

Árvores

Algoritmos e Estruturas de Dados

Prof. Daniel Guerreiro e Silva

Roteiro

- Introdução
 - Árvore de Busca Binária
 - Busca e Inserção na BST
 - Remoção na BST
 - Árvores balanceadas
-
- Leitura sugerida: Seções 6.1 a 6.7 do livro-texto (Drozdek)

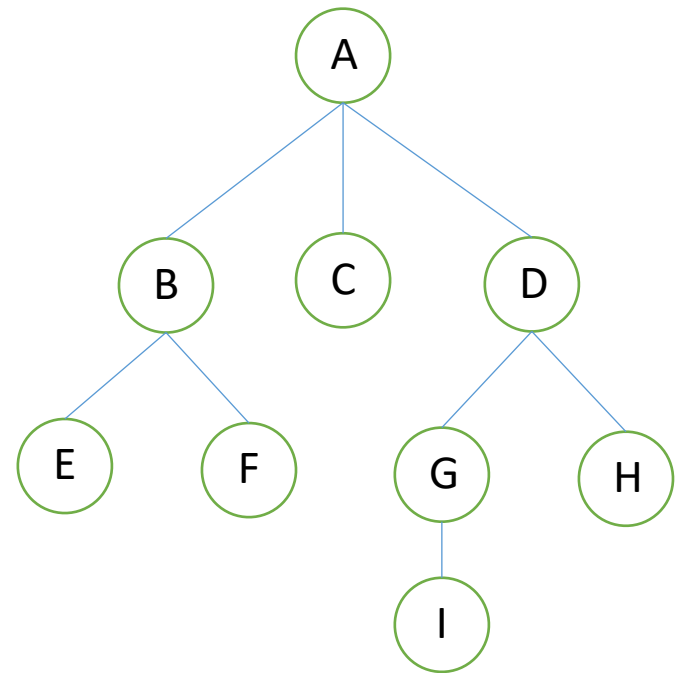
Introdução

Árvores

- São estruturas de dados que permitem implementar uma **relação de hierarquia** entre os elementos
- É composta de **nós** (vértices) e **arcos** (arestas)

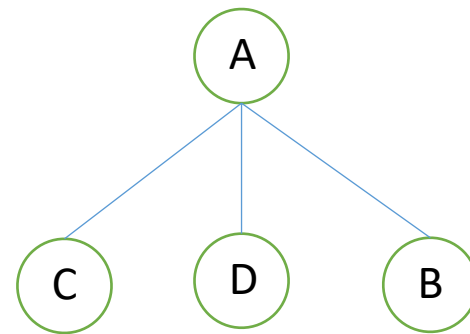
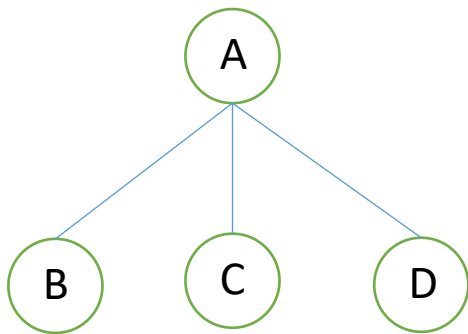
Definição

- Uma árvore do tipo T é constituída de
 - Uma estrutura vazia, ou
 - Um elemento ou nó do tipo T chamado **raiz**, com arcos para um número finito de árvores do tipo T, chamadas de **sub-árvores** da raiz



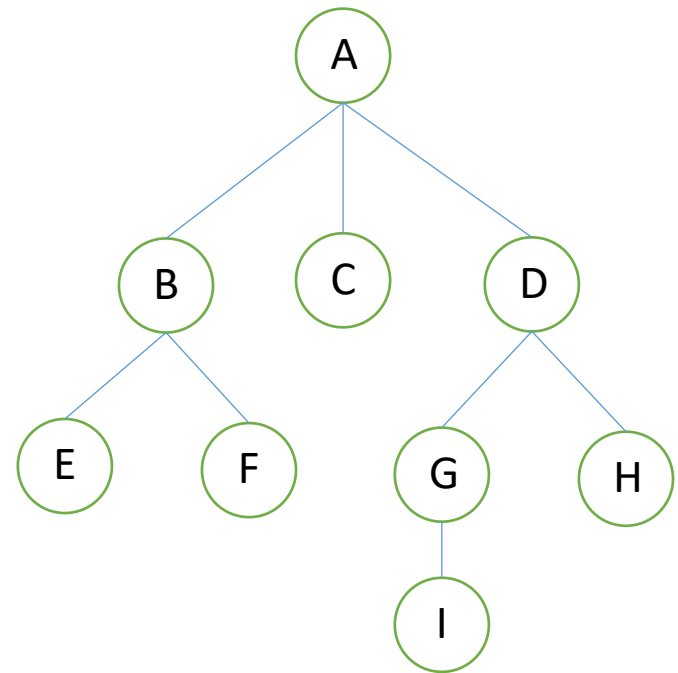
Árvore ordenada

- Uma árvore é dita ordenada quando a ordem das sub-árvores é relevante.
- Neste sentido, as duas árvores a seguir são diferentes



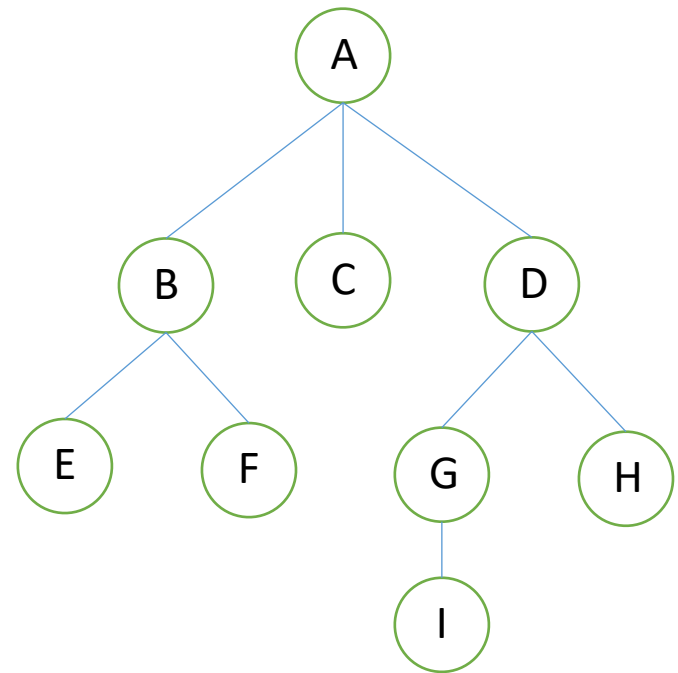
Nomenclatura de árvores

- **Pai** (parent) e **filho** (child): um nó y abaixo de um nó x é chamado de filho de x , enquanto x é denominado pai de y
 - Ex.: B é pai de E e F
- **Irmão**: Nós com o mesmo pai são denominados irmãos
 - Ex.: B, C e D são irmãos
- **Nível** de um nó: a raiz de uma árvore tem nível 1, se um nó tem nível i , então seus filhos têm nível $i+1$.
 - Ex.: E, F, G e H têm nível 3, I tem nível 4



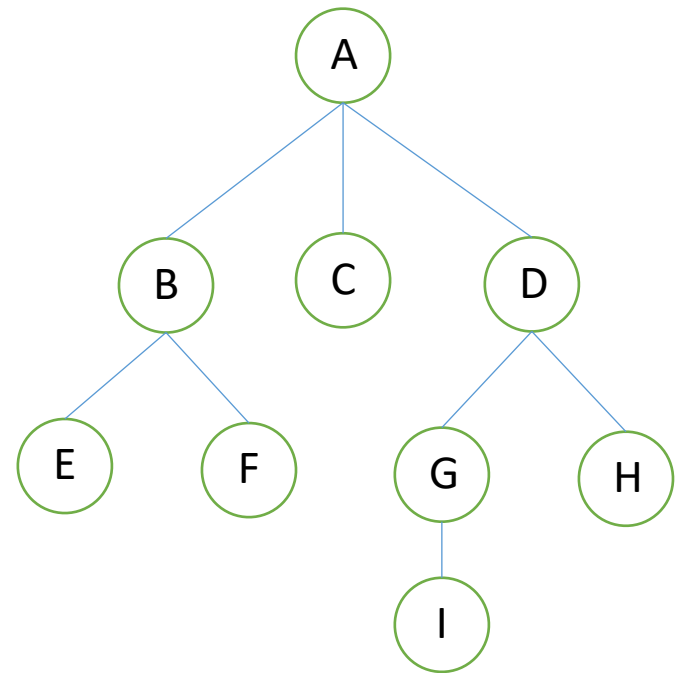
Nomenclatura de árvores

- **Altura ou Profundidade:** é o máximo nível dos nós da árvore.
 - Ex.: a árvore ao lado tem altura 4
- **Folha:** é um nó que não tem filhos
 - Ex.: E, F, C, I e H são folhas
- **Nó interno:** é um nó que não é folha e nem raiz.
 - Ex.: B, D, G



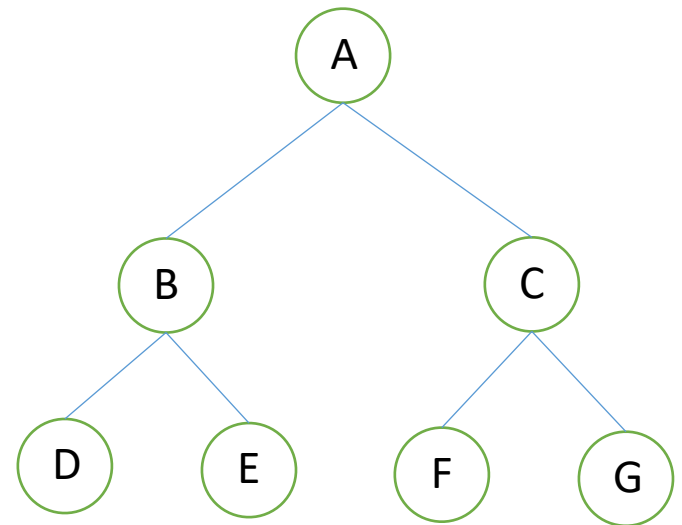
Nomenclatura de árvores

- **Grau de um nó:** é o número de filhos de um nó
 - Ex.: B tem grau 2, G tem grau 1
- **Grau de uma árvore:** é o máximo grau de seus nós
 - Ex.: a árvore tem grau 3



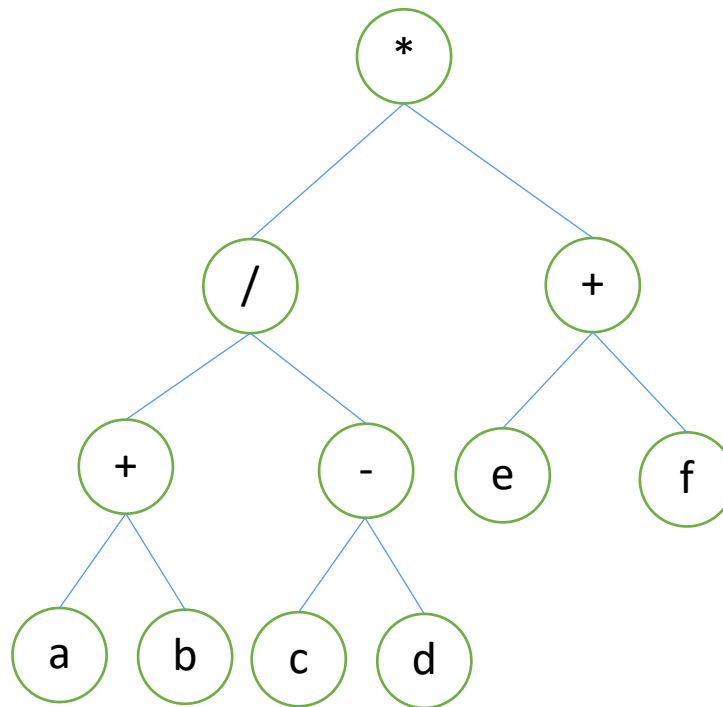
Árvore binária

- É uma árvore ordenada de grau 2.
- Uma árvore binária é
 - vazia, ou
 - um nó raiz mais duas subárvores disjuntas chamadas **subárvore esquerda** e **subárvore direita**



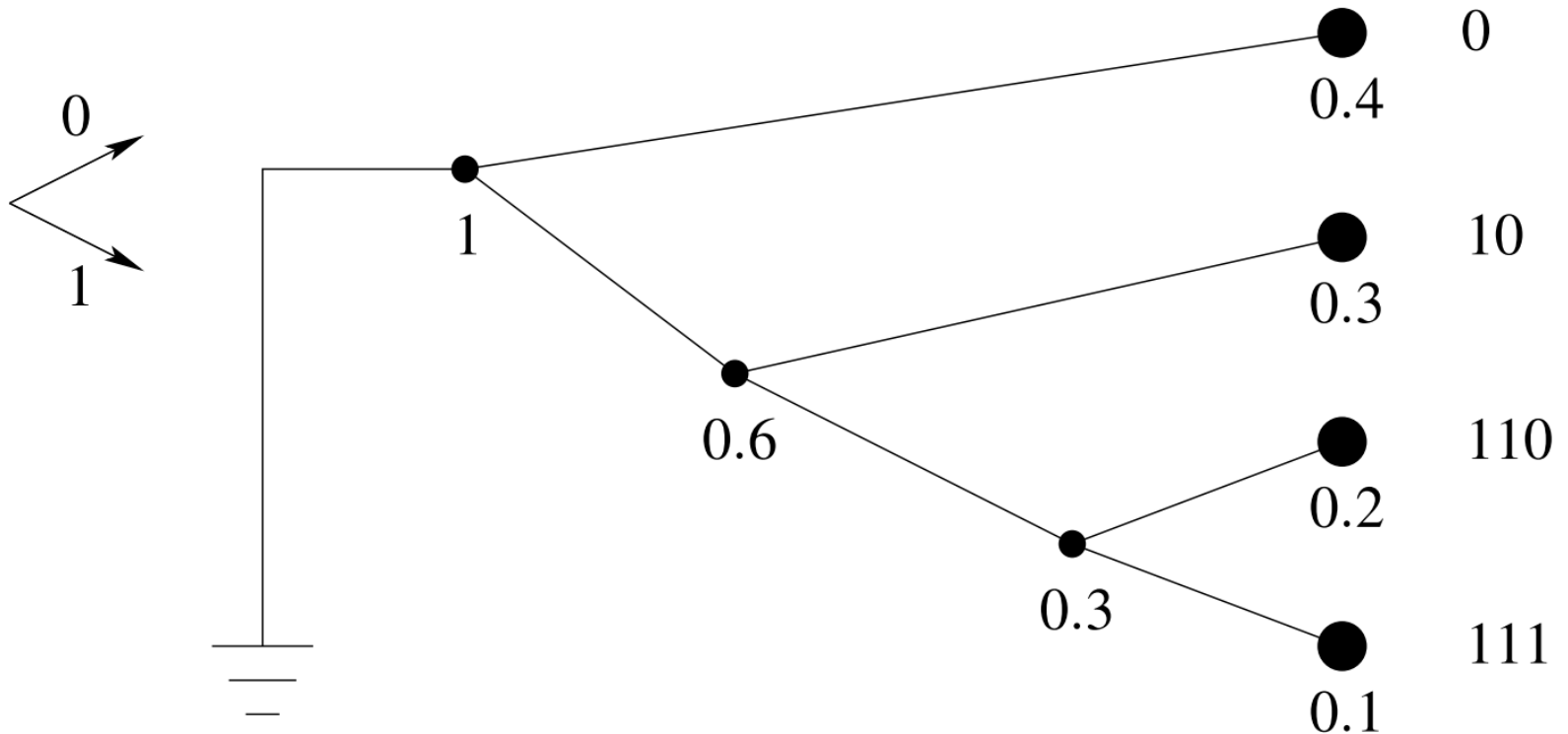
Árvore binária - Exemplo

- Representação da expressão aritmética
 $((a + b) / (c - d)) * (e + f)$



Árvore binária - Exemplo

- Algoritmo de compressão de Huffman



Aplicações de árvores e árvores binárias

- Problemas de busca na memória principal do computador: árvore binária de busca, árvore AVL
- Problemas de busca de dados na memória secundária (disco rígido): árvores B
- Problemas de busca e resolução de problemas na Inteligência Artificial, ex.: jogo de xadrez, árvores de decisão
- Compressão de dados: código de Huffman
- Entre outras...

Aplicações de árvores

- A estrutura de uma árvore ordenada permite realizar a busca por um dado sem necessariamente explorar todos os nós para encontrá-lo.
- Exemplo: considere a localização de um elemento em uma lista duplamente ligada com 10.000 elementos
 - Mesmo ordenada e com apontadores para cabeça e cauda da lista, caso se deseje obter o elemento do meio, temos que acessar os 4.999 anteriores.
 - Por outro lado, se os elementos estão numa árvore ordenada, o número de acessos na busca pode cair drasticamente, como veremos adiante.

Árvore de Busca Binária

A árvore de busca binária (BST, *Binary Search Tree*)

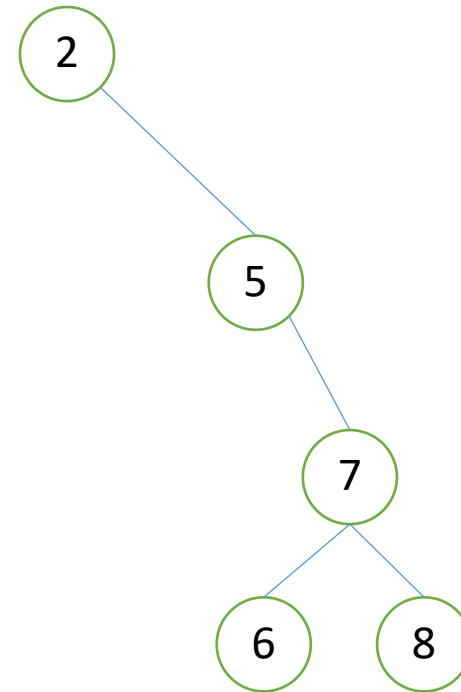
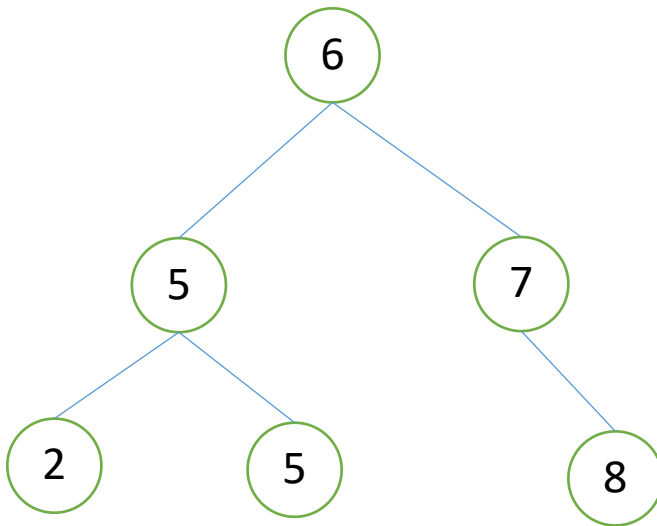
- As chaves dos dados armazenados numa BST obedecem à seguinte propriedade:

“Seja x um nó na BST. Se y é um nó na sub-árvore esquerda de x , então $y.key < x.key$. Se y é um nó na sub-árvore direita de x , então $y.key \geq x.key$ ”

- A propriedade BST permite que se imprima todas as chaves na árvore de forma ordenada, por meio de um algoritmo recursivo simples, chamado de percurso in-ordem da árvore

Árvore de busca binária

- Exemplos:



Implementando uma árvore de busca binária

- O nó de uma árvore é instância de uma classe contendo a informação a armazenar, e dois apontadores para as sub-árvores à esquerda e à direita.
- A classe árvore envolve um apontador para a raiz da árvore, que por sua vez apontará indiretamente para os seus nós sucessores
- Operações principais
 - Visitar os nós de uma árvore
 - Buscar um elemento na árvore
 - Inserir um nó na árvore
 - Apagar um nó na árvore

Implementação do nó

```
template<class T>
class BSTNode {
public:
    //CONSTRUTORES
    BSTNode() {
        left = right = 0;
    }
    BSTNode(const T& el, BSTNode<T> *left = 0, BSTNode<T> *right = 0) {
        this->el = el;
        this->left = left;
        this->right = right;
    }
    //ATRIBUTOS
    T el; //elemento a se armazenar
    BSTNode<T> *left, *right; //apontadores para subarvore esquerda e direita
};
```

Implementação da árvore (parcial)

```
template<class T>
class BST {
public:
    //metodos publicos...

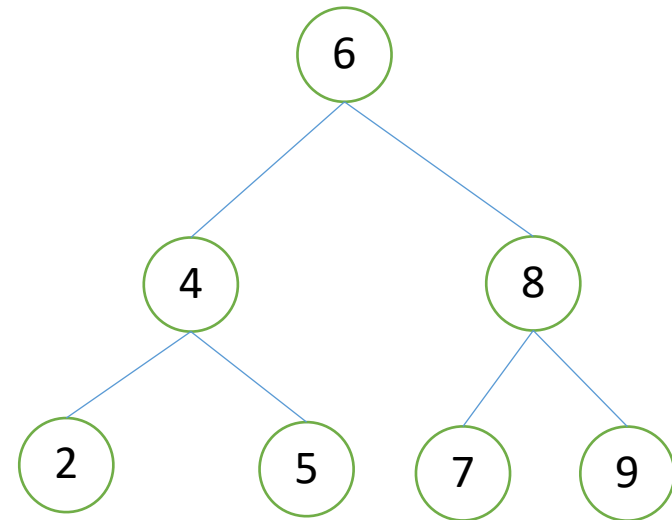
protected:
    BSTNode<T>* root; //atributo - raiz da arvore
    //métodos protegidos...
};
```

Percorrer/visitar os nós de uma árvore

- Percurso em **profundidade**
 - **Pré-ordem**: visita o nó atual, sub-árvore esquerda, sub-árvore direita
 - **In-ordem**: visita sub-árvore esquerda, o nó atual, sub-árvore direita
 - **Pós-ordem**: visita sub-árvore esquerda, sub-árvore direita, nó atual
- Percurso em **largura**
 - Visita todos os nós de cada nível da árvore, da esquerda para a direita

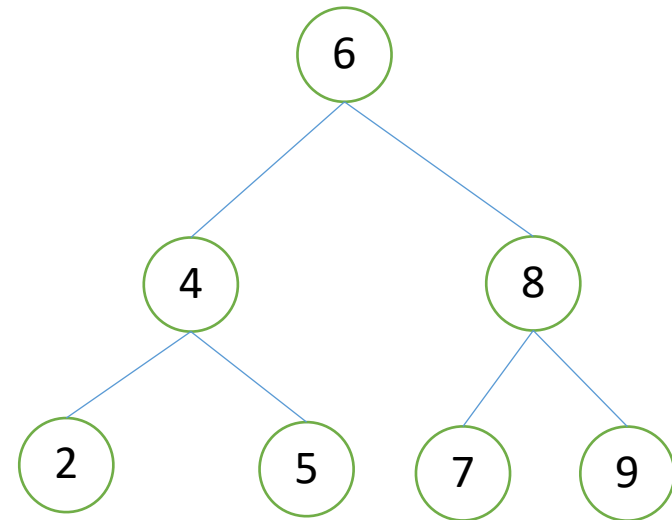
Percurso em profundidade

- Exemplo: considere a árvore binária de busca ao lado, liste a ordem de visitação dos nós para o percurso
 - in-ordem
 - pós-ordem
 - pré-ordem



Percurso em profundidade

- in-ordem
2, 4, 5, 6, 7, 8, 9
- pós-ordem
2, 5, 4, 7, 9, 8, 6
- pré-ordem
6, 4, 2, 5, 8, 7, 9



Método inorder – classe BST

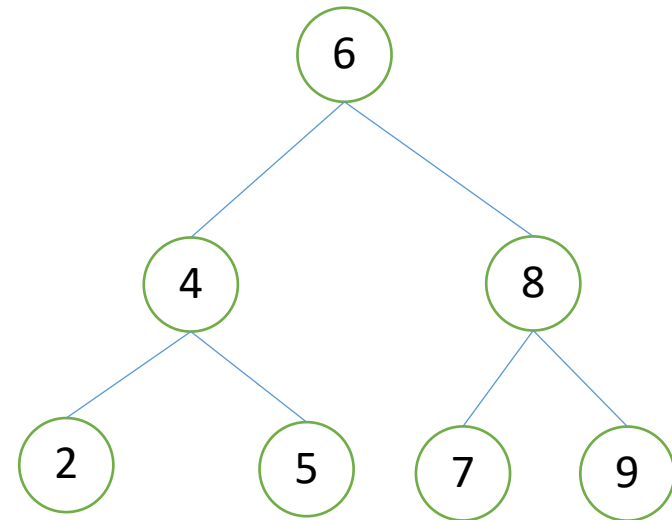
```
template<class T>
void BST<T>::inorder(BSTNode<T> *p)
{
    if (p != 0) {
        inorder(p->left);
        visit(p);
        inorder(p->right);
    }
}
```


Custo computacional – método inorder

- Veja que, após a chamada inicial, o método chama a si mesmo exatamente duas vezes para cada nó da árvore – uma vez para o filho à esquerda e uma vez para o filho à direita.
- Pode-se daí demonstrar que a complexidade do método é $O(n)$, onde n é o número de nós da árvore.
- O mesmo custo é válido para os métodos pós-ordem e pré-ordem.

Percurso em largura

- Exemplo: considere a árvore binária ao lado, o percurso em largura seria
6, 4, 8, 2, 5, 7, 9
- O percurso em largura pode ser implementado com a ajuda de uma **fila**

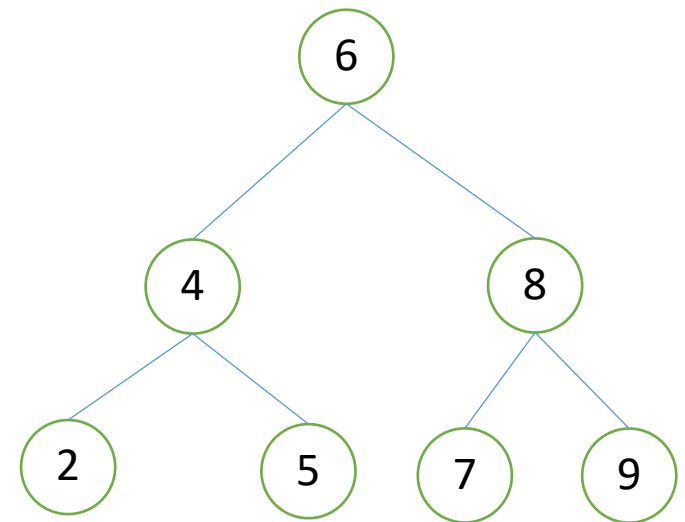


Percurso em largura

1. Inicie fila vazia
2. Ponha raiz no final da fila
3. Repita até esvaziar a fila:
 1. Tira primeiro da fila
 2. Visita nó
 3. Se tem filho à esquerda → põe no final da fila
 4. Se tem filho à direita → põe no final da fila
4. Encerra algoritmo

Percurso em largura

1. Põe 6 na fila - [6]
2. Tira o 6 - []
 1. Põe 4 na fila - [4]
 2. Põe 8 na fila - [4, 8]
3. Tira o 4 - [8]
 1. Põe 2 na fila - [8, 2]
 2. Põe 5 na fila - [8, 2, 5]
4. Tira o 8 - [2, 5]
 1. Põe 7 na fila - [2, 5, 7]
 2. Põe 9 na fila - [2, 5, 7, 9]
5. Tira o 2 - [5, 7, 9]
6. Tira o 5 - [7, 9]
7. Tira o 7 - [9]
8. Tira o 9 - []



Método breadthFirst – classe BST

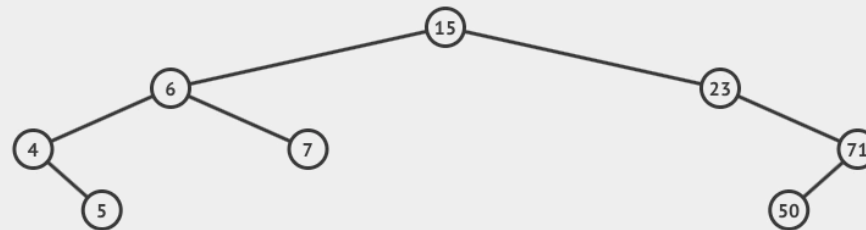
```
template<class T>
void BST<T>::breadthFirst() {
    std::queue< BSTNode<T>* > fila;
    BSTNode<T> *p = root;
    if (p != 0) {
        fila.push(p);
        while (!fila.empty()) {
            p = fila.front(); //pega no do inicio da fila
            fila.pop(); //remove no do inicio da fila
            visit(p);
            if (p->left != 0)
                fila.push(p->left); //insere no final da fila
            if (p->right != 0)
                fila.push(p->right); //insere no final da fila
        }
    }
}
```

Busca e Inserção na BST

Busca na BST

- Considere um dado valor k a ser procurado entre as chaves de uma árvore de busca binária
- Graças à propriedade BST, se comparamos k com a chave do nó atual, é fácil saber qual o próximo nó a visitar
 - Se $k < \text{chave} \rightarrow$ procure no filho à esquerda
 - Se $k > \text{chave} \rightarrow$ procure no filho à direita

Busca na BST



Create
Search
Insert
Remove
Pred-/Succ-essor
Inorder Traversal

Find Min

Find Max

7

Go

In-order traversal of the whole BST is complete.

```
if this is null
  return
inOrder(left)
visit this, then inOrder(right)
```


Método search – classe BST

```
template<class T>
T* BST<T>::search(BSTNode<T>* p, const T& el) const {
    while (p != 0)
        if (el == p->el)
            return &p->el;
        else
            if (el < p->el)
                p = p->left;
            else
                p = p->right;
    return 0;
}
```

Repare na ausência de chaves no comando while... Pois nesse caso não é necessário!

C++: declaração const

1. No caso de declarações de variáveis ou de argumentos de funções, o valor daquela variável não pode mudar dentro do escopo da declaração.

Exemplo:

```
const double pi = 3.14;
```

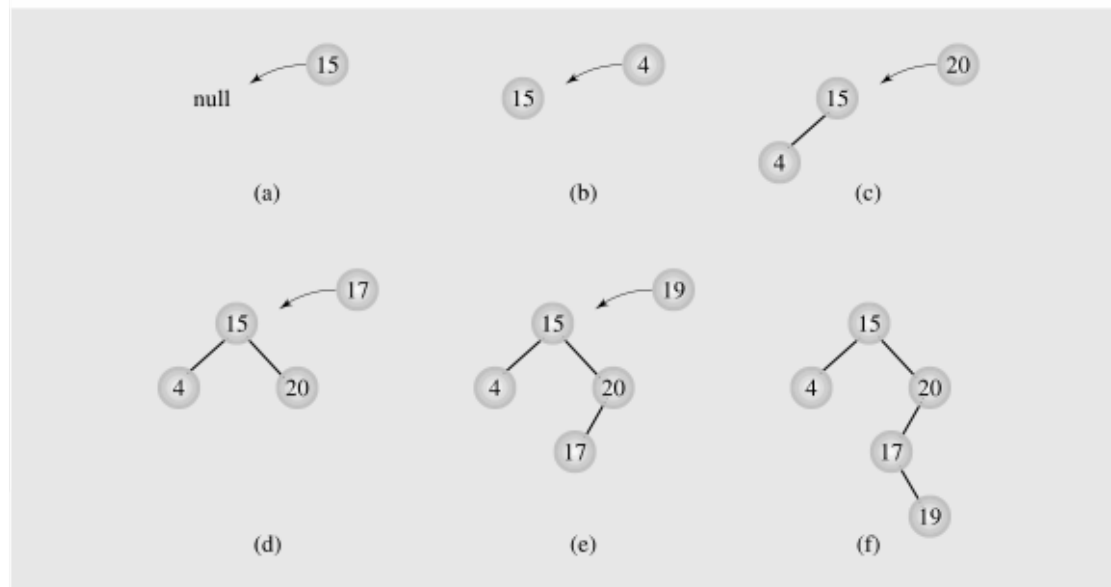
2. Quando é colocada após a declaração de uma função, significa que a função não pode alterar nenhum membro daquela classe
 - Logo este tipo de uso só é permitido para funções-membro (métodos) de uma classe

Custo computacional - busca

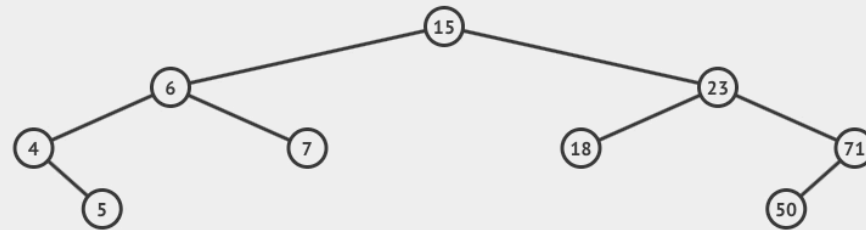
- Repare que os nós visitados ao longo da busca, seja pela recursão ou pelo método iterativo, formam um caminho da raiz até uma folha da árvore de nível i .
- No pior caso, esse caminho será até uma folha de máximo nível, isto é, no valor equivalente à altura da árvore.
- Portanto a busca por um nó na árvore tem complexidade $O(h)$, onde h é a sua altura

Inserção na BST

- A inserção de um nó com chave k é similar ao procedimento de busca
- Ao chegar a uma folha, ali se encontra o ponto de inserção do novo elemento



Inserção na BST



Create
Search
Insert
Remove
Pred-/Succ-essor
Inorder Traversal

21

Insert 18 has been completed.

```
if insertion point is found
  create new vertex
if value to be inserted < this key
  go left
else go right
```

Exercício guiado

Inspirado na busca, escreva o código C++ para o método de inserção

```
template<class T>
void BST<T>::insert(const T& el) {

    //codigo...

}
```

Resposta - método insert – classe BST

```
template<class T>
void BST<T>::insert(const T& el) {
    BSTNode<T> *p = root, *prev = 0;
    while (p != 0) { // find a place for inserting new node;
        prev = p;
        if (el < p->el)
            p = p->left;
        else p = p->right;
    }
    if (root == 0) // tree is empty;
        root = new BSTNode<T>(el);
    else if (el < prev->el)
        prev->left = new BSTNode<T>(el);
    else prev->right = new BSTNode<T>(el);
}
```

Análogo à operação de busca, a inserção tem custo $O(h)$

Exercício de Assimilação de Conceitos

Estude o arquivo `genBST1.h`, complete os trechos faltantes de código e escreva um programa que use a biblioteca e explore as suas operações.

- O que significa a declaração `protected`?

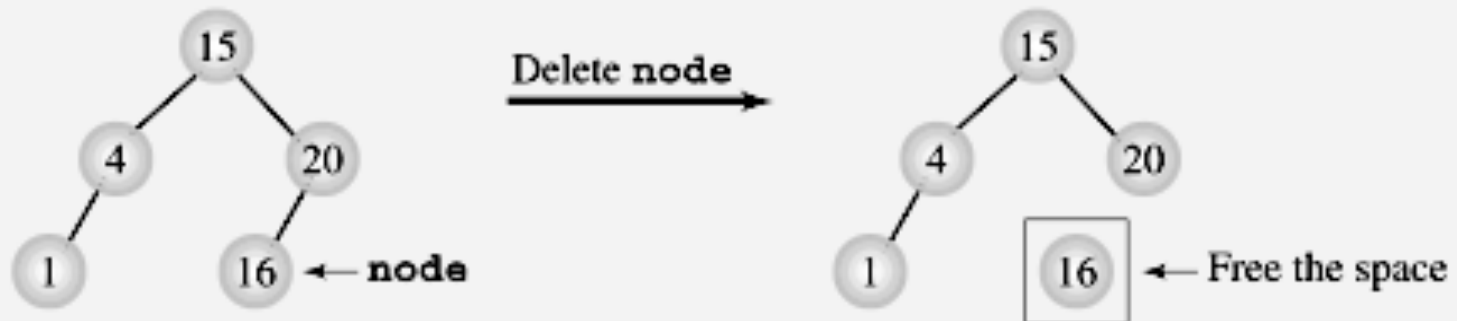
Remoção na BST

Remoção

- A operação de remoção envolve eliminar um dado nó e, adicionalmente, "consertar" a árvore caso a propriedade BST após a eliminação tenha sido violada
- Para isso, vamos quebrar o problema em três casos

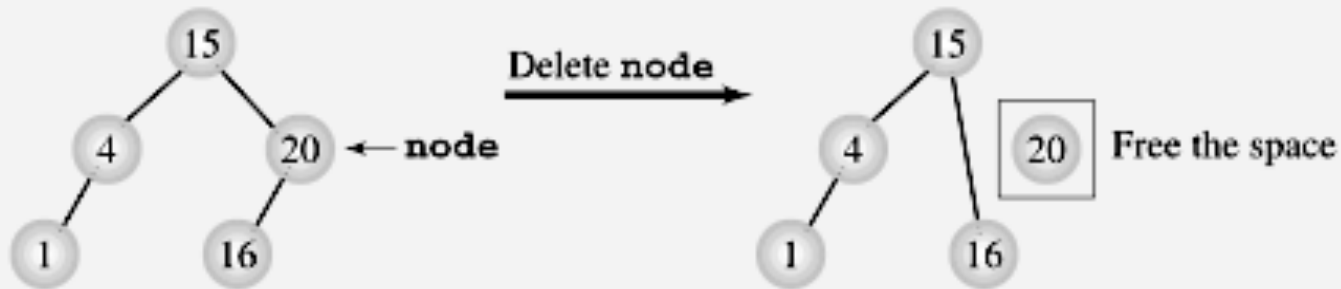
Caso 1: nó a remover é uma folha

- Basta configurar o apontador do nó pai para vazio e o nó é desalocado (operação `delete` em C++)



Caso 2: nó a remover tem um filho

- Basta configurar o apontador do nó pai diretamente para o filho do nó a remover, daí em seguida o nó é desalocado

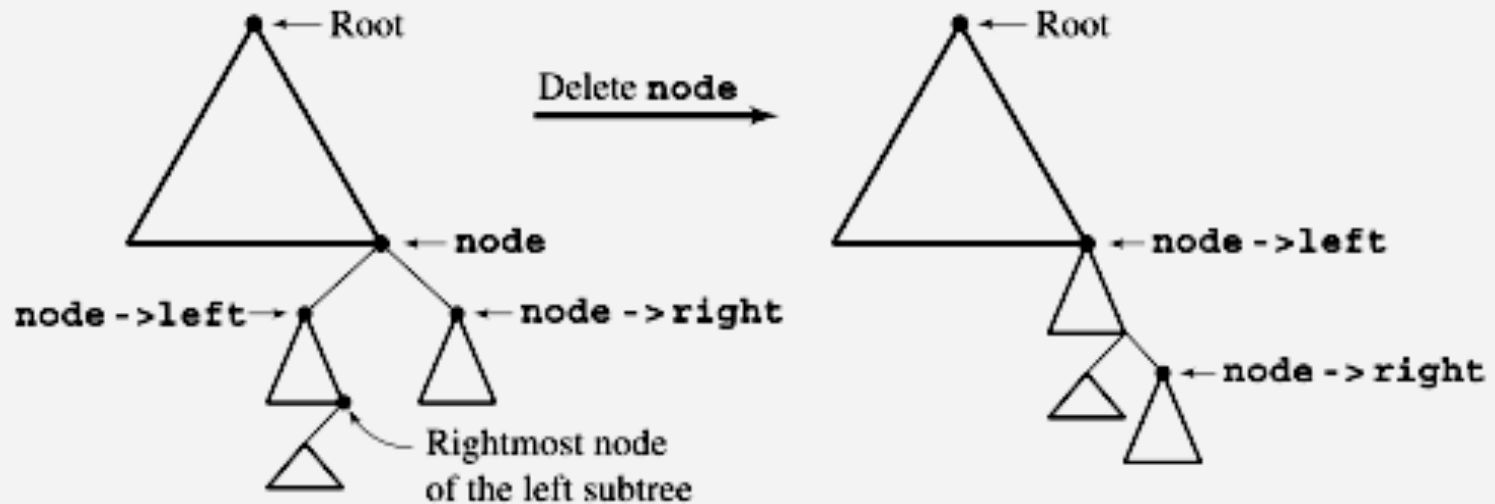


Caso 3: o nó a remover tem dois filhos

- Este é o caso mais complicado e há duas soluções possíveis, de forma a manter a propriedade BST
 - Remoção por fusão
 - Remoção por cópia

Remoção por fusão

- A idéia é "fundir" as sub-árvores do nó a ser removido e colocar no lugar dele.



- Repare que a sub-árvore direita agora é filha do nó mais à direita da sub-árvore esquerda

Remoção por fusão

- Para um dado nó x , pela propriedade BST sabemos que todos os elementos à esquerda são **menores** que sua chave $x.key$.
- Logo, se procurarmos o máximo desta sub-árvore, i.e. o nó mais à direita possível, encontraremos **o maior elemento que é menor** que $x.key$.
- Isto equivale a determinar o **antecessor** de $x.key$ quando os nós da árvore são colocados em ordem crescente
- Como o antecessor de $x.key$ é menor que qualquer nó da sub-árvore direita, colocar esta como filha mantém válida a propriedade BST

Método deleteByMerging - classe BST

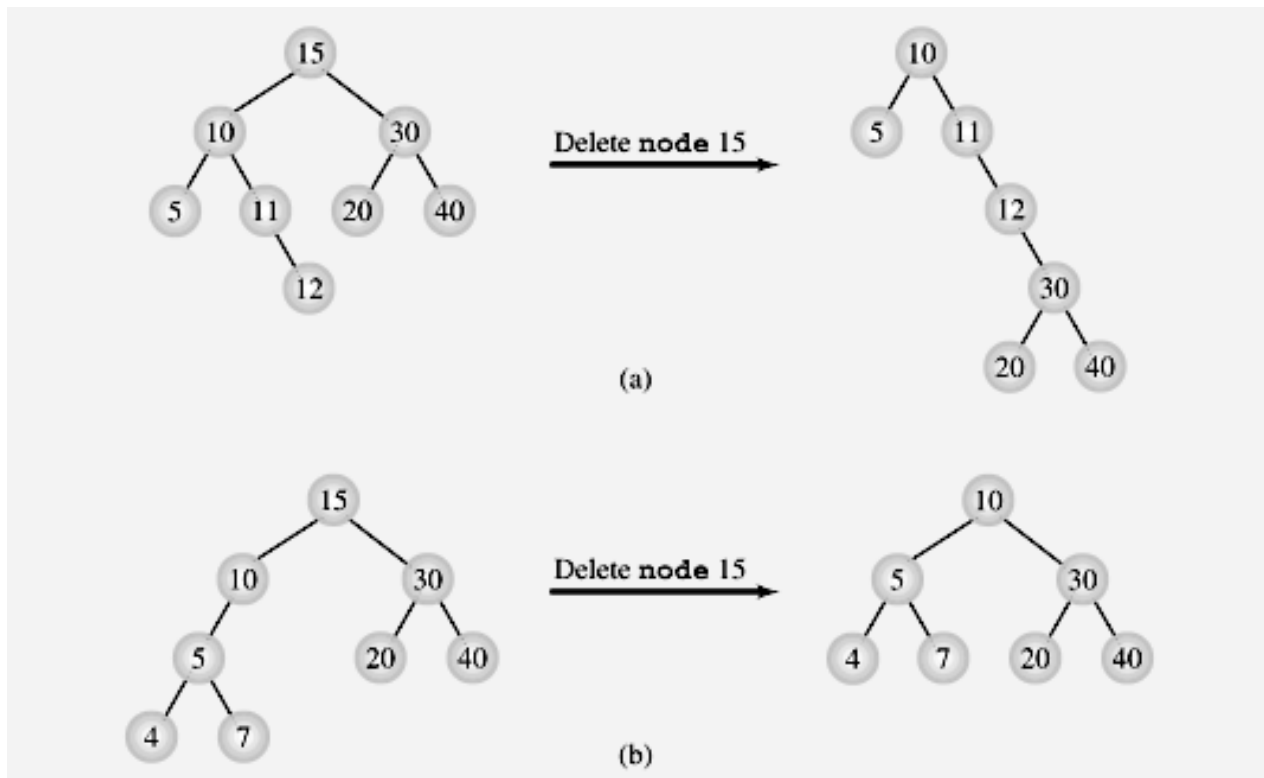
```
template<class T>
void BST<T>::deleteByMerging(BSTNode<T>*& node) {
    BSTNode<T> *tmp = node;
    if (node != 0) {
        if (node->right == 0)           // node has no right child: its left
            node = node->left;           // child (if any) is attached to its parent;
        else if (node->left == 0)       // node has no left child: its right
            node = node->right;          // child is attached to its parent;
        else {                          // be ready for merging subtrees;
            tmp = node->left;             // 1. move left
            while (tmp->right != 0) // 2. and then right as far as possible;
                tmp = tmp->right;
            tmp->right =                  // 3. establish the link between the
                node->right;              // the rightmost node of the left subtree
                                         // and the right subtree;
            tmp = node;                  // 4.
            node = node->left;            // 5. left child is attached to its parent
        }
        delete tmp;                     // 6. remove node
    }
}
```


Método findAndDeleteByMerging - classe BST

```
template<class T>
void BST<T>::findAndDeleteByMerging(const T& el) {
    BSTNode<T> *node = root, *prev = 0;
    while (node != 0) {
        if (node->el == el)
            break;
        prev = node;
        if (el < node->el)
            node = node->left;
        else node = node->right;
    }
    if (node != 0 && node->el == el)
        if (node == root)
            deleteByMerging(root);
        else if (prev->left == node)
            deleteByMerging(prev->left);
        else deleteByMerging(prev->right);
    else if (root != 0)
        cout << "el " << el << " is not in the tree\n";
    else cout << "the tree is empty\n";
}
```

Remoção por fusão

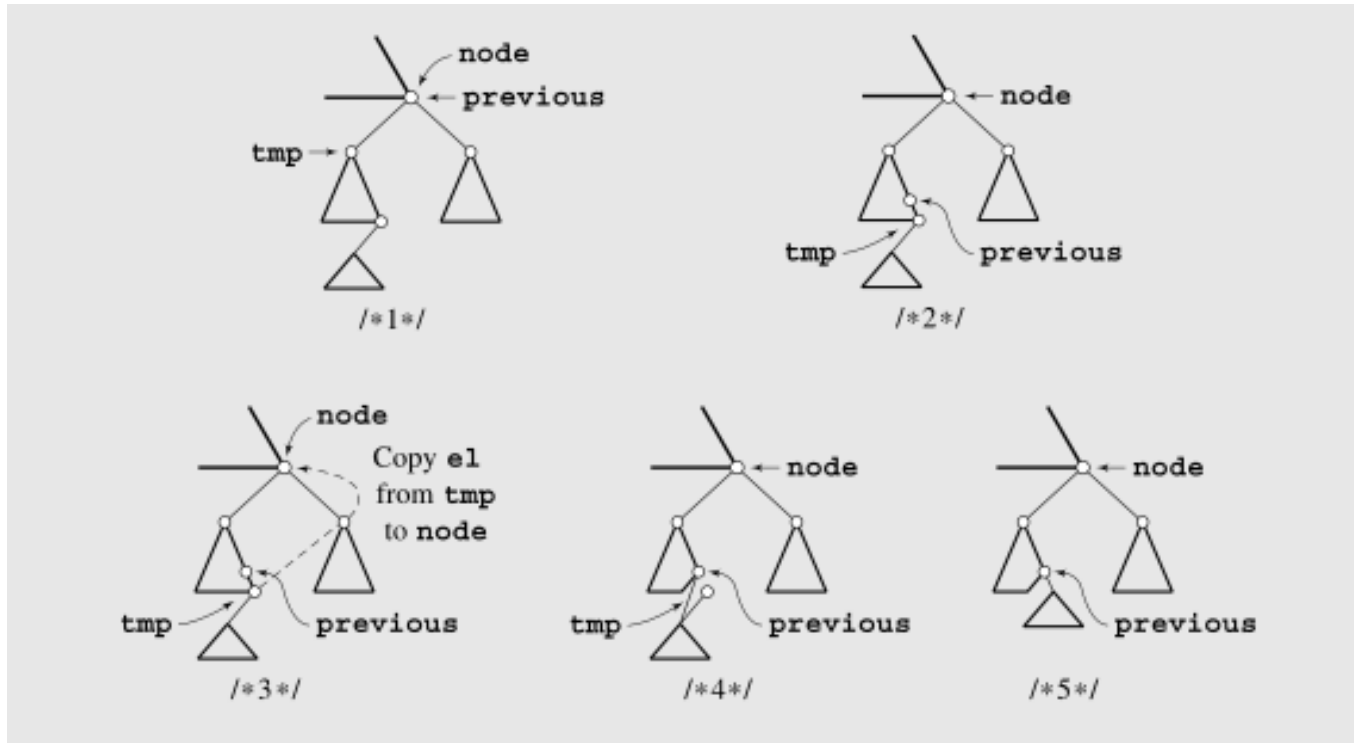
- Repare que o algoritmo de remoção por fusão pode levar ao aumento (ou redução) da altura da árvore, o que não é desejável



Remoção por cópia

- Se o nó a remover tem os dois filhos, pode-se “transformá-lo” em um dos 2 casos simples:
 1. É uma folha: configurar o apontador do nó pai para vazio e o nó é desalocado
 2. Tem um filho só: configurar o apontador do nó pai diretamente para o filho do nó a remover, daí em seguida o nó é desalocado
- Como fazer isso? Substitua os dados do nó a remover pelos dados do seu **antecessor**, **então realize o procedimento para apagar o nó antecessor**.

Remoção por cópia



- Como o antecessor é o nó mais à direita, veja que ou ele só terá um filho (à esquerda), ou será uma folha: logo sua remoção é simples

Método deleteByCopying – classe BST

```
template<class T>
void BST<T>::deleteByCopying(BSTNode<T>*& node) {
    BSTNode<T> *previous, *tmp = node;
    if (node->right == 0)                // node has no right child;
        node = node->left;
    else if (node->left == 0)            // node has no left child;
        node = node->right;
    else {
        tmp = node->left                // node has both children;
        previous = node;                // 1.
        while (tmp->right != 0) {        // 2.
            previous = tmp;
            tmp = tmp->right;
        }
        node->el = tmp->el;                // 3.
        if (previous == node)
            previous->left = tmp->left;
        else previous->right = tmp->left; // 4.
    }
    delete tmp;                          // 5.
}
```

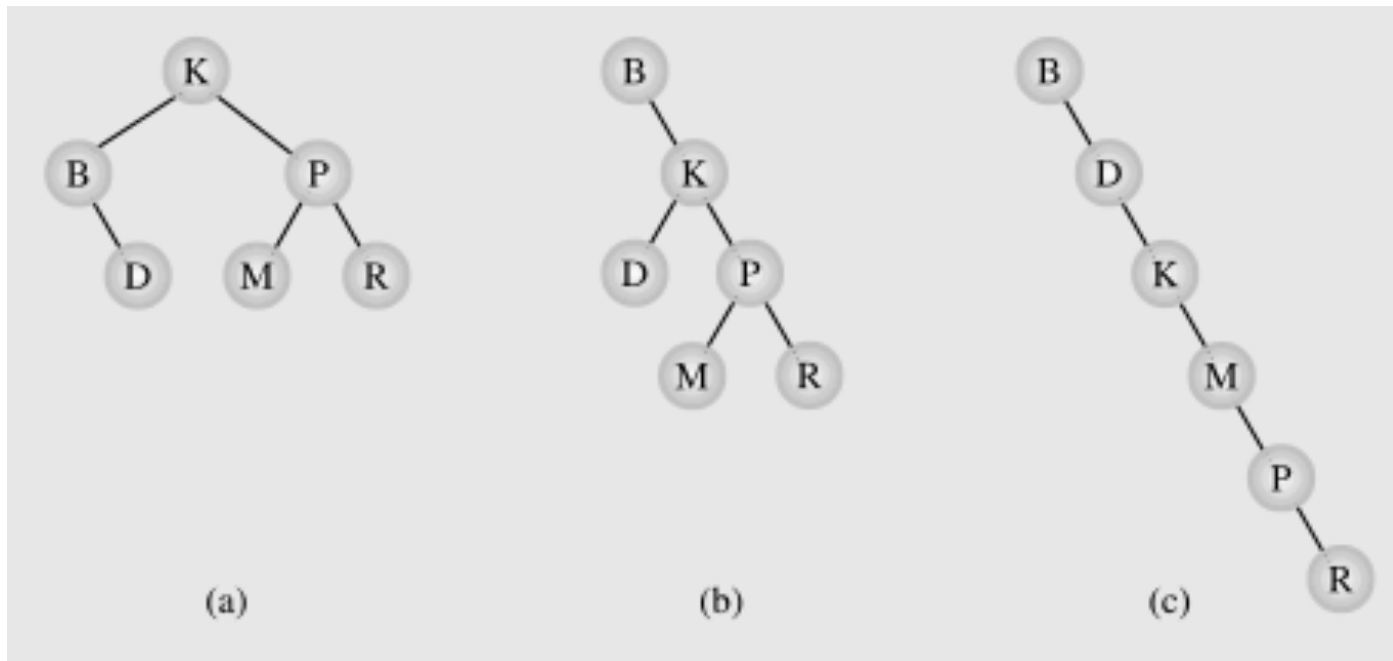
Analogamente pode-se definir o método findAndDeleteByCopying

Árvores balanceadas

Balanceamento

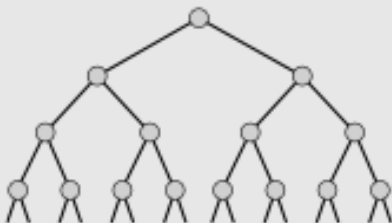
- O algoritmo de remoção por cópia não aumenta a altura da árvore, mas se aplicado muitas vezes em conjunto com a inserção, pode **desbalanceá-la**.
 - Subárvore esquerda pode ser reduzida, enquanto direita fica inalterada
 - Uma solução é remover alternando entre a cópia do antecessor e do **sucessor** (nó mais à esquerda da sub-árvore direita)
- **Uma árvore é balanceada** se a diferença de altura entre duas sub-árvores de um nó qualquer é zero ou um.
 - **Uma árvore é perfeitamente balanceada** se ela é balanceada e todas as folhas da árvore se encontram em um ou dois níveis no máximo.

Balanceamento



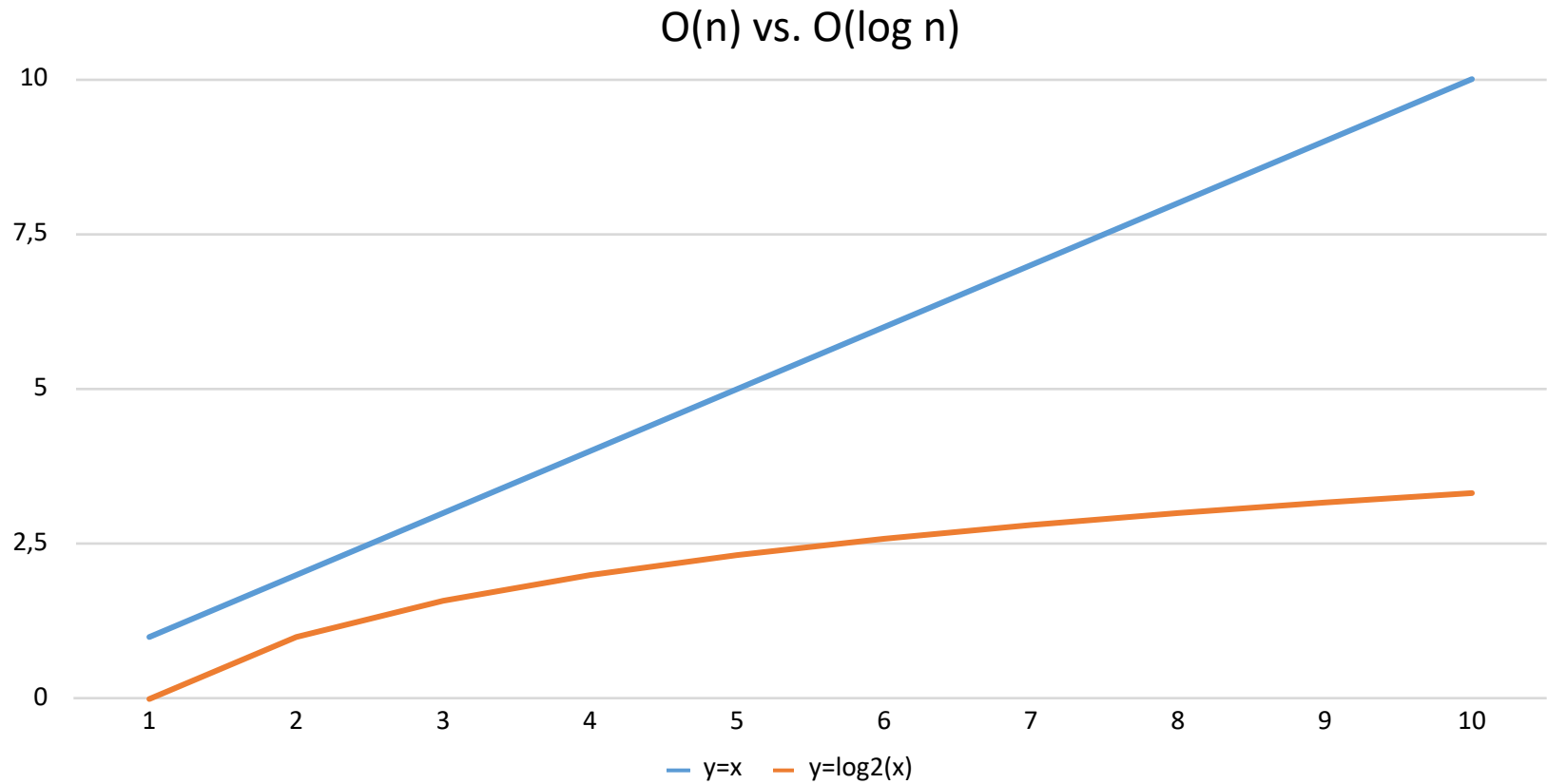
Só a árvore do caso (a) é balanceada

Balanceamento

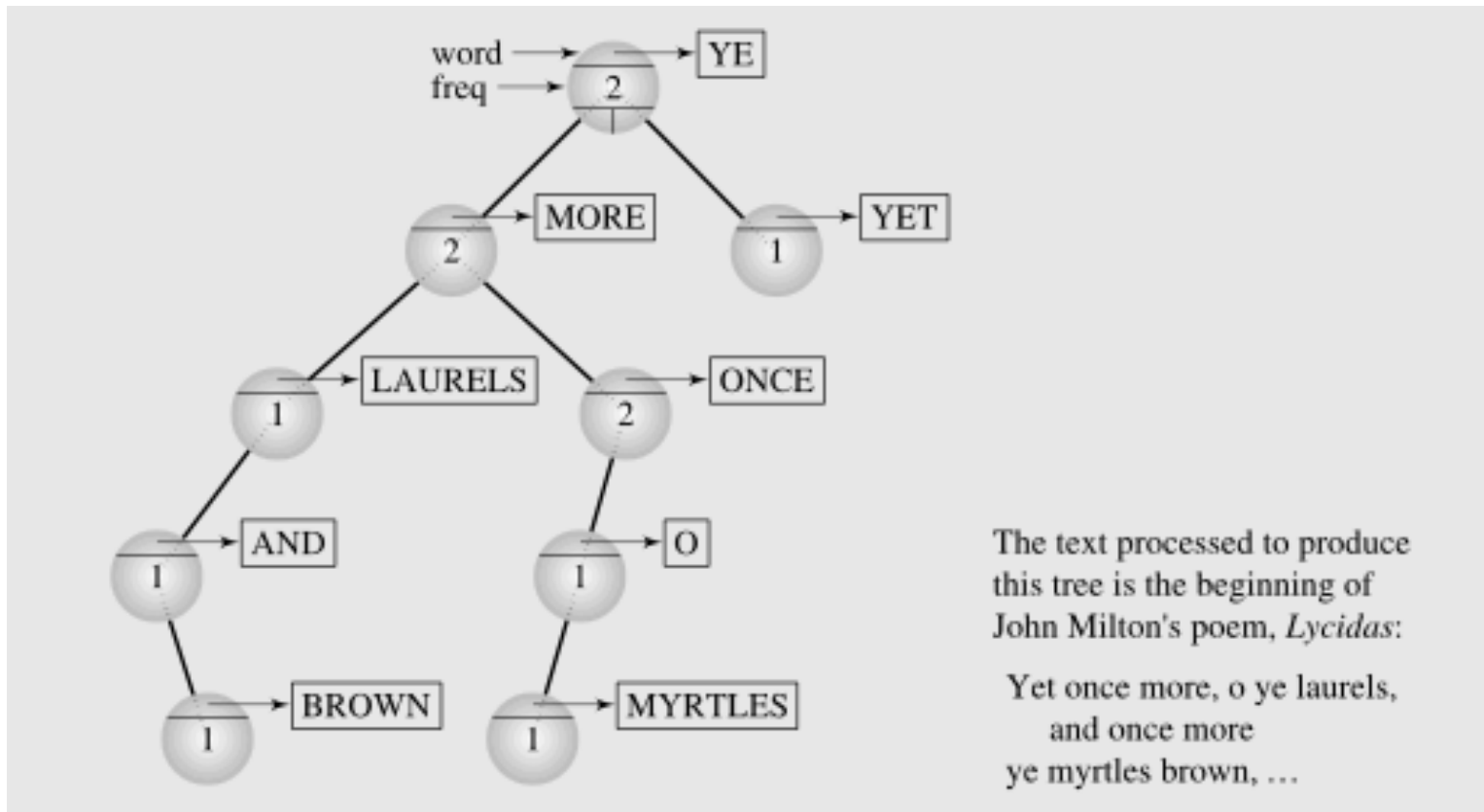
	Height	Nodes at One Level	Nodes at All Levels
	1	$2^0 = 1$	$1 = 2^1 - 1$
	2	$2^1 = 2$	$3 = 2^2 - 1$
	3	$2^2 = 4$	$7 = 2^3 - 1$
	4	$2^3 = 8$	$15 = 2^4 - 1$
	⋮		
	11	$2^{10} = 1,024$	$2,047 = 2^{11} - 1$
	⋮		
	14	$2^{13} = 8,192$	$16,383 = 2^{14} - 1$
	⋮		
	h	2^{h-1}	$n = 2^h - 1$
⋮			

- Veja que uma árvore perfeitamente balanceada tem altura $h \approx \log_2 n$
- Como a busca na BST tem custo $O(h)$, no caso de árvores perfeitamente balanceadas isto equivale a $O(\log_2 n)$, o que assintoticamente cresce **menos** que $O(n)$

Balanceamento



Um exemplo de aplicação: contagem de frequências de palavras



Conclusões

- Árvores de busca são estruturas interessantes para organização hierárquica de dados e pesquisa em tempo computacional médio menor que estruturas lineares, como as listas
- Existem outros tipos de árvores com diversas propriedades interessantes
 - Heaps
 - Árvores Rubro-Negras
 - Árvores AVL