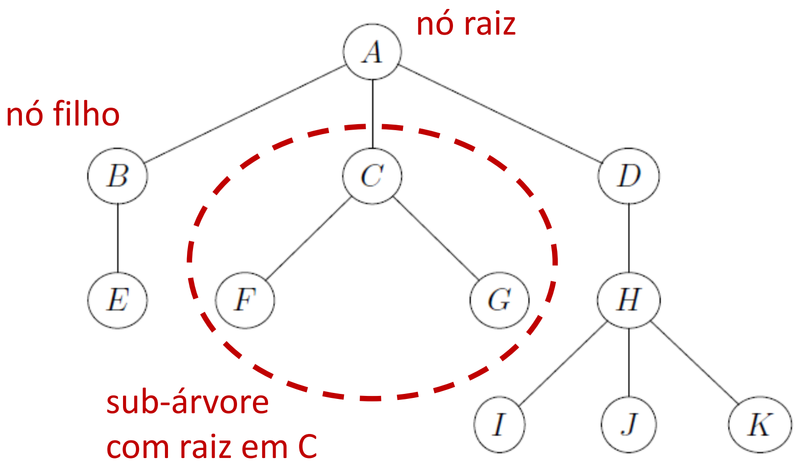
Aula 10:

Árvores

PCO001 – Algoritmos e Estruturas de Dados

Prof. João Paulo Reus Rodrigues Leite
joaopaulo@unifei.edu.br

Na computação, uma árvore é uma **estrutura de dados** **não-linear** amplamente utilizada, que representa uma **estrutura hierárquica** a partir de um conjunto de nós conectados. Veja o exemplo abaixo:

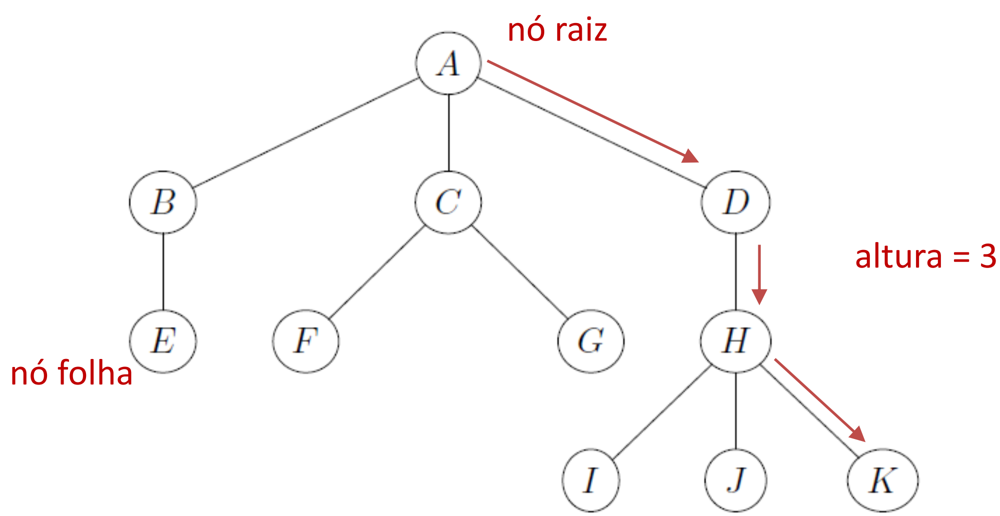


Em uma árvore, cada nó pode estar conectado a muitos nós filhos, mas somente a um pai (exceto pelo nó raiz, que não tem pai).

Cada nó da árvore pode ser considerado a **raiz de uma subárvore** que parte dele (facilita recursão). Além disso, **uma árvore** **não tem ciclos**, o que impede que um nó seja seu próprio ancestral na árvore.


A TREE WITH A CYCLE
YOUR MOVE, GRAPH THEORY

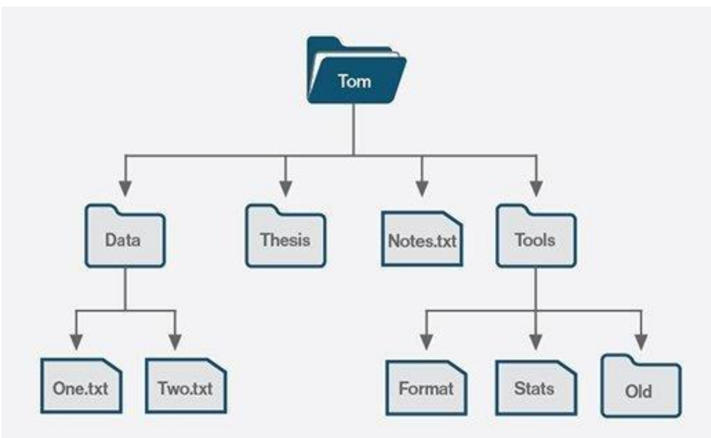
Um nó da árvore que não seja raiz de uma sub-árvore (grau 0) é chamado de **nó folha**. No exemplo abaixo, são nós folha E, F, G, I, J e K.

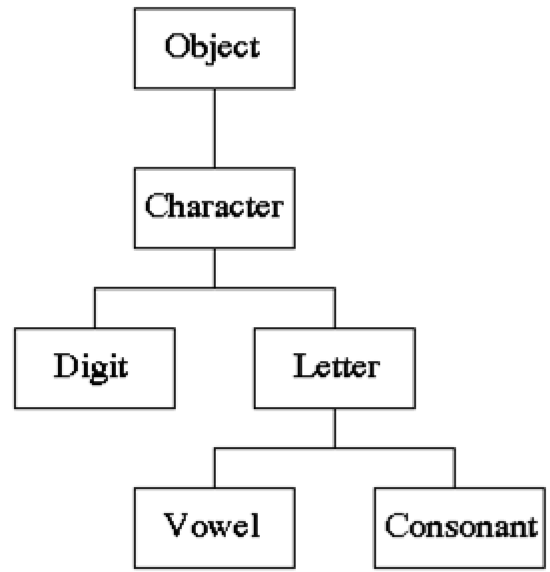


O **nível de profundidade** de um nó na árvore é definido da seguinte maneira: a raiz da árvore tem nível 0, enquanto o nível dos demais nós é igual ao número de arestas que o ligam à raiz, ou seja, é o comprimento do caminho que vai da raiz até este nó. A altura da árvore é igual ao máximo nível de seus nós.

No exemplo, **a árvore tem altura igual a 3**.

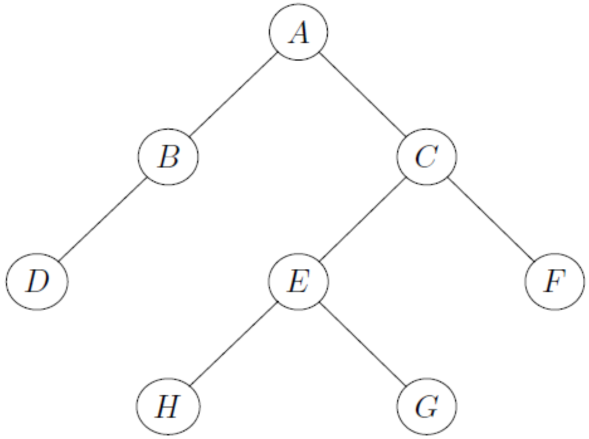
São muito utilizadas para representar estruturas com comportamentos hierárquicos, como **sistemas de arquivos** e relacionamentos de classes em **programas orientados a objetos**.





Árvores Binárias

São estruturas do tipo árvore em que **cada nó tem, no máximo, dois filhos** (grau ≤ 2). Veja o exemplo abaixo:



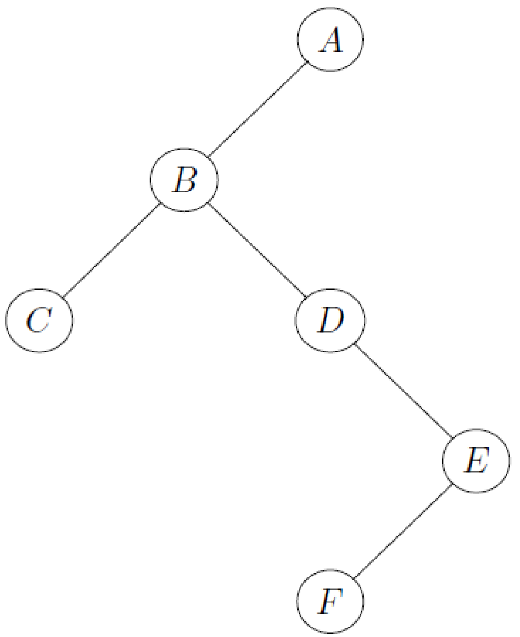

```
struct Node {
  char valor;
  Node* esq = nullptr;
  Node* dir = nullptr;
};

typedef Node* Arvore;
```

Neste tipo de estrutura, cada nó poderá ter uma sub-árvore da **esquerda** e/ou uma sub-árvore da **direita**.

Apesar de poder ser representada como um grafo, será mais útil a representação semelhante ao que fizemos com as listas encadeadas, ou seja, a partir de **estruturas do tipo Nó** que apontam para seus nós descendentes.

Árvores Binárias

Para construir a árvore binária, podemos utilizar um mecanismo chamado de "Notação Ponto". Nesta notação, os elementos são colocados em ordem pré-fixada e um ponto significa uma árvore vazia. Veja um exemplo para a árvore **A B C . . D . E F . . . .** :




```
// Cria Árvore a partir de Notação Ponto
void criaArvore(Arvore &root) {
  char val;
  cin >> val;
  if(val == '.')
    root = nullptr;
  else {
    root = new Node();
    root->valor = val;
    criaArvore(root->esq);
    criaArvore(root->dir);
  }
}
```

Caminhamento em Árvores Binárias

O termo “pré-ordem” foi mencionado no slide anterior. Mas o que isso significa?

Caminharem uma árvore significa percorrer todos os nós da árvore de forma sistemática, de modo que cada nó seja visitado uma única vez. O caminhamento pode ser em profundidade ou largura, como vimos na aula de grafos. O caminhamento em profundidade, no entanto, apresenta **três formar básicas**:

Pré-ordem ou pré-fixado;

Em ordem ou central;

Pós-ordem ou pós-fixado.

Caminhamento em Árvores Binárias

Pré-ordem ou pré-fixado:

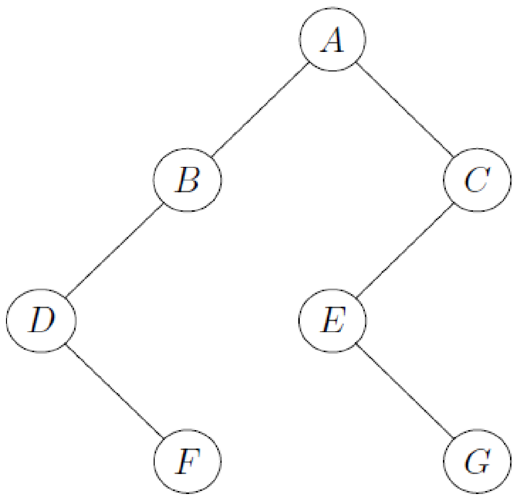
1. Processa o nó;
2. Percorre árvore da esquerda;
3. Percorre árvore da direita.

Em ordem ou central:

1. Percorre árvore da esquerda;
2. Processa o nó;
3. Percorre árvore da direita.

Pós-ordem ou pós-fixado:

1. Percorre árvore da esquerda;
2. Percorre árvore da direita;
3. Processa o nó.



Pré-Ordem: A B D F C E G
Em Ordem: D F B A E G C
Pós-Ordem: F D B G E C A

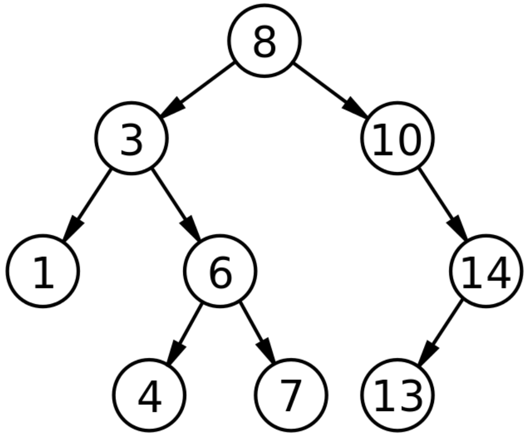
```cpp
void preOrdem(Arvore &root) {
  if(root != nullptr) {
    cout << root->valor; // processa o nó
    preOrdem(root->esq); // percorre árvore esquerda
    preOrdem(root->dir); // percorre árvore direita
  } else {
    cout << ".";
  }
}

void emOrdem(Arvore &root) {
  if(root != nullptr) {
    emOrdem(root->esq); // percorre árvore esquerda
    cout << root->valor; // processa o nó
    emOrdem(root->dir); // percorre árvore direita
  } else {
    cout << ".";
  }
}

void posOrdem(Arvore &root) {
  if(root != nullptr) {
    posOrdem(root->esq); // percorre árvore esquerda
    posOrdem(root->dir); // percorre árvore direita
    cout << root->valor; // processa o nó
  } else {
    cout << ".";
  }
}
```

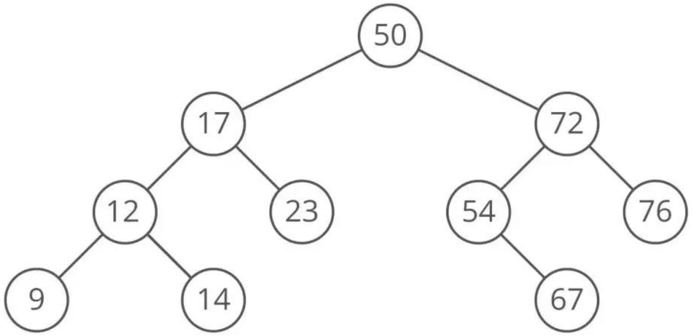
Repare que os códigos para caminhamento na árvore são **muito semelhantes**, mudando apenas a ordem em que as sub-árvores são percorridas com relação ao processamento do nó.

Árvores Binárias de Busca (BST)

A **árvore de busca binária** (*binary search tree*, ou **BST**) é uma árvore binária que obedece a seguinte regra:

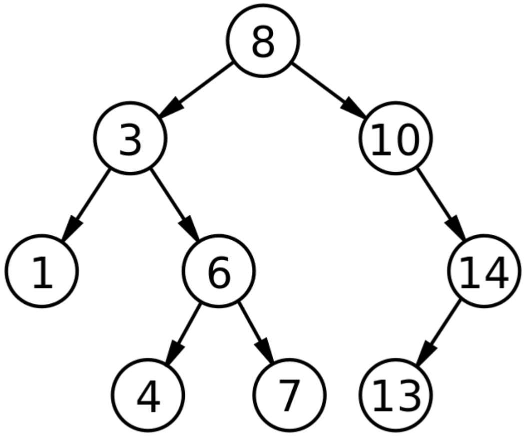
“Seja x um nó em uma BST. Se y é um nó da sub-árvore esquerda de x, então y.valor ≤ x.valor. Se y é um nó da subárvore direita de x, então y.valor ≥ x.valor.”





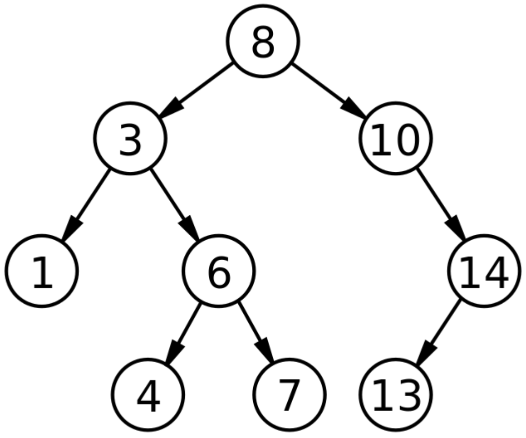
Árvores Binárias de Busca (BST)

A propriedade da BST nos permite imprimir todas as chaves em uma árvore em **sequência ordenada** por meio do algoritmo de percurso de árvore **em-ordem**. Lembrando que, se x for a raiz de uma sub-árvore de n nós, a chamada do método em-ordem demora **tempo Θ(n)**. Repare o exemplo:

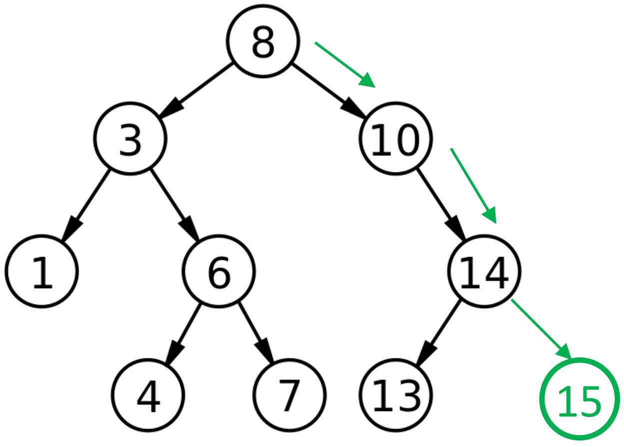


Percorrendo "**em ordem**":
1 – 3 – 4 – 6 – 7 – 8 – 10 – 13 – 14

```
void insert(Arvore &root, char dado) {
  Node* pai = nullptr;
  Node* atual = root;

  // Busca local onde será inserido
  while(atual != nullptr) {
    pai = atual;
    if(dado < atual->valor)
      atual = atual->esq;
    else
      atual = atual->dir;
  }

  // Cria e insere o novo nó
  atual = new Node();
  atual->valor = dado;

  if(pai == nullptr) // árvore estava vazia
    root = atual; // entra na raiz
  else if (dado < pai->valor)
    pai->esq = atual; // entra à esquerda
  else
    pai->dir = atual; // entra à direita
}
```



Imagine que eu queira inserir o valor 15.



Existe ainda a versão **recursiva**, muito **mais concisa** e tão eficiente quanto.

```
// Versão recursiva
void insert_rec(Arvore &root, char dado) {
  if(root == nullptr) { // a (sub)raiz ainda não existe.
    root = new Node();
    root->valor = dado; // cria novo nó
  } else {
    if(dado < root->valor)
      insert_rec(root->esq, dado);
    else
      insert_rec(root->dir, dado);
  }
}
```

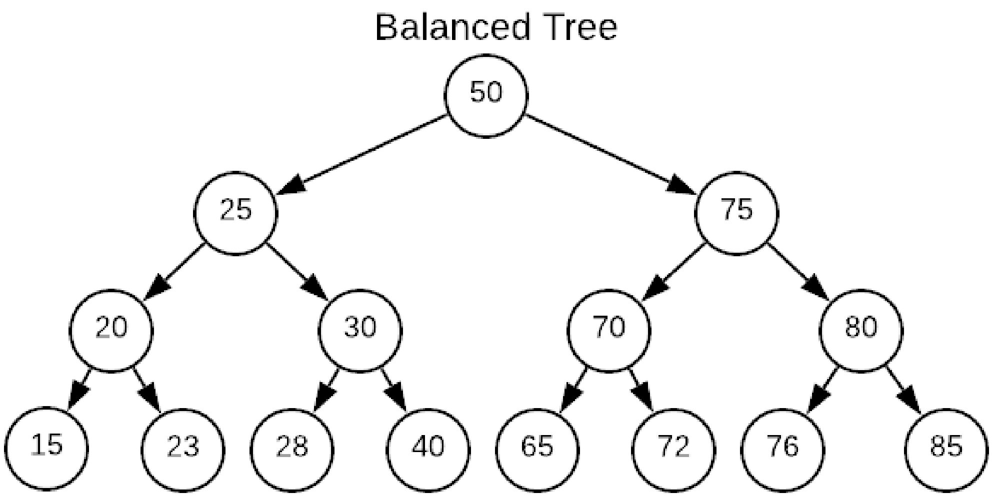
Em termos de complexidade, a inserção de um item na BST **depende da sua altura (h) no momento da inserção**. No pior caso, o item será inserido após o item mais distante da raiz, ou seja, aquele que está a h passos dela. Portanto, **sua complexidade é O(h)**.

Muito importante também é o método de **busca**, também recursivo.

```
bool search(Arvore &root, char key) {
  if(root == nullptr) return false; // Não encontrado
  if(root->valor == key) return true; // encontrado

  if(key < root->valor)
    return search(root->esq, key);
  else
    return search(root->dir, key);
}
```

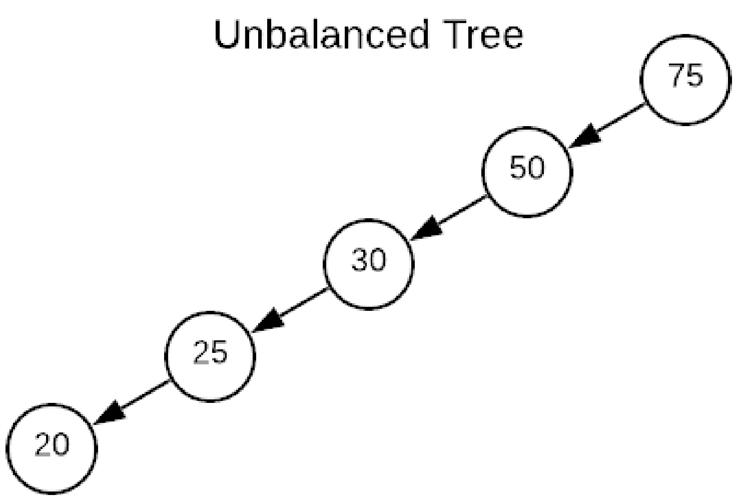
Em termos de complexidade, a busca de um item na BST **também depende da sua altura (h) no momento da busca**. No pior caso, o item buscado estará na parte mais distante da raiz, ou seja, a h passos dela. Portanto, **sua complexidade também é O(h)**.

Lembre-se de que em uma estrutura linear, a busca tem complexidade **O(n)**.
Na BST, temos certeza de que **h ≤ (n-1)**. Somente será igual a (n-1) no caso da BST estar completamente desbalanceada.



Em uma árvore binária totalmente balanceada e cheia como essa, temos 15 nós e altura 3. Altura (h) de aproximadamente **$\log_2(n)$**.

Dê uma pesquisada em árvores **AVL** e **Rubro-Negra**!



Em uma árvore binária totalmente desbalanceada como essa, temos apenas 5 nós e altura 4. Altura (h) aproximadamente igual a **n**.

Muito importante também é o método de **remoção**, que é um pouco mais complicado do que os demais. A **remoção** três casos:

- **Nó a ser removido não tem filhos:** Somente removemos o nó, modificando seu pai de modo a substituí-lo por **nullptr**.

- **Tem apenas um filho (uma subárvore):** Se tiver apenas um filho, elevamos esse filho para que ocupe sua posição na árvore, modificando o pai do nó de modo a substituir o nó removido pelo seu filho.

- **Tem duas subárvores, direita e esquerda:** Nesse caso, encontramos entre os dois filhos do nó a ser removido aquele que deve substituí-lo. Ele deve estar na subárvore à direita e não pode ter filho à esquerda (para manter regra de BST).

OBS: O substituto do nó removido tem que ser aquele nó que mantém o percurso "em ordem" correto! Menor possível da árvore direita.

Caso 1: Nó folha

```
       50                             50
     /     \         delete(20)     /   \
   30       70                     30     70
  /  \     /  \      --------->      \   /  \
20    40  60    80                   40  60   80
```

Caso 2: Um filho

```
      50                             50
   /      \        delete(30)      /    \
30         70                     40      70
   \     /   \     --------->            /   \
    40  60    80                        60    80
```

Caso 3: Dois filhos

```
      50                             60
   /      \         delete(50)     /    \
 40        70                     40     70
         /   \      --------->             \
        60    80                            80
```

Em termos de complexidade, a remoção de um item na BST **também depende da sua altura (h) no momento da busca**. No pior caso, o item (a ser removido ou o sucessor) estará na parte mais distante da raiz, ou seja, a h passos dela. Portanto, **sua complexidade também é O(h)**.

```cpp
bool remove(Arvore &root, char key) {
  Node* aux = root;
  Node* ant = nullptr;

  // encontra o nó a er removido. Se não encontrar, retorne false.
  while(aux != nullptr && aux->valor != key) {
    ant = aux;
    if(key < aux->valor)
      aux = aux->esq;
    else
      aux = aux->dir;
  }
  if(aux == nullptr) return false; // não encontrado
```

```cpp
  if(aux->esq == nullptr) { // nó removido não tem filho à esquerda
    if(ant == nullptr) // é a raiz da árvore
      root = aux->dir;
    else
      if (ant->esq == aux)
        ant->esq = aux->dir;
      else
        ant->dir = aux->dir;
    delete aux; // remove nó
  } else if (aux->dir == nullptr) { // nó removido não tem filho à direita
    if(ant == nullptr) // é a raiz da árvore
      root = aux->esq;
    else
      if (ant->esq == aux)
        ant->esq = aux->esq;
      else
        ant->dir = aux->esq;
    delete aux; // remove nó
  } else { // nó removido tem dois filhos
    Node* sucessor = aux->dir;
    while(sucessor->esq != nullptr) // procura o sucessor
      sucessor = sucessor->esq;
    aux->valor = sucessor->valor; // troca apenas o dado
    return remove(aux->dir, sucessor->valor); // remove o sucessor
  }

  return true;
}
```

1. **Encontra** nó a ser removido.
2. Se ele **não tiver filhos à esquerda**, é substituído pela subárvore da direita (contempla caso em que não tem nenhum dos filhos).
3. Se **não tiver filhos à direita**, é substituído pela subárvore da esquerda.
4. Se **tiver os dois filhos**, buscamos o sucessor, trocamos seus dados e mandamos remover o sucessor.

```cpp
bool remove_rec(Arvore &root, char key) {
  if(root == nullptr) // árvore vazia
    return false;

  if(key < root->valor)
    return remove_rec(root->esq, key);
  else if (key > root->valor)
    return remove_rec(root->dir, key);

  // se key == root->valor
  Node* aux = root; // nó a ser removido
  if(root->esq == nullptr) { // sem filhos à esquerda
    root = root->dir;
    delete aux;
  } else if (root->dir == nullptr) {  // sem filhos à direita
    root = root->esq;
    delete aux;
  } else { // tem dos filhos
    aux = aux->dir;
    while(aux->esq != nullptr) // procura sucessor
      aux = aux->esq;
    root->valor = aux->valor; // troca valores
    return remove_rec(root->dir, root->valor);
  }
  return true;
}
```

Também para a remoção, podemos implementar uma versão **recursiva**, **mais concisa** e tão eficiente quanto. Também executa em **O(h)**.

Exercício **para Casa**

Utilizando a linguagem C++, implemente uma **árvore binária de busca** (BST) que contenha em seus nós apenas **valores inteiros**. Sua BST precisa ter as seguintes funções:

- **Inserção** de novo item (recursiva);
- **Busca** de um item a partir de valor passado pelo usuário (recursiva);
- **Remoção** de um item a partir de valor passado pelo usuário (recursiva);
- Uma função que retorne a **altura** atual da árvore;
- **Impressão** da árvore em formato gráfico.

Seu programa deve ter uma função *main* semelhante à da aula, em que eu consiga inserir os itens na árvore, depois busca-los e removê-los. A cada operação, imprima a altura atual da árvore e seu formato gráfico atual, semelhante ao da imagem abaixo:

```
      50
   /     \
 40       70
        /   \
      60     80
```