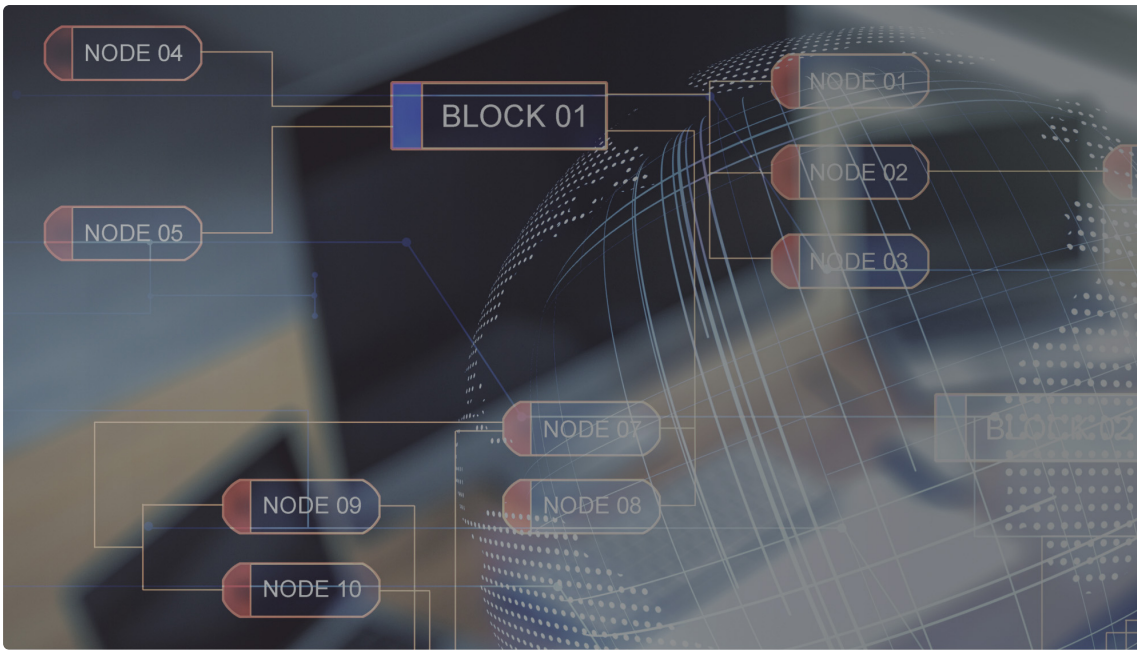


Árvores em Python

Prof.ª Simone Ingrid Monteiro Gama

Descrição

Apresentação dos principais conceitos relacionados à árvores de busca, suas principais operações (busca, inserção e remoção) e seus algoritmos, bem como a sua implementação em linguagem Python.

Propósito

Uma possível forma de organizar a informação em uma estrutura de dados é por meio da árvore binária de busca e o propósito do estudo dessas estruturas é fornecer uma estrutura de dados na qual o problema da busca, inserção e remoção é resolvido em tempo proporcional ao logaritmo do número de chaves na estrutura de dados.

Objetivos

Módulo 1

Conceitos de árvores

Descrever os principais conceitos de árvores de busca e árvores binárias de busca.

Módulo 2

Árvores binárias

Identificar as principais operações em árvores binárias de busca implementadas em linguagem Python.

Módulo 3

Percursos em árvores

Identificar os principais algoritmos de percursos em árvores binárias implementados em linguagem Python.

Módulo 4

Aplicação de percursos em árvores

Descrever exemplo de aplicação dos percursos em árvores sobre o problema da avaliação de expressões aritméticas.

Introdução

O principal problema na manipulação de uma massa de dados é o referente à busca, inserção e remoção. Adotando uma estrutura de dados não organizada, podemos resolver esse problema com

complexidade computacional de O(n), que não pode ser considerada alta, entretanto, pode ser otimizada.

Neste conteúdo, vamos aprender a organizar uma estrutura de dados capaz de realizar busca, inserção e remoção em complexidade computacional de *0 (log n)*, que é a otimização natural para *0(n)*. Para tal, partiremos do conceito de árvores de busca e árvores binárias de busca, explorando suas principais operações, bem como o conceito de percurso em árvores. Em seguida, definiremos as árvores binárias de busca e seus principais algoritmos, comparando a complexidade computacional com as listas lineares e as estruturas de dados não organizados.

No vídeo a seguir, conheça a importância do estudo formal de algoritmos, destacando a principal ferramenta de avaliação: a complexidade computacional.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

1 - Conceitos de árvores

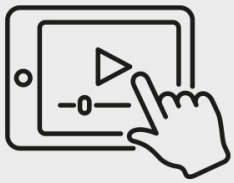
Ao final deste módulo, você será capaz de descrever os principais conceitos de árvores de busca e árvores binárias de busca.

Árvores

O que são árvores?

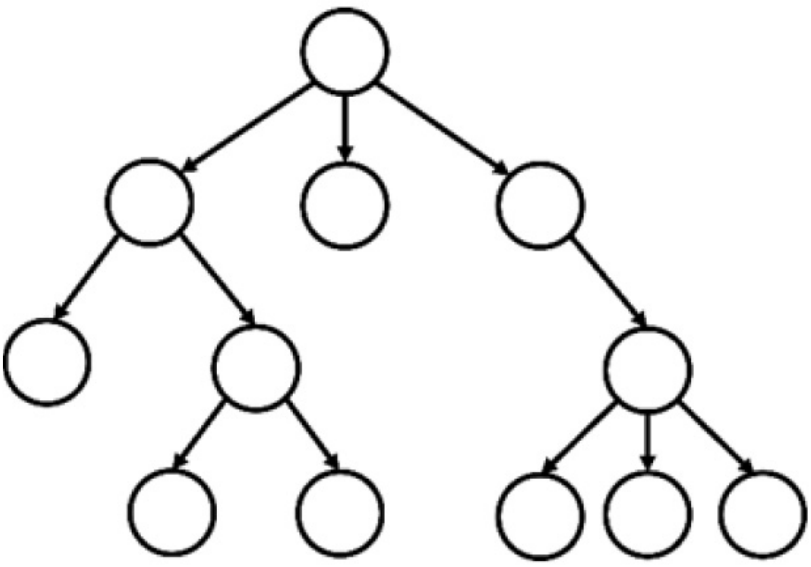
Conheça agora a definição formal de árvores e como elas são representadas em memória.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Árvores (*trees*) são estruturas de dados hierárquicas e não linearizadas. Basicamente, as árvores são formadas por um conjunto de elementos, que chamamos de **nós** (ou nodos ou vértices), conectados por um conjunto de arestas. Um dos nós, que dizemos estar no nível 0, é a raiz da árvore e está no topo da hierarquia.

A raiz está conectada a outros nós, no nível 1, que, por sua vez, estão conectados a outros nós, no nível 2, e assim por diante. Os nós que estão no fim da árvore são chamados de **folhas** <, conforme magem a seguir:



1 - Estrutura de dados árvore.

As conexões entre os nós de uma árvore seguem uma nomenclatura genealógica. Um nó, em dado nível, está conectado a seus filhos (no nível abaixo) e a seu pai (no nível acima). A raiz da árvore, que está no nível 0, possui filhos, mas não possui pai. Os nós que estão no final da árvore (os mais distantes da raiz) são chamados de **nós folhas** (*leafs*).

Árvores podem ser desenhadas de muitas formas, porém, na convenção em computação, convém desenhá-las com a raiz no topo, bem diferente das árvores tradicionais que conhecemos na natureza.

Vejamos algumas das principais nomenclaturas envolvidas em árvores:

Nós (*nodes*)

Elemento que contém a informação.

Arcos ou arestas (*edges*)

Elementos que ligam dois nós na árvore.

Pai (*parents*)

Elemento que atua como o nó superior de um arco.

Filho (*child*)

Elemento que atua como nó inferior de um arco.

Raiz (*root*

Elemento que atua como nó que fica no topo da árvore – não tem um nó pai.

Folhas (*leafs*)

Elementos que atuam como nós das extremidades inferiores – não têm nós filhos.

Grau (*degree*)

Elemento que representa o número de subárvores de um nó.

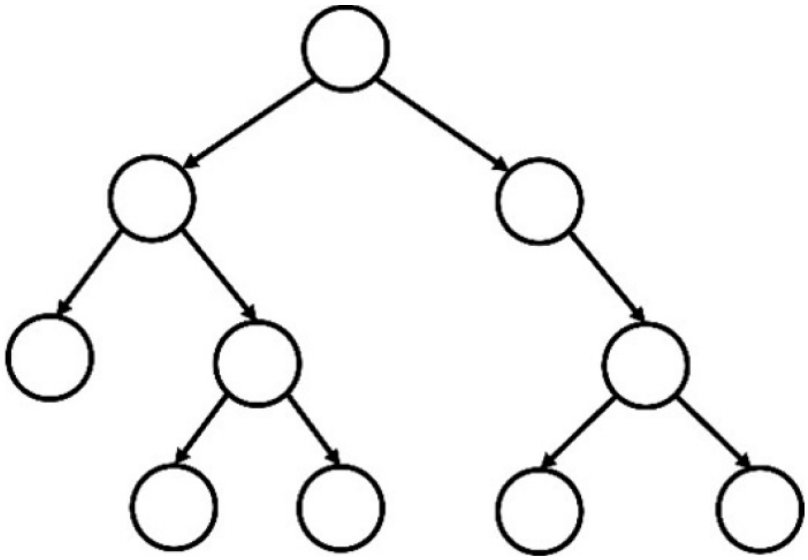
Árvores binárias

O que são árvores binárias?

Confira agora a definição formal de árvores binárias e como elas são representadas em memória, além da diferença entre árvore binária e árvore.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Árvores binárias são árvores nas quais cada nó pode ter, no máximo, dois filhos. Observe a imagem:



2 - Árvore binária.

Uma árvore binária pode ser definida de forma recursiva, de acordo com o seguinte raciocínio:

1

A raiz da árvore possui dois filhos, um à direita e outro à esquerda, que, por sua vez, são raízes de duas subárvores.

→

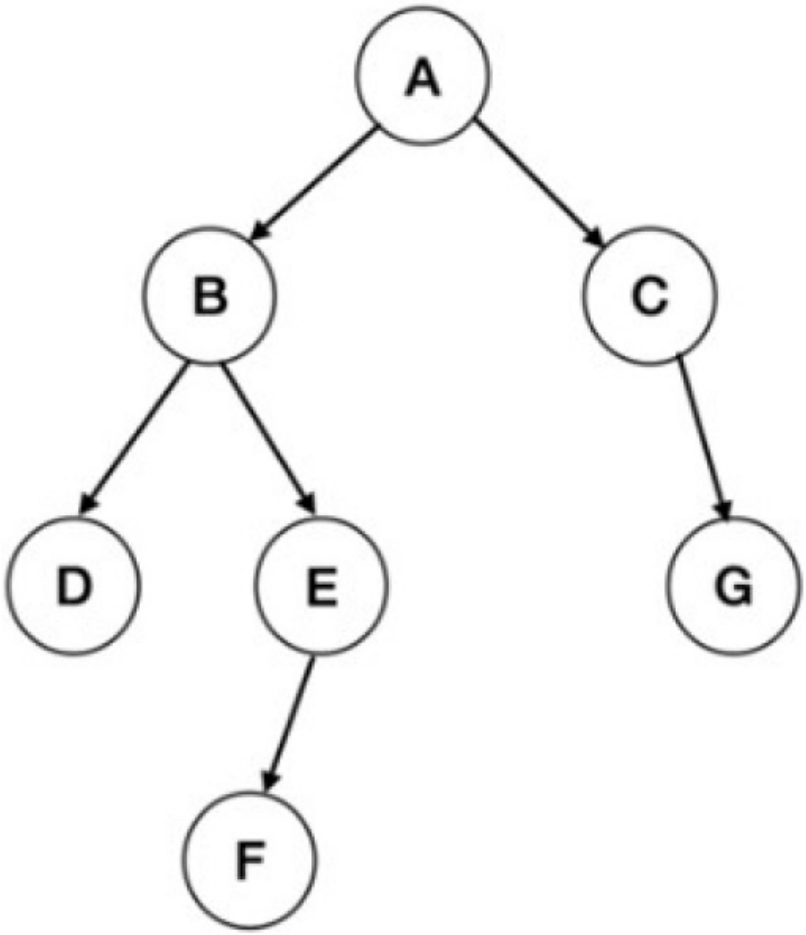
2

Cada uma dessas subárvores possui uma subárvore esquerda e uma subárvore direita, seguindo esse mesmo raciocínio.

Falando formalmente, uma árvore binária T é um conjunto vazio ou é composta pelos seguintes elementos: uma entidade n, chamada nó raiz, e pelas entidades Te e Td, respectivamente, as subárvores esquerda e direita de T, que também são árvores binárias.

Observe que, como Te e Td são árvores binárias, a definição é recursiva, isto é, Te possui um nó raiz ou é uma subárvore vazia. O mesmo vale para Td.

Usualmente, representamos uma árvore por meio da representação gráfica em que se destacam a raiz da árvore T e sua subárvore esquerda e direita. Confira a representação:

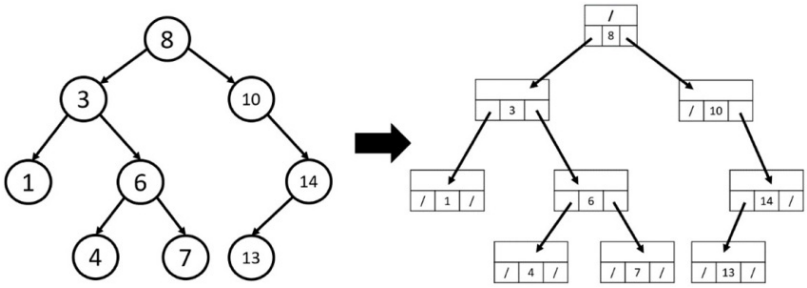


3 - Representação gráfica da árvore binária.

Na imagem 3, o nó " A " é a raiz da árvore T, Te é composta pelos nós: " B ", " D ", " E " e " F "; Td é composta pelos nós: " C " e " G ". " B " e " C " são as raízes de Td e Te, respectivamente.

Observe que existe na árvore binária, por definição, distinção entre a subárvore esquerda e direita. Por isso, na representação gráfica, sempre fica evidente a posição esquerda ou direita do nó subordinado à raiz. Na imagem 3, o nó " G " é o filho direito de " C ", e não existe, por exemplo, filho esquerdo de " C ".

A forma mais comum de representar uma árvore em memória é utilizando alocação dinâmica. Na verdade, não representamos a árvore como um todo, mas uma referência para sua raiz, que guarda a chave (dado) e uma referência para a raiz das subárvores esquerda e direita. Aqui, representaremos nossa árvore usando a linguagem Python.



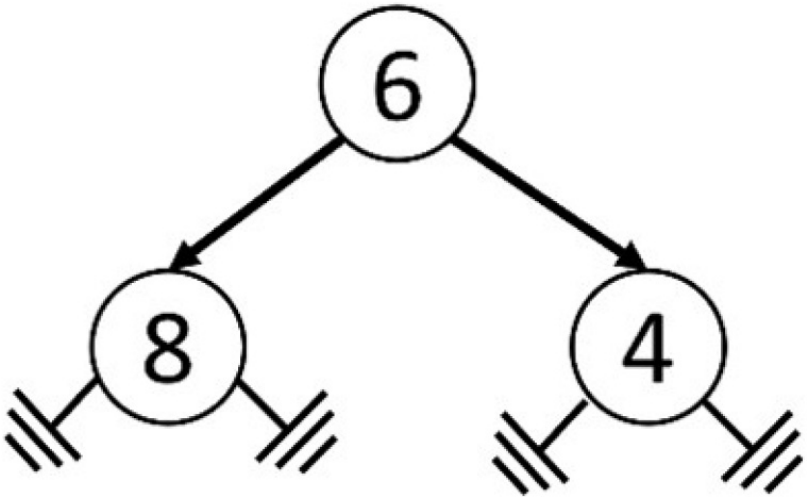
4 - Representação de uma árvore em memória.

Os nós de uma árvore binária possuem um valor (intitulado chave) e dois apontadores, um para o filho da esquerda e outro para o filho da direita. Esses apontadores representam as ligações (arestas) de uma árvore. Veja:

Python

Algoritmo 1 - Implementação em Python da criação de nós de uma árvore binária e seus respectivos filhos, à esquerda e à direita.

A imagem a seguir ilustra a árvore binária implementada no código do algoritmo 1. Note que o nó raíz (com o valor 6), possui dois filhos, um à esquerda (com o valor 8) e outro à direita (com o valor 4).



5 - Abstração da árvore binária implementada no algoritmo 1.

Os nós, cujos valores são 8 e 4, não possuem filhos, seus apontadores à esquerda e à direita são **None**, ou seja, não apontam para qualquer outro

nó.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Assinale a alternativa correta acerca das estruturas de dados árvores e árvores binárias.

A Árvores binárias permitem a inserção de mais de dois filhos por nó.

B Ao acessar uma árvore, deve-se acessar pela referência a sua raiz.

C Árvores são consideradas estruturas de dados linear.

D Os nós de uma árvore que possuem grau zero são chamados, exclusivamente, de folhas raiz.

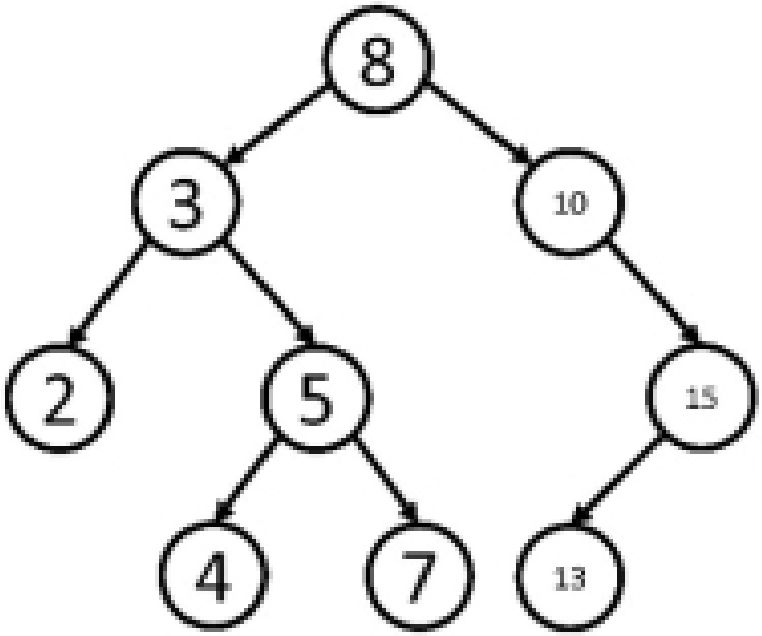
E Os nós de uma árvore que estão no nível 1 são chamados de raiz.

Parabéns! A alternativa B está correta.

Representa-se uma árvore em memória por meio de alocação dinâmica. A árvore não é representada como um todo, mas uma referência para sua raiz, que guarda a chave (dado) e uma referência para a raiz das subárvores esquerda e direita.

Questão 2

Considerando que a seguinte árvore é binária, assinale a alternativa correta.



A A árvore acima possui raiz de valor 3.

B É possível inserir mais um filho à esquerda, no nó de valor 5.

C A quantidade de folhas da árvore é 4.

D A quantidade de nós da árvore é de $n - 1$, sem considerar o nó raiz.

E O nó 5 é raiz da subárvore esquerda do nó 3.

Parabéns! A alternativa C está correta.

A quantidade de folhas da árvore é 4, ou seja, são aqueles nós que possuem grau zero.

2 - Operações em árvores binárias

Ao final deste módulo, você será capaz de identificar as principais operações em árvores binárias de busca implementadas em linguagem Python.

Árvores binárias de busca e sua operacionalização

Principais operações em árvores binárias

Confira a definição de BST (árvore binária de busca) e a implementação em Python dos algoritmos de busca.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

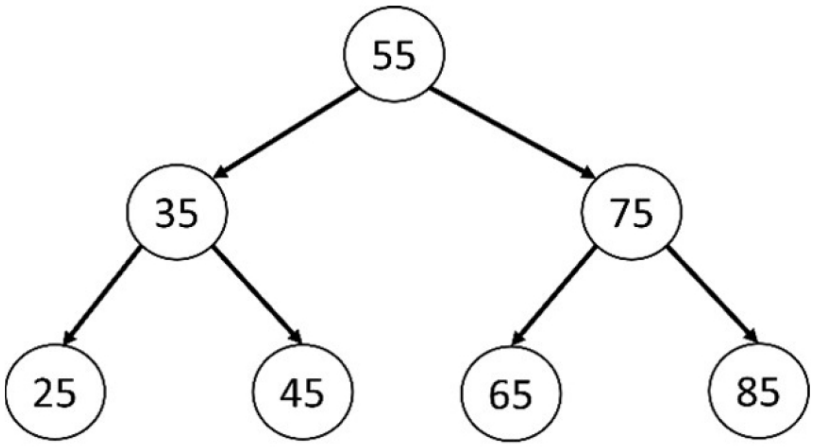
O problema da busca, inserção e remoção é um dos principais objetivos do estudo de estruturas de dados. Utilizar a estrutura de árvores para aplicar regras ao acessar seus dados pode facilitar, computacionalmente, a busca, a inserção e a remoção de dados.

As árvores binárias de busca (do inglês *binary search trees* – BST's) são árvores de nós organizados de acordo com certas propriedades. A partir desse ponto, considere que as árvores não permitem elementos repetidos. Observe a seguinte definição:

> **Definição 1: Seja x um nó em uma árvore binária de busca. Se y é um nó na subárvore esquerda de x, então y. chave $\leq x$. chave. Se y é um nó na subárvore direita de x, então, y. chave $\geq x$. chave.**

Ou seja, árvores binárias de busca são árvores que obedecem às seguintes propriedades:

- Dado um nó qualquer da árvore binária, todos os nós à sua esquerda são menores ou iguais a ele.
- Dado um nó qualquer da árvore binária, todos os nós à direita dele são maiores ou iguais a ele.



6 - Exemplo de árvore binária de busca.

No algoritmo 2, é possível verificar a implementação da árvore binária da imagem 6. Observe:

Python

Algoritmo 2 - Implementação da árvore binária de busca da imagem 6.

Para imprimir a árvore implementada, utilizaremos uma estratégia recursiva para percorrer os nós da árvore e imprimir cada nó. Observe o algoritmo em Python:

Python

Algoritmo 3 - Implementação da impressão da árvore binária de busca.

Para imprimir a árvore implementada, utilizaremos uma estratégia recursiva para percorrer os nós. Se a raiz não for nula (caso base, linha 4), então o algoritmo incrementa um pequeno controlador de níveis da árvore (linha 7), e fazemos duas chamadas recursivas, uma para a subárvore da esquerda e outra para a subárvore da direita. As linhas entre 12 e 15 servem para configurar os nós percorridos em formato de árvore binária.

Busca de nós em BST

O algoritmo de busca decorre diretamente da definição. Considere x a chave que desejamos localizar. Compara-se x com a raiz; se x está nessa raiz, o algoritmo de busca para. Caso contrário, se x é menor que a raiz, executa-se o algoritmo recursivamente na subárvore esquerda, caso contrário, na subárvore direita. Observe:

Python

Algoritmo 4 - Implementação da Busca de um nó em uma árvore binária de busca.

Na busca, existem algumas situações especiais. A primeira ocorre quando a chave buscada está na raiz. Nesse caso, o nó que contém a chave não tem pai, por isso, o algoritmo de busca deverá retornar NULO na referência para o pai. Outra situação especial ocorre quando é realizada uma busca em uma árvore vazia. Nesse cenário, o algoritmo deverá retornar:

NULO

Para o nó que contém a chave.

NULO

Para referência do nó que é pai do nó que contém a chave buscada.

FALSE

Para referência ao booleano.

O algoritmo de busca é utilizado nas operações de inserção e de remoção de nós nas árvores binárias de busca.

Sabemos que a análise de complexidade é baseada na identificação do pior caso e na análise do número de operações elementares que o resolva. O pior caso é, sem dúvida, não encontrar a chave buscada. Nesse cenário, o algoritmo realizará uma comparação por nível até encontrar um nó que não possua o filho no qual a chave buscada poderia estar.

Pela definição de árvore binária de busca, não há restrição para a altura da árvore, sendo uma árvore com topologia zigue-zague, que são árvores nas quais cada nó só possui um filho, podendo ser uma árvore binária de busca. Entretanto, árvores desse teor possuem altura proporcional à n. Assim, a complexidade da busca em uma árvore binária de busca é $O(n)$.

Inserção e remoção em BST

Inserção e remoção em árvores binárias de busca

Confira agora a implementação em Python dos algoritmos de inserção e a remoção em árvores binárias de busca.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Inserção de nós em BST

A inserção de uma nova chave em uma árvore binária de busca ocorre sempre em um novo nó, posicionado como uma nova folha da árvore. Isso decorre do fato de que a posição do nó que contém a nova chave é determinada pela sua busca na árvore.

Já vimos que estruturas de dados, nas quais realizamos busca, inserção e remoção, não permitem chave duplicada. Assim, a busca pela chave que desejamos inserir na árvore deverá, obrigatoriamente, falhar e retornar como resultado o nó que será pai do novo nó inserido na árvore.

Python

Algoritmo 5 - Inserção de nós em uma árvore binária de busca (linha 1 até linha 11).

No algoritmo apresentado, temos a implementação em Python da inserção de nós em uma árvore binária de busca. Observamos ainda que, a partir da linha 13, acontece a inserção de nós para formar a árvore da imagem 6:

O algoritmo primeiro (linha 14) insere a raiz “55”. Logo após, insere seus filhos “35” (esquerda) e “75” (direita).

Logo depois, nas linhas 17 e 18, insere os filhos de 35, no caso, o “25” (esquerda) e o “45” (direita).

A seguir, nas linhas 19 e 20, insere os filhos de 75, no caso, o “65” (esquerda) e o “85” (direita).

Por fim, executa o algoritmo 3 para imprimir a árvore binária final.

A principal operação para realização da inserção de um novo nó é a busca. Ela determina a posição e se é possível ou não realizar a inserção. Após a busca, as operações seguintes são todas realizadas em $O(1)$. Sendo assim, a complexidade da inserção é a complexidade da busca que é $O(n)$.

Remoção de nós em BST

O processo de remover o nó de uma árvore binária deve obedecer a algumas regras. Confira:

1

O nó a ser deletado é uma folha: a remoção é simples. Basta buscar pelo nó e removê-lo.

2

O nó a ser excluído tem apenas um filho: copie o filho para o nó e o exclua.

3

O nó a ser excluído tem dois filhos: encontre o sucessor em ordem do nó. Copie o conteúdo do sucessor em ordem para o nó e o exclua.

Confira agora o algoritmo de deleção de nós de uma árvore binária:

Python

No algoritmo 6, podemos ver os seguintes passos no nosso algoritmo de deleção:

No algoritmo 6, podemos ver os seguintes passos no nosso algoritmo de deleção:

Linhas 2 e 3

Neste passo, se a raiz for None, retorne à raiz (caso base).

Linhas 4 e 5

Neste passo, se a chave for menor que o valor da raiz, defina a instrução raiz.esquerda =

DeleteBST (raiz.esquerda, chave).

Linhas 6 e 7

Neste passo, se a chave for maior que o valor da raiz, defina a instrução raiz.direita = DeleteBST (raiz.direita, chave).

Caso contrário

Neste passo:

- Verifique se a raiz for um nó folha, então retorne nulo.
- (Linhas 9 e 12): caso contrário, se tiver apenas o filho esquerdo, retorne-o;
- (Linhas 13 e 16): caso contrário, se tiver apenas o filho direito, retorne-o;
- (Linha 17 e 19): caso contrário, defina o valor de raiz como seu sucessor em ordem e repita para excluir o nó com o valor do sucessor em ordem. Para esse processo, temos a função **ValorNo** para auxiliar.

Ao final

Neste passo, retorne à raiz.

Após os passos, veja o algoritmo:

Python

Algoritmo 7 - Função de auxílio para deleção de nós.

No estudo da complexidade da remoção, vimos que a remoção é dependente da busca e que ela tem complexidade da $O(n)$. No caso de remoção de uma folha, ela depende somente da busca. Em seguida, realizamos operações elementares para remover o nó. Portanto, a complexidade do caso 1 é $O(n)$.

No pior caso, vamos remover um nó interno do penúltimo nível. Portanto, o custo computacional da busca é $O(n)$, uma vez que vamos percorrer $n-1$ nós para encontrar o nó que será removido. As operações de reapontamento são elementares. Portanto, a complexidade também é $O(n)$.

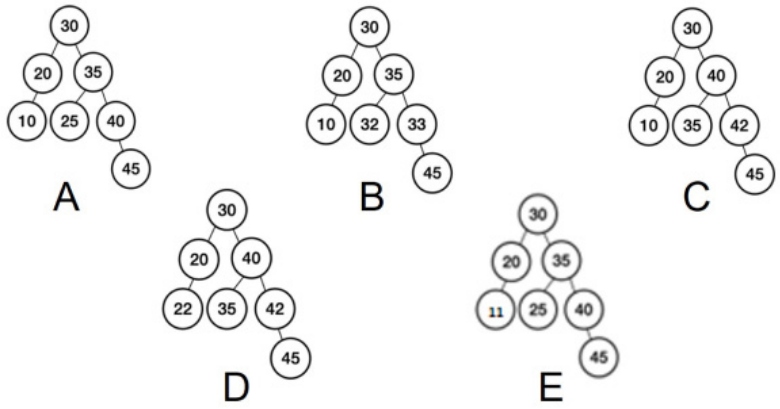
No caso de remover um nó com dois descendentes, novamente, a estrutura é próxima a uma árvore zigue-zague. Haverá somente um nó com dois descendentes e ele está no nível k. A primeira busca para encontrar o nó que desejamos remover executa k comparações, uma vez que esse nó está no nível k. Em seguida, procuraremos o substituto do nó no ramo zigue-zague da estrutura, percorrendo-a até o nível $n-1$. Portanto, o custo da busca e dos reapontamentos é de $n-1$ comparações. Logo, $O(n)$.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Analise as imagens a seguir e assinale a letra da imagem que contém uma árvore binária de busca.



A Imagem A

B Imagem B

C Imagem C

D Imagem D

E Imagem E

Parabéns! A alternativa C está correta.

Em uma árvore binária de busca, os nós contidos na subárvore esquerda devem ser menores que a raiz e maiores na direita. Isso vale para todos os nós.

Questão 2

Tomando como base os conceitos de busca, inserção e remoção em estruturas de dados, é correto afirmar que

A ao se deletar um nó folha, toda a estrutura da árvore será modificada, inclusive a raiz da árvore.

B o objetivo principal da estrutura de dados árvore binária de busca é resolver o problema da busca,

inserção e remoção sem necessariamente ser eficiente.

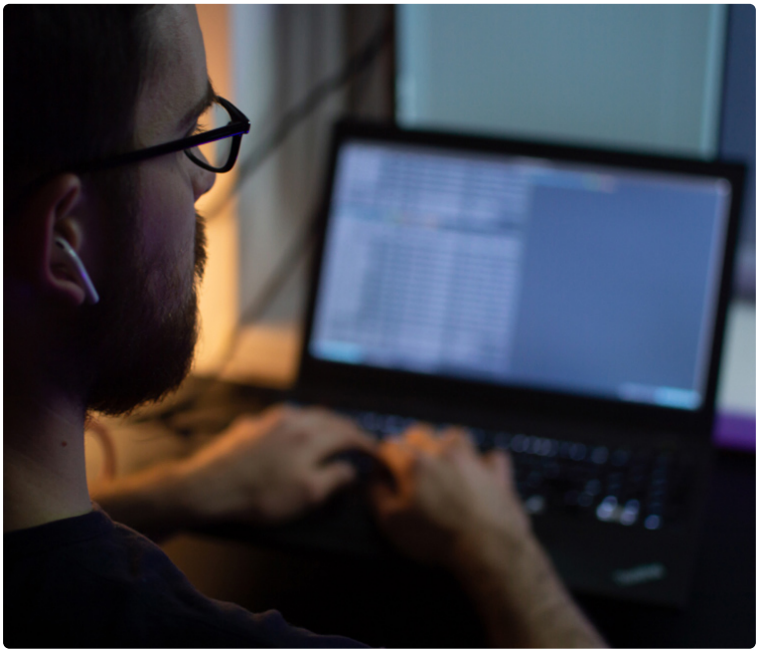
C no processo de inserção de nós, à esquerda, pode entrar nós que são maiores que a sua raiz considerada.

D em todas as estruturas de dados nas quais se realiza busca, inserção e remoção não são admitidas duplicidade de chaves. Isso também inclui as árvores binárias de busca.

E dado um nó qualquer da árvore binária, todos os nós à sua direita são menores ou iguais a ele.

Parabéns! A alternativa D está correta.

Em uma árvore binária de busca, por definição, não são admitidas chaves em duplicidade.



3 - Percursos em árvores

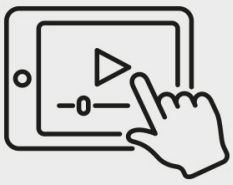
Ao final deste módulo, você será capaz de identificar os principais algoritmos de percursos em árvores binárias implementados em linguagem Python.

Percursos em árvores: formalização

Principais formas de percurso de uma árvore

Confira agora o algoritmo dos percursos em pré, em e pós-ordem.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Um percurso é a visita sistemática aos elementos de uma estrutura de dados. Vale destacar que visitar significa realizar uma operação e não acessar o conteúdo armazenado no elemento da estrutura de dados. Ou seja, especificamente para árvores, acessar um nó da árvore não configura uma visita. A visita é uma operação que tem sentido na semântica da aplicação desejada. A visita mais simples que podemos definir é a impressão do rótulo da chave armazenada no nó visitado.

Observe que o conceito de árvore é recursivo, isto é, a estrutura foi definida em termos recursivos. Por tal razão, as definições dos percursos também são recursivas. Existem três maneiras clássicas de percursos em árvores:

Pré-ordem

Visitar, primeiro, a raiz, depois a subárvore esquerda e, por último, a subárvore direita.

Em ordem

Visitar, primeiro, a subárvore esquerda, depois a raiz e, por último, a subárvore direita.

Pós-ordem

Visitar, primeiro, a subárvore esquerda, depois a subárvore direita e, por último, a raiz.

Percurso pré-ordem

A partir da raiz *r* da árvore *T*, percorre-se a árvore da seguinte forma:

1

Visita-se a raiz.

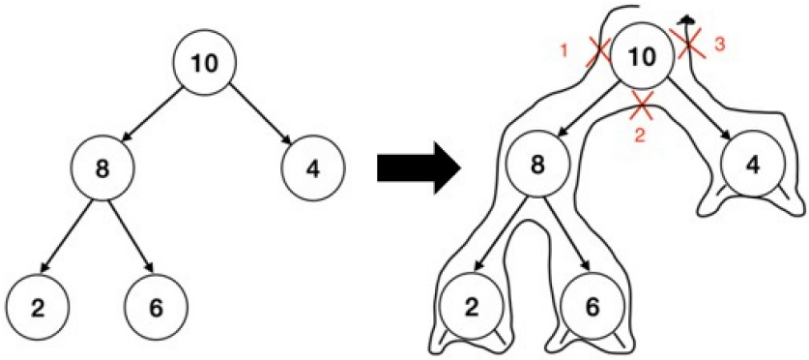
2

Percorre-se a subárvore
esquerda de T, em pré-ordem.

3

Percorre-se a subárvore direita
de T, em pré-ordem.

A imagem a seguir apesenta uma árvore e o resultado da visita pré-ordem>:



7 - Uma árvore e o resultado da visita em pré-ordem.

Considerando como a operação de visita a impressão do rótulo contido no nó visitado, veremos o resultado do percurso em pré-ordem da árvore na imagem 7. Partindo-se da raiz, nó de rótulo “10”, temos que a primeira operação é a visita do nó, isto é, a impressão do seu rótulo.

Impressão: **10**

Em seguida, visita-se recursivamente a subárvore esquerda, cuja raiz é “8”, e a primeira operação é a visita do nó com rótulo “8”.

Impressão: **10, 8**

Após a visita do nó “8”, percorre-se recursivamente a subárvore esquerda do nó “8”, cuja raiz é o nó com rótulo “2”. A primeira operação desse percurso é a visita, isto é, o rótulo “2” é impresso.

Impressão: **10, 8, 2**

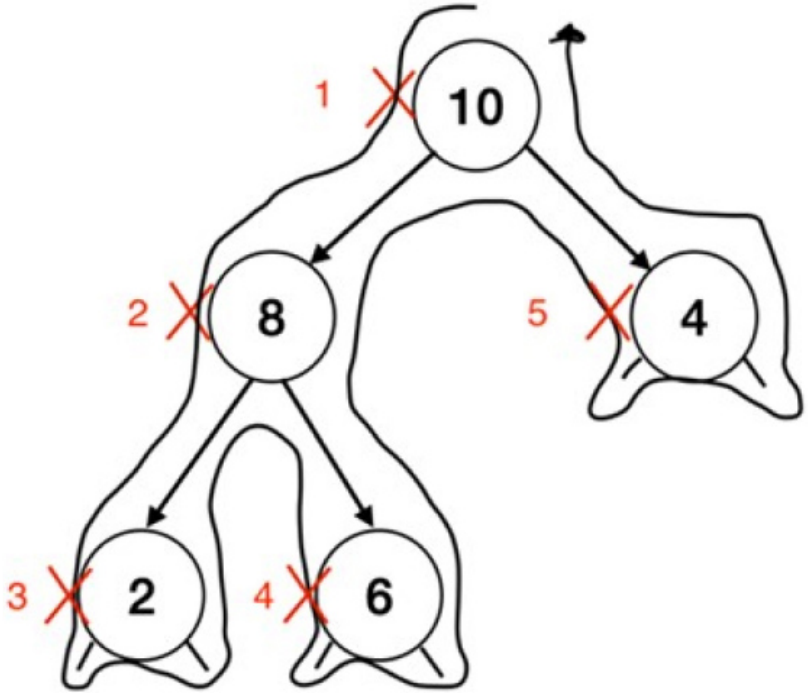
O nó “2” é folha, isto é, não possui subárvores, assim, o percurso da subárvore esquerda do nó “8” é concluído. O próximo passo é percorrer a subárvore direita do nó “8”, cuja raiz é o nó “6”. O primeiro passo é a impressão (visita).

Impressão: **10, 8, 2, 6**

Observe que o nó “6” é folha, o que encerra o percurso da subárvore de raiz “6”. Assim, retornamos ao seu pai, o nó “8”, que é raiz da subárvore, e que já teve seu percurso completo. O pai de “8” é o nó “10”, que já foi visitado e teve sua subárvore esquerda visitada. O próximo passo é visitar sua subárvores direita em pré-ordem. A raiz da subárvore direita é “4”, que é folha, sendo assim, o percurso pré-ordem da árvore é:

Impressão: **10, 8, 2, 6, 4**

Observe que, para cada nó, acessamos seu conteúdo três vezes, porém, somente uma dessas corresponde à visita. No caso do percurso em pré-ordem, a primeira vez. Observe também que, imprimindo o conteúdo do nó (operação de visita), sempre no acesso “1”, teremos o percurso em pré-ordem. Confira a seguir:



8 - Sequência de visitas no acesso “1”.

Observe, na imagem 8, que a sequência de acesso é 10, 8, 2, 6, 4, exatamente o percurso em pré-ordem.

Percurso em ordem

A partir da raiz *r* da árvore *T*, percorre-se a árvore da seguinte forma:

1. Percorre-se a subárvore esquerda de *T* em ordem simétrica.
2. Visita-se a raiz.
3. Percorre-se a subárvore direita de *T* em ordem simétrica.

Aplicando a definição na árvore da imagem 7, temos a seguinte sequência de visitas, que percorre-se a subárvore direita de *T*, em ordem simétrica, pode ser obtida, passo a passo, a partir do nó “10”, raiz de *T*.

Inicialmente, percorremos a subárvore esquerda de “10” em ordem simétrica, em que o nó “8” é a raiz dessa árvore. Mais uma vez, o primeiro passo é percorrer a subárvore esquerda do nó “8” em ordem simétrica.

> **Mais uma vez, o primeiro passo é percorrer a subárvore esquerda do nó “8” em ordem simétrica.**

O nó “2” é a raiz dessa árvore; “2” é folha, assim, não possui subárvore esquerda nem direita. Logo, “2” é visitado após a tentativa de percurso de sua subárvore esquerda.

Impressão: **2**

Após a visita do nó “2”, seria feita a visita do ramo direito de “2”, que não existe, concluindo, assim, o percurso em ordem simétrica da subárvore de raiz “2”. O próximo passo é a visita do nó “8”.

Impressão: **2, 8**

Após a visita de “8”, percorre-se a subárvore direita de “8” em ordem simétrica. A raiz dessa subárvore é “6”, que é folha; logo, ocorre a visita de “6”, resultando na sequência de impressão.

Impressão: **2, 8, 6**

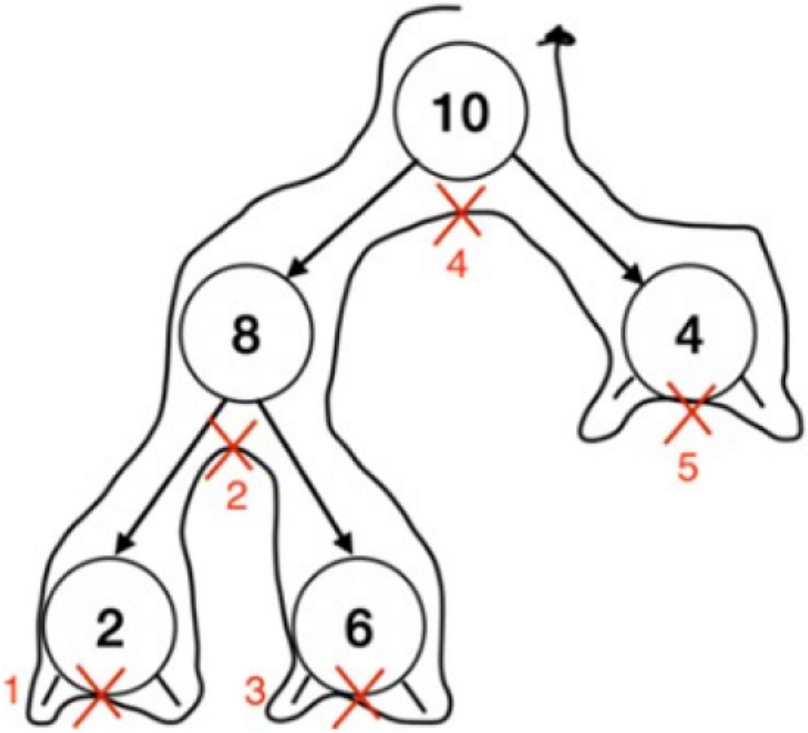
A impressão do rótulo da folha “6” e a tentativa de percurso da subárvore direita do nó “6” termina o percurso da subárvore de raiz “8”. Voltando, na reclusão, ao seu pai, já percorremos a subárvore esquerda do nó “10”. O próximo passo é a vista do nó, resultando na impressão.

Impressão: **2, 8, 6, 10**

O próximo passo é o percurso da subárvore direita de “10”, que contém a folha “4”, resultando na impressão.

Impressão: **2, 8, 6, 10, 4**

Concluindo o percurso em ordem simétrica de *T*, temos o seguinte:



9 - Percurso em ordem simétrica de *T*: 2, 8, 6, 10, 4.

Percurso pós-ordem

A partir da raiz *r* da árvore *T*, percorre-se a árvore da seguinte forma:

1

Percorre-se a subárvore esquerda de *T* em pós-ordem.

2

Percorre-se a subárvore direita de *T* em pós-ordem.

3

Visita-se a raiz.

Aplicando-se a definição na árvore da imagem 7, temos a sequência de visitas apresentada a seguir. Partindo-se da raiz “10” de T, percorre-se a subárvore esquerda de “10” em pós-ordem. O nó “8” é a raiz desta subárvore. Para esta subárvore, o primeiro passo é visitar seu ramo esquerdo em pós-ordem. O nó “2” é a raiz deste ramo, que também é folha, assim, “2” não tem subárvore esquerda e direita, sendo o primeiro nó visitado.

Impressão: **2**

Retorna-se, então, ao seu pai, o nó “8”, a subárvore esquerda de “8” já foi visitada, o próximo passo é visitar sua subárvore direita. O nó “6” é raiz desta subárvore e folha, sendo assim, o nó “6” é visitado.

Impressão: **2, 6**

Com o término do percurso na subárvore de raiz “6”, retorna-se ao pai de “6”, que é o nó “8”. As subárvores esquerda e direita de “8” já foram percorridas, assim, o próximo passo é visitar “8”.

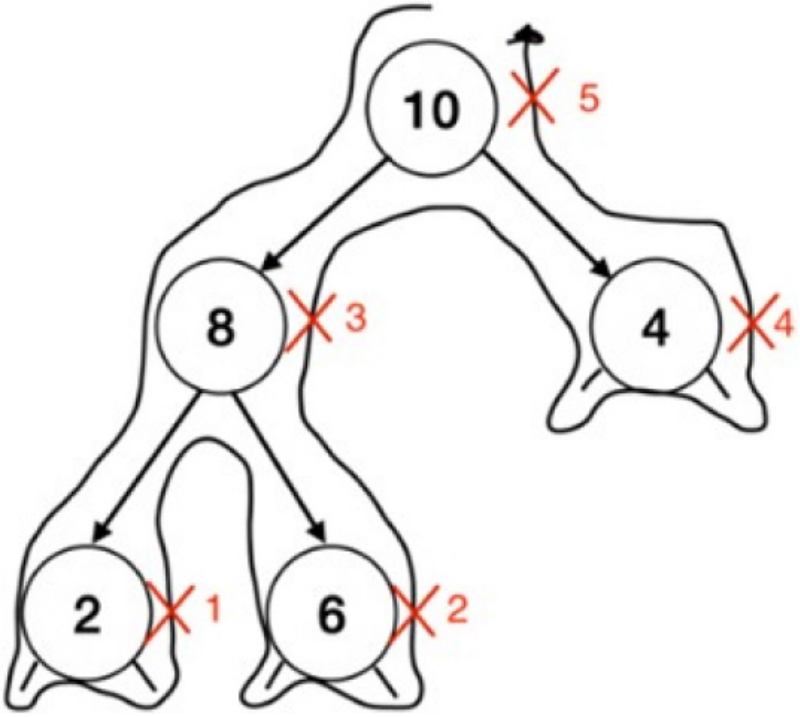
Impressão: **2, 6, 8**

A visita ao nó “8” termina o percurso da subárvores de raiz “8”. Ao retornar ao pai do nó “8”, o nó “10”, devemos percorrer em pós-ordem a subárvore direita do nó “10”. O nó “4” é a raiz da subárvore é folha, sendo assim, visita-se o nó “4”.

Impressão: **2, 6, 8, 4**

Com o término do percurso da subárvore de raiz “4”, retorna-se ao pai de “4”, que é o nó “10”. Já foram percorridas as subárvores esquerda e direita de “10”, assim, o próximo passo é visitar o nó “10”, o que finaliza o percurso.

Impressão: **2, 6, 8, 4, 10**, que é o percurso em pós-ordem.



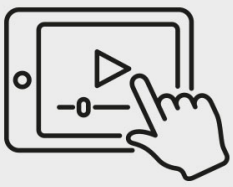
10 - Sequência resultado do percurso pós-ordem.

Algoritmos em Python para percursos em árvores

Percurso de árvores em Python

Confira agora a implementação em Python dos percursos pré, em e pós-ordem.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



Os algoritmos recursivos para os percursos em pré-ordem, ordem simétrica e pós-ordem são consequências diretas das definições dos percursos.

Algoritmo de percurso pré-ordem

Vamos conferir agora o algoritmo do percurso em pré-ordem. Primeiro, apresentaremos um pseudocódigo que utiliza estratégia recursiva, veja:

Terminal

Pseudocódigo 1 - Percurso em pré-ordem.

Em Python, também implementamos o algoritmo de pré-ordem utilizando a estratégia recursiva, ou seja:

Python

Algoritmo 8 - Função Python de percurso em pré-ordem.

Para realizar o percurso em pré-ordem, são necessários três acessos ao nó. No caso da pré-ordem, no primeiro, executamos a visita; no segundo, chamamos recursivamente o algoritmo para a subárvore esquerda e, no terceiro, ocorre a chamada do percurso em pré-ordem do ramo direito. Assim, a complexidade computacional do percurso em pré-ordem é $O(n)$.

Algoritmo de percurso em ordem

O algoritmo do percurso em ordem simétrica é semelhante ao pré-ordem. Modificamos somente o momento da visita, resultando no pseudocódigo 2, que utiliza a seguinte estratégia recursiva:

Terminal

Pseudocódigo 2 - Percurso em ordem simétrica.

Em Python, também implementamos o pseudocódigo utilizando a estratégia recursiva, ou seja:

Python

Algoritmo 9 - Função Python de percurso em ordem.

A análise de complexidade é análoga à realizada no algoritmo do percurso em pré-ordem. Observe que a única diferença é a ordem das visitas. Sendo assim, a complexidade computacional do algoritmo para percurso em ordem simétrica é *O(n)*.

Algoritmo de percurso pós-ordem

Finalmente, temos o algoritmo para o percurso em pós-ordem (algoritmo 10), que é resultado direto da definição, como o da pré-ordem e o da ordem simétrica.

Terminal

Pseudocódigo 3 - Percurso em pós-ordem.

Em Python, também implementamos o pseudocódigo utilizando a estratégia recursiva, ou seja:

Python

Algoritmo 10 - Função Python de percurso pós-ordem.

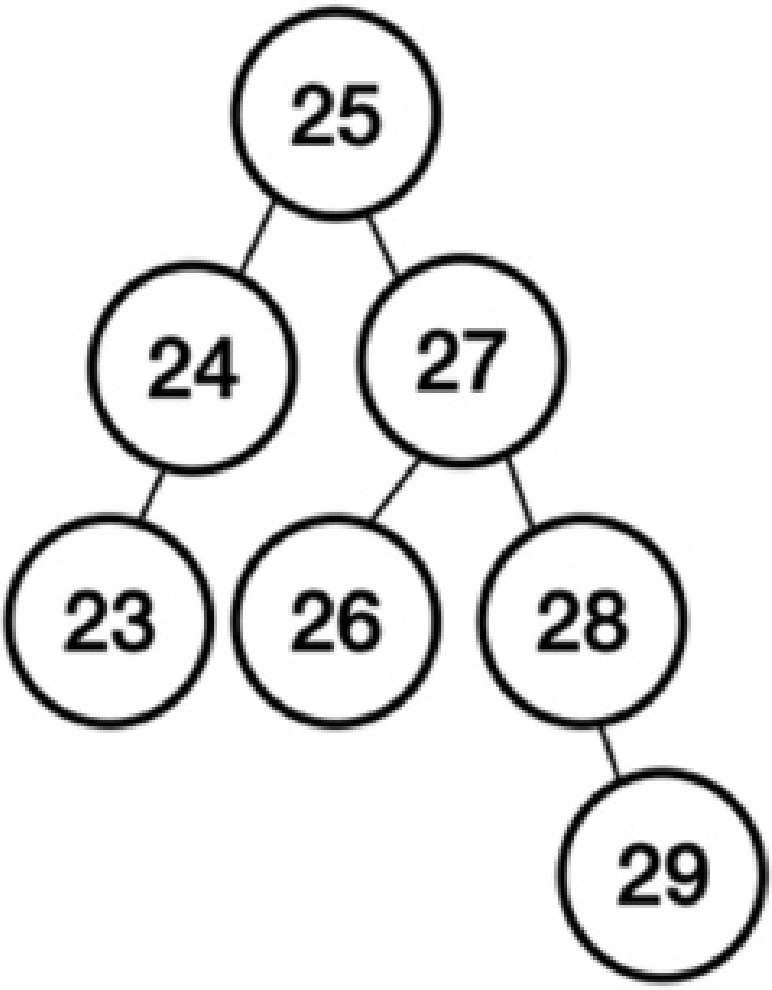
A análise da complexidade é totalmente análoga à análise feita para pré-ordem e ordem simétrica, o que faz com que o algoritmo tenha complexidade *O(n)*.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Dada a árvore a seguir, identifique o percurso em ordem simétrica:



A 23, 24, 25, 26, 27, 28, 29

B 23, 24, 26, 29, 28, 27, 25

C 25, 24, 27, 23, 26, 29, 30

D 25, 24, 23, 27, 26, 28, 29

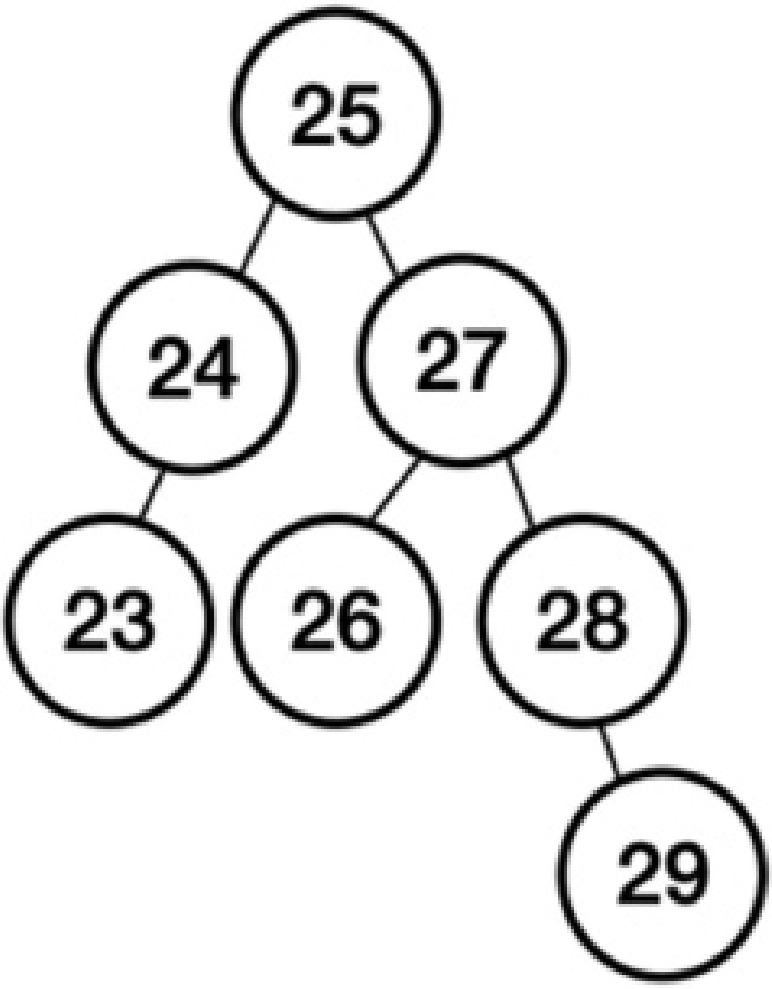
E 29, 28, 27, 26, 25, 24, 23

Parabéns! A alternativa B está correta.

O percurso é definido pela recursão: percorrer recursivamente à esquerda, visitar a raiz e percorrer recursivamente à direita da raiz considerada.

Questão 2

Dada a árvore a seguir, identifique o percurso em pré-ordem:



A 23, 24, 25, 26, 27, 28, 29

B 23, 24, 26, 29, 28, 27, 25

C 25, 24, 27, 23, 26, 29, 30

D 25, 24, 23, 27, 26, 28, 29

E 29, 23, 26, 28, 24, 27, 25

Parabéns! A alternativa D está correta.

O percurso é definido pela recursão, visita à raiz e percorrer recursivamente esquerda e direita da raiz considerada.

4 - Aplicação de percursos em árvores

Ao final deste módulo, você será capaz de descrever exemplo de aplicação dos percursos em árvores sobre o problema da avaliação de expressões aritméticas.

Principais aplicações em árvores binárias de busca

Aplicações de árvores binárias: expressões aritméticas

Confira agora a razão das ambiguidades e entenda como as árvores binárias resolvem esse problema.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.

Uma árvore binária é uma estrutura de dados útil quando precisamos tomar decisões bidirecionais em cada ponto de determinado processo. Muitas aplicações podem ser modeladas por essa estrutura, vejamos a seguir:

Aplicação 1

Neste cenário, suponha que precisamos encontrar todas as repetições em uma lista de números. Uma maneira de fazer isso é comparar cada número com todos que o precedem. Entretanto, isso envolve muitas comparações, que podem ser reduzidas no uso de árvore binária.

Aplicação 2

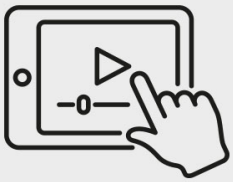
Neste cenário, em uma aplicação de árvores binárias, temos uma expressão aritmética contendo operandos e operadores binários por uma árvore estritamente binária. Na raiz dessa árvore, conterá um operador que deve ser aplicado aos resultados das expressões representadas pelas subárvores esquerda e direita.

Árvores de expressões aritméticas

Traduzindo expressões aritméticas em árvores Python

Confira agora a aplicação dos algoritmos de percursos em árvores binárias para resolver as ambiguidades da grafia de uma expressão aritmética.

Para assistir a um vídeo sobre o assunto, acesse a versão online deste conteúdo.



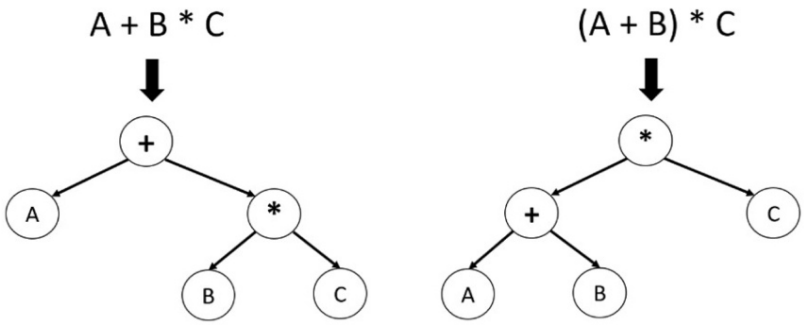
Uma árvore é uma forma natural para representar a estrutura de uma expressão aritmética. Ao contrário de outras notações, a árvore pode representar a computação de forma não ambígua.

Exemplo

A expressão infixa 1 + 2 * 3 é ambígua, a menos que saibamos, de forma antecipada, que a multiplicação é feita antes da adição.

Modelando as expressões aritméticas em árvores, um nó de uma árvore representa um operador, um nó que não é folha, enquanto um nó representando um operando é uma folha.

Vamos considerar a expressão A + B * C. Os operadores + e * ainda aparecem entre os operandos, mas há um problema. Sobre quais operando eles estão atuando? Primeiro, aplicamos o + sobre A e B ou o * sobre B e C? A expressão parece ambígua. Veja:



11 - Exemplo de representação de expressões aritméticas em árvores binárias.

Vamos interpretar a expressão A + B * C usando a suas precedências. B e C são multiplicados primeiro, em seguida, A é adicionado ao resultado. (A + B) * C forçaria que a adição de A e B fosse realizada primeiro, antes da multiplicação. Na expressão A + B + C, pela associatividade da precedência, o + à esquerda seria feito primeiro; adicionamos A a B e, em seguida, o resultado a C.

Embora tudo isso possa ser óbvio, lembre-se de que os sistemas computacionais precisam saber exatamente quais operadores executar, e em que ordem. Uma maneira de escrever uma expressão que garanta que não haverá confusão alguma com respeito à ordem em que as operações são executadas é a criação de uma expressão totalmente **parentizada**.

Esse tipo de expressão usa **um par de parênteses para cada operador**. Os parênteses ditam a ordem em que as operações são executadas e não há ambiguidade. Também não há necessidade de lembrar de regras de precedência.

Expressões prefixas, infixas e pós-fixas

Considere a expressão aritmética infixa A + B. O que acontece se movemos o operador para antes dos dois operandos?

A expressão ficaria + A B. Da mesma forma, poderíamos mover o operador para o fim. Nós obteríamos A B +. Essas expressões parecem estranhas.

A expressão A + B * C + D pode ser reescrita como ((A + (B * C)) + D), para mostrar que a multiplicação acontece primeiro, seguida da adição, mais à esquerda. A + B + C + D pode ser escrito como (((A + B) + C) + D), já que operações de adição são associadas da esquerda para a direita.

Essas mudanças na posição do operador em relação aos operandos criam dois formatos de expressão:

Prefixa		Pós-fixa
Requer que todos os operadores precedam os dois operandos sobre os quais atuam.	×	Requer que seus operadores venham depois dos operandos correspondentes.

Veja alguns exemplos na tabela:

Expressão infixa	Expressão prefixa	Expressão posfixa
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +

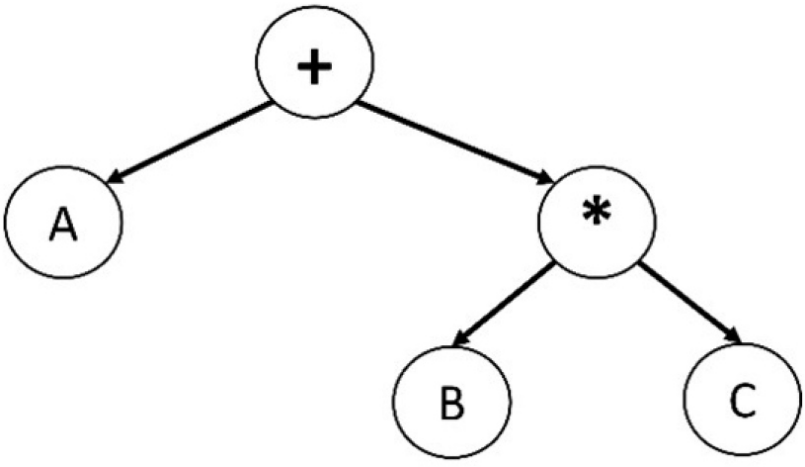
Tabela: Exemplos de expressões infixas, prefixas e pós-fixas.
Simone Ingrid Monteiro Gama

Algoritmos de percursos prefixos, infixos e pós-fixos

Seguem exatamente a mesma filosofia aplicada aos algoritmos de percursos em árvores de busca, ou seja:

- Se a árvore for lida em ordem raiz-esquerda-direita-raiz, teremos a expressão aritmética em notação prefixa.
- Se a árvore for lida em ordem esquerda-raiz-direita, teremos a expressão aritmética em notação infixa.
- Se a árvore for lida em ordem esquerda-direita-raiz, teremos a expressão aritmética em notação pós-fixa.

Para os exemplos da execução, considere a árvore de expressão aritmética:



12 - A expressão aritmética A + B * C modelada em árvore binária.

Algoritmo prefixo em Python

Pode ser implementado sob diversas técnicas de programação. A técnica que usaremos aqui é a recursividade, que apresenta um algoritmo mais elegante e conciso.

A partir da raiz *r* da árvore *T*, percorre-se a árvore de expressões da seguinte forma:

1

Visita-se a raiz.

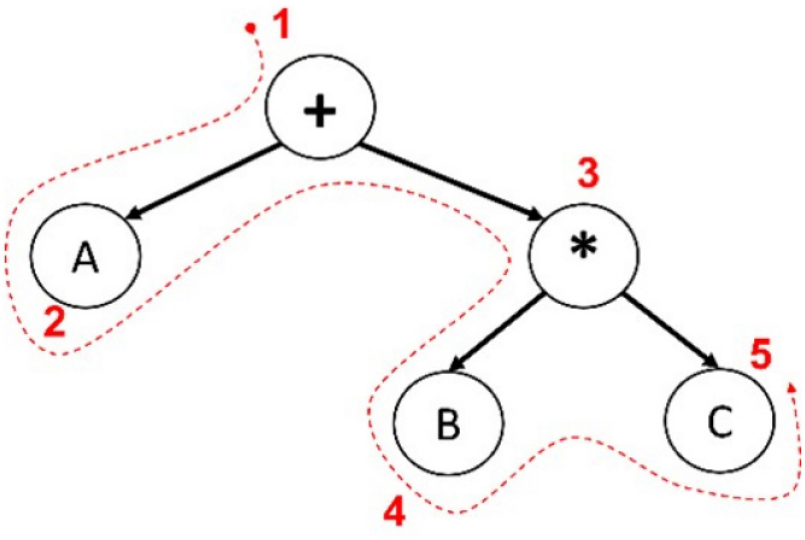
2

Percorre-se a subárvore esquerda de *T* em prefixo.

3

Percorre-se a subárvore direita de *T* em prefixo.

Considerando a árvore da expressão da imagem 12, o resultado da visita prefixa é apresentado pela numeração:



13 - Percurso prefixo da imagem 12. Resultado: **+A*BC**.

No algoritmo 11, considerando que as expressões aritméticas estão modeladas em uma árvore binária, a raiz contém o topo da árvore, esquerda e direita são os filhos de um nó. Confira a implementação, que é semelhante à aplicada na visita pré-ordem:

Python

Algoritmo 11 - Algoritmo Python de percurso prefixo da imagem 12.

Para realizar o percurso prefixo, são necessários três acessos ao nó:

Primeiro acesso

Executa-se a visita.

Segundo acesso

Chama-se recursivamente o algoritmo para a subárvore esquerda.

Terceiro acesso

Realiza-se a chamada do percurso prefixo do ramo direito.

Assim, a complexidade computacional do percurso prefixo é $O(n)$, em que n é a quantidade de nós na árvore de expressões.

Algoritmo infixo em Python

Pode ser implementado sob diversas técnicas de programação. A técnica que usaremos aqui também é a recursividade, que apresenta um algoritmo mais elegante e conciso.

A partir da raiz *r* da árvore *T*, percorre-se a árvore de expressões da seguinte forma:

1

Percorre-se a subárvore esquerda de *T* em ordem simétrica (infixo).

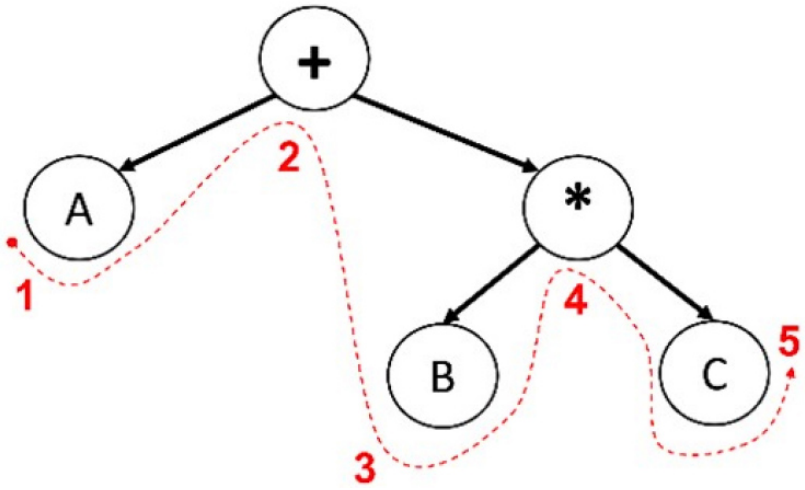
2

Visita-se a raiz.

3

Percorre-se a subárvore direita de *T* em ordem simétrica (infixo).

Considerando a árvore da expressão da imagem 12, o resultado da visita infixa é apresentado pela numeração da:



14 - Percurso *infixo* da imagem 12 . Resultado: **A+B*C**.

No algoritmo 12, considerando que as expressões aritméticas estão modeladas em uma árvore binária, a raiz contém o topo da árvore, esquerda e direita são os filhos de um nó. Segue-se a implementação, que é semelhante à aplicada na visita em ordem (simétrica).

Python

Algoritmo 12 - Algoritmo Python de percurso infixo da imagem 12.

A análise de complexidade é análoga à realizada no algoritmo do percurso prefixo. Observe que a única diferença é a ordem das visitas. Sendo assim, a complexidade computacional do algoritmo para percurso infixo é $O(n)$, em que n é a quantidade de nós na árvore de expressões.

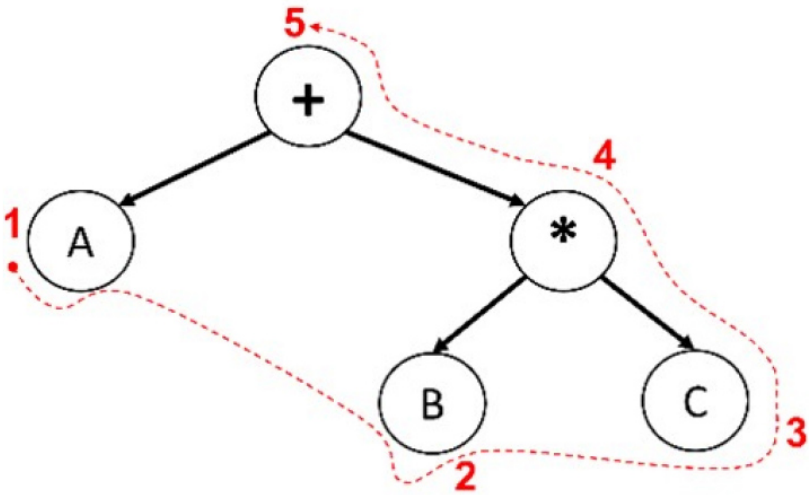
Algoritmo pós-fixo em Python

Também usaremos a técnica da recursividade, que apresenta um algoritmo mais elegante e conciso.

A partir da raiz *r* da árvore *T*, percorre-se a árvore de expressões da seguinte forma:

- Percorre-se a subárvore esquerda de *T* em pós-fixo.
- Percorre-se a subárvore direita de *T* em pós-fixo.
- Visita-se a raiz.

Considerando a árvore da expressão da imagem 12, o resultado da visita pós-fixa é apresentado pela seguinte numeração:



15 - Percurso pós-fixo da imagem 2. Resultado: **ABC*+**.

No algoritmo 13, considerando que as expressões aritméticas estão modeladas em uma árvore binária, a raiz contém o topo da árvore, esquerda e direita são os filhos de um nó. A implementação é semelhante à aplicada na visita em pós-ordem.

Python

Algoritmo 13 – Algoritmo Python de percurso posfixo da imagem 12.

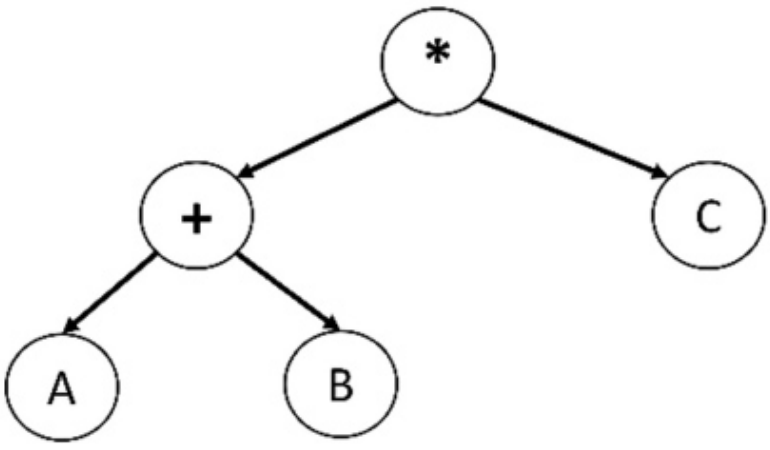
A análise da complexidade é totalmente análoga à análise feita para prefixo e infixo, o que faz com que o algoritmo tenha complexidade $O(n)$, em que n é a quantidade de nós na árvore de expressões.

Falta pouco para atingir seus objetivos.

Vamos praticar alguns conceitos?

Questão 1

Dada a árvore de expressões aritméticas a seguir, identifique, respectivamente, o percurso prefixo e pós-fixo:



A *+ABC / AB+C*

B ABC+* / A+BC*

C *ABC+ / AB*C+

D A+B*C / AB+C*

E AB+C* / A+B*C

Parabéns! A alternativa A está correta.

O percurso prefixo é definido por raiz-esquerda-direita, o resultado é *+ABC; o percurso pós-fixo é definido por esquerda-direita-raiz e o resultado do percurso é AB+C*.

Questão 2

Considere a seguinte expressão aritmética infixa A + B + C + D. A sua representação prefixa e pós-fixa, respectivamente, é:

A A B + C + D + e + + + A B C D

B + + + A B C D e A B + C + D +

C + + A B + C D e A +B + C + D +

D + A + + B C D e A D + C + B +

E AB+C+D+ e +++ABCD

Parabéns! A alternativa B está correta.

O percurso prefixo é definido por raiz-esquerda-direita, o resultado é + + + A B C D; o percurso pós-fixo é definido por esquerda-direita-raiz e o resultado do percurso é A B + C + D +.

Considerações finais

Como vimos ao longo do conteúdo, árvores binárias de busca compreendem requisitos fundamentais para modelagem de diversos problemas teóricos em computação. Além do mais, podem ser abstraídos e aplicados em diversos problemas do dia a dia, bem como em diversos outros tópicos interessantes, como expressões matemáticas.

Dessa maneira, foi possível compreender que a estrutura de árvores binárias é uma estrutura adicional para escolha do programador sobre qual é a melhor estrutura de dados a ser usada e, assim, modelar o processo de armazenar e gerenciar dados em memória.

Podcast

Ouça agora os principais assuntos abordados ao longo deste conteúdo.

Para ouvir o *áudio*, acesse a versão online deste conteúdo.

Explore +

Para aprofundar os conhecimentos adquiridos neste conteúdo:

Acesse o guia **Construindo Árvores de Expressão Aritméticas**, disponível no portal Aprenda Computação com Python.

Acesse, também, o guia do site GeeksforGeeks, intitulado **Árvore Binária Completa**, e conheça outros tipos de estruturas de árvores.

Referências

ADEL'SON-VEL'SKII, G. M.; LANDIS, E. M. **Algorithm for the Organization of Information**. Soviet Mathematics Doklady, 3, 1962.

CORMEN, T. H. *et al*. **Algoritmos**: teoria e prática. São Paulo: Campus, 2002.

NAVES, P. **Lagos andinos dão banho de beleza**. Folha de S. Paulo, São Paulo, 28 jun. 1999.

SZWARCFITER, J. L.; MARKENZON, L. **Estruturas de Dados e seus Algoritmos**. Rio de Janeiro: Livros Técnicos e Científicos, 1994.

Material para download

Clique no botão abaixo para fazer o download do conteúdo completo em formato PDF.

Download material

O que você achou do conteúdo?

☆ ☆ ☆ ☆ ☆

Relatar problema