

```
int main() {
    Node *e = new Node('E'), *f = new Node('F'),
          *g = new Node('G'), *k = new Node('K'),
          *i = new Node('I'), *j = new Node('J'),
          *b = new Node('B', e, f), *c = new Node('C', g),
          *h = new Node('H', k), *d = new Node('D', h, i, j),
          *root = new Node('A', b, c, d);
    cout << strGraphViz(root);
    clean(root);
    return 0;
}
```

## Exercício 5: exercício05-resp.cpp

```
int degree(Node *subtree) {
    int res = 0;
    if (subtree->left != nullptr) ++res;
    if (subtree->right != nullptr) ++res;
    return res;
}

int depth(Node *subtree) {
    int res = 0;
    Node *aux = subtree->parent;
    while (aux != nullptr) {
        ++res;
        aux = aux->parent;
    }
    return res;
}

int height(Node *subtree) {
    if (subtree == nullptr) return -1; // Somente pode ocorrer se a árvore estiver vazia
    int l = (subtree->left == nullptr)? 0 : (1 + height(subtree->left));
    int r = (subtree->right == nullptr)? 0 : (1 + height(subtree->right));
    return (l > r)?l:r;
}

int size(Node *subtree) {
    if (subtree == nullptr) return 0; // Somente pode ocorrer na primeira chamada,
    int res = 1; // ou seja, se a árvore estiver vazia
    if (subtree->left != nullptr) res += size(subtree->left);
    if (subtree->right != nullptr) res += size(subtree->right);
    return res;
}
```

# Exercício 6: NoteCharTree.cpp

```
#ifdef DEBUG
#include <iostream>
#endif
#include <sstream>
#include "NodeCharTree.hpp"

NodeCharTree::NodeCharTree(char i) {
    info = i; parent = nullptr;
#ifdef DEBUG
    cerr << "+_NodeCharTree(" << info << ")_created..." << endl;
#endif
}

NodeCharTree::~NodeCharTree() {
    for (int i=0; i<child.size(); ++i) delete child[i];
#ifdef DEBUG
    cerr << "-_NodeCharTree(" << info << ")_deleted..." << endl;
#endif
}

char NodeCharTree::getInfo() const { return info; }
void NodeCharTree::setInfo(char i) { info = i; }

NodeCharTree *NodeCharTree::getParent() const { return parent; }

NodeCharTree *NodeCharTree::getChild(int index) const { return (index<0 || index>=child.size())?nullptr:child[index]; }

bool NodeCharTree::isRoot() const { return parent == nullptr; }

bool NodeCharTree::isInternal() const { return parent != nullptr && child.size() != 0; }

bool NodeCharTree::isExternal() const { return child.size()==0; }

int NodeCharTree::degree() const { return child.size(); }

int NodeCharTree::depth() const {
    int res = 0;
    NodeCharTree *aux = parent;
    while (aux != nullptr) {
        ++res;
        aux = aux->getParent();
    }
    return res;
}
```

# Exercício 6: NoteCharTree.cpp (continuação)

```
int NodeCharTree::height() const {
    int res = 0;
    for (int i=0; i<childs.size(); ++i) {
        int h = 1+childs[i]->height();
        if (h > res) res = h;
    }
    return res;
}

int NodeCharTree::size() const {
    int res = 1;
    for (int i=0; i<childs.size(); ++i)
        res += childs[i]->size();
    return res;
}

void NodeCharTree::addSubtree(NodeCharTree *subtree) {
    if (subtree == nullptr) return;
    childs.push_back(subtree);
    subtree->parent = this;
}

bool NodeCharTree::removeSubtree(NodeCharTree *subtree) {
    for (int i=0; i<childs.size(); ++i)
        if (childs[i] == subtree) { childs.erase( childs.begin() + i ); return true; }
    return false;
}

bool NodeCharTree::contains(char i) const {
    if (info == i) return true;
    for (int j=0; j<childs.size(); ++j)
        if (childs[j]->contains(i)) return true;
    return false;
}
```

## Exercício 6: NoteCharTree.cpp (continuação)

```
NodeCharTree const *NodeCharTree::find(char i) const {
    if ( info == i ) return this;
    for (int j=0; j<childs.size(); ++j) {
        NodeCharTree const *child = childs[j]->find(i);
        if ( child != nullptr ) return child;
    }
    return nullptr;
}

string NodeCharTree::strGraphVizNode(NodeCharTree const *node) const {
    stringstream ss;
    for (int i=0; i<node->childs.size(); ++i)
        ss << "  " << node->info << " -- " << node->childs[i]->info << endl << strGraphVizNode(node->childs[i]);
    return ss.str();
}

string NodeCharTree::strGraphViz() const {
    stringstream ss;
    ss << "graph TD" << endl << "  node[shape=circle]" << endl << strGraphVizNode(this) << "}" << endl;
    return ss.str();
}
```

## Exercício 7

a)

Pré-ordem: +, \*, A, B, /, C, +, D, E

Pós-ordem: A, B, \*, C, D, E, +, /, +

Em largura: +, \*, /, A, B, C, +, D, E

b)

Pré-ordem: A, B, D, I, J, M, N, O, E, F, K, C, G, H, L, P, Q

Pós-ordem: I, M, N, O, J, D, E, K, F, B, G, P, Q, L, H, C, A

Em largura: A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q

## Exercício 8: NoteCharTree.cpp (Novos Métodos de Caminhamento)

```
string NodeCharTree::preorder() const {
    stringstream ss;
    ss << info << endl;
    for (int i=0; i<childs.size(); ++i) ss << childs[i]->preorder();
    return ss.str();
}

string NodeCharTree::postorder() const {
    stringstream ss;
    for (int i=0; i<childs.size(); ++i) ss << childs[i]->postorder();
    ss << info << endl;
    return ss.str();
}

string NodeCharTree::levelorder() const {
    stringstream ss;
    vector<const NodeCharTree *> nodes;
    nodes.push_back(this);
    while (nodes.size() != 0) {
        const NodeCharTree *node = nodes[0];
        ss << node->getInfo() << endl;
        nodes.erase(nodes.begin());
        for (int i=0; i<node->degree(); ++i)
            nodes.push_back(node->getChild(i));
    }
    return ss.str();
}
```

# Exercício 9 (DESAFIO): NoteTree.hpp

```

#ifndef _NODETREE_HPP
#define _NODETREE_HPP
#ifndef DEBUG
#include <iostream>
#endif
#include <vector>
#include <sstream>

using namespace std;

template <typename T>
class NodeTree {
private:
    T info;    NodeTree *parent;    vector<NodeTree *> childs;
    string strGraphVizNode(NodeTree const *node) const; // Imprime as conexões de um nodo com seus filhos em GraphViz
public:
    NodeTree(T const &i);                // Cria um nodo com o caractere i, e sem pai
    ~NodeTree();                         // Destroi o nodo atual e seus descendentes
    T getInfo() const;                  // Retorna o caractere armazenado no nodo atual
    void setInfo(T &i);                // Altera o caractere armazenado no nodo atual
    NodeTree *getParent() const;        // Retorna a referência para o nodo-pai do nodo atual
    NodeTree *getChild(int index) const; // Retorna a referência para o nodo-filho de índice index ou nullptr
    bool isRoot() const;                // Retorna true se o nodo for raiz
    bool isInternal() const;            // Retorna true se o nodo for interno
    bool isExternal() const;            // Retorna true se o nodo for externo
    int degree() const;                // Retorna o grau (número de filhos) do nodo atual
    int depth() const;                 // Retorna o nível do nodo atual
    int height() const;                // Retorna a altura da subárvore/árvore a partir do nodo atual
    int size() const;                  // Retorna o número de nodos a partir do nodo atual
    void addSubtree(NodeTree *subtree); // Adiciona uma subárvore como filho do nodo atual
    bool removeSubtree(NodeTree *subtree); // Remove a subárvore dentro os filhos do atual (true indica sucesso)
    bool contains(T &i) const;          // Retorna true se o nodo atual ou algum de seus descendentes contiverem i
    NodeTree const *find(T const &i) const; // Retorna a referência para o nodo, a partir do atual, que contém i ou nullptr
    string strGraphViz() const;         // Gera uma representação da árvore a partir do nodo atual no formato GraphViz
    string preorder() const;            // Gera uma cadeia de caracteres (um nodo por linha) em pré-ordem
    string postorder() const;           // Gera uma cadeia de caracteres (um nodo por linha) em pós-ordem
    string levelorder() const;          // Gera uma cadeia de caracteres (um nodo por linha) em largura
};
```

# Exercício 9 (DESAFIO): NoteTree.hpp (continuação)

```

template <typename T>
NodeTree<T>::NodeTree(T const &i) {
    info = i; parent = nullptr;
#ifndef DEBUG
    cerr << "+_NodeTree<" << info << ")_created..." << endl;
#endif
}

template <typename T>
NodeTree<T>::~NodeTree() {
    for (int i=0; i<childs.size(); ++i) delete childs[i];
#ifndef DEBUG
    cerr << "-_NodeTree<" << info << ")_deleted..." << endl;
#endif
}

template <typename T> T NodeTree<T>::getInfo() const { return info; }
template <typename T> void NodeTree<T>::setInfo(T &i) { info = i; }
template <typename T> NodeTree<T>* NodeTree<T>::getParent() const { return parent; }
template <typename T> NodeTree<T>* NodeTree<T>::getChild(int index) const { return (index<0 || index>=childs.size())?nullptr:childs[index]; }
template <typename T> bool NodeTree<T>::isRoot() const { return parent == nullptr; }
template <typename T> bool NodeTree<T>::isInternal() const { return parent != nullptr && childs.size() != 0; }
template <typename T> bool NodeTree<T>::isExternal() const { return childs.size()==0; }
template <typename T> int NodeTree<T>::degree() const { return childs.size(); }

template <typename T>
int NodeTree<T>::depth() const {
    int res = 0;
    NodeTree *aux = parent;
    while (aux != nullptr) {
        ++res;
        aux = aux->getParent();
    }
    return res;
}

```

# Exercício 9 (DESAFIO): NoteTree.hpp (continuação)

```
template <typename T>
int NodeTree<T>::height() const {
    int res = 0;
    for (int i=0; i<child.size(); ++i) {
        int h = child[i]->height();
        if (h > res) res = h;
    }
    return res;
}

template <typename T>
int NodeTree<T>::size() const {
    int res = 1;
    if (child.size() == 0) return res;
    for (int i=0; i<child.size(); ++i)
        res += child[i]->size();
    return res;
}

template <typename T>
void NodeTree<T>::addSubtree(NodeTree *subtree) {
    if (subtree == nullptr) return;
    child.push_back(subtree);
    subtree->parent = this;
}

template <typename T>
bool NodeTree<T>::removeSubtree(NodeTree *subtree) {
    for (int i=0; i<child.size(); ++i)
        if (child[i] == subtree) { child.erase( child.begin() + i ); return true; }
    return false;
}
```

# Exercício 9 (DESAFIO): NoteTree.hpp (continuação)

```
template <typename T>
bool NodeTree<T>::contains(T &i) const {
    if (info == i) return true;
    for (int j=0; j<child.size(); ++j)
        if (child[j]->contains(i)) return true;
    return false;
}

template <typename T>
NodeTree<T> const *NodeTree<T>::find(T const &i) const {
    if (info == i) return this;
    for (int j=0; j<child.size(); ++j) {
        NodeTree const *child = child[j]->find(i);
        if (child != nullptr) return child;
    }
    return nullptr;
}

template <typename T>
string NodeTree<T>::strGraphVizNode(NodeTree const *node) const {
    stringstream ss;
    for (int i=0; i<node->child.size(); ++i)
        ss << "u" << node->info << "--u" << node->child[i]->info << endl << strGraphVizNode(node->child[i]);
    return ss.str();
}

template <typename T>
string NodeTree<T>::strGraphViz() const {
    stringstream ss;
    ss << "graph NodeTree{" << endl << "u" << node[shape=circle]" << endl << strGraphVizNode(this) << "}" << endl;
    return ss.str();
}
```

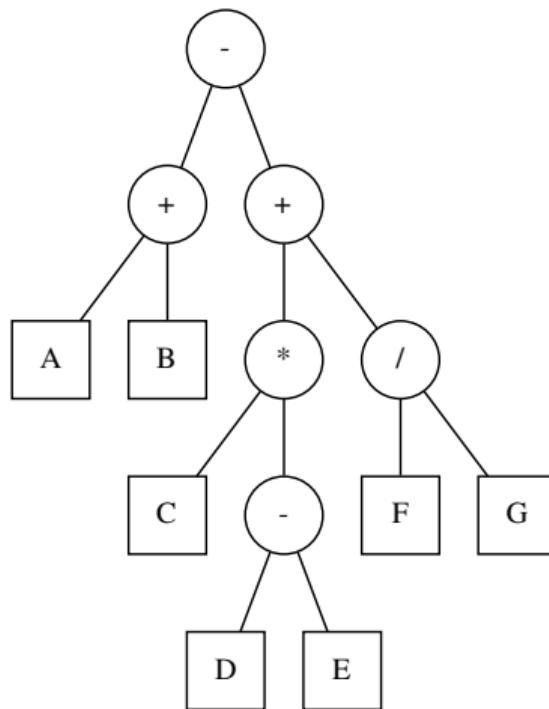
# Exercício 9 (DESAFIO): NoteTree.hpp (continuação)

```
template <typename T>
string NodeTree<T>::preorder() const {
    stringstream ss;
    ss << info << endl;
    for (int i=0; i<childs.size(); ++i) ss << childs[i]->preorder();
    return ss.str();
}

template <typename T>
string NodeTree<T>::postorder() const {
    stringstream ss;
    for (int i=0; i<childs.size(); ++i) ss << childs[i]->postorder();
    ss << info << endl;
    return ss.str();
}

template <typename T>
string NodeTree<T>::levelorder() const {
    stringstream ss;
    vector<const NodeTree *> nodes;
    nodes.push_back(this);
    while (nodes.size() != 0) {
        const NodeTree *node = nodes[0];
        ss << node->getInfo() << endl;
        nodes.erase(nodes.begin());
        for (int i=0; i<node->degree(); ++i)
            nodes.push_back(node->getChild(i));
    }
    return ss.str();
}
#endif
```

## Exercício 11



# Exercício 12

a)

Pré-ordem: A, B, G, C, D, E, F

Pós-ordem: G, F, E, D, C, B, A

Central: G, B, C, E, F, D, A

Em largura: A, B, G, C, D, E, F

b)

Pré-ordem: X, Y, Z, A, W, B, C, D, E

Pós-ordem: A, B, C, W, Z, D, Y, E, X

Central: A, Z, B, W, C, Y, D, X, E

Em largura: X, Y, E, Z, D, A, W, B, C

c)

Pré-ordem: 1, 2, 4, 6, 3, 5, 7

Pós-ordem: 6, 4, 2, 7, 5, 3, 1

Central: 4, 6, 2, 1, 5, 7, 3

Em largura: 1, 2, 3, 4, 5, 6, 7

d)

Pré-ordem: -, +, A, B, -, /, C, G, \*, F, -, D, E

Pós-ordem: A, B, +, C, G, /, F, D, E, -, \*, -, -

Central: A, +, B, -, C, /, G, -, F, \*, D, -, E

Em largura: -, +, -, A, B, /, \*, C, G, F, -, D, E

# Exercício 13: NoteBT.hpp

```

#ifndef _NODEBT_HPP
#define _NODEBT_HPP
#ifndef DEBUG
#include <iostream>
#endif
#include <vector>
#include <sstream>

using namespace std;

template <typename T>
class NodeBT {
private:
    T info;    NodeBT *parent, *left, *right;
    string strGraphVizNode(NodeBT const *node) const; // Imprime as conexões de um nodo com seus filhos em GraphViz
public:
    NodeBT(T const &i);                                // Cria um nodo com o caractere i, e sem pai
    ~NodeBT();                                         // Destroi o nodo atual e seus descendentes
    T getInfo() const;                                 // Retorna o caractere armazenado no nodo atual
    void setInfo(T &i);                               // Altera o caractere armazenado no nodo atual
    NodeBT *getParent() const;                         // Retorna a referência para o nodo-pai do nodo atual
    NodeBT *getLeft() const;                           // Retorna a referência para o nodo-filho/subárvore da esquerda
    NodeBT *getRight() const;                          // Retorna a referência para o nodo-filho/subárvore da direita
    void setLeft(NodeBT *subtree);                    // Adiciona uma subárvore como subárvore da esquerda do nodo atual
    void setRight(NodeBT *subtree);                   // Adiciona uma subárvore como subárvore da direita do nodo atual
    void removeLeft();                                // Remove a subárvore da esquerda (desalocando-a, se necessário)
    void removeRight();                              // Remove a subárvore da direita (desalocando-a, se necessário)
    bool isRoot() const;                            // Retorna true se o nodo for raiz
    bool isInternal() const;                         // Retorna true se o nodo for interno
    bool isExternal() const;                         // Retorna true se o nodo for externo
    int degree() const;                             // Retorna o grau (número de filhos) do nodo atual
    int depth() const;                            // Retorna o nível do nodo atual
    int height() const;                           // Retorna a altura da subárvore/árvore a partir do nodo atual
    int size() const;                            // Retorna o número de nodos a partir do nodo atual
    bool contains(T &i) const;                     // Retorna true se o nodo atual ou algum de seus descendentes contiverem i
    NodeBT const *find(T const &i) const;          // Retorna a referência para o nodo, a partir do atual, que contém i ou nullptr
    string strGraphViz() const;                    // Gera uma representação da árvore a partir do nodo atual no formato GraphViz
    string preorder() const;                       // Gera uma cadeia de caracteres (um nodo por linha) em pré-ordem
}

```



# Exercício 13: NoteBT.hpp (continuação)

```

string postorder() const;           // Gera uma cadeia de caracteres (um nodo por linha) em pós-ordem
string inorder() const;            // Gera uma cadeia de caracteres (um nodo por linha) em in-ordem
string levelorder() const;         // Gera uma cadeia de caracteres (um nodo por linha) em largura
};

template <typename T>
NodeBT<T>::NodeBT(T const &i) {
    info = i; parent = left = right = nullptr;
    #ifdef DEBUG
    cerr << "+_NodeBT<" << info << ")_created..." << endl;
    #endif
}

template <typename T>
NodeBT<T>::~NodeBT() {
    delete left; delete right;
    #ifdef DEBUG
    cerr << "-_NodeBT<" << info << ")_deleted..." << endl;
    #endif
}

template <typename T> T NodeBT<T>::getInfo() const { return info; }
template <typename T> void NodeBT<T>::setInfo(T &i) { info = i; }
template <typename T> NodeBT<T> *NodeBT<T>::getParent() const { return parent; }
template <typename T> NodeBT<T> *NodeBT<T>::getLeft() const { return left; }
template <typename T> NodeBT<T> *NodeBT<T>::getRight() const { return right; }

template <typename T> void NodeBT<T>::setLeft(NodeBT *subtree) {
    left = subtree;
    if (subtree != nullptr) subtree->parent = this;
}

template <typename T> void NodeBT<T>::setRight(NodeBT *subtree) {
    right = subtree;
    if (subtree != nullptr) subtree->parent = this;
}

```

## Exercício 13: NoteBT.hpp (continuação)

```
template <typename T> void NodeBT<T>::removeLeft() { if (left != nullptr) { delete left; left = nullptr; } }
template <typename T> void NodeBT<T>::removeRight() { if (right != nullptr) { delete right; right = nullptr; } }
template <typename T> bool NodeBT<T>::isRoot() const { return parent == nullptr; }
template <typename T> bool NodeBT<T>::isInternal() const { return parent != nullptr && (left!=nullptr || right!=nullptr); }
template <typename T> bool NodeBT<T>::isExternal() const { return left==nullptr && right==nullptr; }

template <typename T> int NodeBT<T>::degree() const {
    int res = 0;
    if (left != nullptr) ++res;
    if (right != nullptr) ++res;
    return res;
}

template <typename T>
int NodeBT<T>::depth() const {
    int res = 0;
    NodeBT *aux = parent;
    while (aux != nullptr) {
        ++res;
        aux = aux->getParent();
    }
    return res;
}

template <typename T>
int NodeBT<T>::height() const {
    int l = (left == nullptr)? 0 : (1+left->height());
    int r = (right == nullptr)? 0 : (1+right->height());
    return (l>r)?l:r;
}

template <typename T>
int NodeBT<T>::size() const {
    int res = 1;
    if (left != nullptr) res += left->size();
    if (right != nullptr) res += right->size();
    return res;
}
```

# Exercício 13: NoteBT.hpp (continuação)

```

template <typename T>
bool NodeBT<T>::contains(T &i) const {
    if ( info == i ) return true;
    if ( left != nullptr && left->contains(i) ) return true;
    if ( right != nullptr && right->contains(i) ) return true;
    return false;
}

template <typename T>
NodeBT<T> const *NodeBT<T>::find(T const &i) const {
    if ( info == i ) return this;
    if ( left != nullptr ) {
        NodeBT const *child = left->find(i);
        if ( child != nullptr ) return child;
    }
    if ( right != nullptr ) {
        NodeBT const *child = right->find(i);
        if ( child != nullptr ) return child;
    }
    return nullptr;
}

template <typename T>
string NodeBT<T>::strGraphVizNode(NodeBT const *node) const {
    stringstream ss;
    if ( node->left != nullptr ) ss << "uu" << node->info << "u--u" << node->left->info << endl << strGraphVizNode(node->left);
    if ( node->right != nullptr ) ss << "uu" << node->info << "u--u" << node->right->info << endl << strGraphVizNode(node->right);
    return ss.str();
}

template <typename T>
string NodeBT<T>::strGraphViz() const {
    stringstream ss;
    ss << "graph BT" << endl << "  node [shape=circle]" << endl << strGraphVizNode(this) << "}" << endl;
    return ss.str();
}

```

## Exercício 13: NoteBT.hpp (continuação)

```
template <typename T>
string NodeBT<T>::preorder() const {
    stringstream ss;
    ss << info << endl;
    if (left != nullptr) ss << left->preorder();
    if (right != nullptr) ss << right->preorder();
    return ss.str();
}

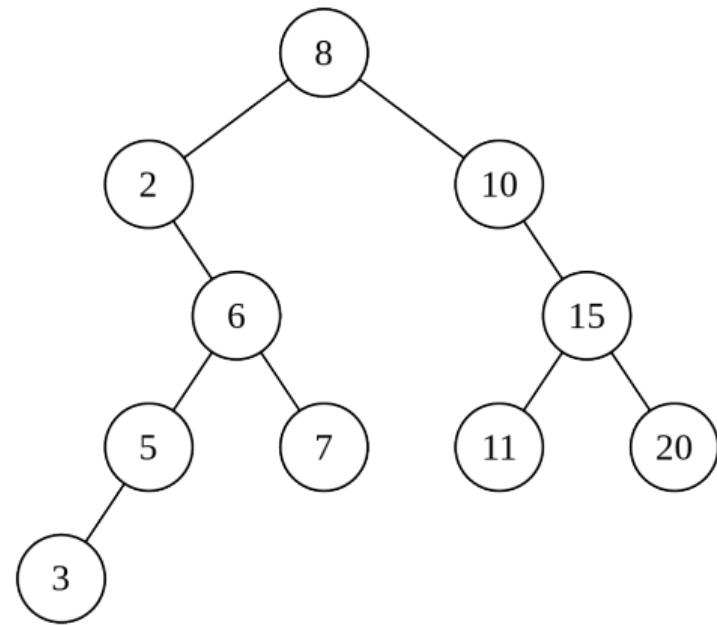
template <typename T>
string NodeBT<T>::postorder() const {
    stringstream ss;
    if (left != nullptr) ss << left->postorder();
    if (right != nullptr) ss << right->postorder();
    ss << info << endl;
    return ss.str();
}

template <typename T>
string NodeBT<T>::inorder() const {
    stringstream ss;
    if (left != nullptr) ss << left->inorder();
    ss << info << endl;
    if (right != nullptr) ss << right->inorder();
    return ss.str();
}
```

## Exercício 13: NoteBT.hpp (continuação)

```
template <typename T>
string NodeBT<T>::levelorder() const {
    stringstream ss;
    vector<const NodeBT *> nodes;
    nodes.push_back(this);
    while (nodes.size() != 0) {
        const NodeBT *node = nodes[0];
        ss << node->getInfo() << endl;
        nodes.erase(nodes.begin());
        if (node->left != nullptr) nodes.push_back(node->left);
        if (node->right != nullptr) nodes.push_back(node->right);
    }
    return ss.str();
}
#endif
```

## Exercício 14



# Exercício 15: BST.hpp

```
#ifndef _BST_HPP
#define _BST_HPP
#include "NodeBT.hpp"

using namespace std;

template <typename T>
class BST {
private:
    void add(NodeBT<T> *node, NodeBT<T> *newNode);
    bool contains(NodeBT<T> *node, const T &e);
    NodeBT<T> *root;
public:
    BST();
    ~BST();
    string strGraphViz() const;
    void add(const T &e);
    bool contains(const T &e);
};

template <typename T>
BST<T>::BST() {
    root = nullptr;
}

template <typename T>
BST<T>::~BST() {
    if (root != nullptr) delete root;
}

template <typename T>
string BST<T>::strGraphViz() const {
    if (root == nullptr) return ""; else return root->strGraphViz();
}
```

## Exercício 15: BST.hpp (continuação)

```
template <typename T>
void BST<T>::add(NodeBT<T> *node, NodeBT<T> *newNode) {
    if (newNode->getInfo() < node->getInfo()) { // Left
        NodeBT<T> *left = node->getLeft();
        if (left == nullptr) node->setLeft(newNode);
        else add(left, newNode);
    } else { // Right
        NodeBT<T> *right = node->getRight();
        if (right == nullptr) node->setRight(newNode);
        else add(right, newNode);
    }
}

template <typename T>
void BST<T>::add(const T &e) {
    NodeBT<T> *newNode = new NodeBT<T>(e);
    if (root == nullptr) root = newNode;
    else add(root, newNode);
}
```

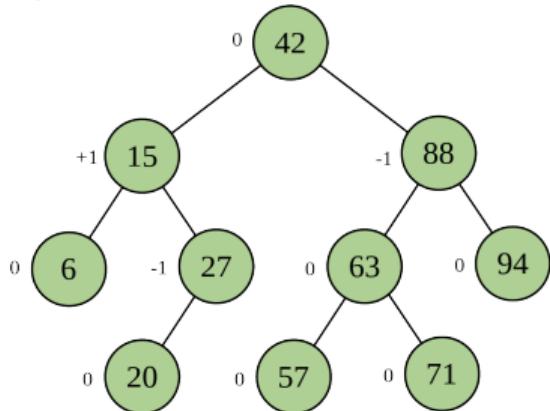
## Exercício 15: BST.hpp (continuação)

```
template <typename T>
bool BST<T>::contains(NodeBT<T> *node, const T &e) {
    if ( e == node->getInfo() ) return true;
    if ( e < node->getInfo() ) { // Left
        NodeBT<T> *left = node->getLeft();
        if (left == nullptr) return false;
        else return contains(left,e);
    }
    else { // Right
        NodeBT<T> *right = node->getRight();
        if (right == nullptr) return false;
        else return contains(right,e);
    }
}

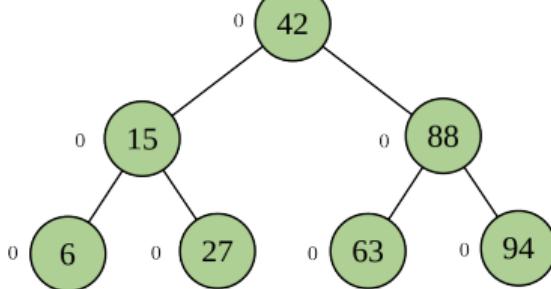
template <typename T>
bool BST<T>::contains(const T &e) {
    if (root == nullptr) return false;
    return contains(root, e);
}
#endif
```

## Exercício 16

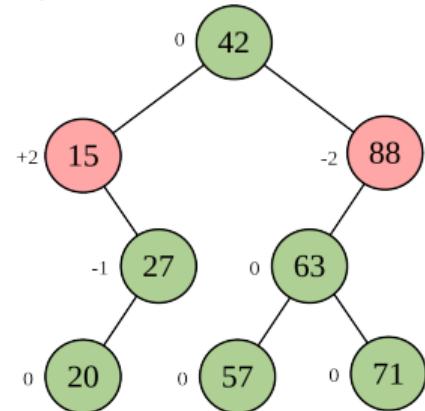
a)



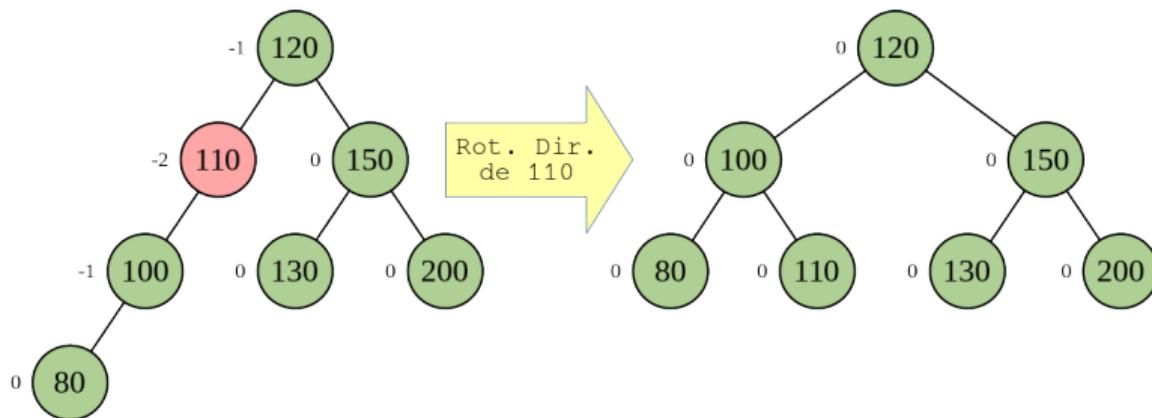
b)



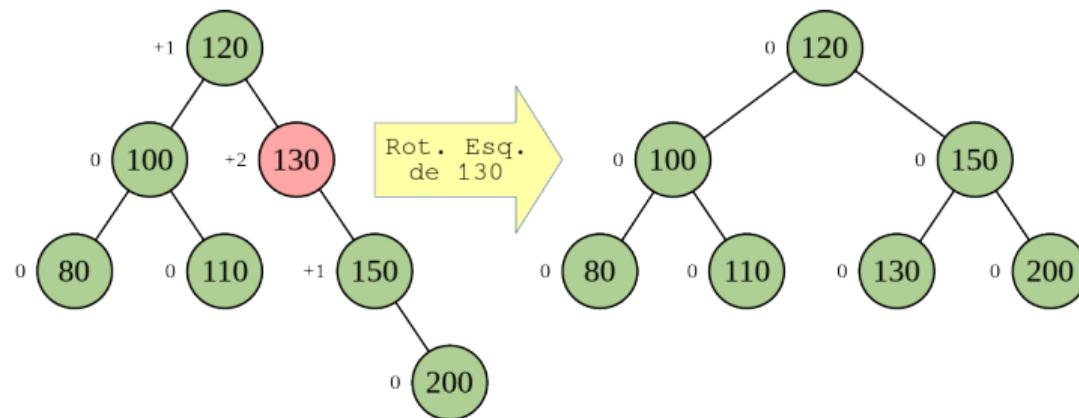
c)



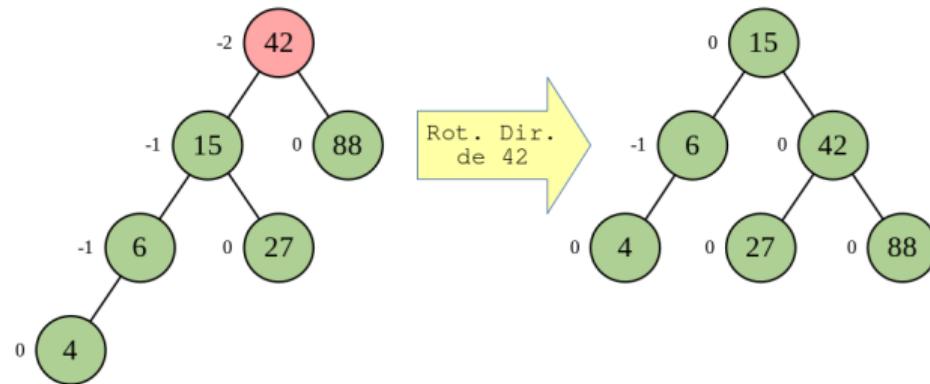
## Exercício 17a



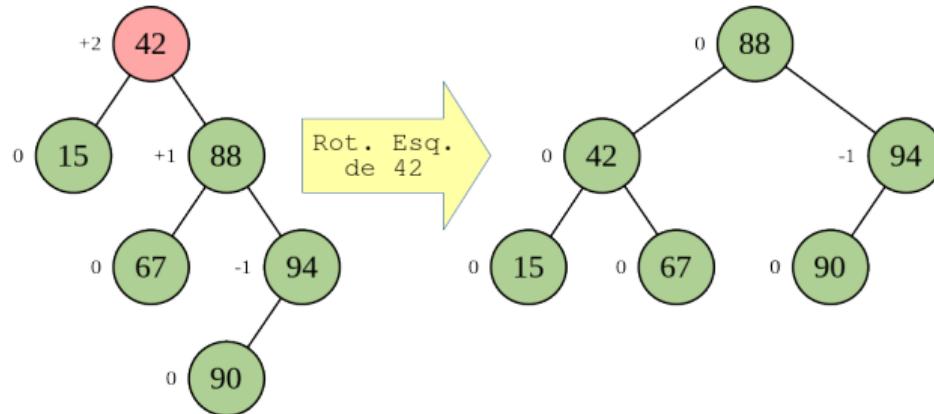
## Exercício 17b



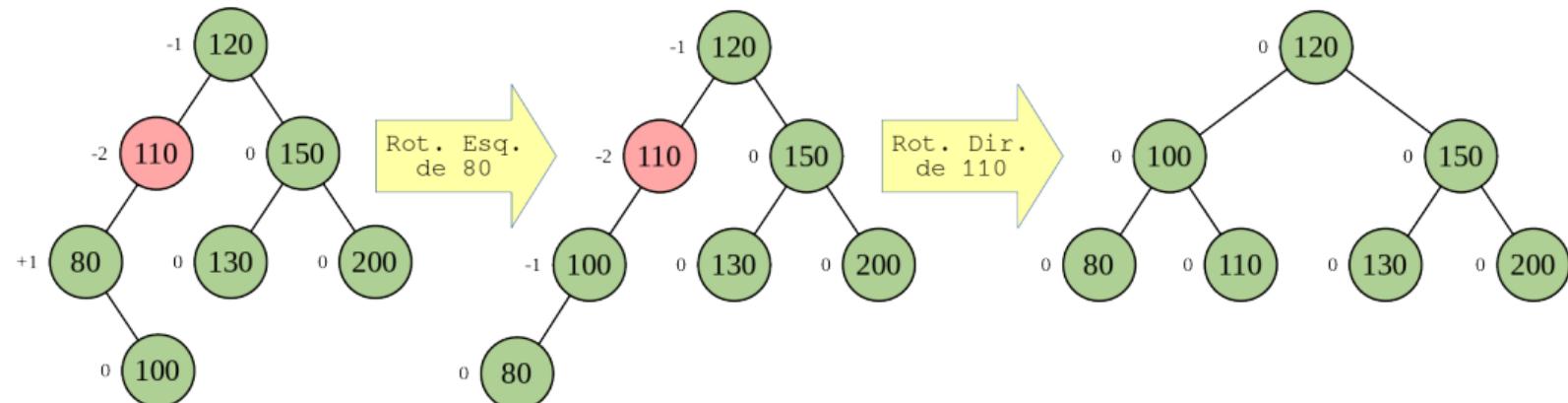
## Exercício 18a



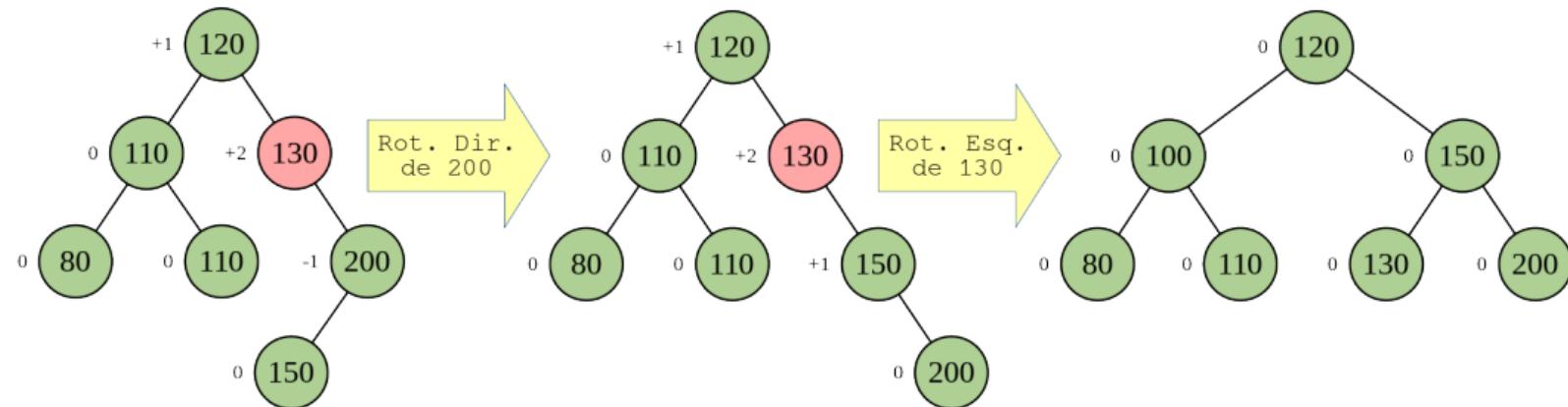
## Exercício 18b



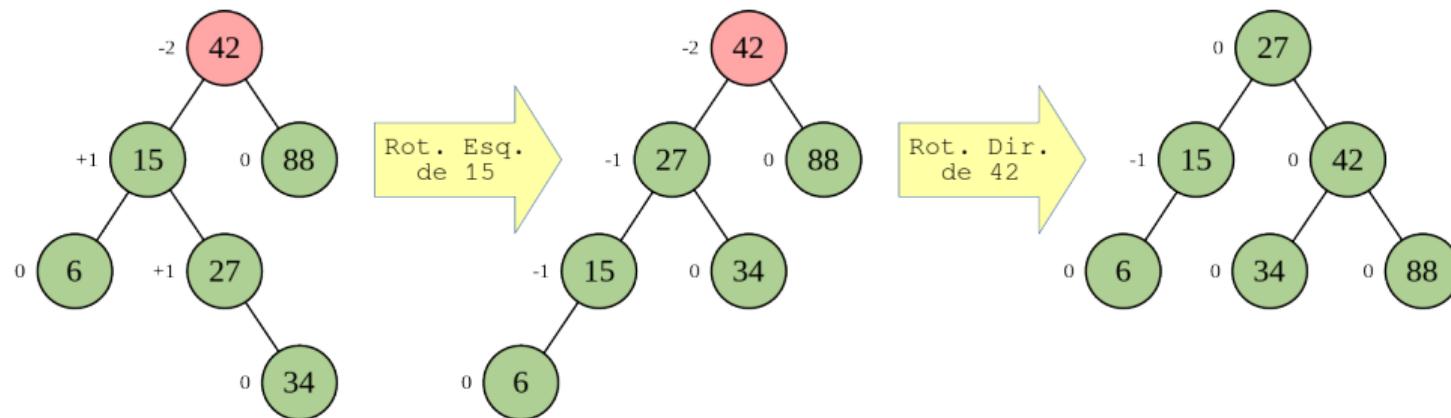
## Exercício 19a



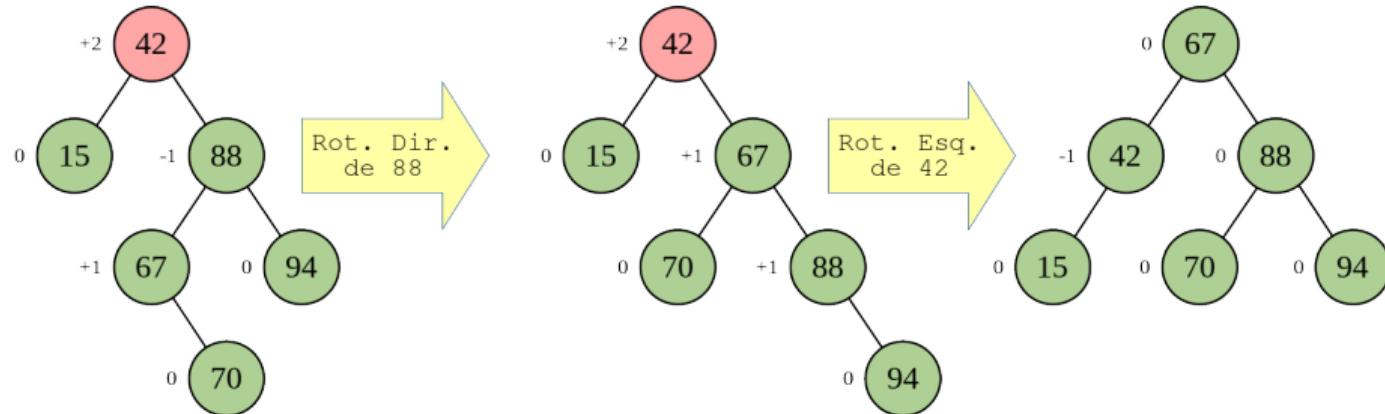
## Exercício 19b



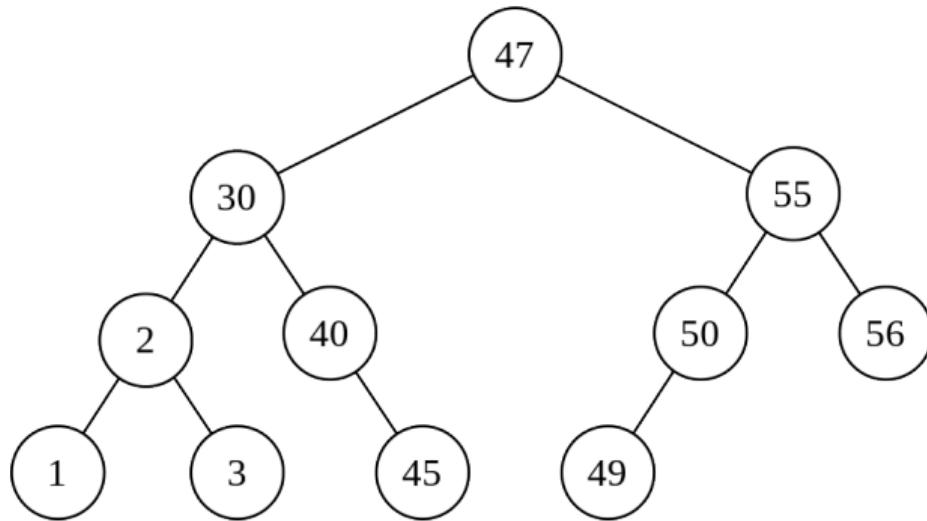
## Exercício 20a



## Exercício 20b

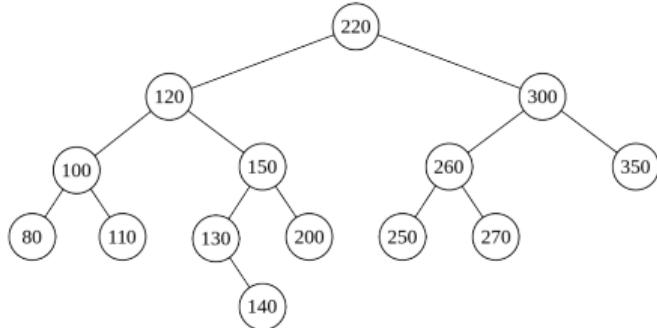


## Exercício 21

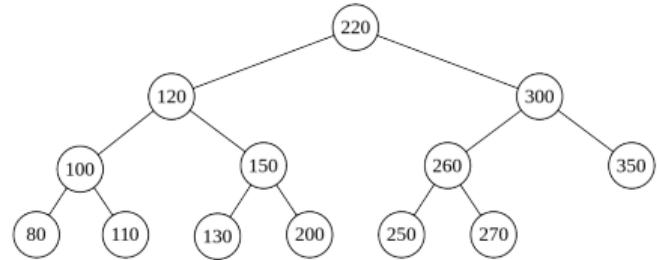


## Exercício 22

```
avl22->remove(400);
```

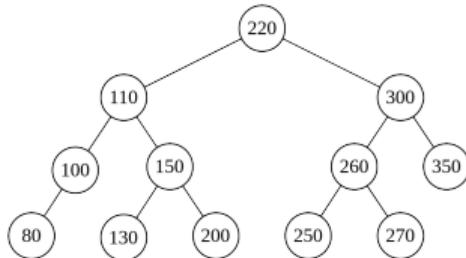


```
avl22->remove(140);
```

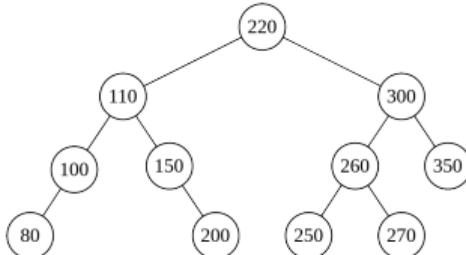


## Exercício 22 (cont.)

```
avl22->remove(120);
```

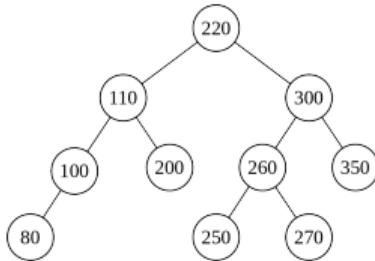


```
avl22->remove(130);
```

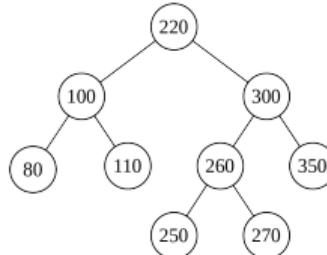


## Exercício 22 (cont.)

```
avl22->remove(150);
```

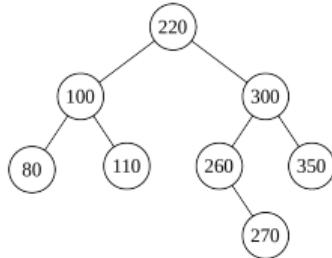


```
avl22->remove(200);
```



## Exercício 22 (cont.)

```
avl22->remove(250);
```



```
avl22->remove(350);
```

