Métodos de Pesquisa e Ordenação em Estruturas de Dados

UNIDADE 07

Estrutura de dados: Pesquisando coisas, parte I

Chegou o momento de nos aprofundarmos em um tópico bem interessante nos estudos sobre computação: os métodos de busca. Imagine um cenário em que você está em uma biblioteca vasta, em que cada livro é um pedaço de dado. Você quer encontrar o seu livro favorito lá. E agora, como encontraria exatamente o livro que quer em meio a milhares? Os métodos de busca funcionam como se fossem bibliotecários treinados dessa imensa biblioteca, e cada um tem a sua própria técnica e experiência para localizar rapidamente o que você precisa.

Outra forma de pensarmos os métodos de busca de dados é com a dança. Elas realmente se assemelham a diferentes estilos de dança: algumas são rápidas e ágeis como o tango, enquanto outras são metodicamente calculadas como uma valsa clássica. Aqui, a nossa ideia não é a de que você seja um especialista pronto para

dançar cada estilo do zero – afinal, alguns deles demandam anos de prática e aperfeiçoamento. O que eu quero é que você tenha uma noção sobre a beleza de cada dança, reconhecendo suas diferenças, nuances e, ocasionalmente, suas semelhanças.

IMPORTANTE

Em outras palavras, o nosso objetivo é o de que você entenda as diferenças e semelhanças entre os principais tipos de métodos de pesquisa, e não necessariamente que domine cada técnica do zero. Cursos como Ciência da Computação e Engenharia de Computação se concentram nisto: já aqui, nós nos concentraremos mais na aplicação.

Entender a essência de cada método, suas vantagens, desvantagens e situações de uso é fundamental. Afinal, em sua carreira, você não necessariamente terá de recriar cada técnica do zero – e acredite, algumas são muito complexas! Mas, ao compreender as características de cada uma, você será capaz de escolher a "dança" certa para cada "música" que o seu trabalho na programação tocar.

Vamos lá?

Procurando coisas em listas ou vetores

Antes de começarmos, eu sei que já testamos algumas formas básicas de pesquisa de dados enquanto estávamos testando estas estruturas de dados nas semanas anteriores. Por outro lado, existem também *outras* formas de procurarmos dados nestas estruturas, e este é o momento para isso.

Pesquisa linear (ou sequencial)

O jeito mais simples de pesquisarmos dados em uma lista ou vetor é pela *pesquisa linear* (também chamada de **pesquisa sequencial**). Tanto é que, ao ensinarmos estruturas de dados, geralmente começamos por esta técnica.

#ParaTodosVerem

Fonte: Foto de Becca McHaffie para Unsplash.

Vamos pensar em uma analogia? Imagine que você está procurando por uma roupa em uma loja, mas as roupas não estão organizadas nem por cor, nem por tipo, nem por tamanho: tudo está desorganizado. Neste caso, você **não tem como adivinhar** onde está a roupa que você quer, certo? Portanto, **precisará olhar de peça em peça de roupa até achar o que você quer**.

A **pesquisa linear** (ou pesquisa sequencial) segue exatamente esta mesma lógica: é como se você começasse olhando pela primeira peça de roupa mais à esquerda, depois a segunda, depois a terceira e assim sucessivamente até encontrar a peça que estava procurando. Se a roupa estiver na metade da prateleira, você teria de passar por todas as roupas anteriores até chegar nela. E, se a roupa não estiver no cabideiro, você teria de olhar todas as roupas até o fim para ter certeza disso.

Pesquisa binária

Agora, vamos pensar em **outro** jeito de encontrarmos as informações em uma lista ou vetor. Imagine que você está jogando um jogo de adivinhação, onde você tem que adivinhar um número entre 1 e 100. Você faz um palpite, e a única resposta que você recebe é "mais alto" ou "mais baixo". **A estratégia mais eficiente não é adivinhar números aleatoriamente; é começar no meio.** Se você adivinhar 50 e a resposta for "mais alto", você já eliminou metade dos números possíveis de uma só vez! Isso é o coração da **pesquisa binária**: a arte de dividir e conquistar.

Quando aplicamos a pesquisa binária a um vetor ou lista, estamos usando essa mesma lógica. Agora, se essa pesquisa aparenta ser mais inteligente do que a busca linear, por que não usamos busca binária para tudo? Ora, veja só: para que esta pesquisa funcione é obrigatório que a sua lista esteja **ordenada**, assim como os números no jogo de adivinhação. Em breve demonstraremos como isso funciona, beleza?

Tabelas *hash*

Este conceito é um pouco mais chato, mas espero que fique claro para você com o nosso exemplo. Vamos lá: imagine um cenário em que você está em um *fast-food* ou em uma cafeteria. Nesses lugares, você faz o seu pedido e é comum que anotem o seu nome. Ao ficar pronto, os atendentes lhe chamam pelo nome e você busca o seu pedido.

Isso pode funcionar bem em lugares menores, mas começa a se tornar um problema se for em uma praça de alimentação de *shopping* em que dezenas ou centenas de pessoas estão esperando pelo seu pedido, certo? Quer dizer: se o seu nome for razoavelmente comum, é muito provável que tenha mais de uma pessoa com o mesmo nome esperando pelo mesmo pedido, e isso pode tornar a coisa toda bem confusa e ineficiente.

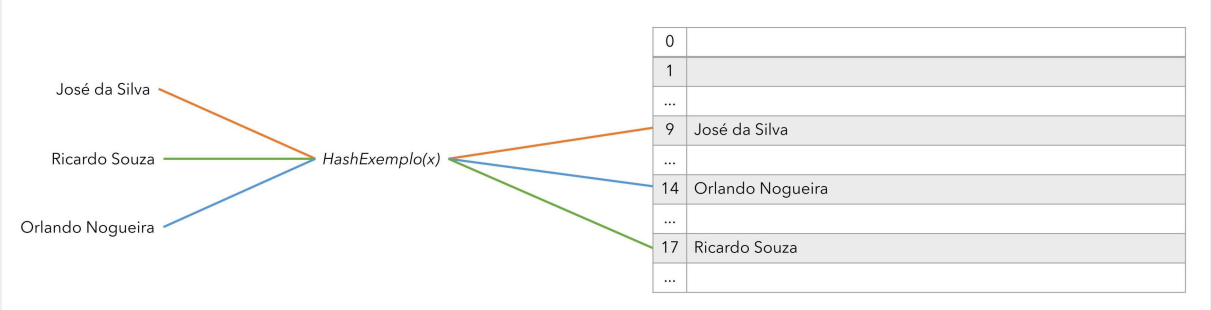
Para resolver isso, existem as senhas: é um sistema numérico onde cada pedido é convertido em um número específico. Você pode estar em qualquer lugar da praça de alimentação, mas o seu número será só seu, e sem confusões. Esta analogia é a essência de como funciona uma tabela *hash*: um sistema que **converte chaves (neste caso, pedidos) em índices de tabela por meio de uma função matemática**.

No universo das estruturas de dados funciona assim: quando armazenamos dados, usamos uma função *hash* (isto é: uma fórmula matemática) para transformar a chave em um índice numérico. Esse índice nos diz onde armazenar o valor no vetor que está por trás da tabela *hash*. Da mesma forma que funcionam as senhas da praça de alimentação, este índice nos permite encontrar rapidamente onde o valor está armazenado.

Veja um exemplo do funcionamento a seguir. Aqui, temos três *strings* com nomes diferentes. Estes nomes passam por uma função matemática (aqui chamada de "HashExemplo") que convertem estes nomes para códigos numéricos diferentes. À direita, temos uma tabela *hash*. O uso deste tipo de técnica pode ser muito bom porque:

1. É muito mais rápido encontrarmos um valor por estes códigos em vez do dado original.
2. Podemos usar este conceito em vários tipos de dados (*strings*, data e hora, número e outros tipos de objetos).
3. Podemos reaproveitar este conceito em áreas como criptografia e segurança da informação.

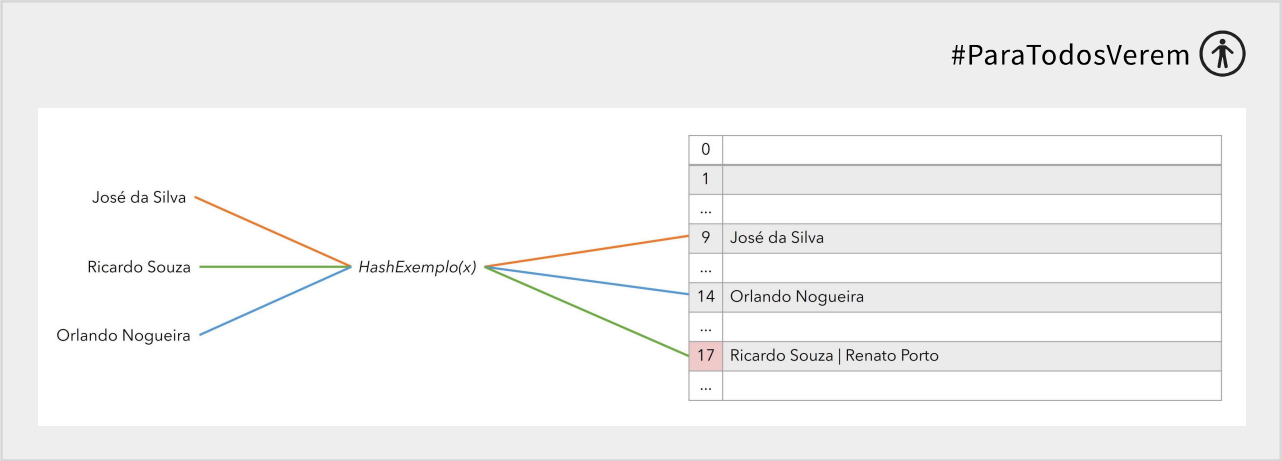
#ParaTodosVerem



Fonte: Os Autores (2024).

Por outro lado, surge um dilema. E se duas pessoas tiverem nomes que levam ao mesmo número? No mundo das tabelas *hash*, isso é chamado de **colisão**, e há várias maneiras de lidar com isso. Uma delas é como decidir que, se uma senha já tenha sido distribuída, a pessoa será direcionada para a próxima senha livre. Em termos técnicos,

isso é conhecido como **encadeamento** ou **sondagem linear**, dependendo de como o conflito é resolvido. Veja um exemplo de colisão a seguir: o novo nome (Renato Porto) também ocuparia o código 17. A solução para isso seria atribuí-lo para o próximo valor livre, que poderia ser 18 ou 19, por exemplo.



Fonte: Os Autores (2024).

Agora, como usamos isso para **pesquisar dados**? Vamos retornar à nossa analogia com o ato de buscar por um nome na praça de alimentação. Em vez de andar por cada mesa perguntando pelo nome da pessoa (ou sair gritando por aí), simplesmente convertemos o nome para o número com nossa função *hash* e vamos diretamente à mesa correspondente. Se a mesa estiver ocupada por uma pessoa diferente (devido a uma colisão), simplesmente verificamos a próxima mesa até encontrarmos a pessoa certa. **Isso faz com que a busca seja incrivelmente rápida**, mesmo em uma "praça de alimentação" com milhares ou mesmo milhões de "clientes".

DICA

Em Java, o HashMap e o HashTable já implementam este método “de fábrica”.

Prós e contras

Bom, o que acha de vermos melhor a comparação entre estas técnicas? Veja só:

Critério	Pesquisa linear	Pesquisa binária	Tabelas *hash*
Pré-condições.	Nenhuma.	*Array*/Vetor deve estar ordenado.	Nenhuma (mas requer função *hash*).

Melhor caso de tempo.	O(1).	O(1).	O(1) (em média).
Pior caso de tempo.	O(n).	O(log n).	O(n) (caso raro de muitas colisões).
Caso médio de tempo.	O(n/2) em média.	O(log n).	O(1) (em média).
Requerimentos de memória.	O(1) adicional.	O(1) adicional.	O(n) para a tabela, mais o armazenamento de possíveis colisões.
Operações de adição.	O(1) no fim do vetor/lista.	Não aplicável (a ordenação é necessária após inserção).	O(1) (em média).
Operações de remoção.	O(n) (precisa de deslocamento).	Não aplicável (a ordenação é necessária após remoção).	O(1) (em média).
Tipos de dados aplicáveis.	Qualquer tipo de dados.	Tipos que podem ser ordenados.	Tipos que podem ser “hasheados”.
Complexidade.	Simples e direta.	Moderada (depende da ordenação prévia).	Mais complexa (devido ao manuseio de colisões).
Uso recomendado.	**Dados pequenos ou operações não frequentes.**	**Grandes conjuntos de dados em que a ordenação é mantida.**	**Dados em que inserções e buscas rápidas são frequentes e podem ser únicos ou ter poucas colisões.**

Observações:

- **Melhor caso de tempo** se refere à complexidade de tempo quando o elemento está na primeira posição que verificamos.
- **Pior caso de tempo** se refere à complexidade de tempo quando o elemento está na última posição possível ou não está presente.
- **Caso médio de tempo** leva em conta todas as possibilidades de posições do elemento no conjunto de dados.
- As **operações de adição e remoção** para pesquisa binária não são aplicáveis de forma direta, pois após cada inserção ou remoção, você teria de reordenar o

vetor/lista para manter a propriedade de ordenação necessária para a pesquisa binária funcionar.
- **Requerimentos de memória adicional** se referem à memória extra necessária para realizar a busca além do armazenamento dos próprios dados.
- Os tipos que podem ser **"hasheados"** geralmente são números, *strings*, booleanos e tuplas.

Lembre-se: estas são generalizações e o desempenho real pode variar dependendo de fatores como implementação específica, tamanho do conjunto de dados, distribuição dos dados e arquitetura do sistema.

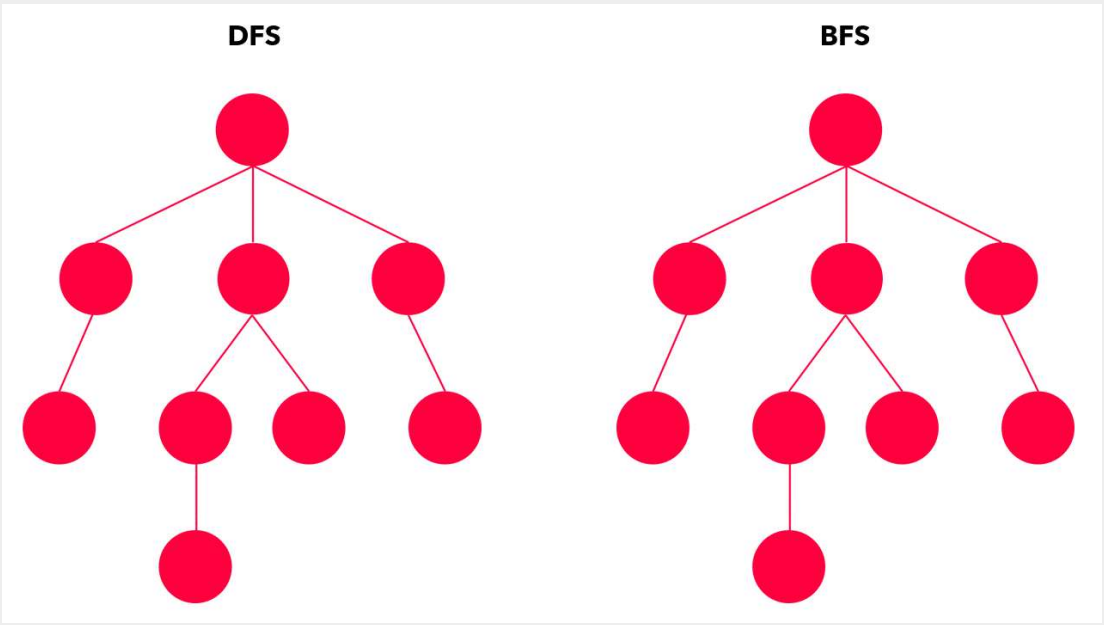
DICA

Muitos cursos se focam somente na pesquisa linear porque, afinal de contas, é mais rápido e fácil de entendermos como funciona. Por outro lado, em empresas é comum termos uma complexidade maior nos dados que requer estas outras técnicas. É por este motivo que entendemos que é importante que você também as conheça.

Procurando coisas em árvores binárias e grafos

E em árvores binárias, como funciona? Como pesquisamos por dados em estruturas assim? Acredito que seja importante você conhecer as duas principais maneiras de pesquisarmos dados em árvores: a **busca em largura** (*breadth-first search*, ou BFS) e a **busca em profundidade** (*depth-first search*, ou DFS). Já explicarei as diferenças entre ambos, mas acho que às vezes uma imagem vale mais do que mil palavras. Portanto, veja a imagem a seguir – perceba a ordem na qual os diferentes nós são acessados:

#ParaTodosVerem



Fonte: Os Autores (2024).

Percebeu as diferenças na ordem de acesso aos nós? Agora, vamos explorar melhor as diferenças entre estas estratégias?

Busca em profundidade (DFS)

A DFS é uma técnica fundamental de travessia de grafos e árvores, que explora tão longe quanto possível ao longo de cada ramo antes de retroceder. Essa abordagem usa uma estratégia de *backtracking* (isto é: saber onde o algoritmo já passou) que pode ser facilmente implementada com recursão ou com uma pilha.

Vamos usar a árvore binária como exemplo. Imagine uma árvore familiar, em que começamos com um ancestral e queremos listar todos os descendentes, começando pela geração mais próxima e movendo-nos para as gerações mais distantes. Neste caso, começamos pelo ancestral mais antigo que temos conhecimento (como uma bisavó). Depois, analisamos somente um dos filhos que esta bisavó teve (como a sua avó ou avô, e todos os seus irmãos). Depois, vamos para somente um dos filhos da avó ou avô (como o seu pai ou mãe, e todos os tios). Finalmente, analisamos o seu caso. Se não encontramos a informação que gostaríamos em você, analisaremos os seus irmãos. Se ainda assim não encontramos, refaremos toda esta análise com cada um dos tios e tias, bem como os seus primos e primas. Se ainda assim não encontrarmos, aí partiremos para os seus tios-avôs e seus respectivos filhos e netos.

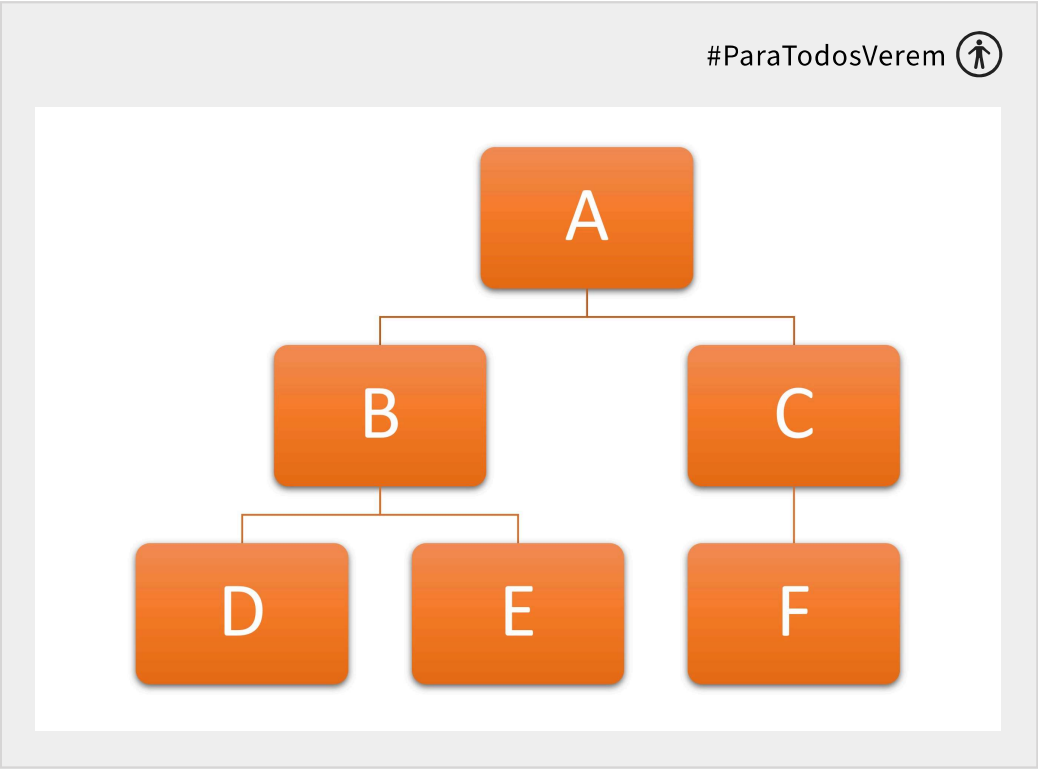
DICA

Outro jeito de imaginarmos o DFS é pensarmos que estamos em um labirinto que representa nossa árvore e queremos explorar o máximo possível em um caminho antes de voltar atrás e tentar outra rota.

Aqui está a lógica básica do algoritmo DFS para uma árvore binária:

1. **Inicie pelo nó raiz**: partimos do nó raiz.
2. **Visite o nó**: processamos o nó (por exemplo, imprimindo seu valor).
3. **Travessia à esquerda**: recursivamente, aplique o DFS ao filho à esquerda do nó atual.
4. **Travessia à direita**: após retornar da travessia à esquerda, recursivamente aplique o DFS ao filho à direita do nó atual.
5. ***Backtracking***: continue este processo até que todos os nós sejam visitados. Quando chegar a um nó folha ou um nó cujos filhos já foram visitados, retorne ao nó pai (isto é, *backtracking*).

Essencialmente, o DFS vai tão profundo quanto pode em um ramo, e então volta atrás (*backtracking*) para tentar outros ramos. Vejamos um exemplo de uma árvore binária:



Fonte: Os Autores (2024).

Aqui está como o DFS funcionaria nesta árvore, se escolhermos explorar o filho à esquerda antes:

1. **Iniciamos com A**.
2. **Visitamos A e, depois, vamos para B**.
3. **Visitamos B e, depois, vamos para D**.
4. **Visitamos D e voltamos para B** (já que D é uma folha).
5. **De B, vamos para E**.
6. **Visitamos E e, depois, voltamos para B**, e então para A (já que todos os filhos de B foram visitados).
7. **De A, vamos para C**.
8. **Visitamos C e, depois, vamos para F**.
9. **Visitamos F e, depois, vamos para C**, e então para A (já que todos os filhos de C foram visitados).

A ordem de visitação seria: A, B, D, E, C, F.

Busca em largura (BFS)

A BFS é outra técnica de travessia de árvore ou grafo que começa na raiz (ou algum nó arbitrário) e explora **todos os vizinhos desse nó**. Em seguida, para cada vizinho, explora seus vizinhos não visitados e assim por diante, até que todos os nós sejam visitados. A ideia principal por trás do BFS é que ele visita os nós em "ondas".

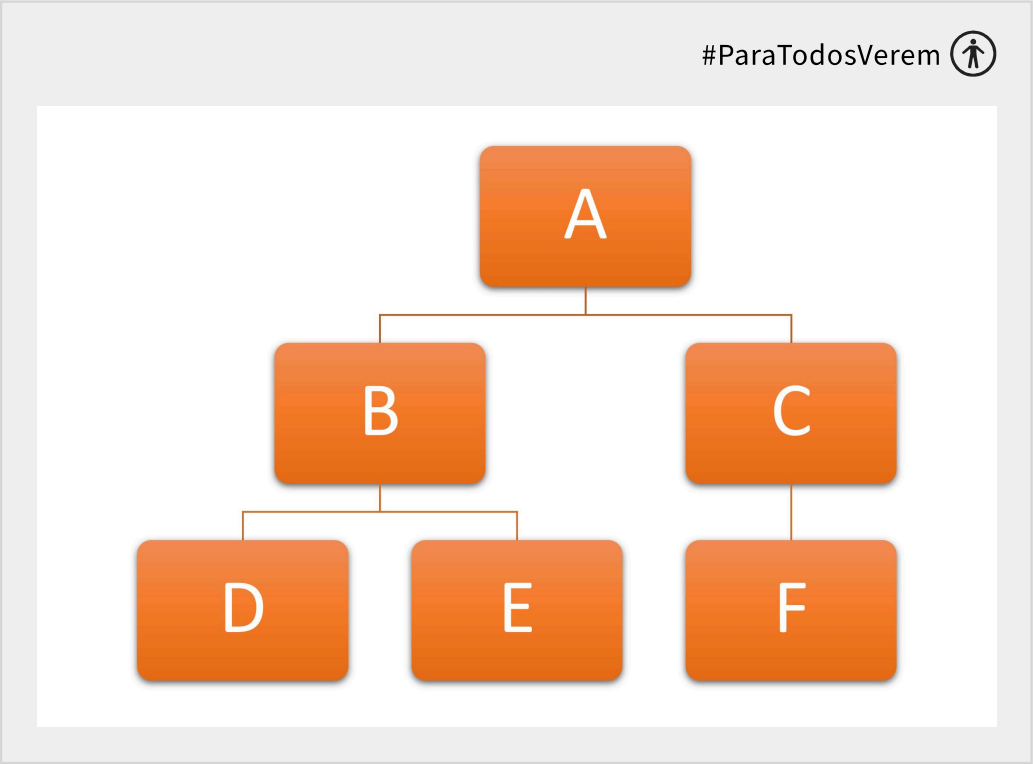
Vamos usar uma árvore binária como exemplo. Imagine uma árvore familiar, onde começamos com um ancestral e queremos listar todos os descendentes, começando pela geração mais próxima e movendo-nos para as gerações mais distantes, de forma parecida com o que comentamos no caso do DFS. Neste caso, começamos pelo ancestral mais antigo que temos conhecimento (como uma bisavó). Depois, analisamos todos os filhos que esta bisavó teve (como a sua avó ou avô, e todos os seus irmãos). Depois, vamos para os filhos da avó ou avô (como o seu pai ou mãe, e todos os tios). Finalmente, chegamos em você e todos os seus irmãos e primos.

Aqui está a lógica básica do algoritmo BFS para uma árvore binária:

1. **Inicie pela raiz**: coloque o nó raiz em uma fila.
2. **Nó atual**: enquanto a fila não estiver vazia, remova o nó da frente da fila. Este é o nosso nó "atual".
3. **Visite o nó**: processamos o nó atual (por exemplo, imprimindo seu valor).
4. **Enfileire filhos**: adicione os filhos do nó atual (se houver) ao final da fila.
5. **Repita o processo**: continue o processo até que a fila esteja vazia.

A cada passo, estamos visitando os nós de uma "camada" da árvore antes de passar para a próxima. A fila nos ajuda a manter a ordem de visitação correta. Vamos para um outro exemplo?

Vamos reutilizar a mesma árvore binária no exemplo do BFS:



Fonte: Os Autores (2024).

Vejamos como o BFS funcionaria nesta árvore:

1. **Iniciar**: Colocamos a raiz (A) na fila.

Fila: A

2. **Visitar A e enfileirar B e C**: Removemos A da fila e enfileiramos B e C.

Fila: B, C

3. **Visitar B e enfileirar D e E**: Removemos B da fila e enfileiramos D e E.

Fila: C, D, E

4. **Visitar C e enfileirar F**: Removemos C da fila e enfileiramos F.

Fila: D, E, F

5. **Visitar D**: Removemos D da fila.

Fila: E, F

6. **Visitar E**: Removemos E da fila.

Fila: F

7. **Visitar F**: Removemos F da fila. A fila está agora vazia.

Fila: (vazia)

A ordem de visitação seria: A, B, C, D, E, F.

Prós e contras

O que acha de entendermos melhor as diferenças entre estas duas técnicas? Vejamos a tabela a seguir:

Critério	**Busca em profundidade (DFS)**	**Busca em largura (BFS)**
Ordem de visitação.	Depende da variação (pré-ordem: raiz > esquerda > direita / em ordem: esquerda > raiz > direita / pós-ordem: esquerda > direita > raiz).	Por níveis da árvore, de cima para baixo e da esquerda para a direita.
Utilidade.	Pesquisa de rotas, verificar a existência de uma rota, encontrar componentes conectados, ordenação topológica.	Encontrar o caminho mais curto em árvores sem ponderação, resolver problemas de minimização-maximização em jogos, algoritmos de roteamento.
Complexidade de tempo.	O(V + E) para grafos, O(n) para árvores, onde V é o número de vértices e E é o número de arestas.	O(V + E) para grafos, O(n) para árvores.
Complexidade de espaço.	O(V) no pior caso para grafos devido à pilha de recursão, O(h) para árvores onde h é a altura da árvore.	O(V) no pior caso para grafos, O(n) no pior caso para árvores quando todos os nós estão no mesmo nível ou muito desbalanceados.
Comportamento.	Recursivo (normalmente) ou iterativo com uma pilha.	Iterativo (normalmente implementado com uma fila).
Aplica-se bem a:	Problemas que requerem **exploração de todas as possibilidades** ou para encontrar soluções que exigem o exame de todos os nós.	Grafos e árvores onde o objetivo é encontrar algo o **mais rápido possível sem considerar a profundidade**.

Garantia de encontrar solução.	Pode encontrar uma solução mais profunda antes de uma mais superficial, não garante o caminho mais curto.	Encontra o caminho mais curto ou solução de nível mínimo (se houver) antes de outros.
Uso na prática.	Cálculo de expressões, ordenação topológica em sistemas de compilação de código, resolução de labirintos, *puzzles* etc.	Algoritmos de roteamento como o OSPF em redes, problemas de IA em jogos, análise de rede social, níveis de decisão, mineração de dados.

Notas:

- **Ordem de visitação:** se refere à sequência na qual os nós são visitados durante a travessia.
- **Utilidade:** o que cada travessia é comumente usada para realizar.
- **Complexidade de tempo:** para todas as travessias, visitamos cada nó uma vez, portanto a complexidade de tempo é O(n), em que n é o número de nós.
- **Complexidade de espaço:** depende da altura da árvore (h) para travessias que podem ser implementadas de forma recursiva, pois essa é a profundidade máxima da pilha de chamadas de função. Para a travessia por níveis, precisamos de espaço para manter uma fila que pode crescer até O(n) no pior caso (quando todos os nós estão no mesmo nível).
- **Tipos de árvore:** todos esses métodos podem ser aplicados a árvores binárias, mas algumas, como a travessia em ordem, são particularmente úteis para árvores binárias de busca (BSTs), em que a ordem dos elementos é significativa.
- **Aplicação prática:** exemplos de onde essas travessias podem ser aplicadas em algoritmos reais e problemas de estrutura de dados.

Conclusão

Vimos durante esta semana um pouco sobre a riqueza e diversidade das estratégias de busca e armazenamento de dados na ciência da computação. A pesquisa linear e a pesquisa binária, com as suas abordagens diretas (mas contrastantes), oferecem uma compreensão básica sobre a eficiência e a otimização na busca de dados. Enquanto a pesquisa linear exemplifica simplicidade e versatilidade, percorrendo sequencialmente por meio dos elementos, a pesquisa binária introduz a eficácia da divisão e conquista em conjuntos de dados ordenados, reduzindo significativamente o tempo de busca. Já as tabelas *hash*, por outro lado, destacam a importância de algoritmos eficientes de armazenamento e recuperação de dados, utilizando funções de *hash* para mapeamento rápido de chaves para valores.

Por outro lado, os algoritmos DFS e BFS nos aproximam do mundo dos grafos, revelando métodos para explorar estruturas de dados mais complexas de forma sistemática e eficiente. O DFS, mergulhando profundamente em cada caminho antes de retroceder, e o BFS, explorando todos os vizinhos de um nó antes de avançar, mostram abordagens diferentes e complementares para a navegação em grafos. Estes algoritmos não só são fundamentais para a compreensão de estruturas de dados mais complexas, mas também são aplicáveis em uma ampla gama de problemas práticos, desde a resolução de labirintos até a análise de redes sociais.

Por meio do estudo desses métodos, você ganha não apenas habilidades técnicas, mas também uma perspectiva mais ampla sobre como abordar problemas computacionais, preparando você para enfrentar desafios tanto no campo acadêmico quanto no profissional com uma base sólida e um pensamento analítico aguçado. Dito isso, ainda não acabou: teremos mais para explorar em breve!

Referências Bibliográficas

TENENBAUM, A. M. **Estrutura de dados usando C**. São Paulo: Makron Books, 1995.