DESCRIÇÃO

Árvores como estrutura de dados, suas propriedades básicas e terminologia empregadas; Árvores binárias, árvores de pesquisa e suas aplicações.

PROPÓSITO

As árvores formam uma estrutura de dados muito importante. Compreender os conceitos e o funcionamento de árvores significa perceber como ocorre a manipulação de memória, a programação, a arquitetura de computadores e sistemas operacionais, por exemplo. As árvores têm aplicações diversas, estando presentes em sistemas simples de busca, bancos de dados, sistemas de arquivos e muitos outros. Assim, enriquece e capacita o profissional para lidar com os problemas inerentes à área de TI.

PREPARAÇÃO

Para melhor absorção do conhecimento, recomenda-se o uso de computador com compilador de linguagem C e uma IDE (Integrated Development Environment) instalados.

OBJETIVOS

MÓDULO 1

Descrever os principais conceitos, as propriedades básicas e os termos relativos à árvore

MÓDULO 2

Reconhecer as árvores binárias, suas particularidades e aplicações

MÓDULO 3

Reconhecer as principais árvores de pesquisa, suas particularidades e aplicações

INTRODUÇÃO

Árvore é uma das mais importantes estruturas de dados na computação. Seu uso é abrangente e comum, estando presentes na visualização da estrutura de diretórios de um sistema operacional, como indexadoras de bancos de dados, em interfaces gráficas e em muitos outras aplicações.

Uma árvore é uma estrutura que se caracteriza pela hierarquização das informações. Isto é, há uma relação de ancestralidade entre seus elementos que desempenha papel essencial. Além disso, uma árvore é uma estrutura que pode ser definida recursivamente, mantendo sempre válida a definição.

Apresentaremos os conceitos básicos e as principais propriedades de árvores. Uma vez estabelecida esta base, veremos uma importante aplicação de árvores chamada árvore binária de busca. Compreenderemos a sua relevância e características para, finalmente, estudar árvores de pesquisa.

Na última seção, abordaremos os principais tipos de árvores de pesquisa e os problemas enfrentados. Finalizaremos com algumas atividades para consolidar os conceitos e o entendimento. Assim, você disporá de mais uma ferramenta.

MÓDULO 1

⦿ **Descrever os principais conceitos, as propriedades básicas e os termos relativos à árvore**

ÁRVORES

Em computação, a forma como os dados são armazenados tem impacto para operações de inserção, remoção, atualização, seleção de mínimo/máximo, entre outras. Por isso, estruturas que tornem tais operações eficientes são de grande interesse para os profissionais da área. As árvores oferecem vantagens para a realização dessas operações, como veremos nas próximas seções.

📢 ATENÇÃO

Estudaremos nesta seção os principais conceitos envolvendo árvores, as propriedades básicas e a terminologia que você precisa dominar para entender o assunto. Nosso objetivo será desenvolver gradualmente o entendimento da estrutura de dado, permitindo a conexão entre os conceitos e definições mostrados. Com isso, estabeleceremos as bases para discussões mais elaboradas que serão conduzidas nos demais módulos.

Esta seção oferecerá o ferramental para as discussões mais aprofundadas das próximas. Para facilitar o seu estudo, os conceitos, propriedades e terminologias apresentam-se destacados em negrito, facilitando a sua localização no texto.

CONCEITOS BÁSICOS E TERMINOLOGIA

O primeiro conceito que apresentaremos é justamente o que define o que é uma árvore.

Uma **árvore** T é um conjunto finito de elementos que obedecem a:

Se $T = \{\emptyset\}$, então a árvore é dita vazia.

> Caso contrário, existe pelo menos um elemento chamado de “nó raiz”. Os demais elementos ou são um único conjunto vazio ou são divididos em $m \geq 1$ conjuntos disjuntos não vazios, chamados subárvores do nó raiz, cada qual, por sua vez, uma árvore.

Os elementos que formam uma árvore são chamados **nós** ou **vértices** da árvore, sendo que um nó é especial: O **nó raiz**, que é o único nó existente na posição hierárquica mais alta. Uma árvore vazia não possui nó raiz.

Todos os demais nós da árvore, se existirem, pertencem a alguma subárvore do nó raiz. Usaremos, assim, a notação $T(k)$ para designar uma subárvore T, cujo nó raiz é “k”. Já um conjunto de árvores, a seu turno, é chamado de floresta.

Entendamos melhor a definição de **árvore**.

Primeiro, observe que é uma definição recursiva. Cada subárvore é, ela própria, uma árvore. Logo, cada subárvore atende às condições da definição e possui, ela mesma, o seu nó raiz.

Observe agora as duas condições impostas. Na primeira, o conjunto de elementos que formam a árvore é vazio. Ou seja, trata-se de uma árvore que não possui nós. Por isso, essa árvore é chamada de vazia.

A segunda situação diz respeito ao caso em que há nós na árvore. Neste caso, você deve prestar bastante atenção ao seguinte: Os elementos são divididos em $m \geq 1$ conjuntos disjuntos não vazios.

💬 COMENTÁRIO

Em outras palavras, dizer que esses conjuntos são disjuntos significa dizer que eles não possuem elementos em comum, ou, colocando ainda de outra forma, nenhum nó faz parte ao mesmo tempo de duas subárvores distintas. Finalmente, isto quer dizer que uma árvore não possui ciclos.

Veja a Figura 1 e a Figura 2. A primeira exibe uma árvore. Já a segunda figura mostra um grafo, outro tipo de estrutura de dados. Repare que no segundo caso, o nó D pertence simultaneamente a dois conjuntos: A subárvore esquerda de A e a subárvore direita, formando um ciclo (ABDC).

A razão pela qual esse ponto foi sucessivamente apresentado de forma cada vez menos formal, ao invés de darmos logo a definição de ausência de ciclos, é que você precisa compreender as nuances envolvidas na definição de árvore. Entender o que são as subárvores e como estas se relacionam será essencial para compreender as propriedades de árvores que veremos nas próximas sessões.

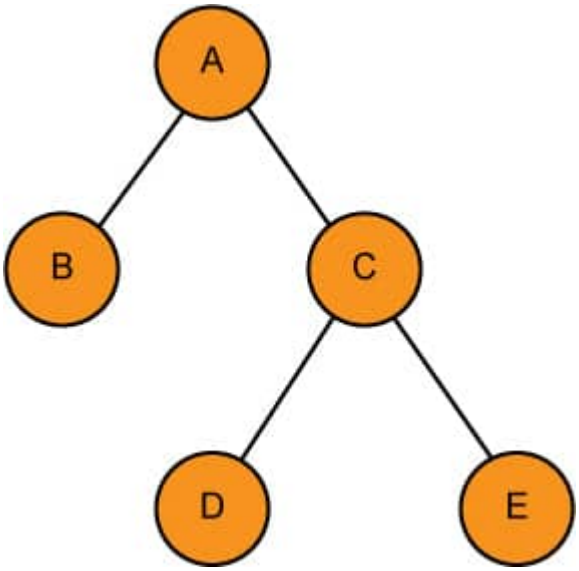


Figura 1: Árvore.

Fonte: EnsineMe.

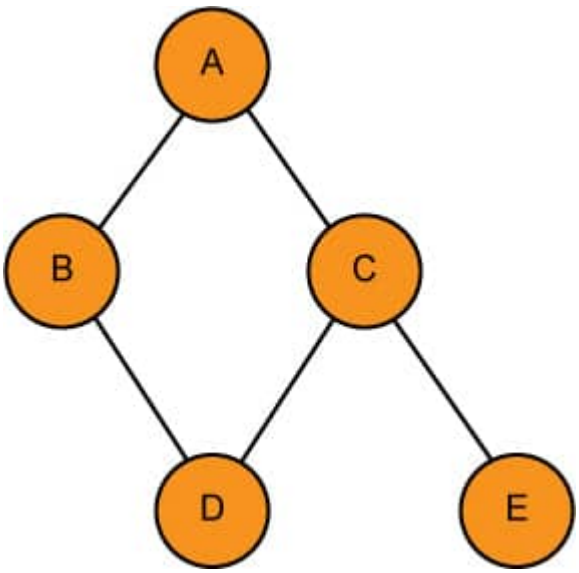
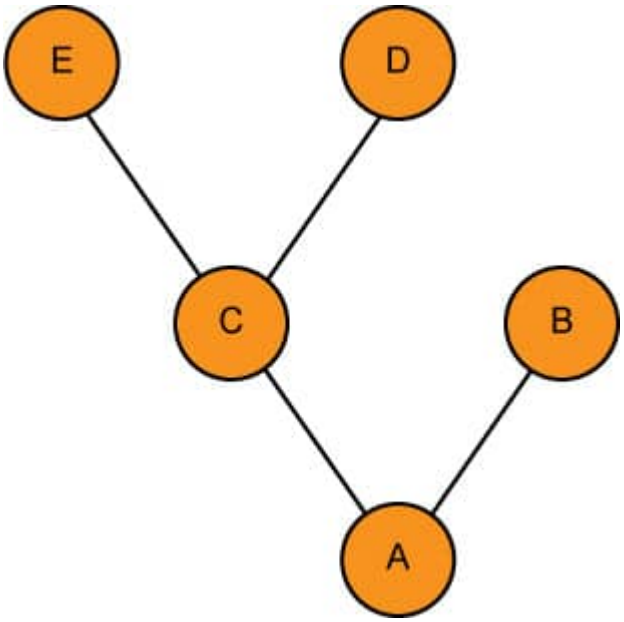


Figura 2: Grafo.

Fonte: EnsineMe.

A Figura 1 mostra uma forma clássica de se representar graficamente as árvores. Esta representação, chamada de **hierárquica**, mostra a árvore com a raiz no topo. Cada nó está ligado, mostrando a relação hierárquica entre eles. A representação hierárquica pode apresentar, ainda, o nó raiz na base, conforme se vê na Figura 3.



📷 Figura 3: Representação hierárquica invertida.

Fonte: EnsineMe.

Há outras formas de se representar graficamente uma árvore. Você pode ver algumas delas na Figura 4, que mostra uma árvore representada por **diagramas de inclusão** e na Figura 5, que exibe a representação por **diagramas de barras**.

📢 ATENÇÃO

Note que, em todas, é possível se observar a hierarquia dos nós e a sua ordem, mas a Figura 1 mostra estas relações com mais clareza, razão pela qual é mais popular na área de Computação.

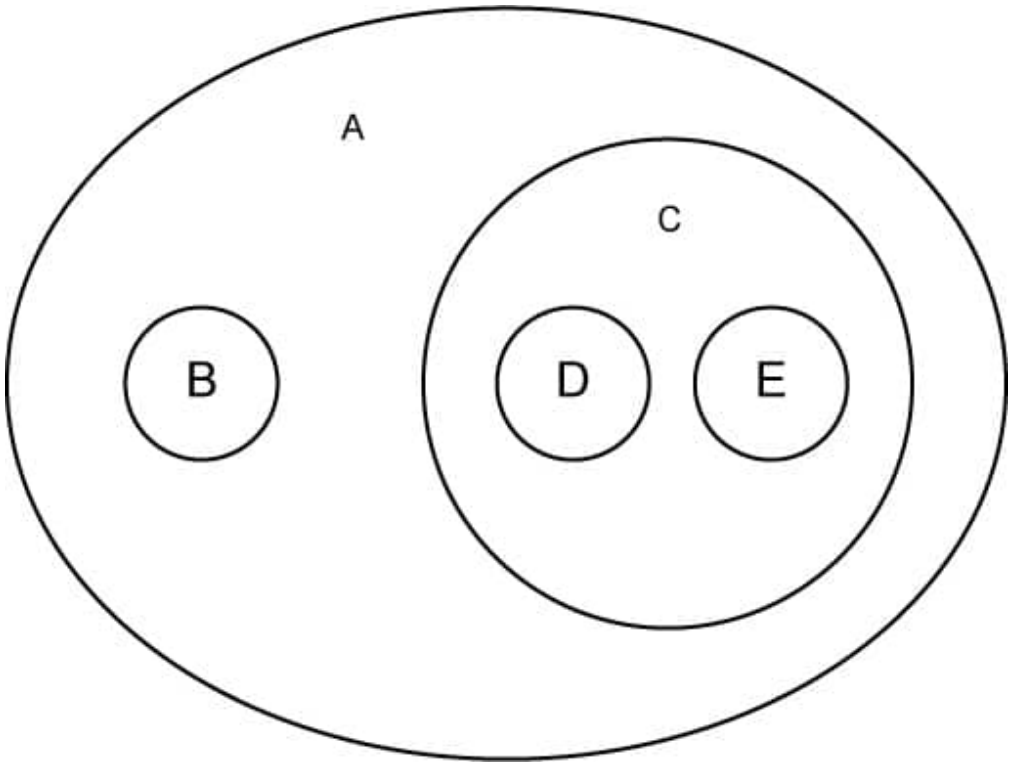


Figura 4: Representação por diagramas de inclusão.

Fonte: EnsineMe.

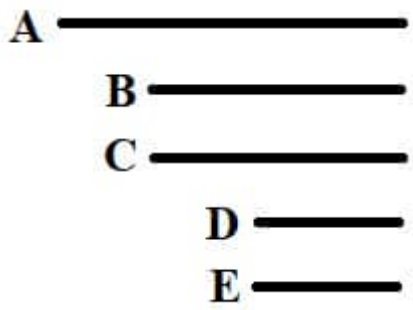


Figura 5: Representação por diagramas de barras.

Fonte: EnsineMe.

Há, ainda, a **forma parentizada**, que usa parênteses aninhados para representar a hierarquia dos nós. Por exemplo, a árvore da Figura 1 é representada por parênteses aninhados como (A (B) (C (D) (E))). É fácil ver a semelhança entre esta notação e uma expressão aritmética. Na verdade, uma expressão aritmética completamente parentizada pode ser representada por uma árvore, colocando-se os operandos nos nós e seus respectivos operadores nas subárvores. Assim, a expressão $(a * (b + c/d))$ é representada, sem ambiguidade, pela árvore da Figura 6.

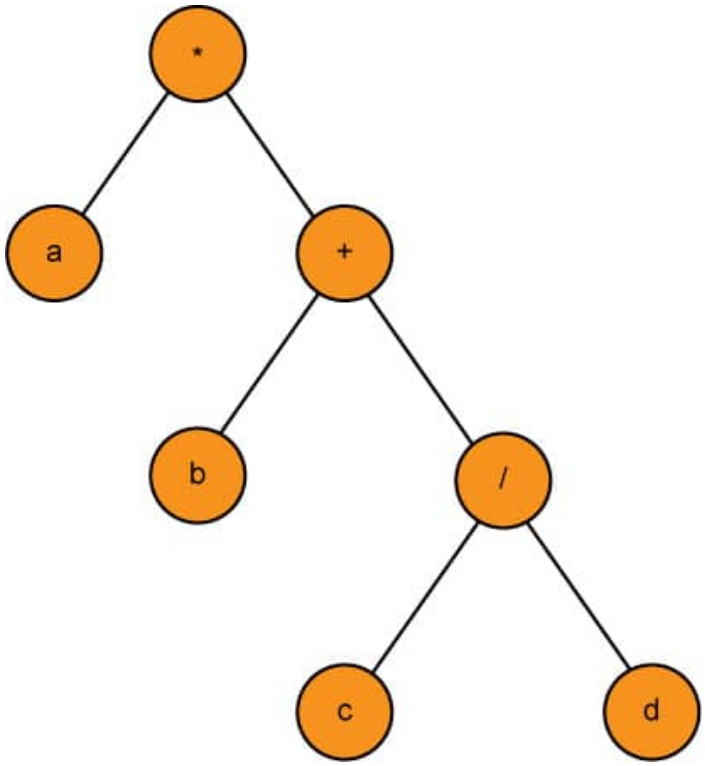


Figura 6: Expressão aritmética representada em árvore.

Fonte: EnsineMe.

ATENÇÃO

A respeito da relação hierárquica entre os nós, chamamos de **filhos** de k a todos os nós raízes $n_1,\ n_2,\ \ldots,\ n_j$ das subárvores $T(k)$. Os nós $n_1,\ n_2,\ \ldots,\ n_j$ são **irmãos** e k é nó **pai** destes nós. Se um nó w é filho de n_1, então n_2 é **tio** de w (e w é sobrinho de $n_2,\ \ldots,\ n_j$) e k é **avô** de w (e w, por sua vez, é neto de k). Se p pertence à subárvore $T(k)$, p é **descendente** de k, o qual é **ancestral** de p. Se p for diferente de k, então w é **descendente próprio** de k e este é **ancestral próprio** de w (ou seja, um nó é descendente e ancestral de si mesmo, pela definição).

O **grau de saída** (ou simplesmente grau) **de um nó** é definido como o número de filhos deste nó. Um nó que não possui descendentes próprios é chamado de nó **folha**. Nós que não são folha são chamados de **nós interiores**. O **nó raiz** de uma árvore ele próprio um nó interior, pode agora ser definido como o único nó de uma árvore que não possui ancestrais próprios. Quando falamos em ancestrais e descendentes próprios para definir as folhas e a raiz, estamos evitando a confusão que poderia se originar pela possibilidade de um nó ser seu próprio ancestral/descendente. Um filho é um descendente próprio. Uma vez que definimos o grau de um nó, definiremos o **grau de uma árvore** como sendo o maior valor entre os graus de

seus nós. Uma árvore de grau 2 é chamada binária, de grau 3, ternária e uma árvore genérica de grau *m*, *m-ária*.

Voltando ao nosso exemplo da Árvore na Figura 1, o nó A é o nó raiz da árvore, enquanto o nó C é nó raiz da subárvore direita de A. A é pai de C, que é irmão de B e D é neto de A. Os nós B, D e E são nós folhas, enquanto A e C são nós interiores. Nesse exemplo, os nós A e C possuem grau 2 e B, D e E grau zero.

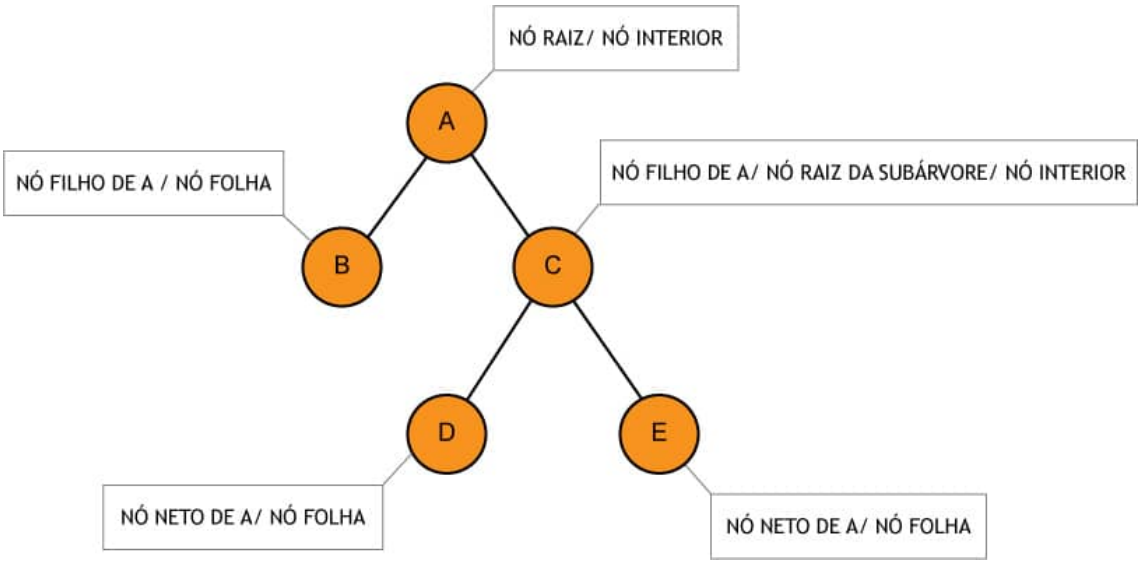


Figura 7: Relação hierárquica entre os nós.

Fonte: EnsineMe.

ATENÇÃO

Outra definição importante é a de **caminho** ou **percurso** em uma árvore. Dada uma sequência de nós distintos entre si $n_1, n_2, \ldots, n_i$, se entre dois nós consecutivos quaisquer sempre existir uma relação do tipo “é pai de” ou do tipo “é filho de”, então a sequência define um caminho ou percurso da árvore. Neste caso, n_1 alcança n_i e vice-versa.

Convém notar que a imposição da existência de relação de ancestralidade garante que o caminho seja sempre verticalizado. Ou seja, nós irmãos estão excluídos, não podendo formar entre si um caminho. Isto fica claro de se ver revisitando a Figura 1. ACE definem um caminho na árvore, mas ABC não (pois não há relação de ancestralidade entre B e C). Este exemplo também é útil para mostrar que um caminho de 3 vértices envolve 2 pares consecutivos de vértices (AC e CE).

Generalizando, um caminho de q vértices, tem $q - 1$ pares consecutivos. Assim, o **comprimento de um caminho** de q nós é dado pelo número de pares de relação deste caminho, ou seja, $q - 1$.

Mais uma definição importante é a de **nível** de um nó. Define-se o nível de um nó como sendo o número de nós no caminho da raiz até o nó considerado. O nível do nó raiz é, portanto, 1.

Lembre-se de que, pela definição, as subárvores de uma árvore sempre formam conjuntos disjuntos. Isto significa que sempre há um, e somente um, caminho de um nó até a raiz.

Já a **altura de um nó** é definida de maneira invertida. Ou seja, a altura de um nó t é o número de nós do maior caminho de t até um de seus nós descendentes. Neste caso, as folhas têm altura 1.

★ EXEMPLO

Observe que, nesta definição, admite-se a existência de mais de um caminho do nó considerado até as folhas. É essencial que você perceba que isto não contraria a inexistência de ciclos em uma árvore. Basta reparar que, ao se caminhar em direção às folhas, há uma multiplicidade de caminhos possíveis, conforme o grau de cada nó. Na Figura 1, por exemplo, há 3 caminhos levando do nó A até as folhas. Entretanto, se fixarmos a folha, temos novamente um, e somente um, caminho. A definição de altura do nó, contudo, deixa em aberto a escolha da outra extremidade do caminho, pois estamos interessados no caminho mais longo. Assim, olhando novamente o exemplo, a altura do nó A é 3, apesar de existir um caminho mais curto até uma folha (AB).

A partir da definição de nível de um nó, podemos definir a **altura de uma árvore**, representada por $h(T)$, como sendo o maior valor entre os níveis de seus nós. Analogamente, temos que a altura de uma subárvore de raiz v, representada por $h(v)$ é definida da mesma forma, considerando-se apenas os nós que pertencem à $U(v)$ (U sendo uma subárvore de T com raiz em v). Para o exemplo da Figura 1, a Tabela 1 mostra o nível e altura de cada nó. A árvore, a seu turno, tem altura 3 e grau 2 (é uma árvore binária).

Nó	Nível	Altura	Grau
A	1	**3**	**2**

B	2	1	0
C	2	2	2
D	3	1	0
E	3	1	0

Atenção! Para visualizaçãocompleta da tabela utilize a rolagem horizontal

Tabela 1: Nível, altura e grau dos nós da árvore da Figura 1.
Fonte: EnsineMe.

O último conceito que vamos apresentar nesta subseção é o de **árvores ordenadas**. Diz-se que uma árvore é ordenada quando os filhos de cada um de seus nós estão ordenados. Vamos convencionar que essa ordenação se dá da esquerda para a direita. Observando-se a árvore da Figura 1, é fácil ver que se trata de uma árvore ordenada, pois todos os seus nós estão ordenados alfabeticamente da esquerda para a direita. O conceito de árvores ordenadas implica que a posição relativa das subárvores é importante.

Assim, as árvores da Figura 8 e Figura 9 são diferentes entre si, se consideradas ordenadas.

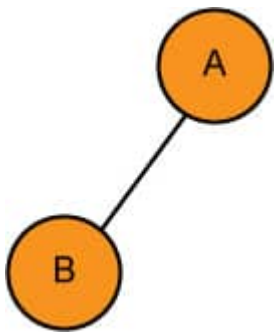


Figura 8: Árvore com a subárvore direita vazia.
Fonte: EnsineMe.

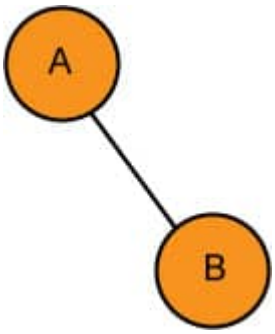


Figura 9: Árvore com a subárvore esquerda vazia.

Fonte: EnsineMe.

A seguir, veremos as propriedades básicas de árvores.

PROPRIEDADES BÁSICAS

A primeira propriedade que apresentaremos é que toda árvore com $n > 1$ nós possui no mínimo 1 e no máximo $n - 1$ folhas.

Em outras palavras, uma árvore com 5 nós $(n = 5)$ possuirá no mínimo 1 folha e no máximo 4. Isto não é difícil de se enxergar.

O número mínimo de nós será sempre obtido tendo-se todos os nós em um mesmo galho. Ou seja, se todos os n nós da árvore guardarem entre si uma relação de ancestralidade, quaisquer que sejam esses nós, então o que teremos serão todos os nós alinhados verticalmente da raiz até a folha. Como não há nós irmãos, temos uma única folha nessa árvore. Essa é, também, a árvore mais alta que pode ser obtida com n nós: uma árvore de altura n.

Em contrapartida, o número máximo de folhas será dado pela árvore de menor altura que se pode construir com n nós. Neste caso, um nó deve ser a raiz da árvore. Restam, assim, $n - 1$ nós que serão colocados como filhos do nó raiz. Assim, essa árvore, que tem altura igual a 2, possui $n - 1$ folhas ($n - 1$ nós irmãos).

DICA

Embora esta não seja uma prova formal desta propriedade, ela deve ter lhe convencido da validade da mesma. Sugerimos que você faça a prova formal como maneira de consolidar os

conhecimentos. Para isto, desenvolva a argumentação mostrada.

Observe, agora, a árvore da Figura 10. Note que essa árvore possui os nós ordenados de forma diferente da árvore da Figura 1. Mas você deve ter notado que há semelhanças estruturais e nos elementos que formam os nós de ambas as árvores. A bem da verdade, a Figura 10 mostra uma árvore que é uma imagem especular da árvore da Figura 1.

Veja que, mediante a reordenação das subárvores da Figura 10, podemos transformá-la na árvore da Figura 1. Aliás, para que as árvores se tornem coincidentes, esta reordenação é necessária e suficiente. A esta propriedade, chamamos de isomorfismo e dizemos que a árvore da Figura 1 é isomorfa da árvore da Figura 10 (e vice-versa).

De maneira formal, duas árvores exibem **isomorfismo** quando puderem se tornar coincidentes mediante a permutação na ordem das subárvores de seus nós. Obviamente, duas árvores ordenadas são isomorfas quando forem coincidentes segundo sua regra de ordenação.

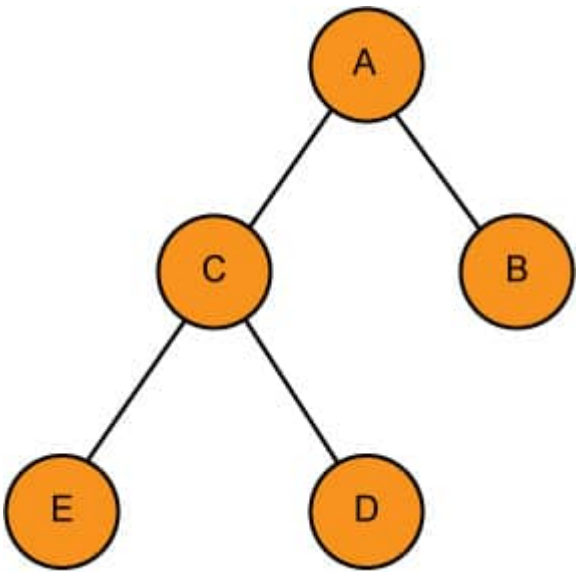
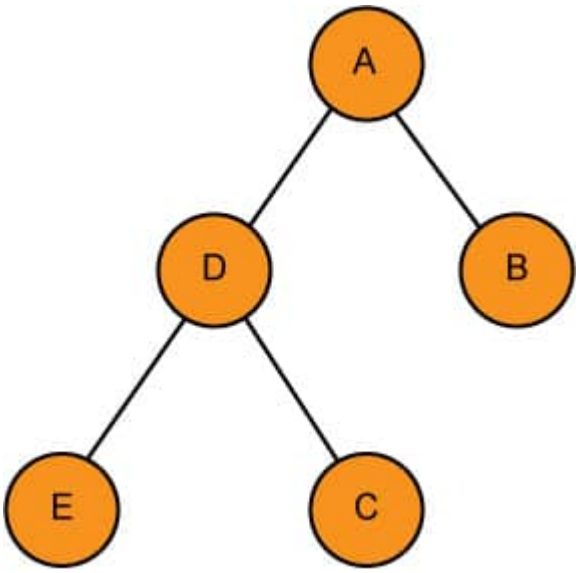


Figura 10: Árvore isomorfa.

Fonte: EnsineMe.



📷 Figura 11: Árvore não isomorfa da árvore da Figura 1 ou da Figura 10.
Fonte: EnsineMe.

Cabe esmiuçarmos um pouco mais esta propriedade.

Dizer que as árvores podem se tornar coincidentes pela permutação de suas subárvores, implica dizer que os rearranjos devem ocorrer na horizontal. Isto é, entre nós irmãos.

A árvore da Figura 11, por exemplo, não é isomorfa da árvore da Figura 1 e nem da árvore da Figura 10. Neste caso, a simples permutação na ordem das subárvores não as torna coincidentes. Para isto, faz-se necessário também permutar o nível dos nós C e D.

Além disto, para que duas árvores sejam isomorfas, não é necessário que todas as subárvores devam ser reordenadas.

💬 COMENTÁRIO

O exemplo da Figura 10 é um caso extremo. Mas poderia ocorrer de uma ou mais subárvores já se encontrarem na posição necessária.

REPRESENTAÇÃO COMPUTACIONAL DE ÁRVORES

Até agora, o que falamos sobre árvores situa-se no campo teórico. Mas uma questão fundamental é como uma árvore é implementada de fato num software. Uma das formas de fazê-lo é com o uso de alocação sequencial (array). É perfeitamente possível implementar uma árvore com vetores, embora esta não seja a maneira mais eficaz por diversos motivos:

> Em primeiro lugar, um vetor limita o espaço de memória a ser utilizado pela árvore, que não pode ultrapassar a quantidade definida em tempo de programação. Isto diminui as vantagens de se empregar uma estrutura de dados dinâmica como a árvore.
>
> Além disto, a unidimensionalidade do vetor torna menos intuitivo operar com a árvore implementada desta forma.

Tanto as operações quanto as relações hierárquicas entre os nós passam a depender fortemente de cálculos matemáticos, trazendo um grande esforço para a programação.

Por estas razões, não vamos nos aprofundar neste tipo pouco usual de implementação de árvores. O Código 1 apresenta, apenas como ilustração, uma função em linguagem C usada para posicionar uma chave à direita de um nó passado como parâmetro em uma árvore implementada por meio de um vetor. Veja, na linha 6, que a posição correta do vetor que representa o filho direito de um dado nó é fornecida pela fórmula $(pai * 2) + 2$, aonde "pai" é o índice vetorial do nó pai.

```
1  int insere_filho_direita ( char chave, int pai ) {
2      if ( arvore [ pai ] == NULL ) {
3          printf ( "Erro: noh pai inexistente!" )
4          return 0; //falha
5      } else
6          arvore [ ( pai * 2 ) + 2 ] = chave
7      return 1; //sucesso
8  }
```

Código 1: Função de inserção de nó filho na subárvore direita.

A outra maneira de se implementar árvores, e a mais comum, é através da alocação encadeada. Esta maneira tem vantagens óbvias com relação ao uso de alocação sequencial. Ela permite que a árvore cresça sob demanda, em tempo de execução. Se a árvore encolher, não há desperdício de memória, pois os nós removidos são desalocados. Além disto, a implementação se torna consideravelmente mais simples. É claro que, no uso de alocação encadeada, será necessário empregar tantos campos ponteiros quanto for o grau da árvore e mais um campo para se guardar a chave do nó.

📢 ATENÇÃO

Logo, o nó de uma árvore *m-ária*, com uma chave do tipo inteiro, ocupará um espaço de memória igual a $\left(m\ *\ tamanho\ ponteiro\ para\ o\ nó\ +\ tamanho\ da\ representação\ de\ inteiro\right)$. Ou, utilizando-se linguagem C: $m\ *\ sizeof\ (No*)\ +\ sizeof\ (int)$, sendo m o grau da árvore e “No” a estrutura que define o nó da árvore.

O Código 2 mostra a estrutura de um nó de uma árvore ternária e o código 3 a generalização desta estrutura para uma árvore *m-ária*.

```
1  struct No {
2      int chave;
3      No *filho_esq;
4      No *filho_meio;
5      No *filho_dir;}
```

Código 2: Estrutura do nó de uma árvore ternária.

```
1  struct No {
2      <tipo> chave;
3      No *filho_1;
4      No *filho_2;
5      [...]
6      No *filho_m;
7  }
```

Código 3: Estrutura genérica do nó de uma árvore *m-ária*.

Observe agora o Código 4, que mostra a função de inserção na subárvore direita de um nó. A linha 6 é onde a inserção propriamente dita ocorre. Perceba como a linha se tornou mais simples que a linha 6 do Código 1 e como agora não são necessários cálculos matemáticos para se determinar a posição do nó filho.

Para ser mais preciso, tudo que ocorre agora são acessos à memória. Na verdade, ocorre apenas uma escrita em memória no espaço reservado para “filho_dir”.

Na linha 6 do Código 1, eram feitas duas operações aritméticas e uma escrita em memória. Assim, podemos esperar uma performance melhor do Código 4 em comparação com o Código 1.

```
int insere_filho_direita ( No novo_no, No *pai ) {
    if ( pai == NULL ) {
        printf ( "Erro: noh pai inexistente!" )
        return 0; //falha
    } else
        pai -> filho_dir = novo_no;
    return 1; //sucesso
}
```

Código 4: Função de inserção de nó filho na subárvore direita (alocação encadeada).

É claro que o código apresentado aborda um caso simples, no qual o filho à direita do nó passado como argumento ou não existe ou está sendo sobrescrito. Mas esta condição vale tanto para o Código 4 quanto para o Código 1.

Você deve notar que, no caso do Código 1, sobrescrever o nó não exige ações adicionais, já que a relação hierárquica é estabelecida pela posição dos nós no vetor. No Código 4, contudo, sobrescrever um nó envolve considerar a existência de filhos deste nó.

Pelo menos duas soluções de modificação do Código 4 podem ser feitas para este caso:

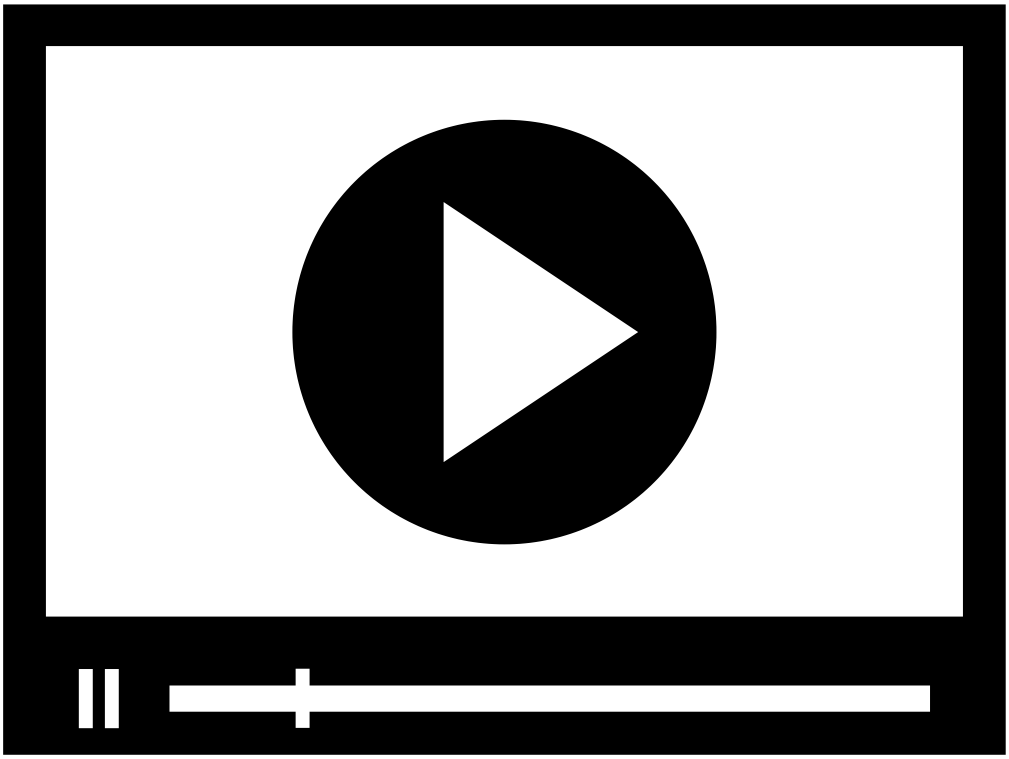
> A primeira consiste em se verificar se o nó a ser sobrescrito possui filhos e, neste caso, atualizar apenas a chave com o novo valor.

> Outra solução é fazer com que os nós filhos do nó a ser sobrescrito se tornem também filhos do novo nó e depois inseri-lo na árvore na posição correta.

Veja que sempre há vantagens e desvantagens. Portanto, a decisão de qual mecanismo de alocação a ser usado deve ser sempre uma decisão técnica tomada segundo o problema a ser resolvido.

📢 ATENÇÃO

Por isso, você deve ser capaz de analisar o problema e de compreender como empregar as ferramentas que tem à disposição. Nas seções seguintes, veremos em mais detalhes casos especiais de árvores, suas aplicações e propriedades.



Assista ao vídeo a seguir e reveja de forma prática o conceito de árvore.

VERIFICANDO O APRENDIZADO

MÓDULO 2

- **Reconhecer as árvores binárias, suas particularidades e aplicações**

INTRODUÇÃO

Neste módulo, vamos explorar um tipo especial de árvore chamada de árvore binária. Trata-se, dentre as árvores, de um dos tipos mais empregados, devido às suas características e propriedades. Já vimos os conceitos básicos, terminologias e propriedades básicas de árvores em geral. Agora, porém, veremos isto para o caso de árvores binárias.

Também apresentaremos algoritmos para as principais operações com árvores binárias. Mas a compreensão destes algoritmos demandará o entendimento dos conceitos e propriedades já vistos e os que serão apresentados agora. As notações que empregamos antes, continuarão válidas. Em caso de dúvidas, reveja a seção anterior.

CONCEITOS E PROPRIEDADES RELATIVOS ÀS ÁRVORES BINÁRIAS

Nós já apresentamos a definição geral de árvores. Entretanto, vamos formalizar o significado deste caso especial. Você notará a semelhança com a definição geral. Definimos uma **árvore binária**, chamada T, como sendo um conjunto finito de elementos denominados vértices ou nós e que obedecem a:

> Se $T = \{\emptyset\}$, então a árvore é dita vazia.
>
> Caso contrário, existe pelo menos um elemento chamado de “nó raiz”. Os demais elementos podem ser divididos em dois subconjuntos disjuntos, chamados subárvore esquerda e subárvore direita da raiz, cada qual, por sua vez, uma árvore binária.

Veja que a definição é praticamente a mesma para o caso geral, porém particularizada para o caso em que os nós possuem no máximo 2 filhos. Novamente, trata-se de uma definição recursiva, de tal maneira que, quando o conceito estipula que as subárvores são também árvores binárias, estas podem ser vazias e possuir ou não suas próprias subárvores esquerda ou direita.

Ou seja, um nó pode ter 0 (zero) filhos (subárvores esquerda e direita vazias), 1 filho (apenas uma subárvore vazia) ou 2 filhos (nenhuma subárvore vazia).

Para facilitar, vamos particularizar, também, a simbologia usada. Para representar o nó raiz da árvore T usaremos $r(T)$. Já, para representar a subárvore esquerda do nó raiz, usaremos $T_E(r(T))$ e $T_D(r(T))$ para indicar a subárvore direita. Lembre-se de que a definição é recursiva, assim, essa simbologia pode ser usada para os nós raízes das subárvores.

Quando a subárvore esquerda de um nó v não for vazia, a raiz dessa subárvore será chamada filho esquerdo de v. Analogamente, a raiz da subárvore direita é chamada de filho direito de v. Adaptaremos à notação geral, de maneira que $T_E(v)$ e $T_D(v)$ representarão, respectivamente, as subárvores esquerda e direita de v.

Uma diferença entre as árvores binárias e o caso geral é que a posição das subárvores identifica unicamente uma árvore binária.

Repare a Figura 12 (a) e a Figura 12 (b). Você notará que ambas são isomorfas e idênticas, mesmo se consideradas como árvores ordenadas. Entretanto, ambas as árvores são distintas se consideradas binárias. Esta diferença reside, justamente, nas subárvores do nó B. No caso da primeira árvore, a $T_E(B)$ é vazia e a $T_D(B)$ possui um nó filho (D). Já na segunda árvore, a situação se inverte.

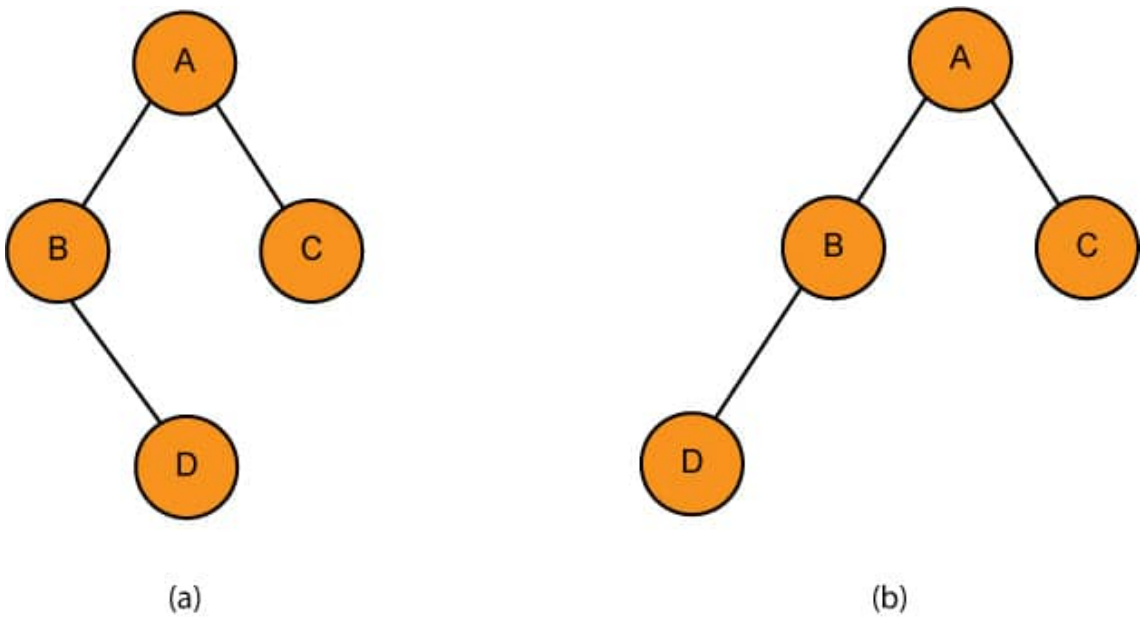


Figura 12: Árvores binárias.

Fonte: EnsineMe.

ATENÇÃO

Compreenda que a ordenação, conforme vimos, diz respeito ao ordenamento entre nós irmãos. É por isso que, mesmo se consideradas árvores ordenadas, ambas são idênticas (apesar de graficamente representadas de forma diferente). Para o caso geral, a posição de cada subárvore não importa, mas apenas a ordem entre elas.

Mas se considerarmos ambas as árvores como binárias, então, elas se tornam diferentes, pois, neste caso, a posição das subárvores passa a ser relevante. Em uma, o nó D está na subárvore esquerda e na outra, na direita.

Já vimos o que é uma árvore binária. Agora, veremos alguns tipos particulares de árvores binárias. Começaremos por definir o que é uma árvore estritamente binária.

SAIBA MAIS

Uma **árvore estritamente binária** é uma árvore em que cada nó possui zero ou dois filhos, ou seja, uma árvore que possua pelo menos um nó com apenas um filho é uma árvore binária, mas não é uma árvore estritamente binária. Perceba que nós que não possuem filhos, pelos conceitos que já vimos, são obrigatoriamente nós folha. Assim, uma árvore estritamente binária é uma árvore na qual todo nó interior possui dois filhos.

A Figura 13 mostra um exemplo de árvore estritamente binária:

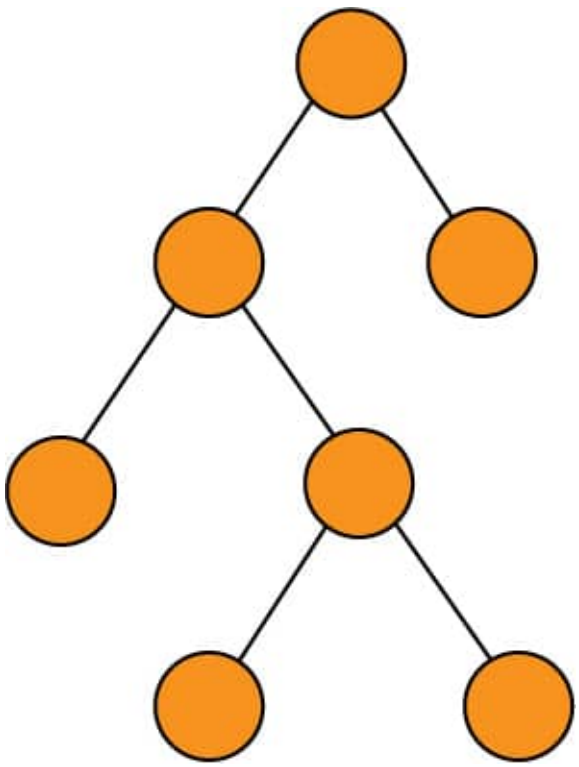


Figura 13: Árvore estritamente binária.

Fonte: EnsineMe.

Veja agora outros conceitos de árvore binária:

ÁRVORE BINÁRIA CHEIA

Uma **árvore binária cheia** é uma árvore na qual qualquer nó que possua alguma subárvore vazia situa-se no maior (último) nível. Ou seja, este tipo de árvore binária não possui nós com subárvores vazias fora do último nível. A Figura 14 exemplifica isso.

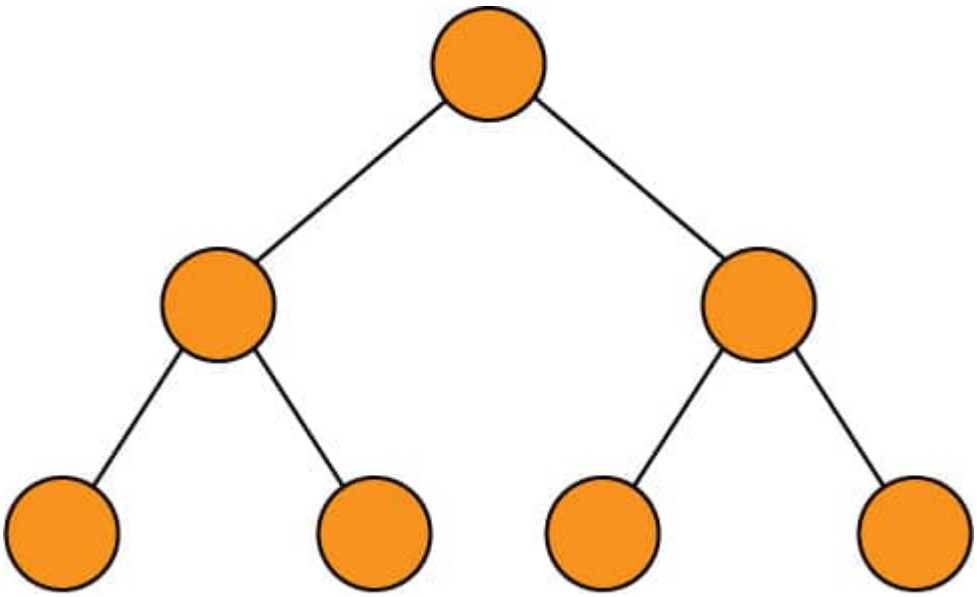


Figura 14: Árvore binária cheia.

Fonte: EnsineMe.

ÁRVORE BINÁRIA COMPLETA

Define-se, ainda, uma **árvore binária completa** como uma árvore binária na qual qualquer que seja o nó que possua uma subárvore vazia, tal nó situa-se no último ou no penúltimo nível. Veja que uma árvore binária cheia é uma árvore binária completa, mas o inverso não é verdadeiro. Uma árvore binária cheia é mostrada na Figura 15.

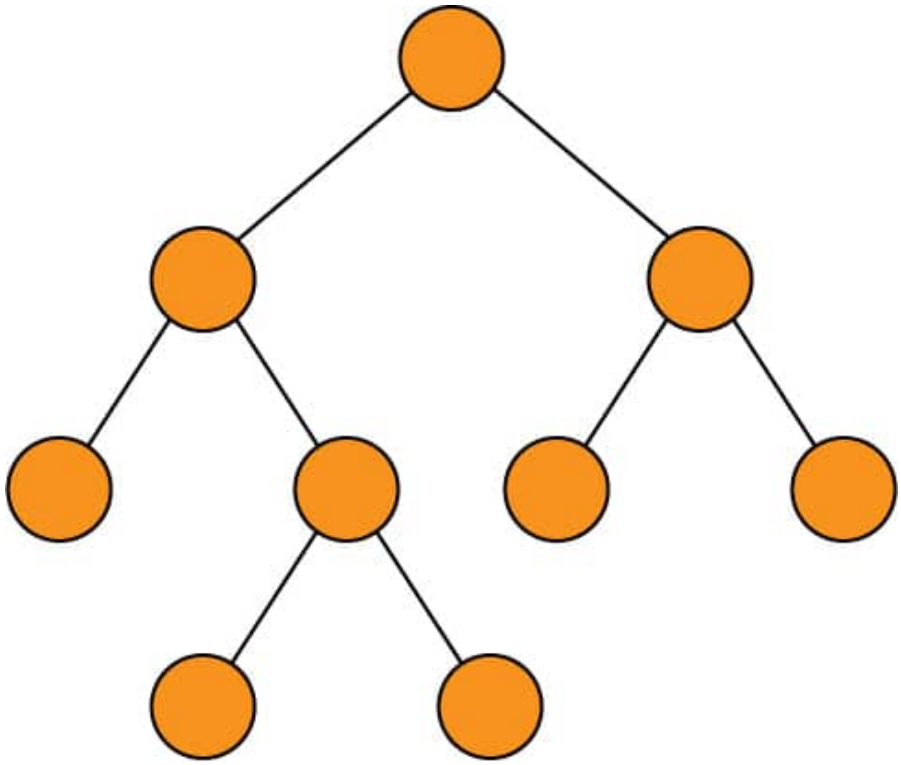


Figura 15: Árvore binária completa.

Fonte: EnsineMe.

ÁRVORE ZIGUE-ZAGUE

Chamamos, finalmente, de **árvore zigue-zague** àquela árvore binária em que todos os nós interiores possuem exatamente uma subárvore vazia. Note que, neste caso, não pode haver nós interiores com duas subárvores não vazias. Este tipo de árvore é exibido na Figura 16.

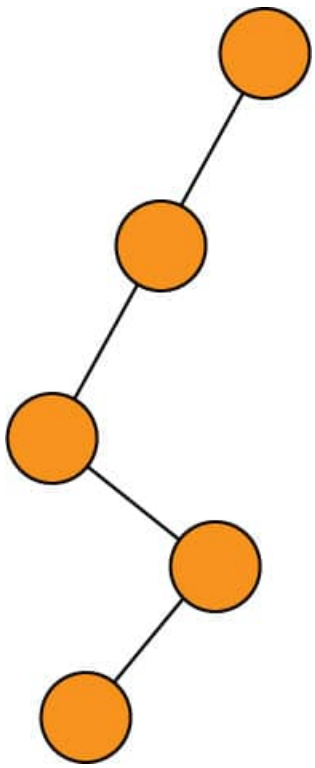


Figura 16: Árvore zigue-zague.

Fonte: EnsineMe.

PROPRIEDADES DE ÁRVORES BINÁRIAS

Já vimos as propriedades de árvores gerais. Agora veremos quais propriedades podem ser derivadas pelo fato de restringirmos o grau da árvore a no máximo 2.

COMENTÁRIO

Mas, antes, não se esqueça de que o grau da árvore corresponde ao maior grau de seus nós. Entretanto, pela definição de árvore binária, a árvore da Figura 16 é uma árvore binária, apesar de ser uma árvore de grau 1. Isto é importante para as discussões a seguir e para o correto entendimento de o que é uma árvore binária.

Nós não faremos a prova formal das propriedades. Ao invés disso, vamos nos concentrar em compreender estas propriedades através de uma discussão suficientemente estruturada. O nosso objetivo será entender como as propriedades são estabelecidas e qual a sua utilidade.

A primeira propriedade que vamos discutir é relativa à altura de uma árvore binária. Olhemos os casos extremos:

ALTURA MÁXIMA

ALTURA MÍNIMA

ALTURA MÁXIMA

Qual seria a altura máxima de uma árvore binária com n nós? Para responder a esta pergunta, precisamos imaginar como construir uma árvore binária de altura máxima. Ora, se esta árvore for construída de maneira que cada nó possua exatamente uma subárvore vazia, então vamos garantir que em cada nível haja somente um nó. Assim, conseguimos enfileirar todos os n nós

na vertical, maximizando a altura da árvore. Estamos falando, neste caso, de uma árvore zigue-zague como a da Figura 16 e que terá, portanto, altura n.

ALTURA MÍNIMA

Qual seria a altura mínima de uma árvore binária com n nós? Novamente, devemos considerar como construir tal árvore. Ora, se na construção, conseguirmos garantir que todos os nós sejam completados antes de iniciarmos um novo nível da árvore, estaremos garantindo que somente no penúltimo ou no último níveis existam nós com alguma subárvore vazia. Isto é, estamos impedindo uma árvore como a da Figura 13 e construindo uma árvore como a da Figura 15. Então, a árvore binária de altura mínima é uma árvore completa.

Consideremos melhor este caso. Imagine uma árvore binária completa T com n nós e altura h. Sendo completa, temos certeza de que somente há nós com subárvores vazias no último ou no penúltimo nível. Assim, devemos nos perguntar se há alguma forma de rearranjar os nós do último nível, de maneira que a nova altura h' seja menor do que h.

Bem, tudo que podemos fazer é tentar mover os nós do último nível para o penúltimo, pois todos os níveis diferentes do penúltimo e do último não possuem subárvores vazias. Mas, para que algum nó possa ser movido do último nível para o penúltimo, se faz necessário que algum nó do antepenúltimo nível possua pelo menos uma subárvore vazia. Mas, isto, como vimos, contraria a construção de uma árvore binária completa. Logo, nenhum rearranjo pode ser feito para eliminar o último nível e, portanto, a altura da árvore não pode ser reduzida.

Assim, a árvore binária completa tem altura mínima e tal altura é $1 + piso\ (\log\ n)$.

Lembre-se de que o logaritmo em estrutura de dados está na base 2 e não na decimal, como usualmente. Com base no que discutimos, podemos afirmar que:

Dada uma árvore binária T com $n\ >\ 0$ nós, a altura h de T é tal que
$n\ \leq\ h\ \leq\ 1\ +\ piso\ (\log\ n)$.

Outra propriedade de árvores binárias diz respeito ao número de subárvores vazias em uma árvore com $n\ >\ 0$ nós. Vejamos para o caso mais simples, ou seja, uma árvore binária com apenas um nó.

Neste caso, o nó é raiz e folha ao mesmo tempo e a árvore tem duas subárvores vazias (as subárvores esquerda e direita do nó raiz). Ou seja, se $n\ =\ 1$, o número de subárvores vazias é igual a $n\ +\ 1$.

Olhemos, então, o caso em que $n = 2$. Nesta situação, o nó raiz tem um filho, que pode estar à esquerda ou à direita, e uma subárvore vazia. O nó filho da raiz, por sua vez, é um nó folha e possui, assim, duas subárvores vazias. Essa árvore tem, então, três subárvores vazias. Então, se $n = 2$, o número de subárvores vazias se mantém igual a $n + 1$

.

Pelo princípio da indução finita, podemos demonstrar que **o número de subárvores esquerdas e direitas vazias em uma árvore binária com $n > 0$ nós é $n + 1$.**

PERCURSO EM ÁRVORES BINÁRIAS

Agora, apresentaremos alguns algoritmos usados para percorrer árvores binárias. Você reparará que são algoritmos simples e de implementação igualmente simples – todos fazem uso da recursividade. **O mais importante é você compreender o funcionamento desses algoritmos**.

Percorrer uma árvore binária significa visitar sistematicamente seus nós.

📢 ATENÇÃO

A razão de estudarmos esse tipo de operação é que árvores binárias possuem diversas aplicações, sendo muito utilizadas como indexadores, por exemplo. Assim, percorrer de forma eficiente uma árvore, tem impacto nas aplicações que a utilizam.

Antes de passarmos aos algoritmos, estabeleceremos o que significa visitar um nó. Para nosso propósito, podemos ver que:

Visitar um nó é uma generalização que engloba qualquer operação que seja feita sobre a informação contida neste nó.

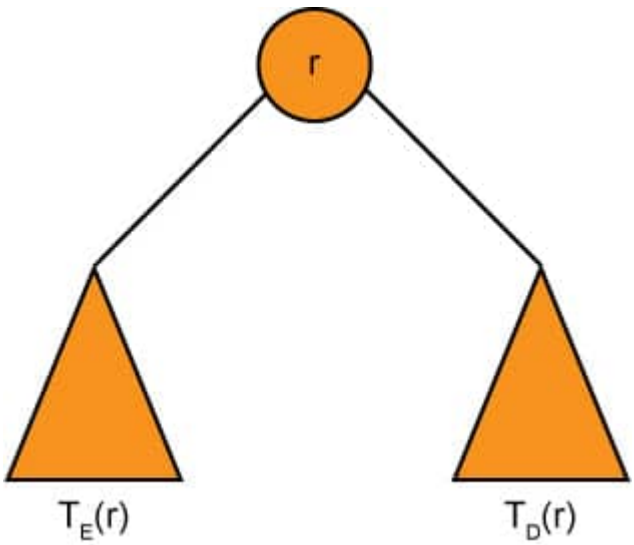
Assim, podemos visitar um nó com o intuito de imprimir seu conteúdo ou atualizá-lo, por exemplo. Estamos considerando o caso de árvores implementadas por alocação encadeada. Assim, cabe fazermos uma distinção. **Só consideramos que um nó é visitado quando a operação é realizada sobre a informação contida no nó**.

As operações realizadas sobre os campos do tipo ponteiro de um nó não são parte do conceito de visitação. Ou seja, se acessarmos um nó apenas para buscarmos o endereço de um nó filho, não o estamos visitando, mas apenas passando por ele em nosso percurso pela árvore. **Contudo, se imprimimos a informação de um nó, consideramos que este nó foi visitado**.

Para discutirmos como o percurso é feito, vamos começar com a representação de uma árvore genérica, vista na Figura 17. Nesta árvore, notamos o nó raiz (r) e suas duas subárvores $(T_E(r)$ e $T_D(r))$. Temos que ter em mente que, se desejamos percorrer uma árvore, isto significa visitar todos os seus nós. Mas, como vemos na árvore apresentada, isto implica percorrer também as subárvores esquerda e direita do nó raiz.

📣 ATENÇÃO

Veja que a recursividade do algoritmo emerge da análise do problema de forma imediata. Uma vez que desejamos percorrer uma árvore, precisamos percorrer também suas subárvores, e assim, sucessivamente, até encontrarmos uma subárvore vazia. **Eis a recursão**.



📷 Figura 17: Árvore binária genérica com raiz r.
Fonte: EnsineMe.

Contudo, a ordem na qual fazemos as operações de percorrer a subárvore esquerda, percorrer a subárvore direita e visitar o nó impacta na forma como o percurso é feito. Conforme a ordem, uma forma de se realizar o percurso na árvore pode torná-lo mais adequado para solucionar um problema.

Felizmente, há apenas três combinações dessas operações, uma vez que não se inverte a ordem em que as subárvores são percorridas, que dão origem aos **chamados percursos em**

pré-ordem, em **ordem simétrica** e em **pós-ordem**.

PERCURSO EM PRÉ-ORDEM

Consiste em se visitar o nó, percorrer em pré-ordem a subárvore esquerda e, depois, percorrer em pré-ordem a subárvore direita. O Código 5 mostra a implementação desse algoritmo. Aplicando-o à arvore da Figura 1 e considerando que visitar o nó é imprimir sua chave, temos como saída a sequência A B C D E.

```
1  void pre_ordem ( No *ptr ) {
2      printf ( ptr -> chave ); //visita o nó
3      if ptr -> filho_esquerda != NULL
4          pre_ordem ( ptr -> filho_esquerda ); //percorre recursivamente em pré-ord
5      if ptr -> filho_direita != NULL
6          pre_ordem ( ptr -> filho_direita ); //percorre recursivamente em pré-orde
7  }
```

Código 5: Percurso em pré-ordem.

PERCURSO EM ORDEM SIMÉTRICA

Consiste em se percorrer em ordem simétrica a subárvore esquerda, depois se visitar o nó e finalmente se percorrer em ordem simétrica a subárvore direita. A implementação em linguagem C é exibida no Código 6 e a aplicação do algoritmo à árvore da Figura 1 produz a saída B A D C E.

```
1  void ordem_simetrica ( No *ptr ) {
2      if ptr -> filho_esquerda != NULL
3          ordem_simetrica ( ptr -> filho_esquerda ); //percorre recursivamente em o
4      printf ( ptr -> chave ); //visita o nó
5      if ptr -> filho_direita != NULL
6          ordem_simetrica ( ptr -> filho_direita ); //percorre recursivamente em ord
7  }
```

Código 6: Percurso em ordem simétrica.

PERCURSO EM PÓS-ORDEM

Consiste em se percorrer em pós-ordem a subárvore esquerda, depois percorrer em pós-ordem a subárvore direita e, finalmente, visitar o nó. Este algoritmo é mostrado implementado no Código 7. Aplicando-o à árvore da Figura 1, temos como saída B D E C A.

```c
void pos_ordem ( No *ptr ) {
    if ptr -> filho_esquerda != NULL
        pos_ordem ( ptr -> filho_esquerda ); //percorre recursivamente em pós-or
    if ptr -> filho_direita != NULL
        pos_ordem ( ptr -> filho_direita ); //percorre recursivamente em pós-ord
    printf ( ptr -> chave ); //visita o nó
}
```

Código 7: Percurso em pós-ordem.

COMENTÁRIO

Os códigos apresentados implementam os três tipos de percursos vistos. Como há grandes semelhanças entre os mesmos, pela simplicidade envolvida, vamos nos limitar a explorar o Código 6. Extrapolar para os demais algoritmos é trivial.

```c
void ordem_simetrica ( No *ptr ) {
    if ptr -> filho_esquerda != NULL
        ordem_simetrica ( ptr -> filho_esquerda ); //percorre recursivamente em o
    printf ( ptr -> chave ); //visita o nó
    if ptr -> filho_direita != NULL
        ordem_simetrica ( ptr -> filho_direita ); //percorre recursivamente em or
}
```

Código 6: Percurso em ordem simétrica.

Primeiramente, note que os códigos apresentados precisam receber como parâmetro um ponteiro para a árvore a ser percorrida. Na primeira chamada do algoritmo, a função receberá como parâmetro o ponteiro para o nó raiz da árvore binária a ser percorrida. Nós não testamos se o nó raiz é nulo, porque não faz sentido tentar percorrer uma árvore vazia. Este teste deve ser feito, portanto, pelo código chamador da função.

Uma vez que a função de percurso foi invocada, no caso do Código 6, a primeira coisa que é feita é a verificação se a subárvore esquerda do nó apontado por “ptr” é vazia (linha 2).

Caso não seja, o algoritmo de percurso é chamado recursivamente, passando como parâmetro o ponteiro para o nó raiz da subárvore esquerda (linha 3).

Se a subárvore esquerda for vazia, passa-se à operação seguinte que, no caso do Código 6, é a visitação do nó (linha 4).

Uma vez que o nó tenha sido visitado, a função verifica se a subárvore direita do nó apontado por “ptr” é vazia. Tal procedimento é análogo ao que foi feito para a subárvore esquerda.

Assim, a linha 5 realiza a chamada recursiva da função de percurso passando como parâmetro o ponteiro para o nó raiz da subárvore direita. Mas a linha 5 só é executada se a subárvore direita não for vazia.

Esse princípio de funcionamento é o mesmo para os três códigos mostrados. Todos têm em comum o recebimento de um ponteiro para o nó raiz da árvore na primeira chamada. Da mesma forma, as chamadas recursivas sempre passam como parâmetro ponteiros para os nós

raízes das subárvores. Estas, entretanto, são sempre testadas para verificar sua existência, pois uma subárvore vazia é condição de impedimento para a recursão.

Olhando, ainda, os códigos mostrados, você perceberá que, nestes casos, a visitação significou a impressão da informação armazenada no nó. Mas, os códigos podem ser facilmente alterados com outros fins, por exemplo, retornar o endereço de cada nó da árvore.

📢 ATENÇÃO

Outra observação relevante válida para todos os algoritmos é que estes são chamados tantas vezes quanto são os nós a serem visitados. Logo, numa árvore com n nós, haverá n chamadas às funções de percurso. Veja, também, que você pode ser obrigado a passar mais de uma vez por um nó, para acessar sua subárvore direita. Contudo, dentro da argumentação que fizemos antes, os nós são visitados apenas uma vez.

Embora o algoritmo recursivo seja consideravelmente mais simples, é possível realizar-se a implementação dos percursos na árvore através de algoritmos não recursivos. Neste caso, contudo, a programação se torna mais complexa, pois é necessário a cada passo do algoritmo que se mantenha a informação de qual direção (esquerda ou direita) se está seguindo. Também é preciso controlar quais ramos já foram percorridos.

Outras dificuldades podem surgir, mas o ponto relevante é ilustrar como a recursividade simplifica essa implementação.

Esses algoritmos apresentados foram construídos para árvores binárias. Mas podem ser generalizados, sem grandes dificuldades, para árvores m-árias. Você só precisa considerar que os percursos em pré-ordem e em pós-ordem se caracterizam apenas pela ordem em que o nó é visitado e as subárvores são percorridas recursivamente:

PERCURSO EM PRÉ-ORDEM

PERCURSO EM PÓS-ORDEM

PERCURSO EM ORDEM SIMÉTRICA

PERCURSO EM PRÉ-ORDEM

Primeiro se visita o nó e depois se percorre recursivamente em pré-ordem todas as subárvores.

PERCURSO EM PÓS-ORDEM

Primeiro se percorre recursivamente em pós-ordem todas as subárvores e depois se visita o nó. Esse procedimento abarca perfeitamente árvores *m*-árias.

PERCURSO EM ORDEM SIMÉTRICA

Tem, apenas como desafio a mais, o posicionamento da visita ao nó entre os percursos recursivos das subárvores.

Por fim, os percursos apresentados não são os únicos. Outros algoritmos podem ser estabelecidos, mas os três apresentados são de fundamental interesse.

★ EXEMPLO

Podemos empregar o Código 7 para calcular a altura de uma árvore. Basta, para tanto, alterar a visita para que, ao invés de imprimir a informação existente no nó, ela realize o cálculo do nível do nó. O Código 8 mostra uma forma de se realizar tal cálculo. Para usá-lo, a linha 6 do Código 7 deve ser substituída pela chamada "$altura\ (No * ptr)$".

```
1  static void altura ( No *ptr) {
2      if ptr -> filho_esquerda != NULL
3          altura_subarvore_esquerda = ptr -> filho_esquerda -> altura;
4      else
5          altura_subarvore_esquerda = 0;
6      if ptr -> filho_direita != NULL
7          altura_subarvore_direita = ptr -> filho_direita -> altura;
8      else
9          altura_subarvore_direita = 0;
10     if altura_subarvore_esquerda > altura_subarvore_direita
11         ptr -> altura = altura_subarvore_esquerda + 1;
12     else
13
```

```
14          ptr -> altura = altura_subarvore_direita + 2;
    }
```

Código 8: Função para calcular a altura de uma árvore binária.

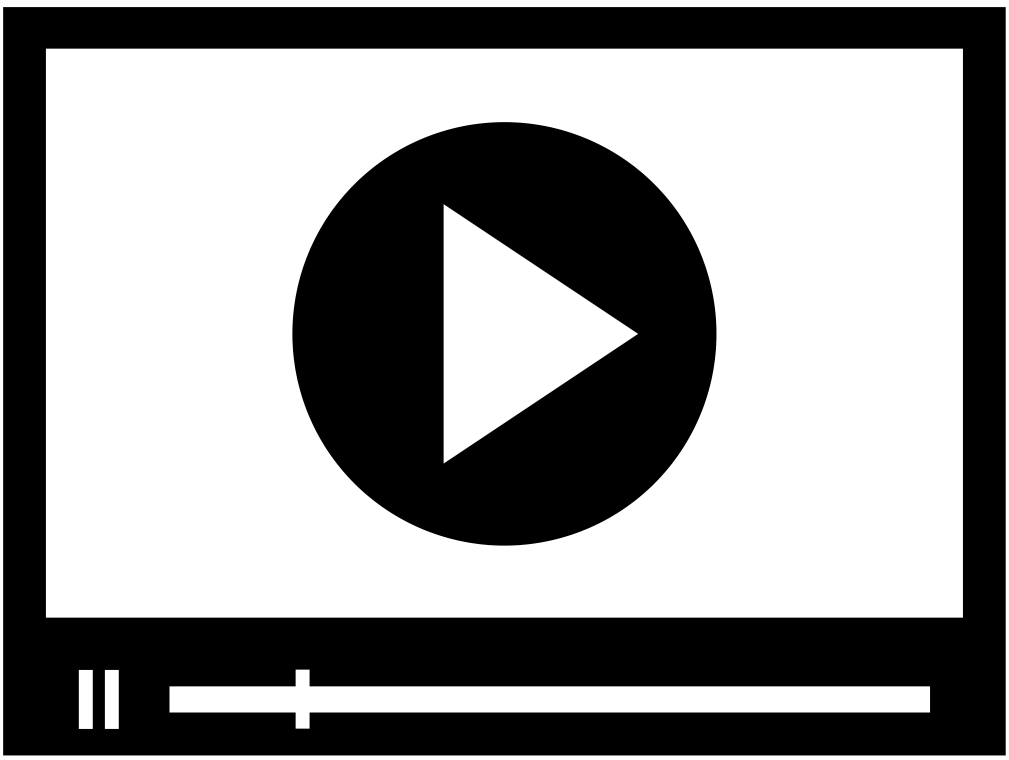
Algumas observações são necessárias para melhor compreensão do código mostrado.

Em primeiro lugar, o código considera que há mais um campo nos nós, chamado “altura”, e que armazena a altura calculada de cada nó.

As variáveis “altura_subarvore_esquerda” e “altura_subarvore_direita” são variáveis estáticas.

A altura da árvore emergirá do maior valor entre as variáveis citadas. Esta é apenas uma maneira de se aproveitar o percurso para se realizar o cálculo da altura.

Na próxima seção, apresentaremos alguns tipos específicos de árvores empregadas para pesquisa. São tipos de árvores voltadas para a busca e se constituem numa aplicação de árvores para um problema fundamental da computação.



No vídeo a seguir, reveja de forma prática o conceito de árvore binária.

VERIFICANDO O APRENDIZADO

MÓDULO 3

- **Reconhecer as principais árvores de pesquisa, suas particularidades e aplicações**

ÁRVORES DE PESQUISA

Esta seção coroa o estudo realizado. Ela se propõe a apresentar um uso prático de árvores. Você verá como árvores são utilizadas para solucionar problemas de busca.

Este tipo de problema surge quando dado um conjunto de elementos, cada um identificado por uma chave (a qual é única), se quer localizar um elemento específico mediante a procura por sua chave.

Aplicar árvores a este tipo de problema significa distribuir os elementos e suas respectivas chaves pelos nós da árvore. Para fins didáticos, em nossos exemplos e discussões, vamos considerar apenas a chave, que é o indexador do conjunto. Porém, uma vez que há vínculo entre a chave e o elemento do conjunto, recuperar a chave é recuperar o elemento.

O problema de busca está presente em diversas aplicações. Já mencionamos antes o uso de árvores em bancos de dados, mas elas também são empregadas, por exemplo, na indexação de arquivos, em sistemas operacionais, de espaços usados e disponíveis de alocação em disco e várias outras situações. Por esta razão, encontrar maneiras de se otimizar as

operações em árvores tem grande relevância. Por isso, examinaremos, também, o balanceamento de árvores.

A ÁRVORE BINÁRIA DE BUSCA (ABB)

Uma árvore binária de busca é uma árvore binária, estabelecida com algumas particularidades, ou seja, é uma árvore binária rotulada. Antes de definimo-la, contudo, precisamos formalizar o nosso problema de busca. Como vimos, trata-se de encontrar um elemento através da busca de sua chave, caso esta exista no espaço de busca.

Formalmente, dado um conjunto de chaves $C = \{c_1, c_2, \ldots, c_n\}$ tal que $c_1 < c_2 < \ldots < c_n$, o problema de busca consiste em se identificar o valor de n que corresponde à chave procurada, ou identificar que a chave não pertence ao conjunto C, se este for o caso.

📢 ATENÇÃO

Repare que as chaves possuem uma relação de ordem entre si. Este fato será considerado na rotulagem da árvore binária.

Para fazer frente a esse problema, define-se uma árvore binária de busca para C como sendo uma árvore binária rotulada T segundo as seguintes regras:

> T possui k nós, sendo que cada nó de T corresponde a uma única chave distinta de C, e vice-versa. Cada nó possui como rótulo o valor c_i que é a chave correspondente.
>
> Qualquer que seja o nó v de T e os nós v_1 e v_2, respectivamente pertencentes à subárvore esquerda e à direita de v, então: O rótulo de v_1 é menor do que o rótulo de v e o rótulo de v_2 é maior do que o rótulo de v.

Uma coisa interessante de se observar é de que forma esta definição afeta a árvore binária. Veja que, para que ela seja uma ABB e, desta forma, possa solucionar o problema de busca, uma regra de ordenação foi imposta.

Todos os nós da subárvore esquerda de um nó v qualquer ou são inexistentes ou são menores do que v. Analogamente, todos os nós da subárvore direita de v ou são inexistentes ou são maiores que v.

Mas isto não implica na existência de uma única árvore binária de busca para o conjunto C quando o número de elementos for maior do que 1. De fato, existe mais de uma ABB para um conjunto C neste caso.

Veja dois exemplos para o conjunto $C = \{1, 2, 3, 4, 5\}$, mostrados na Figura 18. Cabe também observar que usaremos chaves numéricas para facilitar o entendimento, mas desde que haja uma relação de precedência, chaves alfabéticas também poderiam ser utilizadas (por exemplo: $A < B < C \ldots < Z$).

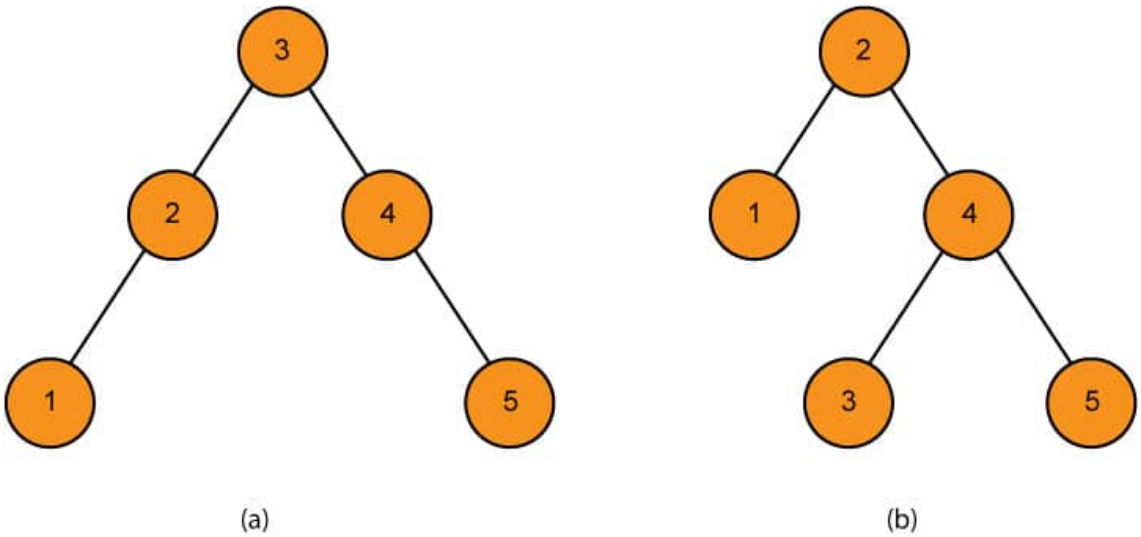


Figura 18: ABB para o conjunto C = {1, 2, 3, 4, 5}.

Fonte: EnsineMe.

Utilizemos, agora, a árvore “b” da Figura 18 para entender como ela nos ajuda na busca por uma chave.

★ EXEMPLO

Suponha que estejamos buscando a chave 3. Neste caso, o primeiro nó a ser analisado é a raiz. Se a raiz contiver a chave buscada, nossa busca terminou. Mas este não é o caso. Vamos, então, comparar a chave buscada com o rótulo da raiz. Como $3 > 2$, sabemos que, se a chave existir, ela estará na subárvore direita da raiz. Passamos, então, a analisar o nó “4”. Agora, vemos que $3 < 4$, o que indica que a chave buscada, se existir, estará à esquerda do nó “4”. Na próxima iteração, finalmente encontramos a chave buscada.

No nosso exemplo, estamos conduzindo uma busca num conjunto pequeno, com apenas 5 chaves. Mas observe que foram feitas apenas 3 comparações, de 5 possíveis. Se aplicássemos a busca na árvore “a”, faríamos apenas 1 comparação.

Então, duas considerações merecem ser feitas:

> Em primeiro, a construção da árvore binária de busca tem impacto na busca.

> Em segundo, no caso médio, vemos que a ABB reduziu o número de comparações necessárias para se encontrar a chave.

É claro que se a árvore fosse um caso degenerado em árvore zigue-zague, com a raiz sendo 1, no pior caso teríamos que fazer 5 comparações, e não tiraríamos nenhuma vantagem da ABB. Mas é por isso que o balanceamento de árvores de busca é importante, como veremos mais à frente. Note que o pior caso para as árvores da Figura 18 é realizar 3 comparações.

💬 COMENTÁRIO

Você deve ter notado, também, que no pior caso, o número de vezes em que a comparação ocorre é igual à altura da árvore. Assim, árvores binárias de busca de altura mínima são vantajosas para as buscas.

O caso de uma busca em que a chave não está na árvore é trivial. Imagine que estejamos buscando a chave de valor 6. Todo o processo se passaria como descrito anteriormente até atingir-se o nó “5”. Neste ponto, uma nova comparação $(6 > 5)$ indicaria que a chave 6, se existir, estará à direita do nó. Entretanto, a subárvore direita do nó “5” é vazia. Logo, a chave 6 não faz parte do conjunto C.

O Código 9 mostra uma implementação do algoritmo de busca em uma ABB.

```
int busca_arvore ( int chave , No *ptr ) {
    if ( ptr->chave == chave )
        return 1; //chave encontrada
    else if ( chave < ptr->chave )
        if ( ptr->filho_esquerdo == NULL )
            return 0; //chave não encontrada
        else
            busca_arvore ( chave , ptr->filho_esquerdo );
    else if ( ptr->filho_direito == NULL )
```

```
10            return 0; //chave não encontrada
11        else
12            busca_arvore ( chave , ptr->filho_direito );
13    }
```

Código 9: Algoritmo de busca em uma ABB.

Como vimos, a altura da árvore binária de busca é de especial interesse. Logo, é interessante estabelecer uma relação entre o número de nós (n) de uma árvore binária completa e sua altura.

Sabemos que, para uma árvore binária completa, sua altura (h) é mínima, de forma que $h \geq 1 + piso\,(\log\, n)$ – lembre-se de que o logaritmo em estrutura de dados está na base 2 e não na decimal, como usualmente. Resolvendo esta desigualdade para n, temos que $n \geq 2^{(h-1)}$. Por outro lado, observando-se uma árvore cheia, vemos que o número de nós dobra a cada nível. É fácil ver que com altura igual a 2, há 3 nós, com altura igual a 3, há 7 nós, e assim, sucessivamente. Logo, o número de nós em uma árvore binária cheia é $n = 2^h - 1$.

Podemos, então, estabelecer a relação pretendida:

Numa árvore binária completa com n nós e altura h, $2(h-1) \leq n \leq 2h-1$.

A esta altura, já deve ter ficado claro que a forma como a árvore binária de busca é construída tem impacto na busca, ou seja, na sua principal finalidade. Vamos, então, considerar de forma mais profunda o problema de construção de uma ABB.

Inicialmente, consideremos um procedimento que, à medida que recebe as chaves do conjunto C de chaves, as insere na árvore respeitando a definição dada de ABB. Este algoritmo, aplicado ao conjunto $C = \{1, 2, 3, 4, 5, 6, 7\}$ produzirá uma árvore zigue-zague com todos os nós inseridos nas subárvores direitas e raiz “1”. Tal árvore terá altura máxima, igual ao número de nós inseridos. Ou seja, tal procedimento é sensível à ordem em que as chaves são ordenadas no conjunto C. Mas, se você revir a Figura 18 (a), notará que ter na raiz a chave média permitiu distribuir de forma equitativa as demais chaves.

📢 ATENÇÃO

A meta é, portanto, construir a árvore binária de busca de maneira que ela seja uma árvore binária completa.

ÁRVORES BALANCEADAS

A discussão da subseção anterior vem reforçar a preocupação em mantermos uma árvore sempre eficiente para a busca.

Ao longo do tempo, uma árvore binária de busca pode sofrer inserções e remoções que piorem a execução da busca. Mesmo que uma árvore comece completa, nada impede que ela, após algumas inserções ou remoções, perca essa propriedade. Num caso extremo, ela pode se degenerar numa árvore zigue-zague. O que desejamos é manter o custo de acesso às chaves no seu mínimo, mesmo com a execução de operações que alterem a árvore. Isto é, queremos garantir que a árvore continue tendo desempenho ótimo.

É claro que, para que a árvore se mantenha ótima mesmo com operações de inserção e remoção ocorrendo, um esforço adicional deve ser realizado. Tal esforço será empregado no sentido de corrigir os desvios inseridos pelas operações, fazendo com que a árvore retorne ao seu estado mais eficiente no acesso às chaves.

Uma árvore que mantém o custo de acesso às chaves na mesma ordem de grandeza de uma árvore ótima, durante seu uso, é chamada de **balanceada**.

Já estabelecemos na subseção anterior que uma árvore binária completa é a que minimiza as comparações necessárias, no pior caso, para uma busca. Por isso, como já vimos, a árvore binária completa é uma meta na construção.

💬 COMENTÁRIO

Você pode pensar que, por isso, manter a árvore completa também é uma meta do balanceamento da árvore. Mas, neste ponto, é importante considerarmos a razão que nos leva a tentar manter a árvore de busca ótima.

Esta razão é a eficiência.

Tentamos usar uma árvore de busca completa para que a eficiência da busca seja ótima. Estamos pensando, assim, no desempenho no emprego da árvore de busca para solucionar algum problema prático.

Como já apontamos, balancear uma árvore significa empreender um esforço no sentido de mantê-la com um desempenho com a mesma ordem de grandeza que a árvore ótima.

Mas, e se o esforço para se balancear uma árvore for tal que, apesar de a busca se manter ótima, supere em muito o custo da busca?

RESPOSTA

Neste caso, as operações de inserção e remoção, que já possuem um custo, seriam acompanhadas do custo adicional de balanceamento, que tenderia a prejudicar o emprego da árvore como uma solução viável em um problema real. Infelizmente, este é o tipo de coisa que pode vir a ocorrer com árvores completas.

Uma saída para este problema é relaxarmos o desempenho de busca. Ao invés de visarmos a um desempenho com máxima eficiência, ou seja, realizado no número mínimo de passos $1 + piso\,(\log\, n)$, aceitamos que o desempenho na busca possa ser pior, mas exigimos que a altura da árvore seja a mesma da altura de uma árvore completa com o mesmo número de nós.

Tal imposição deve ser feita recursivamente, de forma que as subárvores obedeçam a mesma limitação, considerados os números de nós de cada subárvore. Uma tal árvore é dita balanceada.

A motivação para esse relaxamento é melhorar o desempenho do esforço de balanceamento, sem comprometer o desempenho de busca. Com essa limitação, a altura da árvore poderá ser até da ordem de grandeza de $\log\, n$, o que ainda garante um bom desempenho de busca.

Estudaremos, na próxima subseção, um exemplo de árvore balanceada chamada de árvore AVL.

ÁRVORE AVL

Árvores AVL são um tipo especial de árvores binárias que buscam atingir e manter um bom desempenho de busca em tempo de execução.

Alguns autores consideram a árvore AVL como uma árvore balanceada, outros, no entanto, consideram-na praticamente balanceada, adotando uma definição bem mais rígida para o balanceamento. Para nossa discussão, consideraremos a árvore AVL como uma árvore

balanceada e vamos defini-la como se segue: Uma árvore AVL é uma árvore binária de busca tal que:

> As alturas das subárvores esquerda e direita do nó raiz diferem por, no máximo, uma unidade.
>
> As subárvores esquerda e direita do nó raiz são, por sua vez, árvores AVL.

📢 ATENÇÃO

Um nó que atende à primeira parte da definição (que estipula que a altura entre as subárvores do nó deve diferir de no máximo uma unidade) é chamado **de nó regulado**. Quando um nó não atende a essa limitação, ele é chamado de **desregulado**.

Veja, também, que essa definição é recursiva. Isto significa que, ao dizer que as subárvores esquerda e direita do nó raiz são, por sua vez, árvores AVL, estamos dizendo que o nó raiz dessas subárvores também é um nó regulado. E assim, sucessivamente.

Uma árvore que possui todos os nós regulados é uma árvore balanceada. Porém, uma árvore que possua ao menos um nó desregulado é dita desregulada. É imediato ver que toda árvore completa é uma árvore AVL. Pela sua definição, só há subárvores vazias no penúltimo ou no último nível. Suponha um nó v do penúltimo nível que possua a subárvore esquerda vazia e um filho d à direita. Pela definição, d necessariamente é folha. Então, a diferença de altura entre as subárvores de v é de uma unidade. Nos demais casos, as diferenças serão zero. **Assim, evidenciamos que uma árvore completa é AVL**.

Perceber que o inverso não é necessariamente verdadeiro também deve ser fácil. Basta lembrar que uma árvore AVL não impõe restrições a que haja nós com subárvores vazias no antepenúltimo nível.

A Figura 19 mostra um exemplo de árvore AVL que não é completa.

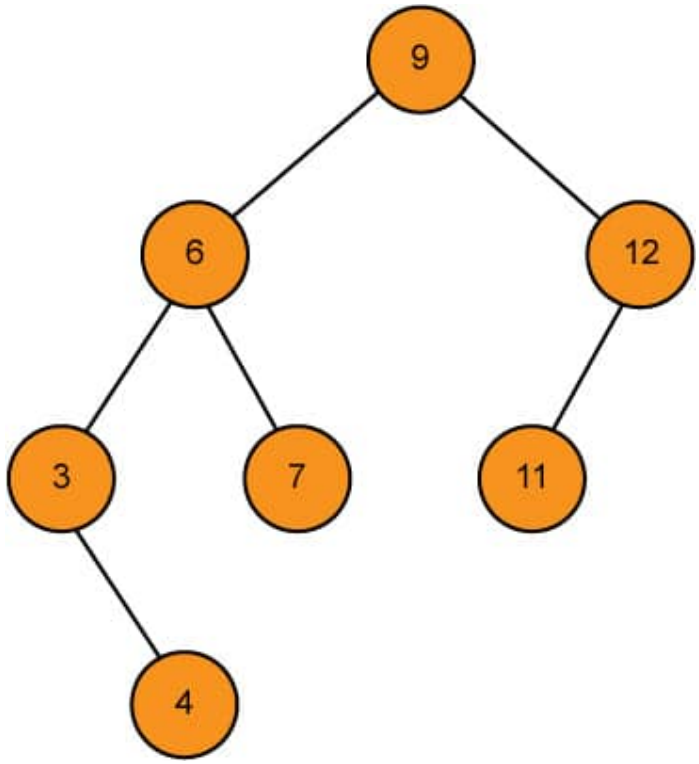


Figura 19: Árvore AVL.

Fonte: EnsineMe.

Agora que já apresentamos a definição de uma árvore AVL, vamos entender como essa definição produz uma árvore balanceada. Para isso, iremos determinar o valor mínimo de nós que podem existir em uma árvore AVL de altura h.

O primeiro nó a ser examinado é o nó raiz. Neste caso, como se trata de uma árvore AVL, sabemos que a diferença entre as alturas de suas subárvores pode ser, no máximo, igual a 1. Ora, se a altura da árvore é h, então a maior altura das subárvores da raiz pode ser $h - 1$. Assim, a outra subárvore poderá ter altura $h - 1$ ou $h - 2$, a fim de não violar a condição de árvore AVL. Se buscamos o número mínimo de nós, então devemos olhar para o caso em que a outra subárvore possui altura $h - 2$. A aplicação recursiva deste raciocínio permitirá construir uma árvore AVL com altura h. Para esta árvore AVL, o número de nós n se relaciona com a altura h de forma que h é sempre menor que a ordem de grandeza de log n. Isto, conforme vimos na subseção anterior, é a altura imposta a uma árvore balanceada.

Precisamos, agora, compreender o mecanismo de balanceamento. Começaremos por estudar a inserção em árvores AVL. Durante a inserção, como já debatemos, alterações podem ocorrer que demandem o esforço de balanceamento do nó. Isto implica que, a cada inserção, os nós da árvore precisam ser checados para verificar-se se algum se tornou desregulado. Caso nenhum nó tenha se desregulado, nada precisa ser feito. Contudo, se um nó se tornou desregulado, faz-se necessário operar sobre a árvore, de forma a restabelecer o seu balanceamento.

Para restabelecer o balanceamento de uma árvore AVL que se tornou desbalanceada, podemos empregar quatro transformações. Estas transformações são:

ROTAÇÃO SIMPLES À DIREITA.

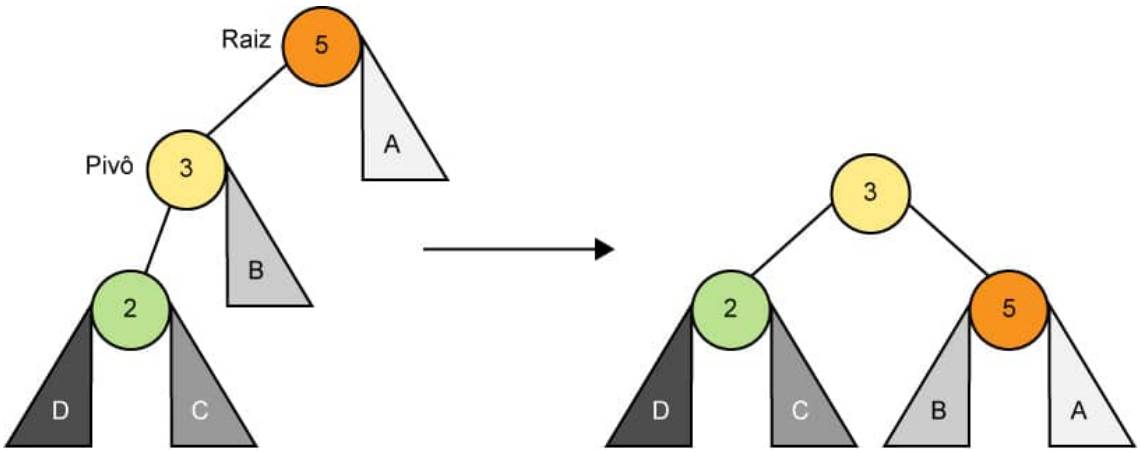


Figura 20: Rotação simples à direita.

Fonte: Luizcbn, 2014a.

ROTAÇÃO SIMPLES À ESQUERDA.

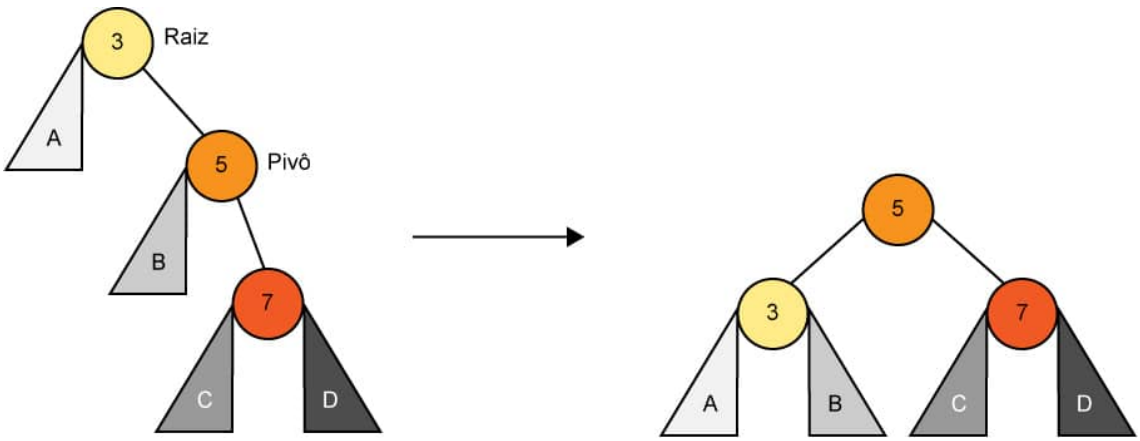


Figura 21: Rotação simples à esquerda.

Fonte: Luizcbn, 2014a.

ROTAÇÃO DUPLA À DIREITA.

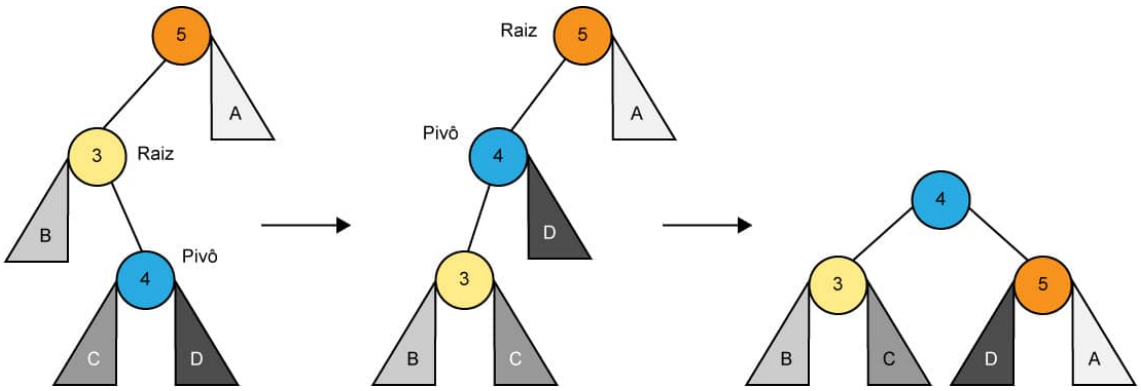


Figura 22: Rotação dupla à direita.

Fonte: Luizcbn, 2014a.

ROTAÇÃO DUPLA À ESQUERDA.

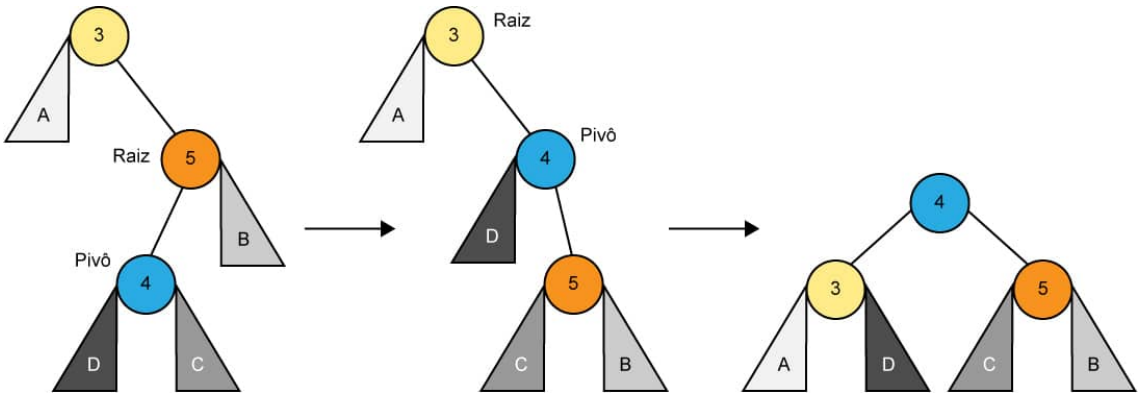


Figura 23: Rotação dupla à esquerda.

Fonte: Luizcbn, 2014a.

Vejamos, agora, de que forma a inclusão pode ensejar o uso destas transformações para se balancear novamente a árvore.

Seja uma árvore AVL T inicialmente regulada. Suponha que um nó q seja inserido em T. Se tal inserção não provocou a desregulagem de nenhum nó de T, então nada precisa ser feito, pois a árvore permanece AVL. No entanto, caso a inserção tenha provocado o desbalanceamento de T, precisaremos empregar as transformações para regular os nós desregulados.

Seja, então, o nó p desregulado mais próximo das folhas de T. Sabemos que p se encontra no caminho de q até a raiz, pois do contrário ele não teria sido desregulado pela inserção de q.

Usaremos $h_E(p)$ e $h_D(p)$ para representar respectivamente as alturas das subárvores esquerda e direita de p. Como p se tornou desregulado, então sabemos que $| \, h_E \, (p) - \, h_D \, (p) \, | \, = \, 2$.

Para sabermos qual transformação empregar, necessitamos avaliar as quatro a seguir:

1

A inserção ocorreu na subárvore esquerda do filho u esquerdo de $p \, (\, h_E \, (p) \, > \, h_D \, (p)$ **e** $h_E \, (u) \, > \, h_D \, (u) \,)$**.**

Neste caso, $h_E \, (u) \, - \, h_D \, (u) \, = \, 1$, pois u não é desregulado e $h_D \, (p) \, = \, h_D \, (u)$, pois p está desregulado. O balanceamento pode, então, ser recuperado através de uma rotação à direita.

2

A inserção ocorreu na subárvore direita do filho u **esquerdo de** $p \, (\, h_E \, (p) \, > \, h_D \, (p)$ **e** $h_E \, (u) \, < \, h_D \, (u) \,)$**.**

Neste caso, u também possui um filho v, tal que $| \, h_E \, (v) \, - \, h_D \, (v) \, | \, = \, 1$. Além disso, $h_D(p) = h_E(u) = a \, máximo \, \{h_E(v), h_D(v)\}$. Para balancear a árvore, deve-se realizar uma rotação dupla à direita.

3

A inserção ocorreu na subárvore direita do filho z direito de $p \, (\, h_E(p) \, < \, h_D(p)$ **e** $h_E(z) \, < \, h_D(z) \,)$**.**

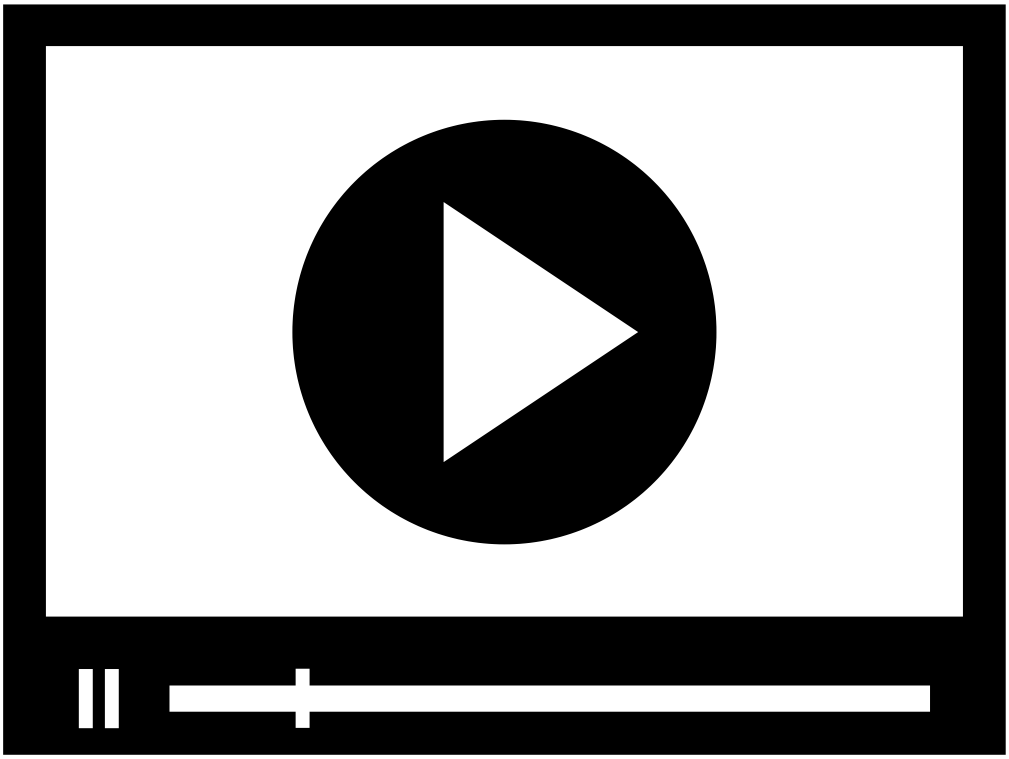
Neste caso, temos que $h_D \, (z) \, - \, h_E \, (z) \, = \, 1$ e $h_E \, (p) \, = \, h_E \, (z)$. O balanceamento, então, pode ser feito através de uma rotação à esquerda.

4

A inserção ocorreu na subárvore esquerda do filho z **esquerdo de** $p \, (\, h_E \, (p) \, < \, h_D \, (p)$ **e** $h_E \, (z) \, > \, h_D \, (z) \,)$**.**

Neste caso, z possui um filho à esquerda y e as relações entre as alturas das subárvores de p, z e y são análogas ao segundo caso. Logo, o balanceamento pode ser feito através de uma rotação dupla à esquerda.

As árvores AVL são um importante exemplo de árvores balanceadas, mas não são as únicas árvores que obedecem a este conceito. Entretanto, a compreensão do seu funcionamento é importante para fortalecer os conceitos envolvidos no uso de árvores binárias de busca.



No vídeo a seguir, reveja de forma prática o conceito de árvore binária de busca e árvore AVL.

VERIFICANDO O APRENDIZADO

CONCLUSÃO

CONSIDERAÇÕES FINAIS

Como vimos, árvores são de grande importância na Computação. Nesta oportunidade, apresentamos os conceitos básicos e a terminologia de árvores. Procuramos nos aprofundar nas definições mais importantes para sedimentar o conhecimento geral de árvores, apresentamos as principais propriedades e discutimos sua representação computacional.

Em seguida, trouxemos os conceitos e propriedades de árvore binária, a qual foi definida e seu conceito esmiuçado. Mostramos diversas terminologias comuns e relevantes para árvores binárias. Em seguida, exibimos as principais propriedades e os percursos em pré-ordem, ordem simétrica e pós-ordem.

Após a árvore binária, discutimos as árvores de pesquisa, começando pela árvore binária de busca (ABB). A ABB foi definida e procuramos compreender as consequências desta definição. Também apresentamos conceitos e propriedades relativos à ABB. Prosseguimos conceituando balanceamento de árvore e explicando sua importância. Após isso, apresentamos a árvore AVL, um importante exemplo de ABB balanceada. Para a árvore AVL, mostramos as formas de regulagem de um nó desregulado e explicamos como este tipo de árvore representa uma melhora no uso de ABB.

⚠ PODCAST

REFERÊNCIAS

MALIK, D. S. **Data Structures using C++**. 2nd. ed. Course Technology CENCAGE Learning, 2010. Consultado em meio eletrônico em: 18 nov. 2020.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de Dados e seus Algoritmos**. 3. ed. Rio de Janeiro: LTC/Grupo Gen, 2010.

EXPLORE+

Para saber mais sobre os assuntos tratados neste tema, pesquise na internet:

Aplicação de árvores em sistemas de arquivo e banco de dados.

Outras formas de percurso em árvores, como o percurso em profundidade ou o percurso em nível.

Árvores de partilha, árvores rubro-negras e árvores B.

CONTEUDISTA

Marlos de Mendonça Corrêa

🔗 CURRÍCULO LATTES